Citrus

Authors: The Citrus Community

Version 4.7.0, 2025-06-26

citrus

1. Preface . . . 2
2. Introduction . . . 3
   2.1. Overview . . . 3
   2.2. Usage scenarios . . . 3
3. Setup . . . 5
   3.1. Using Maven . . . 5
   3.2. Using Gradle . . . 8
4. Runtimes . . . 10
   4.1. TestNG . . . 10
   4.2. JUnit5 . . . 15
   4.3. JUnit4 . . . 19
   4.4. QuarkusTest . . . 21
   4.5. Cucumber . . . 30
   4.6. Main CLI . . . 40
   4.7. JBang . . . 41
   4.8. Sharding Test Cases . . . 48
5. Run tests . . . 50
   5.1. Run Java tests . . . 50
   5.2. Test source loader . . . 64
   5.3. XML . . . 67
   5.4. YAML . . . 75
   5.5. Groovy . . . 83
   5.6. Spring Bean XML . . . 95
6. Test variables . . . 106
   6.1. Global variables . . . 106
   6.2. Extract variables . . . 108
   6.3. Path expressions . . . 110
   6.4. Escape variables . . . 113
7. Message validation . . . 114
   7.1. Validation registry . . . 114
   7.2. Validation modules . . . 115
   7.3. Json validation . . . 116
   7.4. XML validation . . . 130
   7.5. Schema validation . . . 160
   7.6. Plain text validation . . . 165
   7.7. Binary validation . . . 171
   7.8. Hamcrest validation . . . 174
   7.9. Custom validation . . . 175

8. Test actions . . . 179
8.1. Send . . . 179
8.2. Receive . . . 196
8.3. SQL . . . 225
8.4. Sleep . . . 239
8.5. Delay . . . 240
8.6. Java . . . 242
8.7. Expect timeout . . . 245
8.8. Echo . . . 248
8.9. Print . . . 249
8.10. Stop time . . . 251
8.11. Create variables . . . 256
8.12. Create endpoints . . . 258
8.13. Trace variables . . . 263
8.14. Transform . . . 265
8.15. Groovy script execution . . . 271
8.16. Failing the test . . . 279
8.17. Input . . . 280
8.18. Load Properties . . . 283
8.19. Purging JMS destinations . . . 284
8.20. Purging Spring Message Channels . . . 289
8.21. Purging endpoints . . . 294
8.22. Assert failure . . . 299
8.23. Catch exceptions . . . 300
8.24. Apache Ant build . . . 302
8.25. Start/Stop server . . . 307
8.26. Stop Timer . . . 312
8.27. Custom action . . . 315
9. Containers . . . 319
9.1. Sequential . . . 319
9.2. Conditional . . . 320
9.3. Parallel . . . 324
9.4. Iterate . . . 330
9.5. Repeat . . . 333
9.6. Repeat on error . . . 336
9.7. Timer . . . 341
9.8. Async . . . 346
9.9. Wait . . . 352
9.10. Custom containers . . . 358
10. Endpoints . . . 362
10.1. Send messages . . . 363

10.2. Receive messages . . . 365
10.3. Local message store . . . 367
11. Direct endpoint . . . 369
11.1. Channel endpoint . . . 369
11.2. Synchronous direct endpoints . . . 371
11.3. Message selectors . . . 374
11.4. Payload matching selector . . . 375
11.5. Root QName selector . . . 376
11.6. Xpath selector . . . 378
11.7. JsonPath selector . . . 379
12. JMS support . . . 381
12.1. JMS endpoints . . . 381
12.2. JMS synchronous endpoints . . . 385
12.3. JMS topics . . . 389
12.4. JMS topic durable subscription . . . 391
12.5. JMS topic purging . . . 394
12.6. JMS message headers . . . 394
12.7. Dynamic destination names . . . 395
12.8. SOAP over JMS . . . 396
13. Apache Kafka Support . . . 397
13.1. Kafka Endpoint . . . 397
13.2. Kafka Synchronous Endpoints . . . 403
13.3. Kafka Message Headers . . . 403
13.4. Kafka Message . . . 405
13.5. Kafka Message Selector . . . 406
13.6. Dynamic Kafka Endpoints . . . 409
13.7. Embedded Kafka Server . . . 410
14. Http REST support . . . 412
14.1. Http REST client . . . 413
14.2. Http client interceptors . . . 421
14.3. Http REST server . . . 422
14.4. Http headers . . . 426
14.5. Http query parameter . . . 430
14.6. Http server interceptors . . . 432
14.7. Http form urlencoded data . . . 433
14.8. Http error handling . . . 436
14.9. Http client basic authentication . . . 438
14.10. Http server basic authentication . . . 441
14.11. Http client SSL . . . 443
14.12. Http server SSL . . . 445
14.13. Http cookies . . . 446

14.14. Http Gzip compression . . . 451
14.15. Http servlet filters . . . 453
14.16. Http servlet context customization . . . 455
15. SOAP WebServices . . . 459
15.1. SOAP client . . . 459
15.2. SOAP client interceptors . . . 461
15.3. SOAP server . . . 462
15.4. SOAP send and receive . . . 464
15.5. SOAP headers . . . 466
15.6. SOAP HTTP mime headers . . . 468
15.7. SOAP Envelope handling . . . 469
15.8. SOAP server interceptors . . . 470
15.9. SOAP 1.2 . . . 471
15.10. SOAP faults . . . 472
15.11. Send SOAP faults . . . 472
15.12. Receive SOAP faults . . . 474
15.13. Multiple SOAP fault details . . . 480
15.14. Send HTTP error codes with SOAP . . . 483
15.15. SOAP attachment support . . . 484
15.16. Send SOAP attachments . . . 484
15.17. Receive SOAP attachments . . . 485
15.18. SOAP MTOM support . . . 486
15.19. SOAP client basic authentication . . . 489
15.20. SOAP server basic authentication . . . 491
15.21. WS-Addressing support . . . 492
15.22. SOAP client fork mode . . . 494
15.23. SOAP servlet context customization . . . 495
16. Apache Camel support . . . 499
16.1. Camel endpoint . . . 500
16.2. Synchronous Camel endpoint . . . 503
16.3. Camel exchange headers . . . 505
16.4. Camel exception handling . . . 506
16.5. Camel context handling . . . 509
16.6. Camel route actions . . . 511
16.7. Camel controlbus actions . . . 514
16.8. Camel infrastructure services . . . 517
16.9. Camel endpoint DSL . . . 522
16.10. Camel processor support . . . 523
16.11. Camel data format support . . . 526
17. Message channel support . . . 528
17.1. Channel endpoint . . . 528

17.2. Synchronous channel endpoints . . . 530
17.3. Message selectors . . . 533
17.4. Payload matching selector . . . 534
17.5. Root QName selector . . . 535
17.6. Xpath selector . . . 537
17.7. JsonPath selector . . . 539
18. WebSocket support . . . 540
18.1. WebSocket client . . . 540
18.2. WebSocket server endpoints . . . 542
18.3. WebSocket headers . . . 543
19. Mail support . . . 546
19.1. Mail client . . . 547
19.2. Mail server . . . 550
20. FTP support . . . 554
20.1. FTP client . . . 554
20.2. FTP server . . . 570
21. SFTP/SCP support . . . 586
21.1. SFTP client . . . 586
21.2. SFTP server . . . 594
21.3. SCP client . . . 599
22. File support . . . 604
22.1. Write files . . . 604
22.2. Read files . . . 605
23. Selenium support . . . 607
23.1. Selenium browser . . . 607
23.2. Selenium actions . . . 608
23.3. Start/stop browser . . . 612
23.4. Find . . . 612
23.5. Click . . . 613
23.6. Hover . . . 614
23.7. Form input actions . . . 614
23.8. Fill form . . . 615
23.9. Page actions . . . 616
23.10. Page validation . . . 617
23.11. Wait . . . 619
23.12. Navigate . . . 620
23.13. Window actions . . . 620
23.14. Alert . . . 621
23.15. Make screenshot . . . 621
23.16. Clear browser cache . . . 622
24. Vert.x event bus support . . . 624

24.1. Vert.x endpoint . . . 624
24.2. Synchronous Vert.x endpoint . . . 626
24.3. Vert.x instance factory . . . 627
25. JDBC support . . . 629
25.1. The Citrus-JDBC-Driver . . . 629
25.2. The Citrus-JDBC-Server . . . 630
25.3. JdbcMessage . . . 636
26. Docker support . . . 641
26.1. Docker client . . . 641
26.2. Docker commands . . . 642
27. SSH support . . . 647
27.1. SSH Client . . . 648
27.2. SSH Server . . . 650
28. RMI support . . . 653
28.1. RMI client . . . 654
28.2. RMI server . . . 661
29. JMX support . . . 668
29.1. JMX client . . . 669
29.2. JMX server . . . 677
30. Zookeeper support . . . 687
30.1. Zookeeper client . . . 687
30.2. Zookeeper commands . . . 688
31. Spring Restdocs support . . . 693
31.1. Spring Restdocs using Http . . . 693
31.2. Spring Restdocs using SOAP . . . 696
31.3. Spring Restdocs in Java DSL . . . 697
32. Dynamic endpoint components . . . 700
33. Endpoint adapter . . . 706
33.1. Empty response endpoint adapter . . . 706
33.2. Static response endpoint adapter . . . 706
33.3. Request dispatching endpoint adapter . . . 708
33.4. Channel endpoint adapter . . . 710
33.5. JMS endpoint adapter . . . 710
34. Connectors . . . 712
35. OpenAPI support . . . 713
35.1. OpenAPI specification . . . 713
35.2. OpenAPI client . . . 713
35.3. OpenAPI server . . . 718
36. JBang support . . . 721
36.1. JBang action . . . 721
37. Kubernetes support . . . 723

37.1. Kubernetes client . . . 723
37.2. Kubernetes actions . . . 725
37.3. Create Labels . . . 725
37.4. Create Annotations . . . 726
37.5. Create Service . . . 728
37.6. Connect with Service . . . 729
37.7. Disconnect from Service . . . 733
37.8. Create ConfigMap . . . 734
37.9. Create Secret . . . 736
37.10. Create Resources . . . 738
37.11. Delete Resources . . . 739
37.12. Create Custom Resource . . . 741
37.13. Delete Custom Resource . . . 743
37.14. Verify Custom Resource . . . 744
37.15. Verify Pod Status . . . 746
37.16. Verify Pod Logs . . . 747
37.17. Kubernetes Citrus agent . . . 748
37.18. Spring bean config . . . 752
37.19. Kubernetes commands . . . 755
37.20. Kubernetes command bus . . . 766
38. Knative support . . . 770
38.1. Knative client . . . 770
38.2. Knative broker . . . 771
38.3. Verify broker status . . . 772
38.4. Send events . . . 773
38.5. Receive events . . . 776
38.6. Knative channels . . . 782
38.7. Knative subscription . . . 782
39. Testcontainers support . . . 783
39.1. Start and stop Testcontainers . . . 783
39.2. PostgreSQL . . . 787
39.3. MongoDB . . . 789
39.4. LocalStack . . . 790
39.5. Kafka . . . 795
39.6. Redpanda . . . 797
39.7. Docker compose . . . 798
40. Functions . . . 801
40.1. concat() . . . 801
40.2. substring() . . . 802
40.3. stringLength() . . . 803
40.4. translate() . . . 803

40.5. substringBefore() . . . 804
40.6. substringAfter() . . . 804
40.7. round() . . . 804
40.8. floor() . . . 805
40.9. ceiling() . . . 805
40.10. randomNumber() . . . 805
40.11. randomNumberGenerator() . . . 806
40.12. randomString() . . . 807
40.13. randomPattern() . . . 808
40.14. randomEnumValue() . . . 809
40.15. currentDate() . . . 810
40.16. upperCase() . . . 810
40.17. lowerCase() . . . 810
40.18. average() . . . 811
40.19. minimum() . . . 811
40.20. maximum() . . . 811
40.21. sum() . . . 811
40.22. absolute() . . . 812
40.23. mapValue() . . . 812
40.24. randomUUID() . . . 812
40.25. encodeBase64() . . . 813
40.26. decodeBase64() . . . 813
40.27. escapeXml() . . . 813
40.28. escapeJson() . . . 814
40.29. cdataSection() . . . 814
40.30. digestAuthHeader() . . . 814
40.31. localHostAddress() . . . 815
40.32. changeDate() . . . 815
40.33. readFile() . . . 816
40.34. message() . . . 817
40.35. xpath() . . . 818
40.36. jsonPath() . . . 818
40.37. urlEncode()/urlDecode() . . . 819
40.38. systemProperty() . . . 820
40.39. env() . . . 820
40.40. unixTimestamp() . . . 820
41. Validation Matchers . . . 821
41.1. ignore() . . . 822
41.2. matchesXml() . . . 823
41.3. equalsIgnoreCase() . . . 824
41.4. contains() . . . 824

41.5. startsWith() . . . 824
41.6. endsWith() . . . 824
41.7. matches() . . . 824
41.8. matchesDatePattern() . . . 825
41.9. isNumber() . . . 825
41.10. lowerThan() . . . 825
41.11. greaterThan() . . . 825
41.12. isWeekday() . . . 825
41.13. variable() . . . 826
41.14. dateRange() . . . 826
41.15. assertThat() . . . 827
41.16. ignoreNewLine() . . . 828
41.17. trim() . . . 829
41.18. trimAllWhitespaces() . . . 829
41.19. isUUIDv4() . . . 829
42. Citrus agent . . . 831
42.1. Agent REST service API . . . 831
42.2. Configuration . . . 833
42.3. Web UI . . . 835
42.4. Docker container image . . . 835
42.5. Agent JBang commands . . . 839
42.6. Maven plugin . . . 839
43. Data dictionaries . . . 841
43.1. XML data dictionaries . . . 841
43.2. JSON data dictionaries . . . 843
43.3. Dictionary scopes . . . 844
43.4. Path mapping strategies . . . 846
44. Test actors . . . 848
44.1. Create test actors . . . 848
44.2. Reference test actors . . . 850
44.3. Disable test actors . . . 852
44.4. Test actor environment settings . . . 854
45. Test suite actions . . . 855
45.1. Before suite . . . 855
45.2. After suite . . . 858
45.3. Before test . . . 861
45.4. After test . . . 864
46. Customize meta information . . . 867
47. Tracing incoming/outgoing messages . . . 869
48. Reporting and Test Results . . . 871
48.1. Console logging . . . 871

48.2. JUnit reports . . . 872
48.3. HTML reports . . . 873
49. Configuration options . . . 874
49.1. Environment settings . . . 874
49.2. Spring configuration settings . . . 875
49.3. Property file settings . . . 876
50. Spring support . . . 877
50.1. Spring XML application context . . . 877
50.2. Spring Java config . . . 878
51. Samples . . . 880
51.1. The FlightBooking sample . . . 880
52. Appendix . . . 890
Maven archetype . . . 890

Version: 4.7.0

Chapter 1. Preface

Integration tests are a critical part of software testing. In contrast to unit tests where the primary goal is to verify a single class or method in isolation to other parts of the software the integration tests look at a wider scope with several components and software parts interacting with each other.

Integration tests often rely on infrastructure components such as I/O, databases, 3rd party services and so on. In combination with messaging and multiple message transports as client and/or server the automated tests become a tough challenge. Testers need sufficient tool support to master this challenge for automated integration test. Citrus as an Open Source framework is here to help you master this challenge.

In a typical enterprise application and middleware scenario automated integration testing of message-based interfaces is exhausting and sometimes barely possible. Usually the tester is forced to simulate several interface partners in an end-to-end integration test.

The first thing that comes to one's mind is manual testing. No doubt manual testing is fast. In a long term perspective manual testing is time-consuming and causes severe problems regarding maintainability as they are error prone and not repeatable.

Citrus provides a complete test automation tool for integration testing of message-based enterprise applications. You can test your message interfaces (Http REST, SOAP, JMS, Kafka, TCP/IP, FTP, ...) to other applications as client and server. Every time a code change is made all automated Citrus tests ensure the stability of software interfaces and its message communication.

Regression testing and continuous integration is very easy as Citrus fits into your build lifecycle (Maven or Gradle) as usual Java unit test (JUnit, TestNG, Cucumber).

With powerful validation capabilities for various message formats like XML, CSV or JSON Citrus is ready to provide fully automated integration tests for end-to-end use cases. Citrus effectively composes complex messaging use cases with response generation, error simulation, database interaction and more.

This documentation provides a reference guide to all features of the Citrus test framework. It gives a detailed picture of effective integration testing with automated integration test environments. Since this document is open, please do not hesitate to give feedback in form of comments, change requests, fixes and pull requests. We are more than happy to continuously improve this documentation with your help!

Chapter 2. Introduction

Citrus provides automated integration tests for message-based enterprise applications. The framework is Open Source and supports various messaging transports (Http REST, SOAP, JMS, Kafka, TCP/IP, FTP, …) and data formats (XML, Json, plaintext, binary).

The Citrus tests use well-known unit test frameworks (JUnit, TestNG, Cucumber) for execution and integrates with build tools like Maven or Gradle. In addition, Citrus leverages standard libraries like Quarkus, the Spring Framework or Apache Camel.

It is also good to know that Citrus is written purely in Java. Thus, it is also fully compatible with Kotlin.

2.1. Overview

Citrus supports simulating interface partners across different messaging transports. You can easily produce and consume messages with a wide range of protocols like HTTP, JMS, TCP/IP, FTP, SMTP and more. The framework is able to act both as a client and server. In each communication step Citrus is able to validate message contents towards syntax and semantics.

In addition to that the Citrus framework offers a wide range of test actions to take control of the process flow during a test (e.g. iterations, system availability checks, database connectivity, parallelism, delays, error simulation, scripting and many more).

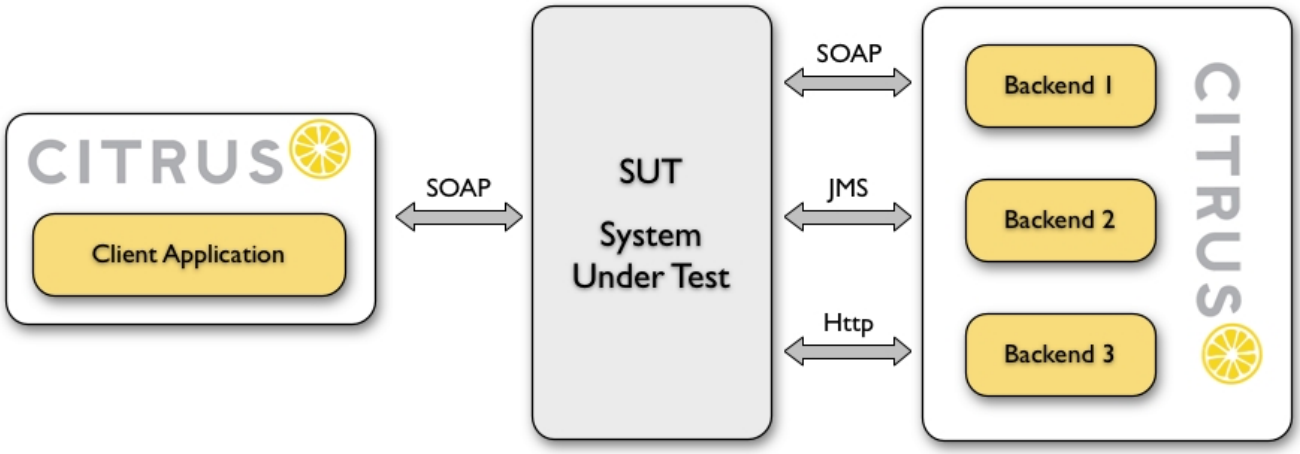
The test is able to describe a whole use case scenario including several interface partners that exchange many messages with each other. The composition of complex message flows in a single test case with several test steps is one of the major features in Citrus.

You can choose how to describe the test case definition either with pure XML or a Java domain specific language. The tests can be executed multiple times as automated integration test.

With JUnit and TestNG integration Citrus can easily be integrated into your build lifecycle process (Maven or Gradle). During a test Citrus simulates all surrounding interface partners (client or server) without any coding effort. With easy definition of expected message content (header and body) for XML, CSV, SOAP, JSON or plaintext messages Citrus is able to validate the incoming data towards syntax and semantics.

2.2. Usage scenarios

Citrus should help you whenever it comes to verify a message-based software with its interfaces to other components and partners using automated integration tests. Every project that interacts with other components over messaging transports needs to simulate these interface partners on the client or server side in a test scenario. Citrus is here to help you master these test automation tasks.



This test set up is typical for a Citrus use case. In such a test scenario we have a system under test (SUT) with several messaging interfaces to other applications. A client application invokes services on the SUT and triggers business logic. The SUT is linked to several backend applications over various messaging transports (here SOAP, JMS, and Http). As part of the business logic one or more of these backend services is called and interim message notifications and responses are sent back to the client application.

This generates a bunch of messages that are exchanged throughout the components involved.

In the automated integration test Citrus needs to send and receive those messages over different transports. Citrus takes care of all interface partners (ClientApplication, Backend1, Backend2, Backend3) and simulates their behavior by sending proper response messages in order to keep the message flow alive.

Each communication step comes with message validation and comparison against an expected message template (e.g. XML or JSON data). In addition to messaging steps Citrus is also able to perform arbitrary other test actions (e.g. perform a database query between requests).

In fact a Citrus test case is nothing but a normal JUnit or TestNG test case. This makes it very straight forward to run the tests from your favorite Java IDE (Eclipse, IntelliJ, VSCode, ...) and as part of your software build process (Maven or Gradle). The Citrus tests become repeatable and give you fully automated integration tests to ensure software quality and interface stability.

The following reference guide walks through all Citrus capabilities and shows how to have a great integration test experience.

Chapter 3. Setup

This chapter discusses how to get started with Citrus. It deals with the installation and set up of the framework, so you are ready to start writing test cases after reading this chapter.

Usually you add Citrus as a test-scoped dependency library in your project. Build tools like Maven or Gradle provide standard integration for test libraries such as Citrus. As Citrus tests are nothing but normal unit tests (JUnit, TestNG, Cucumber) you can run the tests with the standard unit test build integration (e.g. via maven-failsafe plugin).

This chapter describes the Citrus project setup possibilities, choose one of them that fits best to include Citrus into your project.

3.1. Using Maven

Citrus uses Maven internally as a project build tool and provides extended support for Maven projects. Maven will ease up your life as it manages project dependencies and provides extended build life cycles and conventions for compiling, testing, packaging and installing your Java project.

In case you already use Maven in your project you can just add Citrus as a test-scoped dependency.

As Maven handles all project dependencies automatically you do not need to download any Citrus project artifacts in advance. If you are new to Maven please refer to the official Maven documentation and find out how to set up a Maven project.

Assuming you have a proper Maven project setup you can integrate Citrus with it. Just add the Citrus project dependencies in your Maven pom.xml as a dependency like follows.

- We add Citrus as test-scoped project dependency to the project POM (pom.xml)

Add Citrus base dependency

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-base</artifactId>
  <version>${citrus.version}</version>
  <scope>test</scope>
</dependency>
```

- The dependency above adds the base functionality of Citrus. You need to add modules as you require them.

Add modules as required in your project. For instance Http support

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-http</artifactId>
  <version>${citrus.version}</version>
  <scope>test</scope>
</dependency>
```

Choose test runtime (JUnit, TestNG, Cucumber) that is used to run the tests.

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-testng</artifactId>
  <version>${citrus.version}</version>
  <scope>test</scope>
</dependency>
```

- Citrus integrates nicely with the Spring framework. In case you want to use the Spring dependency injection and bean configuration capabilities just add the Spring support in Citrus.

Add Spring support

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-spring</artifactId>
  <version>${citrus.version}</version>
  <scope>test</scope>
</dependency>
```

- Also, Citrus provides a Maven plugin that you can add. The plugin provides some convenience functionalities such as creating new tests from command line.

Add Citrus Maven plugin

```
<plugin>
  <groupId>org.citrusframework.mvn</groupId>
  <artifactId>citrus-maven-plugin</artifactId>
  <version>${citrus.version}</version>
  <configuration>
    <author>Donald Duck</author>
    <targetPackage>org.citrusframework</targetPackage>
  </configuration>
</plugin>
```

The Maven project is now ready to use Citrus. You can start writing new test cases with the Citrus Maven plugin:

Create new test

```
mvn citrus:create-test
```

The command above starts an interactive command line interface that helps you to create a test.

Once you have written the Citrus test cases you can execute them automatically in your Maven software build lifecycle. The tests will be included into your project's integration-test phase using the Maven failsafe plugin. Here is a sample failsafe configuration for Citrus.

Maven failsafe plugin

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>${maven.failsafe.version}</version>
  <executions>
    <execution>
      <id>integration-tests</id>
      <goals>
        <goal>integration-test</goal>
        <goal>verify</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

The Citrus test sources go to the default Maven test source directory `src/test/java` and `src/test/resources`.

You are now ready to call the usual Maven **verify** goal (`mvn verify`) in order to build your project and run the tests. The Citrus integration tests are executed automatically during the build process.

Run all tests with Maven

```
mvn verify
```

Run single test by its name

```
mvn verify -Dit.test=MyFirstCitrusIT
```

The Maven failsafe plugin by default executes tests with specific name pattern. This is because integration tests should not execute in Maven unit test phase, too. Your integration tests should follow the failsafe name pattern with each test name beginning or ending with **'IT'**.

If you need additional assistance in setting up a Citrus Maven project please visit our Maven setup tutorial on https://citrusframework.org/tutorials.html.

3.2. Using Gradle

As Citrus tests are nothing but normal JUnit or TestNG tests the integration to Gradle as build tool is as easy as adding the source files to a folder in your project. With the Gradle task execution for integration tests you are able to execute the Citrus tests like you would do with normal unit tests.

The Gradle build configuration goes to the **build.gradle** and **settings.gradle** files. The files define the project name and the project version.

Gradle project configuration

```
rootProject.name = 'citrus-sample-gradle'
group 'org.citrusframework.samples'
version '${citrus.version}'
```

The Citrus libraries are available on Maven central repository. This means you should add this repository so Gradle knows how to download the required Citrus artifacts.

Add Maven central repository

```
repositories {
    mavenCentral()
}
```

Citrus stable release versions are available on Maven central. Now let's move on with adding the Citrus libraries to the project.

Add Citrus test scoped dependencies

```
dependencies {
    testCompile group: 'org.citrusframework', name: 'citrus-base', version:
'${citrus.version}'
    testCompile group: 'org.citrusframework', name: 'citrus-http', version:
'${citrus.version}'
    testCompile group: 'org.testng', name: 'testng', version: '6.11'
    [...]
}
```

Citrus provides various modules that encapsulate different functionalities. The `citrus-base` module is the basis and holds core functionality. In addition, you may add further modules that match your project needs (e.g. add Http support with `citrus-http`).

As a runtime the project chose to use TestNG. You can also use JUnit or Cucumber as a test runtime. Each of those frameworks integrates seamlessly with the Gradle build.

Choose test runtime provider

```
test {
    useTestNG()
}
```

Of course JUnit is also supported. This completes the Gradle build configuration settings. You can move on to writing some Citrus integration tests and add those to **src/test/java** directory.

You can use the Gradle wrapper for compile, package and test the sample with Gradle build command line.

Run the build with Gradle

```
gradlew clean build
```

This executes all Citrus test cases during the build. You will be able to see Citrus performing some integration test logging output.

If you just want to execute all tests you can call:

Run all tests

```
gradlew clean check
```

Of course, you can also run the Citrus tests from your favorite Java IDE. Just start the Citrus test as a normal unit test using the Gradle integration in IntelliJ, Eclipse or VSCode.

Chapter 4. Runtimes

A Citrus test case is nothing but Java unit test leveraging well-known standard tools such as JUnit, TestNG or Cucumber as a runtime.

Chances are very high that Java developers are familiar with at least one of the standard tools. Everything you can do with JUnit and TestNG you can do with Citrus tests as well (e.g. Maven build integration, run tests from your favorite IDE, include tests into a continuous build tool).

> Why is Citrus related to unit test frameworks, although it represents a framework for integration testing? The answer to this question is quite simple: This is because Citrus wants to benefit from standard libraries such as JUnit and TestNG for Java test execution. Both unit testing frameworks offer various ways of execution and are widely supported by other tools (e.g. continuous build, build lifecycle, development IDE).

You can write the Citrus test code in a Java domain specific language or in form of an XML test declaration file that gets loaded as part of the test. The Java domain specific language in Citrus is a set of classes and methods to leverage the test code in form of a fluent API. Users can simply configure a test action with the different options using a fluent builder pattern style DSL.

Citrus Java DSL

```
@CitrusTest(name = "Hello_IT")
public void helloTest() {
    given(
        variable("user", "Citrus")
    );

    then(
        echo().message("Hello ${user}!"
    ));
}
```

The sample above is a very simple Citrus test that creates a test variable and prints a message to the console. The Java DSL you write is the same for all runtimes (JUnit, TestNG, Cucumber, etc.) and should help you to also solve very complex test scenarios.

The following sections have a closer look at the different runtimes for Citrus.

4.1. TestNG

TestNG stands for next generation testing and has had a great influence in adding Java annotations to the unit test community. Citrus is able to define tests as executable TestNG Java classes.

> The TestNG support is shipped in a separate Maven module. You need to include the module as a dependency in your project.

TestNG module dependency

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-testng</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

4.1.1. TestNG tests

See the following sample showing how to write a Citrus test on top of TestNG:

TestNG Citrus test

```
package org.citrusframework.samples;

import org.testng.annotations.Test;
import org.citrusframework.annotations.CitrusTest;
import org.citrusframework.testng.TestNGCitrusSupport;

@Test
public class Simple_IT extends TestNGCitrusSupport {

    @CitrusTest(name = "Simple_IT")
    public void simpleTest() {
        description("First example showing the basic Java DSL!");

        given(
            variable("user", "Citrus")
        );

        then(
            echo().message("Hello ${user}!"
        ));
    }
}
```

If you are familiar with TestNG you will see that the Java class is a normal TestNG test class using the usual `@Test` annotation. For convenience reasons you can extend a basic Citrus TestNG base class `TestNGCitrusSupport` which enables the Citrus test execution as well as the Java DSL features for us.

> You can also combine Citrus with the Spring framework and its dependency injection and IoC capabilities. In order to enable Spring support in Citrus add the `citrus-spring` module to your project and extend `TestNGCitrusSpringSupport` as a base class. With the Spring support in Citrus the test is able to use `@Autowired` annotations for injecting Spring beans into the test class and you can define the Spring application context with `@Configuration` annotations for instance.

In addition, the test methods use the **@CitrusTest** annotation which allows setting properties such as test names and packages.

The Citrus test logic goes directly as the method body with using the Citrus Java domain specific language features. As you can see the Java DSL is able to follow BDD (Behavior Drive Design) principles with Given-When-Then syntax. As an alternative to that you can just use `run()` for all test actions.

Pure test action DSL

```
@CitrusTest(name = "Simple_IT")
public void simpleTest() {
    description("First example showing the basic Java DSL!");

    run(variable("user", "Citrus"));

    run(
        echo().message("Hello ${user}!"
    ));
}
```

The great news is that you can still use the awesome TestNG features in with the Citrus test class (e.g. parallel test execution, test groups, setup and tear down operations and so on). Just to give an example we can simply add a test group to our test like this:

Set test groups

```
@Test(groups = {"long-running"})
public void longRunningTest() {
    ...
}
```

For more information on TestNG please visit the official TestNG website, where you find a complete reference documentation. The following sections deal with a subset of these TestNG features in particular.

4.1.2. Use TestNG data providers

TestNG as a framework comes with lots of great features such as data providers. Data providers execute a test case several times with different test data. Each test execution works with a specific parameter value. You can use data provider parameter values as test variables in Citrus. See the next listing on how to use TestNG data providers in Citrus:

TestNG Citrus data provider test

```
public class DataProviderIT extends TestNGCitrusSupport {

    @CitrusTest
    @CitrusParameters({"message", "delay"})
    @Test(dataProvider = "messageDataProvider")
    public void dataProvider(String message, Long sleep) {
        run(echo(message));
        run(sleep().milliseconds(sleep));

        run(echo("${message}"));
        run(echo("${delay}"));
    }

    @DataProvider
    public Object[][] messageDataProvider() {
        return new Object[][] {
                { "Hello World!", 300L },
                { "Citrus rocks!", 1000L },
                { "Hi from Citrus!", 500L },
        };
    }
}
```

Above test case method is annotated with TestNG data provider called **messageDataProvider** . In the same class you can write the data provider that returns a list of parameter values. TestNG will execute the test case several times according to the provided parameter list. Each execution is shipped with the respective parameter value.

According to the **@CitrusParameter** annotation the test will have test variables called `message` and `delay`.

With Sharding

TestNG data provider based tests can additionally be run in a distributed manner, facilitating test sharding. The following snipped showcases you how to get started within a Spring test case environment.

Sharded TestNG Citrus data provider test

```
import static org.citrusframework.common.TestLoader.SPRING;
import static org.citrusframework.sharding.Shard.createShard;

import org.citrusframework.annotations.CitrusTestSource;
import org.citrusframework.testng.CitrusParameters;
import org.citrusframework.testng.spring.TestNGCitrusSpringSupport;
import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

public class DataProviderIT extends TestNGCitrusSupport {

    @CitrusParameters({"message", "delay"})
    @Test(dataProvider = "messageDataProvider")
    @CitrusTestSource(type = SPRING, name = "DataProviderIT")
    public void dataProvider(String message, Long sleep) {
        run(echo(message));
        run(sleep().milliseconds(sleep));
    }

    @DataProvider("messageDataProvider")
    public Object[][] messageDataProvider() {
        return createShard(
            new Object[][] {
                { "Hello World!", 300L },
                { "Citrus rocks!", 1000L },
                { "Hi from Citrus!", 500L },
            }
        );
    }
}
```

All the available configuration options are documented withing the chapter Sharding Test Cases

4.1.3. Run tests in parallel

Integration tests tend to be more time-consuming compared to pure unit tests when it comes to execute tests. This is because integration tests often need to initialize test infrastructure (e.g. test servers, database connections). Running tests in parallel can reduce the overall test suite time a lot.

When running tests in parallel you need to make sure each test operates on its own set of resources. Tests must not share components such as the Citrus Java DSL test action runner or the test context.

You should be using the resource injection to make sure each test operates on its own resources.

Resource injection

```
public class ResourceInjection_IT extends TestNGCitrusSupport {

    @Test
    @CitrusTest
    public void injectResources(@Optional @CitrusResource TestCaseRunner runner,
                                @Optional @CitrusResource TestContext context) {

        runner.given(
            createVariable("random", "citrus:randomNumber(10)")
        );

        runner.run(
            echo("The random number is: ${random}")
        );
    }
}
```

First of all the method parameters must be annotated with `@Optional` because the values are not injected by TestNG itself but by the Citrus base test class. Finally, the parameter requires the `@CitrusResource` annotation in order to mark the parameter for Citrus resource injection.

Now each method uses its own resource instances which makes sure that parallel test execution can take place without having the risk of side effects on other tests running at the same time. Of course, you also need to make sure that the message exchange in your tests is ready to be performed in parallel (e.g. use message selectors).

4.2. JUnit5

With JUnit version 5 the famous unit test framework offers a new major version. The JUnit platform provides awesome extension points for other frameworks like Citrus to integrate with the unit testing execution.

Citrus provides extensions in order to enable Citrus related dependency injection and parameter resolving in your JUnit5 test.

> The JUnit5 support is shipped in a separate Maven module. You need to include the module as a dependency in your project.

JUnit5 module dependency

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-junit5</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

4.2.1. Citrus extension

You can use the Citrus JUnit5 extension on your test as follows:

JUnit5 Citrus test

```
package org.citrusframework.samples;

import org.citrusframework.GherkinTestActionRunner;
import org.citrusframework.annotations.CitrusTest;
import org.citrusframework.junit.jupiter.CitrusSupport;
import org.junit.jupiter.api.Test;

@CitrusSupport
public class Simple_IT {

    @Test
    @CitrusTest(name = "Simple_IT")
    public void simpleTest(@CitrusResource GherkinTestActionRunner runner) {
        runner.description("First example showing the basic Java DSL!");

        runner.given(
            variable("user", "Citrus")
        );

        runner.then(
            echo().message("Hello ${user}!"
        ));
    }
}
```

The class above is using the JUnit5 `@Test` annotation as a normal unit test would do. The `@CitrusSupport` annotation marks the test to use the Citrus JUnit5 extension. This enables us to use the `@CitrusTest` annotation on the test and adds support for the parameter injection for the `TestActionRunner`.

> You can use the `@CitrusSupport` annotation, or you can use the classic `@ExtendWith(CitrusExtension.class)` annotation to enable the Citrus support for JUnit5.

The Citrus Java DSL runner is the entrance to the Java fluent API provided by Citrus. The sample above uses the Gherkin test runner variation for leveraging the BDD (Behavior Driven Development) style Given-When-Then syntax.

You can also inject the current `TestContext` in order to get access to the current test variables used by Citrus.

You can also combine Citrus with the Spring framework and its dependency injection and IoC capabilities. In order to enable Spring support in Citrus add the `citrus-spring` module to your project and use the `@ExtendWith(CitrusSpringExtension.class)` annotation. With the Spring support in Citrus the test is able to load components via the Spring application context.

4.2.2. Endpoint injection

In addition to injecting test resources you can also inject endpoints via `@CitrusEndpoint` annotated field injection in your test class. This enables you to inject endpoint components that are defined in the Citrus context configuration.

JUnit5 Citrus endpoint injection

```
package org.citrusframework.samples;

import org.citrusframework.annotations.*;
import org.citrusframework.GherkinTestActionRunner;
import org.citrusframework.junit.jupiter.CitrusSupport;
import org.citrusframework.http.client.HttpClient;
import org.junit.jupiter.api.Test;
import org.springframework.http.HttpStatus;

@CitrusSupport
public class Simple_IT {

    @CitrusEndpoint
    private HttpClient httpClient;

    @Test
    @CitrusTest
    public void test(@CitrusResource GherkinTestActionRunner runner) {
        runner.http().client(httpClient)
                     .send()
                     .get("/hello");

        runner.http().client(httpClient)
                     .receive()
                     .response(HttpStatus.OK);
    }
}
```

4.2.3. Citrus Spring extension

Spring is a famous dependency injection framework that also provides support for JUnit5. Citrus is able to load its components as Spring beans in an application context. The Citrus JUnit5 extension works great with the Spring extension.

The Spring extension loads the application context and Citrus adds all components to the Spring

bean configuration.

JUnit5 Citrus Spring test

```
@CitrusSpringSupport
@ContextConfiguration(classes = CitrusSpringConfig.class)
public class SpringBean_IT {

    @Autowired
    private DirectEndpoint direct;

    @Test
    @CitrusTest
    void springBeanTest(@CitrusResource TestActionRunner actions) {
        actions.$(send().endpoint(direct)
                    .message()
                    .body("Hello from Citrus!"));

        actions.$(receive().endpoint(direct)
                    .message()
                    .body("Hello from Citrus!"));
    }
}
```

The test now uses the `@CitrusSpringSupport` annotation which combines the `@ExtendWith(CitrusSpringExtension.class)` and `@ExtendWith(SpringExtension.class)` annotation. This way the test combines the Spring application context management with the Citrus Java DSL functionality.

You can load Spring beans with `@Autowired` into your test. Also, you can use the `@CitrusResource` annotations to inject the test action runner fluent Java API.

The Spring application context should use the basic `CitrusSpringConfig` configuration class to load all Citrus components as Spring beans. You can customize the Spring application context by adding more configuration classes.

4.2.4. Test Sharding

You can run your JUnit5 test cases in order in a distributed manner, facilitating test sharding. The following snipped showcases you how to get started within a Spring/XML test case environment. Note that sharding is available with all kind of `Stream`-sources.

Sharded JUnit5 Spring/XML test

```
import static org.citrusframework.junit.jupiter.CitrusTestFactorySupport.springXml;
import static org.citrusframework.sharding.Shard.createShard;

import org.citrusframework.config.CitrusSpringConfig;
import org.citrusframework.junit.jupiter.spring.CitrusSpringSupport;
import org.citrusframework.junit.jupiter.spring.CitrusSpringXmlTestFactory;
import org.springframework.test.context.ContextConfiguration;

@CitrusSpringSupport
@ContextConfiguration(classes = {CitrusSpringConfig.class})
class SpringBeanXml_IT {

    @CitrusSpringXmlTestFactory
    Stream<DynamicTest> shardedSpringBeanXml_IT() {
        return createShard(
            springXml().packageScan("org.citrusframework.junit.jupiter.simple")
        );
    }
}
```

All the available configuration options are documented withing the chapter Sharding Test Cases

4.3. JUnit4

JUnit4 is still very popular and widely supported by many tools even though there is a new major version with JUnit5 already available. In general Citrus supports both JUnit4 and JUnit5 as test execution framework.

> The JUnit4 support is shipped in a separate Maven module. You need to include the module as a dependency in your project.

JUnit4 module dependency

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-junit</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

4.3.1. JUnit4 tests

See the following sample test class that uses JUnit4.

JUnit4 Citrus test

```
package org.citrusframework.samples;

import org.junit.Test;
import org.citrusframework.annotations.CitrusTest;
import org.citrusframework.junit.JUnit4CitrusSupport;

public class Simple_IT extends JUnit4CitrusSupport {

    @Test
    @CitrusTest(name = "Simple_IT")
    public void simpleTest() {
        description("First example showing the basic Java DSL!");

        given(
            variable("user", "Citrus")
        );

        then(
            echo().message("Hello ${user}!"
        ));
    }
}
```

The simple test class above uses the normal `@Test` annotation and extends the base class `JUnit4CitrusSupport`. This is the most convenient way to access the Citrus Java DSL capabilities. As an alternative you may switch to using the `CitrusJUnit4Runner` in your test class.

The fine thing here is that we are still able to use all JUnit features such as before/after hooks or ignoring tests.

After the test run the result is reported exactly like a usual JUnit unit test would do. This also means that you can execute this Citrus JUnit class like every other JUnit test, especially out of any Java IDE, with Maven, with Gradle and so on.

> You can also combine Citrus with the Spring framework and its dependency injection and IoC capabilities. In order to enable Spring support in Citrus add the `citrus-spring` module to your project and extend `JUnit4CitrusSpringSupport` as a base class. With the Spring support in Citrus the test is able to use `@Autowired` annotations for injecting Spring beans into the test class and you can define the Spring application context with `@Configuration` annotations for instance.

4.3.2. Run tests in parallel

Integration tests tend to be more time-consuming compared to pure unit tests when it comes to execute tests. This is because integration tests often need to initialize test infrastructure (e.g. test servers, database connections). Running tests in parallel can reduce the overall test suite time a lot.

When running tests in parallel you need to make sure each test operates on its own set of resources. Tests must not share components such as the Citrus Java DSL test action runner or the test context.

You should be using the resource injection to make sure each test operates on its own resources.

Resource injection

```
public class ResourceInjection_IT extends JUnit4CitrusSupport {

    @Test
    @CitrusTest
    public void injectResources(@CitrusResource TestCaseRunner runner,
                                @CitrusResource TestContext context) {

        runner.given(
            createVariable("random", "citrus:randomNumber(10)")
        );

        runner.run(
            echo("The random number is: ${random}")
        );
    }
}
```

The method parameters require the `@CitrusResource` annotations in order to mark the parameter for Citrus resource injection.

Now each method uses its own resource instances which makes sure that parallel test execution can take place without having the risk of side effects on other tests running at the same time. Of course, you also need to make sure that the message exchange in your tests is ready to be performed in parallel (e.g. use message selectors).

4.4. QuarkusTest

Quarkus has emerged into a popular enterprise Java framework. For unit and integration testing the Quarkus framework provides integrations with JUnit Jupiter. Citrus adds a Quarkus test resource implementation that allows developers to combine Citrus with Quarkus during testing. You can use the Citrus test resource annotations on your Quarkus tests and include Citrus capabilities into arbitrary Quarkus tests.

The Citrus QuarkusTest extension is shipped in a separate Maven module. You need to include the module as a dependency in your project accordingly.

Citrus Quarkus module dependency

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-quarkus</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

Usually a Quarkus test is annotated with the `@QuarkusTest` or `QuarkusIntegrationTest` annotation. Users just add an annotation named `@CitrusSupport` to also enable Citrus capabilities on the test.

The Citrus support will automatically hook into the QuarkusTest lifecycle management to inject Citrus resources with `@CitrusResource` annotation. Also, the Citrus extension makes sure to start a proper Citrus instance and call before/after suite and before/after test handlers.

This way you are able to combine Citrus with `@QuarkusTest` annotated classes very easily.

Enable Citrus support on QuarkusTest

```
@QuarkusTest
@CitrusSupport
public class DemoApplicationTest {

    @CitrusFramework
    private Citrus citrus;

    @CitrusResource
    private TestCaseRunner t;

    @CitrusResource
    private TestContext context;

    @Test
    void shouldVerifyDemoApp() {
        t.when(
            send()
                .endpoint("messageEndpoint")
                .message()
                .body("How about Citrus!?")
        );

        t.when(
            receive()
                .endpoint("messageEndpoint")
                .message()
                .body("Citrus rocks!")
        );
    }
}
```

The `@CitrusSupport` annotation enables the Citrus features on the test. First of all users may inject Citrus related resources such as `TestCaseRunner` or the `TestContext`.

As usual the `TestCaseRunner` is the entrance to the Citrus domain specific language for running arbitrary Citrus actions as part of the test.

4.4.1. Endpoint configuration

The test is able to configure Message endpoints to connect to different messaging transports as part of the test.

Configure message endpoints

```
@QuarkusTest
@CitrusSupport
public class DemoApplicationTest {

    @BindToRegistry
    private final KafkaEndpoint bookings = new KafkaEndpointBuilder()
            .topic("bookings")
            .build();

    @CitrusResource
    private TestCaseRunner t;

    @Test
    void shouldVerifyDemoApp() {
        t.when(
            send()
                .endpoint(bookings)
                .message()
                .body("How about Citrus!?")
        );

        t.when(
            receive()
                .endpoint(bookings)
                .message()
                .body("Citrus rocks!")
        );
    }
}
```

Creating new message endpoints is very easy. Just use the proper endpoint builder and optionally bind the new endpoint to the Citrus bean registry via `@BindToRegistry` annotation. You may then use the message endpoint in all `send` and `receive` test actions in order to exchange messages.

You may move the endpoint configuration into a separate class and load the endpoints with the configuration class as follows:

EndpointConfig.class

```
public class EndpointConfig {

    @BindToRegistry
    public KafkaEndpoint bookings() {
        return new KafkaEndpointBuilder()
            .topic("bookings")
            .build();
    }
}
```

The endpoint configuration class uses `@BindToRegistry` members or methods to add beans to the Citrus registry. The configuration class may be referenced by many tests then using the `@CitrusConfiguration` annotation.

Load endpoint config classes

```
@QuarkusTest
@CitrusSupport
@CitrusConfiguration(classes = EndpointConfig.class)
public class DemoApplicationTest {

    @CitrusResource
    private KafkaEndpoint bookings;

    @CitrusResource
    private TestCaseRunner t;

    @Test
    void shouldVerifyDemoApp() {
        t.when(
            send()
                .endpoint(bookings)
                .message()
                .body("How about Citrus!?")
        );

        t.when(
            receive()
                .endpoint(bookings)
                .message()
                .body("Citrus rocks!")
        );
    }
}
```

Citrus loads the configuration class and injects the `KafkaEndpoint` instance to the test with `@CitrusResource` annotation.

4.4.2. Load dynamic tests

Citrus supports many test languages besides writing tests in pure Java. Users can load tests written in XML, YAML, Groovy and many more via dynamic tests.

Load YAML tests

```
@QuarkusTest
@CitrusSupport
@CitrusConfiguration(classes = EndpointConfig.class)
public class DemoApplicationTest {

    @CitrusTestFactory
    public Stream<DynamicTest> loadYamlTests() {
        return
CitrusTestFactorySupport.factory(TestLoader.YAML).packageScan("some.package.name");
    }
}
```

The example above loads YAML test case definitions and runs those as dynamic tests with JUnit Jupiter. The package scan loads all files in the given folder and runs the tests via Citrus. All YAML tests are able to reference the message endpoints configured in the configuration class `EndpointConfig.class`.

A sample YAML test may look like this:

my-test.yaml

```
name: my-test
actions:
  - send:
      endpoint: bookings
      message:
        body:
          data: How about Citrus!?
  - receive:
      endpoint: bookings
      timeout: 5000
      message:
        body:
          data: Citrus rocks!
```

4.4.3. Set application properties

The `@QuarkusTest` annotation will automatically start the application under test. Citrus provides the ability to programmatically set application properties before the Quarkus application is started. This is important when you need to overwrite configuration based on test message endpoints configured in the test.

The next example shows a Citrus enabled Quarkus test that supplies a set of application properties to configure the application under test.

Supply application properties

```
@QuarkusTest
@CitrusSupport(applicationPropertiesSupplier = DemoAppConfigurationSupplier.class)
@CitrusConfiguration(classes = EndpointConfig.class)
public class DemoApplicationTest {

    // ...
}
```

The `DemoAppConfiguration` class implements the `Supplier` interface and set a config property. This property will be set on the application under test.

DemoAppConfigurationSupplier.class

```
public class DemoAppConfigurationSupplier implements ApplicationPropertiesSupplier {

    @Override
    public Map<String, String> get() {
        Map<String, String> conf = new Hasmap<>();
        conf.put("quarkus.log.level", "INFO");
        conf.put("greeting.message", "Hello, Citrus rocks!");
        return conf;
    }
}
```

The application properties supplier is able to set Quarkus properties as well as application domain properties. The example above sets `greeting.message` property which can be referenced in the Quarkus application:

DemoApplication

```
@ApplicationScoped
public class DemoApplication {

    private static final Logger logger = Logger.getLogger(DemoApplication.class);

    @ConfigProperty(name = "greeting.message")
    String message;

    void onStart(@Observes StartupEvent ev) {
        logger.info(message);
    }
}
```

4.4.4. Testcontainers support

Citrus integrates with Testcontainers to easily start/stop Testcontainers instances as part of the test. You can leverage the Citrus Testcontainers features within a Quarkus test very easily. Citrus implements Quarkus test resources for each of the supported containers (AWS LocalStack, Kafka, Redpanda, ...).

The following example starts an AWS LocalStack Testcontainers instance and uses the S3 service on that container to upload a file to the S3 bucket. The Quarkus application under test should handle this S3 file then.

AwsS3SourceTest

```
@QuarkusTest
@CitrusSupport
@LocalStackContainerSupport(services = LocalStackContainer.Service.S3,
containerLifecycleListener = AwsS3SourceTest.class)
public class AwsS3SourceTest implements
ContainerLifecycleListener<LocalStackContainer> {

    @CitrusResource
    private TestCaseRunner tc;

    @CitrusResource
    private LocalStackContainer localStackContainer;

    @Test
    public void shouldHandleUploadedS3File() {
        tc.given(this::uploadS3File);

        // verify that the Quarkus application has handled the S3 file
    }

    private void uploadS3File(TestContext context) {
        S3Client s3Client = createS3Client(localStackContainer);

        CreateMultipartUploadResponse initResponse = s3Client.createMultipartUpload(b
-> b.bucket(s3BucketName).key(s3Key));
        String etag = s3Client.uploadPart(b -> b.bucket(s3BucketName)
                        .key(s3Key)
                        .uploadId(initResponse.uploadId())
                        .partNumber(1),
                RequestBody.fromString(s3Data)).eTag();
        s3Client.completeMultipartUpload(b -> b.bucket(s3BucketName)
                .multipartUpload(CompletedMultipartUpload.builder()
                        .parts(Collections.singletonList(CompletedPart.builder()
                                .partNumber(1)
                                .eTag(etag).build())).build())
                .key(s3Key)
                .uploadId(initResponse.uploadId()));
    }
```

```
    @Override
    public Map<String, String> started(LocalStackContainer container) {
        S3Client s3Client = createS3Client(container);

        s3Client.createBucket(b -> b.bucket(s3BucketName));

        Map<String, String> conf = new HashMap<>();
        conf.put("my.app.aws-s3-source.accessKey", container.getAccessKey());
        conf.put("my.app.aws-s3-source.secretKey", container.getSecretKey());
        conf.put("my.app.aws-s3-source.region", container.getRegion());
        conf.put("my.app.aws-s3-source.bucketNameOrArn", s3BucketName);
        conf.put("my.app.aws-s3-source.uriEndpointOverride",
container.getServiceEndpoint().toString());
        conf.put("my.app.aws-s3-source.overrideEndpoint", "true");
        conf.put("my.app.aws-s3-source.forcePathStyle", "true");

        return conf;
    }

    private static S3Client createS3Client(LocalStackContainer container) {
        return S3Client.builder()
                .endpointOverride(container.getServiceEndpoint())
                .credentialsProvider(
                        StaticCredentialsProvider.create(
                                AwsBasicCredentials.create(container.getAccessKey(),
container.getSecretKey())
                        )
                )
                .forcePathStyle(true)
                .region(Region.of(container.getRegion()))
                .build();
    }
}
```

A few things happened in this example and let's explain those features one after another. First thing to notice is the `@LocalStackContainerSupport` annotation that makes Citrus run the AWS LocalStack Testcontainers instance. Also, the annotation provides the enabled services on that container (`services = LocalStackContainer.Service.S3`). This starts the Testcontainers instance as part of the Quarkus test.

The test also implements the `ContainerLifecycleListener` interface. This enables the test to handle the container instance after it has been started. This is a good place to create an S3 client and the bucket for the test.

Create S3 client

```
@Override
public Map<String, String> started(LocalStackContainer container) {
    S3Client s3Client = createS3Client(container);

    s3Client.createBucket(b -> b.bucket(s3BucketName));

    Map<String, String> conf = new HashMap<>();
    conf.put("my.app.aws-s3-source.accessKey", container.getAccessKey());
    conf.put("my.app.aws-s3-source.secretKey", container.getSecretKey());
    conf.put("my.app.aws-s3-source.region", container.getRegion());
    conf.put("my.app.aws-s3-source.bucketNameOrArn", s3BucketName);
    conf.put("my.app.aws-s3-source.uriEndpointOverride",
container.getServiceEndpoint().toString());
    conf.put("my.app.aws-s3-source.overrideEndpoint", "true");
    conf.put("my.app.aws-s3-source.forcePathStyle", "true");

    return conf;
}
```

Also, the started listener may return some application properties that get set for the Quarkus application under test. This is the opportunity to set the Testcontainers connection settings for the Quarkus application.

Obviously the Quarkus application uses some property based configuration with the `my.app.*` properties. The test is able to reference the Testcontainers exposed settings as values for these properties (e.g. `my.app.aws-s3-source.accessKey=container.getAccessKey()`).

With this configuration in place the test is able to upload and S3 file to the test bucket on the Testcontainers instance with the `uploadS3File()` method. This should trigger the Quarkus application under test to handle the new file accordingly. We can add some verification and assertion steps to verify that the Quarkus application has handled the S3 file.

This is how Citrus is able to start Testcontainers instances as part of a Quarkus test. The application properties supplier as well as the container lifecycle listener interfaces allow us to connect the Quarkus application with the Testcontainers instance. The test is able to use the services on the Testcontainers instance to trigger some test data that is consumed by the application under test.

Please also have a look into the other provided Testcontainers annotations in Citrus:

- @LocalStackContainerSupport
- @KakfaContainerSupport
- @RedpandaContainerSupport
- @TestcontainersSupport

All of these annotations allow you to start Testcontainers instances as part of your Quarkus test and provides the opportunity to participate in the container lifecycle to access managed ports and connectivity settings for instance.

4.5. Cucumber

Behavior driven development (BDD) is a very popular concept when it comes to find a common understanding of test scopes test logic. The idea of defining and describing the software behavior as basis for all tests in prior to translating those feature descriptions into executable tests is a very interesting approach because it includes the technical experts as well as the domain experts.

With BDD the domain experts should be able to read and verify tests and the technical experts get a detailed description of what should happen in the test.

The test scenario descriptions follow the Gherkin syntax with a **"Given-When-Then"** structure. The Gherkin language is business readable and helps to explain business logic with help of concrete examples.

There are several frameworks in the Java community supporting BDD concepts. Citrus has dedicated support for the Cucumber framework because Cucumber is well suited for extensions and plugins. So with the Citrus and Cucumber integration you can write Gherkin syntax scenarios in order to run those as Citrus integration tests.

The Cucumber components in Citrus are located in a separate Maven module. You need to include the module as a Maven dependency to your project.

Cucumber module dependency

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-cucumber</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

Cucumber works with both JUnit and TestNG as unit testing framework. You can choose which framework to use with Cucumber. So following from that we need a Maven dependency for the unit testing framework support:

Cucumber JUnit support

```
<dependency>
  <groupId>io.cucumber</groupId>
  <artifactId>cucumber-junit</artifactId>
  <version>${cucumber.version}</version>
</dependency>
```

In order to enable Citrus Cucumber support we need to specify a special object factory in the environment. The most comfortable way to specify a custom object factory is to add this property to the `cucumber.properties` in classpath.

cucumber.properties

```
cucumber.object-factory=org.citrusframework.cucumber.backend.CitrusObjectFactory
```

This special object factory takes care of creating all step definition instances. The object factory is able to inject `@CitrusResource` annotated fields in step classes. We will see this later on in the examples. The usage of this special object factory is mandatory in order to combine Citrus and Cucumber capabilities.

The **CitrusObjectFactory** will automatically initialize the Citrus world for us. This includes the default Citrus context configuration that is automatically loaded within the object factory. So you can define and use Citrus components as usual within your test.

After these preparation steps you are able to combine Citrus and Cucumber in your project.

In case you want to use Spring support in Citrus with a Spring application context you should use the following factory implementation.

cucumber.properties

```
cucumber.object-
factory=org.citrusframework.cucumber.backend.spring.CitrusSpringObjectFactory
```

4.5.1. Cucumber options

Cucumber is able to run tests with JUnit. The basic test case is an empty test which uses the respective JUnit runner implementation from cucumber.

MyFeature.java

```
@RunWith(Cucumber.class)
@CucumberOptions(
  plugin = { "pretty", "org.citrusframework.cucumber.CitrusReporter" } )
public class MyFeatureIT {
}
```

The test case above uses the **Cucumber** JUnit test runner. In addition to that we give some options to the Cucumber execution. In case you want to have the usual Citrus test results reported you can add the special Citrus reporter implementation `org.citrusframework.cucumber.CitrusReporter`. This class is responsible for printing the Citrus test summary. This reporter extends the default Cucumber reporter so the default Cucumber report summary is also printed to the console.

That completes the JUnit class configuration. Now we are able to add feature stories and step definitions to the package of our test **MyFeatureIT** . Cucumber and Citrus will automatically pick up step definitions and glue code in that test package. So let's write a feature story **echo.feature** right next to the **MyFeatureIT** test class.

echo.feature

```
Feature: Echo service

  Scenario: Say hello
    Given My name is Citrus
    When I say hello to the service
    Then the service should return: "Hello, my name is Citrus!"

  Scenario: Say goodbye
    Given My name is Citrus
    When I say goodbye to the service
    Then the service should return: "Goodbye from Citrus!"
```

As you can see this story defines two scenarios with the Gherkin **Given-When-Then** syntax. Now we need to add step definitions that glue the story description to Citrus test actions. Let's do this in a new class **EchoSteps** .

EchoSteps.java

```
public class EchoSteps {

    @CitrusResource
    private TestCaseRunner runner;

    @Given("^My name is (.*)$")
    public void my_name_is(String name) {
        runner.variable("username", name);
    }

    @When("^I say hello.*$")
    public void say_hello() {
        runner.when(
                send("echoEndpoint")
                    .message()
                    .type(MessageType.PLAINTEXT)
                    .body("Hello, my name is ${username}!"));
    }

    @When("^I say goodbye.*$")
    public void say_goodbye() {
        runner.when(
                send("echoEndpoint")
                    .message()
                    .type(MessageType.PLAINTEXT)
                    .body("Goodbye from ${username}!"));
    }

    @Then("^the service should return: \"([^\"]*)\"$")
    public void verify_return(final String body) {
        runner.then(
                receive("echoEndpoint")
                    .message()
                    .type(MessageType.PLAINTEXT)
                    .body(body));
    }
}
```

The step definition class is a normal POJO that uses some annotations such as `@CitrusResource` annotated `TestCaseRunner`. The Citrus backend injects the test runner instance at runtime.

The step definition contains normal @Given, @When or @Then annotated methods that match the scenario descriptions in our feature file. Cucumber will automatically find matching methods and execute them. The methods add test actions to the test runner as we used to do in normal Java DSL tests.

That is a first combination of Citrus and Cucumber BDD. The feature file gets translated into step implementations that use Citrus test action runner Java API to run integration tests with behavior

driven development.

4.5.2. Cucumber XML steps

The previous section handled glue code in Java in form of step definitions accessing the Java test runner fluent API. This chapter deals with the same concept with just XML configuration.

Citrus provides a separate configuration namespace and schema definition for Cucumber related step definitions. Include this namespace into your Spring configuration in order to use the Citrus Cucumber configuration elements.

Spring bean configuration schema

```
<spring:beans xmlns:spring="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns="http://www.citrusframework.org/schema/cucumber/testcase"
     xsi:schemaLocation="
     http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.citrusframework.org/schema/cucumber/testcase
     http://www.citrusframework.org/schema/cucumber/testcase/citrus-cucumber-
testcase.xsd">

    [...]

</spring:beans>
```

The JUnit Cucumber feature class itself does not change. We still use the Cucumber JUnit runner implementation with some options specific to Citrus:

MyFeatureIT.java

```
@RunWith(Cucumber.class)
@CucumberOptions(
    plugin = { "pretty", "org.citrusframework.cucumber.CitrusReporter" } )
public class MyFeatureIT {
}
```

The feature file with its Gherkin scenarios does also not change:

echo.feature

```
Feature: Echo service

  Scenario: Say hello
    Given My name is Citrus
    When I say hello to the service
    Then the service should return: "Hello, my name is Citrus!"

  Scenario: Say goodbye
    Given My name is Citrus
    When I say goodbye to the service
    Then the service should return: "Goodbye from Citrus!"
```

In the feature package **my.company.features** we add a new XML file `EchoSteps.xml` that holds the new XML step definitions:

EchoSteps.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<spring:beans xmlns:citrus="http://www.citrusframework.org/schema/testcase"
      xmlns:spring="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns="http://www.citrusframework.org/schema/cucumber/testcase"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
                          http://www.springframework.org/schema/beans/spring-beans.xsd
                          http://www.citrusframework.org/schema/cucumber/testcase

http://www.citrusframework.org/schema/cucumber/testcase/citrus-cucumber-testcase.xsd">

  <step given="^My name is (.*)$" parameter-names="username">
    <citrus:create-variables>
      <citrus:variable name="username" value="${username}"/>
    </citrus:create-variables>
  </step>

  <step when="^I say hello.*$">
    <citrus:send endpoint="echoEndpoint">
      <citrus:message type="plaintext">
        <citrus:data>Hello, my name is ${username}!</citrus:data>
      </citrus:message>
    </citrus:send>
  </step>

  <step when="^I say goodbye.*$">
    <citrus:send endpoint="echoEndpoint">
      <citrus:message type="plaintext">
        <citrus:data>Goodbye from ${username}!</citrus:data>
      </citrus:message>
    </citrus:send>
  </step>

  <step then="^the service should return: &quot;([^&quot;]*)&quot;$" parameter-
names="body">
    <citrus:receive endpoint="echoEndpoint">
      <citrus:message type="plaintext">
        <citrus:data>You just said: ${body}</citrus:data>
      </citrus:message>
    </citrus:receive>
  </step>

</spring:beans>
```

The above step definition uses pure XML actions. Citrus will automatically read the step definition and add those to the Cucumber runtime. Following from that the step definitions are executed when matching a statement in the feature story.

The XML step files follow a naming convention. Citrus will look for all files located in the feature package with name pattern `**/**.Steps.xml` and load those definitions when Cucumber starts up.

The XML steps are able to receive parameters from the Gherkin regexp matcher. The parameters are passed to the step as test variable. The parameter names get declared in the optional attribute `parameter-names`. In the step definitions you can use the parameter names as test variables.

> The test variables are visible in all upcoming steps, too. This is because the test variables are global by default. If you need to set local state for a step definition you can use another attribute `global-context` and set it to `false` in the step definition. This way all test variables and parameters are only visible in the step definition. Other steps will not see the test variables.

> Another notable thing is the XML escaping of reserved characters in the pattern definition. You can see that in the last step where the **then** attribute is escaping quotation characters.

Escape reserved characters

```
<step then="^the service should return: &quot;([^&quot;]*)&quot;$" parameter-
names="body">
...
</step>
```

We have to do this because otherwise the quotation characters will interfere with the XML syntax in the attribute.

This completes the description of how to add XML step definitions to the cucumber BDD tests.

4.5.3. Cucumber Spring support

Cucumber provides support for Spring dependency injection in step definition classes. The Cucumber Spring capabilities are included in a separate module. So first of all we have to add this dependency to our project:

```
<dependency>
  <groupId>io.cucumber</groupId>
  <artifactId>cucumber-spring</artifactId>
  <version>${cucumber.version}</version>
</dependency>
```

The Citrus Cucumber extension has to handle things different when Cucumber Spring support is enabled. Therefore we use another object factory implementation that also support Cucumber Spring features. Change the object factory property in **cucumber.properties** to the following:

cucumber.properties

```
cucumber.object-
factory=org.citrusframework.cucumber.backend.spring.CitrusSpringObjectFactory
```

Now we are ready to add **@Autowired** Spring bean dependency injection to step definition classes:

EchoSteps.java

```
@ContextConfiguration(classes = CitrusSpringConfig.class)
public class EchoSteps {
    @Autowired
    private Endpoint echoEndpoint;

    @CitrusResource
    protected TestDesigner designer;

    @Given("^My name is (.*)$")
    public void my_name_is(String name) {
        designer.variable("username", name);
    }

    @When("^I say hello.*$")
    public void say_hello() {
        designer.send(echoEndpoint)
            .messageType(MessageType.PLAINTEXT)
            .payload("Hello, my name is ${username}!");
    }

    @When("^I say goodbye.*$")
    public void say_goodbye() {
        designer.send(echoEndpoint)
            .messageType(MessageType.PLAINTEXT)
            .payload("Goodbye from ${username}!");
    }

    @Then("^the service should return: \"([^\"]*)\"$")
    public void verify_return(final String body) {
        designer.receive(echoEndpoint)
            .messageType(MessageType.PLAINTEXT)
            .payload("You just said: " + body);
    }
}
```

As you can see we used Spring autowiring mechanism for the **echoEndpoint** field in the step definition. Also be sure to define the **@ContextConfiguration** annotation on the step definition. The Cucumber Spring support loads the Spring application context and takes care of dependency injection. We use the Citrus **CitrusSpringConfig** Java configuration because this is the main entrance for Citrus test cases. You can add custom beans and further Spring related configuration to

this Spring application context. If you want to add more beans for autowiring do so in the Citrus Spring configuration. Usually this is the default **citrus-context.xml** which is automatically loaded.

Of course, you can also use a custom Java Spring configuration class here. Please be sure to always import the Citrus Spring Java configuration classes, too.

As usual, we are able to use **@CitrusResource** annotated **TestCaseRunner** fields for building the Citrus integration test logic. With this extension you can use the full Spring testing power in your tests in particular dependency injection and also transaction management for data persistence tests.

4.5.4. YAKS step definitions

YAKS is a side project of Citrus and provides some predefined steps for typical integration test scenarios that you can use out-of-the-box.

You can basically define send/receive operations and many other predefined steps to handle Citrus test actions. As these steps are predefined in YAKS you just need to use them in your feature stories. The step definitions with glue to test actions is handled automatically in YAKS.

If you want to enable predefined steps support in your test you need to include the YAKS module as a Maven dependency.

YAKS module dependency

```
<dependency>
  <groupId>org.citrusframework.yaks</groupId>
  <artifactId>yaks-standard</artifactId>
  <version>${yaks.version}</version>
  <scope>test</scope>
</dependency>
```

After that you need to include the glue code package in your test class like this:

Include YAKS steps

```
@RunWith(Cucumber.class)
@CucumberOptions(
    extraGlue = { "org.citrusframework.yaks.standard" },
    plugin = { "pretty", "org.citrusframework.cucumber.CitrusReporter" } )
public class MyFeatureIT {

}
```

Instead of writing the glue code on our own in step definition classes we include the glue package `org.citrusframework.yaks.standard` as extra glue. This automatically loads all YAKS step definitions in this module. Once you have done this you can use predefined steps without having to write any glue code in Java.

The YAKS framework provides the following modules with predefined steps:

Table 1. YAKS modules

Module	Description
yaks-standard	Standard steps such as test variables, sleep/delay, log/print, ...
yaks-http	Http steps for client and server side communication
yaks-openapi	Load Open API specifications and invoke/verify operations with generated test data
yaks-kubernetes	Manage Kubernetes resources (e.g. pods, deployments, custom resources)
yaks-knative	Steps to connect with Knative eventing and messaging
yaks-jms	Send/receive steps via JMS queues/topics
yaks-kafka	Steps to publish/subscribe on Kafka messaging
yaks-jdbc	Steps to connect to relational databases
yaks-camel	Steps to access Apache Camel components and Camel routes
yaks-camel-k	Manage Camel-K resources on Kubernetes
yaks-selenium	Run UI tests with Selenium using Selenium grid or standalone containers
yaks-groovy	Leverage Groovy scripts as Citrus endpoint and component configuration

Once again it should be said that the step definitions included in this modules can be used out-of-the-box. You can start to write feature stories in Gherkin syntax that trigger the predefined steps.

4.6. Main CLI

Citrus provides a main Java class that you can run from command line. The Citrus Main CLI class supports some command line arguments that you can use to specify which tests to run.

Please also have a look into Citrus JBang CLI support as it makes running Citrus tests from a command line even more comfortable as it also takes care of setting up Java and the Citrus Maven project for free. With JBang running Citrus tests becomes as easy as running a single file (`.java`, `.xml`, `.groovy`, `.yaml`) from the command line.

The Main CLI support is shipped in a separate Maven module. You need to include the module as a dependency in your project.

Main CLI module dependency

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-main</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

You can call the Citrus Main CLI and provide some arguments.

Run Citrus main

```
CitrusApp.main(new String[] { "--help" });
```

This calls the Citrus Main CLI and prints the help message, so you can see which command line arguments you can choose from.

```
Citrus application option usage:
  -h or --help = Displays cli option usage
  -d or --duration = Maximum time in milliseconds the server should be up and running
- server will terminate automatically when time exceeds
  -c or --config = Custom configuration class
  -s or --skipTests = Skip test execution
  -p or --package = Test package to execute
  -D or --properties = Default system properties to set
  --exit = Force system exit when finished
  -e or --engine = Set test engine name used to run the tests
  -t or --test = Test class/method to execute
  -j or --jar = External test jar to load tests from
```

The `--engine` argument specifies which test framework to use when running the tests. Citrus supports these test engines as an argument value:

- JUnit Jupiter (`junit5` or `junit-jupiter`)
- Cucumber (`cucumber`)
- TestNG (`testng`)
- JUnit 4 (`junit`)

You may now use `--package`, `--test` and `--jar` to give a Java package name or test class name to run. With `--config` you can give a fully qualified class name that points to a Citrus endpoint configuration class. The configuration class is automatically loaded when the Main CLI is running.

4.7. JBang

You can easily create and run Citrus tests with JBang. The JBang tool support in Citrus is described in more detail in reference guide

Running Citrus via JBang does not require any project setup which is a fantastic match for fast prototyping of integration tests. The JBang command will automatically set up everything you need to run the Citrus test. This means you can run your test case sources directly from your command line.

To initialize a test you can run the script `citrus@citrusframework/citrus` with the `init` command as follows:

Initialize my-test.yaml

```
jbang citrus@citrusframework/citrus init my-test.yaml
```

The command above uses the JBang catalog `citrus@citrusframework/citrus` located on the Citrus GitHub repository. JBang will automatically resolve all dependencies and execute the command line script tool. This initializes the Citrus test file. You will find the created test source file in the current directory.

my-test.yaml

```
name: my-test
author: Citrus
status: FINAL
description: Sample test in YAML
variables:
  - name: message
    value: Citrus rocks!
actions:
  - echo:
      message: "${message}"
```

The JBang script is able to initialize any supported Citrus test domain specific language `.java`, `.xml`, `.yaml`, `.groovy` or `.feature`.

You can now run this test source file without any prior project setup using Citrus JBang:

Run my-test.yaml

```
jbang citrus@citrusframework/citrus run my-test.yaml
```

The command output will be like this:

Output

```
INFO 53887 --- [           main] citrusframework.testng.TestNGEngine : Running test
source my-test.yaml
INFO 53887 --- [           main] org.testng.internal.Utils           : [TestNG]
Running:
INFO 53887 --- [           main] rusframework.report.LoggingReporter :        .__  __
INFO 53887 --- [           main] rusframework.report.LoggingReporter :   ____ |__|/
```

```
|_______ __ __  ______
INFO 53887 --- [           main] rusframework.report.LoggingReporter : _/ ___\|  \
__\_  __ \  |  \/  ___/
INFO 53887 --- [           main] rusframework.report.LoggingReporter : \  \___|  ||  |
|  | \/  |  /\___ \
INFO 53887 --- [           main] rusframework.report.LoggingReporter :  \___  >__||__|
|__|  |____//____  >
INFO 53887 --- [           main] rusframework.report.LoggingReporter :      \/
\/
INFO 53887 --- [           main] rusframework.report.LoggingReporter :
INFO 53887 --- [           main] rusframework.report.LoggingReporter : C I T R U S  T
E S T S  ${citrus.version}
INFO 53887 --- [           main] rusframework.report.LoggingReporter :
INFO 53887 --- [           main] rusframework.report.LoggingReporter :
----------------------------------------------------------------------
INFO 53887 --- [           main] .citrusframework.actions.EchoAction : Citrus rocks!
INFO 53887 --- [           main] rusframework.report.LoggingReporter :
INFO 53887 --- [           main] rusframework.report.LoggingReporter : TEST SUCCESS
my-test (org.citrusframework)
INFO 53887 --- [           main] rusframework.report.LoggingReporter :
----------------------------------------------------------------------
INFO 53887 --- [           main] rusframework.report.LoggingReporter :
INFO 53887 --- [           main] rusframework.report.LoggingReporter :
INFO 53887 --- [           main] rusframework.report.LoggingReporter : CITRUS TEST
RESULTS
INFO 53887 --- [           main] rusframework.report.LoggingReporter :
INFO 53887 --- [           main] rusframework.report.LoggingReporter : SUCCESS (     3
ms) my-test
INFO 53887 --- [           main] rusframework.report.LoggingReporter :
INFO 53887 --- [           main] rusframework.report.LoggingReporter : TOTAL:       1
INFO 53887 --- [           main] rusframework.report.LoggingReporter : SUCCESS: 1
(100.0%)
INFO 53887 --- [           main] rusframework.report.LoggingReporter : FAILED:      0
(0.0%)
INFO 53887 --- [           main] rusframework.report.LoggingReporter : PERFORMANCE: 0
ms
INFO 53887 --- [           main] rusframework.report.LoggingReporter :
INFO 53887 --- [           main] rusframework.report.LoggingReporter :
----------------------------------------------------------------------

===================================================
Default Suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
===================================================
```

4.7.1. Install Citrus JBang app

For a more convenient command line usage you can install Citrus as a JBang app.

Install Citrus as JBang app

```
jbang trust add https://github.com/citrusframework/citrus/
jbang app install citrus@citrusframework/citrus
```

Now you can just call `citrus` and create and run tests with Citrus JBang.

Run my-test.yaml

```
citrus run my-test.yaml
```

4.7.2. Run tests

You can directly run test sources with Citrus JBang. This includes test sources written in Java (`.java`), XML (`.xml`), YAML (`.yaml`), Groovy (`.groovy`) or as a Cucumber Gherkin feature file (`.feature`).

Java test sources

Initialize MyTest.java

```
citrus init MyTest.java
```

MyTest.java

```
import org.citrusframework.TestCaseRunner;
import org.citrusframework.annotations.CitrusResource;

import static
org.citrusframework.actions.CreateVariablesAction.Builder.createVariables;
import static org.citrusframework.actions.EchoAction.Builder.echo;

public class MyTest implements Runnable {

    @CitrusResource
    TestCaseRunner t;

    @Override
    public void run() {
        t.given(
            createVariables().variable("message", "Citrus rocks!")
        );

        t.then(
            echo().message("${message}")
        );
    }
}
```

Run MyTest.java

```
citrus run MyTest.java
```

XML test sources

Initialize my-test.xml

```
citrus init my-test.xml
```

my-test.xml

```
<test name="EchoTest" author="Christoph" status="FINAL"
xmlns="http://citrusframework.org/schema/xml/testcase"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://citrusframework.org/schema/xml/testcase
http://citrusframework.org/schema/xml/testcase/citrus-testcase.xsd">
  <description>Sample test in XML</description>
  <variables>
    <variable name="message" value="Citrus rocks!"/>
  </variables>
  <actions>
    <echo message="${message}"/>
  </actions>
</test>
```

Run my-test.xml

```
citrus run my-test.xml
```

YAML test sources

Initialize my-test.yaml

```
citrus init my-test.yaml
```

my-test.yaml

```
name: EchoTest
description: "Sample test in YAML"
variables:
  - name: "message"
    value: "Citrus rocks!"
actions:
  - echo:
      message: "${message}"
```

Run my-test.yaml

```
citrus run my-test.yaml
```

Groovy test sources

Initialize my-test.groovy

```
citrus init my-test.groovy
```

my-test.groovy

```
import static org.citrusframework.actions.EchoAction.Builder.echo

name "EchoTest"
description "Sample test in Groovy"

variables {
    message="Citrus rocks!"
}

actions {
    $(echo().message('${message}'))
}
```

Run my-test.groovy

```
citrus run my-test.groovy
```

Cucumber feature sources

Initialize my-test.feature

```
citrus init my-test.feature
```

my-test.feature

```
Feature: EchoTest

  Background:
    Given variables
    | message | Citrus rocks! |

  Scenario: Print message
    Then print '${message}'
```

Run my-test.feature

```
jbang --deps org.citrusframework.yaks:yaks-standard:0.20.0 citrus run my-test.feature
```

> Many of the predefined Cucumber steps (e.g. `Then print '<message>'`) in Citrus are provided in a separate Citrus child project called YAKS. You need to add additional project dependencies for that steps to be loaded as part of the JBang script. The `--deps` option adds dependencies using Maven artifact coordinates. You may add the additional modules to the `jbang.properties` as described in the next section.

4.7.3. Additional JBang dependencies

Citrus JBang comes with a set of default dependencies that makes the scripts run as tests.

The default modules that you can use in Citrus JBang are:

- org.citrusframework:citrus-base
- org.citrusframework:citrus-jbang-connector
- org.citrusframework:citrus-groovy
- org.citrusframework:citrus-xml
- org.citrusframework:citrus-yaml
- org.citrusframework:citrus-http
- org.citrusframework:citrus-validation-json
- org.citrusframework:citrus-validation-yaml
- org.citrusframework:citrus-validation-xml

This enables you to run Java, YAML, XML, Groovy tests out of the box. In case your tests uses an additional feature from the Citrus project you may need to add the module so JBang can load the dependency at startup.

The easiest way to do this is to create a `jbang.properties` file that defines the additional dependencies:

jbang.properties

```
# Declare required additional dependencies
run.deps=org.citrusframework:citrus-camel:${citrus.version},\
org.citrusframework:citrus-testcontainers:${citrus.version},\
org.citrusframework:citrus-kafka:${citrus.version}
```

The file above adds the modules `citrus-camel`, `citrus-testcontainers` and `citrus-kafka` so you can use them in your JBang Citrus test source.

The `jbang.properties` file may be located right next to the test source file or in your user home directory for global settings.

In case you want to run Cucumber BDD Gherkin feature files and use the predefined steps included in the YAKS project, you need to add this YAKS runtime dependency accordingly: `org.citrusframework.yaks:yaks-standard:0.20.0`

4.7.4. Run from clipboard

You can run tests from your current clipboard. Just use the file name `clipboard.xxx` where the file extension defines the type of the test source (`.java`, `.yaml`, `.xml`, `.groovy`, `.feature`).

Run YAML test from Clipboard

```
citrus run clipboard.yaml
```

4.7.5. List tests

The `ls` command lists all running Citrus tests. These tests may be started

List running tests

```
citrus ls
```

Command output

```
PID   NAME         STATUS  AGE
19201 my-test.yaml Running 20s
```

4.8. Sharding Test Cases

Sharding is a powerful option within the Citrus framework designed to facilitate the loading and execution of test cases in a distributed manner. Sharding test execution is expected to save roughly `[TIME] * 1 / 2` of execution time per runner.

This chapter provides an overview of the available functionality and guides you through the process of configuring and using test loaders with sharding.

4.8.1. Configuration

First of all, you need to wrap your test case loading with the `org.citrusframework.sharding.Shard#createShard(Stream<T> items)` method. Have a look at the previous runtime chapters to get to know how to do this with the test framework of your choice.

Once you're done doing so, the sharding can be configured using environment variables or system properties. The following two key configuration parameters are required for sharded test loading:

- **Total number of shards**: Defines how many shards the test cases should be divided into.
- **Shard number**: Specifies the specific shard number that this loader instance will handle.

The following table summarizes the full configuration options:

Configuration	Environment Variable	System Property
Total number of shards	`CITRUS_SHARDING_TOTAL`	`citrus.sharding.total`
Shard number	`CITRUS_SHARDING_NUMBER`	`citrus.sharding.number`
Shard seed	`CITRUS_SHARDING_SEED`	`citrus.sharding.seed`

When writing test cases in Java, additional constructors exist, such that the total number of shards, as well as the current shard number can be programmatically configured.

Chapter 5. Run tests

You can write Citrus tests in various programming languages. The following chapters describe the individual domain specific languages that support you in writing tests in Citrus.

> We do our best to make all features equal in all supported languages. Some feature may only make sense in a specific test project setup and are not applicable to all languages (e.g. test behaviors only run in Java classes).

5.1. Run Java tests

You can code the Citrus test as you would do with any other unit test framework (e.g. JUnit Jupiter or TestNG). In fact Citrus builds on top of those famous unit testing frameworks because the tooling and adoption throughout the industry is very high for these libraries.

Citrus has a set of test engines that you can use to run Citrus tests with your favorite unit test framework. For instance Citrus provides a JUnit Jupiter extension that you can use to code Citrus tests in Java. Or you can use the Citrus Cucumber integration to run your tests as Gherkin feature files.

No matter what test engine you choose the Citrus test will assist you in writing tests through a Java domain specific language. This is a set of predefined methods, builders and classes that you can use to code the test in Java. In conclusion the Citrus test holds a sequence of test actions and each action performs a very special step in your test such as sending or receiving a message.

Despite the fact that message exchange is one of the main actions in an integration test framework for message-based applications Citrus is more than just that. Each test case in Citrus is able to perform various actions such as connecting to the database, transforming data, adding iterations and conditional steps.

With the default Citrus actions provided out of the box users can accomplish very complex use cases in integration testing. As a tester you can configure test actions in Java using a test action runner API that leverages a fluent builder pattern.

5.1.1. Test action runner

The test action runner is the entry to the fluent Citrus Java API. You can configure test actions with a builder pattern style API and immediately runs the test actions.

See the following example to see the Java domain specific language in action.

Test action runner

```
import org.testng.annotations.Test;
import org.citrusframework.annotations.CitrusTest;
import org.citrusframework.dsl.testng.TestNGCitrusSupport;

@Test
public class Service_IT extends TestNGCitrusSupport {

    @CitrusTest(name = "Service_IT")
    public void serviceTest() {
        run(echo("Before service call"));

        run(echo("After service call"));
    }
}
```

The test action runner executes each test action immediately as you use the provided `run()` method. All actions provided in Citrus represent a certain functionality (e.g. send/receive messages, delay the test, access a database). Citrus ships with a wide range of test actions, but you are also able to write your own test actions and execute them during a test.

By default, all actions run sequentially in the same order as they are defined in the test case. In case one single action fails the whole test case is failing. Of course, you can leverage parallel action execution with the usage of test containers.

> The **TestNGCitrusSupport** and **JUnit4CitrusSupport** base classes are not thread safe by default. This is simply because the base class is holding state to the current test action runner instance in order to delegate method calls to this instance. Parallel test execution is not available with this approach. Fortunately there is a way to support parallel test execution through resource injection. Read more about this in JUnit4 or TestNG support.

5.1.2. Gherkin test action runner

The test action runner is also available as Gherkin style runner with `given()`, `when()`, `then()` methods. The Gherkin test action runner follows the Behavior Driven Development concepts of structuring the test into the three parts: **Given** a certain context, **when** an event occurs, **then** an outcome should be verified.

Gherkin test action runner

```
@Test
public class Service_IT extends TestNGCitrusSupport {

    @CitrusTest(name = "Service_IT")
    public void serviceTest() {
        given(
            echo("Setup the context")
        );

        when(
            echo("Trigger the event")
        );

        then(
            echo("Verify the outcome")
        );
    }
}
```

5.1.3. Test meta information

The user is able to provide some additional information about the test case. The meta-info section at the very beginning of the test case holds information like author, status or creation date.

Test meta information

```
@CitrusTest
public void sampleTest() {
    description("This is a Test");
    author("Christoph");
    status(Status.FINAL);

    run(echo("Hello Citrus!"));
}
```

The status allows the following values:

- DRAFT
- READY_FOR_REVIEW
- DISABLED
- FINAL

This information gives the reader first impression about the test and is also used to generate test documentation. By default, Citrus is able to generate test reports in HTML and Excel in order to list all tests with their metadata information and description.

Tests with the status DISABLED will not be executed during a test suite run. So someone can just start adding planned test cases that are not finished yet in status DRAFT. In case a test is not runnable yet because it is not finished, someone may disable a test temporarily to avoid causing failures during a test run.

The test description should give a short introduction to the intended use case scenario that will be tested. The user should get a short summary of what the test case is trying to verify.

5.1.4. Finally block

Java developers might be familiar with the concept of try-catch-finally blocks. The ***finally*** section contains a list of test actions that will be executed guaranteed at the very end of the test case even if errors did occur during the execution before.

This is the right place to tidy up things that were previously created by the test like cleaning up the database for instance.

Finally block

```
@CitrusTest
public void sampleTest() {
    given(
        doFinally()
            .actions(echo("Do finally - regardless of any error before"))
    );

    echo("Hello Test Framework");
}
```

As an example imagine that you have prepared some data inside the database at the beginning of the test and you need to make sure the data is cleaned up at the end of the test case.

Finally block example

```
@CitrusTest
public void finallyBlockTest() {
    variable("orderId", "citrus:randomNumber(5)");
    variable("date", "citrus:currentDate('dd.MM.yyyy')");

    given(
        doFinally()
            .actions(sql(dataSource).statement("DELETE FROM ORDERS WHERE
ORDER_ID='${orderId}'"))
    );

    when(
        sql(dataSource).statement("INSERT INTO ORDERS VALUES (${orderId}, 1, 1,
'${date}')")
    );

    then(
        echo("ORDER creation time: citrus:currentDate('dd.MM.yyyy')")
    );
}
```

In the example the first action creates an entry in the database using an ***INSERT*** statement. To be sure that the entry in the database is deleted after the test, the `finally` section contains the respective ***DELETE*** statement that is always executed regardless the test case state (successful or failed).

> The `finally` section must be placed at the very beginning of the test. This is because the test action runner is immediately executing each test action as it is called within the Java DSL methods. This is the only way the test case can perform the final actions also in case of previous error.

A `finally` block placed at the very end of the test will not take action unless put in a traditional Java try-finally-block:

Traditional try-finally block

```
@CitrusTest
public void finallyBlockTest() {
    variable("orderId", "citrus:randomNumber(5)");
    variable("date", "citrus:currentDate('dd.MM.yyyy')");

    try {
        when(
            sql(dataSource).statement("INSERT INTO ORDERS VALUES (${orderId}, 1, 1,
'${date}')")
        );

        then(
            echo("ORDER creation time: citrus:currentDate('dd.MM.yyyy')")
        );
    } finally {
        then(
            sql(dataSource).statement("DELETE FROM ORDERS WHERE
ORDER_ID='${orderId}'")
        );
    }
}
```

Using the traditional Java `try-finally` feels more natural no doubt. Please notice that the Citrus report and logging will not account the traditional finally block actions then. Good news is whatever layout you choose the outcome is always the same.

The `finally` block is executed safely even in case some previous test action raises an error for some reason.

5.1.5. Test behaviors

The concept of test behaviors is a good way to reuse test action blocks in the Java DSL. Test behaviors combine action sequences to a logical unit. The behavior defines a set of test actions that can be applied multiple times to different test cases.

The behavior is a separate Java DSL class with a single *apply* method that configures the test actions. Test behaviors follow this basic interface:

Test behaviors

```
@FunctionalInterface
public interface TestBehavior {

    /**
     * Behavior building method.
     */
    void apply(TestActionRunner runner);

}
```

The behavior is provided with the test action runner and all actions in the behavior should run on that runner. Every time the behavior is applied to a test the actions get executed accordingly.

Test behaviors

```
public class FooBehavior implements TestBehavior {
    public void apply(TestActionRunner runner) {
        runner.run(createVariable("foo", "test"));

        runner.run(echo("fooBehavior"));
    }
}

public class BarBehavior implements TestBehavior {
    public void apply(TestActionRunner runner) {
        runner.run(createVariable("bar", "test"));

        runner.run(echo("barBehavior"));
    }
}
```

The listing above shows two test behaviors that add very specific test actions and test variables to the test case. As you can see the test behavior is able to use the same Java DSL action methods and defines test variables and actions as a normal test case would do. You can apply the behaviors multiple times in different tests:

```
@CitrusTest
public void behaviorTest() {
    run(apply(new FooBehavior()));

    run(echo("Successfully applied bar behavior"));

    run(apply(new BarBehavior()));

    run(echo("Successfully applied bar behavior"));
}
```

The behavior is applied to the test case by calling the **apply()** method. As a result the behavior is executed adding its logic at this point of the test execution. The same behavior can now be called in multiple test cases so we have a reusable set of test actions.

A behavior may use different variable names than the test and vice versa. No doubt the behavior will fail as soon as special variables with respective values are not present. Unknown variables cause the behavior and the whole test to fail with errors.

So a good approach would be to harmonize variable usage across behaviors and test cases, so that templates and test cases do use the same variable naming. The behavior automatically knows all variables in the test case and all test variables created inside the behavior are visible to the test case after applying.

> When a behavior changes variables this will automatically affect the variables in the whole test. So if you change a variable value inside a behavior and the variable is defined inside the test case the changes will affect the variable in a global test context. This means we have to be careful when executing a behavior several times in a test, especially in combination with parallel containers (see containers-parallel).

5.1.6. Run custom code

In general, you are able to mix Citrus Java DSL actions with custom Java code as you like.

Run custom code

```
import org.testng.annotations.Test;
import org.citrusframework.annotations.CitrusTest;
import org.citrusframework.dsl.testng.TestNGCitrusSupport;

@Test
public class Service_IT extends TestNGCitrusSupport {

    private MyService myService = new MyService();

    @CitrusTest(name = "Service_IT")
    public void serviceTest() {
        run(echo("Before service call"));

        myService.doSomething("Now calling custom service");

        run(echo("After service call"));
    }
}
```

The test above uses a mix of Citrus test actions and custom service calls. The test logic will execute as expected. It is recommended though to wrap custom code in a test action in order to have a consistent test reporting and failure management in Citrus.

Test action wrapper

```
import org.testng.annotations.Test;
import org.citrusframework.annotations.CitrusTest;
import org.citrusframework.dsl.testng.TestNGCitrusSupport;

@Test
public class Service_IT extends TestNGCitrusSupport {

    private MyService myService = new MyService();

    @CitrusTest(name = "Service_IT")
    public void serviceTest() {
        run(echo("Before service call"));

        run(
            action(context -> {
                myService.doSomething("Now calling custom service");
            })
        );

        run(echo("After service call"));
    }
}
```

The sample above wraps the call to the custom service `myService` in an abstract test action represented as Java lambda expression. This way the service call becomes part of the Citrus test execution and failures are reported properly. Also you have access to the current test context which holds the list of test variables as well as many other Citrus related test objects (e.g. message store).

This is why you should wrap custom code in a test action and run that code via the test action runner methods. You can also put your custom code in a test action implementation and reference the logic from multiple tests.

5.1.7. Bind objects to registry

The Citrus context is a place where objects can register themselves in order to enable dependency injection and instance sharing in multiple tests. Once you register the object in the context others can resolve the reference with its given name.

In a simple example the context can register a new endpoint that is injected in several tests.

You can access the Citrus context within the provided before/after methods on the test.

Register endpoint in Citrus context

```
public class CitrusRegisterEndpoint_IT extends TestNGCitrusSupport {

    @Override
    public void beforeSuite(CitrusContext context) {
        context.bind("foo", new FooEndpoint());
    }
}
```

With the CitrusContext you can bind objects to the registry. Each binding receives a name so others can resolve the instance reference for injection.

Inject endpoint in other tests

```
public class InjectEndpoint_IT extends TestNGCitrusSupport {

    @CitrusEndpoint
    private FooEndpoint foo;

    @Test
    @CitrusTest
    public void injectEndpointTest() {
        $(send(foo)
                .message()
                .body("Hello foo!"));

        $(receive(foo)
                .message()
                .body("Hello Citrus!"));
    }
}
```

The `@CitrusEndpoint` annotation injects the endpoint resolving the instance with the given name `foo`. Test methods can use this endpoint in the following send and receive actions.

@BindToRegistry

An alternative to using the `bind()` method on the CitrusContext is to use the `@BindToRegistry` annotation. Methods and fields annotated will automatically register in the CitrusContext registry.

@BindToRegistry annotation

```
public class CitrusRegisterEndpoint_IT extends TestNGCitrusSupport {

    @CitrusFramework
    private Citrus citrus;

    @BindToRegistry(name = "fooQueue")
    private MessageQueue queue = new DefaultMessageQueue("fooQueue");

    @BindToRegistry
    public FooEndpoint foo() {
        return new FooEndpoint();
    }
}
```

The annotation is able to specify an explicit binding name. The annotation works with public methods and fields in tests.

Configuration classes

As an alternative to adding the registry binding configuration directly to the test you can load configuration classes.

Configuration classes are automatically loaded before a test suite run and all methods and fields are parsed for potential bindings. You can use the environment settings `citrus.java.config` and/or `CITRUS_JAVA_CONFIG` to set a default configuration class.

citrus-application.properties

```
citrus.java.config=MyConfig.class
```

MyConfig.class

```
public class MyConfig {

    @BindToRegistry(name = "fooQueue")
    private MessageQueue queue = new DefaultMessageQueue("fooQueue");

    @BindToRegistry
    public FooEndpoint foo() {
        return new FooEndpoint();
    }
}
```

@CitrusConfiguration

Each test is able to use the `@CitrusConfiguration` annotation to add registry bindings, too.

@CitrusConfiguration annotation

```
@CitrusConfiguration(classes = MyConfig.class)
public class CitrusRegisterEndpoint_IT extends TestNGCitrusSupport {

    @CitrusEndpoint
    private FooEndpoint foo;

    @Test
    @CitrusTest
    public void injectEndpointTest() {
        $(send(foo)
                .message()
                .body("Hello foo!"));

        $(receive(foo)
                .message()
                .body("Hello Citrus!"));
    }
}
```

The `@CitrusConfiguration` annotation is able to load configuration classes and bind all components to the registry for later usage. The test can inject endpoints and other components using the `@CitrusEndpoint` and `@CitrusResource` annotation on fields.

5.1.8. Resource injection

Resource injection is a convenient mechanism to access Citrus internal objects such as TestRunner or TestContext instances. The following sections deal with resource injection of different objects.

Inject Citrus framework

You can access the Citrus framework instance in order to access all components and functionalities. Just use the `@CitrusFramework` annotation in your test class.

Citrus framework injection

```
public class CitrusInjection_IT extends TestNGCitrusSupport {

    @CitrusFramework
    private Citrus citrus;

    @Test
    @CitrusTest
    public void injectCitrusTest() {
        citrus.getCitrusContext().getMessageListeners().addMessageListener(new
MyListener());
    }
}
```

The framework instance provides access to the Citrus context which is a central registry for all components. The example above adds a new message listener.

> The Citrus context is a shared component. Components added will perform with all further tests and changes made affect all tests.

Test action runner injection

The test action runner is the entry to the fluent Java API. You can inject the runner as a method parameter.

Test action runner injection

```
public class RunnerInjection_IT extends JUnit4CitrusSupport {

    @Test
    @CitrusTest
    public void injectResources(@CitrusResource TestCaseRunner runner) {

        runner.given(
            createVariable("random", "citrus:randomNumber(10)")
        );

        runner.run(
            echo("The random number is: ${random}")
        );
    }
}
```

The parameter requires the `@CitrusResource` annotations in order to mark the parameter for Citrus resource injection.

Now each method uses its own runner instance which makes sure that parallel test execution can take place without having the risk of side effects on other tests running at the same time.

Test context injection

The Citrus test context combines a set of central objects and functionalities that a test is able to make use of. The test context holds all variables and is able to resolve functions and validation matchers.

In general a tester will not have to explicitly access the test context because the framework is working with it behind the scenes. In terms of advanced operations and customizations accessing the test context may be a good idea though.

Each test action implementation has access to the test context as it is provided to the execution method in the interface:

Test action interface

```
@FunctionalInterface
public interface TestAction {
    /**
     * Main execution method doing all work
     * @param context
     */
    void execute(TestContext context);
}
```

In addition Citrus provides a resource injection mechanism that allows to access the current test context in a test class or test method.

Inject as method parameter

```
public class TestContextInjection_IT extends JUnit4CitrusSupport {

    @Test
    @CitrusTest
    public void resourceInjectionIT(@CitrusResource TestContext context) {
        context.setVariable("myVariable", "some value");

        run(echo("${myVariable}"));
    }
}
```

As you can see the test method defines a parameter of type **org.citrusframework.context.TestContext**. The annotation **@CitrusResource** tells Citrus to inject this parameter with the according instance of the context for this test.

Now you have access to the context and all its capabilities such as variable management. As an alternative you can inject the test context as a class member variable.

Inject as member

```
public class TestContextInjection_IT extends JUnit4CitrusSupport {

    @CitrusResource
    private TestContext context;

    @Test
    @CitrusTest
    public void resourceInjectionIT() {
        context.setVariable("myVariable", "some value");

        run(echo("${myVariable}"));
    }
}
```

Endpoint injection

Endpoints play a significant role when sending/receiving messages over various transports. An endpoint defines how to connect to a message transport (e.g. Http endpoint URL, JMS message broker connection, Kafka connection and topic selection).

Endpoints can live inside the Citrus context (e.g. in Spring application context) or you can inject the endpoint into the test class with given configuration.

Endpoint injection

```
public class EndpointInjectionJavaIT extends TestNGCitrusSpringSupport {

    @CitrusEndpoint
    @DirectEndpointConfig(queueName = "FOO.test.queue")
    private Endpoint directEndpoint;

    @Test
    @CitrusTest
    public void injectEndpoint() {
        run(send(directEndpoint)
                .message()
                .type(MessageType.PLAINTEXT)
                .body("Hello!"));

        run(receive(directEndpoint)
                .message()
                    .type(MessageType.PLAINTEXT)
                    .body("Hello!"));
    }
}
```

The sample above creates a new endpoint as a direct in-memory channel endpoint. Citrus reads the `@CitrusEndpoint` annotation and adds the configuration as given in the `@DirectEndpointConfig` annotation. This way you can create and inject endpoints directly to your test.

Citrus also supports the Spring framework as a central bean registry. You can add endpoints as Spring beans and use the `@Autowired` annotation to inject the endpoint in your test.

5.2. Test source loader

Besides writing Citrus tests directly in Java you can code tests in domain specific languages using one of the supported tests sources:

- Java runnable
- XML
- YAML

- Cucumber
- Groovy
- Spring XML

The test case definition file is loaded by Citrus and run as a normal test. In Citrus the `@CitrusTestSource` annotation in the Java class defines the test source location to be loaded. This annotation makes Citrus search for the XML, YAML, Groovy, Cucumber, Spring XML files that represent the Citrus test.

The next example shows how the annotation is used in a normal Java test to load XML test case definitions.

MyFirstCitrus_IT.java

```
package org.citrusframework;

import org.citrusframework.annotations.CitrusTestSource;
import org.citrusframework.testng.TestNGCitrusSupport;
import org.testng.annotations.Test;

public class MyFirstCitrus_IT extends TestNGCitrusSpringSupport {

    @Test
    @CitrusTestSource(type = "xml", name="MyFirstCitrus_IT")
    public void myFirstCitrus_IT() {
    }
}
```

In the basic example above this means that Citrus searches for an XML test file in **org/citrusframework/MyFirstCitrus_IT.xml** .

You can customize this path and tell Citrus to search for another XML file by using the `@CitrusTestSource` annotation properties.

Following annotation properties are available:

type	Type of the test source to load (spring, groovy, xml, ...)
name	List of test case names to execute. Names also define XML file names to look for (**.xml** file extension is not needed here).
packageName	Custom package location for the XML files to load

packageScan List of packages that are automatically scanned for XML test files to execute. For each XML file found separate test is executed. Note that this performs a Java Classpath package scan so all XML files in package are assumed to be valid Citrus XML test cases. In order to minimize the amount of accidentally loaded XML files the scan will only load XML files with `**/*Test.xml` and `**/*IT.xml` file name pattern.

Customize @CitrusTestSource

```
public class Sample_IT extends TestNGCitrusSpringSupport {

    @Test
    @CitrusTestSource(type = "xml", name = "CustomName_IT", packageName =
"com.other.test.package")
    public void customXmlTest() {}
}
```

The annotation above loads a different XML test file named `CustomName_IT` in package `com.other.test.package`.

You can also load multiple XML files and run each of them.

Load multiple XML files

```
public class Sample_IT extends TestNGCitrusSpringSupport {

    @Test
    @CitrusTestSource(type = "xml", name = { "Test_1", "Test_2" })
    public void multipleTests() {}
}
```

This tells Citrus to search for the files `Test_1.xml` and `Test_2.xml`. Citrus loads the files and runs each of them as a separate test. You can also load all test in a package with a `packageScan`.

Load multiple XML files

```
public class Sample_IT extends TestNGCitrusSpringSupport {

    @Test
    @CitrusTestSource(type = "xml", packageScan =  { "com.some.test.package",
"com.other.test.package" })
    public void packageScanTest() {}
}
```

This loads all XML files in the given packages and executes each of them as a separate test.

You can also mix the various `@CitrusTestSource` annotations in a single Java class. The class can have several methods with different annotations. Each annotated method represents one or more

Citrus XML test cases.

@CitrusTestSource annotations

```
public class SampleIT extends TestNGCitrusSpringSupport {

    @Test
    @CitrusTestSource(type = "xml", name = "SampleIT")
    public void sampleTest() {}

    @Test
    @CitrusTestSource(type = "xml", name = { "Test_1", "Test_2" })
    public void multipleTests() {}

    @Test
    @CitrusTestSource(type = "xml", name = "CustomName_IT", packageName =
"com.other.test.package")
    public void customXmlTest() {}

    @Test
    @CitrusTestSource(type = "xml", packageScan =  { "com.some.test.package",
"com.other.test.package" })
    public void packageScanTest() {}
}
```

You are free to combine these test annotations as you like in your class. Each XML test loaded as part of the class will be reported separately as a unit test. So the test reports will have the exact number of tests executed with proper success and failed stats. You can use the reports as normal unit test reports, for instance in a continuous build.

When test execution takes place each test method annotation is evaluated in sequence. XML test cases that match several times, for instance by explicit name reference and a package scan will be executed several times respectively.

The best thing about using the **@CitrusTestSource** annotation is that you can continue to use the test framework capabilities (e.g. test groups, invocation count, thread pools, data providers, and so on).

The following chapters describe the different supported test sources that you can load as a Citrus test.

5.3. XML

As an alternative to coding tests in a programming language you can specify the test logic in XML only, too. The XML language gives you some XSD schema and a set of elements that you can choose from to specify test actions.

As usual, you need to choose one of the supported test engines in Citrus to actually run the test (e.g.

JUnit Jupiter or TestNG). You can then use the `@CitrusTestSource` annotation to load the XML file that represents the test case.

MyFirstXmlIT.java

```
package org.citrusframework;

import org.citrusframework.annotations.CitrusTestSource;
import org.citrusframework.testng.TestNGCitrusSupport;
import org.testng.annotations.Test;

public class MyFirstXmlIT extends TestNGCitrusSpringSupport {

    @Test
    @CitrusTestSource(type = "xml", name="my-test.xml")
    public void myFirstXmlIT() {
    }
}
```

The test logic itself is defined in the XML file like this:

my-test.xml

```
<test name="EchoTest" author="Christoph" status="FINAL"
      xmlns="http://citrusframework.org/schema/xml/testcase"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://citrusframework.org/schema/xml/testcase
      http://citrusframework.org/schema/xml/testcase/citrus-testcase.xsd">
  <description>Sample test in XML</description>

  <variables>
    <variable name="text" value="Hello from Citrus!"/>
  </variables>

  <actions>
    <echo message="${text}"/>
  </actions>
</test>
```

5.3.1. Configuration scripts

The XML test is able to create endpoints and other configuration entities as part of the test. The test leverages the Groovy programming language that provides a comfortable way to define beans in the Citrus registry. You can set any of the supported properties on the configuration components.

my-test.xml

```
<test name="EchoTest" author="Christoph" status="FINAL"
      xmlns="http://citrusframework.org/schema/xml/testcase"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://citrusframework.org/schema/xml/testcase
      http://citrusframework.org/schema/xml/testcase/citrus-testcase.xsd">
  <description>Sample test in XML</description>

  <variables>
    <variable name="text" value="Hello from Citrus!"/>
  </variables>

  <actions>
    <groovy>
      <endpoints>
        <script>
        configuration {
            queues {
                queue('say-hello')
            }

            endpoints {
                direct('hello')
                    .asynchronous()
                    .queue('say-hello')
            }
        }
        </script>
      </endpoints>
    </groovy>

    <send endpoint="hello">
      <message>
        <body>
          <data>Hello Citrus!</data>
        </body>
      </message>
    </send>
  </actions>
</test>
```

The test above creates a new `direct` endpoint called `hello` that uses an im-memory queue `say-hello`. The send action is able to reference the endpoint by its name to send a message body to the queue.

5.3.2. Templates

Templates group action sequences to a logical unit. You can think of templates as reusable components that are used in several tests. The maintenance is much more efficient because you need to apply changes only on the templates and all referenced use cases are updated

automatically.

The template gets identified by a unique name. Inside a test case we apply the template referencing this unique name. Have a look at a first example:

echo-template.xml

```
<template name="echo">
  <description>Sample template in XML</description>
  <parameters>
    <parameter name="message" value="Citrus rocks!"/>
  </parameters>

  <actions>
    <echo message="${message}"/>
  </actions>
</template>
```

The code example above describes a template called `echo`. Templates hold a sequence of test actions or calls other templates themselves. The template may define a set of parameters that represent customizable user input when applying the template.

The template is available for multiple tests that may apply the template as part of their test action sequence.

Apply templates

```
<test name="ApplyTemplateTest" author="Christoph" status="FINAL"
      xmlns="http://citrusframework.org/schema/xml/testcase"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://citrusframework.org/schema/xml/testcase
      http://citrusframework.org/schema/xml/testcase/citrus-testcase.xsd">
  <description>Sample test in XML</description>

  <actions>
    <apply-template name="echo"/>

    <apply-template name="echo">
      <parameters>
        <parameter name="message" value="Hello from Citrus!"/>
      </parameters>
    </apply-template>
  </actions>
</test>
```

The example references the `echo` template and call it as part of the test with `<apply-template/>`. You can customize the template parameter to overwrite the default variable `text` with a custom value.

The example above assumes that you have added the template to the Citrus bean registry with the name `echo`. If this is not the case you may also reference the file resource path when applying the

template:

Apply templates

```
<test name="ApplyTemplateTest" author="Christoph" status="FINAL"
      xmlns="http://citrusframework.org/schema/xml/testcase"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://citrusframework.org/schema/xml/testcase
      http://citrusframework.org/schema/xml/testcase/citrus-testcase.xsd">
  <description>Sample test in XML</description>

  <actions>
    <apply-template file="classpath:templates/xml/echo-template.xml">
      <parameters>
        <parameter name="message" value="Hello from Citrus!"/>
      </parameters>
    </apply-template>
  </actions>
</test>
```

Template parameters

How to handle test variables when dealing with templates, in particular when templates are defined in separate source files? A template may use different variable names compared to those names used a test and vice versa. The template fails immediately when respective variables are not present. Therefore, templates introduce the concept of parameters to declare its variable names.

Assume you have the following template:

sayHello template

```
<template name="sayHello">
  <description>Sample template in XML</description>
  <parameters>
    <parameter name="user" value="Citrus"/>
  </parameters>

  <actions>
    <echo message="Hello ${user}!"/>
  </actions>
</template>
```

The template `sayHello` in the example uses the variable `${user}`.

In case the test case already uses the template parameter as a value you do not need to set this variable explicitly on the apply template action.

Implicit template parameter

```
<test name="SayHelloTemplateTest" author="Christoph" status="FINAL"
      xmlns="http://citrusframework.org/schema/xml/testcase"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://citrusframework.org/schema/xml/testcase
      http://citrusframework.org/schema/xml/testcase/citrus-testcase.xsd">
  <description>Sample test in XML</description>

  <variables>
    <variable name="user" value="Christoph"/>
  </variables>

  <actions>
    <apply-template name="sayHello"/>
  </actions>
</test>
```

Output

```
Hello Christoph!
```

The implicit template parameter may break as soon as the template or the test changes its parameter or variable name.

The test case may also explicitly set the template parameter when applying the template:

Explicit template parameter

```
<test name="SayHelloTemplateTest" author="Christoph" status="FINAL"
      xmlns="http://citrusframework.org/schema/xml/testcase"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://citrusframework.org/schema/xml/testcase
      http://citrusframework.org/schema/xml/testcase/citrus-testcase.xsd">
  <description>Sample test in XML</description>

  <variables>
    <variable name="username" value="Mickey"/>
  </variables>

  <actions>
    <apply-template name="sayHello">
      <parameters>
        <parameter name="user" value="${username}"/>
      </parameters>
    </apply-template>
  </actions>
</test>
```

Output

```
Hello Mickey!
```

The variable **username** is already present in the test case and gets translated into the **user** parameter. Following from that the template works fine although test and template do work on different variable names.

With template parameters you are able to solve the calibration effort when working with templates and variables. It is always a good idea to check the used variables/parameters inside a template when calling it. There might be a variable that is not declared yet inside your test. So you need to define this value as a parameter.

Template parameter scope

A template is able to change a test variable value for the rest of the test case. It is important to know about this when changing test variables in a template as it affects the descendant test actions. Especially when running test actions or templates in parallel to each other this might lead to unexpected behavior. You may choose the template parameter scope for this scenario.

> When a template works on variable values and parameters changes to these variables will automatically affect the variables in the whole test. So if you change a variable's value inside a template and the variable is defined inside the test case the changes will affect the variable in a global context. We have to be careful with this when executing a template several times in a test, especially in combination with parallel containers (see containers-parallel).

print template

```
<template name="print">
  <description>Sample template in XML</description>
  <parameters>
    <parameter name="index" value="0"/>
    <parameter name="area" value="Atlantis"/>
  </parameters>

  <actions>
    <echo message="#${index} Hello ${area}!"/>
  </actions>
</template>
```

Global scope parameter (default)

```
<test name="SayHelloTemplateTest" author="Christoph" status="FINAL"
      xmlns="http://citrusframework.org/schema/xml/testcase"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://citrusframework.org/schema/xml/testcase
      http://citrusframework.org/schema/xml/testcase/citrus-testcase.xsd">
  <description>Sample test in XML</description>

  <actions>
    <parallel>
      <actions>
        <apply-template name="print">
          <parameters>
            <parameter name="index" value="1"/>
            <parameter name="area" value="Europe"/>
          </parameters>
        </apply-template>
        <apply-template name="print">
          <parameters>
            <parameter name="index" value="2"/>
            <parameter name="area" value="Asia"/>
          </parameters>
        </apply-template>
        <apply-template name="print">
          <parameters>
            <parameter name="index" value="3"/>
            <parameter name="area" value="Africa"/>
          </parameters>
        </apply-template>
      </actions>
    </parallel>
  </actions>
</test>
```

In the listing above a template **print** is called several times in a parallel container. The parameter values will be handled in a global context, so it is quite likely to happen that the template instances influence each other during execution. We might get such print messages:

Output

```
#2 Hello Europe!
#2 Hello Africa!
#3 Hello Africa!
```

Index parameters do not fit and the message **'Hello Asia'** is completely gone. This is because templates overwrite parameters to each other as they are executed in parallel at the same time. To avoid this behavior we need to tell the template that it should handle parameters as well as variables in a local context. This will enforce that each template instance is working on a dedicated

local context. See the **global-context** attribute that is set to **false** in this example:

print template

```
<template name="print" global-context="false">
  <description>Sample template in XML</description>
  <parameters>
    <parameter name="index" value="0"/>
    <parameter name="area" value="Atlantis"/>
  </parameters>

  <actions>
    <echo message="#${index} Hello ${area}!"/>
  </actions>
</template>
```

After that template instances will not influence each other anymore. But notice that variable changes inside the template then do not affect the test case neither.

5.4. YAML

As an alternative to coding tests in a programming language you can specify the test logic in YAML only, too. The YAML language gives you some XSD schema and a set of elements that you can choose from to specify test actions.

As usual, you need to choose one of the supported test engines in Citrus to actually run the test (e.g. JUnit Jupiter or TestNG). You can then use the `@CitrusTestSource` annotation to load the YAML file that represents the test case.

MyFirstYamlIT.java

```
package org.citrusframework;

import org.citrusframework.annotations.CitrusTestSource;
import org.citrusframework.testng.TestNGCitrusSupport;
import org.testng.annotations.Test;

public class MyFirstYamlIT extends TestNGCitrusSpringSupport {

    @Test
    @CitrusTestSource(type = "yaml", name="my-test.yaml")
    public void myFirstXmlIT() {
    }
}
```

The test logic itself is defined in the YAML file like this:

my-test.yaml

```
name: EchoTest
author: Christoph
status: FINAL
description: "Sample test in YAML"
variables:
  - name: "message"
    value: "Hello from Citrus!"
actions:
  - echo:
      message: "${message}"
```

5.4.1. Configuration scripts

The YAML test is able to create endpoints and other configuration entities as part of the test. The test leverages the Groovy programming language that provides a comfortable way to define beans in the Citrus registry.

The script creates the instances as beans in the Citrus registry.

A configuration script may use methods and closures to define the bean instances. The configuration script supports following methods:

- queues()
- endpoints()
- beans()

my-test.yaml

```
name: EchoTest
author: Christoph
status: FINAL
description: "Sample test in YAML"
variables:
  - name: "message"
    value: "Hello from Citrus!"
actions:
  - groovy:
      endpoints:
        script: |
          configuration {
              queues {
                  queue('say-hello')
              }

              endpoints {
                  direct('hello')
                      .asynchronous()
                      .queue('say-hello')
              }
          }
  - send:
      endpoint: "hello"
      message>:
        body:
          data: Hello Citrus!
```

The test above creates a new `direct` endpoint called `hello` that uses an in-memory queue named `say-hello`.

The `send` test action is able to reference the new endpoint by its name to send a message body to the queue.

You can also create bean instances with a Groovy script:

my-test.yaml

```
actions:
  - groovy:
      beans:
        script: |
          beans {
            bean(DefaultTextEqualsMessageValidator.class)
          }
```

The configuration script initializes a new bean instance of type `org.citrusframework.validation.DefaultTextEqualsMessageValidator.class`. A `receive` test action will

automatically resolve the new message validator instance from the Citrus bean registry to perform its message validation.

The bean type must have a default constructor in this case.

Create beans in registry

```
beans {
    bean('foo', org.example.foo.FooBean.class)
    fooClient = new org.example.foo.FooClientBuilder().build()
}
```

The example creates a new instance of `FooBean` with the id `foo` in the Citrus bean registry. The `FooBean` class must have a default constructor in this case, too.

You can also create the instances on your own (e.g. using a builder factory) as seen with the `fooClient` bean in this example.

All beans added to the Citrus registry are subject to being injected into or resolved by test actions and other components in Citrus.

5.4.2. Templates

Templates group action sequences to a logical unit. You can think of templates as reusable components that are used in several tests. The maintenance is much more efficient because you need to apply changes only on the templates and all referenced use cases are updated automatically.

The template gets identified by a unique name. Inside a test case we apply the template referencing this unique name. Have a look at a first example:

echo-template.yaml

```
name: echo
description: Sample template in YAML
parameters:
  - name: message
    value: "Citrus rocks!"
actions:
  - echo:
      message: "${message}"
```

The code example above describes a template called `echo`. Templates hold a sequence of test actions or calls other templates themselves. The template may define a set of parameters that represent customizable user input when applying the template.

The template is available for multiple tests that may apply the template as part of their test action sequence.

Apply templates

```
name: ApplyTemplateTest
author: Christoph
status: FINAL
description: Sample test in YAML
actions:
  - applyTemplate:
      name: echo
  - applyTemplate:
      name: echo
      parameters:
        - name: message
          value: "Hello from Citrus!"
```

The example references the `echo` template and call it as part of the test with `<apply-template/>`. You can customize the template parameter to overwrite the default variable `text` with a custom value.

The example above assumes that you have added the template to the Citrus bean registry with the name `echo`. If this is not the case you may also reference the file resource path when applying the template:

Apply templates

```
name: ApplyTemplateTest
author: Christoph
status: FINAL
description: Sample test in YAML
actions:
  - applyTemplate:
      file: "classpath:templates/yaml/echo-template.yaml"
      parameters:
        - name: message
          value: "Citrus rocks!"
```

Template parameters

How to handle test variables when dealing with templates, in particular when templates are defined in separate source files? A template may use different variable names compared to those names used a test and vice versa. The template fails immediately when respective variables are not present. Therefore, templates introduce the concept of parameters to declare its variable names.

Assume you have the following template:

sayHello template

```
name: sayHello
description: Sample template in YAML
parameters:
  - name: user
    value: "Citrus"
actions:
  - echo:
      message: "Hello ${user}!"
```

The template `sayHello` in the example uses the variable `${user}`.

In case the test case already uses the template parameter as a value you do not need to set this variable explicitly on the apply template action.

Implicit template parameter

```
name: SayHelloTemplateTest
author: Christoph
status: FINAL
description: Sample test in YAML
variables:
  - name: "user"
    value: "Christoph"
actions:
  - applyTemplate:
      name: sayHello
```

Output

```
Hello Christoph!
```

The implicit template parameter may break as soon as the template or the test changes its parameter or variable name.

The test case may also explicitly set the template parameter when applying the template:

Explicit template parameter

```
name: SayHelloTemplateTest
author: Christoph
status: FINAL
description: Sample test in YAML
variables:
  - name: "username"
    value: "Mickey"
actions:
  - applyTemplate:
      name: sayHello
      parameters:
        - name: user
          value: "${username}"
```

Output

```
Hello Mickey!
```

The variable **username** is already present in the test case and gets translated into the **user** parameter. Following from that the template works fine although test and template do work on different variable names.

With template parameters you are able to solve the calibration effort when working with templates and variables. It is always a good idea to check the used variables/parameters inside a template when calling it. There might be a variable that is not declared yet inside your test. So you need to define this value as a parameter.

Template parameter scope

A template is able to change a test variable value for the rest of the test case. It is important to know about this when changing test variables in a template as it affects the descendant test actions. Especially when running test actions or templates in parallel to each other this might lead to unexpected behavior. You may choose the template parameter scope for this scenario.

When a template works on variable values and parameters changes to these variables will automatically affect the variables in the whole test. So if you change a variable's value inside a template and the variable is defined inside the test case the changes will affect the variable in a global context. We have to be careful with this when executing a template several times in a test, especially in combination with parallel containers (see containers-parallel).

print template

```
name: print
description: Sample template in YAML
parameters:
  - name: index
    value: "0"
  - name: area
    value: "Atlantis"
actions:
  - echo:
      message: "#${index} Hello ${area}!"
```

Global scope parameter (default)

```
name: SayHelloTemplateTest
author: Christoph
status: FINAL
description: Sample test in YAML
variables:
  - name: "username"
    value: "Mickey"
actions:
  - parallel:
      actions:
      - applyTemplate:
          name: print
          parameters:
            - name: index
              value: "1"
            - name: area
              value: "Europe"
      - applyTemplate:
          name: print
          parameters:
            - name: index
              value: "2"
            - name: area
              value: "Asia"
      - applyTemplate:
          name: print
          parameters:
            - name: index
              value: "3"
            - name: area
              value: "Africa"
```

In the listing above a template **print** is called several times in a parallel container. The parameter values will be handled in a global context, so it is quite likely to happen that the template instances influence each other during execution. We might get such print messages:

Output

```
#2 Hello Europe!
#2 Hello Africa!
#3 Hello Africa!
```

Index parameters do not fit and the message **'Hello Asia'** is completely gone. This is because templates overwrite parameters to each other as they are executed in parallel at the same time. To avoid this behavior we need to tell the template that it should handle parameters as well as variables in a local context. This will enforce that each template instance is working on a dedicated local context. See the **globalContext** key that is set to **false** in this example:

print template

```
name: print
description: Sample template in YAML
globalContext: false
parameters:
  - name: index
    value: "0"
  - name: area
    value: "Atlantis"
actions:
  - echo:
      message: "#${index} Hello ${area}!"
```

After that template instances will not influence each other anymore. But notice that variable changes inside the template then do not affect the test case neither.

5.5. Groovy

Running Citrus tests in Groovy is a specialization of running Citrus tests in Java. The Groovy language gives you some syntax sugar and optimized statements that you can use to write your tests.

As usual, you need to choose one of the supported test engines in Citrus to actually run the test (e.g. JUnit Jupiter or TestNG). You can then use the `@CitrusTestSource` annotation to load the Groovy code that represents the test case.

MyFirstGroovyIT.java

```
package org.citrusframework;

import org.citrusframework.annotations.CitrusTestSource;
import org.citrusframework.testng.TestNGCitrusSupport;
import org.testng.annotations.Test;

public class MyFirstGroovyIT extends TestNGCitrusSpringSupport {

    @Test
    @CitrusTestSource(type = "groovy", name="my-test.groovy")
    public void myFirstGroovyIT() {
    }
}
```

The test logic itself is coded in a Groovy file like this:

my-test.groovy

```
import static org.citrusframework.actions.EchoAction.Builder.echo

name "MyFirstGroovyTest"
author "Christoph"
status "FINAL"
description "Sample test in Groovy"

variables {
    text: "Hello from Citrus!"
}

actions {
    $(echo().message("${text}"))
}
```

5.5.1. Configuration scripts

A Groovy test is able to create endpoints and other configuration entities as part of the test. The script creates these instances as beans in the Citrus registry.

A configuration script may use methods and closures to define the bean instances. The configuration script supports following methods:

- queues()
- endpoints()
- beans()

my-test.groovy

```
configuration {
    queues {
        queue('say-hello')
    }

    endpoints {
        direct('hello')
            .asynchronous()
            .queue('say-hello')
    }

    beans {
        bean(DefaultTextEqualsMessageValidator.class)
    }
}

actions {
    $(send().endpoint(hello)
            .message()
            .body("Hello Citrus!")
    )
}
```

The test above creates a new `direct` endpoint called `hello` that uses an in-memory queue named `say-hello`. Also, the configuration script initializes a new bean instance of type `org.citrusframework.validation.DefaultTextEqualsMessageValidator.class`.

The `send` test action is able to reference the new endpoint by its name to send a message body to the queue. A `receive` test action will automatically resolve the new message validator instance from the Citrus bean registry to perform its message validation.

The Groovy language provides a comfortable way to define common beans in the Citrus registry.

Create beans in registry

```
configuration {
    beans {
        bean('foo', org.example.foo.FooBean.class)
        fooClient = new org.example.foo.FooClientBuilder().build()
    }
}
```

The example creates a new instance of `FooBean` with the id `foo` in the Citrus bean registry. The `FooBean` class must have a default constructor in this case.

You can also create the instances on your own (e.g. using a builder factory) as seen with the `fooClient` bean in this example.

All beans added to the Citrus registry are subject to being injected into or resolved by test actions and other components in Citrus. For instance a JMS test action requires a client connection factory bean instance in the Citrus registry.

5.5.2. Gherkin style syntax

The Gherkin `Given-When-Then` syntax helps you to categorize and structure the individual steps in your test. The Groovy test language in Citrus can use this concept, too.

my-test.groovy

```
configuration {
    queues {
        queue('say-hello')
    }

    endpoints {
        direct('hello') {
            asynchronous()
                .queue('say-hello')
        }
    }
}

given:
    $(createVariables()
        .variable("text", "Citrus rocks!")
    )

when:
    $(send().endpoint(hello)
            .message()
            .body('${text}')
    )

then:
    $(receive().endpoint(hello)
            .message()
            .body('${text}')
    )
```

5.5.3. Message builders

The Groovy language is good choice when it comes to creating domain specific languages. You can provide syntactical sugar to make it as comfortable as possible to use your domain.

As an example you can use message builders to create message body payloads in XML or Json format.

my-test.groovy

```
configuration {
    queues {
        queue('say-hello')
    }

    endpoints {
        direct('hello') {
            asynchronous()
                .queue('say-hello')
        }
    }
}

given:
    $(createVariables()
        .variable("text", "Citrus rocks!"))

when:
    $(send().endpoint(hello)
        .message {
            body {
                json()
                    .greeting {
                        text '${text}'
                        language 'eng'
                    }
            }
            headers {
                operation = "sayHello"
            }
        })

then:
    $(receive().endpoint(hello)
        .message {
            body().json {
                greeting {
                    text '${text}'
                    language 'eng'
                }
            }
            headers {
                operation = "sayHello"
            }
        })
```

The example above deals with Json message payloads and leverages the message builders to create a Json body like this:

body.json

```
{
  "greeting": {
    "text": "${text}",
    "language": "eng"
  }
}
```

The same approach works fine with XML payloads:

my-test.groovy

```
configuration {
    queues {
        queue('say-hello')
    }

    endpoints {
        direct('hello') {
            asynchronous()
                .queue('say-hello')
        }
    }
}

given:
    $(createVariables()
        .variable("text", "Citrus rocks!"))

when:
    $(send().endpoint(hello)
        .message {
            body {
                xml()
                    .greeting(language: 'eng') {
                        text '${text}'
                    }
            }
            headers {
                operation = "sayHello"
            }
        })

then:
    $(receive().endpoint(hello)
        .message {
            body().xml {
                greeting(language: 'eng') {
                    text '${text}'
                }
            }
            headers {
                operation = "sayHello"
            }
        })
```

The test actions use the XML message builder and create XML body payloads such as:

body.xml

```
<greeting language='eng'>
  <text>Citrus rocks!</text>
</greeting>
```

5.5.4. Templates

Templates group action sequences to a logical unit. You can think of templates as reusable components that are used in several tests. The maintenance is much more efficient because you need to apply changes only on the templates and all referenced use cases are updated automatically.

The template gets identified by a unique name. Inside a test case we apply the template referencing this unique name. Have a look at a first example:

echo-template.groovy

```
import static org.citrusframework.actions.EchoAction.Builder.echo

name "echo"
description "Sample template in Groovy"

parameters {
    message="Citrus rocks!"
}

actions {
    $(echo().message('${message}'))
}
```

The code example above describes a template called `echo`. Templates hold a sequence of test actions or calls other templates themselves. The template may define a set of parameters that represent customizable user input when applying the template.

The template is available for multiple tests that may apply the template as part of their test action sequence.

Apply templates

```
import static org.citrusframework.container.Template.Builder.applyTemplate

name "ApplyTemplateTest"
author "Christoph"
status "FINAL"
description "Sample test in Groovy"

actions {
    $(applyTemplate().templateName("echo"))

    $(applyTemplate().templateName("echo")
        .parameter("text", "Hello from Citrus!"))
}
```

The example references the `echo` template and call it as part of the test with `applyTemplate()`. You can customize the template parameter to overwrite the default variable `text` with a custom value.

The example above assumes that you have added the template to the Citrus bean registry with the name `echo`. If this is not the case you may also reference the file resource path when applying the template:

Apply templates

```
import static org.citrusframework.container.Template.Builder.applyTemplate

actions {
    $(applyTemplate().file("classpath:templates/groovy/echo-template.groovy"))
}
```

Template parameters

How to handle test variables when dealing with templates, in particular when templates are defined in separate source files? A template may use different variable names compared to those names used a test and vice versa. The template fails immediately when respective variables are not present. Therefore, templates introduce the concept of parameters to declare its variable names.

Assume you have the following template:

sayHello template

```
import static org.citrusframework.actions.EchoAction.Builder.echo

name "sayHello"
description "Sample template in Groovy"

parameters {
    user="Citrus"
}

actions {
    $(echo().message('Hello ${user}!'))
}
```

The template `sayHello` in the example uses the variable `${user}`.

In case the test case already uses the template parameter as a value you do not need to set this variable explicitly on the apply template action.

Implicit template parameter

```
import static org.citrusframework.container.Template.Builder.applyTemplate

name "SayHelloTemplateTest"
description "Sample test in Groovy"

variables {
    user = "Christoph"
}

actions {
    $(applyTemplate().templateName("sayHello"))
}
```

Output

```
Hello Christoph!
```

The implicit template parameter may break as soon as the template or the test changes its parameter or variable name.

The test case may also explicitly set the template parameter when applying the template:

Explicit template parameter

```
import static org.citrusframework.container.Template.Builder.applyTemplate

name "SayHelloTemplateTest"
description "Sample test in Groovy"

variables {
    username = "Mickey"
}

actions {
    $(applyTemplate().templateName("sayHello")
        .parameter("user", "${username}"))
}
```

Output

```
Hello Mickey!
```

The variable **username** is already present in the test case and gets translated into the **user** parameter. Following from that the template works fine although test and template do work on different variable names.

With template parameters you are able to solve the calibration effort when working with templates and variables. It is always a good idea to check the used variables/parameters inside a template when calling it. There might be a variable that is not declared yet inside your test. So you need to define this value as a parameter.

Template parameter scope

A template is able to change a test variable value for the rest of the test case. It is important to know about this when changing test variables in a template as it affects the descendant test actions. Especially when running test actions or templates in parallel to each other this might lead to unexpected behavior. You may choose the template parameter scope for this scenario.

> When a template works on variable values and parameters changes to these variables will automatically affect the variables in the whole test. So if you change a variable's value inside a template and the variable is defined inside the test case the changes will affect the variable in a global context. We have to be careful with this when executing a template several times in a test, especially in combination with parallel containers (see containers-parallel).

print template

```
import static org.citrusframework.actions.EchoAction.Builder.echo

name "print"
description "Sample template in Groovy"

parameters {
    index = "0"
    area = "Atlantis"
}

actions {
    $(echo().message('#${index} Hello ${area}!'))
}
```

Global scope parameter (default)

```
actions {
    $(parallel().actions(
        applyTemplate().templateName("print")
            .parameter("index", "1")
            .parameter("area", "Europe"),
        applyTemplate().templateName("print")
            .parameter("index", "2")
            .parameter("area", "Asia"),
        applyTemplate().templateName("print")
            .parameter("index", "3")
            .parameter("area", "Africa")
        )
    )
}
```

In the listing above a template **print** is called several times in a parallel container. The parameter values will be handled in a global context, so it is quite likely to happen that the template instances influence each other during execution. We might get such print messages:

Output

```
#2 Hello Europe!
#2 Hello Africa!
#3 Hello Africa!
```

Index parameters do not fit and the message **'Hello Asia'** is completely gone. This is because templates overwrite parameters to each other as they are executed in parallel at the same time. To avoid this behavior we need to tell the template that it should handle parameters as well as variables in a local context. This will enforce that each template instance is working on a dedicated local context. See the **globalContext()** method that is set to **false** in this example:

print template

```
import static org.citrusframework.actions.EchoAction.Builder.echo

name "print"
description "Sample template in Groovy"

globalContext(false)

parameters {
    index = "0"
    area = "Atlantis"
}

actions {
    $(echo().message('#${index} Hello ${area}!'))
}
```

After that template instances will not influence each other anymore. But notice that variable changes inside the template then do not affect the test case neither.

5.6. Spring Bean XML

> The Spring bean XML domain specific language for writing tests is deprecated. Please consider using the arbitrary XML test language syntax instead (see chapter Run XML tests])

The Spring bean XML domain specific language is a variation of the arbitrary XML domain specific language in Citrus. The Spring XML file holds all test actions in the form of Spring XML beans.

> You need to have the runtime module `citrus-spring` enabled in your project to use the Spring bean XML test case language.

Citrus uses Spring bean definition parsers to load the XML test actions. The Spring application context loads the Spring XML test case file as a set of Spring bean configurations. Citrus runs the resulting test Spring bean as a normal unit test using one of the provided runtimes like JUnit Jupiter or TestNG.

You define the complete test logic in the Spring bean XML file:

MyFirstCitrus_IT.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<spring:beans
    xmlns="http://www.citrusframework.org/schema/testcase"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:spring="http://www.springframework.org/schema/beans"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
                        http://www.citrusframework.org/schema/testcase
http://www.citrusframework.org/schema/testcase/citrus-testcase.xsd">

    <testcase name="MyFirstCitrus_IT">
      <description>
          First example showing the basic test case definition elements!
      </description>
      <variables>
        <variable name="text" value="Hello Test Framework"/>
      </variables>
      <actions>
        <echo>
          <message>${text}</message>
        </echo>
      </actions>
    </testcase>
</spring:beans>
```

All Spring XML test definitions use a custom Spring XML schema that aims to reach the convenience of a domain specific language (DSL).

The definition uses the `<spring:beans>` root element that declares all XML namespaces used in the file. This is because the XML file will be loaded as a Spring bean definition. The root element defines a `testcase` element which represents the actual Citrus test.

The test case itself gets a mandatory name that must be unique throughout all test cases in a project. You will receive errors when using duplicate test names. The test name has to follow the common Java naming conventions and rules for Java classes. This means names must not contain whitespace characters except `'-'`, `'.'` and `'_'`.

For example, ***TestFeature_1*** is valid but ***Test Feature 1*** is not because of the space characters.

5.6.1. Java test runner

Despite the fact that the complete test logic is defined in XML we need a Java class to actually load the XML file and run the test as part of JUnit, Cucumber or TestNG depending on the test engine you choose. The Java class acts as a wrapper to load the XML file and execute the test.

The Spring XML test case definition in Citrus uses two files that are connected via naming conventions. In a Maven project these two files would be:

XML test case files

```
src/test/java/org/citrusframework/MyFirstCitrus_IT.java
src/test/resources/org/citrusframework/MyFirstCitrus_IT.xml
```

The files above represent a test called `MyFirstCitrus_IT`. The `.java` file defines the runtime that should be used to execute the test. This Java file does not have any test logic and is not likely to be changed. In fact, you can generate the Java file from Citrus (e.g. Maven plugin).

Generate test files via Maven plugin

```
mvn citrus:create-test
```

Interactive test creation

```
Enter test name:: MyFirstCitrus_IT
Enter test author: Unknown::
Enter test description:: Sample XML test
Enter test package: org.citrusframework::
Choose unit test framework: (testng/junit4/junit5) testng::
Choose target code base type: (java/xml) java:: xml
Create test with XML schema? (y/n) n::
Create test with WSDL? (y/n) n::
Create test with Swagger API? (y/n) n::

Confirm test creation:
type: xml
framework: testng
name: MyFirstCitrus_IT
author: Unknown
description:
package: org.citrusframework
 (y/n) y:: y

[INFO] Successfully created new test case org.citrusframework.MyFirstCitrus_IT
```

The command above creates the two test files (`.java` and `.xml`) using an interactive mode. The user provides the test case information such as test name, package, runtime and so on.

MyFirstCitrus_IT.java

```
package org.citrusframework;

import org.citrusframework.annotations.CitrusResource;
import org.citrusframework.annotations.CitrusTestSource;
import org.citrusframework.testng.TestNGCitrusSupport;
import org.testng.annotations.Optional;
import org.testng.annotations.Parameters;
import org.testng.annotations.Test;

public class MyFirstCitrus_IT extends TestNGCitrusSpringSupport {

    @Test
    @CitrusTestSource(type = "spring", name="MyFirstCitrus_IT")
    public void myFirstCitrus_IT() {
    }
}
```

The purpose of the generated Java class is to load the XML file as a Spring bean test and execute the test case.

The default naming convention requires the XML file with the test name in the same package as the generated Java class. This makes sure that the Java class is able to find the XML file when loading and running the test.

5.6.2. Test meta information

The user is able to provide some additional information about the test case. The meta-info section at the very beginning of the test case holds information like author, status or creation date.

Test meta information

```
<testcase name="metaInfoTest">
    <meta-info>
        <author>Christoph Deppisch</author>
        <creationdate>2008-01-11</creationdate>
        <status>FINAL</status>
        <last-updated-by>Christoph Deppisch</last-updated-by>
        <last-updated-on>2008-01-11T10:00:00</last-updated-on>
    </meta-info>
    <description>
        ...
    </description>
    <actions>
        ...
    </actions>
</testcase>
```

The status allows the following values:

- DRAFT
- READY_FOR_REVIEW
- DISABLED
- FINAL

This information gives the reader first impression about the test and is also used to generate test documentation. By default, Citrus is able to generate test reports in HTML and Excel in order to list all tests with their metadata information and description.

> Tests with the status DISABLED will not be executed during a test suite run. So someone can just start adding planned test cases that are not finished yet in status DRAFT. In case a test is not runnable yet because it is not finished, someone may disable a test temporarily to avoid causing failures during a test run.

The test description should give a short introduction to the intended use case scenario that will be tested. The user should get a short summary of what the test case is trying to verify. You can use free text in your test description no limit to the number of characters. Please be aware of the XML validation rules of well-formed XML (e.g. special character escaping). The usage of CDATA sections for large descriptions may be a good idea, too.

5.6.3. Finally block

Java developers might be familiar with the concept of try-catch-finally blocks. The ***finally*** section contains a list of test actions that will be executed guaranteed at the very end of the test case even if errors did occur during the execution before.

This is the right place to tidy up things that were previously created by the test like cleaning up the database for instance.

Finally block

```
<finally>
    <echo>
        <message>Do finally - regardless of what has happened before</message>
    </echo>
</finally>
```

As an example imagine that you have prepared some data inside the database at the beginning of the test and you need to make sure the data is cleaned up at the end of the test case.

Finally block example

```
<testcase name="finallyTest">
    <variables>
        <variable name="orderId" value="citrus:randomNumber(5)"/>
        <variable name="date" value="citrus:currentDate('dd.MM.yyyy')"/>
    </variables>
    <actions>
        <sql datasource="testDataSource">
            <statement>
                INSERT INTO ORDERS VALUES (${orderId}, 1, 1, '${date}')
            </statement>
        </sql>

        <echo>
            <message>
                ORDER creation time: ${date}
            </message>
        </echo>
    </actions>
    <finally>
        <sql datasource="testDataSource">
            <statement>
              DELETE FROM ORDERS WHERE ORDER_ID='${orderId}'
            </statement>
        </sql>
    </finally>
</testcase>
```

In the example the first action creates an entry in the database using an ***INSERT*** statement. To be sure that the entry in the database is deleted after the test, the finally section contains the respective ***DELETE*** statement that is always executed regardless the test case state (successful or failed).

5.6.4. Variables with CDATA sections

When using the XML test definition you must obey the XML rules for variable values. First of all you need to escape XML reserved characters such as `<`, `&` or `"` with `<`, `&` or `"`. Other values such as XML snippets would also interfere with the XML well-formed paradigm. You can use CDATA sections within the variable value element as a solution.

Variable CDATA sections

```
<variables>
  <variable name="persons">
    <value>
      <data>
        <![CDATA[
          <persons>
            <person>
              <name>Theodor</name>
              <age>10</age>
            </person>
            <person>
              <name>Alvin</name>
              <age>9</age>
            </person>
          </persons>
        ]]>
      </data>
    </value>
  </variable>
</variables>
```

That is how you can use structured variable values in the XML DSL.

5.6.5. Variables with Groovy

You can also use a script to create variable values. This is extremely handy when you have very complex variable values. Just code a small Groovy script for instance in order to define the variable value. A small sample should give you the idea how that works:

Groovy variable script

```
<variables>
  <variable name="avg">
    <value>
      <script type="groovy">
        <![CDATA[
          a = 4
          b = 6
          return (a + b) / 2
        ]]>
      </script>
    </value>
  </variable>
  <variable name="sum">
    <value>
      <script type="groovy">
        <![CDATA[
          5 + 5
        ]]>
      </script>
    </value>
  </variable>
</variables>
```

Just use the script code right inside the variable value definition. The value of the variable is the result of the last operation performed within the script. For longer script code the use of `<![CDATA[ ]]>` sections is recommended.

Citrus uses the JavaScript engine mechanism to evaluate the script code. By default, Groovy is supported as a script engine implementation. You can add additional engine implementations to your project and support other script types, too.

5.6.6. Templates

Templates group action sequences to a logical unit. You can think of templates as reusable components that are used in several XML tests. The maintenance is much more efficient because you need to apply changes only on the templates and all referenced use cases are updated automatically.

The template always has a unique name. Inside a test case we call the template by this unique name. Have a look at a first example:

XML templates

```
<template name="doCreateVariables">
    <create-variables>
        <variable name="var" value="123456789"/>
    </create-variables>

    <call-template name="doTraceVariables"/>
</template>

<template name="doTraceVariables">
    <echo>
        <message>Current time is: ${time}</message>
    </echo>

    <trace-variables/>
</template>
```

The code example above describes two template definitions. Templates hold a sequence of test actions or call other templates themselves as seen in the example above.

The `<call-template>` action calls other templates by their name. The called template not necessarily has to be located in the same test case XML file. The template might be defined in a separate XML file other than the test case itself:

Call XML templates

```
<testcase name="templateTest">
    <variables>
        <variable name="myTime" value="citrus:currentDate()"/>
    </variables>
    <actions>
        <call-template name="doCreateVariables"/>

        <call-template name="doTraceVariables">
            <parameter name="time" value="${myTime}">
        </call-template>
    </actions>
</testcase>
```

There is an open question when dealing with templates that are defined somewhere else outside the test case. How to handle variables? A templates may use different variable names then the test and vice versa. No doubt the template will fail as soon as special variables with respective values are not present. Unknown variables cause the template and the whole test to fail with errors.

So a first approach would be to harmonize variable usage across templates and test cases, so that templates and test cases do use the same variable naming. But this approach might lead to high calibration effort. Therefore, templates support parameters to solve this problem. When a template is called the calling actor is able to set some parameters. Let us discuss an example for this issue.

The template "doDateConversion" in the next sample uses the variable ${date}. The calling test case can set this variable as a parameter without actually declaring the variable in the test itself:

Template parameter

```
<call-template name="doDateConversion">
    <parameter name="date" value="${sampleDate}"/>
</call-template>
```

The variable **sampleDate** is already present in the test case and gets translated into the **date** parameter. Following from that the template works fine although test and template do work on different variable namings.

With template parameters you are able to solve the calibration effort when working with templates and variables. It is always a good idea to check the used variables/parameters inside a template when calling it. There might be a variable that is not declared yet inside your test. So you need to define this value as a parameter.

Template parameters may contain more complex values like XML fragments. The call-template action offers following CDATA variation for defining complex parameter values:

Complex parameter values

```
<call-template name="printXMLPayload">
    <parameter name="payload">
      <value>
        <![CDATA[
          <HelloRequest
xmlns="http://citrusframework.org/schemas/samples/sayHello.xsd">
            <Text>Hello South ${var}</Text>
          </HelloRequest>
        ]]>
      </value>
    </parameter>
</call-template>
```

When a template works on variable values and parameters changes to these variables will automatically affect the variables in the whole test. So if you change a variable's value inside a template and the variable is defined inside the test case the changes will affect the variable in a global context. We have to be careful with this when executing a template several times in a test, especially in combination with parallel containers (see containers-parallel).

Global scope parameter

```
<parallel>
    <call-template name="print">
        <parameter name="param1" value="1"/>
        <parameter name="param2" value="Hello Europe"/>
    </call-template>
    <call-template name="print">
        <parameter name="param1" value="2"/>
        <parameter name="param2" value="Hello Asia"/>
    </call-template>
    <call-template name="print">
        <parameter name="param1" value="3"/>
        <parameter name="param2" value="Hello Africa"/>
    </call-template>
</parallel>
```

In the listing above a template **print** is called several times in a parallel container. The parameter values will be handled in a global context, so it is quite likely to happen that the template instances influence each other during execution. We might get such print messages:

Output

```
2. Hello Europe
2. Hello Africa
3. Hello Africa
```

Index parameters do not fit and the message **'Hello Asia'** is completely gone. This is because templates overwrite parameters to each other as they are executed in parallel at the same time. To avoid this behavior we need to tell the template that it should handle parameters as well as variables in a local context. This will enforce that each template instance is working on a dedicated local context. See the **global-context** attribute that is set to **false** in this example:

Local scope parameter

```
<template name="print" global-context="false">
    <echo>
        <message>${param1}.${param2}</message>
    </echo>
</template>
```

After that template instances won't influence each other anymore. But notice that variable changes inside the template then do not affect the test case neither.

Chapter 6. Test variables

The usage of test variables is a core concept when writing maintainable tests. The key identifiers of a test case should be exposed as test variables at the very beginning of a test. This avoids hard coded identifiers and multiple redundant values inside the test.

Java

```
public void fooService_IT() {
    variable("text", "Hello Citrus!");
    variable("customerId", "123456789");

    run(echo("Text: ${text} Id: ${id}"));
}
```

XML

```
<testcase name="FooService_IT">
  <variables>
    <variable name="text" value="Hello Citrus!"/>
    <variable name="customerId" value="123456789"/>
  </variables>

  <actions>
    <echo>
      <message>Text: ${text} Id: ${id}</message>
    </echo>
  </actions>
</testcase>
```

Test variables help significantly when writing complex tests with lots of identifiers and semantic data. The variables are valid for the whole test case. You can reference a variable multiple times using a common variable expression `${variable-name}`.

The usage of variables should make the test easier to maintain and more flexible. All essential entities and identifiers are present right at the beginning of the test, which may also give the opportunity to easily create test variants by simply changing the variable values for other test scenarios (e.g. different error codes, identifiers).

The name of the variable is arbitrary. Of course, you need to be careful with special characters and reserved XML entities like '&', '<', '>'. In general, you can apply to the Java naming convention, and you will be fine.

6.1. Global variables

You can work with different variable scopes (local or global). Local variables are accessible throughout a single test. Global variables are visible for all tests yet global variables are immutable, so tests cannot change their values.

This is a good opportunity to declare constant values for all tests. As these variables are global we need to add those to the basic Citrus context. The following example demonstrates how to add global variables in Citrus:

Java DSL

```
@Bean
public GlobalVariables globalVariables() {
    return new GlobalVariables.Builder()
        .variable("projectName", "Citrus Integration Testing")
        .variable("userName", "TestUser")
        .build();
}
```

XML DSL

```
<citrus:global-variables>
  <citrus:variable name="projectName" value="Citrus Integration Testing"/>
  <citrus:variable name="userName" value="TestUser"/>
</citrus:global-variables>
```

We add the Spring bean component to the application context file. The component receives a list of name-value variable elements. You can reference the global variables in your test cases as usual.

Another possibility to set global variables is to load those from external property files. This may give you more powerful global variables with user specific properties for instance. See how to load property files as global variables in this example:

Java DSL

```
@Bean
public GlobalVariablesPropertyLoader propertyLoader() {
    GlobalVariablesPropertyLoader propertyLoader = new
GlobalVariablesPropertyLoader();

    propertyLoader.getPropertyFiles().add("classpath:global-variable.properties");

    return propertyLoader;
}
```

XML DSL

```
<citrus:global-variables>
  <citrus:file path="classpath:global-variable.properties"/>
</citrus:global-variables>
```

You can use the `GlobalVariablesPropertyLoader` component and add it to the context as a Spring bean. Citrus loads the given property file content as global test variables. You can mix property file

and name-value pair variable definitions in the global variables component.

> The global variables can have variable expressions and Citrus functions. It is possible to use previously defined global variables as values of new variables, like in this example:

global-variable.properties

```
user=Citrus
greeting=Hello ${user}!
date=citrus:currentDate('yyyy-MM-dd')
```

6.2. Extract variables

Imagine you receive a message in your test with some generated message identifier values. You have no chance to predict the identifier value because it was generated at runtime by a foreign application. You can ignore the value in order to protect your validation. In many cases you might want to save this identifier in order to use this value in the respective response message or somewhat later on in the test.

The solution is to extract dynamic values from received messages and save those to test variables at runtime.

6.2.1. JsonPath expressions

When an incoming message is passing the message validation the user can extract some values of that received message to new test variables for later use in the test.

```
<message type="json">
  <data>
    { "user":
      {
        "name": "Admin",
        "password": "secret",
        "admin": "true",
        "aliases": ["penny","chef","master"]
      }
    }
  </data>
  <extract>
    <message path="$.user.name" variable="userName"/>
    <message path="$.user.aliases" variable="userAliases"/>
    <message path="$.user[?(@.admin)].password" variable="adminPassword"/>
  </extract>
</message>
```

With this example we have extracted three new test variables via JSONPath expression evaluation.

The three test variables will be available to all upcoming test actions. The variable values are:

```
userName=Admin
userAliases=["penny","chef","master"]
adminPassword=secret
```

As you can see we can also extract complex JSONObject items or JSONArray items. The test variable value is a String representation of the complex object.

6.2.2. XPath expressions

Add this code to your message receiving action.

Java DSL

```
@CitrusTest
public void receiveMessageTest() {
    when(
        receive("helloService")
            .extract(fromBody()
                    .expression("//TestRequest/VersionId", "versionId"))
            .extract(fromHeaders()
                    .header("Operation", "operation"))
    );

    then(
        echo("Extracted operation from header is: ${operation}")
    );

    then(
        echo("Extracted version from body is: ${versionId}")
    );
}
```

XML DSL

```
<receive endpoint="helloService">
    <message>
      ...
    </message>
    <extract>
      <header name="Operation" variable="operation"/>
      <message path="/TestRequest/VersionId" variable="versionId"/>
    </extract>
</receive>

<echo>
  <message>Extracted operation from header is: ${operation}</message>
</echo>

<echo>
  <message>Extracted version from body is: ${versionId}</message>
</echo>
```

As you can see Citrus is able to extract both header and message body content into test variables. The extraction will automatically create a new variable in case it does not exist. The time the variable was created all following test actions can access the test variables as usual. So you can reference the variable values in response messages or other test steps ahead.

We can also use expression result types in order to manipulate the test variable outcome. In case we use a **boolean** result type the existence of elements can be saved to variable values. The result type **node-set** translates a node list result to a comma separated string of all values in this node list. Simply use the expression result type attributes as shown in previous sections.

6.3. Path expressions

Some elements in message body might be of dynamic nature. Just think of generated identifiers or timestamps. This is the right time to use test variables and dynamic message element overwrite. You can overwrite specific elements in the message body with path expressions (XPath or JsonPath).

6.3.1. JsonPath expressions

First thing we want to do with JsonPath is to manipulate a message content before it is actually processed. This is very useful when working with message file resources that are reused across multiple test cases. Each test case can manipulate the message content individually with JsonPath before processing the message content.

Let’s have a look at this simple sample Json message body:

Json message body user.json

```
{ "user":
  {
    "id": citrus:randomNumber(10),
    "name": "Unknown",
    "admin": "?",
    "projects":
      [{
        "name": "Project1",
        "status": "open"
      },
      {
        "name": "Project2",
        "status": "open"
      },
      {
        "name": "Project3",
        "status": "closed"
      }]
  }
}
```

Citrus can load the file content and uses it as message body when sending or receiving messages in a test case. You can apply JsonPath expressions in order to manipulate the message content.

```
<message type="json">
  <resource file="file:path/to/user.json" />
  <element path="$.user.name" value="Admin" />
  <element path="$.user.admin" value="true" />
  <element path="$..status" value="closed" />
</message>
```

When all path expressions are evaluated the resulting message looks like follows:

```
{ "user":
  {
    "id": citrus:randomNumber(10),
    "name": "Admin",
    "admin": "true",
    "projects":
      [{
        "name": "Project1",
        "status": "closed"
      },
      {
        "name": "Project2",
        "status": "closed"
      },
      {
        "name": "Project3",
        "status": "closed"
      }]
  }
}
```

The JsonPath expressions set the username to **Admin** . The **admin** boolean property was set to **true** and all project status values were set to **closed**. In case a JsonPath expression should fail to find a matching element within the message structure the test case will fail.

With this JsonPath mechanism you are able to manipulate message content before it is sent or received within Citrus. This makes life very easy when using message resource files that are reused across multiple test cases.

6.3.2. XPath expressions

In case of XML message bodies you can use XPath expressions to manipulate the body content before any message processing takes place.

XML DSL

```
<message>
  <payload>
    <TestMessage>
      <MessageId>${messageId}</MessageId>
      <CreatedAt>?</CreatedAt>
      <VersionId>${version}</VersionId>
    </TestMessage>
  </payload>
  <element path="/TestMessage/CreatedAt" value="${date}"/>
</message>
```

The program listing above shows ways of setting variable values inside a message template. First you can simply place variable expressions inside the message (see how `${messageId}` is used in the

sample). In addition to that you can also use path expressions to explicitly overwrite message elements before message processing takes place.

The sample above uses an XPath expression that evaluates and searches for the right element in the message body in order to set the given value. The previously defined variable **${date}** replaces the respective element value. Of course this works with XML attributes too (e.g. path expression `/TestMessage/Person/@age`).

Both ways via XPath or JsonPath or inline variable expressions are equal to each other. With respect to the complexity of XML namespaces and XPath you may find the inline variable expression more comfortable to use. Anyway feel free to choose the way that fits best for you.

This is how you can overwrite values in message templates in order to increase maintainability and robustness of your test.

> Validation matchers put validation mechanisms to a new level offering dynamic assertion statements for validation. Have a look at the possibilities with assertion statements in validation-matcher.

6.4. Escape variables

The test variable expression syntax `${variable-name}` is preserved to evaluate to a test variable within the current test context. In case the same syntax is used in one of your message content values you need to escape the syntax from being interpreted as test variable expression. You can do this by using the variable expression escaping character sequence `//` wrapping the actual variable name like this:

Plain text message content with escapes

```
This is an escaped variable expression ${//escaped//} and should not lead to unknown
variable exceptions within Citrus.
```

The escaped expression `${//escaped//}` above will result in the string **${escaped}** where *escaped* is not treated as a test variable name but as a normal string in the message body.

This way you are able to have the same variable syntax in a message content without interfering with the Citrus variable expression syntax. As a result Citrus will not complain about not finding the test variable **escaped** in the current context.

The variable syntax escaping characters `//` are automatically removed when the expression is processed by Citrus. So we will get the following result after processing.

Parsed plain text mesage content

```
This is an escaped variable expression ${escaped} and should not lead to unknown
variable exceptions within Citrus.
```

Chapter 7. Message validation

When Citrus receives a message from external applications it is time to verify the message content. This message validation includes syntax rules with schema validation and message content comparison to expected templates. Citrus provides powerful message validation capabilities for different data formats. The tester is able to define expected message headers and body content. The Citrus message validator finds values not matching the expectations and reports the difference as test failure.

7.1. Validation registry

Citrus provides default message validator implementations for different data formats. The Citrus project context automatically loads these default message validators. In case one of these message validators matches the incoming message the message validator performs its validation steps with the message.

All default message validators can be overwritten by binding a component with the same id to the project context (e.g. as Spring bean in the application context).

The default message validator implementations of Citrus are:

defaultXmlMessageValidator	org.citrusframework.validation.xml.DomXmlMessageValidator
defaultXpathMessageValidator	org.citrusframework.validation.xml.XpathMessageValidator
defaultJsonMessageValidator	org.citrusframework.validation.json.JsonTextMessageValidator
defaultJsonPathMessageValidator	org.citrusframework.validation.json.JsonPathMessageValidator
defaultPlaintextMessageValidator	org.citrusframework.validation.text.PlainTextMessageValidator
defaultYamlMessageValidator	org.citrusframework.validation.yaml.YamlMessageValidator
defaultMessageHeaderValidator	org.citrusframework.validation.DefaultMessageHeaderValidator
defaultBinaryBase64MessageValidator	org.citrusframework.validation.text.BinaryBase64MessageValidator

defaultGzipBinaryBase64MessageValidator	org.citrusframework.validation.text.GzipBinaryBase64MessageValidator
defaultXhtmlMessageValidator	org.citrusframework.validation.xhtml.XhtmlMessageValidator
defaultGroovyXmlMessageValidator	org.citrusframework.validation.script.GroovyXmlMessageValidator
defaultGroovyTextMessageValidator	org.citrusframework.validation.script.GroovyScriptMessageValidator
defaultGroovyJsonMessageValidator	org.citrusframework.validation.script.GroovyJsonMessageValidator

You can overwrite a default message validator with a custom implementation. Just add your customized validator implementation as a bean to the Citrus context and use one of the default bean identifiers.

You can add a custom message validator as a component in the context (e.g. as Spring bean in the application context).

Java DSL

```
@Bean
public CustomMessageValidator customMessageValidator() {
    return new CustomMessageValidator();
}
```

XML DSL

```
<bean id="customMessageValidator"
class="org.citrusframework.validation.custom.CustomMessageValidator"/>
```

The listing above adds a custom message validator implementation. The message validator registry will automatically add this validator to the list of available validators in the project.

The custom implementation class has to implement the basic interface **org.citrusframework.validation.MessageValidator**<>. Now Citrus will try to match the custom implementation to incoming message types and occasionally execute the message validator logic when applicable.

7.2. Validation modules

The list of available message validators in your project is controlled by the available message validator implementations on the project classpath.

You need to add validator modules to the project accordingly. For instance if you want to use the default Json message validation capabilities in Citrus you need to add the following dependency:

Json validation module dependency

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-validation-json</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

This adds the Citrus message validator component for dealing with Json message format. This includes message validators and JsonPath support. Now your Citrus project is able to validate Json messages.

Citrus provides the following validation modules:

- citrus-validation-json
- citrus-validation-xml
- citrus-validation-yaml
- citrus-validation-text
- citrus-validation-binary
- citrus-validation-groovy
- citrus-validation-hamcrest

Read more about the individual validation modules in the next sections.

7.3. Json validation

Message formats such as Json have become very popular, in particular when dealing with RESTful services. Citrus is able to expect and validate Json messages with a powerful comparison of Json structures.

> By default, Citrus will use XML message formats when sending and receiving messages. This also reflects to the message validation logic Citrus uses for incoming messages. So by default Citrus will try to parse the incoming message as XML DOM element tree. In case we would like to enable Json message validation we have to tell Citrus that we expect a Json message right now.

Json message validation is not enabled by default in your project. You need to add the validation module to your project as a Maven dependency.

Json validation module dependency

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-validation-json</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

Citrus provides several default message validator implementations for Json messages:

JsonTextMessageValidator

Basic Json message validator implementation compares Json objects/arrays (expected and received).

- The order of Json object properties can differ, whereas array order is always validated.
- Test variables and ignored-placeholders (`@ignore@`) can be used.
- JsonArray as well as nested JsonObjects are supported.

GroovyJsonMessageValidator

Extended groovy message validator provides specific Json slurper support. With Json slurper the tester can validate the Json message body with closures for instance.

The Json validator offers two different modes to operate. By default, **strict** mode is enabled and the validator will also check the exact amount of entries (object properties / array items) to match in received and control message. No additional fields in received Json data structure will be accepted. In **soft** mode the validator allows additional entries (object properties / array items) in received Json, so the control Json can be a subset of the received. Additional entries in the received Json data structure are ignored.

The Json validation mode (strict or soft) is settable via environment variable `CITRUS_JSON_MESSAGE_VALIDATION_STRICT` or system property `citrus.json.message.validation.strict=false`. This will set soft mode to all Json text message validators.

You can also overwrite these default message validators for Json by placing a bean into the Spring Application context. The bean uses a default name as identifier. Then your custom bean will overwrite the default validator:

Java

```
@Bean
public JsonTextMessageValidator defaultJsonMessageValidator() {
    return new JsonTextMessageValidator();
}
```

XML

```
<bean id="defaultJsonMessageValidator"
class="org.citrusframework.validation.json.JsonTextMessageValidator"/>
```

The same approach applies to the Groovy message validator implementation.

Java

```
@Bean
public GroovyJsonMessageValidator defaultGroovyJsonMessageValidator() {
    return new GroovyJsonMessageValidator();
}
```

XML

```
<bean id="defaultGroovyJsonMessageValidator"
class="org.citrusframework.validation.script.GroovyJsonMessageValidator"/>
```

This is how you can customize the message validators used for Json message data.

When a message has been received in Citrus the message validation will try to find a matching message validator according to the message content. You can also specify the Json message format on a receive action in order to force Json message validation.

Java

```
receive(someEndpoint)
    .message()
    .type(MessageType.JSON)
    .body("{" +
            "\"type\" : \"read\"," +
            "\"mbean\" : \"java.lang:type=Memory\"," +
            "\"attribute\" : \"HeapMemoryUsage\"," +
            "\"path\" : \"@equalsIgnoreCase('USED')@\"," +
            "\"value\" : \"${heapUsage}\"," +
            "\"timestamp\" : \"@ignore@\"" +
          "}");
```

XML

```
<receive endpoint="someEndpoint">
    <message type="json">
        <data>
          {
            "type" : "read",
            "mbean" : "java.lang:type=Memory",
            "attribute" : "HeapMemoryUsage",
            "path" : "@equalsIgnoreCase('USED')@",
            "value" : "${heapUsage}",
            "timestamp" : "@ignore@"
          }
        </data>
    </message>
</receive>
```

The message receiving action in our test case specifies a message format type `type="json"` . This tells Citrus to look for some message validator implementation capable of validating Json messages. As we have added the proper message validator to the Spring application context Citrus will pick the right validator and Json message validation is performed on this message.

Instead of defining an expected message body template we can also use Groovy validation scripts. Let's have a look at the Groovy Json message validator example. As usual the default Groovy Json message validator is active by default. But the special Groovy message validator implementation will only jump in when we used a validation script in our receive message definition. Let's have an example for that.

Java

```
receive(someEndpoint)
    .message()
    .type(MessageType.JSON)
    .validate(groovy()
        .script("assert json.type == 'read'\n" +
                "assert json.mbean == 'java.lang:type=Memory'\n" +
                "assert json.attribute == 'HeapMemoryUsage'\n" +
                "assert json.value == '${heapUsage}'"));
```

XML

```
<receive endpoint="someEndpoint">
    <message type="json">
        <validate>
            <script type="groovy">
                <![CDATA[
                  assert json.type == 'read'
                  assert json.mbean == 'java.lang:type=Memory'
                  assert json.attribute == 'HeapMemoryUsage'
                  assert json.value == '${heapUsage}'
                ]]>
            </script>
        </validate>
    </message>
</receive>
```

Again the message type tells Citrus that we expect a message of type **json**. The action uses a validation script written in Groovy to verify the incoming message. Citrus will automatically activate the special message validator that executes our Groovy script.

The script validation is very powerful as we can use the full power of the Groovy language. The validation script automatically has access to the incoming Json message object **json**. We can use the Groovy Json dot notated syntax in order to navigate through the Json structure. The Groovy Json slurper object **json** is automatically injected in the validation script. This way you can access the Json object elements in your code doing some assertions.

There is even more object injection for the validation script. With the automatically added object ***receivedMessage*** you have access to the Citrus message object for this receive action. This enables you to do whatever you want with the message body or header.

Java

```
receive(someEndpoint)
    .message()
    .type(MessageType.JSON)
    .validate(groovy()
        .script("assert receivedMessage.getPayload(String.class).contains(\"Hello
Citrus!\")\n" +
                "assert receivedMessage.getHeader(\"Operation\") == 'sayHello'\n" +
                "context.setVariable(\"request_body\",
receivedMessage.getPayload(String.class))"));
```

XML

```
<receive endpoint="someEndpoint">
    <message type="json">
        <validate>
            <script type="groovy">
                assert receivedMessage.getPayload(String.class).contains("Hello
Citrus!")
                assert receivedMessage.getHeader("Operation") == 'sayHello'

                context.setVariable("request_body",
receivedMessage.getPayload(String.class))
            </script>
        </validate>
    </message>
</receive>
```

The listing above shows some power of the validation script. We can access the message body, we can access the message header. With test context access we can also save the whole message body as a new test variable for later usage in the test.

In general Groovy code inside the XML test case definition or as part of the Java DSL code is not very comfortable to maintain. Neither you do have code syntax assist nor code completion when using inline Groovy scripts.

Also, in case the validation script gets more complex you might want to load the script from an external file resource.

Java

```
receive(someEndpoint)
    .message()
    .type(MessageType.JSON)
    .validate(groovy()
        .script(new ClassPathResource("path/to/validationScript.groovy")));
```

XML

```
<receive endpoint="someEndpoint">
    <message type="json">
        <validate>
            <script type="groovy" file="path/to/validationScript.groovy"/>
        </validate>
    </message>
</receive>
```

We referenced some external file resource ***validationScript.groovy*** . This file content is loaded at runtime and is used as script body. Now that we have a normal groovy file we can use the code completion and syntax highlighting of our favorite Groovy editor.

Using several message validator implementations at the same time in the Spring application context is also no problem. Citrus automatically searches for all available message validators applicable for the given message format and executes these validators in sequence. This means that multiple message validators can coexist in a Citrus project.

Multiple message validators that all apply to the message content format will run in sequence. In case you need to explicitly choose a message validator implementation you can do so in the receive action:

Java

```
receive(someEndpoint)
    .validator(groovyJsonMessageValidator)
    .message()
    .type(MessageType.JSON)
    .validate(groovy()
        .script(new ClassPathResource("path/to/validationScript.groovy")));
```

XML

```
<receive endpoint="someEndpoint">
    <message type="json" validator="groovyJsonMessageValidator">
        <validate>
            <script type="groovy" file="path/to/validationScript.groovy"/>
        </validate>
    </message>
</receive>
```

In this example we use the **groovyJsonMessageValidator** explicitly in the receive test action. The message validator implementation was added as Spring bean with id **groovyJsonMessageValidator** to the Spring application context before. Now Citrus will only execute the explicit message validator. Other implementations that might also apply are skipped.

By default, Citrus consolidates all available message validators. You can explicitly pick a special message validator in the receive message action as shown in the example above. In this case all other validators will not take part in this special message validation. But be careful: When picking a message validator explicitly you are of course limited to this message validator capabilities. Validation features of other validators are not valid in this case (e.g. message header validation, XPath validation, etc.)

7.3.1. Ignore with JsonPath

The next usage scenario for JsonPath expressions in Citrus is the ignoring of elements during message validation. As you already know Citrus provides powerful validation mechanisms for XML and Json message format. The framework is able to compare received and expected message contents with powerful validator implementations. You can use a JsonPath expression for ignoring

a very specific entry in the Json object structure.

Java

```
receive(someEndpoint)
    .message()
    .type(MessageType.JSON)
    .body("{\"users\":" +
            "[{" +
                "\"name\": \"Jane\"," +
                "\"token\": \"?\"," +
                "\"lastLogin\": 0" +
            "}," +
            "{" +
                "\"name\": \"Penny\"," +
                "\"token\": \"?\"," +
                "\"lastLogin\": 0" +
            "}," +
            "{" +
                "\"name\": \"Mary\"," +
                "\"token\": \"?\"," +
                "\"lastLogin\": 0" +
            "}]" +
        "}")
    .validate(json()
                .ignore("$.users[*].token")
                .ignore("$..lastLogin"));
```

XML

```
<receive endpoint="someEndpoint">
    <message type="json">
      <data>
        {
          "users":
          [{
            "name": "Jane",
            "token": "?",
            "lastLogin": 0
          },
          {
            "name": "Penny",
            "token": "?",
            "lastLogin": 0
          },
          {
            "name": "Mary",
            "token": "?",
            "lastLogin": 0
          }]
        }
      </data>
      <ignore expression="$.users[*].token" />
      <ignore expression="$..lastLogin" />
    </message>
</receive>
```

This time we add JsonPath expressions as ignore statements. This means that we explicitly leave out the evaluated elements from validation. Obviously this mechanism is a good thing to do when dynamic message data simply is not deterministic such as timestamps and dynamic identifiers. In the example above we explicitly skip the **token** entry and all **lastLogin** values that are obviously timestamp values in milliseconds.

The JsonPath evaluation is very powerful when it comes to select a set of Json objects and elements. This is how we can ignore several elements with one single JsonPath expression which is very powerful.

7.3.2. JsonPath validation

Let’s continue to use JsonPath expressions when validating a received message in Citrus:

Java

```
receive(someEndpoint)
    .message()
    .type(MessageType.JSON)
    .validate(jsonPath()
        .expression("$.user.name", "Penny")
        .expression("$['user']['name']", "${userName}")
        .expression("$.user.aliases", "[\"penny\",\"jenny\",\"nanny\"]")
        .expression("$.user[?(@.admin)].password", "@startsWith('$%00')@")
        .expression("$.user.address[?(@.type='office')]",
"{\"city\":\"Munich\",\"street\":\"Company Street\",\"type\":\"office\"}"));
```

XML

```
<receive endpoint="someEndpoint">
  <message type="json">
    <validate>
      <json-path expression="$.user.name" value="Penny"/>
      <json-path expression="$['user']['name']" value="${userName}"/>
      <json-path expression="$.user.aliases" value="['penny','jenny','nanny']"/>
      <json-path expression="$.user[?(@.admin)].password"
value="@startsWith('$%00')@"/>
      <json-path expression="$.user.address[?(@.type='office')]"
          value="{&quot;city&quot;:&quot;Munich&quot;,&quot;street&quot;:&quot;Company
Street&quot;,&quot;type&quot;:&quot;office&quot;}"/>
    </validate>
  </message>
</receive>
```

Use path expression map

```
final Map<String, Object> validationMap = new HashMap<>();
validationMap.put("$.user.name", "Penny");
validationMap.put("$['user']['name']", "${userName}");
validationMap.put("$.user.aliases", "[\"penny\",\"jenny\",\"nanny\"]");
validationMap.put("$.user[?(@.admin)].password", "@startsWith('$%00')@");
validationMap.put("$.user.address[?(@.type='office')]",
"{\"city\":\"Munich\",\"street\":\"Company Street\",\"type\":\"office\"}");

receive(someEndpoint)
    .message()
    .type(MessageType.JSON)
    .validate(jsonPath().expressions(validationMap));
```

The above JsonPath expressions will be evaluated when Citrus validates the received message. The expression result is compared to the expected value where expectations can be static values as well as test variables and validation matcher expressions. In case a JsonPath expression should not be able to find any elements the test case will also fail.

Json is a pretty simple yet powerful message format. Simply put, a Json message just knows JsonObject, JsonArray and JsonValue items. The handling of JsonObject and JsonValue items in JsonPath expressions is straight forward. We just use a dot notated syntax for walking through the JsonObject hierarchy. The handling of JsonArray items is also not very difficult either. Citrus will try the best to convert JsonArray items to String representation values for comparison.

> JsonPath expressions will only work on Json message formats. This is why we have to tell Citrus the correct message format. By default, Citrus is working with XML message data and therefore the XML validation mechanisms do apply by default. With the message type attribute set to **json** we make sure that Citrus enables Json specific features on the message validation such as JsonPath support.

Now let's get a bit more complex with validation matchers and Json object functions. Citrus tries to give you the most comfortable validation capabilities when comparing Json object values and Json arrays. One first thing you can use is object functions like **keySet()** or **size()** . This functionality is not covered by JsonPath out of the box but added by Citrus. See the following example on how to use it:

Java

```
receive(someEndpoint)
    .message()
    .type(MessageType.JSON)
    .validate(jsonPath()
        .expression("$.user.keySet()", "[id,name,admin,projects]")
        .expression("$.user.aliases.size()", "3"));
```

XML

```
<receive endpoint="someEndpoint">
  <message type="json">
    <validate>
      <json-path expression="$.user.keySet()" value="[id,name,admin,projects]"/>
      <json-path expression="$.user.aliases.size()" value="3"/>
    </validate>
  </message>
</receive>
```

The object functions do return special Json object related properties such as the set of **keys** for an object or the size of a Json array.

Now let's get even more comfortable validation capabilities with matchers. Citrus supports Hamcrest matchers which gives us a very powerful way of validating Json object elements and arrays. See the following examples that demonstrate how this works:

Java

```
receive(someEndpoint)
    .message()
    .type(MessageType.JSON)
    .validate(jsonPath()
                .expression("$.user.keySet()",
contains("id","name","admin","projects"))
                .expression("$.user.aliases.size()", allOf(greaterThan(0),
lessThan(5))));
```

XML

```
<receive endpoint="someEndpoint">
  <message type="json">
    <validate>
      <json-path expression="$.user.keySet()"
value="@assertThat(contains(id,name,admin,projects))@"/>
      <json-path expression="$.user.aliases.size()"
value="@assertThat(allOf(greaterThan(0), lessThan(5)))@"/>
    </validate>
  </message>
</receive>
```

When using the XML DSL we have to use the **assertThat** validation matcher syntax for defining the Hamcrest matchers. You can combine matcher implementation as seen in the **allOf(greaterThan(0), lessThan(5))** expression. When using the Java DSL you can just add the matcher as expected result object. Citrus evaluates the matchers and makes sure everything is as expected. This is a very powerful validation mechanism as it combines the Hamcrest matcher capabilities with Json message validation.

7.3.3. Json schema validation

The Json schema validation in Citrus is based on the drafts provided by json-schema.org. Because Json schema is a fast evolving project, only Json schema V3 and V4 are currently supported.

> In contrast to the XML validation, the Json validation is an optional feature. You have to activate it within every receive-message action by setting `schema-validation="true"`

Java

```
receive(someEndpoint)
    .message()
    .type(MessageType.JSON)
    .body()
    .validate(json()
        .schemaValidation(true)
        .schema("bookStore"));
```

XML

```
<receive endpoint="echoHttpServer">
  <message type="json" schema="bookStore" schema-validation="true">
    <data>
      {
        "isbn" : "0345391802",
        "title": "The Hitchhiker's Guide to the Galaxy",
        "author": "Douglas Adams"
      }
    </data>
  </message>
</receive>
```

Json schema validation in Citrus is optional and disabled by default. This is why the action required to explicitly enable the schema validation with `schemaValidation(true)`. The schema references a bean in the Citrus context (e.g. a Spring bean in the application context). Read more about how to declare schemas in schema validation.

We encourage you to add Json schema validation to your test cases as soon as possible, because we think that message validation is an important part of integration testing.

7.3.4. Json schema repositories

Because Citrus supports different types of schema repositories, it is necessary to declare a Json schema repository as `type="json"`. This allows Citrus to collect all Json schema files for the message validation.

Java

```
@Bean
public JsonSchemaRepository schemaRepository() {
    JsonSchemaRepository repository = new JsonSchemaRepository();
    repository.getSchemas().add(productSchema());
    return repository;
}
```

XML

```
<citrus:schema-repository type="json" id="jsonSchemaRepository">
    <citrus:schemas>
        <citrus:schema ref="productSchema"
location="classpath:org/citrusframework/validation/ProductsSchema.json"/>
    </citrus:schemas>
</citrus:schema-repository>
```

The referenced schema is another bean in the configuration that represents the schema definition.

Java

```
@Bean
public SimpleJsonSchema productSchema() {
    return new SimpleJsonSchema(
            new
ClassPathResource("classpath:org/citrusframework/validation/ProductsSchema.json"));
}
```

XML

```
<citrus:schema id="productSchema"
location="classpath:org/citrusframework/validation/ProductsSchema.json"/>
```

7.3.5. Json schema filtering and validation strategy

In reference to the current Json schema definition, it is not possible to create a direct reference between a Json message and a set of schemas, as it would be possible with XML namespaces. Because of that, Citrus follows a rule set for choosing the relevant schemas based on the configuration within the test case in relation to the given context. The following table assumes that the Json schema validation is activated for the test action.

Scenario	Validation rules
No Json schema repositories are defined in the context.	No Json schema validation applies.
There is at least one Json schema repository defined in the context.	The message of the test action must be valid regarding at least one of the available schemas within the context.
A schema overruling is configured in the test case.	The configured schema must exist and the message must be valid regarding the specified schema.
A schema repository overruling is configured in the test case.	The configured schema repository must exist and the message must be valid regarding at least one of the schemas within the specified schema repository.

7.4. XML validation

XML is a very common message format especially in the SOAP WebServices and JMS messaging world. Citrus provides XML message validator implementations that are able to compare XML message structures. The validator will notice differences in the XML message and supports XML namespaces, attributes and XML schema validation.

XML message validation is not enabled by default in your project. You need to add the validation module to your project as a Maven dependency.

XML validation module dependency

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-validation-xml</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

The default XML message validator implementation is active by default and can be overwritten with a custom implementation using the bean id **defaultXmlMessageValidator** .

Java

```
@Bean
public DomXmlMessageValidator defaultXmlMessageValidator() {
    return new DomXmlMessageValidator();
}
```

XML

```
<bean id="defaultXmlMessageValidator"
class="org.citrusframework.validation.xml.DomXmlMessageValidator"/>
```

The default XML message validator is very powerful when it comes to compare XML structures. The validator supports namespaces with different prefixes and attributes as well as namespace qualified attributes. See the following sections for a detailed description of all capabilities.

7.4.1. XML payload validation

Citrus is able to verify XML message content on a received message. You as a tester can compare the whole XML message body to a predefined control message template. The Citrus message validator will walk through the XML document and compare the elements, attributes and values.

You can define the expected XML message template in multiple ways:

- Defines the message body as nested XML message template in the test code. The whole message body is defined inside the test case.
- Defines an expected XML message template via external file resources. This time the body

content is loaded at runtime from the external file.

In Java tests that use the Citrus domain specific language you must use verbose String concatenation when constructing XML message contents inline. You need to escape reserved characters like quotes and line breaks. This is why you should consider to using external file resources in Java when dealing with large complex message data.

Java

```
@CitrusTest
public void externalBodyResourceTest() {
    receive("someEndpoint")
        .message()
        .body(new
ClassPathResource("org/citrusframework/message/data/TestRequest.xml"))
        .header("Operation", "sayHello")
        .header("MessageId", "${messageId}");
}
```

XML

```
<receive endpoint="someEndpoint">
  <message>
    <payload file="classpath:org/citrusframework/message/data/TestRequest.xml"/>
  </message>
</receive>
```

Inline message body definition or external file resource give us a control message template that the test case expects to validate. Citrus uses this control template for extended message comparison. All elements, namespaces, attributes and node values are validated in this comparison. When using XML message bodies Citrus will navigate through the whole XML structure validating each element and its content.

Only in case received message and control message are equal to each other as expected the message validation will pass. In case differences occur the test case fails with detailed error message.

Citrus supports various ways to add dynamic message content in the message template. Secondly, Citrus can ignore some elements that should not be part of message comparison (e.g. when generated content or timestamps are part of the message content). The tester can enrich the expected message template with test variables or ignore-expressions so we get a more robust validation mechanism. We will talk about this in the next sections to come.

7.4.2. XML header validation

A message can have multiple headers in addition to the body content. You can validate headers with expected name and the control value. Just add the following header validation to your receiving action.

Java

```
receive("someEndpoint")
    .message()
    .header("Operation", "sayHello")
    .header("MessageId", "${messageId}");
```

XML

```
<receive endpoint="someEndpoint">
  <message>
    <header>
        <element name="Operation" value="GetCustomer"/>
        <element name="RequestTag" value="${requestTag}"/>
    </header>
  </message>
</receive>
```

Message headers are represented as name-value pairs. Each expected header element identified by its name has to be present in the received message. In addition to that the header value is compared to the given control value. If a header entry is not available, or the value does not match the expectations the test raises validation errors.

7.4.3. XML header fragment validation

Sometimes message headers may not apply to the name-value pair pattern. For example SOAP headers can also contain XML fragments as header values. You can add complex header data as expected value in the validation.

Java

```
receive("someEndpoint")
    .message()
    .header("Operation", "SayHello")
    .header("<ns0:HelloHeader
xmlns:ns0=\"http://citrusframework.org/schemas/samples/HelloService.xsd\">" +
                "<ns0:MessageId>${messageId}</ns0:MessageId>" +
                "<ns0:CorrelationId>${correlationId}</ns0:CorrelationId>" +
                "<ns0:User>${user}</ns0:User>" +
                "<ns0:Text>Hello from Citrus!</ns0:Text>" +
            "</ns0:HelloHeader>");
```

XML

```
<receive endpoint="someEndpoint">
  <message>
    <header>
      <data>
        <![CDATA[
          <ns0:HelloHeader
xmlns:ns0="http://citrusframework.org/schemas/samples/HelloService.xsd">
              <ns0:MessageId>${messageId}</ns0:MessageId>
              <ns0:CorrelationId>${correlationId}</ns0:CorrelationId>
              <ns0:User>${user}</ns0:User>
              <ns0:Text>Hello from Citrus!</ns0:Text>
          </ns0:HelloHeader>
        ]]>
      </data>
      <element name="Operation" value="SayHello"/>
    </header>
  </message>
</receive>
```

The header data has no name but uses a complex XML fragment as a value. In SOAP this header fragment will be added as a `SOAP-ENV:Header` then. Please read more about this in SOAP support.

7.4.4. Ignore XML elements

Some elements in the message payload might not apply for validation at all. Just think of communication timestamps or dynamic values that have been generated from a foreign service.

You as a tester may not be able to predict such a timestamp or dynamic value for expected validation. This is why you can safely ignore elements and attributes in the XML message validation.

Java

```
receive("someEndpoint")
    .message()
    .header("<TestMessage>" +
                "<VersionId>${versionId}</VersionId>" +
                "<Timestamp>?</Timestamp>" +
                "<MessageId>?</MessageId>" +
            "</TestMessage>")
    .validate(xpath()
            .ignore("/TestMessage/Timestamp")
            .ignore("/TestMessage/MessageId"));
```

XML

```
<receive endpoint="someEndpoint">
  <message>
    <payload>
      <TestMessage>
        <VersionId>${versionId}</VersionId>
        <Timestamp>?</Timestamp>
        <MessageId>?</MessageId>
      </TestMessage>
    </payload>
    <ignore path="/TestMessage/Timestamp"/>
    <ignore path="/TestMessage/MessageId"/>
  </message>
</receive>
```

The receive action above is not able to verify the elements `Timestamp` and `MessageId`. This is because the timestamp uses milliseconds and the message id has been generated by the server application. Both values must be excluded from XML validation.

You can use ignore XPath expressions that match elements in the message content that should be excluded. XPath expressions can be cumbersome and error prone though.

You can also use inline **@ignore@** expressions as expected template values in order to exclude elements from valdidation. This is for those of you that do not like to write XPath expressions. As a result the ignored message elements are automatically skipped when Citrus compares and validates message contents and do not break the test case.

Java

```
receive("someEndpoint")
    .message()
    .header("<TestMessage>" +
                "<VersionId>${versionId}</VersionId>" +
                "<Timestamp>@ignore@</Timestamp>" +
                "<MessageId>@ignore@</MessageId>" +
            "</TestMessage>");
```

XML

```
<receive endpoint="someEndpoint">
  <message>
    <payload>
      <TestMessage>
        <VersionId>${versionId}</VersionId>
        <Timestamp>@ignore@</Timestamp>
        <MessageId>@ignore@</MessageId>
      </TestMessage>
    </payload>
  </message>
</receive>
```

Feel free to mix both mechanisms to ignore message elements. Ignore expressions are valid as elements, sub-tree nodes and attributes. You can use the **@ignore@** placeholder in external file resources, too.

7.4.5. XPath validation

The section XML payload validation showed how to validate the complete XML message structure with control message template. All elements are validated and compared one after another.

In some cases this approach might be too extensive. Imagine the tester only needs to validate a small subset of message elements. You would rather want to use explicit element validation with XPath.

Java

```
receive("someEndpoint")
    .message()
    .validate(xpath()
        .expression("/TestRequest/MessageId", "${messageId}")
        .expression("//VersionId", "2"));
}
```

XML

```
<receive endpoint="someEndpoint">
  <message>
    <validate>
      <xpath expression="/TestRequest/MessageId" value="${messageId}"/>
      <xpath expression="/TestRequest/VersionId" value="2"/>
    </validate>
  </message>
</receive>
```

In Java the use of a map may be the easiest way to declare multiple expressions for XPath validation.

Java DSL

```
final Map<String, Object> expressions = new HashMap<>();
expressions.put("/TestRequest/MessageId", "${messageId}");
expressions.put("//VersionId", "2");

receive("someEndpoint")
    .message()
    .validate(xpath()
        .expressions(expressions));
}
```

Instead of comparing the whole message some message elements are validated explicitly via XPath. Citrus evaluates the XPath expression on the received message and compares the result value to the control value. The basic message structure as well as all other message elements are not included into this explicit validation.

If this type of element validation is chosen neither <payload> nor <data> nor <resource> template definitions are allowed in Citrus XML test cases.

Citrus offers an alternative dot-notated syntax in order to walk through XML trees. In case you are not familiar with XPath or simply need a very easy way to find your element inside the XML tree you might use this way. Every element hierarchy in the XML tree is represented with a simple dot - for example:

`TestRequest.VersionId`

The expression will search the XML tree for the respective `<TestRequest><VersionId>` element. Attributes are supported too. In case the last element in the dot-notated expression is an XML attribute the framework will automatically find it.

Of course this dot-notated syntax is very simple and might not be applicable for more complex tree navigation. XPath is much more powerful - no doubt. The dot-notated syntax might help those of you that are not familiar with XPath. So the dot-notation is supported wherever XPath expressions might apply.

The Xpath expressions can evaluate to different result types. By default, Citrus is operating on **NODE** and **STRING** result types so that you can validate some element value. But you can also use different result types such as **NODESET** and **BOOLEAN** .

Java

```
receive("someEndpoint")
    .validate(xpath()
        .expression("boolean:/TestRequest/Error", false)
        .expression("number:/TestRequest/Status[.='success']", 3)
        .expression("node-set:/TestRequest/OrderType", "[single, multi, multi]");
```

XML

```
<receive endpoint="someEndpoint">
  <message>
    <validate>
      <xpath expression="/TestRequest/Error" value="false" result-type="boolean"/>
      <xpath expression="/TestRequest/Status[.='success']" value="3" result-
type="number"/>
      <xpath expression="/TestRequest/OrderType" value="[single, multi, multi]"
result-type="node-set"/>
    </validate>
  </message>
</receive>
```

In the example above we use different expression result types. First we want to make sure no **/TestRequest/Error** element is present. This can be done with a boolean result type and **false** value. Second we want to validate the number of found elements for the expression **/TestRequest/Status[.='success']** . The XPath expression evaluates to a node list that results in its list size to be checked. And last but not least we evaluate to a **node-set** result type where all values in the node list will be translated to a comma delimited string value.

You can use even more powerful validation expressions with matcher implementations. With validation matchers you are able to use validations such as **greaterThan**, **lessThan**, **hasSize** and much more.

Java

```
receive("someEndpoint")
    .validate(xpath()
        .expression("/TestRequest/Error", anyOf(empty(), nullValue()))
        .expression("number:/TestRequest/Status[.='success']", greaterThan(0.0))
        .expression("integer:/TestRequest/Status[.='failed']", lowerThan(1))
        .expression("node-set:/TestRequest/OrderType", hasSize(3));
```

XML

```
<receive endpoint="someEndpoint">
  <message>
    <validate>
      <xpath expression="/TestRequest/Error" value="@assertThat(anyOf(empty(),
nullValue()))@"/>
      <xpath expression="/TestRequest/Status[.='success']"
value="@assertThat(greaterThan(0.0))@" result-type="number"/>
      <xpath expression="/TestRequest/Status[.='failed']"
value="@assertThat(lowerThan(1))@" result-type="integer"/>
      <xpath expression="/TestRequest/OrderType" value="@assertThat(hasSize(3))@"
result-type="node-set"/>
    </validate>
  </message>
</receive>
```

The validation matchers used in the example above use the citrus-hamcrest-validation module.

XPath uses decimal number type **Double** by default when evaluating expressions with **number** result type. This means we have to use Double typed expected values, too. Citrus also provides the result type **integer** that automatically converts the XPath expression result to an **Integer** type.

When using the XML DSL we have to use the **assertThat** validation matcher syntax for defining the Hamcrest matcher. You can combine matcher implementation as seen in the **anyOf(empty(), nullValue())** expression. When using the Java DSL you can just add the matcher as expected result object. Citrus evaluates the matchers and makes sure everything is as expected. This is a very powerful validation mechanism as it also works with node-sets containing multiple values as list.

This is how you can add very powerful message element validation in XML using XPath expressions.

7.4.6. XML namespaces

Namespaces represent an essential concept in XML. A namespace declares an element to be part of a very specific ruleset. You have to specify namespaces also when using XPath expressions. Let's have a look at an example message that uses XML namespaces:

Sample XML body with namepaces

```
<ns1:TestMessage xmlns:ns1="http://citrus.com/namespace">
    <ns1:TestHeader>
        <ns1:CorrelationId>_</ns1:CorrelationId>
        <ns1:Timestamp>2001-12-17T09:30:47.0Z</ns1:Timestamp>
        <ns1:VersionId>2</ns1:VersionId>
    </ns1:TestHeader>
    <ns1:TestBody>
        <ns1:Customer>
            <ns1:Id>1</ns1:Id>
        </ns1:Customer>
    </ns1:TestBody>
</ns1:TestMessage>
```

Now we would like to validate some elements in this message using XPath

Java

```
receive("someEndpoint")
    .validate(xpath()
        .expression("//TestMessage/TestHeader/VersionId", 2L)
        .expression("//TestMessage/TestHeader/CorrelationId", "${correlationId}");
```

XML

```
<receive endpoint="someEndpoint">
  <message>
    <validate>
      <xpath expression="//TestMessage/TestHeader/VersionId" value="2"/>
      <xpath expression="//TestMessage/TestHeader/CorrelationId"
value="${correlationId}"/>
    </validate>
  </message>
</receive>
```

The validation will fail although the XPath expression looks correct regarding the XML tree. This is because the message uses the namespace `xmlns:ns1="http://citrus.com/namespace"`. The XPath expression is not able to find the elements because of the missing namespace declaration in the expression. The correct XPath expression uses the namespace prefix as defined in the message.

Java

```
receive("someEndpoint")
    .validate(xpath()
        .expression("//ns1:TestMessage/ns1:TestHeader/ns1:VersionId", 2L)
        .expression("//ns1:TestMessage/ns1:TestHeader/ns1:CorrelationId",
"${correlationId}");
```

XML

```
<receive endpoint="someEndpoint">
  <message>
    <validate>
      <xpath expression="//ns1:TestMessage/ns1:TestHeader/ns1:VersionId" value="2"/>
      <xpath expression="//ns1:TestMessage/ns1:TestHeader/ns1:CorrelationId"
value="${correlationId}"/>
    </validate>
  </message>
</receive>
```

Now the expressions works fine, and the validation is successful. Relying on the namespace prefix `ns1` is quite error-prone though. This is because the test depends on the very specific namespace prefix. As soon as the message is sent with a different namespace prefix (e.g. ns2) the validation will fail again.

You can avoid this effect when specifying your own namespace context and your own namespace prefix during validation.

Java

```
receive("someEndpoint")
    .validate(xpath()
        .expression("//pfx:TestMessage/pfx:TestHeader/pfx:VersionId", 2L)
        .expression("//pfx:TestMessage/pfx:TestHeader/pfx:CorrelationId",
"${correlationId}")
        .namespaceContext("pfx", "http://citrus.com/namespace"));
```

XML

```
<receive endpoint="someEndpoint">
  <message>
    <validate>
      <xpath expression="//pfx:TestMessage/pfx:TestHeader/pfx:VersionId" value="2"/>
      <xpath expression="//pfx:TestMessage/pfx:TestHeader/pfx:CorrelationId"
value="${correlationId}"/>
      <namespace prefix="pfx" value="http://citrus.com/namespace"/>
    </validate>
  </message>
</receive>
```

Now the test is independent of any namespace prefix in the received message. The namespace context will resolve the namespaces and find the elements although the message might use different prefixes. The only thing that matters is that the namespace value (http://citrus.com/namespace) matches.

Instead of this namespace context on validation level you can also have a global namespace context which is valid in all test cases. We just add a bean in the basic Spring application context configuration which defines global namespace mappings.

Java

```
@Bean
public NamespaceContextBuilder namespaceContext() {
    NamespaceContextBuilder builder = new NamespaceContextBuilder();
    builder.getNamepspaceMappings().put("pfx",
"http://citrusframework.org/samples/sayHello");
    return builder;
}
```

XML

```
<namespace-context>
    <namespace prefix="def" uri="http://citrusframework.org/samples/sayHello"/>
</namespace-context>
```

Once defined the **def** namespace prefix is valid in all test cases and all XPath expressions. This enables you to free your test cases from namespace prefix bindings that might be broken with time. You can use these global namespace mappings wherever XPath expressions are valid inside a test case (validation, ignore, extract).

In the previous section we have seen that XML namespaces can get tricky with XPath validation. Default namespaces can do even more! So let's look at the example with default namespaces:

Sample XML body with default namespaces

```
<TestMessage xmlns="http://citrus.com/namespace">
    <TestHeader>
        <CorrelationId>_</CorrelationId>
        <Timestamp>2001-12-17T09:30:47.0Z</Timestamp>
        <VersionId>2</VersionId>
    </TestHeader>
    <TestBody>
        <Customer>
            <Id>1</Id>
        </Customer>
    </TestBody>
</TestMessage>
```

The message uses default namespaces. The following approach in XPath will fail due to namespace problems.

Java

```
receive("someEndpoint")
    .validate(xpath()
        .expression("//TestMessage/TestHeader/VersionId", 2L)
        .expression("//TestMessage/TestHeader/CorrelationId", "${correlationId}"));
```

XML

```
<receive endpoint="someEndpoint">
  <message>
    <validate>
      <xpath expression="//TestMessage/TestHeader/VersionId" value="2"/>
      <xpath expression="//TestMessage/TestHeader/CorrelationId"
value="${correlationId}"/>
    </validate>
  </message>
</receive>
```

Even default namespaces need to be specified in the XPath expressions. Look at the following code listing that works fine with default namespaces:

Java

```
receive("someEndpoint")
    .validate(xpath()
        .expression("//:TestMessage/:TestHeader/:VersionId", 2L)
        .expression("//:TestMessage/:TestHeader/:CorrelationId", "${correlationId}"));
```

XML

```
<receive endpoint="someEndpoint">
  <message>
    <validate>
      <xpath expression="//:TestMessage/:TestHeader/:VersionId" value="2"/>
      <xpath expression="//:TestMessage/:TestHeader/:CorrelationId"
value="${correlationId}"/>
    </validate>
  </message>
</receive>
```

It is recommended to use the namespace context as described in the previous chapter when validating. Only this approach ensures flexibility and stable test cases regarding namespace changes.

7.4.7. Customize XML parser and serializer

When working with XML data format parsing and serializing is a common task. XML structures are

parsed to a DOM (Document Object Model) representation in order to process elements, attributes and text nodes. DOM node objects get serialized to a String message payload representation. The XML parser and serializer is customizable to a certain level. By default, Citrus uses the DOM Level 3 Load and Save implementation with following settings:

Parser settings

cdata-sections	**true**
split-cdata-sections	**false**
validate-if-schema	**true**
element-content-whitespace	**false**

Serializer settings

format-pretty-print	**true**
split-cdata-sections	**false**
element-content-whitespace	**true**

The parameters are also described in W3C DOM configuration documentation. We can customize the default settings by adding an *XmlConfigurer* Spring bean to the Citrus application context.

Java

```
@Bean
public XmlConfigurer xmlConfigurer() {
    XmlConfigurer configurer = new XmlConfigurer();
    configurer.getParseSettings().put("validate-if-schema", false);

    configurer.getSerializeSettings().put("comments", false);
    configurer.getSerializeSettings().put("format-pretty-print", false);
    return configurer;
}
```

XML

```
<bean id="xmlConfigurer" class="org.citrusframework.xml.XmlConfigurer">
    <property name="parseSettings">
        <map>
            <entry key="validate-if-schema" value="false" value-
type="java.lang.Boolean"/>
        </map>
    </property>
    <property name="serializeSettings">
        <map>
            <entry key="comments" value="false" value-type="java.lang.Boolean"/>
            <entry key="format-pretty-print" value="false" value-
type="java.lang.Boolean"/>
        </map>
    </property>
</bean>
```

This configuration is of global nature. All XML processing operations will be affected with this configuration.

7.4.8. Groovy XML validation

With the Groovy XmlSlurper you can easily validate XML message payloads without having to deal directly with XML. People who do not want to deal with XPath may also like this validation alternative.

The tester directly navigates through the message elements and uses simple code assertions in order to control the message content. Here is an example how to validate messages with Groovy script:

Java

```
receive("someEndpoint")
    .validate(groovy().script("assert root.children().size() == 4\n" +
                              "assert root.MessageId.text() == '${messageId}'\n" +
                              "assert root.CorrelationId.text() ==
'${correlationId}'\n")
                              "assert root.Text.text() == 'Hello ' +
context.getVariable(\"user\")"))
    .header("Operation", "sayHello")
    .header("CorrelationId", "${correlationId}")
    .timeout(5000L);
```

XML

```
<receive endpoint="someEndpoint" timeout="5000">
    <message>
        <validate>
            <script type="groovy">
                assert root.children().size() == 4
                assert root.MessageId.text() == '${messageId}'
                assert root.CorrelationId.text() == '${correlationId}'
                assert root.Text.text() == 'Hello ' + context.getVariable("user")
            </script>
        </validate>
    </message>
    <header>
        <element name="Operation" value="sayHello"/>
        <element name="CorrelationId" value="${correlationId}"/>
    </header>
</receive>
```

The Groovy XmlSlurper validation script goes right into the message-tag instead of an XML control template or XPath validation. The Groovy script supports Java ***assert*** statements for message element validation. Citrus automatically injects the root element ***root*** to the validation script. This is the Groovy XmlSlurper object and the start of element navigation. Based on this root element you can access child elements and attributes with a dot notated syntax. Just use the element names separated by a simple dot. Very easy! If you need the list of child elements use the ***children()*** function on any element. With the ***text()*** function you get access to the element's text-value. The ***size()*** is very useful for validating the number of child elements which completes the basic validation statements.

As you can see from the example, we may use test variables within the validation script, too. Citrus has also injected the actual test context to the validation script. The test context object holds all test variables. So you can also access variables with ***context.getVariable("user")*** for instance. On the test context you can also set new variable values with ***context.setVariable("user", "newUserName")*** .

There is even more object injection for the validation script. With the automatically added object ***receivedMessage*** You have access to the Citrus message object for this receive action. This enables you to do whatever you want with the message payload or header.

Java

```
receive("someEndpoint")
    .validate(groovy().script("assert
receivedMessage.getPayload(String.class).contains(\"Hello Citrus!\")\n" +
                              "assert receivedMessage.getHeader(\"Operation\") ==
'sayHello'\n" +
                              "context.setVariable(\"request_payload\",
receivedMessage.getPayload(String.class))"))
    .timeout(5000L);
```

XML

```
<receive endpoint="someEndpoint" timeout="5000">
    <message>
        <validate>
            <script type="groovy">
                assert receivedMessage.getPayload(String.class).contains("Hello
Citrus!")
                assert receivedMessage.getHeader("Operation") == 'sayHello'

                context.setVariable("request_payload",
receivedMessage.getPayload(String.class))
            </script>
        </validate>
    </message>
</receive>
```

The listing above shows some power of the validation script. We can access the message payload, we can access the message header. With test context access we can also save the whole message payload as a new test variable for later usage in the test.

In general Groovy code inside the XML test case definition or as part of the Java DSL code is not very comfortable to maintain. You do not have code syntax assist or code completion. This is why we can also use external file resources for the validation scripts. The syntax looks like follows:

Java

```
receive("someEndpoint")
    .validate(groovy()
            .script(new ClassPathResource("validationScript.groovy"))
    .timeout(5000L);
```

XML

```
<receive endpoint="someEndpoint" timeout="5000">
    <message>
        <validate>
            <script type="groovy" file="classpath:validationScript.groovy"/>
        </validate>
    </message>
</receive>
```

We referenced some external file resource ***validationScript.groovy*** . This file content is loaded at runtime and is used as script body. Now that we have a normal groovy file we can use the code completion and syntax highlighting of our favorite Groovy editor.

> You can use the Groovy validation script in combination with other validation types like XML tree comparison and XPath validation.

For further information on the Groovy XmlSlurper please see the official Groovy website and documentation

7.4.9. XML schema validation

There are several possibilities to describe the structure of XML documents. The two most popular ways are DTD (Document type definition) and XSD (XML Schema definition). In case the XML document is classified using a schema definition the structure of the document has to fit the predefined rules inside the schema definition. XML document instances are valid only in case they meet all these structure rules defined in the schema definition. Citrus can validate XML documents using the schema languages DTD and XSD.

XSD schema repositories

Citrus tries to validate all incoming XML messages against a schema definition in order to ensure that all rules are fulfilled. This means that the message receiving actions in Citrus have to know the XML schema definition file resources that belong to our test context.

Java

```
@Bean
public XsdSchemaRepository schemaRepository() {
    XsdSchemaRepository repository = new XsdSchemaRepository();
    repository.getSchemas().add(travelAgencySchema());
    repository.getSchemas().add(royalAirlineSchema());
    repository.getSchemas().add(smartAirlineSchema());
    return repository;
}

@Bean
public SimpleXsdSchema travelAgencySchema() {
    return new SimpleXsdSchema(
            new
ClassPathResource("classpath:citrus/flightbooking/TravelAgencySchema.xsd"));
}

@Bean
public SimpleXsdSchema royalAirlineSchema() {
    return new SimpleXsdSchema(
            new
ClassPathResource("classpath:citrus/flightbooking/RoyalAirlineSchema.xsd"));
}

@Bean
public SimpleXsdSchema smartAirlineSchema() {
    return new SimpleXsdSchema(
            new
ClassPathResource("classpath:citrus/flightbooking/SmartAirlineSchema.xsd"));
}
```

XML

```
<citrus:schema-repository id="schemaRepository">
    <citrus:schemas>
        <citrus:schema id="travelAgencySchema"
            location="classpath:citrus/flightbooking/TravelAgencySchema.xsd"/>
        <citrus:schema id="royalAirlineSchema"
            location="classpath:citrus/flightbooking/RoyalAirlineSchema.xsd"/>
        <citrus:reference schema="smartAirlineSchema"/>
    </citrus:schemas>
</citrus:schema-repository>

<citrus:schema id="smartAirlineSchema"
      location="classpath:citrus/flightbooking/SmartAirlineSchema.xsd"/>
```

By convention there is a default schema repository component defined in the Citrus Spring application context with the id **schemaRepository**. Spring application context is then able to inject the schema repository into all message receiving test actions at runtime. The receiving test action consolidates the repository for a matching schema definition file in order to validate the incoming XML document structure.

The connection between incoming XML messages and xsd schema files in the repository is done by a mapping strategy which we will discuss later in this chapter. By default, Citrus picks the right schema based on the target namespace that is defined inside the schema definition. The target namespace of the schema definition has to match the namespace of the root element in the received XML message. With this mapping strategy you will not have to wire XML messages and schema files manually, all is done automatically by the Citrus schema repository at runtime. All you need to do is to register all available schema definition files regardless of which target namespace or nature inside the Citrus schema repository.

> XML schema validation is mandatory in Citrus. This means that Citrus always tries to find a matching schema definition inside the schema repository in order to perform syntax validation on incoming schema qualified XML messages. A classified XML message is defined by its namespace definitions. Consequently you will get validation errors in case no matching schema definition file is found inside the schema repository. So if you explicitly do not want to validate the XML schema for some reason you have to disable the validation explicitly in your test with **schema-validation="false"**.

Java

```
receive("someEndpoint")
    .validate(xpath()
            .expression("//ns1:TestMessage/ns1:MessageHeader/ns1:MessageId",
"${messageId}")
            .expression("//ns1:TestMessage/ns1:MessageHeader/ns1:CorrelationId",
"${correlationId}")
            .schemaValidation(false)
            .namespaceContext("ns1", "http://citrus.com/namespace"))
    .timeout(5000L);
```

XML

```
<receive endpoint="someEndpoint">
    <message schema-validation="false">
      <validate>
        <xpath expression="//ns1:TestMessage/ns1:MessageHeader/ns1:MessageId"
             value="${messageId}"/>
        <xpath expression="//ns1:TestMessage/ns1:MessageHeader/ns1:CorrelationId"
             value="${correlationId}"/>
        <namespace prefix="ns1" value="http://citrus.com/namespace"/>
      </validate>
    </message>
</receive>
```

This mandatory schema validation might sound annoying to you but in our opinion it is very important to validate the structure of the received XML messages, so disabling the schema validation should not be the standard for all tests. Disabling automatic schema validation should only apply to very special situations. So please try to put all available schema definitions to the schema repository and you will be fine.

WSDL schemas

In SOAP WebServices world the WSDL (WebService Schema Definition Language) defines the structure and nature of the XML messages exchanged across the interface. Often the WSDL files do hold the XML schema definitions as nested elements. In Citrus you can directly set the WSDL file as location of a schema definition like this:

Java

```
@Bean
public WsdlXsdSchema airlineWsdl() {
    return new WsdlXsdSchema(
            new
ClassPathResource("classpath:citrus/flightbooking/AirlineSchema.wsdl"));
}
```

XML

```
<citrus:schema id="airlineWsdl"
    location="classpath:citrus/flightbooking/AirlineSchema.wsdl"/>
```

Citrus is able to find the nested schema definitions inside the WSDL file in order to build a valid schema file for the schema repository. So incoming XML messages that refer to the WSDL file can be validated for syntax rules.

Schema collections

Sometimes an XML schema definition is separated into multiple files. This is a problem for the Citrus schema repository as the schema mapping strategy then is not able to pick the right file for validation, in particular when working with target namespace values as key for the schema mapping strategy. As a solution for this problem you have to put all schemas with the same target namespace value into a schema collection.

Java

```
@Bean
public XsdSchemaCollection flightbookingSchemaCollection() {
    XsdSchemaCollection collection = new XsdSchemaCollection();
    collection.getSchemas().add("classpath:citrus/flightbooking/BaseTypes.xsd");
    collection.getSchemas().add("classpath:citrus/flightbooking/AirlineSchema.xsd");
    return collection;
}
```

XML

```
<citrus:schema-collection id="flightbookingSchemaCollection">
  <citrus:schemas>
    <citrus:schema location="classpath:citrus/flightbooking/BaseTypes.xsd"/>
    <citrus:schema location="classpath:citrus/flightbooking/AirlineSchema.xsd"/>
  </citrus:schemas>
</citrus:schema-collection>
```

Both schema definitions **BaseTypes.xsd** and **AirlineSchema.xsd** share the same target namespace and therefore need to be combined in schema collection component. The schema collection can be referenced in any schema repository as normal schema definition.

Java

```
@Bean
public XsdSchemaRepository schemaRepository() {
    XsdSchemaRepository repository = new XsdSchemaRepository();
    repository.getSchemas().add(flightbookingSchemaCollection());
    return repository;
}
```

XML

```
<citrus:schema-repository id="schemaRepository">
  <citrus:schemas>
    <citrus:reference schema="flightbookingSchemaCollection"/>
  </citrus:schemas>
</citrus:schema-repository>
```

Schema mapping strategy

The schema repository in Citrus holds one to many schema definition files and dynamically picks up the right one according to the validated message payload. The repository needs to have some strategy for deciding which schema definition to choose. See the following schema mapping strategies and decide which of them is suitable for you.

Target Namespace Mapping Strategy

This is the default schema mapping strategy. Schema definitions usually define some target namespace which is valid for all elements and types inside the schema file. The target namespace is also used as root namespace in XML message payloads. According to this information Citrus can pick up the right schema definition file in the schema repository. You can set the schema mapping strategy as property in the configuration files:

Java

```
@Bean
public XsdSchemaRepository schemaRepository() {
    XsdSchemaRepository repository = new XsdSchemaRepository();
    repository.setSchemaMappingStrategy(schemaMappingStrategy());
    repository.getSchemas().add(helloSchema());
    return repository;
}

@Bean
public TargetNamespaceSchemaMappingStrategy schemaMappingStrategy() {
    return new TargetNamespaceSchemaMappingStrategy();
}

@Bean
public SimpleXsdSchema helloSchema() {
    return new SimpleXsdSchema(
            new ClassPathResource("classpath:citrus/samples/sayHello.xsd"));
}
```

XML

```
<citrus:schema-repository id="schemaRepository"
    schema-mapping-strategy="schemaMappingStrategy">
  <citrus:schemas>
    <citrus:schema id="helloSchema"
        location="classpath:citrus/samples/sayHello.xsd"/>
  </citrus:schemas>
</citrus:schema-repository>

<bean id="schemaMappingStrategy"
    class="org.citrusframework.xml.schema.TargetNamespaceSchemaMappingStrategy"/>
```

The **sayHello.xsd** schema file defines a target namespace (http://citrusframework.org/schemas/sayHello.xsd):

Schema target namespace

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns="http://citrusframework.org/schemas/sayHello.xsd"
    targetNamespace="http://citrusframework.org/schemas/sayHello.xsd"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

</xs:schema>
```

Incoming request messages should also have the target namespace set in the root element and this is how Citrus matches the right schema file in the repository.

HelloRequest.xml

```
<HelloRequest xmlns="http://citrusframework.org/schemas/sayHello.xsd">
   <MessageId>123456789</MessageId>
   <CorrelationId>1000</CorrelationId>
   <User>Christoph</User>
   <Text>Hello Citrus</Text>
</HelloRequest>
```

Root QName Mapping Strategy

The next possibility for mapping incoming request messages to a schema definition is via the XML root element QName. Each XML message payload starts with a root element that usually declares the type of an XML message. According to this root element you can set up mappings in the schema repository.

Java

```
@Bean
public XsdSchemaRepository schemaRepository() {
    XsdSchemaRepository repository = new XsdSchemaRepository();
    repository.setSchemaMappingStrategy(schemaMappingStrategy());
    repository.getSchemas().add(helloSchema());
    repository.getSchemas().add(goodbyeSchema());
    return repository;
}

@Bean
public RootQNameSchemaMappingStrategy schemaMappingStrategy() {
    RootQNameSchemaMappingStrategy rootQnameStrategy = new
RootQNameSchemaMappingStrategy();
    rootQnameStrategy.getMappings().put("HelloRequest", helloSchema());
    rootQnameStrategy.getMappings().put("GoodbyeRequest", goodbyeSchema());

    return rootQnameStrategy;
}

@Bean
public SimpleXsdSchema helloSchema() {
    return new SimpleXsdSchema(
            new ClassPathResource("classpath:citrus/samples/sayHello.xsd"));
}

@Bean
public SimpleXsdSchema goodbyeSchema() {
    return new SimpleXsdSchema(
            new ClassPathResource("classpath:citrus/samples/sayGoodbye.xsd"));
}
```

XML

```
<citrus:schema-repository id="schemaRepository"
    schema-mapping-strategy="schemaMappingStrategy">
  <citrus:schemas>
    <citrus:reference schema="helloSchema"/>
    <citrus:reference schema="goodbyeSchema"/>
  </citrus:schemas>
</citrus:schema-repository>

<bean id="schemaMappingStrategy"
    class="org.citrusframework.xml.schema.RootQNameSchemaMappingStrategy">
  <property name="mappings">
    <map>
      <entry key="HelloRequest" value="helloSchema"/>
      <entry key="GoodbyeRequest" value="goodbyeSchema"/>
    </map>
  </property>
</bean>

<citrus:schema id="helloSchema"
    location="classpath:citrus/samples/sayHello.xsd"/>

<citrus:schema id="goodbyeSchema"
     location="classpath:citrus/samples/sayGoodbye.xsd"/>
```

The listing above defines two root qname mappings - one for **HelloRequest** and one for **GoodbyeRequest** message types. An incoming message of type <HelloRequest> is then mapped to the respective schema and so on. With these dedicated mappings you are able to control which schema is used on an XML request, regardless of target namespace definitions.

Schema mapping strategy chain

Let’s discuss the possibility to combine several schema mapping strategies in a logical chain. You can define more than one mapping strategy that are evaluated in sequence. The first strategy to find a proper schema definition file in the repository wins.

Java

```
@Bean
public XsdSchemaRepository schemaRepository() {
    XsdSchemaRepository repository = new XsdSchemaRepository();
    repository.setSchemaMappingStrategy(schemaMappingStrategy());
    repository.getSchemas().add(helloSchema());
    repository.getSchemas().add(goodbyeSchema());
    return repository;
}

@Bean
public SchemaMappingStrategyChain schemaMappingStrategy() {
    SchemaMappingStrategyChain chain = new SchemaMappingStrategyChain();

    RootQNameSchemaMappingStrategy rootQnameStrategy = new
RootQNameSchemaMappingStrategy();
    rootQnameStrategy.getMappings().put("HelloRequest", helloSchema());

    chain.setStrategies(Arrays.asList(
        rootQnameStrategy,
        new TargetNamespaceSchemaMappingStrategy()
    ));

    return chain;
}
```

XML

```
<citrus:schema-repository id="schemaRepository"
    schema-mapping-strategy="schemaMappingStrategy">
  <citrus:schemas>
    <citrus:reference schema="helloSchema"/>
    <citrus:reference schema="goodbyeSchema"/>
  </citrus:schemas>
</citrus:schema-repository>

<bean id="schemaMappingStrategy"
    class="org.citrusframework.xml.schema.SchemaMappingStrategyChain">
  <property name="strategies">
    <list>
      <bean class="org.citrusframework.xml.schema.RootQNameSchemaMappingStrategy">
        <property name="mappings">
          <map>
            <entry key="HelloRequest" value="helloSchema"/>
          </map>
        </property>
      </bean>
      <bean
class="org.citrusframework.xml.schema.TargetNamespaceSchemaMappingStrategy"/>
    </list>
  </property>
</bean>
```

So the schema mapping chain uses both **RootQNameSchemaMappingStrategy** and **TargetNamespaceSchemaMappingStrategy** in combination. In case the first root qname strategy fails to find a proper mapping the next target namespace strategy comes in and tries to find a proper schema.

DTD validation

XML DTD (document type definition) is another way to validate the structure of an XML document. Many people say that DTD is outdated and XML schema is the much more efficient way to describe the rules of an XML structure. We do not disagree with that, but we also know that legacy systems might still use DTD. So in order to avoid validation errors we have to deal with DTD validation as well.

First thing you can do about DTD validation is to specify an inline DTD in your expected message template.

Java

```
receive("someEndpoint")
    .message()
    .body("<!DOCTYPE root [\n" +
                "<!ELEMENT root (message)>\n" +
                "<!ELEMENT message (text)>\n" +
                "<!ELEMENT text (#PCDATA)>\n" +
            "]>\n" +
            "<root>\n" +
                "<message>\n" +
                    "<text>Hello from Citrus!</text>\n" +
                "</message>\n" +
            "</root>")
    .timeout(5000L);
```

XML

```
<receive endpoint="someEndpoint">
    <message schema-validation="false">
        <data>
        <![CDATA[
            <!DOCTYPE root [
                <!ELEMENT root (message)>
                <!ELEMENT message (text)>
                <!ELEMENT text (#PCDATA)>
                ]>
            <root>
                <message>
                    <text>Hello from Citrus!</text>
                </message>
            </root>
        ]]>
        </data>
    </message>
</receive>
```

The system under test may also send the message with an inline DTD definition. So validation will succeed.

In most cases the DTD is referenced as external .dtd file resource. You can do this in your expected message template as well.

Java

```
receive("someEndpoint")
    .message()
    .body("<!DOCTYPE root SYSTEM \"org/citrusframework/validation/example.dtd\">\n" +
            "<root>\n" +
                "<message>\n" +
                    "<text>Hello from Citrus!</text>\n" +
                "</message>\n" +
            "</root>")
    .timeout(5000L);
```

XML

```
<receive endpoint="someEndpoint">
    <message schema-validation="false">
        <data>
        <![CDATA[
            <!DOCTYPE root SYSTEM "org/citrusframework/validation/example.dtd">
            <root>
                <message>
                    <text>Hello from Citrus!</text>
                </message>
            </root>
        ]]>
        <data/>
    </message>
</receive>
```

7.4.10. XHTML validation

Html message content is hard to verify with XML validation capabilities such as XML tree comparison or XPath support. Usually Html messages do not follow the XML well-formed rules very strictly. This implies that XML message validation will fail because of non-well-formed Html code.

XHTML closes this gap by automatically fixing the most common Html XML incompatible rule violations such as missing end tags (e.g.
).

Please add a new library dependency to your project. Citrus is using the **jtidy** library in order to prepare the HTML and XHTML messages for validation. As this 3rd party dependency is optional in Citrus we have to add it now to our project dependency list. Just add the **jtidy** dependency to your Maven project POM.

Jtidy library

```
<dependency>
    <groupId>net.sf.jtidy</groupId>
    <artifactId>jtidy</artifactId>
    <version>r938</version>
</dependency>
```

Please refer to the **jtidy** project documentation for the latest versions. Now everything is ready. As usual the Citrus message validator for XHTML is active in background by default. You can overwrite this default implementation by placing a Spring bean with id **defaultXhtmlMessageValidator** to the Citrus application context.

Java

```
@Bean
public XhtmlMessageValidator defaultXhtmlMessageValidator() {
    return new XhtmlMessageValidator();
}
```

XML

```
<bean id="defaultXhtmlMessageValidator"
class="org.citrusframework.validation.xhtml.XhtmlMessageValidator"/>
```

Now you can use the XHTML message validation in your test case.

Java

```
receive("someEndpoint")
    .message()
    .type(MessageType.XHTML)
    .body("<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" \"org/w3c/xhtml/xhtml1-
strict.dtd\">" +
            "<html xmlns=\"http://www.w3.org/1999/xhtml\">" +
                "<head>" +
                    "<title>Citrus Hello World</title>" +
                "</head>" +
                "<body>" +
                    "<h1>Hello World!</h1>" +
                    "<br/>" +
                    "<p>This is a test!</p>" +
                "</body>" +
            "</html>")
    .timeout(5000L);
```

XML

```
<receive endpoint="someEndpoint">
    <message type="xhtml">
        <data>
          <![CDATA[
            <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "org/w3c/xhtml/xhtml1-
strict.dtd">
            <html xmlns="http://www.w3.org/1999/xhtml">
              <head>
                <title>Citrus Hello World</title>
              </head>
              <body>
                <h1>Hello World!</h1>
                <br/>
                <p>This is a test!</p>
              </body>
            </html>
          ]]>
        </data>
    </message>
</receive>
```

The message receiving action in our test case has to specify a message format type **type="xhtml"** . As you can see the Html message payload get XHTML specific DOCTYPE processing instruction. The **xhtml1-strict.dtd** is mandatory in the XHTML message validation. For better convenience all XHTML dtd files are packaged within Citrus so you can use this as a relative path.

The incoming Html message is automatically converted into proper XHTML code with well formed XML. So now the XHTML message validator can use the XML message validation mechanism of Citrus for comparing received and expected data. You can also use test variables, ignore element expressions and XPath expressions.

7.5. Schema validation

When structured data is transmitted from one system to another, it is important that both sender and receiver agree on an interface contract. The contract introduces rules of communication for both parties.

Schemas represent the most popular way to define contracts. Citrus is able to manage multiple schemas in the project context. You can define mapping rules to pick the right schema for a message validation.

Let's start with this chapter by introducing some basic concepts of the schema validation.

7.5.1. Schema definition

Complex applications require multiple schemas that are relevant for different use cases. You should organize these schemas in your test project. First you need to add a schema definition that points to

the schema location.

Java

```
@Bean
public SimpleXsdSchema bookstoreSchema() {
    return new SimpleXsdSchema(new
ClassPathResource("classpath:org/citrusframework/xml/BookStore.wsdl"));
}
```

XML

```
<citrus:schema id="bookstoreSchema"
location="classpath:org/citrusframework/xml/BookStore.wsdl"/>
```

Please keep in mind, that the id of the schema has to be unique within the context.

> The samples above are using XML XSD/WSDL schema definitions. The same approach applies to Json schemas, too. You just need to use the `SimpleJsonSchema` class in the Java configuration. The XML configuration components derive the schema type automatically based on the file extension (`.xsd`, `.wsdl`, `.json`).

7.5.2. Schema repository

You can now reference the schema definition in a repository component. The repository is able to hold multiple schema definitions.

Java

```
@Bean
public XsdSchemaRepository schemaRepository() {
    XsdSchemaRepository repository = new XsdSchemaRepository();
    repository.getSchemas().add(bookstoreSchema());
    return repository;
}
```

XML

```
<citrus:schema-repository id="schemaRepository">
  <citrus:schemas>
    <citrus:reference schema="bookstoreSchema" />
  </citrus:schemas>
</citrus:schema-repository>
```

Citrus allows you to reuse your schema definition within your context by referencing them. For a valid reference, the id of the schema and the value of the schema attribute within the reference element have to match.

As an alternative to a schema reference you can also provide the schema location in a repository.

Java

```
@Bean
public XsdSchemaRepository schemaRepository() {
    XsdSchemaRepository repository = new XsdSchemaRepository();
    repository.getLocations().add("classpath:schemas/flightbooking.xsd");
    return repository;
}
```

XML

```
<citrus:schema-repository id="schemaRepository">
  <citrus:locations>
    <citrus:location path="classpath:schemas/flightbooking.xsd"/>
  </citrus:locations>
</citrus:schema-repository>
```

The given location points to the schema definition file. Setting all schemas one by one can be verbose and cumbersome, especially when dealing with lots of schema files. Therefore, Citrus provides schema location patterns which will import all matching schema files within the given location.

Java

```
@Bean
public XsdSchemaRepository schemaRepository() {
    XsdSchemaRepository repository = new XsdSchemaRepository();
    repository.getLocations().add("classpath:schemas/*.xsd");
    return repository;
}
```

XML

```
<citrus:schema-repository id="schemaRepository">
  <citrus:locations>
    <citrus:location path="classpath:schemas/*.xsd"/>
  </citrus:locations>
</citrus:schema-repository>
```

The schema repository is able to receive many schemas with different locations and schema sources.

Java

```
@Bean
public SimpleXsdSchema testSchema() {
    return new SimpleXsdSchema(new
ClassPathResource("classpath:org/citrusframework/xml/test.xsd"));
}

@Bean
public XsdSchemaRepository schemaRepository() {
    SimpleXsdSchema bookstoreSchema = new SimpleXsdSchema(
            new
ClassPathResource("classpath:org/citrusframework/xml/BookStore.wsdl"));

    XsdSchemaRepository repository = new XsdSchemaRepository();
    repository.getSchemas().add(bookstoreSchema);
    repository.getSchemas().add(testSchema());
    repository.getLocations().add("classpath:schemas/*.xsd");
    return repository;
}
```

XML

```
<citrus:schema id="testSchema" location="classpath:org/citrusframework/xml/test.xsd"/>

<citrus:schema-repository id="xmlSchemaRepository">
  <citrus:schemas>
    <citrus:schema id="bookstoreSchema"
location="classpath:org/citrusframework/xml/BookStore.wsdl"/>
    <citrus:reference schema="testSchema"/>
    <citrus:location path="classpath:schemas/*.xsd"/>
  </citrus:schemas>
</citrus:schema-repository>
```

The examples in this chapter have been using XML XSD schema repository components. Of course, the same components are available for Json schema repositories, too. By default, the type of the schema repository is `type=xml`. You can use `type=json` to mark the schema repository as Json nature. In Java configuration please use the `JsonSchemaRepository` class.

The schema repository component holds a set of schema files for a project disjoint by their type (xml, json, etc.) and identified by its unique id.

As you can see the schema repository is a simple bean defined inside the Spring application context. The repository can hold nested schema definitions, references and location definitions for all types of schema repositories.

In case you have several schema repositories in your project do always define a default repository (name="schemaRepository"). This helps Citrus to always find at least one repository to interact with.

7.5.3. Schema definition mapping

Depending on the type of message you want to validate, there are different attempts to find the correct schema for the given message. The XML schema repository will apply a mapping strategy that decides which schema should verify the current message. Citrus knows multiple mapping strategies that you can review in chapter XML schema validation.

As a user you always have the chance to explicitly pick the right schema definition for a `receive` operation. You can overrule all schema mapping strategies in Citrus by directly setting the desired schema in your receiving message action.

Java

```
receive(httpMessageEndpoint)
    .message()
    .validate(
        xml().schema("helloSchema")
    );
```

XML

```
<receive endpoint="httpMessageEndpoint">
    <message schema="helloSchema">
      <payload>
        ...
      </payload>
    </message>
</receive>
```

In the example above the tester explicitly sets a schema definition in the `receive` action (schema="helloSchema"). The schema value refers to named schema bean defined in the project context (e.g. Spring application context).

This overrules all schema mapping strategies used in the central schema repository as the given schema is directly used for validation. This feature is helpful when dealing with different schema versions at the same time.

Another possibility would be to set a custom schema repository at this point. This means you can have more than one schema repository in your Citrus project and you pick the right one by yourself in the `receive` action.

Java

```
receive(httpMessageEndpoint)
    .message()
    .validate(
        xml().schemaRepository("helloSchemaRepository")
    );
```

XML

```
<receive endpoint="httpMessageEndpoint">
    <message schema-repository="helloSchemaRepository">
      <payload>
        ...
      </payload>
    </message>
</receive>
```

The **schema-repository** attribute refers to a Citrus schema repository component which is defined as bean in the project context.

7.6. Plain text validation

Plain text message validation performs an exact Java String match of received and expected message payloads.

Plaintext message validation is not enabled by default in your project. You need to add the validation module to your project as a Maven dependency.

Plaintext validation module dependency

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-validation-text</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

This adds the default message validator for plaintext messages. Citrus will pick this message validator for all messages of **type="plaintext"**. The default message validator implementation can be overwritten by placing a Spring bean with id **defaultPlaintextMessageValidator**.

Java

```
@Bean
public PlainTextMessageValidator defaultPlaintextMessageValidator() {
    return new PlainTextMessageValidator();
}
```

XML

```
<bean id="defaultPlaintextMessageValidator"
class="org.citrusframework.validation.text.PlainTextMessageValidator"/>
```

Citrus will try to auto-guess the appropiate message validator for the incoming message. You can explicitly set the message type on the receive action so Citrus knows how to apply plaintext message validation.

Java

```
receive("someEndpoint")
    .message()
    .type(MessageType.PLAINTEXT)
    .body("Hello World!");
```

XML

```
<receive endpoint="someEndpoint">
    <message type="plaintext">
        <data>Hello World!</data>
    </message>
</receive>
```

With the message format type **type="plaintext"** set Citrus performs String equals on the message payloads (received and expected). Only exact match will pass the test.

Of course test variables are supported in the plaintext payloads. The variables are replaced before processing the message template.

Java

```
receive("someEndpoint")
    .message()
    .type(MessageType.PLAINTEXT)
    .body("${hello} ${world}!");
```

XML

```
<receive endpoint="someEndpoint">
    <message type="plaintext">
        <data>${hello} ${world}!</data>
    </message>
</receive>
```

7.6.1. Whitespace characters

Plaintext message payloads may only differ in system-dependent line separator characters (**CR**, *LF*,

CRLF). By default, the plaintext message validation fails because of that differences even if only whitespace characters are different.

You can disable this default validation behavior and ignore new line types with following system property or environment variable:

Plaintext validation settings

```
citrus.plaintext.validation.ignore.newline.type=true
CITRUS_PLAINTEXT_VALIDATION_IGNORE_NEWLINE_TYPE=true
```

In case you need to ignore all whitespaces during plaintext validation such as multiple new line characters or tabs you need to set this system property or environment variable:

Plaintext validation settings

```
citrus.plaintext.validation.ignore.whitespace=true
CITRUS_PLAINTEXT_VALIDATION_IGNORE_WHITESPACE=true
```

This property will not only ignore new line types but also normalize the whitespaces. As a result all empty lines, tabs and double whitespace characters are filtered before comparison.

Of course, you can also set the properties directly on the plaintext message validator bean:

Java

```
@Bean
public PlainTextMessageValidator defaultPlaintextMessageValidator() {
    PlainTextMessageValidator validatoe = new PlainTextMessageValidator();
    validator.setIgnoreNewLineType(true);
    validator.setIgnoreWhitespace(true);
    return validator;
}
```

XML

```
<bean id="defaultPlaintextMessageValidator"
class="org.citrusframework.validation.text.PlainTextMessageValidator">
  <property name="ignoreNewLineType" value="true"/>
  <property name="ignoreWhitespace" value="true"/>
</bean>
```

7.6.2. Ignoring text parts

The plaintext validator performs a String equals operation. Test variables are automatically replaced before that comparison takes place but what about ignore statements? The plaintext message validator is able to ignore words and character sequences based on their position in the text value. Given a source plaintext value:

Received text

```
Your current id is "1234567890"
```

In the plaintext validation you need to ignore the actual id value due to some reason. Maybe the id is generated on a foreign application and you simply do not know the actual value at runtime.

In this case we can use the common `@ignore@` statement in the control message payload as follows:

Control text

```
Your current id is "@ignore@"
```

Citrus and the plaintext message validator will ignore the marked part of the text during validation. This mechanism is based on the fact that the `@ignore@` statement is placed at the exact same position as the actual id value. So this mechanism requires you to know the exact structure of the plaintext value including all whitespace characters. In case Citrus finds the `@ignore@` keyword in the control value the placeholder is replaced with the actual character sequence that is located at the exact same position in the source message payload that is validated.

The character sequence is defined as sequence of Java word characters. This word sequence is ending with a non-word character defined in Java (`\\W` which is a character that is not in `[a-zA-Z_0-9]`).

Instead of ignoring a single word you can also specify the amount of characters that should be ignored. This is when you have Java non-word characters that you need to ignore. Let's have an example for that, too:

Received text

```
Your current id is "#12345-67890"
```

Given that text the simple `@ignore@` statement will fail because of the non-word characters '#' and '-' that are located in the id value. This time we ignore the whole id sequence with:

Control text

```
Your current id is "@ignore(12)@"
```

This will ignore exactly **12** characters starting from the exact position of the `@ignore@` keyword. So knowing that the id is exactly **12** characters long we can ignore that part.

7.6.3. Creating variables

Instead of just ignoring certain text parts we can also extract those parts into test variables. The actual character sequence is ignored during validation and in addition to that the actual value is stored to a new test variable. Given the following text payload:

Received text

```
Your current id is "1234567890"
```

And the expected control text:

Control text

```
Your current id is "@variable('id')@"
```

The validation will automatically ignore the id part in the text and create a new test variable with name `id` that holds the actual value. The name of the variable to create is given in the `@variable()@` statement. This enables us to extract dynamic text parts that we are not able to validate. After that we can access the dynamic text part using the normal test variable syntax `${id}`.

Also notice that the `@variable()@` keyword expression has to be placed at the exact same position in the text as the actual value. The variable extractor will read the variable value from the source message payload starting from that position. The ending of the variable value is defined by a non-word Java character. Dashes **'-'** and dots **'.'** are automatically included in these values, too. So this will also work for you:

Received text

```
Today is "2017-12-24"
```

And the expected control text:

Control text

```
Today is "@variable('date')@"
```

This results in a new variable called `date` with value `2017-12-24`. In addition, the European date representation works fine here, too because dots and dashes are automatically included:

Received text

```
Today is "24.12.2017"
```

7.6.4. Gzip validation

Gzip is a message compression library to optimize the message transport of large content. Citrus is able to handle compressed message payloads on send and receive operations. Sending compressed data sets the message type to **gzip**.

Java

```
send("someEndpoint")
    .message()
    .type(MessageType.GZIP)
    .body("Hello World!")
```

XML

```
<send endpoint="someEndpoint">
    <message type="gzip">
        <data>Hello World!</data>
    </message>
</send>
```

Just use the **type="gzip"** message type in the send operation. Citrus now converts the message payload to a gzip binary stream as payload.

When validating gzip binary message content the messages are compared with a given control message in binary base64 String representation. The gzip binary data is automatically unzipped and encoded as base64 character sequence in order to compare with an expected content.

The received message content is using gzip format but the actual message content does not have to be base64 encoded. Citrus is doing this conversion automatically before validation takes place. The binary data can be anything e.g. images, pdf or plaintext content.

The default message validator for gzip messages is active by default. Citrus will pick this message validator for all messages of **type="gzip_base64"** . The default message validator implementation can be overwritten by placing a Spring bean with id **defaultGzipBinaryBase64MessageValidator** to the Spring application context.

Java

```
@Bean
public GzipBinaryBase64MessageValidator defaultGzipBinaryBase64MessageValidator() {
    return new GzipBinaryBase64MessageValidator();
}
```

XML

```
<bean id="defaultGzipBinaryBase64MessageValidator"
      class="org.citrusframework.validation.text.GzipBinaryBase64MessageValidator"/>
```

In the test case receiving action we tell Citrus to use gzip message validation.

Java

```
receive("someEndpoint")
    .message()
    .type(MessageType.GZIP_BASE64)
    .body("citrus:encodeBase64('Hello World!')")
```

XML

```
<receive endpoint="someEndpoint">
    <message type="gzip_base64">
        <data>citrus:encodeBase64('Hello World!')</data>
    </message>
</receive>
```

With the message format type **type="gzip_base64"** Citrus performs the gzip base64 character sequence validation. Incoming message content is automatically unzipped and encoded as base64 String and compared to the expected data. This way we can make sure that the binary content is as expected.

> If you are using http client and server components the gzip compression support is built in with the underlying Spring and http commons libraries. So in http communication you just have to set the header **Accept-Encoding=gzip** or **Content-Encoding=gzip**. The message data is then automatically zipped/unzipped before Citrus gets the message data for validation. Read more about this http specific gzip compression in chapter http.

7.7. Binary validation

Binary message validation is not an easy task in particular when it comes to compare data with a given control message.

Binary message validation is not enabled by default in your project. You need to add the validation module to your project as a Maven dependency.

Binary validation module dependency

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-validation-binary</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

There are basically two ways in Citrus how to compare binary message content for validation purpose.

7.7.1. Stream message validation

A first approach to validate incoming binary message content is to compare the binary stream data with an expected stream. This comparison is straight forward as each byte in the binary stream is compared to an expected stream.

The default message validator for binary messages is active by default. Citrus will pick this message validator for all messages of **type="binary_base64"** . The default message validator implementation can be overwritten by placing a Spring bean with id **defaultBinaryBase64MessageValidator** to the Spring application context.

Java

```
@Bean
public BinaryMessageValidator defaultBinaryMessageValidator() {
    return new BinaryMessageValidator();
}
```

XML

```
<bean id="defaultBinaryMessageValidator"
class="org.citrusframework.validation.text.BinaryMessageValidator"/>
```

You can use the binary message type in a receive action in order to enable this stream comparison during validation.

Java

```
.receive("someEndpoint")
    .message(new DefaultMessage(FileCopyUtils.copyToByteArray(new
ClassPathResource("templates/foo.png").getFile())))
    .type(MessageType.BINARY);
}
```

XML

```
<receive endpoint="someEndpoint">
    <message type="binary">
        <resource file="classpath:templates/foo.png"/>
    </message>
</receive>
```

It is very important to set the message type to `MessageType.BINARY` as this is the message type that is automatically handled by the binary stream message validator.

By the way sending binary messages in Citrus is also very easy. Just use the **type="binary"** message type in the send operation. Citrus now converts the message payload to a binary stream as payload.

Java

```
send("someEndpoint")
    .message()
    .type(MessageType.BINARY)
    .body("Hello World")
```

XML

```
<send endpoint="someEndpoint">
    <message type="binary">
        <data>Hello World!</data>
    </message>
</send>
```

7.7.2. Base64 message validation

Another way to validate binary message content is to use base64 String encoding. The binary data is encoded as base64 character sequence and therefore is comparable with an expected content.

The received message content does not have to be base64 encoded. Citrus is doing this conversion automatically before validation takes place. The binary data can be anything e.g. images, pdf or gzip content.

The default message validator for binary messages is active by default. Citrus will pick this message validator for all messages of **type="binary_base64"** . The default message validator implementation can be overwritten by placing a Spring bean with id **defaultBinaryBase64MessageValidator** to the Spring application context.

Java

```
@Bean
public BinaryBase64MessageValidator defaultBinaryBase64MessageValidator() {
    return new BinaryBase64MessageValidator();
}
```

XML

```
<bean id="defaultBinaryBase64MessageValidator"
class="org.citrusframework.validation.text.BinaryBase64MessageValidator"/>
```

In the test case receiving action we tell Citrus to use binary base64 message validation.

Java

```
receive("someEndpoint")
    .message()
    .body("citrus:encodeBase64('Hello World!')")
```

XML

```
<receive endpoint="someEndpoint">
    <message type="binary_base64">
        <data>citrus:encodeBase64('Hello World!')</data>
    </message>
</receive>
```

With the message format type **type="binary_base64"** Citrus performs the base64 character sequence validation. Incoming message content is automatically encoded as base64 String and compared to the expected data. This way we can make sure that the binary content is as expected.

Base64 encoding is also supported in outbound messages. Just use the **encodeBase64** function in Citrus. The result is a base64 encoded String as message payload.

Java

```
send("someEndpoint")
    .message()
    .body("citrus:encodeBase64('Hello World!')")
```

XML

```
<send endpoint="someEndpoint">
    <message>
        <data>citrus:encodeBase64('Hello World!')</data>
    </message>
</send>
```

7.8. Hamcrest validation

Hamcrest validation empowers the validation capabilities by adding special assertions.

Hamcrest message validation capability is not enabled by default in your project. You need to add the validation module to your project as a Maven dependency.

Binary validation module dependency

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-validation-hamcrest</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

You can use the Hamcrest assertions when receiving a message in order to validate elements, for instance with XPath.

Java

```
receive("someEndpoint")
    .validate(xpath()
        .expression("/TestRequest/Error", anyOf(empty(), nullValue()))
        .expression("number:/TestRequest/Status[.='success']", greaterThan(0.0))
        .expression("integer:/TestRequest/Status[.='failed']", lowerThan(1))
        .expression("node-set:/TestRequest/OrderType", hasSize(3));
```

> If you want to match text containing any of the following characters: `'`, `(` `)` You need to enclose the respective string in quotation marks when defining your matcher. If you intend to match an actual single quote, it should be escaped with a backslash (\').

For example:

```
anyOf(equalTo('text containing a \\' (quote) and a , (comma)  '),
anyOf(isEmptyOrNullString()))
anyOf(equalTo('text containing a backslash and quote \\\\' and a , (comma)  '),
anyOf(isEmptyOrNullString()))
```

XML

```
<receive endpoint="someEndpoint">
  <message>
    <validate>
      <xpath expression="/TestRequest/Error" value="@assertThat(anyOf(empty(),
nullValue()))@"/>
      <xpath expression="/TestRequest/Status[.='success']"
value="@assertThat(greaterThan(0.0))@" result-type="number"/>
      <xpath expression="/TestRequest/Status[.='failed']"
value="@assertThat(lowerThan(1))@" result-type="integer"/>
      <xpath expression="/TestRequest/OrderType" value="@assertThat(hasSize(3))@"
result-type="node-set"/>
    </validate>
  </message>
</receive>
```

7.9. Custom validation

In a previous section you have seen how to customize the global set of available message validators in a Citrus project using the validator registry.

7.9.1. Custom message validator

You can also explicitly use a custom message validator in a receive test action. This approach skips the automatic message validator resolving mechanism, and the receive action uses the defined

validator implementation.

Java

```
receive("someEndpoint")
    .validator(groovyJsonMessageValidator)
    .message()
    .type(MessageType.JSON)
    .validate(groovy()
                .script("assert json.type == 'read'\n" +
                  "assert json.mbean == 'java.lang:type=Memory'\n" +
                  "assert json.attribute == 'HeapMemoryUsage'\n" +
                  "assert json.value == '${heapUsage}'");
```

XML

```
<receive endpoint="someEndpoint">
    <message type="json" validator="groovyJsonMessageValidator">
        <validate>
            <script type="groovy">
                <![CDATA[
                  assert json.type == 'read'
                  assert json.mbean == 'java.lang:type=Memory'
                  assert json.attribute == 'HeapMemoryUsage'
                  assert json.value == '${heapUsage}'
                ]]>
            </script>
        </validate>
    </message>
</receive>
```

The receive action defines the message validator to use in this specific use case. You can set a custom message validator implementation here.

> Be careful when overwriting default message validation behavior. Setting a specific message validator has the effect that your receive action may not use the other default validators. The explicit validator definition in a receive action is exclusive so no further message validator resolving is performed on this receive action.

You may want to set multiple validators here in order to meet your validation requirements. For instance when setting a `DomXmlMessageValidator` explicitly you may not be able to use the `XpathMessageValidator` capabilities on that specific receive action anymore. Fortunately you can set multiple validators on a receive action that will all perform validation tasks on the received message.

Java

```
receive("someEndpoint")
    .validators(myXmlMessageValidator, defaultXpathMessageValidator)
    .message()
    .body("...")
    .validate(xpath().expression("//some/xpath/expression", "someControlValue"));
```

XML

```
<receive endpoint="someEndpoint">
    <message type="json"
validators="myXmlMessageValidator,defaultXpathMessageValidator">
        <payload>...</data>
        <validate path="//some/xpath/expression" value="someControlValue"/>
    </message>
</receive>
```

7.9.2. Validation processor

The Java DSL offers some additional validation tricks and possibilities when dealing with messages that are sent and received over Citrus. One of them is the validation processor functionality. With this feature you can marshal/unmarshal message payloads and code validation steps on Java objects.

Validation processor usage

```
receive(bookResponseEndpoint)
    .validate(new XmlMarshallingValidationProcessor<AddBookResponseMessage>() {
        @Override
        public void validate(AddBookResponseMessage response, MessageHeaders headers)
{
            Assert.isTrue(response.isSuccess());
        }
    });
```

By default, the validation processor needs some XML unmarshaller implementation for transforming the XML payload to a Java object. Citrus will automatically search for the unmarshaller bean in your Spring application context if nothing specific is set. Of course, you can also set the unmarshaller instance explicitly.

Use autowired unmarshaller

```
@Autowired
private Unmarshaller unmarshaller;

@CitrusTest
public void receiveMessageTest() {
    receive(bookResponseEndpoint)
        .validate(new
XmlMarshallingValidationProcessor<AddBookResponseMessage>(unmarshaller) {
            @Override
            public void validate(AddBookResponseMessage response, MessageHeaders
headers) {
                Assert.isTrue(response.isSuccess());
            }
        });
}
```

Obviously working on Java objects is much more comfortable than using the XML String concatenation. This is why you can also use this feature when sending messages.

Message body marshalling

```
@Autowired
private Marshaller marshaller;

@CitrusTest
public void sendMessageTest() {
    send(bookRequestEndpoint)
        .message()
        .body(createAddBookRequestMessage("978-citrus:randomNumber(10)"), marshaller)
        .header(SoapMessageHeaders.SOAP_ACTION, "addBook");
}

private AddBookRequestMessage createAddBookRequestMessage(String isbn) {
    AddBookRequestMessage requestMessage = new AddBookRequestMessage();
    Book book = new Book();
    book.setAuthor("Foo");
    book.setTitle("FooTitle");
    book.setIsbn(isbn);
    book.setYear(2008);
    book.setRegistrationDate(Calendar.getInstance());
    requestMessage.setBook(book);
    return requestMessage;
}
```

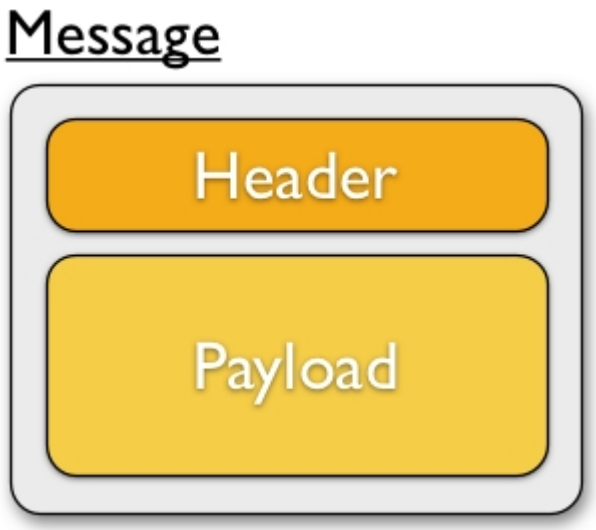
The example above creates an **AddBookRequestMessage** object and puts this as payload to a send action. In combination with a marshaller instance Citrus is able to create a proper XML message payload then.

Chapter 8. Test actions

This chapter gives a brief description to all test actions that a tester can incorporate into the test case. Besides sending and receiving messages the tester may access these actions in order to build a more complex test scenario that fits the desired use case.

8.1. Send

Integration test scenarios need to trigger business logic on foreign applications by calling service interfaces on the system under test. The Citrus test is able to send messages over various message transports (e.g. Http REST, JMS, Kafka, filte transfer).



A message consists of a message header (name-value pairs) and a message body. Later in this section we will see different ways of constructing a message with body and header values.

Java

```
@CitrusTest
public void sendMessageTest() {
    variable("text", "Hello Citrus!");
    variable("messageId", "Mx1x123456789");

    $(send("helloService")
        .message()
        .name("helloMessage")
        .header("Operation", "sayHello")
        .header("RequestTag", "${messageId}")
        .body("""
            <HelloMessage>
                <Text>Hello Citrus!</Text>
            </HelloMessage>
        """)
    );
}
```

XML

```
<test name="ReceiveMessageTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <variables>
        <variable name="text" value="Hello Citrus!"/>
        <variable name="messageId" value="Mx1x123456789"/>
    </variables>

    <actions>
        <send endpoint="helloService">
            <message name="helloMessage">
                <headers>
                    <header name="Operation" value="sayHello"/>
                    <header name="MessageId" value="${messageId}"/>
                </headers>
                <body>
                    <![CDATA[
                    <HelloMessage>
                        <Text>Hello Citrus!</Text>
                    </HelloMessage>
                    ]]>
                </body>
            </message>
        </send>
    </actions>
</test>
```

YAML

```
name: ReceiveMessageTest
variables:
    - name: "text"
      value: "Hello Citrus!"
    - name: "messageId"
      value: "Mx1x123456789"
actions:
  - send:
      endpoint: "helloService"
      message:
        name: "helloMessage"
        headers:
          - name: "Operation"
            value: "sayHello"
          - name: "MessageId"
            value: '${messageId}'
        body: |
          <HelloMessage>
              <Text>Hello Citrus!</Text>
          </HelloMessage>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="SendMessageTest">
        <variables>
            <variable name="text" value="Hello Citrus!"/>
            <variable name="messageId" value="Mx1x123456789"/>
        </variables>

        <actions>
            <send endpoint="helloService">
                <message name="helloMessage">
                    <payload>
                        <![CDATA[
                        <HelloMessage>
                            <Text>${text}</Text>
                        </HelloMessage>
                        ]]>
                    </payload>
                </message>
                <header>
                    <element name="Operation" value="sayHello"/>
                    <element name="MessageId" value="${messageId}"/>
                </header>
            </send>
        </actions>
    </testcase>
</spring:beans>
```

The message name is optional and defines the message identifier in the local message store. This message name is very useful when accessing the message content later on during the test case. The local message store is handled per test case and contains all exchanged messages.

The sample uses both header and payload as message parts to send. In both parts you can use variable definitions (see **${text}** and **${messageId}**). So first of all let us recap what variables do. Test variables are defined at the very beginning of the test case and are valid throughout all actions that take place in the test. This means that actions can simply reference a variable by the expression ***${variable-name}*** .

> Use variables wherever you can! At least the important entities of a test should be defined as variables at the beginning. The test case improves maintainability and flexibility when using variables.

Now let's have a closer look at the sending action. The **'endpoint'** attribute might catch your attention first. This attribute references a message endpoint in Citrus configuration by name. As previously mentioned the message endpoint definition lives in a separate configuration file and contains the actual message transport settings. In this example the **"helloService"** is referenced which is a message endpoint for sending out messages via JMS or HTTP for instance.

The test case is not aware of any transport details, because it does not have to. The advantages are obvious: On the one hand multiple test cases can reference the message endpoint definition for better reuse. Secondly test cases are independent of message transport details. So connection factories, user credentials, endpoint uri values and so on are not present in the test case.

In other words the **"endpoint"** attribute of the `<send>` element specifies which message endpoint definition to use and therefore where the message should go to. Once again all available message endpoints are configured in a separate Citrus configuration file. We will come to this later on. Be sure to always pick the right message endpoint type in order to publish your message to the right destination.

Now that the message sender pattern is clear we can concentrate on how to specify the message content to be sent. There are several possibilities for you to define message content in Citrus:

8.1.1. Send message body

The most important thing when dealing with sending actions is to prepare the message payload and header. You can specify the body as nested String value.

Java

```
@CitrusTest
public void sendMessageTest() {
    $(send("helloService")
        .message()
        .body("""
            <HelloMessage>
                <Text>Hello Citrus!</Text>
            </HelloMessage>
        """)
    );
}
```

XML

```
<test name="SendMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <send endpoint="helloService">
            <message>
                <body>
                    <![CDATA[
                    <HelloMessage>
                        <Text>Hello Citrus!</Text>
                    </HelloMessage>
                    ]]>
                </body>
            </message>
        </receive>
    </actions>
</test>
```

YAML

```
name: SendMessageTest
actions:
  - send:
      endpoint: "helloService"
      message:
        body: |
          <HelloMessage>
              <Text>Hello Citrus!</Text>
          </HelloMessage>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="SendMessageTest">
        <actions>
            <send endpoint="helloService">
                <message>
                    <payload>
                        <![CDATA[
                        <HelloMessage>
                            <Text>Hello Citrus!</Text>
                        </HelloMessage>
                        ]]>
                    </payload>
                </message>
            </receive>
        </actions>
    </testcase>
</spring:beans>
```

A simple way of defining the message body content is to provide the message body as a String. You can do this in the different supported languages by embedding the message content in the body section of the `send` action.

In XML you can embed the content as a `CDATA` section and in Java, or YAML you may want to use text blocks.

When the message body content is an XML payload you can also use nested XML elements in the XML domain specific languages as the next example shows:

XML

```
<test name="ReceiveMessageTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <send endpoint="helloService">
            <message name="helloRequest">
                <body>
                    <HelloMessage xmlns="http://sample.org/">
                        <Text>Hello Citrus!</Text>
                    </HelloMessage>
                </body>
            </message>
        </send>
    </actions>
</test>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="ReceiveMessageTest">
        <actions>
            <send endpoint="helloService">
              <message>
                <payload>
                  <HelloMessage xmlns="http://sample.org/">
                      <Text>Hello Citrus!</Text>
                  </HelloMessage>
                </payload>
              </message>
            </send>
        </actions>
    </testcase>
</spring:beans>
```

In XML you can use nested XML elements or CDATA sections. Sometimes the nested XML message payload elements may cause XSD schema violations. This is because of variable values not fitting the XSD schema rules for example. In this scenario you could also use simple CDATA sections as payload data. In this case you need to use the `<***data***>` element in contrast to the `<***payload***>` element that we have used in our examples so far.

With this alternative you can skip the XML schema validation from your IDE at design time. Unfortunately you will lose the XSD auto-completion features many XML editors offer when constructing your payload.

Message body content may be quite huge, so you can also load the message content form an external file resource. The file path is given as either a classpath or file system resource.

When writing tests in Java you can use one of the classpath or file system resource implementations to resolve the file path. In XML and other languages you may use a resource path given as: `file="classpath:path/to/request.xml"`. The file path prefix indicates the file resource type (`file:` or `classpath:`), so the file location is resolved either as file system resource (`file:`) or classpath resource (`classpath:`).

Java

```
@CitrusTest
public void sendMessageTest() {
    $(send("helloService")
        .message()
        .body(new ClassPathResource("path/to/request.xml"))
    );
}
```

XML

```
<test name="SendMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <send endpoint="helloService">
            <message>
                <body>
                    <resource file="classpath:path/to/request.xml"/>
                </body>
            </message>
        </send>
    </actions>
</test>
```

YAML

```
name: SendMessageTest
actions:
  - send:
      endpoint: "helloService"
      message:
        body:
          resource:
            file: "classpath:path/to/request.xml"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="SendMessageTest">
        <actions>
            <send endpoint="helloService">
                <message>
                    <resource file="classpath:path/to/request.xml" />
                </message>
            </send>
        </actions>
    </testcase>
</spring:beans>
```

In addition to defining message payloads as normal Strings and via external file resource (classpath and file system) you can also provide a POJO model object as a message payload. The model object will be serialized with a marshaller or object mapper implementation which gets loaded from the Citrus context.

You can use the marshalling message payload builders in a `send` action as follows.

Marshalling message payload builder

```
$(send("helloService")
    .message()
    .body(marshal(new TestRequest("Hello Citrus!")))
);
```

The `send` action uses the marshalling message builder provided with Citrus and just provides the model object `new TestRequest()`. The marshalling message builder automatically loads a proper XML message marshaller that should be available as a bean in the project context (e.g. the Spring application context). By default, Citrus is searching for a bean of type **org.citrusframework.xml.Marshaller**. You can add the marshaller to your project context as a bean.

Marshaller bean

```
@Bean
public Marshaller xmlMarshaller() {
    return new Jaxb2Marshaller();
}
```

Now Citrus is able to automatically resolve the marshaller from the project context as soon as the `receive` action uses the model object in combination with the `marshal` instruction.

When you have multiple message marshaller instances in your project context you have to tell Citrus which one to use in this particular receive message action.

Reference message marshaller

```
$(send("helloService")
    .message()
    .body(marshal(new TestRequest("Hello Citrus!"), "myMessageMarshallerBean"))
);
```

Now Citrus will marshal the message body with the message marshaller bean named **myMessageMarshallerBean**. This way you can have multiple message marshaller implementations in your project (XML, JSON, and so on).

You can implement your own message payload builder or use one of the provided Citrus message payload builders.

Custom message payload builder

```
@Autowired
private MessagePayloadBuilder mySpecialPayloadBuilder = new FooPayloadBuilder();

$(send("helloService")
    .message()
    .body(mySpecialPayloadBuilder)
);
```

The message payload builder must implement the `MessagePayloadBuilder` interface with the method `buildPayload(TestContext context)`.

MessagePayloadBuilder interface

```
public class FooPayloadBuilder implements MessagePayloadBuilder {

    @Override
    public Object buildPayload(TestContext context) {
        // return some special payload
        return new FooModel();
    }
}
```

You can also use a Citrus message instance directly. Citrus provides different message implementations with fluent APIs to have a convenient way of setting properties (e.g. HttpMessage, MailMessage, FtpMessage, SoapMessage, ...).

Citrus message object

```
$(send("helloService")
    .message(new DefaultMessage("Hello World!"))
);
```

You can explicitly overwrite some message values in the body before the validations is performed. This is for overwriting specific message elements with variable values for instance. Also, you can overwrite values using XPath (xpath) or JsonPath (json-path) expressions.

Java

```
@CitrusTest
public void jsonPathTest() {
    $(receive("someEndpoint")
        .message()
        .type(MessageType.JSON)
        .body(new ClassPathResource("path/to/request.xml"))
        .process(jsonPath()
            .expression("$.user.name", "Penny")
            .expression("$['user']['name']", "${userName}"));
    );
}
```

XML

```
<test name="JsonPathTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <receive endpoint="someEndpoint">
            <message type="json">
                <body>
                    <resource file="classpath:path/to/request.xml" />
                </body>
                <expression path="$.user.name" value="Penny"/>
                <expression path="$['user']['name']" value="${userName}"/>
            </message>
        </receive>
    </actions>
</test>
```

YAML

```
name: JsonPathTest
actions:
  - receive:
      endpoint: "someEndpoint"
      message:
        type: json
        resource:
          file: "classpath:path/to/request.xml"
        expression:
          - path: '$.user.name'
            value: "Penny"
          - path: '$["user"]["name"]'
            value: '${userName}'
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="JsonPathTest">
        <actions>
            <receive endpoint="someEndpoint">
                <message type="json">
                    <resource file="classpath:path/to/request.xml" />
                    <element path="$.user.name" value="Penny"/>
                    <element path="$['user']['name']" value="${userName}"/>
                </message>
            </receive>
        </actions>
    </testcase>
</spring:beans>
```

8.1.2. Send message headers

Defining the message header is an essential part. Citrus uses name-value pairs like "Operation" and "MessageId" in the next example to set message header entries.

Depending on what message endpoint and which message transport underneath is used the header values will be shipped in different ways. In JMS for instance the headers go to the header section of the message, in Http we set mime headers accordingly, in SOAP we can access the SOAP header elements and so on. Citrus knows how to set headers on different message transports and aims to do the hard work for you .

Java

```
@CitrusTest
public void messageHeaderTest() {
    $(send("helloService")
        .message()
        .header("Operation", "sayHello")
        .header("MessageId", "${messageId}")
        .body("""
            <TestMessage xmlns="http://citrusframework.org/schema">
                <Text>Hello!</Text>
            </TestMessage>
        """)
    );
}
```

XML

```
<test name="MessageHeaderTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <send endpoint="someEndpoint">
            <message>
                <headers>
                    <header name="Operation" value="sayHello"/>
                </headers>
                <body>
                    <payload>
                        <TestMessage xmlns="http://citrusframework.org/schema">
                            <Text>Hello!</Text>
                        </TestMessage>
                    </payload>
                </body>
            </message>
        </send>
    </actions>
</test>
```

YAML

```
name: MessageHeaderTest
actions:
  - send:
      endpoint: "helloService"
      message:
        headers:
          - name: "Operation"
            value: "sayHello"
        body:
          data: |
            <TestMessage xmlns="http://citrusframework.org/schema">
                <Text>Hello!</Text>
            </TestMessage>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="MessageHeaderTest">
        <actions>
            <send endpoint="helloService">
                <message>
                    <payload>
                        <TestMessage xmlns="http://citrusframework.org/schema">
                            <Text>Hello!</Text>
                        </TestMessage>
                    </payload>
                </message>
                <header>
                    <element name="Operation" value="sayHello"/>
                </header>
            </send>
        </actions>
    </testcase>
</spring:beans>
```

The message headers to send are defined by a simple name and value pairs. Of course you can use test variables in header values as well.

This is basically how to send messages in Citrus. The test case is responsible for constructing the message content while the predefined message endpoint holds transport specific settings. Test cases reference endpoint components to publish messages to the outside world. The variable support in message payload and message header enables you to add dynamic values before sending out the message.

8.1.3. Groovy Markup builder

With the Groovy markup builder you can build XML message body content in a simple way, without having to write the typical XML overhead.

The Groovy test action support lives in a separate module. You need to add the module to your project to use the functionality.

citrus-groovy dependency module

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-groovy</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

For example, we use a Groovy script to construct the XML message to be sent out. Instead of a plain CDATA XML section or the nested body XML data we write a Groovy script snippet.

Java

```
@CitrusTest
public void scriptMessageBuilderTest() {
    $(send("helloService")
        .message()
        .body(new GroovyScriptPayloadBuilder("""
                    markupBuilder.TestRequest(xmlns:
'https://citrus.schemas/samples/sayHello.xsd') {
                        Message('Hello World!')
                    }
        """))
    );
}
```

XML

```
<test name="ScriptMessageBuilderTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <send endpoint="helloService">
            <message>
                <body>
                    <builder type="groovy">
                        markupBuilder.TestRequest(xmlns:
'https://citrus.schemas/samples/sayHello.xsd') {
                            Message('Hello World!')
                        }
                    </builder>
                </body>
            </message>
        </send>
    </actions>
</test>
```

YAML

```
name: ScriptMessageBuilderTest
actions:
  - send:
      endpoint: "helloService"
      message:
        builder:
          type: "groovy"
          value: |
            markupBuilder.TestRequest(xmlns:
'https://citrus.schemas/samples/sayHello.xsd') {
                Message('Hello World!')
            }
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="ScriptMessageBuilderTest">
        <actions>
            <send endpoint="helloService">
              <message>
                <builder type="groovy">
                    markupBuilder.TestRequest(xmlns:
'https://citrus.schemas/samples/sayHello.xsd') {
                        Message('Hello World!')
                    }
                </builder>
              </message>
            </send>
        </actions>
    </testcase>
</spring:beans>
```

The Groovy markup builder generates the XML message body with following content:

Genereted markup

```
<TestRequest xmlns="https://citrus.schemas/samples/sayHello.xsd">
  <Message>Hello World</Message>
</TestRequest>
```

We use the **builder** element with type **groovy** and the markup builder code is directly written to this element. As you can see from the example above, you can mix XPath and Groovy markup builder code. The markup builder syntax is very easy and follows the simple rule: **markupBuilder.ROOT-ELEMENT{ CHILD-ELEMENTS }** . However the tester has to follow some simple rules and naming conventions when using the Citrus markup builder extension:

- The markup builder is accessed within the script over an object named markupBuilder. The name of the custom root element follows with all its child elements.
- Child elements may be defined within curly brackets after the root-element (the same applies for further nested child elements)
- Attributes and element values are defined within round brackets, after the element name
- Attribute and element values have to stand within apostrophes (e.g. attribute-name: 'attribute-value')

The Groovy markup builder script may also be used as external file resource:

Java

```
@CitrusTest
public void scriptMessageBuilderTest() {
    $(send("helloService")
        .message()
        .body(new
GroovyFileResourcePayloadBuilder("classpath:path/to/helloRequest.groovy"))
    );
}
```

XML

```
<test name="ScriptMessageBuilderTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <send endpoint="helloService">
            <message>
                <body>
                    <builder type="groovy"
file="classpath:path/to/helloRequest.groovy"/>
                </body>
            </message>
        </send>
    </actions>
</test>
```

YAML

```
name: ScriptMessageBuilderTest
actions:
  - send:
      endpoint: "helloService"
      message:
        builder:
          type: "groovy"
          file: "classpath:path/to/helloRequest.groovy"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="ScriptMessageBuilderTest">
        <actions>
            <send endpoint="helloService">
              <message>
                <builder type="groovy" file="classpath:path/to/helloRequest.groovy"/>
              </message>
            </send>
        </actions>
    </testcase>
</spring:beans>
```

The markup builder implementation in Groovy offers great possibilities in defining message body content. We do not need to write XML the tag overhead anymore. The approach also enables us to construct complex message body content with Groovy script logic like iterations and conditional elements. For detailed markup builder descriptions please see the official Groovy documentation.

8.2. Receive

Receiving and validating messages is an essential part of an integration test. You can perform assertions and checks on incoming messages in order to verify that everything works as expected.

A message consists of a message header (name-value pairs) and a message body. You can specify expected message content on a receive message test action. Citrus will perform validation steps and raise errors in case the incoming message does not match these expectations.

Java

```
@CitrusTest
public void receiveMessageTest() {
    $(receive("helloService")
        .message()
        .name("helloRequest")
        .header("Operation", "sayHello")
        .header("MessageId", "${messageId}")
        .body("""
            <HelloMessage>
                <Text>Hello Citrus!</Text>
            </HelloMessage>
        """)
    );
}
```

XML

```
<test name="ReceiveMessageTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <receive endpoint="helloService">
            <message name="helloRequest">
                <headers>
                    <header name="Operation" value="sayHello"/>
                    <header name="MessageId" value="${messageId}"/>
                </headers>
                <body>
                    <![CDATA[
                    <HelloMessage>
                        <Text>Hello Citrus!</Text>
                    </HelloMessage>
                    ]]>
                </body>
            </message>
        </receive>
    </actions>
</test>
```

YAML

```
name: ReceiveMessageTest
actions:
  - receive:
      endpoint: "helloService"
      message:
        name: "helloRequest"
        headers:
          - name: "Operation"
            value: "sayHello"
          - name: "MessageId"
            value: '${messageId}'
        body: |
          <HelloMessage>
              <Text>Hello Citrus!</Text>
          </HelloMessage>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="ReceiveMessageTest">
        <actions>
            <receive endpoint="helloService">
                <message name="helloRequest">
                    <payload>
                        <![CDATA[
                        <HelloMessage>
                            <Text>Hello Citrus!</Text>
                        </HelloMessage>
                        ]]>
                    </payload>
                </message>
                <header>
                    <element name="Operation" value="sayHello"/>
                    <element name="MessageId" value="${messageId}"/>
                </header>
            </receive>
        </actions>
    </testcase>
</spring:beans>
```

The message name is optional and defines the message identifier in the local message store. This message name is very useful when accessing the message content later on during the test case. The local message store is handled per test case and contains all exchanged messages.

The test action waits for a message to arrive. The whole test execution is blocked while waiting for the expected message. This is important to ensure the step by step test workflow processing. Of course, you can specify a message timeout setting so the receiver will only wait a given amount of time before raising a timeout error. Following from that timeout exception the test case fails as the message did not arrive in time. Citrus defines default timeout settings for all message receiving tasks.

In case the message arrives in time, the test action moves on to validate the message content as a next step. The user is able to choose from different validation capabilities. On the one hand you can specify a whole message body content that you expect as control template. In this case the received message structure is compared to the expected message content element by element. On the other hand you can use explicit element validation where only a small subset of message elements is validated.

In addition to verifying the message body content, Citrus will also perform validation on the received message header values. Test variable usage is supported as usual during the whole validation process for body and header checks.

In general the validation component (validator) in Citrus works hand in hand with a message receiving component as the following figure shows:

The message receiving component passes the message to the validator where the individual validation steps are performed. Let us have a closer look at the validation options and features step by step.

8.2.1. Validate message body

The most detailed validation of incoming messages is to define some expected message body. The Citrus message validator will then perform a detailed message body comparison. The incoming message has to match exactly to the expected message body. The different message validator implementations in Citrus provide deep comparison of message structures such as XML, JSON and so on.

So by defining an expected message body we validate the incoming message in syntax and semantics. In case a difference is identified by the message validator the validation and the test case fails with respective exceptions. This is how you can define message body content in receive action:

Java

```
@CitrusTest
public void receiveMessageTest() {
    $(receive("helloService")
        .message()
        .body("""
            <HelloMessage>
                <Text>Hello Citrus!</Text>
            </HelloMessage>
        """)
    );
}
```

XML

```
<test name="ReceiveMessageTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <receive endpoint="helloService">
            <message>
                <body>
                    <![CDATA[
                    <HelloMessage>
                        <Text>Hello Citrus!</Text>
                    </HelloMessage>
                    ]]>
                </body>
            </message>
        </receive>
    </actions>
</test>
```

YAML

```
name: ReceiveMessageTest
actions:
  - receive:
      endpoint: "helloService"
      message:
        body: |
          <HelloMessage>
              <Text>Hello Citrus!</Text>
          </HelloMessage>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="ReceiveMessageTest">
        <actions>
            <receive endpoint="helloService">
                <message>
                    <payload>
                        <![CDATA[
                        <HelloMessage>
                            <Text>Hello Citrus!</Text>
                        </HelloMessage>
                        ]]>
                    </payload>
                </message>
            </receive>
        </actions>
    </testcase>
</spring:beans>
```

A simple way of defining the expected message body content is to provide the message body as a String. You can do this in the different supported languages by embedding the message content in the body section of the `receive` action.

In XML you can embed the content as a `CDATA` section and in Java, or YAML you may want to use text blocks.

When the message body content is an XML payload you can also use nested XML elements in the XML domain specific languages as the next example shows:

XML

```
<test name="ReceiveMessageTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <receive endpoint="helloService">
            <message name="helloRequest">
                <body>
                    <HelloMessage xmlns="http://sample.org/">
                        <Text>Hello Citrus!</Text>
                    </HelloMessage>
                </body>
            </message>
        </receive>
    </actions>
</test>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="ReceiveMessageTest">
        <actions>
            <receive endpoint="helloService">
              <message>
                <payload>
                  <HelloMessage xmlns="http://sample.org/">
                      <Text>Hello Citrus!</Text>
                  </HelloMessage>
                </payload>
              </message>
            </receive>
        </actions>
    </testcase>
</spring:beans>
```

In XML you can use nested XML elements or CDATA sections. Sometimes the nested XML message payload elements may cause XSD schema violations. This is because of variable values not fitting the XSD schema rules for example. In this scenario you could also use simple CDATA sections as payload data. In this case you need to use the `<***data***>` element in contrast to the `<***payload***>` element that we have used in our examples so far.

With this alternative you can skip the XML schema validation from your IDE at design time. Unfortunately you will lose the XSD auto-completion features many XML editors offer when constructing your payload.

Message body content may be quite huge, so you can also load the message content form an external file resource. The file path is given as either a classpath or file system resource.

When writing tests in Java you can use one of the classpath or file system resource implementations to resolve the file path. In XML and other languages you may use a resource path given as: `file="classpath:path/to/request.xml"`. The file path prefix indicates the file resource type (`file:` or `classpath:`), so the file location is resolved either as file system resource (`file:`) or classpath resource (`classpath:`).

Java

```
@CitrusTest
public void receiveMessageTest() {
    $(receive("helloService")
        .message()
        .body(new ClassPathResource("path/to/request.xml"))
    );
}
```

XML

```
<test name="ReceiveMessageTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <receive endpoint="helloService">
            <message>
                <body>
                    <resource file="classpath:path/to/request.xml"/>
                </body>
            </message>
        </receive>
    </actions>
</test>
```

YAML

```
name: ReceiveMessageTest
actions:
  - receive:
      endpoint: "helloService"
      message:
        body:
          resource:
            file: "classpath:path/to/request.xml"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="ReceiveMessageTest">
        <actions>
            <receive endpoint="helloService">
                <message>
                    <resource file="classpath:path/to/request.xml" />
                </message>
            </receive>
        </actions>
    </testcase>
</spring:beans>
```

In addition to defining message payloads as normal Strings and via external file resource (classpath and file system) you can also provide a POJO model object as a message payload. The model object will be serialized with a marshaller or object mapper implementation which gets loaded from the Citrus context.

You can use the marshalling message payload builders in a `receive` action as follows.

Marshalling message payload builder

```
$(receive("helloService")
    .message()
    .body(marshal(new TestRequest("Hello Citrus!")))
);
```

The `receive` action uses the marshalling message builder provided with Citrus and just provides the model object `new TestRequest()`. The marshalling message builder automatically loads a proper XML message marshaller that should be available as a bean in the project context (e.g. the Spring application context). By default, Citrus is searching for a bean of type **org.citrusframework.xml.Marshaller**. You can add the marshaller to your project context as a bean.

Marshaller bean

```
@Bean
public Marshaller xmlMarshaller() {
    return new Jaxb2Marshaller();
}
```

Now Citrus is able to automatically resolve the marshaller from the project context as soon as the `receive` action uses the model object in combination with the `marshal` instruction.

When you have multiple message marshaller instances in your project context you have to tell Citrus which one to use in this particular receive message action.

Reference message marshaller

```
$(receive("helloService")
    .message()
    .body(marshal(new TestRequest("Hello Citrus!"), "myMessageMarshallerBean"))
);
```

Now Citrus will marshal the message body with the message marshaller bean named **myMessageMarshallerBean**. This way you can have multiple message marshaller implementations in your project (XML, JSON, and so on).

You can implement your own message payload builder or use one of the provided Citrus message payload builders.

Custom message payload builder

```
@Autowired
private MessagePayloadBuilder mySpecialPayloadBuilder = new FooPayloadBuilder();

$(receive("helloService")
    .message()
    .body(mySpecialPayloadBuilder)
);
```

The message payload builder must implement the `MessagePayloadBuilder` interface with the method `buildPayload(TestContext context)`.

MessagePayloadBuilder interface

```
public class FooPayloadBuilder implements MessagePayloadBuilder {

    @Override
    public Object buildPayload(TestContext context) {
        // return some special payload
        return new FooModel();
    }
}
```

You can also use a Citrus message instance directly. Citrus provides different message implementations with fluent APIs to have a convenient way of setting properties (e.g. HttpMessage, MailMessage, FtpMessage, SoapMessage, ...).

Citrus message object

```
$(receive("helloService")
    .message(new DefaultMessage("Hello World!"))
);
```

You can explicitly overwrite some message values in the body before the validations is performed. This is for overwriting specific message elements with variable values for instance. Also, you can overwrite values using XPath (xpath) or JsonPath (json-path) expressions.

Java

```
@CitrusTest
public void jsonPathTest() {
    $(receive("someEndpoint")
        .message()
        .type(MessageType.JSON)
        .body(new ClassPathResource("path/to/request.xml"))
        .validate(jsonPath()
            .expression("$.user.name", "Penny")
            .expression("$['user']['name']", "${userName}"))
    );
}
```

XML

```
<test name="JsonPathTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <receive endpoint="someEndpoint">
            <message type="json">
                <body>
                    <resource file="classpath:path/to/request.xml" />
                </body>
            </message>
            <validate path="$.user.name" value="Penny"/>
            <validate path="$['user']['name']" value="${userName}"/>
        </receive>
    </actions>
</test>
```

YAML

```
name: JsonPathTest
actions:
  - receive:
      endpoint: "someEndpoint"
      message:
        type: json
        resource:
          file: "classpath:path/to/request.xml"
        validate:
          jsonPath:
            - expression: '$.user.name'
              value: "Penny"
            - expression: '$["user"]["name"]'
              value: '${userName}'
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="JsonPathTest">
        <actions>
            <receive endpoint="someEndpoint">
                <message type="json">
                    <resource file="classpath:path/to/request.xml" />
                    <validate path="$.user.name" value="Penny"/>
                    <validate path="$['user']['name']" value="${userName}"/>
                </message>
            </receive>
        </actions>
    </testcase>
</spring:beans>
```

In addition to that you can ignore some elements that are skipped in comparison. We will describe this later on in this section. Now let's continue with message header validation.

8.2.2. Validate message headers

Message headers are used widely in enterprise messaging. The message headers often represent a critical part of the message semantics and need to be validated, too. Citrus is able to validate message headers by name and value.

Java

```
@CitrusTest
public void messageHeaderTest() {
    $(receive("helloService")
        .message()
        .header("Operation", "sayHello")
        .body("""
            <TestMessage xmlns="http://citrusframework.org/schema">
                <Text>Hello!</Text>
            </TestMessage>
        """)
    );
}
```

XML

```
<test name="MessageHeaderTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <receive endpoint="someEndpoint">
            <message>
                <headers>
                    <header name="Operation" value="sayHello"/>
                </headers>
                <body>
                    <payload>
                        <TestMessage xmlns="http://citrusframework.org/schema">
                            <Text>Hello!</Text>
                        </TestMessage>
                    </payload>
                </body>
            </message>
        </receive>
    </actions>
</test>
```

YAML

```
name: MessageHeaderTest
actions:
  - receive:
      endpoint: "helloService"
      message:
        headers:
          - name: "Operation"
            value: "sayHello"
        body:
          data: |
            <TestMessage xmlns="http://citrusframework.org/schema">
                <Text>Hello!</Text>
            </TestMessage>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="MessageHeaderTest">
        <actions>
            <receive endpoint="helloService">
                <message>
                    <payload>
                        <TestMessage xmlns="http://citrusframework.org/schema">
                            <Text>Hello!</Text>
                        </TestMessage>
                    </payload>
                </message>
                <header>
                    <element name="Operation" value="sayHello"/>
                </header>
            </receive>
        </actions>
    </testcase>
</spring:beans>
```

The expected message headers are defined by a name and value pair. Citrus will check that the expected message header is present and will verify the value accordingly. In case the message header is not found or the value does not match Citrus will raise an exception and the test fails. You can use validation matchers (validation-matcher) for a more powerful validation of header values, too.

Header definition in Java DSL is straight forward as we just define name and value as usual. This completes tha basic message validation in Citrus. The individual message validator implementations may add additional validation capabilities such as XML schema validation or XPath and JSONPath validation. Please refer to the respective chapters in this guide to learn more about that.

8.2.3. Ignore elements

Sometimes a tester can not verify all values because specifying expected values is not possible for non deterministic values (e.g. timestamps, dynamic and generated identifiers).

You can use a path expressions (e.g. Xpath, JsonPath) to ignoring a very specific entry in the message body.

Java

```
@CitrusTest
public void ignorePathTest() {
    $(receive(someEndpoint)
        .message()
        .type(MessageType.JSON)
        .body("""
            {
              "users":
              [{
                "name": "Jane",
                "token": "?",
                "lastLogin": 0
              },
              {
                "name": "Penny",
                "token": "?",
                "lastLogin": 0
              },
              {
                "name": "Mary",
                "token": "?",
                "lastLogin": 0
              }]
            }
        """)
        .validate(json()
                    .ignore("$.users[*].token")
                    .ignore("$..lastLogin"))
    );
}
```

XML

```
<test name="IgnorePathTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <receive endpoint="someEndpoint">
            <message type="json">
                <body>
                    <data>
                    {
                      "users":
                      [{
                        "name": "Jane",
                        "token": "?",
                        "lastLogin": 0
                      },
                      {
                        "name": "Penny",
                        "token": "?",
                        "lastLogin": 0
                      },
                      {
                        "name": "Mary",
                        "token": "?",
                        "lastLogin": 0
                      }]
                    }
                    </data>
                </body>
            </message>
            <ignore path="$..lastLogin" />
            <ignore path="$.users[*].token" />
        </receive>
    </actions>
</test>
```

YAML

```yaml
name: IgnorePathTest
actions:
  - receive:
      endpoint: "someEndpoint"
      message:
        type: "json"
        body:
          data: |
            {
              "users":
              [{
                "name": "Jane",
                "token": "?",
                "lastLogin": 0
              },
              {
                "name": "Penny",
                "token": "?",
                "lastLogin": 0
              },
              {
                "name": "Mary",
                "token": "?",
                "lastLogin": 0
              }]
            }
          ignore:
            - path: "$.users[*].token"
            - path: "$..lastLogin"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="IgnorePathTest">
        <actions>
            <receive endpoint="someEndpoint">
                <message type="json">
                    <data>
                    {
                      "users":
                      [{
                        "name": "Jane",
                        "token": "?",
                        "lastLogin": 0
                      },
                      {
                        "name": "Penny",
                        "token": "?",
                        "lastLogin": 0
                      },
                      {
                        "name": "Mary",
                        "token": "?",
                        "lastLogin": 0
                      }]
                    }
                    </data>
                    <ignore path="$.users[*].token" />
                    <ignore path="$..lastLogin" />
                </message>
            </receive>
        </actions>
    </testcase>
</spring:beans>
```

The sample above adds JsonPath expressions as ignore statements. This means that we explicitly leave out the evaluated elements from validation. In the example above we explicitly skip the **token** entry and all **lastLogin** values that are obviously timestamp values in milliseconds.

The path evaluation is very powerful when it comes to select a set of objects and elements. This is how you can ignore several elements with path expressions.

As an alternative you can also use the `@ignore@` placeholder in the message content. The placeholder tells Citrus to skip the element in validation.

Java

```
@CitrusTest
public void ignorePathTest() {
    $(receive(someEndpoint)
        .message()
        .type(MessageType.JSON)
        .body("""
            {
              "users":
              [{
                "name": "Jane",
                "token": "@ignore@",
                "lastLogin": "@ignore@"
              },
              {
                "name": "Penny",
                "token": "@ignore@",
                "lastLogin": "@ignore@"
              },
              {
                "name": "Mary",
                "token": "@ignore@",
                "lastLogin": "@ignore@"
              }]
            }
        """)
    );
}
```

XML

```
<test name="IgnorePathTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <receive endpoint="someEndpoint">
            <message type="json">
                <body>
                    <data>
                    {
                      "users":
                      [{
                        "name": "Jane",
                        "token": "@ignore@",
                        "lastLogin": "@ignore@"
                      },
                      {
                        "name": "Penny",
                        "token": "@ignore@",
                        "lastLogin": "@ignore@"
                      },
                      {
                        "name": "Mary",
                        "token": "@ignore@",
                        "lastLogin": "@ignore@"
                      }]
                    }
                    </data>
                </body>
            </message>
        </receive>
    </actions>
</test>
```

YAML

```yaml
name: IgnorePathTest
actions:
  - receive:
      endpoint: "someEndpoint"
      message:
        type: "json"
        body:
          data: |
            {
              "users":
              [{
                "name": "Jane",
                "token": "@ignore@",
                "lastLogin": "@ignore@"
              },
              {
                "name": "Penny",
                "token": "@ignore@",
                "lastLogin": "@ignore@"
              },
              {
                "name": "Mary",
                "token": "@ignore@",
                "lastLogin": "@ignore@"
              }]
            }
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="IgnorePathTest">
        <actions>
            <receive endpoint="someEndpoint">
                <message type="json">
                  <data>
                    {
                      "users":
                      [{
                        "name": "Jane",
                        "token": "@ignore@",
                        "lastLogin": "@ignore@"
                      },
                      {
                        "name": "Penny",
                        "token": "@ignore@",
                        "lastLogin": "@ignore@"
                      },
                      {
                        "name": "Mary",
                        "token": "@ignore@",
                        "lastLogin": "@ignore@"
                      }]
                    }
                  </data>
                </message>
            </receive>
        </actions>
    </testcase>
</spring:beans>
```

You can also ignore a whole sub-tree in XML and whole objects and arrays in Json with the ignore expression/placeholder.

The `ignore` expression as well as the `@ignore@` placeholder will only skip the value matching validations for the selected element or object. The element still has to be present in the message structure. In case the element is missing for any reason the validation fails even for ignored values.

8.2.4. Message selectors

The `<selector>` element inside the receiving action defines key-value pairs in order to filter the messages being received. The filter applies to the message headers. This means that a receiver will only accept messages matching a header element value. In messaging applications the header information often holds message ids, correlation ids, operation names and so on. With this information given you can explicitly listen for messages that belong to your test case. This is very

helpful to avoid receiving messages that are still available on the message destination.

Let's say the tested software application keeps sending messages that belong to previous test cases. This could happen in retry situations where the application error handling automatically tries to solve a communication problem that occurred during previous test cases. As a result a message destination (e.g. a JMS message queue) contains messages that are not valid anymore for the currently running test case. The test case might fail because the received message does not apply to the actual use case. So we will definitely run into validation errors as the expected message control values do not match.

Now we have to find a way to avoid these problems. The test could filter the messages on a destination to only receive messages that apply for the use case that is being tested. The Java Messaging System (JMS) came up with a message header selector that will only accept messages that fit the expected header values.

Let us have a closer look at a message selector inside a receiving action:

Java

```
@CitrusTest
public void selectorTest() {
    Map<String, String> selectorMap = new HashMap<>();
    selectorMap.put("correlationId", "Cx1x123456789");
    selectorMap.put("operation", "getOrders");

    $(receive(someEndpoint)
        .selector(selectorMap)
        .message()
        .body("...")
    );
}
```

XML

```
<test name="SelectorTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <receive endpoint="someEndpoint">
            <selector>
                <element name="correlationId" value="Cx1x123456789"/>
                <element name="operation" value="getOrders"/>
            </selector>
            <message>
                <!-- -->
            </message>
        </receive>
    </actions>
</test>
```

YAML

```
name: SelectorTest
actions:
  - receive:
      endpoint: "someEndpoint"
      selector:
        element:
          - name: correlationId
            value: "Cx1x123456789"
          - name: operation
            value: "getOrders"
      message:
        # ...
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="SelectorTest">
        <actions>
            <receive endpoint="someEndpoint">
              <selector>
                <element name="correlationId" value="Cx1x123456789"/>
                <element name="operation" value="getOrders"/>
              </selector>
              <!-- ... -->
            </receive>
        </actions>
    </testcase>
</spring:beans>
```

This example shows how message selectors work. The selector will only accept messages that meet the correlation id and the operation in the header values. All other messages on the message destination are ignored. The selector elements are automatically associated to each other using the logical AND operator. This means that the message selector string would look like this:

```
correlationId = 'Cx1x123456789' AND operation = 'getOrders'
```

Instead of using several elements in the selector you can also define a selector string directly which gives you more power in constructing the selection logic yourself. This way you can use **AND** logical operators yourself.

Java

```
@CitrusTest
public void selectorTest() {
    $(receive(someEndpoint)
        .selector("correlationId = 'Cx1x123456789' AND operation = 'getOrders'")
        .message()
        .body("...")
    );
}
```

XML

```
<test name="SelectorTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <receive endpoint="someEndpoint">
            <selector>
                <value>
                    correlationId = 'Cx1x123456789' AND operation = 'getOrders'
                </value>
            </selector>
            <message>
                <!-- -->
            </message>
        </receive>
    </actions>
</test>
```

YAML

```
name: SelectorTest
actions:
  - receive:
      endpoint: "someEndpoint"
      selector:
        value: "correlationId = 'Cx1x123456789' AND operation = 'getOrders'"
      message:
        # ...
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="SelectorTest">
        <actions>
            <receive endpoint="someEndpoint">
              <selector>
                <value>
                    correlationId = 'Cx1x123456789' AND operation = 'getOrders'
                </value>
              </selector>
              <!-- ... -->
            </receive>
        </actions>
    </testcase>
</spring:beans>
```

In case you want to run tests in parallel message selectors become essential in your test cases. The different tests running at the same time will steal messages from each other when you lack of message selection mechanisms.

8.2.5. Groovy Markup builder

With the Groovy markup builder you can build XML message body content in a simple way, without having to write the typical XML overhead.

The Groovy test action support lives in a separate module. You need to add the module to your project to use the functionality.

citrus-groovy dependency module

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-groovy</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

For example, we use a Groovy script to construct the XML message to be sent out. Instead of a plain CDATA XML section or the nested body XML data we write a Groovy script snippet.

Java

```
@CitrusTest
public void scriptMessageBuilderTest() {
    $(receive("helloService")
        .message()
        .body(new GroovyScriptPayloadBuilder("""
                    markupBuilder.TestRequest(xmlns:
'https://citrus.schemas/samples/sayHello.xsd') {
                        Message('Hello World!')
                    }
        """))
    );
}
```

XML

```
<test name="ScriptMessageBuilderTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <receive endpoint="helloService">
            <message>
                <body>
                    <builder type="groovy">
                        markupBuilder.TestRequest(xmlns:
'https://citrus.schemas/samples/sayHello.xsd') {
                            Message('Hello World!')
                        }
                    </builder>
                </body>
            </message>
        </receive>
    </actions>
</test>
```

YAML

```
name: ScriptMessageBuilderTest
actions:
  - receive:
      endpoint: "helloService"
      message:
        builder:
          type: "groovy"
          value: |
            markupBuilder.TestRequest(xmlns:
'https://citrus.schemas/samples/sayHello.xsd') {
                Message('Hello World!')
            }
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="ScriptMessageBuilderTest">
        <actions>
            <receive endpoint="helloService">
              <message>
                <builder type="groovy">
                    markupBuilder.TestRequest(xmlns:
'https://citrus.schemas/samples/sayHello.xsd') {
                        Message('Hello World!')
                    }
                </builder>
              </message>
            </receive>
        </actions>
    </testcase>
</spring:beans>
```

The Groovy markup builder generates the XML message body with following content:

Generated markup

```
<TestRequest xmlns="https://citrus.schemas/samples/sayHello.xsd">
  <Message>Hello World</Message>
</TestRequest>
```

We use the **builder** element with type **groovy** and the markup builder code is directly written to this element. As you can see from the example above, you can mix XPath and Groovy markup builder code. The markup builder syntax is very easy and follows the simple rule: **markupBuilder.ROOT-ELEMENT{ CHILD-ELEMENTS }** . However the tester has to follow some simple rules and naming conventions when using the Citrus markup builder extension:

- The markup builder is accessed within the script over an object named markupBuilder. The name of the custom root element follows with all its child elements.
- Child elements may be defined within curly brackets after the root-element (the same applies for further nested child elements)
- Attributes and element values are defined within round brackets, after the element name
- Attribute and element values have to stand within apostrophes (e.g. attribute-name: 'attribute-value')

The Groovy markup builder script may also be used as external file resource:

Java

```
@CitrusTest
public void scriptMessageBuilderTest() {
    $(receive("helloService")
        .message()
        .body(new
GroovyFileResourcePayloadBuilder("classpath:path/to/helloRequest.groovy"))
    );
}
```

XML

```
<test name="ScriptMessageBuilderTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <receive endpoint="helloService">
            <message>
                <body>
                    <builder type="groovy"
file="classpath:path/to/helloRequest.groovy"/>
                </body>
            </message>
        </receive>
    </actions>
</test>
```

YAML

```
name: ScriptMessageBuilderTest
actions:
  - receive:
      endpoint: "helloService"
      message:
        builder:
          type: "groovy"
          file: "classpath:path/to/helloRequest.groovy"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="ScriptMessageBuilderTest">
        <actions>
            <receive endpoint="helloService">
              <message>
                <builder type="groovy" file="classpath:path/to/helloRequest.groovy"/>
              </message>
            </receive>
        </actions>
    </testcase>
</spring:beans>
```

The markup builder implementation in Groovy offers great possibilities in defining message body content. We do not need to write XML the tag overhead anymore. The approach also enables us to construct complex message body content with Groovy script logic like iterations and conditional elements. For detailed markup builder descriptions please see the official Groovy documentation.

8.3. SQL

In many cases it is necessary to access the database during a test. This enables a tester to also validate the persistent data in a database. It might also be helpful to prepare the database with some test data before running a test. You can do this using the two database actions that are described in the following sections.

> The SQL test actions live in a separate module. You need to add the module to your project to use the actions.

citrus-sql dependency module

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-sql</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

In general Citrus handles SELECT statements differently to other statements like INSERT, UPDATE and DELETE. When executing an SQL query with SELECT you are able to add validation steps on the result sets returned from the database. This is not allowed when executing update statements like INSERT, UPDATE, DELETE.

> Do not mix statements of type ***SELECT*** with others in a single sql test action. This will lead to errors because validation steps are not valid for statements other than SELECT. Please use separate test actions for update statements.

8.3.1. SQL update, insert, delete

The <sql> action simply executes a group of SQL statements in order to change data in a database. Typically the action is used to prepare the database at the beginning of a test or to clean up the database at the end of a test. You can specify SQL statements like INSERT, UPDATE, DELETE, CREATE TABLE, ALTER TABLE and many more.

On the one hand you can specify the statements as inline SQL or stored in an external SQL resource file as shown in the next two examples.

Java

```
@Autowired
@Qualifier("myDataSource")
private DataSource dataSource;

@CitrusTest
public void sqlTest() {
    $(sql()
        .dataSource(dataSource)
        .statement("DELETE FROM CUSTOMERS")
        .statement("DELETE FROM ORDERS"))
    );

    $(sql()
        .dataSource(dataSource)
        .sqlResource("file:tests/unit/resources/script.sql"))
    );
}
```

XML

```
<test name="SQLTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <sql datasource="myDataSource">
            <statements>
                <statement>DELETE FROM CUSTOMERS</statement>
                <statement>DELETE FROM ORDERS</statement>
            </statements>
        </sql>

        <sql datasource="myDataSource">
            <statements file="file:tests/unit/resources/script.sql"/>
        </sql>
    </actions>
</test>
```

YAML

```yaml
name: "SQLTest"
actions:
  - sql:
      dataSource: "myDataSource"
      statements:
        - statement: "DELETE FROM CUSTOMERS"
        - statement: "DELETE FROM ORDERS"

  - sql:
      dataSource: "myDataSource"
      statements:
        - file: "file:tests/unit/resources/script.sql"
```

Spring XML

```xml
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="SQLTest">
        <actions>
            <sql datasource="myDataSource">
                <statement>DELETE FROM CUSTOMERS</statement>
                <statement>DELETE FROM ORDERS</statement>
            </sql>

            <sql datasource="myDataSource">
                <resource file="file:tests/unit/resources/script.sql"/>
            </sql>
        </actions>
    </testcase>
</spring:beans>
```

The first action uses inline SQL statements defined directly inside the test case. The next action uses an external SQL resource file instead. The file resource can hold several SQL statements separated by new lines. All statements inside the file are executed sequentially by the framework.

You have to pay attention to some rules when dealing with external SQL resources.

- Each statement should begin in a new line
- It is not allowed to define statements with word wrapping
- Comments begin with two dashes "–"

The external file is referenced either as file system resource or class path resource, by using the "file:" or "classpath:" prefix.

Both examples use the "datasource" attribute. This value defines the database data source to be used. The connection to a data source is mandatory, because the test case does not know about user

credentials or database names. The 'datasource' attribute references predefined data sources that are located in a separate Spring configuration file.

8.3.2. SQL query

The <sql> query action is specially designed to execute SQL queries (SELECT * FROM). So the test is able to read data from a database. The query results are validated against expected data as shown in the next example.

Java

```
@Autowired
@Qualifier("myDataSource")
private DataSource dataSource;

@CitrusTest
public void sqlQueryTest() {
    $(sql()
        .dataSource(dataSource)
        .query()
            .statement("select NAME from CUSTOMERS where CUSTOMER_ID='${customerId}'")
            .statement("select COUNT(1) as overall_cnt from ERRORS")
            .statement("select ORDER_ID from ORDERS where DESCRIPTION LIKE
'Migrate%'")
            .statement("select DESCRIPTION from ORDERS where ORDER_ID = 2")
        .validate("ORDER_ID", "1")
        .validate("NAME", "Christoph")
        .validate("OVERALL_CNT", "${rowsCount}")
        .validate("DESCRIPTION", "NULL"));
}
```

XML

```
<test name="SqlQueryTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <sql datasource="myDataSource">
            <statements>
                <statement>select NAME from CUSTOMERS where
ID='${customerId}'</statement>
                <statement>select count(*) from ERRORS</statement>
                <statement>select ID from ORDERS where DESC LIKE 'Def%'</statement>
                <statement>select DESCRIPTION from ORDERS where ID='${id}'</statement>
            </statements>
            <validate column="ID" value="1"/>
            <validate column="NAME" value="Christoph"/>
            <validate column="COUNT(*)" value="${rowsCount}"/>
            <validate column="DESCRIPTION" value="null"/>
        </sql>
    </actions>
</test>
```

YAML

```
name: "SqlQueryTest"
actions:
  - sql:
      dataSource: "dataSource"
      statements:
        - statement: "select NAME from CUSTOMERS where CUSTOMER_ID='${customerId}'"
        - statement: "select COUNT(1) as overall_cnt from ERRORS"
        - statement: "select ORDER_ID from ORDERS where DESCRIPTION LIKE 'Migrate%'"
        - statement: "select DESCRIPTION from ORDERS where ORDER_ID = 2"
      validate:
        - column: "ID"
          value: "1"
        - column: "NAME"
          value: "Christoph"
        - column: "COUNT(*)"
          value: '${rowsCount}'
        - column: "DESCRIPTION"
          value: "null"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- ... -->
    <sql datasource="myDataSource">
        <statement>select NAME from CUSTOMERS where ID='${customerId}'</statement>
        <statement>select count(*) from ERRORS</statement>
        <statement>select ID from ORDERS where DESC LIKE 'Def%'</statement>
        <statement>select DESCRIPTION from ORDERS where ID='${id}'</statement>

        <validate column="ID" value="1"/>
        <validate column="NAME" value="Christoph"/>
        <validate column="COUNT(*)" value="${rowsCount}"/>
        <validate column="DESCRIPTION" value="null"/>
    </sql>
</spring:beans>
```

The action offers a wide range of validating functionality for database result sets. First of all you have to select the data via SQL statements. Here again you have the choice to use inline SQL statements or external file resource pattern.

The result sets are validated through <validate> elements. It is possible to do a detailed check on every selected column of the result set. Simply refer to the selected column name in order to validate its value. The usage of test variables is supported as well as database expressions like count(), avg(), min(), max().

You simply define the <validate> entry with the column name as the "column" attribute and any expected value expression as expected "value". The framework then will check the column to fit the expected value and raise validation errors in case of mismatch.

Looking at the first SELECT statement in the example you will see that test variables are supported in the SQL statements. The framework will replace the variable with its respective value before sending it to the database.

In the validation section variables can be used too. Look at the third validation entry, where the variable "${rowsCount}" is used. The last validation in this example shows, that ***NULL*** values are also supported as expected values.

If a single validation happens to fail, the whole action will fail with respective validation errors.

> The validation with ***"<validate column="..." value="..."/>"*** meets single row result sets as you specify a single column control value. In case you have multiple rows in a result set you can pass a variable argument list to the validate method like this:

Java

```
$(sql()
    .query()
    .statement("...")
    .validate("SOME_COLUMN",
        "Value in 1st row",
        "Value in 2nd row",
        "Value in 3rd row",
        "Value in x row")
);
```

XML

```
<sql datasource="myDataSource">
    <statements>
        <statement>...</statement>
    </statement>
    <validate column="SOME_COLUMN">
      <values>
          <value>Value in 1st row</value>
          <value>Value in 2nd row</value>
          <value>Value in 3rd row</value>
          <value>Value in x row</value>
      </values>
    </validate>
</sql>
```

YAML

```
- sql:
  dataSource: myDataSource
  statements:
    - statement: {}
  validate:
    - column: "SOME_COLUMN"
      values:
        - "Value in 1st row"
        - "Value in 2nd row"
        - "Value in 3rd row"
        - "Value in x row"
```

Spring XML

```
<sql dataSource="myDataSource">
    <statement>...</statement>
    <validate column="SOME_COLUMN">
      <values>
          <value>Value in 1st row</value>
          <value>Value in 2nd row</value>
          <value>Value in 3rd row</value>
          <value>Value in x row</value>
      </values>
    </validate>
</sql>
```

Next example shows how to work with multiple row result sets and multiple values to expect within one column:

Java

```
$(sql()
    .dataSource(myDataSource)
    .query()
    .statement("select WEEKDAY as DAY, DESCRIPTION from WEEK")
    .validate("DAY",
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "@ignore@",
        "@ignore@")
    .validate("DESCRIPTION",
        "I hate Mondays!",
        "Tuesday is sports day",
        "The mid of the week",
        "Thursday we play chess",
        "Friday, the weekend is near!",
        "@ignore@",
        "@ignore@")
);
```

XML

```
<sql datasource="myDataSource">
    <statements>
        <statement>select WEEKDAY as DAY, DESCRIPTION from WEEK</statement>
    </statements>
    <validate column="DAY">
      <values>
          <value>Monday</value>
          <value>Tuesday</value>
          <value>Wednesday</value>
          <value>Thursday</value>
          <value>Friday</value>
          <value>@ignore@</value>
          <value>@ignore@</value>
      </values>
    </validate>
    <validate column="DESCRIPTION">
      <values>
          <value>I hate Mondays!</value>
          <value>Tuesday is sports day</value>
          <value>The mid of the week</value>
          <value>Thursday we play chess</value>
          <value>Friday, the weekend is near!</value>
          <value>@ignore@</value>
          <value>@ignore@</value>
      </values>
    </validate>
</sql>
```

YAML

```
- sql:
  dataSource: "myDataSource"
  statements:
    - statement: "select WEEKDAY as DAY, DESCRIPTION from WEEK"
  validate:
    - column: "SOME_COLUMN"
      values:
        - "Value in 1st row"
        - "Value in 2nd row"
        - "Value in 3rd row"
        - "Value in x row"
```

Spring XML

```
<sql datasource="myDataSource">
    <statement>select WEEKDAY as DAY, DESCRIPTION from WEEK</statement>
    <validate column="DAY">
      <values>
          <value>Monday</value>
          <value>Tuesday</value>
          <value>Wednesday</value>
          <value>Thursday</value>
          <value>Friday</value>
          <value>@ignore@</value>
          <value>@ignore@</value>
      </values>
    </validate>
    <validate column="DESCRIPTION">
      <values>
          <value>I hate Mondays!</value>
          <value>Tuesday is sports day</value>
          <value>The mid of the week</value>
          <value>Thursday we play chess</value>
          <value>Friday, the weekend is near!</value>
          <value>@ignore@</value>
          <value>@ignore@</value>
      </values>
    </validate>
</sql>
```

For the validation of multiple rows the `<***validate***>` element is able to host a list of control values for a column. As you can see from the example above, you have to add a control value for each row in the result set. This also means that we have to take care of the total number of rows. Fortunately we can use the ignore placeholder, in order to skip the validation of a specific row in the result set. Functions and variables are supported as usual.

> It is important, that the control values are defined in the correct order, because they are compared one on one with the actual result set coming from database query. You may need to add "order by" SQL expressions to get the right order of rows returned. If any of the values fails in validation or the total number of rows is not equal, the whole action will fail with respective validation errors.

8.3.3. Transaction management

By default no transactions are used when Citrus executes SQL statements on a datasource. You can enable transaction management by selecting a transaction manager.

Java

```
$(sql()
    .dataSource(myDataSource)
    .transactionManager(transactionManager)
    .transactionTimeout(15000)
    .transactionIsolationLevel("ISOLATION_READ_COMMITTED")
    .statement("DELETE FROM CUSTOMERS")
    .statement("DELETE FROM ORDERS")
);
```

XML

```
<sql datasource="someDataSource">
    <transaction>
        <manager>"someTransactionManager"</manager>
        <timeout>"15000"</timeout>
        <isolation-level>"ISOLATION_READ_COMMITTED"</isolation-level>
    </transaction>
    <statements>
        <statement>DELETE FROM CUSTOMERS</statement>
        <statement>DELETE FROM ORDERS</statement>
    </statements>
</sql>
```

YAML

```
- sql:
    datasource: "myDataSource"
    transaction:
      manager: "someTransactionManager"
      timeout: 15000
      isolation-level: "ISOLATION_READ_COMMITTED"
    statements:
      - statement: "DELETE FROM CUSTOMERS"
      - statement: "DELETE FROM ORDERS"
```

Spring XML

```
<sql datasource="myDataSource"
     transaction-manager="someTransactionManager"
     transaction-timeout="15000"
     transaction-isolation-level="ISOLATION_READ_COMMITTED">
    <statement>DELETE FROM CUSTOMERS</statement>
    <statement>DELETE FROM ORDERS</statement>
</sql>
```

The *transaction-manager* attribute references a Spring bean of type

"*org.springframework.transaction.PlatformTransactionManager*". You can add this transaction manager to the Spring bean configuration:

Spring Bean

```
@Bean
public TransactionManager myTransactionManager(DataSource myDataSource) {
    return new DataSourceTransactionManager(myDataSource);
}
```

Spring XML

```
<bean id="myTransactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <constructor-arg ref="myDataSource"/>
</bean>
```

The transaction isolation level as well as the transaction timeout get set on the transaction definition used during SQL statement execution. The isolation level should evaluate to one of the constants given in "*org.springframework.transaction.TransactionDefinition*". Valid isolation level are:

- ISOLATION_DEFAULT
- ISOLATION_READ_UNCOMMITTED
- ISOLATION_READ_COMMITTED
- ISOLATION_REPEATABLE_READ
- ISOLATION_SERIALIZABLE

8.3.4. Groovy SQL result set validation

Groovy provides great support for accessing Java list objects and maps. As a Java SQL result set is nothing but a list of map representations, where each entry in the list defines a row in the result set and each map entry represents the columns and values. So with Groovy's list and map access we have great possibilities to validate a SQL result set - out of the box.

Java

```
$(sql()
    .dataSource(myDataSource)
    .query()
    .statement("select ORDERTYPE, STATUS from ORDERS where ID='${orderId}'")
    .validateScript("assert rows.size() == 2;" +
            "assert rows[0].ID == '1';" +
            "assert rows[0].STATUS == 'in progress';", "groovy")
);
```

XML

```
<sql datasource="myDataSource">
    <statements>
        <statement>select ID from CUSTOMERS where NAME='${customerName}'</statement>
        <statement>select ORDERTYPE, STATUS from ORDERS where
ID='${orderId}'</statement>
    </statements>
    <validate>
        <script type="groovy">
          assert rows.size() == 2
          assert rows[0].ID == '1'
          assert rows[1].STATUS == 'in progress'
          assert rows[1] == [ORDERTYPE:'SampleOrder', STATUS:'in progress']
        </script>
    </validate>
</sql>
```

YAML

```
- sql:
  dataSource: "myDataSource"
  statements:
    - statement: "select ID from CUSTOMERS where NAME='${customerName}'"
    - statement: "select ORDERTYPE, STATUS from ORDERS where ID='${orderId}'"
  validate:
    - script:
        type: "groovy"
        value: |
          assert rows.size() == 2
          assert rows[0].ID == '1'
          assert rows[1].STATUS == 'in progress'
          assert rows[1] == [ORDERTYPE:'SampleOrder', STATUS:'in progress']
```

Spring XML

```
<sql datasource="myDataSource">
    <statement>select ID from CUSTOMERS where NAME='${customerName}'</statement>
    <statement>select ORDERTYPE, STATUS from ORDERS where ID='${orderId}'</statement>

    <validate-script type="groovy">
        assert rows.size() == 2
        assert rows[0].ID == '1'
        assert rows[1].STATUS == 'in progress'
        assert rows[1] == [ORDERTYPE:'SampleOrder', STATUS:'in progress']
    </validate-script>
</sql>
```

As you can see Groovy provides fantastic access methods to the SQL result set. We can browse the

result set with named column values and check the size of the result set. We are also able to search for an entry, iterate over the result set and have other helpful operations. For a detailed description of the list and map handling in Groovy my advice for you is to have a look at the official Groovy documentation.

> In general other script languages do also support this kind of list and map access. For now we just have implemented the Groovy script support, but the framework is ready to work with all other great script languages out there, too (e.g. Scala, Clojure, Fantom, etc.). So if you prefer to work with another language join and help us implement those features.

8.3.5. Save result set values

Now the validation of database entries is a very powerful feature but sometimes we simply do not know the persisted content values. The test may want to read database entries into test variables without validation. Citrus is able to do that with the following <extract> expressions:

Java

```
$(sql()
    .dataSource(myDataSource)
    .query()
        .statement("select ID from CUSTOMERS where NAME='${customerName}'")
        .statement("select STATUS from ORDERS where ID='${orderId}'")
    .extract("ID", "customerId")
    .extract("STATUS", "orderStatus")
);
```

XML

```
<sql datasource="myDataSource">
    <statements>
        <statement>select ID from CUSTOMERS where NAME='${customerName}'</statement>
        <statement>select STATUS from ORDERS where ID='${orderId}'</statement>
    </statements>
    <extract  column="ID" variable="customerId"/>
    <extract  column="STATUS" variable="orderStatus"/>
</sql>
```

YAML

```
- sql:
  dataSource: "myDataSource"
  statements:
    - statement: "select ID from CUSTOMERS where NAME='${customerName}'"
    - statement: "select ORDERTYPE, STATUS from ORDERS where ID='${orderId}'"
  extract:
    - column: "ID"
      variable: "customerId"
    - column: "STATUS"
      variable: "orderStatus"
```

Spring XML

```
<sql datasource="testDataSource">
    <statement>select ID from CUSTOMERS where NAME='${customerName}'</statement>
    <statement>select STATUS from ORDERS where ID='${orderId}'</statement>

    <extract column="ID" variable="${customerId}"/>
    <extract column="STATUS" variable="${orderStatus}"/>
</sql>
```

We can save the database column values directly to test variables. Of course you can combine the value extraction with the normal column validation described earlier in this chapter. Please keep in mind that we can not use these operations on result sets with multiple rows. Citrus will always use the first row in a result set.

8.4. Sleep

This action shows how to make the test framework execution stop for a given amount of time. The attribute 'time' defines the amount of time to wait in seconds. As shown in the next example decimal values are supported too. When no waiting time is specified the default time of 50000 milliseconds applies.

Java

```
@CitrusTest
public void sleepTest() {
    $(sleep().seconds(3.5));

    $(sleep().milliseconds(500));

    $(sleep());
}
```

XML

```
<test name="SleepTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <sleep seconds="3.5"/>

        <sleep milliseconds="500"/>

        <sleep/>
    </actions>
</test>
```

YAML

```
name: SleepTest
actions:
  - sleep:
      seconds: "3.5"
  - sleep:
      milliseconds: "500"
  - sleep: {}
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="SleepTest">
        <actions>
            <sleep seconds="3.5"/>

            <sleep milliseconds="500"/>

            <sleep/>
        </actions>
    </testcase>
</spring:beans>
```

When should somebody use this action? To us this action was always very useful in case the test needed to wait until an application had done some work. For example in some cases the application took some time to write some data into the database. We waited then a small amount of time in order to avoid unnecessary test failures, because the test framework simply validated the database too early. Or as another example the test may wait a given time until retry mechanisms are triggered in the tested application and then proceed with the test actions.

8.5. Delay

This action shows how to make the test framework execution stop for a given amount of time. The attribute 'time' defines the amount of time to wait in seconds. As shown in the next example

decimal values are supported too. When no waiting time is specified the default time of 50000 milliseconds applies.

Java

```
@CitrusTest
public void delayTest() {
    $(delay().seconds(3.5));

    $(delay().milliseconds(500));

    $(delay());
}
```

XML

```
<test name="DelayTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <delay seconds="3.5"/>

        <delay milliseconds="500"/>

        <delay/>
    </actions>
</test>
```

YAML

```
name: DelayTest
actions:
  - delay:
      seconds: "3.5"
  - delay:
      milliseconds: "500"
  - delay: {}
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="DelayTest">
        <actions>
            <delay seconds="3.5"/>

            <delay milliseconds="500"/>

            <delay/>
        </actions>
    </testcase>
</spring:beans>
```

When should somebody use this action? To us this action was always very useful in case the test needed to wait until an application had done some work. For example in some cases the application took some time to write some data into the database. We waited then a small amount of time in order to avoid unnecessary test failures, because the test framework simply validated the database too early. Or as another example the test may wait a given time until retry mechanisms are triggered in the tested application and then proceed with the test actions.

8.6. Java

The test framework is written in Java and runs inside a Java virtual machine. The functionality of calling other Java objects and methods in this same Java VM through Java Reflection is self-evident. With this action you can call any Java API available at runtime through the specified Java classpath.

The action syntax looks like follows:

Java

```
@CitrusTest
public void javaActionTest() {
    $(java("org.citrusframework.test.util.InvocationDummy")
        .constructorArgs("Test Invocation")
        .method("invoke")
        .methodArgs(new String[] { "1","2" })
    );

    $(java(new InvocationDummy("Test Invocation"))
        .method("invoke")
        .methodArgs(new Object[] { 4, "Test Invocation", true })
    );

    $(java(InvocationDummy.class)
        .method("main")
        .methodArgs(new String[] { "4", "Test Invocation", "true" })
    );
}
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="JavaTest">
        <actions>
            <java class="org.citrusframework.test.util.InvocationDummy">
                <constructor>
                    <argument type="">Test Invocation</argument>
                </constructor>
                <method name="invoke">
                    <argument type="String[]">1,2</argument>
                </method>
            </java>

            <java class="org.citrusframework.test.util.InvocationDummy">
                <constructor>
                    <argument type="">Test Invocation</argument>
                </constructor>
                <method name="invoke">
                    <argument type="int">4</argument>
                    <argument type="String">Test Invocation</argument>
                    <argument type="boolean">true</argument>
                </method>
            </java>

            <java class="org.citrusframework.test.util.InvocationDummy">
                <method name="main">
                    <argument type="String[]">4,Test,true </argument>
                </method>
            </java>
        </actions>
    </testcase>
</spring:beans>
```

The Java class is specified by fully qualified class name. Constructor arguments are added using the <constructor> element with a list of <argument> child elements. The type of the argument is defined within the respective attribute "type". By default the type would be String.

The invoked method on the Java object is simply referenced by its name. Method arguments do not bring anything new after knowing the constructor argument definition, do they?.

Method arguments support data type conversion too, even string arrays (useful when calling CLIs). In the third action in the example code you can see that colon separated strings are automatically converted to string arrays.

Simple data types are defined by their name (int, boolean, float etc.). Be sure that the invoked method and class constructor fit your arguments and vice versa, otherwise you will cause errors at runtime.

Besides instantiating a fully new object instance for a class how about reusing a bean instance available in Spring bean container. Simply use the **ref** attribute and refer to an existing bean in Spring application context.

Java

```
InvocationDummy invocationDummy = new InvocationDummy("Test Invocation");
$(java(invocationDummy)
    .method("invoke")
    .methodArgs(new Object[] { 4, "Test Invocation", true })
);
```

Spring XML

```
<java ref="invocationDummy">
    <method name="invoke">
        <argument type="int">4</argument>
        <argument type="String">Test Invocation</argument>
        <argument type="boolean">true</argument>
    </method>
</java>

<bean id="invocationDummy" class="org.citrusframework.test.util.InvocationDummy"/>
```

The method is invoked on the Spring bean instance. This is very useful as you can inject other objects (e.g. via Autowiring) to the Spring bean instance before method invocation in test takes place. This enables you to execute any Java logic inside a test case.

8.7. Expect timeout

In some cases it might be necessary to validate that a message is **not** present on a destination. This means that this action expects a timeout when receiving a message from an endpoint destination. For instance the tester intends to ensure that no message is sent to a certain destination in a time period. In that case the timeout would not be a test aborting error but the expected behavior. And in contrast to the normal behavior when a message is received in the time period the test will fail with error.

In order to validate such a timeout situation the action <expectTimout> shall help. The usage is very simple as the following example shows:

Java

```
@Autowired
@Qualifier("myEndpoint")
private Endpoint myEndpoint;

@CitrusTest
public void expectTimeoutTest() {
    $(expectTimeout()
            .endpoint(myEndpoint)
            .timeout(500)
    );
}
```

XML

```
<test name="ExpectTimeoutTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <expect-timeout endpoint="myEndpoint" wait="500"/>
    </actions>
</test>
```

YAML

```
name: ExpectTimeoutTest
actions:
  - expect-timeout:
      endpoint: "myEndpoint"
      wait: "500"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="ExpectTimeoutTest">
        <actions>
            <expect-timeout endpoint="myEndpoint" wait="500"/>
        </actions>
    </testcase>
</spring:beans>
```

The action offers two attributes:

endpoint Reference to a message endpoint that will try to receive messages.

wait/timeout Time period to wait for messages to arrive

Sometimes you may want to add some selector on the timeout receiving action. This way you can

very selective check on a message to not be present on a message destination. This is possible with defining a message selector on the test action as follows.

Java

```
@Autowired
@Qualifier("myEndpoint")
private Endpoint myEndpoint;

@CitrusTest
public void expectTimeoutTest() {
    $(expectTimeout()
            .endpoint(myEndpoint)
            .timeout(500)
            .selector("MessageId = '123456789'")
    );
}
```

XML

```
<test name="ExpectTimeoutTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <expect-timeout endpoint="myEndpoint" wait="500">
            <selector>MessageId='123456789'</selector>
        </expect-timeout>
    </actions>
</test>
```

YAML

```
name: ExpectTimeoutTest
actions:
  - expect-timeout:
      endpoint: "myEndpoint"
      wait: "500"
      selector:
        value: "MessageId='123456789'"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="ExpectTimeoutTest">
        <actions>
            <expect-timeout endpoint="myEndpoint" wait="500">
                <select>MessageId='123456789'<select/>
            </expect-timeout>
        </actions>
    </testcase>
</spring:beans>
```

8.8. Echo

The <echo> action prints messages to the console/logger. This functionality is useful when debugging test runs. The property "message" defines the text that is printed. Tester might use it to print out debug messages and variables as shown in the next code example:

Java

```
@CitrusTest
public void echoTest() {
    variable("today", "citrus:currentDate()");

    $(echo().message("Hello Test Framework"));
    $(echo().message("Current date is: ${today}"));
}
```

XML

```
<test name="EchoTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <variables>
        <variable name="today" value="citrus:currentDate()"/>
    </variables>
    <actions>
        <echo>
            <message>Hello Test Framework</message>
        </echo>

        <echo>
            <message>Current date is: ${today}</message>
        </echo>
    </actions>
</test>
```

YAML

```
name: EchoTest
variables:
  - name: "today"
    value: "citrus:currentDate()"
actions:
  - echo:
      message: "Hello Test Framework"
  - echo:
      message: "Current date is: ${today}"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="EchoTest">
        <variables>
            <variable name="today" value="citrus:currentDate()"/>
        </variables>
        <actions>
            <echo>
                <message>Hello Test Framework</message>
            </echo>

            <echo>
                <message>Current date is: ${today}</message>
            </echo>
        </actions>
    </testcase>
</spring:beans>
```

Result on the console:

```
Hello Test Framework
Current time is: 05.08.2008
```

8.9. Print

The <print> action prints messages to the console/logger. This functionality is useful when debugging test runs. The property "message" defines the text that is printed. Tester might use it to print out debug messages and variables as shown the next code example:

Java

```
@CitrusTest
public void printTest() {
    variable("today", "citrus:currentDate()");

    $(print().message("Hello Test Framework"));
    $(print().message("Current date is: ${today}"));
}
```

XML

```
<test name="PrintTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <variables>
        <variable name="today" value="citrus:currentDate()"/>
    </variables>
    <actions>
        <print>
            <message>Hello Test Framework</message>
        </print>

        <print>
            <message>Current date is: ${today}</message>
        </print>
    </actions>
</test>
```

YAML

```
name: PrintTest
variables:
  - name: "today"
    value: "citrus:currentDate()"
actions:
  - print:
      message: "Hello Test Framework"
  - print:
      message: "Current date is: ${today}"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="PrintTest">
        <variables>
            <variable name="today" value="citrus:currentDate()"/>
        </variables>
        <actions>
            <print>
                <message>Hello Test Framework</message>
            </print>

            <print>
                <message>Current date is: ${today}</message>
            </print>
        </actions>
    </testcase>
</spring:beans>
```

Result on the console:

```
Hello Test Framework
Current time is: 05.08.2008
```

8.10. Stop time

Time measurement during a test can be very helpful. The `<trace-time>` action creates and monitors multiple timelines. The action offers the attribute *id* to identify a timeline. The tester can of course use more than one timeline with different ids simultaneously.

Read the next example, and you will understand the mix of different timelines:

Java

```
@CitrusTest
public void sleepTest() {
    $(stopTime());

    $(stopTime().id("time_line_id"));

    $(sleep().seconds(3.5));

    $(stopTime().id(" time_line_id "));

    $(sleep().milliseconds(5000));

    $(stopTime());

    $(stopTime().id(" time_line_id "));

}
```

XML

```
<test name="SleepTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <stop-time/>

        <stop-time id="time_line_id"/>

        <sleep seconds="3.5"/>

        <stop-time id=" time_line_id "/>

        <sleep milliseconds="5000"/>

        <stop-time/>

        <stop-time id=" time_line_id "/>
    </actions>
</test>
```

YAML

```
name: SleepTest
actions:
  - stop-time: {}

  - stop-time:
      id: "time_line_id"

  - sleep:
      seconds: "3.5"

  - stop-time:
      id: "time_line_id"

  - sleep:
      milliseconds: "5000"

  - stop-time: {}

  - stop-time:
      id: "time_line_id"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="SleepTest">
        <actions>
            <trace-time/>

            <trace-time id="time_line_id"/>

            <sleep seconds="3.5"/>

            <trace-time id=" time_line_id "/>

            <sleep milliseconds="5000"/>

            <trace-time/>

            <trace-time id=" time_line_id "/>
        </actions>
    </testcase>
</spring:beans>
```

The test output looks like follows:

```
Starting TimeWatcher:
Starting TimeWatcher: time_line_id
TimeWatcher time_line_id after 3500 milliseconds
TimeWatcher after 8500 seconds
TimeWatcher time_line_id after 8500 milliseconds
```

Timeline ids should not exist as test variables before the action is called for the first time. This would break the timeline initialization.

In case no timeline id is specified the framework will measure the time for a default timeline. To print out the current elapsed time for a timeline you simply have to place the `<trace-time> action into the action chain again and again, using the respective timeline identifier. The elapsed time will be printed out to the console every time.

Each timeline is stored as test variable in the test case. By default you will have the following test variables set for each timeline:

CITRUS_TIMELINE	first timestamp of timeline
CITRUS_TIMELINE_VALUE	latest time measurement value (time passed since first timestamp in milliseconds)

According to your timeline id you will get different test variable names. Also you can customize the time value suffix (default: **_VALUE**):

Java

```
@CitrusTest
public void sleepTest() {
    $(stopTime().id("custom_watcher").suffix("_1st"));

    $(sleep());

    $(stopTime().id(" custom_watcher ").suffix("_2nd"));
}
```

XML

```
<test name="SleepTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <stop-time id="custom_watcher" suffix="_1st"/>

        <sleep/>

        <stop-time id="custom_watcher" suffix="_2nd"/>
    </actions>
</test>
```

YAML

```
name: SleepTest
actions:
  - stop-time:
      id: "custom_watcher"
      suffix: "_1st"

  - sleep: {}

  - stop-time:
      id: "custom_watcher"
      suffix: "_2nd"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="SleepTest">
        <actions>
            <trace-time id="custom_watcher" suffix="_1st"/>

            <sleep/>

            <trace-time id="custom_watcher" suffix="_2nd"/>
        </actions>
    </testcase>
</spring:beans>
```

You will get following test variables set:

custom_watcher	first timestamp of timeline
custom_watcher_1st	time passed since start

custom_watcher_2nd time passed since start

Of course using the same suffix multiple times will overwrite the timestamps in test variables.

8.11. Create variables

As you know variables usually are defined at the beginning of the test case (test-variables). It might also be helpful to reset existing variables as well as to define new variables during the test. The action `<create-variables>` is able to declare new variables or overwrite existing ones.

Java

```
@CitrusTest
public void createVariableTest() {
    variable("myVariable", "12345");
    variable("id", "54321");

    $(echo()
        .message("Current variable value: ${myVariable}"));

    $(createVariable("myVariable", "${id}"));
    $(createVariable("newVariable", "this is a test"));

    $(echo()
        .message("Current variable value: ${myVariable}"));

    $(echo()
        .message("New variable 'newVariable' has the value: ${newVariable}"));
}
```

XML

```
<test name="CreateVariablesTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <variables>
        <variable name="myVariable" value="12345"/>
        <variable name="id" value="54321"/>
    </variables>
    <actions>
        <echo>
            <message>Current variable value: ${myVariable}</message>
        </echo>

        <create-variables>
            <variable name="myVariable" value="${id}"/>
            <variable name="newVariable" value="'this is a test'"/>
        </create-variables>

        <echo>
            <message>Current variable value: ${myVariable} </message>
        </echo>

        <echo>
            <message>New variable 'newVariable' has the value:
${newVariable}</message>
        </echo>
    </actions>
</test>
```

YAML

```
name: "CreateVariablesTest"
variables:
  - name: "myVariable"
    value: "12345"
  - name: "id"
    value: "54321"
actions:
  - echo:
      message: "Current variable value: ${myVariable}"
  - create-variable:
      variables:
        - name: "myVariable"
          value: "${id}"
        - name: "newVariable"
          value: "this is a test"
  - echo:
      message: "Current variable value: ${myVariable}"
  - echo:
      message: "New variable 'newVariable' has the value: ${newVariable}"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="createVariablesTest">
        <variables>
            <variable name="myVariable" value="12345"/>
            <variable name="id" value="54321"/>
        </variables>
        <actions>
            <echo>
                <message>Current variable value: ${myVariable}</message>
            </echo>

            <create-variables>
                <variable name="myVariable" value="${id}"/>
                <variable name="newVariable" value="'this is a test'"/>
            </create-variables>

            <echo>
                <message>Current variable value: ${myVariable} </message>
            </echo>

            <echo>
                <message>
                  New variable 'newVariable' has the value: ${newVariable}
                </message>
            </echo>
        </actions>
    </testcase>
</spring:beans>
```

Please note the difference between the **variable()** method and the **createVariable()** method. The first initializes the test case with the test variables. So all variables defined with this method are valid from the very beginning of the test. In contrary to that the **createVariable()** is executed within the test action chain. The newly created variables are then valid for the rest of the test. Trailing actions can reference the variables as usual with the variable expression.

8.12. Create endpoints

Endpoints represent the connectivity to other services in Citrus. An endpoint may create producers and consumers to send and receive messages as part of the test.

Users are able to declare endpoints in the form of a project wide configuration in Citrus. Also, users are able to use dynamic endpoint definitions with a `send` or `receive` action where the endpoint is defined as a URI.

Java

```
@CitrusTest
public void useDynamicEndpoints() {
    $(send()
        .endpoint("kafka:bookings?server=${kafka.bootstrapServer}")
        .message()
        // ...
    );

    $(receive()
        .endpoint("kafka:bookings?server=${kafka.bootstrapServer}")
        .message()
        // ...
    );
}
```

XML

```
<test name="UseDynamicEndpoints"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <send endpoint="kafka:bookings?server=${kafka.bootstrapServer}">
            <message>
              <!--
              ...
              -->
            </message>
        </send>
        <receive endpoint="kafka:bookings?server=${kafka.bootstrapServer}">
            <message>
              <!--
              ...
              -->
            </message>
        </receive>
    </actions>
</test>
```

YAML

```yaml
name: "UseDynamicEndpoints"
actions:
  - send:
      endpoint: "kafka:bookings?server=${kafka.bootstrapServer}"
      message:
        # ...
  - receive:
      endpoint: "kafka:bookings?server=${kafka.bootstrapServer}"
      message:
        # ...
```

Spring XML

```xml
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="useDynamicEndpoints">
        <actions>
            <send endpoint="kafka:bookings?server=${kafka.bootstrapServer}">
                <message>
                  <!--
                  ...
                  -->
                </message>
            </send>
            <receive endpoint="kafka:bookings?server=${kafka.bootstrapServer}">
                <message>
                  <!--
                  ...
                  -->
                </message>
            </receive>
        </actions>
    </testcase>
</spring:beans>
```

The dynamic endpoint receives its properties in the form of endpoint URI parameters: `kafka:bookings?server=${kafka.bootstrapServer}`

The test needs to repeat the same configuration endpoint URI several times with multiple send/receive test actions.

This can be improved by using the `createEndpoint()` test action that declares the endpoint once with a given name. Following test actions are able to reference the endpoint as usual by its give name.

Java

```
@CitrusTest
public void useDynamicEndpoints() {
    $(createEndpoint()
        .endpointName("bookings")
        .uri("kafka:bookings?server=${kafka.bootstrapServer}")
    );

    $(send()
        .endpoint("bookings")
        .message()
        // ...
    );

    $(receive()
        .endpoint("bookings")
        .message()
        // ...
    );
}
```

XML

```
<test name="UseDynamicEndpoints"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <create-endpoint name="bookings"
                         uri="kafka:bookings?server=${kafka.bootstrapServer}"/>

        <send endpoint="bookings">
            <message>
              <!--
              ...
              -->
            </message>
        </send>
        <receive endpoint="bookings">
            <message>
              <!--
              ...
              -->
            </message>
        </receive>
    </actions>
</test>
```

YAML

```yaml
name: "UseDynamicEndpoints"
actions:
  - createEndpoint:
      name: bookings
      uri: "kafka:bookings?server=${kafka.bootstrapServer}"
  - send:
      endpoint: "bookings"
      message:
        # ...
  - receive:
      endpoint: "bookings"
      message:
        # ...
```

Spring XML

```xml
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

As an alternative syntax aou may also declare the properties with key-value pairs in the `createEndpoint()` action:

Java

```java
@CitrusTest
public void useDynamicEndpoints() {
    $(createEndpoint()
        .endpointName("bookings")
        .type("kafka")
        .property("topic", "bookings")
        .property("server", "${kafka.bootstrapServer}")
    );
}
```

XML

```
<test name="UseDynamicEndpoints"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <create-endpoint name="bookings" type="kafka">
          <properties>
            <property name="topic" value="bookings"/>
            <property name="server" value="${kafka.bootstrapServer}"/>
          </properties>
        </create-endpoint>
    </actions>
</test>
```

YAML

```
name: "UseDynamicEndpoints"
actions:
  - createEndpoint:
      name: bookings
      type: "kafka"
      properties:
        topic: bookings
        server: "${kafka.bootstrapServer}"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

8.13. Trace variables

You already know the <echo> action that prints messages to the console or logger. The <trace-variables> action is specially designed to trace all currently valid test variables to the console. This was mainly used by us for debug reasons. The usage is quite simple:

Java

```
@CitrusTest
public void traceVariablesTest() {
    variable("myVariable", "12345");
    variable("nextVariable", "54321");

    $(traceVariables("myVariable", "nextVariable"));
    $(traceVariables());
}
```

XML

```
<test name="TraceVariablesTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <variables>
        <variable name="myVariable" value="12345"/>
        <variable name="nextVariable" value="54321"/>
    </variables>
    <actions>
        <trace>
            <variable name="myVariable"/>
            <variable name="nextVariable"/>
        </trace>

        <trace-variables/>
    </actions>
</test>
```

YAML

```
name: TraceVariablesTest
variables:
  - name: "myVariable"
    value: "12345"
  - name: "nextVariable"
    value: "54321"
actions:
  - trace:
      variables:
          - name: "myVariable"
          - name: "nextVariable"
  - trace: {}
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="TraceVariablesTest">
        <variables>
            <variable name="myVariable" value="12345"/>
            <variable name="nextVariable" value="54321"/>
        </variables>
        <actions>
            <trace-variables>
                <variable name="myVariable"/>
                <variable name="nextVariable"/>
            </trace-variables>

            <trace-variables/>
        </actions>
    </testcase>
</spring:beans>
```

Simply add the <trace-variables> action to your action chain and all variables will be printed out to the console. You are able to define a special set of variables by using the <variable> child elements. See the output that was generated by the test example above:

```
Current value of variable myVariable = 12345
Current value of variable nextVariable = 54321
```

8.14. Transform

The `<***transform***>` action transforms XML fragments with XSLT in order to construct various XML representations. The transformation result is stored into a test variable for further usage. The property **xml-data** defines the XML source, that is going to be transformed, while **xslt-data** defines the XSLT transformation rules. The attribute **variable** specifies the target test variable which receives the transformation result. The tester might use the action to transform XML messages as shown in the next code example:

Java

```
@CitrusTest
public void transformTest() {
    $(transform()
        .variable("result")
        .source("""
            <TestRequest>
                <Message>Hello World!</Message>
            </TestRequest>
        """)
        .xslt("""
            <xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
              <xsl:template match="/">
                  <html>
                      <body>
                          <h2>Test Request</h2>
                          <p>Message: <xsl:value-of select="TestRequest/Message"/></p>
                      </body>
                  </html>
              </xsl:template>
            </xsl:stylesheet>
        """)
    );

    $(echo().message("${result}"));
}
```

XML

```
<test name="TransformTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <transform variable="result">
            <source>
              <![CDATA[
                  <TestRequest>
                      <Message>Hello World!</Message>
                  </TestRequest>
              ]]>
            </source>
            <xslt>
              <![CDATA[
                  <xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
                      <xsl:template match="/">
                          <html>
                              <body>
                                  <h2>Test Request</h2>
                                  <p>Message: <xsl:value-of
select="TestRequest/Message"/></p>
                              </body>
                          </html>
                      </xsl:template>
                  </xsl:stylesheet>
              ]]>
            </xslt>
        </transform>
        <echo>
          <message>${result}</message>
        </echo>
    </actions>
</test>
```

YAML

```
name: TransformTest
actions:
  - transform:
      variable: result
      source:
        value: |
          <TestRequest>
              <Message>Hello World!</Message>
          </TestRequest>
      xslt:
        value: |
          <xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
              <xsl:template match="/">
              <html>
                  <body>
                      <h2>Test Request</h2>
                      <p>Message: <xsl:value-of select="TestRequest/Message"/></p>
                  </body>
              </html>
              </xsl:template>
          </xsl:stylesheet>
  - echo:
      message: '${result}'
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="TransformTest">
        <actions>
            <transform variable="result">
                <xml-data>
                  <![CDATA[
                      <TestRequest>
                          <Message>Hello World!</Message>
                      </TestRequest>
                  ]]>
                </xml-data>
                <xslt-data>
                    <![CDATA[
                    <xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
                        <xsl:template match="/">
                          <html>
                              <body>
                                  <h2>Test Request</h2>
                                  <p>Message: <xsl:value-of
select="TestRequest/Message"/></p>
                              </body>
                          </html>
                        </xsl:template>
                    </xsl:stylesheet>
                    ]]>
                </xslt-data>
            </transform>
            <echo>
              <message>${result}</message>
            </echo>
        </actions>
    </testcase>
</spring:beans>
```

The transformation above results to:

```
  <html>
      <body>
          <h2>Test Request</h2>
          <p>Message: Hello World!</p>
      </body>
  </html>
```

In the example we used CDATA sections to define the transformation source as well as the XSL transformation rules. As usual you can also use external file resources here. The transform action

with external file resources looks like follows:

Java

```
@CitrusTest
public void transformTest() {
    $(transform()
        .variable("result")
        .source(new ClassPathResource("org/citrusframework/actions/transform-
source.xml"))
        .xslt(new ClassPathResource("org/citrusframework/actions/transform.xslt"))
    );

    $(echo().message("${result}"));
}
```

XML

```
<test name="TransformTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <transform variable="result">
            <source file="classpath:transform-source.xml"/>
            <xslt file="classpath:transform.xslt"/>
        </transform>
        <echo>
          <message>${result}</message>
        </echo>
    </actions>
</test>
```

YAML

```
name: TransformTest
actions:
  - transform:
      variable: result
      source:
        file: "classpath:transform-source.xml"
      xslt:
        file: "classpath:transform.xslt"
  - echo:
      message: '${result}'
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="TransformTest">
        <actions>
            <transform variable="result">
                <xml-resource file="classpath:transform-source.xml"/>
                <xslt-resource file="classpath:transform.xslt"/>
            </transform>
            <echo>
              <message>${result}</message>
            </echo>
        </actions>
    </testcase>
</spring:beans>
```

Defining multi-line Strings with nested quotes is no fun in Java. So you may want to use external file resources for your scripts as shown in the second part of the example. In fact, you could also use script languages like Groovy or Scala that have much better support for multi-line Strings.

8.15. Groovy script execution

Groovy is an agile dynamic language for the Java Platform. Groovy ships with a lot of very powerful features and fits perfectly with Java as it is based on Java and runs inside the JVM.

The Groovy test action support lives in a separate module. You need to add the module to your project to use the functionality.

citrus-groovy dependency module

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-groovy</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

The Citrus Groovy support might be the entrance for you to write customized test actions. You can easily execute Groovy code inside a test case, just like a normal test action. The whole test context with all variables is available to the Groovy action. This means someone can change variable values or create new variables very easily.

Let's have a look at some examples in order to understand the possible Groovy code interactions in Citrus:

Java

```
@CitrusTest
public void GroovyTest() {
    $(groovy()
        .script("println 'Hello Citrus'")
    );

    $(groovy()
        .script("println 'The variable is: ${time}'")
    );

    $(groovy()
        .scriptResourcePath("classpath:org/citrusframework/script/example.groovy")
    );
}
```

XML

```
<test name="GroovyTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <variables>
        <variable name="time" value="citrus:currentDate()"/>
    </variables>
    <actions>
        <groovy>
            println 'Hello Citrus'
        </groovy>
        <groovy>
            println 'The variable is: ${time}'
        </groovy>
        <groovy file="classpath:org/citrusframework/script/example.groovy"/>
    </actions>
</test>
```

YAML

```
name: GroovyTest
variables:
  - name: "time"
    value: "citrus:currentDate()"
actions:
  - groovy:
      script: "println 'Hello Citrus'"
  - groovy:
      script: |
        println 'The variable is: ${time}'
  - groovy:
      file: classpath:org/citrusframework/script/example.groovy
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="GroovyTest">
      <variables>
        <variable name="time" value="citrus:currentDate()"/>
      </variables>
      <actions>
        <groovy>
            println 'Hello Citrus'
        </groovy>
        <groovy>
            println 'The variable is: ${time}'
        </groovy>
        <groovy resource="classpath:org/citrusframework/script/example.groovy"/>
      </actions>
    </testcase>
</spring:beans>
```

As you can see it is possible to write Groovy code directly into the test case. Citrus will interpret and execute the Groovy code at runtime. As usual nested variable expressions are replaced with respective values. In general this is done in advance before the Groovy code is interpreted. For more complex Groovy code sections which grow in lines of code you can also reference external file resources.

After this basic Groovy code usage inside a test case we might be interested accessing the whole TestContext. The TestContext Java object holds all test variables and function definitions for the test case and can be referenced in Groovy code via simple naming convention. Just access the object reference 'context' and you are able to manipulate the TestContext (e.g. setting a new variable which is directly ready for use in following test actions).

Java

```
@CitrusTest
public void GroovyTest() {
    $(groovy()
        .script("""
            context.setVariable("greetingText","Hello Citrus")
            println context.getVariable("greetingText")
        """)
    );

    $(echo()
        .message("New variable: ${greetingText}")
    );
}
```

XML

```
<test name="GroovyTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <groovy>
            context.setVariable("greetingText","Hello Citrus")
            println context.getVariable("greetingText")
        </groovy>
        <echo>
            <message>New variable: ${greetingText}</message>
        </echo>
    </actions>
</test>
```

YAML

```
name: GroovyTest
actions:
  - groovy:
      script: |
        context.setVariable("greetingText","Hello Citrus")
        println context.getVariable("greetingText")
  - echo:
      message: "New variable: ${greetingText}"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="groovyTest">
      <actions>
        <groovy>
          context.setVariable("greetingText","Hello Citrus")
          println context.getVariable("greetingText")
        </groovy>

        <echo>
          <message>New variable: ${greetingText}</message>
        </echo>
      </actions>
    </testcase>
</spring:beans>
```

The implicit TestContext access that was shown in the previous sample works with a default Groovy script template provided by Citrus. The Groovy code you write in the test case is automatically surrounded with a Groovy script which takes care of handling the TestContext. The default template looks like follows:

Groovy script template

```
import org.citrusframework.*
import org.citrusframework.variable.*
import org.citrusframework.context.TestContext
import org.citrusframework.script.GroovyAction.ScriptExecutor

public class GScript implements ScriptExecutor {
    public void execute(TestContext context) {
        @SCRIPTBODY@
    }
}
```

Your code is placed in substitution to the ***@SCRIPTBODY@*** placeholder. Now you might understand how Citrus handles the context automatically. You can also write your own script templates making more advanced usage of other Java APIs and Groovy code. Just add a script template path to the test action like this:

Java

```
$(groovy()
    .template("classpath:my-custom-template.groovy")
    .script("...")
);
```

XML

```
<groovy script-template="classpath:my-custom-template.groovy">
    <!-- ... -->
</groovy>
```

YAML

```
- groovy:
  script-template: "classpath:my-custom-template.groovy"
  script: "..."
```

Spring XML

```
<groovy script-template="classpath:my-custom-template.groovy">
  <!-- ... -->
</groovy>
```

On the other hand you can disable the automatic script template wrapping in your action at all:

Java

```
$(groovy()
    .useScriptTemplate(false)
    .script("println 'Just use some Groovy code'")
);
```

XML

```
<groovy use-script-template="false">
    println 'Just use some Groovy code'
</groovy>
```

YAML

```
- groovy:
  use-script-template: false
  script: "println 'Just use some Groovy code'"
```

Spring XML

```
<groovy use-script-template="false">
  println 'Just use some Groovy code'
</groovy>
```

The next example deals with advanced Groovy code and writing whole classes. We write a new Groovy class which implements the ScriptExecutor interface offered by Citrus. This interface defines a special execute method and provides access to the whole TestContext for advanced test variables access.

Java

```
@CitrusTest
public void GroovyTest() {
    variable("time", "citrus:currentDate()");

    $(groovy()
        .script("""
            import org.citrusframework.*
            import org.citrusframework.variable.*
            import org.citrusframework.context.TestContext
            import org.citrusframework.script.GroovyAction.ScriptExecutor

            public class GScript implements ScriptExecutor {
                public void execute(TestContext context) {
                    println context.getVariable("time")
                }
            }
        """)
    );
}
```

XML

```
<test name="GroovyTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <variables>
        <variable name="time" value="citrus:currentDate()"/>
    </variables>
    <actions>
        <groovy>
        <![CDATA[
            import org.citrusframework.*
            import org.citrusframework.variable.*
            import org.citrusframework.context.TestContext
            import org.citrusframework.script.GroovyAction.ScriptExecutor

            public class GScript implements ScriptExecutor {
                public void execute(TestContext context) {
                    println context.getVariable("time")
                }
            }
        ]]>
        </groovy>
    </actions>
</test>
```

YAML

```
name: GroovyTest
variables:
  - name: time
    value: citrus:currentDate()
actions:
  - groovy:
      script: |
        import org.citrusframework.*
        import org.citrusframework.variable.*
        import org.citrusframework.context.TestContext
        import org.citrusframework.script.GroovyAction.ScriptExecutor

        public class GScript implements ScriptExecutor {
            public void execute(TestContext context) {
                println context.getVariable("time")
            }
        }
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="groovyTest">
      <variables>
        <variable name="time" value="citrus:currentDate()"/>
      </variables>
      <actions>
        <groovy>
          <![CDATA[
            import org.citrusframework.*
            import org.citrusframework.variable.*
            import org.citrusframework.context.TestContext
            import org.citrusframework.script.GroovyAction.ScriptExecutor

            public class GScript implements ScriptExecutor {
                public void execute(TestContext context) {
                    println context.getVariable("time")
                }
            }
          ]]>
        </groovy>
      </actions>
    </testcase>
</spring:beans>
```

Implementing the ScriptExecutor interface in a custom Groovy class is applicable for very special test context manipulations as you are able to import and use other Java API classes in this code.

8.16. Failing the test

The fail action will generate an exception in order to terminate the test case with error. The test case will therefore not be successful in the reports.

The user can specify a custom error message for the exception in order to describe the error cause. Here is a very simple example to clarify the syntax:

Java

```
@CitrusTest
public void failTest() {
    $(fail().message("Test will fail with custom message"));
}
```

XML

```
<test name="FailTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <fail message="Test will fail with custom message"/>
    </actions>
</test>
```

YAML

```
name: FailTest
actions:
  - fail:
      message: "Test will fail with custom message"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="FailTest">
        <actions>
            <fail message="Test will fail with custom message"/>
        </actions>
    </testcase>
</spring:beans>
```

Test results:

```
Execution of test: failTest failed! Nested exception is:
org.citrusframework.exceptions.CitrusRuntimeException:
Test will fail with custom message

[...]

CITRUS TEST RESULTS

failTest          : failed - Exception is: Test will fail with custom message

Found 1 test cases to execute
Skipped 0 test cases (0.0%)
Executed 1 test cases, containing 3 actions
Tests failed:        1 (100.0%)
Tests successfully:  0 (0.0%)
```

While using the Java DSL you can also raise arbitrary Java exceptions and let the test fail.

Java

```
@CitrusTest
public void throwExceptionSample() {
    // some test actions

    throw new ValidationException("This test should fail now");
}
```

The validation exception above will cause the test to fail as expected. However you may consider using the fail action to raise errors because this ensures to properly report the error in the Citrus test reports.

8.17. Input

During the test case execution it is possible to read some user input from the command line. The test execution will stop and wait for keyboard inputs over the standard input stream. The user has to type the input and end it with the return key.

The user input is stored to the respective variable value.

Java

```
@CitrusTest
public void inputActionTest() {
    variable("userinput", "");
    variable("userinput1", "");
    variable("userinput2", "y");
    variable("userinput3", "yes");
    variable("userinput4", "");

    $(input());

    $(echo().message("user input was: ${userinput}"));

    $(input().message("Now press enter:").result("userinput1"));

    $(echo().message("user input was: ${userinput1}"));

    $(input().message("Do you want to continue?").answers("y",
"n").result("userinput2"));

    $(echo().message("user input was: ${userinput2}"));

    $(input().message("Do you want to continue?").answers("yes",
"no").result("userinput3"));

    $(echo().message("user input was: ${userinput3}"));

    $(input().result("userinput4"));

    $(echo().message("user input was: ${userinput4}"));
}
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="InputTest">
        <variables>
            <variable name="userinput" value=""></variable>
            <variable name="userinput1" value=""></variable>
            <variable name="userinput2" value="y"></variable>
            <variable name="userinput3" value="yes"></variable>
            <variable name="userinput4" value=""></variable>
        </variables>
        <actions>
            <input/>
            <echo><message>user input was: ${userinput}</message></echo>

            <input message="Now press enter:" variable="userinput1"/>
            <echo><message>user input was: ${userinput1}</message></echo>

            <input message="Do you want to continue?"
                      valid-answers="y/n" variable="userinput2"/>
            <echo><message>user input was: ${userinput2}</message></echo>

            <input message="Do you want to continue?"
                      valid-answers="yes/no" variable="userinput3"/>
            <echo><message>user input was: ${userinput3}</message></echo>

            <input variable="userinput4"/>
            <echo><message>user input was: ${userinput4}</message></echo>
        </actions>
    </testcase>
</spring:beans>
```

As you can see the input action is customizable with a prompt message that is displayed to the user and some valid answer possibilities. The user input is stored to a test variable for further use in the test case. In detail the input action offers following attributes:

message	message displayed to the user
valid-answers	possible valid answers separated with '/' character
variable	result variable name holding the user input (default = ${userinput})

When the user input is restricted to a set of valid answers the input validation of course can fail due to mismatch. This is the case when the user provides some input not matching the valid answers given. In this case the user is again asked to provide valid input. The test action will continue to ask for valid input until a valid answer is given.

User inputs may not fit to automatic testing in terms of continuous integration testing where no user is present to type in the correct answer over the keyboard. In this case you can always skip the user input in advance by specifying a variable that matches the user input variable name. As the user input variable is then already present the user input is missed out and the test proceeds automatically.

8.18. Load Properties

You are able to load properties from external property files and store them as test variables. The action will require a file resource either from class path or file system in order to read the property values.

Let us look at an example to get an idea about this action:

Content of load.properties

```
username=Mickey Mouse
greeting.text=Hello Test Framework
```

Java

```
@CitrusTest
public void loadPropertiesTest() {
    $(load().filePath("file:tests/resources/load.properties"));
}
```

XML

```
<test name="LoadPropertiesTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <load>
            <properties file="file:tests/resources/load.properties"/>
        </load>
    </actions>
</test>
```

YAML

```
name: LoadPropertiesTest
actions:
  - load:
      properties:
        file: "file:tests/resources/load.properties"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="LoadPropertiesTest">
        <actions>
            <load>
                <properties file="file:tests/resources/load.properties"/>
            </load>
        </actions>
    </testcase>
</spring:beans>
```

Output

```
Current value of variable username = Mickey Mouse
Current value of variable greeting.text = Hello Test Framework
```

The action will load all available properties in the file load.properties and store them to the test case as local variables.

Please be aware of the fact that existing variables are overwritten!

8.19. Purging JMS destinations

Purging JMS destinations during the test run is quite essential. Different test cases can influence each other when sending messages to the same JMS destinations. A test case should only receive those messages that actually belong to it.

Therefore, it is a good idea to purge all JMS queue destinations between the test cases. Obsolete messages that are stuck in a JMS queue for some reason are then removed so that the following test case is not offended.

Java

```
@Autowired
@Qualifier("connectionFactory")
private ConnectionFactory connectionFactory;

@CitrusTest
public void purgeJmsQueuesTest() {
    $(purgeQueues()
        .connectionFactory(connectionFactory)
        .queue("JMS.Queue.1")
        .queue("JMS.Queue.2")
        .queue("JMS.Queue.3")
    );
}
```

XML

```
<test name="PurgeJmsQueuesTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <purge-jms-queues connection-factory="connectionFactory">
            <queue name="JMS.Queue.1"/>
            <queue name="JMS.Queue.2"/>
            <queue name="JMS.Queue.3"/>
        </purge-jms-queues>
    </actions>
</test>
```

YAML

```
name: PurgeJmsQueuesTest
actions:
  - purge-queues:
      connection-factory: "connectionFactory"
      queues:
        - "JMS.Queue.1"
        - "JMS.Queue.2"
        - "JMS.Queue.3"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="purgeJmsQueuesTest">
      <actions>
          <jms:purge-jms-queues connection-factory="connectionFactory">
              <jms:queue name="JMS.Queue.1"/>
              <jms:queue name="JMS.Queue.2"/>
              <jms:queue name="JMS.Queue.3"/>
          </jms:purge-jms-queues>
      </actions>
    </testcase>
</spring:beans>
```

Citrus provides special support for JMS related features when using the Spring XML bean integration. You have to activate those JMS features in our test case by adding a special "jms" namespace and schema definition location to the test case XML.

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
        xmlns:spring="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:jms="http://www.citrusframework.org/schema/jms/testcase"
        xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.citrusframework.org/schema/testcase
        http://www.citrusframework.org/schema/testcase/citrus-testcase.xsd
        http://www.citrusframework.org/schema/jms/testcase
        http://www.citrusframework.org/schema/jms/testcase/citrus-jms-testcase.xsd">

    [...]

</beans>
```

Notice that we have referenced the **jms** namespace when using the **purge-jms-queues** test action.

Purging the JMS queues in every test case is quite exhausting because every test case needs to define a purging action at the very beginning of the test. Fortunately the test suite definition offers tasks to run before, between and after the test cases which should ease up these tasks a lot. The test suite offers a very simple way to purge the destinations between the tests. See testsuite-before-test for more information about this.

As you can see in the next example it is quite easy to specify a group of destinations in the Spring configuration that get purged before a test is executed.

Spring Bean

```
@Bean
public SequenceBeforeTest beforeTest() {
    return SequenceBeforeTest.Builder.beforeTest()
            .actions(
                purgeQueues()
                    .queue("fooChannel")
                    .queue("barChannel")
            );
}
```

Spring XML

```
<spring:beans xmlns:spring="http://www.springframework.org/schema/beans"
              xmlns:citrus="http://www.citrusframework.org/schema/config">
    <citrus:before-test id="purgeBeforeTest">
        <citrus:actions>
            <jms:purge-jms-queues>
                <jms:queue name="fooChannel"/>
                <jms:queue name="barChannel"/>
            </jms:purge-jms-queues>
        </citrus:actions>
    </citrus:before-test>
</spring:beans>
```

Please keep in mind that the JMS related configuration components in Citrus belong to a separate XML namespace **jms:** . We have to add this namespace declaration to each test case XML and Spring bean XML configuration file as described at the very beginning of this section.

The syntax for purging the destinations is the same as we used it inside the test case. So now we are able to purge JMS destinations with given destination names. But sometimes we do not want to rely on queue or topic names as we retrieve destinations over JNDI for instance. We can deal with destinations coming from JNDI lookup like follows:

Spring XML

```
<jee:jndi-lookup id="jmsQueueHelloRequestIn" jndi-name="jms/jmsQueueHelloRequestIn"/>
<jee:jndi-lookup id="jmsQueueHelloResponseOut" jndi-
name="jms/jmsQueueHelloResponseOut"/>

<citrus:before-test id="purgeBeforeTest">
    <citrus:actions>
        <jms:purge-jms-queues>
            <jms:queue ref="jmsQueueHelloRequestIn"/>
            <jms:queue ref="jmsQueueHelloResponseOut"/>
        </jms:purge-jms-queues>
    </citrus:actions>
</citrus:before-test>
```

We just use the attribute **'ref'** instead of **'name'** and Citrus is looking for a bean reference for that identifier that resolves to a JMS destination. You can use the JNDI bean references inside a test case, too.

Java

```
@Autowired
@Qualifier("jmsQueue1")
private Queue jmsQueue1;

@Autowired
@Qualifier("jmsQueue2")
private Queue jmsQueue2;

@CitrusTest
public void purgeJmsQueuesTest() {
    $(purgeQueues()
        .queue(jmsQueue1)
        .queue(jmsQueue1)
    );
}
```

XML

```
<test name="PurgeJmsQueuesTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <purge-jms-queues>
            <queue ref="jmsQueue1"/>
            <queue ref="jmsQueue2"/>
        </purge-jms-queues>
    </actions>
</test>
```

YAML

```
name: PurgeJmsQueuesTest
actions:
  - purge-queues:
      queue: jmsQueue1
  - purge-queues:
      queue: jmsQueue2
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="purgeJmsQueuesTest">
      <actions>
          <jms:purge-jms-queues>
              <jms:queue ref="jmsQueue1"/>
              <jms:queue ref="jmsQueue2"/>
          </jms:purge-jms-queues>
      </actions>
    </testcase>
</spring:beans>
```

Of course you can use queue object references also in Java DSL test cases. Here we easily can use Spring's dependency injection with autowiring to get the object references from the IoC container.

You can mix queue name and queue object references as you like within one single purge queue test action.

8.20. Purging Spring Message Channels

The Spring Integration project defines message channels as a central messaging destination. These are in-memory message queues holding messages for test cases. Messages that are queued on a channel may become obsolete during a test run, especially when test cases fail and stop in their message consumption. Purging these message channel destinations is essential in these scenarios in order to avoid bad influence on upcoming test cases. Each test case should only receive those messages that actually refer to the test model. Therefore, it is a good idea to purge all message channel destinations between the test cases. Obsolete messages that get stuck in a message channel destination for some reason are then removed so that upcoming test case are not broken.

Following action definition purges all messages from a list of message channels:

Java

```
@Autowired
@Qualifier("someChannelName")
private MessageChannel someChannel;

@Autowired
@Qualifier("anotherChannelName")
private MessageChannel anotherChannel;

@CitrusTest
public void purgeChannelTest() {
    $(purgeChannels()
        .channelResolver(channelResolver)
        .channelNames("someChannelName", "anotherChannelName")
    );

    $(purgeChannels()
        .channels(someChannel, anotherChannel)
    );
}
```

XML

```
<test name="PurgeChannelTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <purge-channels channel-resolver="channelResolver">
            <channel name="someChannelName"/>
            <channel name="anotherChannelName"/>
        </purge-channels>

        <purge-channels>
            <channel ref="someChannel"/>
            <channel ref="anotherChannel"/>
        </purge-channels>
    </actions>
</test>
```

YAML

```
name: PurgeChannelTest
actions:
  - purge-channels:
      channel-resolver: "channelResolver"
      channels:
        - "someChannelName"
        - "anotherChannelName"
  - purge-channels:
      channel: someChannel
  - purge-channels:
      channel: anotherChannel
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="PurgeChannelTest">
      <actions>
          <purge-channel>
              <channel name="someChannelName"/>
              <channel name="anotherChannelName"/>
          </purge-channel>

          <purge-channel>
              <channel ref="someChannel"/>
              <channel ref="anotherChannel"/>
          </purge-channel>
      </actions>
    </testcase>
</spring:beans>
```

As you can see the test action supports channel names as well as channel references to Spring bean instances. When using channel references you refer to the bean id or name in your application context.

The channel resolver reference is optional and points to a channel resolver that is used to resolve names to actual channel instances. By default, Citrus will automatically use a Spring application context channel resolver, so you just have to use the respective Spring bean names that are configured in the Spring application context. However, setting a custom channel resolver may be adequate for you in some special cases.

Message selectors enable you to selectively remove messages from the destination. All messages that pass the message selection logic get deleted the other messages will remain unchanged inside the channel destination. The message selector is a Spring bean that implements a special message selector interface. A possible implementation could be a selector deleting all messages that are older than five seconds:

Message selector implementation

```
import org.springframework.messaging.Message;
import org.springframework.integration.core.MessageSelector;

public class TimeBasedMessageSelector implements MessageSelector {

    public boolean accept(Message<?> message) {
        if (System.currentTimeMillis() - message.getHeaders().getTimestamp() > 5000) {
            return false;
        } else {
            return true;
        }
    }
}
```

The message selector returns **false** for those messages that should be deleted from the channel!

You simply define the message selector as a new Spring bean in the Citrus application context and reference it in your test action property.

Spring Bean

```
@Bean
public MessageSelector specialMessageSelector() {
    return new TimeBasedMessageSelector();
}
```

Spring XML

```
<bean id="specialMessageSelector"
    class="org.citrusframework.special.TimeBasedMessageSelector"/>
```

Now let us have a look at how you reference the selector in your test case:

Java

```
@CitrusTest
public void purgeChannelTest() {
    $(purgeChannels()
        .selector(specialMessageSelector)
        .channels(someChannel, anotherChannel)
    );
}
```

XML

```
<test name="PurgeChannelTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <purge-channels message-selector="specialMessageSelector">
            <channel ref="someChannel"/>
            <channel ref="anotherChannel"/>
        </purge-channels>
    </actions>
</test>
```

YAML

```
name: PurgeChannelTest
actions:
  - purge-channels:
      message-selector: "specialMessageSelector"
      channels:
        - "someChannel"
        - "anotherChannel"
```

Spring XML

```
<purge-channel message-selector="specialMessageSelector">
  <channel ref="someChannel"/>
  <channel red="anotherChannel"/>
</purge-channel>
```

In the examples above we use a message selector implementation that gets injected via Spring IoC container.

Purging channels in each test case every time is quite exhausting because every test case needs to define a purging action at the very beginning of the test. A more straight forward approach would be to introduce some purging action which is automatically executed before each test. Fortunately the Citrus test suite offers a very simple way to do this. It is described in testsuite-before-test.

When using the special action sequence before test cases we are able to purge channel destinations every time a test case executes. See the upcoming example to find out how the action is defined in the Spring configuration application context.

Spring Bean

```
@Bean
public SequenceBeforeTest beforeTest() {
    return SequenceBeforeTest.Builder.beforeTest()
            .actions(
                purgeChannels()
                    .channel("fooChannel")
                    .channel("barChannel")
            );
}
```

Spring XML

```
<spring:beans xmlns:spring="http://www.springframework.org/schema/beans"
              xmlns:citrus="http://www.citrusframework.org/schema/config">
    <citrus:before-test id="purgeBeforeTest">
        <citrus:actions>
            <purge-channel>
                <channel name="fooChannel"/>
                <channel name="barChannel"/>
            </purge-channel>
        </citrus:actions>
    </citrus:before-test>
</spring:beans>
```

Just use this before-test bean in the Spring bean application context and the purge channel action is active. Obsolete messages that are waiting on the message channels for consumption are purged before the next test in line is executed.

> Purging message channels becomes also very interesting when working with server instances in Citrus. Each server component automatically has an inbound message channel where incoming messages are stored internally. So if you need to clean up a server that has already stored some incoming messages you can do this easily by purging the internal message channel. The message channel follows a naming convention **{serverName}.inbound** where **{serverName}** is the Spring bean name of the Citrus server endpoint component. If you purge this internal channel in a before test nature you are sure that obsolete messages on a server instance get purged before each test is executed.

8.21. Purging endpoints

Citrus works with message endpoints when sending and receiving messages. In general endpoints can also queue messages. This is especially the case when using JMS message endpoints or any server endpoint component in Citrus. These are in-memory message queues holding messages for test cases. The enqueued messages may become obsolete during a test run, especially when a test case that would consume the messages fails.

Deleting all messages from a message endpoint is therefore a useful task and is essential in such scenarios so that upcoming test cases are not influenced. Each test case should only receive those messages that actually refer to the test model. Therefore it is a good idea to purge all message endpoint destinations between the test cases. Obsolete messages that get stuck in a message endpoint destination for some reason are then removed so that upcoming test case are not broken.

Following action definition purges all messages from a list of message endpoints:

Java

```
@Autowired
@Qualifier("someEndpointName")
private DirectEndpoint someEndpoint;

@Autowired
@Qualifier("anotherEndpointName")
private DirectEndpoint anotherEndpoint;

@CitrusTest
public void purgeEndpointTest() {
    $(purgeEndpoints()
        .endpointNames("someEndpointName", "anotherEndpointName")
    );

    $(purgeEndpoints()
        .endpoints(someEndpoint, anotherEndpoint)
    );
}
```

XML

```
<test name="PurgeEndpointTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <purge-endpoint>
            <endpoint name="someChannelName"/>
            <endpoint name="anotherChannelName"/>
        </purge-endpoint>

        <purge-endpoint>
            <endpoint ref="someChannel"/>
            <endpoint ref="anotherChannel"/>
        </purge-endpoint>
    </actions>
</test>
```

YAML

```
name: PurgeEndpointTest
actions:
  - purge:
      endpoints:
        - name: "someEndpointName"
        - name: "anotherEndpointName"
  - purge:
      endpoints:
        - ref: someEndpoint
        - ref: anotherEndpoint
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="purgeEndpointTest">
      <actions>
          <purge-endpoint>
              <endpoint name="someEndpointName"/>
              <endpoint name="anotherEndpointName"/>
          </purge-endpoint>

          <purge-endpoint>
              <endpoint ref="someEndpoint"/>
              <endpoint ref="anotherEndpoint"/>
          </purge-endpoint>
      </actions>
    </testcase>
</spring:beans>
```

As you can see the test action supports endpoint names as well as endpoint references to Spring bean instances. When using endpoint references you refer to the Spring bean name in your application context.

When using the Java DSL we can inject endpoint objects with Spring bean container IoC.

Message selectors enable you to selectively remove messages from an endpoint. All messages that meet the message selector condition get deleted and the other messages remain inside the endpoint destination. The message selector is either a normal String name-value representation or a map of key value pairs:

Java

```
@CitrusTest
public void purgeEndpointTest() {
    $(purgeEndpoints()
        .selector("operation = 'sayHello'")
        .endpoints(someEndpoint, anotherEndpoint)
    );
}
```

XML

```
<test name="PurgeEndpointTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <purge-endpoint>
            <selector>
                <value>operation = 'sayHello'</value>
            </selector>
            <endpoint ref="someEndpoint"/>
            <endpoint ref="anotherEndpoint"/>
        </purge-endpoint>
    </actions>
</test>
```

YAML

```
name: PurgeEndpointTest
actions:
  - purge-endpoints:
      selector:
        value: "operation = 'sayHello'"
      endpoints:
        - "someEndpoint"
        - "anotherEndpoint"
```

Spring XML

```
<purge-endpoints>
  <selector>
    <value>operation = 'sayHello'</value>
  </selector>
  <endpoint name="someEndpointName"/>
  <endpoint name="anotherEndpointName"/>
</purge-endpoints>
```

In the examples above we use a String to represent the message selector expression. In general the message selector operates on the message header. So following on from that we remove all messages selectively that have a message header **operation** with its value **sayHello** .

Purging endpoints in each test case every time is quite exhausting because every test case needs to define a purging action at the very beginning of the test. A more straight forward approach would be to introduce some purging action which is automatically executed before each test. Fortunately the Citrus test suite offers a very simple way to do this. It is described in testsuite-before-test.

When using the special action sequence before test cases we are able to purge endpoint destinations every time a test case executes. See the upcoming example to find out how the action is defined in the Spring configuration application context.

Spring Bean

```
@Bean
public SequenceBeforeTest beforeTest() {
    return SequenceBeforeTest.Builder.beforeTest()
            .actions(
                purgeEndpoints()
                    .endpoint("fooChannel")
                    .endpoint("barChannel")
            );
}
```

Spring XML

```
<spring:beans xmlns:spring="http://www.springframework.org/schema/beans"
              xmlns:citrus="http://www.citrusframework.org/schema/config">
    <citrus:before-test id="purgeBeforeTest">
        <citrus:actions>
            <purge-endpoints>
                <channel name="fooChannel"/>
                <channel name="barChannel"/>
            </purge-endpoints>
        </citrus:actions>
    </citrus:before-test>
</spring:beans>
```

Just use this before-test bean in the Spring bean application context and the purge endpoint action is active. Obsolete messages that are waiting on the message endpoints for consumption are purged before the next test in line is executed.

> Purging message endpoints becomes also very interesting when working with server instances in Citrus. Each server component automatically has an inbound message endpoint where incoming messages are stored too internally. Citrus will automatically use this incoming message endpoint as target for the purge action so you can just use the server instance as you know it from your configuration in any purge action.

8.22. Assert failure

Citrus test actions fail with Java exceptions and error messages. This gives you the opportunity to expect an action to fail during test execution. You can simply assert a Java exception to be thrown during execution. See the example for an assert action definition in a test case:

Java

```
@CitrusTest
public void assertFailureTest() {
    $(assertException()
            .exception(org.citrusframework.exceptions.CitrusRuntimeException.class)
            .message("Unknown variable ${date}")
            .when(
                echo().message("Current date is: ${date}")
            )
    );
}
```

XML

```
<test name="AssertFailureTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <assert exception="org.citrusframework.exceptions.CitrusRuntimeException"
                message="Unknown variable ${date}">
            <when>
                <echo>
                    <message>Current date is: ${date}</message>
                </echo>
            </when>
        </assert>
    </actions>
</test>
```

YAML

```
name: "AssertFailureTest"
actions:
  - assert:
      exception: "org.citrusframework.exceptions.CitrusRuntimeException"
      message: 'Unknown variable ${date}'
      when:
        - echo:
            message: 'Current date is: ${date}'
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="AssertFailureTest">
        <actions>
            <assert exception="org.citrusframework.exceptions.CitrusRuntimeException"
                    message="Unknown variable ${date}">
                <when>
                    <echo>
                        <message>Current date is: ${date}</message>
                    </echo>
                </when>
            </assert>
        </actions>
    </testcase>
</spring:beans>
```

Note that the assert action requires an exception. In case no exception is thrown by the embedded test action the assertion and the test case will fail!

The assert action always wraps a single test action, which is then monitored for failure. In case the nested test action fails with error you can validate the error in its type and error message (optional). The failure has to fit the expected one exactly otherwise the assertion fails itself.

Important to notice is the fact that asserted exceptions do not cause failure of the test case. As you expect the failure to happen the test continues with its work once the assertion is done successfully.

8.23. Catch exceptions

In the previous chapter we have seen how to expect failures in Citrus with assert action. Now the assert action is designed for single actions to be monitored and for failures to be expected in any case. The **'catch'** action in contrary can hold several nested test actions and exception failure is optional.

The nested actions are error proof for the chosen exception type. This means possible exceptions are caught and ignored - the test case will not fail for this exception type. But only for this particular exception type! Other exception types that occur during execution do cause the test to fail as usual.

Java

```
@CitrusTest
public void catchFailureTest() {
    $(catchException()
        .exception(CitrusRuntimeException.class)
        .when(
            echo().message("Current date is: ${date}")
        );
    );
}
```

XML

```
<test name="CatchFailureTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <catch exception="org.citrusframework.exceptions.CitrusRuntimeException">
            <when>
                <echo>
                    <message>Current date is: ${date}</message>
                </echo>
            </when>
        </assert>
    </actions>
</test>
```

YAML

```
name: "CatchFailureTest"
actions:
  - catch:
      exception: "org.citrusframework.exceptions.CitrusRuntimeException"
      when:
        - echo:
            message: 'Current date is: ${date}'
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="catchExceptionTest">
        <actions>
            <catch exception="org.citrusframework.exceptions.CitrusRuntimeException">
                <echo>
                    <message>Current date is: ${date}</message>
                </echo>
            </catch>
        </actions>
    </testcase>
</spring:beans>
```

Note that there is no validation available in a catch block. So catching exceptions is just to make a test more stable towards errors that can occur. The caught exception does not cause any failure in the test. The test case may continue with execution as if there was no failure. Also notice that the catch action is also happy when no exception at all is raised. In contrary to that the assert action requires the exception and an assert action is failing in positive processing.

Catching exceptions like this may only fit to very error-prone action blocks where failures do not harm the test case success. Otherwise, a failure in a test action should always reflect to the whole test case to fail with errors.

Java developers might ask why not use try-catch Java block instead? The answer is simple yet very important to understand. The test method is called by the Java DSL test case builder for building the Citrus test. This can be referred to as the design time of the test. After the building test method was processed the test gets executed, which can be called the runtime of the test. This means that a try-catch block within the design time method will never perform during the test run. The only reliable way to add the catch capability to the test as part of the test case runtime is to use the Citrus test action which gets executed during test runtime.

8.24. Apache Ant build

The <ant> action loads a build.xml Ant file and executes one or more targets in the Ant project. The target is executed with optional build properties passed to the Ant run. The Ant build output is logged with Citrus logger and the test case success is bound to the Ant build success. This means in case the Ant build fails for some reason the test case will also fail with build exception accordingly.

See this basic Ant run example to see how it works within your test case:

Java

```
@CitrusTest
public void antRunTest() {
    variable("today", "citrus:currentDate()");

    $(antrun()
        .buildFilePath("classpath:org/citrusframework/actions/build.xml")
        .target("sayHello")
        .property("date", "${today}")
        .property("welcomeText", "$Hello!"))
    );
}
```

XML

```
<test name="AntRunTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <variables>
        <variable name="today" value="citrus:currentDate()"/>
    </variables>
    <actions>
        <ant build-file="classpath:org/citrusframework/actions/build.xml">
            <execute target="sayHello"/>
            <properties>
                <property name="date" value="${today}"/>
                <property name="welcomeText" value="Hello!"/>
            </properties>
        </ant>
    </actions>
</test>
```

YAML

```
name: AntRunTest
variables:
  - name: "today"
    value: "citrus:currentDate()"
actions:
  - ant:
      build-file: "classpath:org/citrusframework/actions/build.xml"
      execute:
        target: "sayHello"
      properties:
        - name: "date"
          value: "${today}"
        - name: "welcomeText"
          value: "Hello!"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="AntRunTest">
        <variables>
            <variable name="today" value="citrus:currentDate()"/>
        </variables>
        <actions>
            <ant build-file="classpath:org/citrusframework/actions/build.xml">
                <execute target="sayHello"/>
                <properties>
                    <property name="date" value="${today}"/>
                    <property name="welcomeText" value="Hello!"/>
                </properties>
            </ant>
        </actions>
    </testcase>
</spring:beans>
```

The respective build.xml Ant file must provide the target to call. For example:

build.xml

```
<project name="citrus-build" default="sayHello">
    <property name="welcomeText" value="Welcome to Citrus!"></property>

    <target name="sayHello">
        <echo message="${welcomeText} - Today is ${date}"></echo>
    </target>

    <target name="sayGoodbye">
        <echo message="Goodbye everybody!"></echo>
    </target>
</project>
```

As you can see you can pass custom build properties to the Ant build execution. Existing Ant build properties are replaced and you can use the properties in your build file as usual.

You can also call multiple targets within one single build run by using a comma separated list of target names:

Java

```
@CitrusTest
public void antRunTest() {
    variable("today", "citrus:currentDate()");

    $(antrun()
        .buildFilePath("classpath:org/citrusframework/actions/build.xml")
        .targets("sayHello", "sayGoodbye")
        .property("date", "${today}"))
    );
}
```

XML

```
<test name="AntRunTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <variables>
        <variable name="today" value="citrus:currentDate()"/>
    </variables>
    <actions>
        <ant build-file="classpath:org/citrusframework/actions/build.xml">
            <execute targets="sayHello,sayGoodbye"/>
            <properties>
                <property name="date" value="${today}"/>
            </properties>
        </ant>
    </actions>
</test>
```

YAML

```
name: AntRunTest
variables:
  - name: "today"
    value: "citrus:currentDate()"
actions:
  - ant:
      build-file: "classpath:org/citrusframework/actions/build.xml"
      execute:
        targets: "sayHello,sayGoodbye"
      properties:
        - name: "date"
          value: "${today}"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="AntRunTest">
        <variables>
            <variable name="today" value="citrus:currentDate()"/>
        </variables>
        <actions>
            <ant build-file="classpath:org/citrusframework/actions/build.xml">
                <execute targets="sayHello,sayGoodbye"/>
                <properties>
                    <property name="date" value="${today}"/>
                </properties>
            </ant>
        </actions>
    </testcase>
</spring:beans>
```

The build properties can live in external file resource as an alternative to the inline property definitions. You just have to use the respective file resource path and all nested properties get loaded as build properties.

In addition to that you can also define a custom build listener. The build listener must implement the Ant API interface **org.apache.tools.ant.BuildListener** . During the Ant build run the build listener is called with several callback methods (e.g. buildStarted(), buildFinished(), targetStarted(), targetFinished(), ...). This is how you can add additional logic to the Ant build run from Citrus. A custom build listener could manage the fail state of your test case, in particular by raising some exception forcing the test case to fail accordingly.

Java

```
@Autowired
private BuildListener customBuildListener;

@CitrusTest
public void antRunTest() {
    $(antrun()
        .buildFilePath("classpath:org/citrusframework/actions/build.xml")
        .target("sayHello")
        .propertyFile("classpath:org/citrusframework/actions/build.properties")
        .listener(customBuildListener))
    );
}
```

XML

```
<test name="AntRunTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <ant build-file="classpath:org/citrusframework/actions/build.xml"
             build-listener="customBuildListener">
            <execute target="sayHello"/>
            <properties  file="classpath:org/citrusframework/actions/build.properties"
/>
        </ant>
    </actions>
</test>
```

YAML

```
name: AntRunTest
actions:
  - ant:
      build-file: "classpath:org/citrusframework/actions/build.xml"
      build-listener: "customBuildListener"
      execute:
        target: "sayHello"
      properties-file: "classpath:org/citrusframework/actions/build.properties"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="AntRunTest">
        <actions>
            <ant build-file="classpath:org/citrusframework/actions/build.xml"
                 build-listener="customBuildListener">
                <execute target="sayHello"/>
                <properties
file="classpath:org/citrusframework/actions/build.properties"/>
            </ant>
        </actions>
    </testcase>
</spring:beans>
```

The **customBuildListener** used in the example above should reference a Spring bean in the Citrus application context. The bean implements the interface **org.apache.tools.ant.BuildListener** and controls the Ant build run.

8.25. Start/Stop server

Citrus is working with server components that are started and stopped within a test run. This can be a Http server or some SMTP mail server for instance. Usually the Citrus server components are

automatically started when Citrus is starting and respectively stopped when Citrus is shutting down. Sometimes it might be helpful to explicitly start and stop a server instance within your test case. Here you can use special start and stop test actions inside your test. This is a good way to test downtime scenarios of interface partners with respective error handling when connections to servers are lost

Let me explain with a simple sample test case:

Java

```
@Autowired
@Qualifier("myMailServer")
private MailServer myMailServer;

@CitrusTest
public void manageServerTest() {
    $(start()
        .server(myMailServer));

    $(sequential()
        .actions(...)); // do something

    $(stop()
        .server(myMailServer));
}
```

XML

```
<test name="ManageServerTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <start server="myMailServer"/>

        <sequential>
            <!-- do something -->
        </sequential>

        <stop server="myMailServer"/>
    </actions>
</test>
```

YAML

```
name: ManageServerTest
actions:
  - start:
      server: "myMailServer"
  - sequential:
      actions: {}
  - stop:
      server: "myMailServer"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="manageServerTest">
        <actions>
            <start server="myMailServer"/>

            <sequential>
                <!-- do something -->
            </sequential>

            <stop server="myMailServer"/>
        </actions>
    </testcase>
</spring:beans>
```

The start and stop server test action receive a server name which references a Spring bean component of type **org.citrusframework.server.Server** in your basic Spring application context. The server instance is started or stopped within the test case. As you can see in the next listing we can also start and stop multiple server instances within a single test action.

Java

```
@Autowired
@Qualifier("myMailServer")
private MailServer myMailServer;

@Autowired
@Qualifier("myFtpServer")
private FtpServer myFtpServer;

@CitrusTest
public void manageServerTest() {
    $(start()
        .server(myMailServer, myFtpServer));

    $(sequential()
        .actions(...)); // do something

    $(stop()
        .server(myMailServer, myFtpServer));
}
```

XML

```
<test name="ManageServerTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <start>
            <servers>
                <server name="myMailServer"/>
                <server name="myFtpServer"/>
            </servers>
        </start>

        <sequential>
            <!-- do something -->
        </sequential>

        <stop>
            <servers>
                <server name="myMailServer"/>
                <server name="myFtpServer"/>
            </servers>
        </stop>
    </actions>
</test>
```

YAML

```
name: ManageServerTest
actions:
  - start:
      servers:
        - "myMailServer"
        - "myFtpServer"
  - sequential:
      actions: {}
  - stop:
      servers:
        - "myMailServer"
        - "myFtpServer"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="manageServerTest">
        <actions>
            <start>
                <servers>
                    <server name="myMailServer"/>
                    <server name="myFtpServer"/>
                </servers>
            </start>

            <sequential>
                <!-- do something -->
            </sequential>

            <stop>
                <servers>
                    <server name="myMailServer"/>
                    <server name="myFtpServer"/>
                </servers>
            </stop>
        </actions>
    </testcase>
</spring:beans>
```

When using the Java DSL the best way to reference a server instance is to autowire the Spring bean via dependency injection. The Spring framework takes care of injecting the proper Spring bean component defined in the Spring application context. This way you can easily start and stop server instances within Java DSL test cases.

Starting and stopping server instances is a synchronous test action. This means that your test case is waiting for the server to start before other test actions take place. Startup times and shut down of server instances may delay your test accordingly.

As you can see starting and stopping Citrus server instances is very easy. You can also write your own server implementations by implementing the interface **org.citrusframework.server.Server**. All custom server implementations can then be started and stopped during a test case.

8.26. Stop Timer

The <stop-timer> action can be used for stopping either a specific timer (containers-timer) or all timers running within a test. This action is useful when timers are started in the background (using parallel or fork=true) and you wish to stop these timers at the end of the test. Some examples of using this action are provided below:

Java

```
@CitrusTest
public void timerTest() {
    $(timer()
      .id("forkedTimer")
      .fork(true)
      .actions(sleep(50L))
    );

    $(timer()
      .fork(true)
      .actions(sleep(50L))
    );

    $(timer()
      .repeatCount(5)
      .actions(sleep(50L))
    );

    $(stopTimer("forkedTimer"));

    $(doFinally().actions(
      stopTimer()
    ));
}
```

XML

```xml
<test name="TimerTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <timer id="forkedTimer" fork="true">
            <actions>
                <sleep milliseconds="50" />
            </actions>
        </timer>

        <timer fork="true">
            <actions>
                <sleep milliseconds="50" />
            </actions>
        </timer>

        <timer repeatCount="5">
            <actions>
                <sleep milliseconds="50" />
            </actions>
        </timer>

        <stop-timer id="forkedTimer" />
    </actions>

    <finally>
        <stop-timer/>
    </finally>
</test>
```

YAML

```yaml
name: TimerTest
actions:
  - timer:
      id: "forkedTimer"
      fork: true
      actions:
        - sleep:
            milliseconds: "50"
  - timer:
      fork: true
      actions:
        - sleep:
            milliseconds: "50"
  - timer:
      repeatCount: 5
      actions:
        - sleep:
            milliseconds: "50"
  - stop-timer:
      id: "forkedTimer"
finally:
  - stop-timer: {}
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="TimerTest">
        <actions>
            <timer id="forkedTimer" fork="true">
                <sleep milliseconds="50" />
            </timer>

            <timer fork="true">
                <sleep milliseconds="50" />
            </timer>

            <timer repeatCount="5">
                <sleep milliseconds="50" />
            </timer>

            <stop-timer timerId="forkedTimer" />
        </actions>

        <finally>
            <stop-timer/>
        </finally>
    </testcase>
</spring:beans>
```

In the above example 3 timers are started, the first 2 in the background and the third in the test execution thread. Timer #3 has a repeatCount set to 5 so it will terminate automatically after 5 runs. Timer #1 and #2 however have no repeatCount set so they will execute until they are told to stop.

Timer #1 is stopped explicitly using the first stopTimer action. Here the stopTimer action includes the name of the timer to stop. This is convenient when you wish to terminate a specific timer. However since no timerId was set for timer #2, you can terminate this (and all other timers) using the 'stopTimer' action with no explicit timerId set.

8.27. Custom action

Now we have a look at the opportunity to add custom test actions to the test case flow. Let us start this section with an example:

Java

```
@CitrusTest
public void actionReferenceTest() {
    $(action().reference("clearDatabase"));
    $(action().reference("mySpecialAction"));
}
```

XML

```
<test name="ActionReferenceTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <action reference="clearDatabase"/>
        <action reference="mySpecialAction"/>
    </actions>
</test>
```

YAML

```
name: ActionReferenceTest
actions:
  - action:
      ref: "clearDatabase"
  - action:
      ref: "mySpecialAction"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="ActionReferenceTest">
        <actions>
            <action reference="clearDatabase"/>
            <action reference="mySpecialAction"/>
        </actions>
    </testcase>
</spring:beans>
```

The generic <action> element references Spring beans that implement the Java interface ***org.citrusframework.TestAction*** . This is a very fast way to add your own action implementations to a Citrus test case. This way you can easily implement your own actions in Java and include them into the test case.

In the example above the called actions are special database cleanup implementations. The actions are defined as Spring beans in the Citrus configuration and get referenced by their bean name or id.

Spring Bean

```
@Bean
public SpecialDatabaseCleanupAction clearDatabase(DataSource ds) {
    return new SpecialDatabaseCleanupAction(ds);
}
```

Spring XML

```
<bean id="clearDatabase"
class="my.domain.citrus.actions.SpecialDatabaseCleanupAction">
    <constructor-arg ref="myDataSource"/>
</bean>
```

The Spring application context holds your custom bean implementations. You can set properties and use the full Spring power while implementing your custom test action in Java. Let us have a look on how such a Java class may look like.

SpecialDatabaseCleanupAction

```
import org.citrusframework.actions.AbstractTestAction;
import org.citrusframework.context.TestContext;

public class SpecialDatabaseCleanupAction extends AbstractTestAction {

    @Autowired
    private DataSource dataSource;

    @Override
    public void doExecute(TestContext context) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        jdbcTemplate.execute("...");
    }
}
```

All you need to do in your Java class is to implement the Citrus ***org.citrusframework.TestAction*** interface. The abstract class ***org.citrusframework.actions.AbstractTestAction*** may help you to start with your custom test action implementation as it provides basic method implementations so you just have to implement the ***doExecute()*** method.

When using the Java test case DSL you are also quite comfortable with including your custom test actions.

Java

```
@Autowired
private SpecialDatabaseCleanupAction clearDatabaseAction;

@CitrusTest
public void actionReferenceTest() {
    $(echo().message("Now let's include our special test action"));

    $(clearDatabaseAction);

    $(echo().message("That's it!"));
}
```

Using anonymous class uses as Lambda expression is also possible.

Java

```
@CitrusTest
public void genericActionTest() {
    $(echo().message("Now let's call our special test action anonymously"));

    $(context -> {
        // do something
    });

    $(echo().message("That's it!"));
}
```

The test action receives a single argument `context` which represent the `org.citrusframework.context.TestContext`. The context provides access to test variables and shared components resolvable via reference resolver API.

Chapter 9. Containers

Similar to templates a container element holds one to many test actions. In contrast to the template the container appears directly inside the test case action chain, meaning that the container is not referenced by more than one test case.

Containers execute the embedded test actions in specific logic. This can be an execution in iteration for instance. Combine different containers with each other and you will be able to generate very powerful hierarchical structures in order to create a complex execution logic. In the following sections some predefined containers are described.

9.1. Sequential

The sequential container executes the embedded test actions in strict sequence. Readers now might search for the difference to the normal action chain that is specified inside the test case. The actual power of sequential containers does show only in combination with other containers like iterations and parallels. We will see this later when handling these containers.

For now the sequential container seems not very sensational - one might say boring - because it simply groups a pair of test actions to sequential execution.

Java

```
@CitrusTest
public void sequentialTest() {
    $(sequential()
        .actions(
            sleep().seconds(1),
            echo("Hello Citrus!")
        )
    );
}
```

XML

```
<test name="SequentialTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <sequential>
            <actions>
                <sleep seconds="1"/>
                <echo>
                    <message>Hello Citrus!</message>
                </echo>
            </actions>
        </sequential>
    </actions>
</test>
```

YAML

```
name: SequentialTest
actions:
  - sequential:
      actions:
        - sleep:
            seconds: 1
        - echo:
            message: "Hello Citrus!"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="SequentialTest">
        <actions>
            <sequential>
                <sleep seconds="1"/>
                <echo>
                    <message>Hello Citrus!</message>
                </echo>
            </sequential>
        </actions>
    </testcase>
</spring:beans>
```

9.2. Conditional

Now we deal with conditional executions of test actions. Nested actions inside a conditional container are executed only in case a boolean expression evaluates to true. Otherwise the container execution is not performed at all.

See some example to find out how it works with the conditional expression string.

Java

```
@CitrusTest
public void conditionalTest() {
    variable("index", 5);
    variable("shouldSleep", true);

    $(conditional().when("${index} = 5")
        .actions(
            sleep(10000L)
        )
    );

    $(conditional().when("${shouldSleep}")
        .actions(
            sleep(10000L)
        )
    );

    $(conditional().when("${shouldSleep}", anyOf(is("true"), isEmptyString()))
        .actions(
            sleep(10000L)
        )
    );
}
```

XML

```
<test name="ConditionalTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <variables>
      <variable name="index" value="5"/>
      <variable name="shouldSleep" value="true"/>
    </variables>

    <actions>
        <conditional when="${index} = 5">
            <actions>
                <sleep seconds="10"/>
            </actions>
        </conditional>

        <conditional when="${shouldSleep}">
            <actions>
                <sleep seconds="10"/>
            </actions>
        </conditional>

        <conditional when="@assertThat('${shouldSleep}', 'anyOf(is(true),
isEmptyString())')@">
            <actions>
                <sleep seconds="10"/>
            </actions>
        </conditional>
    </actions>
</test>
```

YAML

```yaml
name: ConditionalTest
variables:
  - name: "index"
    value: 5
  - name: "shouldSleep"
    value: true

actions:
  - conditional:
      when: '${index} = 5'
      actions:
        - sleep:
            seconds: 10
  - conditional:
      when: '${shouldSleep}'
      actions:
        - sleep:
            seconds: 10
  - conditional:
      when: '@assertThat("${shouldSleep}", "anyOf(is(true), isEmptyString())")@'
      actions:
        - sleep:
            seconds: 10
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="ConditionalTest">
        <variables>
          <variable name="index" value="5"/>
          <variable name="shouldSleep" value="true"/>
        </variables>

        <actions>
            <conditional expression="${index} = 5">
                <sleep seconds="10"/>
            </conditional>

            <conditional expression="${shouldSleep}">
                <sleep seconds="10"/>
            </conditional>

            <conditional expression="@assertThat('${shouldSleep}', 'anyOf(is(true),
isEmptyString())')@">
                <sleep seconds="10"/>
            </conditional>
        </actions>
    </testcase>
</spring:beans>
```

The nested sleep action is executed in case the variable ${index} is equal to the value '5'. This conditional execution of test actions is useful when dealing with different test environments such as different operating systems for instance. The conditional container also supports expressions that evaluate to the character sequence "true" or "false" as shown in the ${shouldSleep} example.

The last conditional container in the example above makes use of Hamcrest matchers. The matcher evaluates to **true** or **false** and based on that the container actions are executed or skipped. The Hamcrest matchers are very powerful when it comes to evaluation of multiple conditions at a time.

9.3. Parallel

Parallel containers execute embedded test actions concurrently to each other. Every action in this container will be executed in a separate Java thread. The following example should clarify the usage:

Java

```
@CitrusTest
public void parallelTest() {
    $(parallel()
        .actions(
            sleep(),
            sequential().actions(
                sleep(),
                echo("1")
            ),
            echo("2"),
            echo("3"),
            iterate().condition("i lt= 5").index("i"))
                .actions(
                    echo("10")
                )
        )
    );
}
```

XML

```
<test name="ParallelTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <parallel>
            <actions>
                <sleep/>

                <sequential>
                    <actions>
                        <sleep/>
                        <echo>
                            <message>1</message>
                        </echo>
                    </actions>
                </sequential>

                <echo>
                    <message>2</message>
                </echo>

                <echo>
                    <message>3</message>
                </echo>

                <iterate condition="i lt= 5"
                            index="i">
                    <actions>
                        <echo>
                            <message>10</message>
                        </echo>
                    </actions>
                </iterate>
            </actions>
        </parallel>
    </actions>
</test>
```

YAML

```yaml
name: ParallelTest
actions:
  - parallel:
      actions:
        - sleep: {}
        - sequential:
            actions:
              - sleep: {}
              - echo:
                  message: "1"
        - echo:
            message: "2"
        - echo:
            message: "3"
        - iterate:
            index: "i"
            condition: "i lt 5"
            actions:
              - echo:
                  message: "10"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="ParallelTest">
        <actions>
            <parallel>
                <sleep/>

                <sequential>
                    <sleep/>
                    <echo>
                        <message>1</message>
                    </echo>
                </sequential>

                <echo>
                    <message>2</message>
                </echo>

                <echo>
                    <message>3</message>
                </echo>

                <iterate condition="i lt= 5"
                            index="i">
                    <echo>
                        <message>10</message>
                    </echo>
                </iterate>
            </parallel>
        </actions>
    </testcase>
</spring:beans>
```

By default, test actions are processed and executed one action after another. Since the first action is a sleep of five seconds, the whole test would stop and wait for 5 seconds. Things are different inside the parallel container. Here, the descending test actions will not wait, but execute at the same time.

If you are using this container to send or receive messages, you have to use the unique correlation ID of the message to link the actions concerning this message. Otherwise, the testcase might associate a send or receive action with the wrong message. Please note that this ID is **not** passed to your system under test. The management of correlation IDs as well as the assignment to messages is done internally. Only the mapping between the request and response has to be done by the author of the test. As you can see in the following example, the value of the header `MessageHeaders.ID` is stored in the variable `request#1` respectively `request#2`. This variable is reused in the `receive` action to identify the correct response from the server.

Java

```
@CitrusTest
public void paralletTest() {

    $(parallel()
        .actions(
            sequential()
                .actions(
                    http().client(httpClient)
                            .send()
                            .post("/foo")
                            .extract(fromHeaders().header(MessageHeaders.ID,
"request#1")
                            .payload("{ \"info\": \"foo\"}"),

                    //SUT echoing the input

                    http().client(httpClient)
                            .receive()
                            .response(HttpStatus.OK)
                            .payload("{ \"info\": \"foo\"}")
                            .selector(
                                Collections.singletonMap(
                                    MessageHeaders.ID, "${request#1}"))
                ),

            sequential()
                .actions(
                    http().client(httpClient)
                         .send()
                         .post("/boo")
                         .extract(fromHeaders().header(MessageHeaders.ID, "request#2")
                         .payload("{ \"info\": \"boo\"}"),

                    //SUT echoing the input

                    http().client(httpClient)
                         .receive()
                         .response(HttpStatus.OK)
                         .payload("{ \"info\": \"boo\"}")
                         .selector(
                            Collections.singletonMap(
                                MessageHeaders.ID, "${request#2}"))
                )
        )
    );
}
```

Containers can easily wrap other containers. The example shows a simple combination of sequential and parallel containers that will achieve more complex execution logic. Actions inside the sequential container will execute one after another. But actions in parallel will be executed at the same time.

9.4. Iterate

Iterations are very powerful elements when describing complex logic. The container executes the embedded actions several times. The container will continue with looping as long as the defined breaking condition string evaluates to ***true*** . In case the condition evaluates to ***false*** the iteration will break and finish execution.

Java

```
@CitrusTest
public void iterateTest() {
    $(iterate().condition("i lt 5").index("i")
        .actions(
            echo().message("index is: ${i}")
        )
    );
}
```

XML

```
<test name="IterateTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <iterate index="i" condition="i lt 5">
            <actions>
                <echo>
                    <message>index is: ${i}</message>
                </echo>
            </actions>
        </iterate>
    </actions>
</test>
```

YAML

```
name: IterateTest
actions:
  - iterate:
      index: "i"
      condition: "i lt 5"
      actions:
        - echo:
            message: "index is: ${i}"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="IterateTest">
        <actions>
            <iterate index="i" condition="i lt 5">
                <echo>
                    <message>index is: ${i}</message>
                </echo>
            </iterate>
        </actions>
    </testcase>
</spring:beans>
```

The attribute "index" automatically defines a new variable that holds the actual loop index starting at "1". This index variable is available as a normal variable inside the iterate container. Therefore it is possible to print out the actual loop index in the echo action as shown in the above example.

The condition string is mandatory and describes the actual end of the loop. In iterate containers the loop will break in case the condition evaluates to ***false***.

The condition string can be any Boolean expression and supports several operators:

lt	lower than
lt=	lower than equals
gt	greater than
gt=	greater than equals
=	equals
and	logical combining of two Boolean values
or	logical combining of two Boolean values
()	brackets

> It is very important to notice that the condition is evaluated before the very first iteration takes place. The loop therefore can be executed 0-n times according to the condition value.

Now the boolean expression evaluation as described above is limited to very basic operation such as **lower than**, **greater than** and so on. We also can use Hamcrest matchers in conditions that are way more powerful than that.

Java

```
@CitrusTest
public void iterateTest() {
    $(iterate().condition(assertThat(lessThan(5)).index("i"))
        .actions(
            echo().message("index is: ${i}")
        )
    );
}
```

XML

```
<test name="IterateTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <iterate index="i" condition="@assertThat(lessThan(5))@">
            <actions>
                <echo>
                    <message>index is: ${i}</message>
                </echo>
            </actions>
        </iterate>
    </actions>
</test>
```

YAML

```
name: IterateTest
actions:
  - iterate:
      index: "i"
      condition: "@assertThat(lessThan(5))@"
      actions:
        - echo:
            message: "index is: ${i}"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="IterateTest">
        <actions>
            <iterate index="i" condition="@assertThat(lessThan(5))@">
                <echo>
                    <message>index is: ${i}</message>
                </echo>
            </iterate>
        </actions>
    </testcase>
</spring:beans>
```

In the example above we use Hamcrest matchers as a condition expression. In Java you may use the `org.citrusframework.container.HamcrestConditionExpression.assertThat(org.hamcrest.Matcher<?>)` method. This way you can combine Hamcrest matchers and create very powerful condition evaluations here.

The Hamcrest expression evaluation in Citrus gets added with this module:

citrus-validation-hamcrest Module

```
<dependency>
    <groupId>org.citrusframework</groupId>
    <artifactId>citrus-validation-hamcrest</artifactId>
    <version>${citrus.version}</version>
</dependency>
```

You need to add this module to your project accordingly when using the Hamcrest matcher support.

9.5. Repeat

Quite similar to the previously described iterate container this repeating container will execute its actions in a loop according to an ending condition. The condition describes a Boolean expression using the operators as described in the previous chapter.

The loop continues its work until the provided condition evaluates to **true** . It is very important to notice that the repeat loop will execute the actions before evaluating the condition. This means the actions get executed `n-1` times.

Java

```
@CitrusTest
public void repeatTest() {
    $(repeat().until("(i gt 5) or (i = 3)").index("i")
        .actions(
            echo("index is: ${i}")
        )
    );
}
```

XML

```
<test name="RepeatTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <repeat index="i" until="(i = 3) or (i = 5)">
            <actions>
                <echo>
                    <message>index is: ${i}</message>
                </echo>
            </actions>
        </repeat>
    </actions>
</test>
```

YAML

```
name: RepeatTest
actions:
  - repeat:
      index: "i"
      until: "(i = 3) or (i = 5)"
      actions:
        - echo:
            message: "index is: ${i}"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="RepeatTest">
        <actions>
            <repeat-until-true index="i" condition="(i = 3) or (i = 5)">
                <echo>
                    <message>index is: ${i}</message>
                </echo>
            </repeat-until-true>
        </actions>
    </testcase>
</spring:beans>
```

As you can see the repeat container is only executed when the iterating condition expression evaluates to **false** . By the time the condition is **true** execution is discontinued. You can use basic logical operators such as **and**, **or** and so on.

A more powerful way is given by Hamcrest matchers that are directly supported in condition expressions.

Java

```
@CitrusTest
public void repeatTest() {
    $(repeat().until(assertThat(anyOf(is(3), is(5))).index("i")
        .actions(
            echo("index is: ${i}")
        )
    );
}
```

XML

```
<test name="RepeatTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <repeat index="i" until="@assertThat(anyOf(is(3), is(5))@">
            <actions>
                <echo>
                    <message>index is: ${i}</message>
                </echo>
            </actions>
        </repeat>
    </actions>
</test>
```

YAML

```
name: RepeatTest
actions:
  - repeat:
      index: "i"
      until: "@assertThat(anyOf(is(3), is(5))@"
      actions:
        - echo:
            message: "index is: ${i}"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="RepeatTest">
        <actions>
            <repeat-until-true index="i" condition="@assertThat(anyOf(is(3), is(5))@">
                <echo>
                    <message>index is: ${i}</message>
                </echo>
            </repeat-until-true>
        </actions>
    </testcase>
</spring:beans>
```

The Hamcrest matcher usage simplifies the reading a lot. In addition to that it empowers you to combine more complex condition expressions. In Java you may use the `org.citrusframework.container.HamcrestConditionExpression.assertThat(org.hamcrest.Matcher<?>)` method.

The Hamcrest expression evaluation in Citrus gets added with this module:

citrus-validation-hamcrest Module

```
<dependency>
    <groupId>org.citrusframework</groupId>
    <artifactId>citrus-validation-hamcrest</artifactId>
    <version>${citrus.version}</version>
</dependency>
```

You need to add this module to your project accordingly when using the Hamcrest matcher support.

9.6. Repeat on error

The next looping container is called repeat-on-error-until-true. This container repeats a group of actions in case one embedded action failed with error. In case of an error inside the container the loop will try to execute ***all*** embedded actions again in order to seek for overall success. The

execution continues until all embedded actions were processed successfully ***or*** the ending condition evaluates to true and the error-loop will lead to final failure.

Java

```
@CitrusTest
public void repeatOnErrorTest() {
    $(repeatOnError().until("i = 5").index("i")
        .actions(
            echo().message("index is: ${i}"),
            fail().message("Force loop to fail!")
        )
    );
}
```

XML

```
<test name="RepeatOnErrorTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <repeat-on-error index="i" until="i = 5">
            <actions>
                <echo>
                    <message>index is: ${i}</message>
                </echo>
                <fail message="Force loop to fail!"/>
            </actions>
        </repeat-on-error>
    </actions>
</test>
```

YAML

```
name: RepeatOnErrorTest
actions:
  - repeat-on-error:
      index: "i"
      until: "i = 5"
      actions:
        - echo:
            message: "index is: ${i}"
        - fail:
            message: "Force loop to fail!"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="RepeatOnErrorTest">
        <actions>
            <repeat-onerror-until-true index="i" condition="i = 5">
                <echo>
                    <message>index is: ${i}</message>
                </echo>
                <fail message="Force loop to fail!"/>
            </repeat-onerror-until-true>
        </actions>
    </testcase>
</spring:beans>
```

In the code example the error-loop continues four times as the <fail> action definitely fails the test. During the fifth iteration the condition "i=5" evaluates to true and the loop breaks its processing leading to a final failure as the test actions were not successful.

The overall success of the test case depends on the error situation inside the repeat-onerror-until-true container. In case the loop breaks because of failing actions and the loop will discontinue its work the whole test case is failing too. The error loop processing is successful in case all embedded actions were not raising any errors during an iteration.

The repeat-on-error container also offers an automatic sleep mechanism. This auto-sleep property will force the container to wait a given amount of time before executing the next iteration. We used this mechanism a lot when validating database entries. Let's say we want to check the existence of an order entry in the database. Unfortunately the system under test is not very well performing and may need some time to store the new order. This amount of time is not predictable, especially when dealing with different hardware on our test environments (local testing vs. server testing). Following from that our test case may fail unpredictably only because of runtime conditions.

We can avoid unstable test cases that are based on these runtime conditions with the auto-sleep functionality.

Java

```
@CitrusTest
public void repeatOnErrorTest() {
    $(repeatOnError().until("i = 5").index("i").autoSleep(1000)
        .actions(
            sql()
                .dataSource(testDataSource)
                .query()
                .statement("""
                    SELECT COUNT(1) AS CNT_ORDERS
                    FROM ORDERS
                    WHERE CUSTOMER_ID='${customerId}'
                """)
                .validate("CNT_ORDERS", 1)
        )
    );
}
```

XML

```
<test name="RepeatOnErrorTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <repeat-on-error auto-sleep="1000" until="i = 5" index="i">
            <actions>
                <sql datasource="testDataSource">
                    <statements>
                        <statement>
                          SELECT COUNT(1) AS CNT_ORDERS
                          FROM ORDERS
                          WHERE CUSTOMER_ID='${customerId}'
                        </statement>
                    </statements>
                    <validate column="CNT_ORDERS" value="1"/>
                </sql>
            </actions>
        </repeat-on-error>
    </actions>
</test>
```

YAML

```
name: RepeatOnErrorTest
actions:
  - repeat-on-error:
      index: "i"
      until: "i = 5"
      auto-sleep: 1000
      actions:
        - sql:
            dataSource: "testDataSource"
            statements:
              - statement: |
                  SELECT COUNT(1) AS CNT_ORDERS
                  FROM ORDERS
                  WHERE CUSTOMER_ID='${customerId}'
            validate:
              - column: "CNT_ORDERS"
                value: 1
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="RepeatOnErrorTest">
        <actions>
            <repeat-onerror-until-true auto-sleep="1000" condition="i = 5" index="i">
                <sql datasource="testDataSource">
                    <statement>
                      SELECT COUNT(1) AS CNT_ORDERS
                      FROM ORDERS
                      WHERE CUSTOMER_ID='${customerId}'
                    </statement>
                    <validate column="CNT_ORDERS" value="1"/>
                </sql>
            </repeat-onerror-until-true>
        </actions>
    </testcase>
</spring:beans>
```

We surrounded the database check with a repeat-onerror container having the auto-sleep property set to 1000 milliseconds. The repeat container will try to check the database up to five times with an automatic sleep of 1 second before every iteration. This gives the system under test up to five seconds time to store the new entry to the database. The test case is very stable and just fits to the hardware environment. On slow test environments the test may need several iterations to successfully read the database entry. On very fast environments the test may succeed right on the first try.

> We changed auto sleep time from seconds to milliseconds with Citrus 2.0 release. So if you are coming from previous Citrus versions be sure to now use proper millisecond values.

So fast environments are not slowed down by static sleep operations and slower environments are still able to execute this test case with high stability.

9.7. Timer

Timers are very useful containers when you wish to execute a collection of test actions several times at regular intervals. The timer component generates an event which in turn triggers the execution of the nested test actions associated with the timer. This can be useful in a number of test scenarios for example when Citrus needs to simulate a heart beat or if you are debugging a test and you wish to query the contents of the database, to mention just a few. The following code sample should demonstrate the power and flexibility of timers:

Java

```
@CitrusTest
public void timerTest() {
    $(timer()
        .id("forkedTimer")
        .interval(100L)
        .fork(true)
        .actions(
            echo()
                .message("I'm going to run in the background and let some other test
actions run (nested action run ${forkedTimer-index} times)"),
            sleep().milliseconds(50L)
        )
    );

    $(timer()
        .repeatCount(3)
        .interval(100L)
        .delay(50L)
        .actions(
            sleep().milliseconds(50L),
            echo().message("I'm going to repeat this message 3 times before the next
test actions are executed")
        )
    );

    $(echo()
        .message("Test almost complete. Make sure all timers running in the background
are stopped")
    );

    $(doFinally().actions(
        stopTimer("forkedTimer")
    ));
}
```

XML

```
<test name="TimerTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <timer id="forkedTimer" interval="100" fork="true">
            <actions>
                <echo>
                    <message>I'm going to run in the background and let some other
test actions run (nested action run ${forkedTimer-index} times)</message>
                </echo>
                <sleep milliseconds="50" />
            </actions>
        </timer>

        <timer repeatCount="3" interval="100" delay="50">
            <actions>
                <sleep milliseconds="50" />
                <echo>
                    <message>I'm going to repeat this message 3 times before the next
test actions are executed</message>
                </echo>
            </actions>
        </timer>

        <echo>
          <message>Test almost complete. Make sure all timers running in the
background are stopped</message>
        </echo>
    </actions>

    <finally>
        <stop-timer id="forkedTimer" />
    </finally>
</test>
```

YAML

```
name: TimerTest
actions:
  - timer:
      id: forkedTimer
      fork: true
      delay: 5000
      interval: 100
      repeatCount: 1
      actions:
        - echo:
            message: |
              I'm going to run in the background and let some other test actions run
(nested action run ${forkedTimer-index} times)
        - sleep:
            milliseconds: 50
  - timer:
      delay: 50
      interval: 100
      repeatCount: 3
      actions:
        - sleep:
            milliseconds: 50
        - echo:
            message: |
              I'm going to repeat this message 3 times before the next test actions
are executed
  - echo:
      message: |
        Test almost complete. Make sure all timers running in the background are
stopped
finally:
  - stop-timer:
      id: "forkedTimer"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="TimerTest">
        <actions>
            <timer id="forkedTimer" interval="100" fork="true">
              <echo>
                <message>I'm going to run in the background and let some other test
actions run (nested action run ${forkedTimer-index} times)</message>
              </echo>
              <sleep milliseconds="50" />
            </timer>

            <timer repeatCount="3" interval="100" delay="50">
              <sleep milliseconds="50" />
              <echo>
                <message>I'm going to repeat this message 3 times before the next test
actions are executed</message>
              </echo>
            </timer>

            <echo>
              <message>Test almost complete. Make sure all timers running in the
background are stopped</message>
            </echo>
        </actions>

        <finally>
            <stop-timer timerId="forkedTimer" />
        </finally>
    </testcase>
</spring:beans>
```

In the above example the first timer (timerId = forkedTimer) is started in the background. By default timers are run in the current thread of execution but to start it in the background just use "fork=true". Every 100 milliseconds this timer emits an event which will result in the nested actions being executed. The nested 'echo' action outputs the number of times this timer has already been executed. It does this with the help of an 'index' variable, in this example ${forkedTimer-index}, which is named according to the timer **id** with the suffix '-index'. No limit is set on the number of times this timer should run so it will keep on running until either a nested test action fails or it is instructed to stop (more on this below).

The second timer is configured to run 3 times with a delay of 100 milliseconds between each iteration. Using the attribute 'delay' we can get the timer pause for 50 milliseconds before running the nested actions for the first time. The timer is configured to run in the current thread of execution so the last test action, the 'echo', has to wait for this timer to complete before it is executed.

So how do we tell the forked timer to stop running? If we forget to do this the timer will just execute

indefinitely. To help us out here we can use the 'stop-timer' action. By adding this to the finally block we ensure that the timer will be stopped, even if some nested test action fails. We could have easily added it as a nested test action, to the forkedTimer for example, but if some other test action failed before the stop-timer was called, the timer would never stop.

> You can also configure timers to run in the background using the 'parallel' container, rather than setting the attribute 'fork' to true. Using parallel allows more fine-grained control of the test and has the added advantage that all errors generated from a nested timer action are visible to the test executer. If an error occurs within the timer then the test status is set to failed. Using fork=true an error causes the timer to stop executing, but the test status is not influenced by this error.

9.8. Async

Now we deal with parallel execution of test actions. Nested actions inside an async container are executed in a separate thread. This allows to continue test execution without having to wait for actions inside the async container to complete. The test immediately continues to execute the next test actions, which will be executed in parallel to those actions inside the async container.

This mechanism comes in handy when a test action should be forked to the rest of the test. In send operations we were already able to achieve this by setting `fork="true"`. With async containers, we're able to execute all kinds of test actions asynchronous.

See some example to find out how it works.

Java

```
@CitrusTest
public void asyncTest() {
    $(async()
        .actions(
            send(fooEndpoint)
                .message(fooRequest()),
            receive(fooEndpoint)
                .message(fooResponse())
        )
    );

    $(echo().message("Continue with test"));
}
```

XML

```
<test name="AsyncTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <async>
            <actions>
                <send endpoint="fooEndpoint">
                    <message><!-- ... --></message>
                </send>
                <receive endpoint="fooEndpoint">
                    <message><!-- ... --></message>
                </receive>
            </actions>
        </async>

        <echo>
          <message>Continue with test</message>
        </echo>
    </actions>
</test>
```

YAML

```
name: AsyncTest
actions:
  - async:
      actions:
        - send:
            endpoint: "fooEndpoint"
            message: {}
        - receive:
            endpoint: "fooEndpoint"
            message: {}
  - echo:
      message: "Continue with test"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="AsyncTest">
        <actions>
            <async>
                <actions>
                    <send endpoint="fooEndpoint">
                        <message><!-- ... --></message>
                    </send>
                    <receive endpoint="fooEndpoint">
                        <message><!-- ... --></message>
                    </receive>
                </actions>
            </async>

            <echo>
              <message>Continue with test</message>
            </echo>
        </actions>
    </testcase>
</spring:beans>
```

The nested `send` and `receive` actions get executed in parallel to the other test actions in that test case. So the test will not wait for these actions to finish before executing next actions. Of course possible errors inside the async container will also cause the whole test case to fail. And the test will definitely wait for all async actions to be finished before finishing the whole test case. This safely lets us execute test actions in parallel to each other.

The async container also supports success and error callback actions.

Java

```
@CitrusTest
public void asyncTest() {
    $(async()
        .actions(
            send(fooEndpoint)
                .message(fooRequest()),
            receive(fooEndpoint)
                .message(fooResponse())
        )
        .successActions(
            echo().message("Success!")
        )
        .errorActions(
            echo().message("Error!")
        )
    );

    $(echo().message("Continue with test"));
}
```

XML

```
<test name="AsyncTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <async>
            <actions>
                <send endpoint="fooEndpoint">
                    <message><!-- ... --></message>
                </send>
                <receive endpoint="fooEndpoint">
                    <message><!-- ... --></message>
                </receive>
            </actions>
            <success>
                <echo><message>Success!</message></echo>
            </success>
            <error>
                <echo><message>Failed!</message></echo>
            </error>
        </async>

        <echo>
          <message>Continue with test</message>
        </echo>
    </actions>
</test>
```

YAML

```yaml
name: AsyncTest
actions:
  - async:
      actions:
        - send:
            endpoint: "fooEndpoint"
            message: {}
        - receive:
            endpoint: "fooEndpoint"
            message: {}
      success:
        - echo:
            message: "Success!"
      error:
        - echo:
            message: "Error!"
  - echo:
      message: "Continue with test"
```

Spring XML

```xml
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="AsyncTest">
        <actions>
            <async>
                <actions>
                    <send endpoint="fooEndpoint">
                        <message><!-- ... --></message>
                    </send>
                    <receive endpoint="fooEndpoint">
                        <message><!-- ... --></message>
                    </receive>
                </actions>
                <success>
                    <echo><message>Success!</message></echo>
                </success>
                <error>
                    <echo><message>Failed!</message></echo>
                </error>
            </async>

            <echo>
              <message>Continue with test</message>
            </echo>
        </actions>
    </testcase>
</spring:beans>
```

So you can add test actions which are executed based on the async test actions outcome `success` or `error`.

If you are using this container to send or receive messages, you have to use the unique correlation ID of the message to link the actions concerning this message. Otherwise the testcase might associate a `send` or `receive` action with the wrong message. Please note that this ID is **not** provided to your system under test. The management of correlation IDs as well as the assignment to messages is done internally. Only the mapping between the request and response has to be done by the author of the test. As you can see in the following example, the value of the header `MessageHeaders.ID` is stored in the variable `request#1` respectively `request#2`. This variable is reused in the `receive` action to identify the correct response from the server.

Java

```
@CitrusTest
public void testAsync() {

    $(async()
        .actions(
            http().client(httpClient)
                    .send()
                    .post("/foo")
                    .extract(fromHeaders().header(MessageHeaders.ID, "request#1")
                    .payload("{ \"info\": \"foo\"}"),

            //SUT echoing the input

            http().client(httpClient)
                    .receive()
                    .response(HttpStatus.OK)
                    .payload("{ \"info\": \"foo\"}")
                    .selector(
                        Collections.singletonMap(
                            MessageHeaders.ID, "${request#1}"))
        )
    );

    $(async()
        .actions(
            http().client(httpClient)
                    .send()
                    .post("/boo")
                    .extract(fromHeaders().header(MessageHeaders.ID, "request#2")
                    .payload("{ \"info\": \"boo\"}"),

            //SUT echoing the input

            http().client(httpClient)
                    .receive()
                    .response(HttpStatus.OK)
                    .payload("{ \"info\": \"boo\"}")
                    .selector(
                        Collections.singletonMap(
                            MessageHeaders.ID, "${request#2}"))
        )
    );
}
```

9.9. Wait

With this action you can make your test wait until a certain condition is satisfied. The attribute

seconds defines the amount of time to wait in seconds. You can also use the milliseconds attribute for a more fine grained time value. The attribute **interval** defines the amount of time to wait **between** each check. The interval is always specified as millisecond time interval.

If the check does not exceed within the defined overall waiting time then the test execution fails with an appropriate error message. There are different types of conditions to check.

http This condition is based on an Http request call on a server endpoint. Citrus will wait until the Http response is as defined (e.g. Http 200 OK). This is useful when you want to wait for a server to start.

file This condition checks for the existence of a file on the local file system. Citrus will wait until the file is present.

message This condition checks for the existence of a message in the local message store of the current test case. Citrus will wait until the message with the given name is present.

action This condition executes another test action and checks for successful execution. Citrus will wait until the nested action is executed without any errors.

When should somebody use this action? This action is very useful when you want your test to wait for a certain event to occur before continuing with the test execution. For example if you wish that your test waits until a Docker container is started or for an application to create a log file before continuing, then use this action. You can also create your own condition statements and bind it to the test action.

9.9.1. Http condition

Next let us have a look at a simple example:

Java

```
@CitrusTest
public void waitTest() {
    $(waitFor()
        .time(Duration.ofMillis(10000))
        .interval(2000)
        .http()
            .url("http://sample.org/resource")
            .status(200)
            .timeout(2000);
    );
}
```

XML

```
<test name="WaitTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <wait-for timeout="10" interval="2000">
          <http url="http://sample.org/resource" status="200" timeout="2000" />
        </wait-for>
    </actions>
</test>
```

YAML

```
name: WaitTest
actions:
  - wait-for:
      timeout: "10"
      interval: "2000"
      http:
        url: path/to/resource/file.txt
        status: 200
        timeout: 2000
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="WaitTest">
        <actions>
            <wait seconds="10" interval="2000">
              <http url="http://sample.org/resource" status="200" timeout="2000" />
            </wait>
        </actions>
    </testcase>
</spring:beans>
```

The example waits for some Http server resource to be available with **Http 200 OK** response. Citrus will use **HEAD** request method by default. You can set the request method with the **method** attribute on the Http condition.

9.9.2. File condition

Next let us have a look at the file condition usage:

Java

```
@CitrusTest
public void waitTest() {
    $(waitFor()
        .time(Duration.ofMillis(10000))
        .interval(2000)
        .file().path("path/to/resource/file.txt")
    );
}
```

XML

```
<test name="WaitTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <wait-for timeout="10" interval="2000">
          <file path="path/to/resource/file.txt" />
        </wait-for>
    </actions>
</test>
```

YAML

```
name: WaitTest
actions:
  - wait-for:
      timeout: "10"
      interval: "2000"
      file:
        path: path/to/resource/file.txt
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="WaitTest">
        <actions>
            <wait seconds="10" interval="2000">
              <file path="path/to/resource/file.txt" />
            </wait>
        </actions>
    </testcase>
</spring:beans>
```

Citrus checks for the file to exist under the given path. Only if the file exists the test will continue with further test actions.

9.9.3. Message condition

Next let us have a look at the message condition usage:

Java

```
@CitrusTest
public void waitTest() {
    $(waitFor()
        .time(Duration.ofMillis(10000))
        .interval(2000)
        .message().name("helloRequest")
    );
}
```

XML

```
<test name="WaitTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <wait-for timeout="10" interval="2000">
          <message name="helloRequest" />
        </wait-for>
    </actions>
</test>
```

YAML

```
name: WaitTest
actions:
  - wait-for:
      timeout: "10"
      interval: "2000"
      message:
        name: "helloRequest"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="WaitTest">
        <actions>
            <wait seconds="10" interval="2000">
              <message name="helloRequest" />
            </wait>
        </actions>
    </testcase>
</spring:beans>
```

Citrus checks for the message with the name **helloRequest** in the local message store. Only if the

message with the given name is found the test will continue with further test actions. The local message store is automatically filled with all exchanged messages (send or receive) in a test case. The message names are defined in the respective send or receive operations in the test.

9.9.4. Action condition

Now we would like to wait for some other test action to execute.

Java

```
@CitrusTest
public void waitTest() {
    $(waitFor()
        .time(Duration.ofMillis(10000))
        .interval(2000)
        .execution()
        .action(
            receive(jmsEndpoint)
                .message()
                .body("Wait for me!")
        )
    );
}
```

XML

```
<test name="WaitTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <wait-for timeout="10" interval="2000">
          <action>
            <receive endpoint="jmsEndpoint">
                <message>
                    <body><data>Wait for me!</data></body>
                </message>
            </receive>
          </action>
        </wait-for>
    </actions>
</test>
```

YAML

```
name: WaitTest
actions:
  - wait-for:
      timeout: "10"
      interval: "2000"
      action:
        receive:
          endpoint: "jmsEndpoint"
          message:
            body: |
              Wait for me!
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="WaitTest">
        <actions>
            <wait seconds="10" interval="2000">
              <action>
                <receive endpoint="jmsEndpoint">
                    <message>
                        <payload>Wait for me!</payload>
                    </message>
                </receive>
              </action>
            </wait>
        </actions>
    </testcase>
</spring:beans>
```

You can add any test action to the wait condition so you can execute any other action and wait for its success. This enables us to also use the full send and receive operations on other message transports.

9.10. Custom containers

In case you have a custom action container implementation you might also want to use it in Java DSL. The action containers are handled with special care in the Java DSL because they have nested actions. So when you call a test action container in the Java DSL you always have something like this:

Java

```
@CitrusTest
public void containerTest() {
    $(echo("This echo is outside of the action container"));

    $(sequential()
        .actions(
            echo().message("Inside"),
            echo().message("Inside once more"),
            echo().message("And again: Inside!")
        )
    );

    $(echo().message("This echo is outside of the action container"));
}
```

Now the three nested actions are added to the action **sequential** container rather than to the test case itself although we are using the same action Java DSL methods outside the container. This mechanism is only working because Citrus is handling test action containers with special care.

A custom test action container implementation could look like this:

Custom container implementation

```
public class ReverseActionContainer extends AbstractActionContainer {
    @Override
    public void doExecute(TestContext context) {
        for (int i = getActions().size(); i > 0; i--) {
            getActions().get(i-1).execute(context);
        }
    }
}
```

The container logic is very simple: The container executes the nested actions in reverse order. As already mentioned Citrus needs to take special care on all action containers when executing a Java DSL test. This is why you should not execute a custom test container implementation on your own.

Run custom container

```
@CitrusTest
public void containerTest() {
    ReverseActionContainer reverseContainer = new ReverseActionContainer();
    reverseContainer.addTestActions(
        echo().message("Foo"),
        echo().message("Bar")
    );

    $(reverseContainer);
}
```

The above custom container execution is going to fail with internal error as the Citrus Java DSL was not able to recognise the action container as it should be. Also the **EchoAction** instance creation is not very comfortable. Instead you can use a special container Java DSL syntax also with your custom container implementation:

Generic container syntax

```
@CitrusTest
public void containerTest() {
    $(container(new ReverseActionContainer())
        .actions(
            echo("Foo"),
            echo("Bar")
        )
    );
}
```

The custom container implementation now works fine with the automatically nested echo actions. And we are able to use the usual Java DSL syntactic sugar for test actions like **echo** .

In a next step we add a custom superclass for all our test classes which provides a helper method for the custom container implementation in order to have an even more comfortable syntax.

Container creating helper method

```
public static AbstractTestContainerBuilder<ReverseActionContainer> reverse() {
    return container(new ReverseActionContainer());
}
```

Now all subclasses can use the new **reverse** method for calling the custom container implementation.

Java

```
@CitrusTest
public void containerTest() {
    $(reverse()
        .actions(
            echo().message("Foo"),
            echo().message("Bar")
        )
    );
}
```

Nice! This is how we should integrate customized test action containers to the Citrus Java DSL.

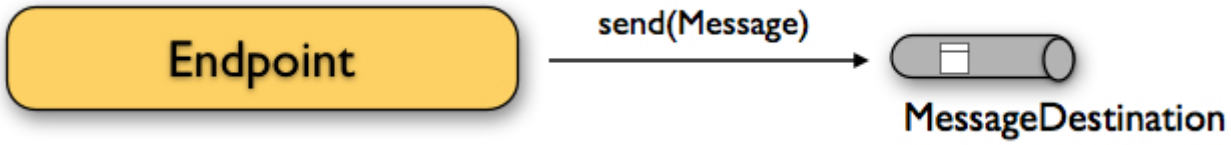
Chapter 10. Endpoints

In one of the previous chapters we have discussed the basic test case structure as we introduced **variables** and **test actions**. The <actions> section contains a list of test actions that take place during the test case. Each test action is executed in sequential order by default. Citrus offers several built-in test actions that the user can choose from to construct a complex testing workflow without having to code everything from scratch. In particular Citrus aims to provide all the test actions that you need as predefined components ready for you to use. The goal is to minimize the coding effort for you so you can concentrate on the test logic itself.

Exactly the same approach is used in Citrus to provide ready-to-use endpoint components for connecting to different message transports. There are several ways in an enterprise application to exchange messages with some other application. We have synchronous interfaces like Http and SOAP WebServices. We have asynchronous messaging with JMS or file transfer FTP interfaces.

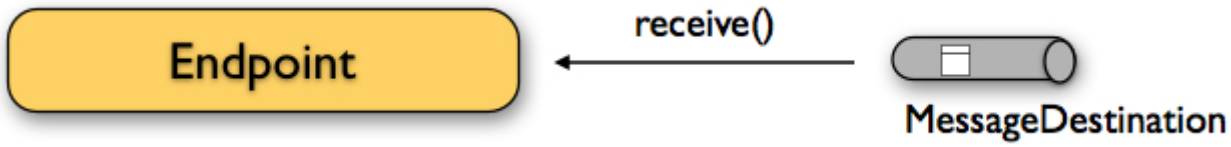
Citrus provides endpoint components as client and server to connect with these typical message transports. So you as a tester must not care about how to send a message to a JMS queue. The Citrus endpoints are configured in the Spring application context and receive endpoint specific properties like endpoint uri or ports or message timeouts as configuration.

The next figure shows a typical message sending endpoint component in Citrus:



The endpoint producer publishes messages to a destination. This destination can be a JMS queue/topic, a SOAP WebService endpoint, an Http URL, an FTP folder destination and so on. The producer just takes a previously defined message definition (header and body) and sends it to the message destination.

Similar to that Citrus defines several endpoint consumer components to consume messages from destinations. This can be a simple subscription on message channels and JMS queues/topics. In case of SOAP WebServices and Http GET/POST things are more complicated as we have to provide a server component that clients can connect to. We will handle server related communication in more detail later on. For now the endpoint consumer component in its most simple way is defined like this:



This is all you need to know about Citrus endpoints. We have mentioned that the endpoints are defined in the Spring application context. Let's have a simple example that shows the basic idea:

```
<citrus-jms:endpoint id="helloServiceEndpoint"
        destination-name="Citrus.HelloService.Request.Queue"
        connection-factory="myConnectionFactory"/>
```

This is a simple JMS endpoint component in Citrus. The endpoint XML bean definition follows a custom XML namespace and defines endpoint specific properties like the JMS destination name and the JMS connection factory. The endpoint id is a significant property as the test cases will reference this endpoint when sending and receiving messages by its identifier.

In the next sections you will learn how a test case uses those endpoint components for producing and consuming messages.

10.1. Send messages

The <send> action in a test case publishes messages to a destination. The actual message transport connection is defined with the endpoint component. The test case simply defines the message contents and references a predefined message endpoint component by its identifier. Endpoint specific configurations are centralized in the Spring bean application context while multiple test cases can reference the endpoint to actually publish the constructed message to a destination. There are several message endpoint implementations in Citrus available representing different transport protocols like JMS, SOAP, HTTP, TCP/IP and many more.

Again the type of transport to use is not specified inside the test case but in the message endpoint definition. The separation of concerns (test case/message sender transport) gives us a good flexibility of our test cases. The test case does not know anything about connection factories, queue names or endpoint uri, connection timeouts and so on. The transport internals underneath a sending test action can change easily without affecting the test case definition. We will see later in this document how to create different message endpoints for various transports in Citrus. For now we concentrate on constructing the message content to be sent.

We assume that the message's body will be plain XML format. Citrus uses XML as the default data format for message body data. But Citrus is not limited to XML message format though; you can always define other message data formats such as JSON, plain text, CSV. As XML is still a very popular message format in enterprise applications and message-based solution architectures we have this as a default format. Anyway Citrus works best on XML bodies and you will see a lot of example code in this document using XML. Finally let us have a look at a first example how a sending action is defined in the test.

XML DSL

```
<testcase name="SendMessageTest">
    <description>Basic send message example</description>

    <actions>
        <send endpoint="helloServiceEndpoint">
            <message>
                <payload>
                    <TestMessage>
                        <Text>Hello!</Text>
                    </TestMessage>
                </payload>
            </message>
            <header>
                <element name="Operation" value="sayHello"/>
            </header>
        </send>
    </actions>
</testcase>
```

Now let's have a closer look at the sending action. The **'endpoint'** attribute might catch your attention first. This attribute references the message endpoint in Citrus configuration by its identifier. As previously mentioned the message endpoint definition lives in a separate configuration file and contains the actual message transport settings. In this example the **"helloServiceEndpoint"** is referenced which is a JMS endpoint for sending out messages to a JMS queue for instance.

The test case is not aware of any transport details, because it does not have to. The advantages are obvious: On the one hand multiple test cases can reference the message endpoint definition for better reuse. Secondly test cases are independent of message transport details. So connection factories, user credentials, endpoint uri values and so on are not present in the test case.

In other words the **"endpoint"** attribute of the `<send>` element specifies which message endpoint definition to use and therefore where the message should go to. Once again all available message endpoints are configured in a separate Citrus configuration file. Be sure to always pick the right message endpoint type in order to publish your message to the right destination.

If you do not like the XML language you can also use pure Java code to define the same test. In Java you would also make use of the message endpoint definition and reference this instance. The same test as shown above in Java DSL looks like this:

Java DSL designer

```
import org.testng.ITestContext;
import org.testng.annotations.Test;
import org.citrusframework.annotations.CitrusTest;
import org.citrusframework.dsl.testng.TestNGCitrusTestDesigner;

@Test
public class SendMessageTestDesigner extends TestNGCitrusTestDesigner {

    @CitrusTest(name = "SendMessageTest")
    public void sendMessageTest() {
        description("Basic send message example");

        send("helloServiceEndpoint")
            .message()
            .body("<TestMessage>" +
                        "<Text>Hello!</Text>" +
                    "</TestMessage>")
            .header("Operation", "sayHello");
    }
}
```

Instead of using the XML tags for send we use methods from **TestNGCitrusTestDesigner** class. The same message endpoint is referenced within the send message action. The body is constructed as plain Java character sequence which is a bit verbose. We will see later on how we can improve this. For now it is important to understand the combination of send test action and a message endpoint.

It is good practice to follow naming conventions when defining names for message endpoints. The intended purpose of the message endpoint as well as the sending/receiving actor should be clear when choosing the name. For instance messageEndpoint1, messageEndpoint2 will not give you much hints to the purpose of the message endpoint.

This is basically how to send messages in Citrus. The test case is responsible for constructing the message content while the predefined message endpoint holds transport specific settings. Test cases reference endpoint components to publish messages to the outside world. This is just the start of action. Citrus supports a whole package of other ways how to define and manipulate the message contents. Read more about message sending actions in actions-send.

10.2. Receive messages

Now we have a look at the message receiving part inside the test. A simple example shows how it works.

XML DSL

```
<receive endpoint="helloServiceEndpoint">
    <message>
        <payload>
            <TestMessage>
                <Text>Hello!</Text>
            </TestMessage>
        </payload>
    </message>
    <header>
        <element name="Operation" value="sayHello"/>
    </header>
</receive>
```

If we recap the send action of the previous chapter we can identify some common mechanisms that apply for both sending and receiving actions. The test action also uses the **endpoint** attribute for referencing a predefined message endpoint. This time we want to receive a message from the endpoint. Again the test is not aware of the transport details such as JMS connections, endpoint uri, and so on. The message endpoint component encapsulates this information.

Before we go into detail on validating the received message we have a quick look at the Java DSL variation for the receive action. The same receive action as above looks like this in Java DSL.

Java DSL designer

```
@CitrusTest
public void messagingTest() {
    receive("helloServiceEndpoint")
        .message()
        .body("<TestMessage>" +
                    "<Text>Hello!</Text>" +
                "</TestMessage>")
        .header("Operation", "sayHello");
}
```

The receive action waits for a message to arrive. The whole test execution is stopped while waiting for the message. This is important to ensure the step by step test workflow processing. Of course you can specify message timeouts so the receiver will only wait a given amount of time before raising a timeout error. Following from that timeout exception the test case fails as the message did not arrive in time. Citrus defines default timeout settings for all message receiving tasks.

At this point you know the two most important test actions in Citrus. Sending and receiving actions will become the main components of your integration tests when dealing with loosely coupled message based components in an enterprise application environment. It is very easy to create complex message flows, meaning a sequence of sending and receiving actions in your test case. You can replicate use cases and test your message exchange with extended message validation capabilities. See actions-receive for a more detailed description on how to validate incoming messages and how to expect message contents in a test case.

10.3. Local message store

All messages that are sent and received during a test case are stored in a local memory storage. This is because we might want to access the message content later on in a test case. We can do so by using message store functions for loading messages that have been exchanged earlier in the test. When storing a message in the local storage Citrus uses a message name as identifier key. This message name is later on used to access the message. You can define the message name in any send or receive action:

XML DSL

```
<receive endpoint="helloServiceEndpoint">
    <message name="helloMessage">
        <payload>
            <TestMessage>
                <Text>Hello!</Text>
            </TestMessage>
        </payload>
    </message>
    <header>
        <element name="Operation" value="sayHello"/>
    </header>
</receive>
```

Java DSL designer

```
@CitrusTest
public void messagingTest() {
    receive("helloServiceEndpoint")
        .message()
        .name("helloMessage")
        .body("<TestMessage>" +
                    "<Text>Hello!</Text>" +
                "</TestMessage>")
        .header("Operation", "sayHello");
}
```

The receive operation above set the message name to **helloMessage**. The message received is automatically stored in the local storage with that name. You can access the message content for instance by using a function:

```
<echo>
    <message>citrus:message(helloMessage.body())</message>
</echo>
```

The function loads the **helloMessage** and prints the body information with the **echo** test action. In combination with Xpath or JsonPath functions this mechanism is a good way to access the exchanged message contents later in a test case.

The storage is for both sent and received messages in a test case. The storage is per test case and contains all sent and received messages.

When no explicit message name is given the local storage will construct a default message name. The default name is built from the action (send or receive) plus the endpoint used to exchange the message. For instance:

```
send(helloEndpoint)
receive(helloEndpoint)
```

The names above would be generated by a send and receive operation on the endpoint named **helloEndpoint**.

The message store is not able to handle multiple message of the same name in one test case. So messages with identical names will overwrite existing messages in the local storage.

Now we have seen the basic endpoint concept in Citrus. The endpoint components represent the connections to the test boundary systems. This is how we can connect to the system under test for message exchange. And this is our main goal with this integration test framework. We want to provide easy access to common message transports on client and server side so that we can test the communication interfaces on a real message transport exchange.

Chapter 11. Direct endpoint

Direct endpoints represent the default in memory messaging solution in Citrus. Producer and consumer components are linked via queues exchanging messages in memory.

> The direct endpoint configuration components use the default "citrus" configuration namespace and schema definition. Include this namespace into your Spring bean configuration in order to use the Citrus configuration elements. The namespace URI and schema location are added to the Spring configuration XML file as follows.

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:citrus="http://www.citrusframework.org/schema/config"
        xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.citrusframework.org/schema/config
        http://www.citrusframework.org/schema/config/citrus-config.xsd">

    [...]

</beans>
```

Right now you are able to use the Citrus XML elements in order to define the direct endpoint components.

11.1. Channel endpoint

Citrus offers a direct endpoint component that is able to create producers and consumers. Producer and consumer send and receive messages both to and from a direct endpoint. By default, the endpoint is asynchronous when configured in the Citrus context.

Java

```
@Bean
public ChannelEndpoint helloEndpoint() {
    return new ChannelEndpointBuilder()
        .queue("helloChannel")
        .build();
}

@Bean
public MessageSelectingQueueChannel helloChannel() {
    return new MessageSelectingQueueChannel();
}
```

XML

```
<citrus:direct-endpoint id="helloEndpoint" queue="helloChannel"/>

<citrus:queue id="helloChannel"/>
```

The Citrus direct endpoint references a queue directly. Inside your test case you can reference the Citrus endpoint as usual to send and receive messages.

The message sender is now ready to publish messages on the defined queue. The communication is supposed to be asynchronous, so the producer is not able to process any reply message. We will deal with synchronous communication and reply messages later in this chapter. You can reference the id of the endpoint in a send and receive test action.

Java

```
when(send("helloEndpoint")
        .message()
        .body("<v1:HelloRequest
xmlns:v1=\"http://citrusframework.org/schemas/HelloService.xsd\">" +
                "<v1:Text>Hello World!</v1:Text>" +
            "</v1:HelloRequest>"));

then(receive("helloEndpoint")
        .message()
        .body("<v1:HelloResponse
xmlns:v1=\"http://citrusframework.org/schemas/HelloService.xsd\">" +
                "<v1:Text>Hello Citrus!</v1:Text>" +
            "</v1:HelloResponse>"));
```

XML

```
<send endpoint="helloEndpoint">
    <message>
        <payload>
            <v1:HelloRequest
xmlns:v1="http://citrusframework.org/schemas/HelloService.xsd">
                <v1:Text>Hello World!</v1:Text>
            </v1:HelloRequest>
        </payload>
    </message>
</send>

<receive endpoint="helloEndpoint">
    <message>
        <payload>
            <v1:HelloResponse
xmlns:v1="http://citrusframework.org/schemas/HelloService.xsd">
                <v1:Text>Hello Citrus!</v1:Text>
            </v1:HelloResponse>
        </payload>
    </message>
</receive>
```

You can send and receive messages from the same direct endpoint. As usual the receiver connects to the message destination and waits for messages to arrive. The user can set a receive timeout which is set to 5000 milliseconds by default. In case no message was received in this time frame the receiver raises timeout errors and the test fails.

11.2. Synchronous direct endpoints

The synchronous direct producer publishes messages and waits synchronously for the response to arrive on a reply queue destination. The reply queue name is set in the message headers. The counterpart in this communication must send its reply to that queue. The basic configuration for a synchronous direct endpoint component looks like follows:

Java

```
@Bean
public ChannelSyncEndpoint helloEndpoint() {
    return new ChannelSyncEndpointBuilder()
        .queue("helloChannel")
        .pollingInterval(1000)
        .build();
}
```

XML

```
<citrus:direct-sync-endpoint id="helloSyncEndpoint"
                             queue="helloChannel"
                             polling-interval="1000"/>
```

Synchronous direct endpoints usually do poll for synchronous reply messages for processing the reply messages. The poll interval is an optional setting in order to manage the amount of reply message handshake attempts. When the endpoint was able to receive the reply message synchronously the test case can verify the reply.

In case all polling attempts have failed the action raises a timeout error, and the test will fail.

> By default, the direct endpoint uses temporary reply queue destinations. The temporary reply queues are only used once for a single communication handshake. The temporary reply queue is deleted automatically.

When sending a message to this endpoint in the first place the producer will wait synchronously for the response message to arrive on the reply queue. You can receive the reply message in your test case using the same endpoint component. So we have two actions on the same endpoint, first send then receive.

Java

```
when(send("helloSyncEndpoint")
        .message()
        .body("<v1:HelloRequest
xmlns:v1=\"http://citrusframework.org/schemas/HelloService.xsd\">" +
                "<v1:Text>Hello World!</v1:Text>" +
            "</v1:HelloRequest>"));

then(receive("helloSyncEndpoint")
        .message()
        .body("<v1:HelloResponse
xmlns:v1=\"http://citrusframework.org/schemas/HelloService.xsd\">" +
                "<v1:Text>Hello Citrus!</v1:Text>" +
            "</v1:HelloResponse>"));
```

XML

```
<send endpoint="helloSyncEndpoint">
    <message>
        <payload>
            <v1:HelloRequest
xmlns:v1="http://citrusframework.org/schemas/HelloService.xsd">
                <v1:Text>Hello World!</v1:Text>
            </v1:HelloRequest>
        </payload>
    </message>
</send>

<receive endpoint="helloSyncEndpoint">
    <message>
        <payload>
            <v1:HelloResponse
xmlns:v1="http://citrusframework.org/schemas/HelloService.xsd">
                <v1:Text>Hello Citrus!</v1:Text>
            </v1:HelloResponse>
        </payload>
    </message>
</receive>
```

This is how you handle synchronous communication as a sender. You publish messages to a queue and wait for reply messages on a temporary reply queue. The next section deals with the same synchronous communication, but now Citrus will receive a request and send a synchronous reply message to a temporary reply queue.

As usual the reply queue name is stored in the message headers. Citrus handles this synchronous communication with the same synchronous direct endpoint component. The handling of temporary reply destinations is done automatically behind the scenes.

So we have again two actions in our test case, but this time first receive then send.

Java

```
when(receive("helloSyncEndpoint")
        .message()
        .body("<v1:HelloRequest
xmlns:v1=\"http://citrusframework.org/schemas/HelloService.xsd\">" +
                "<v1:Text>Hello World!</v1:Text>" +
            "</v1:HelloRequest>"));

then(send("helloSyncEndpoint")
        .message()
        .body("<v1:HelloResponse
xmlns:v1=\"http://citrusframework.org/schemas/HelloService.xsd\">" +
                "<v1:Text>Hello Citrus!</v1:Text>" +
            "</v1:HelloResponse>"));
```

XML

```
<receive endpoint="helloSyncEndpoint">
    <message>
        <payload>
            <v1:HelloRequest
xmlns:v1="http://citrusframework.org/schemas/HelloService.xsd">
                <v1:Text>Hello World!</v1:Text>
            </v1:HelloRequest>
        </payload>
    </message>
</receive>

<send endpoint="helloSyncEndpoint">
    <message>
        <payload>
            <v1:HelloResponse
xmlns:v1="http://citrusframework.org/schemas/HelloService.xsd">
                <v1:Text>Hello Citrus!</v1:Text>
            </v1:HelloResponse>
        </payload>
    </message>
</send>
```

11.3. Message selectors

A queue can hold multiple messages at the same time. Usually you receive messages using first-in-first-out pattern. Message selectors enable you to select messages form that queue so you can pick messages form a queue based on a selector evaluation.

Citrus introduces a special message queue implementation that supports message selectors.

Java

```
@Bean
public MessageSelectingQueueChannel helloChannel() {
    return new MessageSelectingQueueChannel();
}
```

XML

```
<citrus:queue id="orderChannel" capacity="5"/>
```

We can add a capacity attribute for this queue. A receive test action makes use of message selectors on header values as described in message-selector.

In addition to that we have implemented other message filter possibilities on message queues that we discuss in the next sections.

11.4. Payload matching selector

You can select messages based on the payload content. Either you define the expected payload as an exact match in the selector or you make use of Citrus validation matchers which is more adequate in most scenarios.

Assume there are two different plain text messages living on a message queue waiting to be picked up by a consumer.

```
Hello, welcome!
```

```
GoodBye, see you next time!
```

The tester would like to pick up the message starting with **GoodBye** in our test case. The other messages should be left on the queue as we are not interested in it right now. We can define a payload matching selector in the receive action like this:

Java

```
when(receive("orderChannelEndpoint")
        .selector(Collections.singletonMap("payload", "@startsWith(GoodBye)@"))
        .message()
        .body("GoodBye, see you next time!"));
```

XML

```
<receive endpoint="orderChannelEndpoint">
    <selector>
        <element name="payload" value="@startsWith(GoodBye)@"/>
    </selector>
    <message>
        <payload>GoodBye, see you next time!</payload>
    </message>
</receive>
```

The Citrus receiver picks up the **GoodBye** from the queue selected via the payload matching expression defined in the selector element. Of course, you can also combine message header selectors and payload matching selectors as shown in this example below where a message header **sequenceId** is added to the selection logic.

Java

```
Map<String, String> selectorMap = new HashMap<>();
selectorMap.put("payload", "@startsWith(GoodBye)@");
selectorMap.put("sequenceId", "1234");

when(receive("orderChannelEndpoint")
        .selector(selector)
        .message()
        .body("GoodBye, see you next time!"));
```

XML

```
<selector>
    <element name="payload" value="@startsWith(GoodBye)@"/>
    <element name="sequenceId" value="1234"/>
</selector>
```

11.5. Root QName selector

As a special payload matching selector you can use the XML root QName of your message as selection criteria when dealing with XML message content. Let's see how this works in a small example:

We have two different XML messages on a message queue waiting to be picked up by a consumer.

```
<HelloMessage xmlns="http://citrusframework.org/schema">Hello Citrus</HelloMessage>
```

```
<GoodbyeMessage xmlns="http://citrusframework.org/schema">Goodbye
Citrus</GoodbyeMessage>
```

We would like to pick up the **GoodbyeMessage** in our test case. The **HelloMessage** should be left on the message queue as we are not interested in it right now. We can define a root qname message selector in the receive action like this:

Java

```
when(receive("orderChannelEndpoint")
        .selector(Collections.singletonMap("root-qname", "GoodbyeMessage"))
        .message()
        .body("<GoodbyeMessage xmlns=\"http://citrusframework.org/schema\">Goodbye
Citrus</GoodbyeMessage>"));
```

XML

```
<receive endpoint="orderChannelEndpoint">
    <selector>
        <element name="root-qname" value="GoodbyeMessage"/>
    </selector>
    <message>
        <payload>
            <GoodbyeMessage xmlns="http://citrusframework.org/schema">Goodbye
Citrus</GoodbyeMessage>
        </payload>
    </message>
</receive>
```

The Citrus receiver picks up the **GoodbyeMessage** from the queue selected via the root qname of the XML message payload. Of course, you can also combine message header selectors and root qname selectors as shown in this example below where a message header **sequenceId** is added to the selection logic.

Java

```
Map<String, String> selectorMap = new HashMap<>();
selectorMap.put("root-qname", "GoodbyeMessage");
selectorMap.put("sequenceId", "1234");

when(receive("orderChannelEndpoint")
        .selector(selector)
        .message()
        .body("GoodBye, see you next time!"));
```

XML

```
<selector>
    <element name="root-qname" value="GoodbyeMessage"/>
    <element name="sequenceId" value="1234"/>
</selector>
```

As we deal with XML qname values, we can also use namespaces in our selector root qname selection.

Java

```
when(receive("orderChannelEndpoint")
        .selector(Collections.singletonMap("root-qname",
"{http://citrusframework.org/schema}GoodbyeMessage"))
        .message()
        .body("<GoodbyeMessage xmlns=\"http://citrusframework.org/schema\">Goodbye
Citrus</GoodbyeMessage>"));
```

XML

```
<selector>
    <element name="root-qname"
value="{http://citrusframework.org/schema}GoodbyeMessage"/>
</selector>
```

11.6. Xpath selector

It is also possible to evaluate some XPath expression on the message payload in order to select a message from a message queue. The XPath expression outcome must match an expected value and only then the message is consumed from the queue.

The syntax for the XPath expression is to be defined as the element name like this:

Java

```
when(receive("orderChannelEndpoint")
        .selector(Collections.singletonMap("xpath://Order/status", "pending"))
        .message()
        .body("<Order><status>pending</status></Order>"));
```

XML

```
<selector>
    <element name="xpath://Order/status" value="pending"/>
</selector>
```

The message selector looks for order messages with **status="pending"** in the message payload. This means that following messages would get accepted/declined by the message selector.

```
<Order><status>pending</status></Order> <!-- ACCEPTED -->
<Order><status>finished</status></Order> <!-- NOT ACCEPTED -->
```

Of course, you can also use XML namespaces in your XPath expressions when selecting messages from queues.

Java

```
when(receive("orderChannelEndpoint")
        .selector(Collections.singletonMap("xpath://ns1:Order/ns1:status", "pending"))
        .message()
        .body("<Order><status>pending</status></Order>"));
```

XML

```
<selector>
    <element name="xpath://ns1:Order/ns1:status" value="pending"/>
</selector>
```

Namespace prefixes must match the incoming message - otherwise the XPath expression will not work as expected. In our example the message should look like this:

```
<ns1:Order
xmlns:ns1="http://citrus.org/schema"><ns1:status>pending</ns1:status></ns1:Order>
```

Knowing the correct XML namespace prefix is not always easy. If you are not sure which namespace prefix to choose Citrus ships with a dynamic namespace replacement for XPath expressions. The XPath expression looks like this and is most flexible:

Java

```
when(receive("orderChannelEndpoint")
        .selector(Collections.singletonMap(

"xpath://{http://citrus.org/schema}:Order/{http://citrus.org/schema}:status",
"pending"))
        .message()
        .body("<Order><status>pending</status></Order>"));
```

XML

```
<selector>
    <element
name="xpath://{http://citrus.org/schema}:Order/{http://citrus.org/schema}:status"
                value="pending"/>
</selector>
```

This will match all incoming messages regardless the XML namespace prefix that is used.

11.7. JsonPath selector

It is also possible to evaluate some JsonPath expression on the message payload in order to select a message from a message queue. The JsonPath expression outcome must match an expected value and only then the message is consumed from the queue.

The syntax for the JsonPath expression is to be defined as the element name like this:

Java

```
when(receive("orderChannelEndpoint")
        .selector(Collections.singletonMap("jsonPath:$.order.status", "pending"))
        .message()
        .body("{ \"order\": { \"status\": \"pending\" } }"));
```

XML

```
<selector>
    <element name="jsonPath:$.order.status" value="pending"/>
</selector>
```

The message selector looks for order messages with **status="pending"** in the message payload. This means that following messages would get accepted/declined by the message selector.

```
{ "order": { "status": "pending" } } //ACCEPTED
{ "order": { "status": "finished" } } //NOT ACCEPTED
```

Chapter 12. JMS support

Citrus provides support for sending and receiving JMS messages. We have to separate between synchronous and asynchronous communication. So in this chapter we explain how to work with JMS message endpoints for synchronous and asynchronous communication

The JMS components in Citrus are kept in a separate Maven module. If not already done so you have to include the module as Maven dependency to your project

Maven module dependency

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-jms</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

Citrus provides a "citrus-jms" configuration namespace and schema definition for JMS related components and features. Include this namespace into your Spring configuration in order to use the Citrus JMS configuration elements. The namespace URI and schema location are added to the Spring configuration XML file as follows.

Spring configuration namespace

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:citrus-jms="http://www.citrusframework.org/schema/jms/config"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.citrusframework.org/schema/jms/config
       http://www.citrusframework.org/schema/jms/config/citrus-jms-config.xsd">

    [...]

</beans>
```

Now you are able to use customized Citrus XML elements in order to define the JMS endpoint components.

12.1. JMS endpoints

By default, Citrus JMS endpoints are asynchronous. Asynchronous messaging means that the endpoint will not wait for a response message after sending a message.

The test case itself should not know about JMS transport details like queue names or connection credentials. This information is stored in the endpoint component configuration that lives in the basic project configuration files in Citrus. So let us have a look at a simple JMS message endpoint

configuration in Citrus.

Java

```
@Bean
public JmsEndpoint helloServiceEndpoint() {
    return new JmsEndpointBuilder()
        .destination("Citrus.HelloService.Request.Queue")
        .timeout(10000L)
        .build();
}
```

XML

```
<citrus-jms:endpoint id="helloServiceEndpoint"
          destination-name="Citrus.HelloService.Request.Queue"
          timeout="10000"/>
```

The endpoint component receives a unique id as well as a JMS destination name. This can be a queue or topic destination. JMS topics are described later on in this chapter. For now the timeout setting completes the first JMS endpoint component definition example.

In addition to the `destination-name` attribute you can also provide a reference to a destination implementation.

Java

```
@Bean
public JmsEndpoint helloServiceEndpoint() {
    return new JmsEndpointBuilder()
        .destination(helloServiceQueue())
        .build();
}

@Bean
public ActiveMQQueue helloServiceQueue() {
    return new ActiveMQQueue("Citrus.HelloService.Request.Queue");
}
```

XML

```
<citrus-jms:endpoint id="helloServiceEndpoint"
                     destination="helloServiceQueue"/>

<amq:queue id="helloServiceQueue" physicalName="Citrus.HelloService.Request.Queue"/>
```

The destination attribute references to a JMS destination object in the same Spring application context. In the example above we used the ActiveMQ queue destination component. The

destination reference can also refer to a JNDI lookup for instance.

The endpoint needs a JMS connection factory for connecting to a JMS message broker. The connection factory is also added as component bean to the Citrus project (e.g. in the Spring application context).

Java

```
@Bean
public ActiveMQConnectionFactory connectionFactory() {
    ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory();
    factory.setBrokerURL("tcp://localhost:61616");
    return factory;
}
```

XML

```
<bean id="connectionFactory"
class="org.apache.activemq.artemis.jms.client.ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>
```

The JMS connection factory receives the broker URL and is able to hold many other connection specific options. In this example we use the Apache ActiveMQ connection factory implementation as we want to use the ActiveMQ message broker. Citrus works with a bean id **connectionFactory**. All Citrus JMS component will automatically recognize this connection factory.

The configuration makes it very easy to connect to other JMS broker implementations, too (e.g. Apache ActiveMQ, TIBCO Enterprise Messaging Service, IBM Websphere MQ). Just add the required connection factory implementation as **connectionFactory** bean.

All JMS endpoint components in Citrus will automatically load the factory named **connectionFactory**. You can use the **connection-factory** endpoint attribute in order to use another connection factory instance with different bean names.

Java

```
@Bean
public JmsEndpoint helloServiceEndpoint() {
    return new JmsEndpointBuilder()
        .destination("Citrus.HelloService.Request.Queue")
        .connectionFactory(myConnectionFactory())
        .build();
}
```

XML

```
<citrus-jms:endpoint id="helloServiceEndpoint"
      destination-name="Citrus.HelloService.Request.Queue"
      connection-factory="myConnectionFactory"/>
```

As an alternative to that you may want to use a special Spring JMS template implementation as custom bean in your endpoint.

Java

```
@Bean
public JmsEndpoint helloServiceEndpoint() {
    return new JmsEndpointBuilder()
        .destination("Citrus.HelloService.Request.Queue")
        .jmsTemplate(myJmsTemplate())
        .build();
}
```

XML

```
<citrus-jms:endpoint id="helloServiceEndpoint"
              destination-name="Citrus.HelloService.Request.Queue"
              jms-template="myJmsTemplate"/>
```

The endpoint is now ready to be used inside a test case. You can send or receive messages using this endpoint. The test actions reference the JMS endpoint using its unique identifier. When sending a message the message endpoint creates a JMS message producer and will simply publish the message to the defined JMS destination. As the communication is asynchronous by default the producer does not wait for a synchronous response.

When receiving messages the endpoint creates a JMS consumer on the JMS destination. The endpoint then acts as a message driven listener. This means that the message consumer connects to the given destination and waits for messages to arrive.

Java

```
when(send("helloServiceEndpoint")
        .message()
        .body("..."));

then(receive("helloServiceEndpoint")
        .message()
        .body("..."));
```

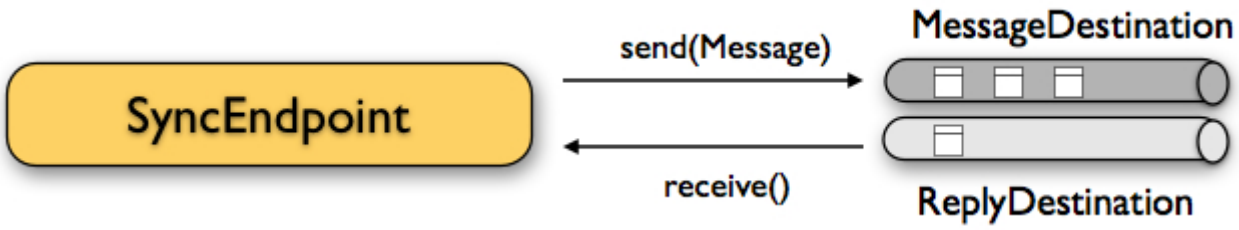
XML

```
<testcase name="jmsMessagingTest">
    <actions>
        <send endpoint="helloServiceEndpoint">
            <message>
                <data>
                  [...]
                </data>
            </message>
        </send>

        <receive endpoint="helloServiceEndpoint">
            <message>
                <data>
                  [...]
                </data>
            </message>
        </receive>
    </actions>
</testcase>
```

12.2. JMS synchronous endpoints

When using synchronous message endpoints Citrus will manage a reply destination for receiving a synchronous response message on the reply destination. The following figure illustrates that we now have two destinations in our communication scenario.



The synchronous message endpoint component is similar to the asynchronous variant that has been discussed before. The only difference is that the endpoint will automatically manage a reply destination behind the scenes. By default, Citrus uses temporary reply destinations that get automatically deleted after the communication handshake is done. Again we need to use a JMS connection factory in the configuration as the component needs to connect to a JMS message broker.

Java

```
@Bean
public JmsSyncEndpoint helloServiceSyncEndpoint() {
    return new JmsSyncEndpointBuilder()
        .destination("Citrus.HelloService.InOut.Queue")
        .build();
}
```

XML

```
<citrus-jms:sync-endpoint id="helloServiceSyncEndpoint"
          destination-name="Citrus.HelloService.InOut.Queue"
          timeout="10000"/>
```

The synchronous component defines a target destination which again is either a queue or topic destination. The endpoint will create the temporary reply destinations on its own. As soon as the endpoint has published a request message it waits synchronously for the response message to arrive at the reply destination. You can receive this reply message in your test case by referencing this same endpoint in a receiving test action. The timeout setting defines how long the endpoint waits for the synchronous reply. In case no reply message arrives in time a message timeout error is raised respectively.

See the following example test case which references the synchronous message endpoint in its send and receive test action in order to send out a message and wait for the synchronous response.

Java

```
when(send("helloServiceSyncEndpoint")
        .message()
        .body("..."));

then(receive("helloServiceSyncEndpoint")
        .message()
        .body("..."));
```

XML

```
<testcase name="jmsSyncMessagingTest">
    <actions>
        <send endpoint="helloServiceSyncEndpoint">
            <message>
                <data>
                  [...]
                </data>
            </message>
        </send>

        <receive endpoint="helloServiceSyncEndpoint">
            <message>
                <data>
                  [...]
                </data>
            </message>
        </receive>
    </actions>
</testcase>
```

We initiated the synchronous communication by sending a message on the synchronous endpoint. The second step then receives the synchronous message on the temporary reply destination that was automatically created for you.

If you rather want to define a static reply destination you can do so, too. The static reply destination is not deleted after the communication handshake. You may need to work with message selectors then in order to pick the right response message that belongs to a specific communication handshake. You can define a static reply destination on the synchronous endpoint component as follows.

Java

```
@Bean
public JmsSyncEndpoint helloServiceSyncEndpoint() {
    return new JmsSyncEndpointBuilder()
        .destination("Citrus.HelloService.InOut.Queue")
        .replyDestination("Citrus.HelloService.Reply.Queue")
        .build();
}
```

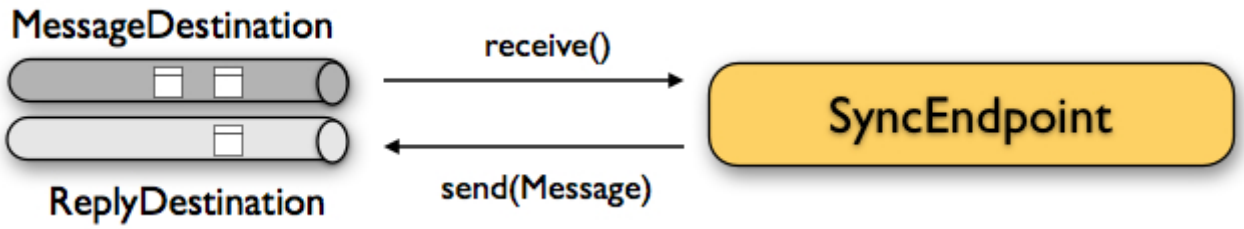
XML

```
<citrus-jms:sync-endpoint id="helloServiceSyncEndpoint"
          destination-name="Citrus.HelloService.InOut.Queue"
          reply-destination-name="Citrus.HelloService.Reply.Queue"
          timeout="10000"/>
```

Instead of using the **reply-destination-name** feel free to use the destination reference with **reply-destination** attribute. Again you can use a JNDI lookup then to reference a destination object.

> Be aware of permissions that are mandatory for creating temporary destinations. Citrus tries to create temporary queues on the JMS message broker. Following from that the Citrus JMS user has to have the permission to do so. Be sure that the user has the sufficient rights when using temporary reply destinations.

Up to now we have sent a message and waited for a synchronous response in the next step. Now it is also possible to switch the directions of send and receive actions. Then we have the situation where Citrus receives a JMS message first and then Citrus is in charge of providing a proper synchronous response message to the initial sender.



In this scenario the foreign message producer has stored a dynamic JMS reply queue destination to the JMS header. So Citrus has to send the reply message to this specific reply destination, which is dynamic of course. Fortunately the heavy lift is done with the JMS endpoint and we do not have to change anything in our configuration. Again we just define a synchronous message endpoint in the application context.

Java

```
@Bean
public JmsSyncEndpoint helloServiceSyncEndpoint() {
    return new JmsEndpointBuilder()
        .destination("Citrus.HelloService.InOut.Queue")
        .build();
}
```

XML

```
<citrus-jms:sync-endpoint id="helloServiceSyncEndpoint"
      destination-name="Citrus.HelloService.InOut.Queue"
      timeout="10000"/>
```

Now the only thing that changes here is that we first receive a message in our test case on this endpoint. The second step is a send message action that references this same endpoint and we are done. Citrus automatically manages the reply destinations for us.

Java

```
when(receive("helloServiceSyncEndpoint")
        .message()
        .body("..."));

then(send("helloServiceSyncEndpoint")
        .message()
        .body("..."));
```

XML

```
<testcase name="jmsSyncMessagingTest">
  <actions>
        <receive endpoint="helloServiceSyncEndpoint">
            <message>
                <data>
                  [...]
                </data>
            </message>
        </receive>

        <send endpoint="helloServiceSyncEndpoint">
            <message>
                <data>
                  [...]
                </data>
            </message>
        </send>
    </actions>
</testcase>
```

12.3. JMS topics

Up to now we have used JMS queue destinations on our endpoints. Citrus is also able to connect to JMS topic destinations. In contrary to JMS queues which represents the **point-to-point** communication JMS topics use **publish-subscribe** mechanism in order to spread messages over JMS.

A JMS topic producer publishes messages to the topic, while the topic accepts multiple message subscriptions and delivers the message to all subscribers.

The Citrus JMS endpoints offer the attribute **'pub-sub-domain'**. Once this attribute is set to **true** Citrus will use JMS topics instead of queue destinations.

When using JMS topics in your project you may want to configure a `jakarta.jms.TopicConnectionFactory` instead of a `jakarta.jms.QueueConnectionFactory`.

See the following example where the publish-subscribe attribute is set to true in JMS message endpoint components.

Java

```
@Bean
public JmsSyncEndpoint helloServiceSyncEndpoint() {
    return new JmsEndpointBuilder()
        .destination("Citrus.HelloService.Topic")
        .pubSubDomain(true)
        .build();
}
```

XML

```
<citrus-jms:endpoint id="helloServiceTopicEndpoint"
            destination="Citrus.HelloService.Topic"
            pub-sub-domain="true"/>
----
```

When using JMS topics you will be able to subscribe several test actions to the topic destination and receive a message multiple times as all subscribers will receive the message. Also other applications besides Citrus are also able to consume messages with a topic subscription. This allows Citrus and other software components to coexist in a test environment.

12.3.1. JMS topic subscriber

By default, Citrus does not deal with durable subscribers when using JMS topics. This means that messages that were sent in advance to the message subscription are not delivered to the Citrus message endpoint. Following from that racing conditions may cause problems when using JMS topic endpoints in Citrus.

Be sure to start the Citrus subscription before messages are sent to the topic. Otherwise, you may lose some messages that were sent in advance to the subscription. By default Citrus will use a subscription per receive action using the JMS endpoint in the test cases. This means that the topic subscription is started and stopped per receive action when the action is performed inside a test case.

In order to solve racing conditions for messages that are sent prior to the subscription you can also use an `auto-start` setting on the JMS endpoint component. This causes Citrus to start/stop the subscription based on the endpoint lifecycle instead of linking the subscription to the receive action. When the endpoint is ready the subscription is started and all incoming message events are cached and stored to an internal in-memory message channel for later consumption in the tests.

Here is the endpoint configuration with `auto-start` enabled.

Java

```
@Bean
public JmsSyncEndpoint helloServiceSyncEndpoint() {
    return new JmsEndpointBuilder()
        .destination("Citrus.HelloService.Topic")
        .pubSubDomain(true)
        .autoStart(true)
        .build();
}
```

XML

```
<citrus-jms:endpoint id="helloServiceTopicEndpoint"
            destination="helloServiceTopic"
            pub-sub-domain="true"
            auto-start="true"/>
```

The `auto-start` option is only valid in combination with `pub-sub-domain` enabled. Other combinations may be ignored or lead to configuration failure at start-up.

Now with `auto-start` set to `true` the Citrus JMS endpoint will setup a subscription at the very beginning when the endpoint is loaded in the project. The internal message channel name is derived from the JMS endpoint id and follows the pattern:

`{citrus-jms:endpoint@id}":subscriber.inbound"`

The in-memory channel id is the combined result of the JMS endpoint id and the postfix `:subscriber.inbound`. In our example this would be `helloServiceTopicEndpoint:subscriber.inbound`. Now all messages sent to the topic in advance to the tests are cached and ready for consumption and verification in the test.

In the test nothing really changes for you. You simply use a receive test action on the JMS endpoint as you would have done before. In the background Citrus will automatically receive the messages from the in memory cache. This mechanism enables us to not loose any messages that were sent to the topic in prior to Citrus firing up the test cases.

There is a small downside of the `auto-start` topic subscriber. As incoming events are cached internally you will not be able to receive the same topic event in multiple receive actions within the Citrus project. If you need to receive the topic message in several places within Citrus you need to set up several JMS topic endpoints with `auto-start` enabled. In case you just have one receive action at a time you are good to go with the `auto-start` subscriber as it is described here.

12.4. JMS topic durable subscription

When using durable subscriptions on JMS message brokers the message events on a topic are preserved for a subscriber even if the subscriber is inactive. This means that the subscriber may

not loose any message events on that particular topic as the subscription is durable and all events are stored for later consumption.

In case you want to activate durable subscriptions on the Citrus JMS endpoint use the `durable-subscription` setting in the configuration:

Java

```
@Bean
public JmsSyncEndpoint helloServiceSyncEndpoint() {
    return new JmsEndpointBuilder()
        .destination("Citrus.HelloService.Topic")
        .pubSubDomain(true)
        .durableSubscription(true)
        .autoStart(true)
        .build();
}

@Bean SingleConnectionFactory topicConnectionFactory() {
    ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory();
    factory.setBrokerURL("tcp://localhost:61616");
    factory.setClientID("citrusDurableConnectionFactory");
    factory.setWatchTopicAdvisories(false);

    return new SingleConnectionFactory(factory);
}
```

XML

```
<citrus-jms:endpoint id="helloServiceTopicEndpoint"
            connection-factory="topicConnectionFactory"
            destination="helloServiceTopic"
            pub-sub-domain="true"
            durable-subscription="true"
            auto-start="true"/>

<bean class="org.apache.activemq.artemis.core.server.embedded.EmbeddedActiveMQ" init-
method="start" destroy-method="stop">
  <property name="SecurityManager" ref="securityManager"/>
</bean>

<bean id="topicConnectionFactory"
class="org.springframework.jms.connection.SingleConnectionFactory">
  <constructor-arg>
    <bean
class="org.apache.activemq.artemis.jms.client.ActiveMQTopicConnectionFactory">
      <property name="brokerURL" value="tcp://localhost:61616"/>
      <property name="clientID" value="citrusDurableConnectionFactory"/>
    </bean>
  </constructor-arg>
</bean>

<bean id="securityManager"
class="org.apache.activemq.artemis.spi.core.security.ActiveMQJAASSecurityManager">
  <constructor-arg
value="org.apache.activemq.artemis.spi.core.security.jaas.InVMLoginModule"/>
  <constructor-arg>
    <bean class="org.apache.activemq.artemis.core.config.impl.SecurityConfiguration">
      <constructor-arg name="users">
        <map>
          <entry key="citrus" value="citrus"/>
        </map>
      </constructor-arg>
      <constructor-arg name="roles">
        <map>
          <entry key="citrus">
            <list>
              <value>citrus</value>
            </list>
          </entry>
        </map>
      </constructor-arg>
      <property name="DefaultUser" value="citrus"/>
    </bean>
  </constructor-arg>
</bean>
```

The durable subscription in Citrus implies that the subscriber is started when the endpoint configuration is done. All messages received on that subscription are cached internally until the receive action in the test case is performed for actual message consumption. The `auto-start` setting is required to be enabled for this reason when using durable subscriptions.

By default, Citrus is using the JMS endpoint subscriber name as durable subscription name (e.g. **helloServiceTopicEndpoint:subscriber**). You can overwrite the durable subscriber name with `durable-subscriber-name` setting on the endpoint.

In addition to that you need to add a client id on the connection factory so the message broker is able to identify the durable subscription with the client address. Also we use the `SingleConnectionFactory` implementation of Spring as a connection factory wrapper so we do not fail because of multiple connections with the same durable subscriber id.

12.5. JMS topic purging

As mentioned earlier, topic subscribers with `autoStart=true` consume all messages sent to the topic during the test run. Durable topic subscribers even consume messages sent outside the test run. Both cache the consumed messages internally in an in-memory-message-queue. To ensure that tests start with a clean in-memory-message-queue, it is necessary to purge the topic-subscribtion JmsEndpoint (which is defined in the code examples above) before the test begins. Endpoint-purging is explained in detail in this section: Purging Endpoints.

12.6. JMS message headers

The JMS specification defines a set of special message header entries that can go into your JMS message. These JMS headers are stored differently in a JMS message header than other custom header entries do. This is why these special header values should be set in a special syntax that we discuss in the next paragraphs.

Java

```
when(receive("helloServiceSyncEndpoint")
        .message()
        .header("citrus_jms_correlationId", "${correlationId}")
        .header("citrus_jms_messageId", "${messageId}")
        .header("citrus_jms_redelivered", "${redelivered}")
        .header("citrus_jms_timestamp", "${timestamp}")
        .body("..."));
```

XML

```
<header>
    <element name="citrus_jms_correlationId" value="${correlationId}"/>
    <element name="citrus_jms_messageId" value="${messageId}"/>
    <element name="citrus_jms_redelivered" value="${redelivered}"/>
    <element name="citrus_jms_timestamp" value="${timestamp}"/>
</header>
```

As you see all JMS specific message headers use the `citrus_jms_` prefix. This prefix comes from Spring Integration message header mappers that take care of setting those headers in the JMS message header properly.

Typing of message header entries may also be of interest in order to meet the JMS standards of typed message headers. For instance the following message header is of type double and is therefore transferred via JMS as a double value.

Java

```
when(receive("jmsEndpoint")
        .message()
        .header("amount", 19.75D)
        .body("..."));
```

XML

```
<header>
    <element name="amount" value="19.75" type="double"/>
</header>
```

12.7. Dynamic destination names

Usually you set the target destination as property on the JMS endpoint component. In some cases it might be useful to set the target destination in a more dynamic way during the test run. You can do this by adding a special message header named **citrus_jms_destination_name**. This header is automatically interpreted by the Citrus JMS endpoint and is set as the target destination before a message is sent.

Java

```
when(send("jmsEndpoint")
        .message()
        .header("citrus_jms_destination_name", "dynamic.destination.name")
        .body("..."));
```

XML

```
<send endpoint="jmsEndpoint">
    <message>
        ...
    </message>
    <header>
        <element name="citrus_jms_destination_name" value="dynamic.destination.name"/>
    </header>
</send>
```

This action above will send the message to the destination "*dynamic.destination.name*" no matter

what default destination is set on the referenced endpoint component named *jmsEndpoint*. The dynamic destination name setting also supports test variables. This means you can use variables and functions in the destination name, too.

Another possibility for dynamic JMS destinations is given with the dynamic endpoints.

12.8. SOAP over JMS

When sending SOAP messages you have to deal with proper envelope, body and header construction. In Citrus you can add a special message converter that performs the heavy lift for you. Just add the message converter to the JMS endpoint as shown in the next program listing:

Java

```
@Bean
public JmsSyncEndpoint helloServiceSoapJmsEndpoint() {
    return new JmsEndpointBuilder()
        .destination("Citrus.HelloService.Request.Queue")
        .messageConverter(soapJmsMessageConverter())
        .build();
}

@Bean
public SoapJmsMessageConverter soapJmsMessageConverter() {
    return new SoapJmsMessageConverter();
}
```

XML

```
<citrus-jms:endpoint id="helloServiceSoapJmsEndpoint"
        destination-name="Citrus.HelloService.Request.Queue"
        message-converter="soapJmsMessageConverter"/>

<bean id="soapJmsMessageConverter"
class="org.citrusframework.jms.message.SoapJmsMessageConverter"/>
```

With this message converter you can skip the SOAP envelope completely in your test case. You just deal with the message body payload and the header entries. The rest is done by the message converter. So you get proper SOAP messages on the producer and consumer side.

Chapter 13. Apache Kafka Support

Kafka is a distributed streaming platform that enables you to publish and subscribe to streams of records, similar to a message queue or enterprise messaging system. Citrus provides support for publishing/consuming records to/from a Kafka topic. Citrus acts as producer or consumer as the Citrus Kafka endpoint can be used bidirectional. In the current version Citrus supports asynchronous communication only.

> The Kafka components in Citrus are shipped in a separate Maven module. If not already done so you have to include the module as Maven dependency to your project

Maven dependency

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-kafka</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

In case you are using XML Spring configuration files Citrus provides a "citrus-kafka" configuration namespace and schema definition for Kafka related components and features. Include this namespace into your Spring XML configuration in order to use the Citrus Kafka configuration elements. The namespace URI and schema location are added to the Spring bean root element.

Spring bean configuration namespace

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:citrus-kafka="http://www.citrusframework.org/schema/kafka/config"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.citrusframework.org/schema/kafka/config
       http://www.citrusframework.org/schema/kafka/config/citrus-kafka-config.xsd">

    [...]

</beans>
```

Now you are able to use customized Citrus XML elements in order to define the Kafka endpoint components. In case you are using the Spring Java configuration with `@Bean` annotations you do not require this step.

13.1. Kafka Endpoint

By default, Citrus Kafka endpoints are asynchronous. Asynchronous messaging means that the

endpoint will not wait for any response message after sending or receiving a message.

The endpoint component configuration holds transport specific configuration details such as topic names and server connectivity settings. So let us have a look at a simple Kafka message endpoint configuration in Citrus.

Java

```
@Bean
public KafkaEndpoint helloKafkaEndpoint() {
    return new KafkaEndpointBuilder()
                     .topic("hello")
                     .server("localhost:9092")
                     .build();
}
```

XML

```
<citrus-kafka:endpoint id="helloKafkaEndpoint"
          topic="hello"
          server="localhost:9092"/>
```

The endpoint component receives a unique id as well as a Kafka topic name. The bootstrap server url that points to the Kafka message brokers completes our first Kafka endpoint component definition. With this configuration you will be able to send and receive records on the given topic.

By default, the endpoint uses the topic partition `0`. The consumer on this endpoint is automatically added to a consumer group `citrus_kafka_group`. You can customize these settings on the endpoint.

Java

```
@Bean
public KafkaEndpoint helloKafkaEndpoint() {
    return new KafkaEndpointBuilder()
                     .topic("hello")
                     .server("localhost:9092")
                     .partition(1)
                     .consumerGroup("citrus_group")
                     .build();
}
```

XML

```
<citrus-kafka:endpoint id="helloKafkaEndpoint"
          topic="hello"
          server="localhost:9092"
          partition="1"
          consumer-group="citrus_group"/>
```

The endpoint is now ready to be used inside a test case. The test simply references the endpoint by its name when sending or receiving.

> The `org.apache.kafka.clients.consumer.KafkaConsumer<K,V>` is **not thread-safe**. When using selective message consumption, it is recommended to configure `useThreadSafeConsumer` respectively `thread-safe-consumer` for the Kafka endpoint. Otherwise, you cannot execute tests in parallel.

> For effective message consumption, it is additionally advisable to use random consumer groups. Both the Java and XML DSL support random consumer groups, when enabled: `randomConsumerGroup(true)` or `random-consumer-group="true"`.

In case of a send operation the endpoint creates a Kafka producer and will simply publish the records to the defined Kafka topic. As the communication is asynchronous by default, the producer does not wait for a response.

In case of a receive operation the endpoint creates a Kafka consumer instance in the defined (possibly random) consumer group.

> By default, the consumer starts a subscription on the given topic and acts as a polling listener. This means that the message consumer connects to the given topic and polls one single record at a time. The action is the ready to perform the validation process that verifies the record, once it has been received. This approach requires looping to find specific Kafka messages, which can be highly inefficient, especially when the Kafka topic experiences high traffic. See Kafka message selectors for a more effective Kafka message selection.

13.1.1. Configuration

The following table shows all available settings on a Kafka endpoint in Citrus:

Property	Mandatory	Default	Description
`id`	Yes	-	Identifying name of the endpoint. Only required for XML configuration.
`topic`	No	-	Default topic to use with this endpoint. Multiple topics are supported by using a comma delimited list of names (e.g. `topic1,topic2,topicN`). If not specified the test case send operation needs to set the topic as message header information.

Property	Mandatory	Default	Description
`server`	No	`localhost:9092`	A comma delimited list of host/port pairs to use for establishing the initial connection to the Kafka cluster. Usually it is only required to connect to one Kafka server instance in the cluster. Kafka then makes sure that the endpoint is automatically introduced to all other servers in the cluster. This list only impacts the initial hosts used to discover the full set of servers.
`timeout`	No	`5000`	Timeout in milliseconds. For producers the timeout is set as time to wait for the message to be accepted by the cluster. For consumers the timeout is used for polling records on a specific topic.
`message-converter`	No	`org.citrusframework.kafka.message.KafkaMessageConverter`	Converter maps internal Citrus message objects to `ProducerRecord`/`ConsumerRecord` objects. The converter implementation takes care of message key, value, timestamp and special message headers.
`header-mapper`	No	`org.citrusframework.kafka.message.KafkaMessageHeaderMapper`	Header mapper maps Kafka record information (e.g. topic name, timestamp, message key) to internal message headers (`org.citrusframework.kafka.message.KafkaMessageHeaders`) and vice versa.
`auto-commit`	No	`true`	When this setting is enabled the consumer will automatically commit consumed records so the offset pointer on the Kafka topic is set to the next record.
`auto-commit-interval`	No	`1000`	Interval in milliseconds the auto commit operation on consumed records is performed.

Property	Mandatory	Default	Description
offset-reset	No	`earliest`	When consuming records from a topic partition and the current offset does not exist on that partition Kafka will automatically seek to a valid offset position on that partition. The `offset-reset` setting sets where to find the new position (`latest`, `earliest`, `none`). If `none` is set the consumer will receive an exception instead of resetting the offset to a valid position.
`partition`	No	`0`	Partition id that the consumer will be assigned to.
`consumer-group`	No	`citrus_kafka_group`	Consumer group name. Please keep in mind that records are load balanced across consumer instances with the same consumer group name set. So you might run into message timeouts when using multiple Kafka endpoints with the same consumer group name.
`random-consumer-group`	No	`false`	Whether to use random consumer group names. Note that these will all be prefixed by `citrus_kafka_` and end with a random 10 characters alphabetic suffix.
`key-serializer`	No	`org.apache.kafka.common.serialization.StringSerializer`	Serializer implementation that converts message key values. By default, keys are serialized to String values.
`key-deserializer`	No	`org.apache.kafka.common.serialization.StringDeserializer`	Deserializer implementation that converts message key values. By default, keys are deserialized as String values.
`value-serializer`	No	`org.apache.kafka.common.serialization.StringSerializer`	Serializer implementation that converts record values. By default values are serialized to String values.
`value-deserializer`	No	`org.apache.kafka.common.serialization.StringDeserializer`	Deserializer implementation that converts record values. By default, values are deserialized as String values.

Property	Mandatory	Default	Description
client-id	No	citrus_kafka_[producer/consumer]_{randomUUID}	An id string to pass to the server when producing/consuming records. Used as logical application name to be included in server-side request logging.
consumer-properties	No	-	Map of consumer property settings to apply to the Kafka consumer configuration. This enables you to overwrite any consumer setting with respective property key value pairs.
producer-properties	No	-	Map of producer property settings to apply to the Kafka producer configuration. This enables you to overwrite any producer setting with respective property key value pairs.
thread-safe-consumer	No	false	Whether to use separate org.apache.kafka.clients.consumer.KafkaConsumer<K,V>'s per thread. Required for parallel test execution, because the consumer is **not thread-safe**.

13.1.2. Producer and Consumer Properties

The Citrus Kafka endpoint component is also able to receive a map of Kafka producer and consumer properties. These property settings overwrite any predefined setting on the producer/consumer instance created by the endpoint. You can use the Kafka property keys with respective values for producer and consumer config maps.

Java

```
@Bean
public KafkaEndpoint helloKafkaEndpoint() {
    return new KafkaEndpointBuilder()
                      .consumerProperties(getConsumerProps())
                      .producerProperties(getProducerProps())
                      .build();
}

private Map<String, Object> getProducerProps() {
    // ...
}

private Map<String, Object> getConsumerProps() {
    // ...
}
```

XML

```
<citrus-kafka:endpoint id="helloKafkaEndpoint"
                                consumer-properties="consumerProps"
                                producer-properties="producerProps"/>


<util:map id="producerProps">
  <entry key="bootstrap.servers" value="localhost:9093,localhost:9094"/>
  <entry key="retries" value="10" value-type="java.lang.Integer"/>
  <entry key="max.request.size" value="1024" value-type="java.lang.Integer"/>
  <entry key="ssl.keystore.location" value="/path/to/keystore.jks"/>
  <entry key="ssl.kestore.password" value="secr3t"/>
</util:map>

<util:map id="consumerProps">
  <entry key="bootstrap.servers" value="localhost:9093,localhost:9094"/>
  <entry key="session.timeout.ms" value="10000" value-type="java.lang.Integer"/>
  <entry key="enable.auto.commit" value="true" value-type="java.lang.Boolean"/>
  <entry key="ssl.truststore.location" value="/path/to/truststore.jks"/>
  <entry key="ssl.truststore.password" value="secr3t"/>
</util:map>
```

13.2. Kafka Synchronous Endpoints

Not implemented yet.

13.3. Kafka Message Headers

The Kafka Citrus integration defines a set of special message header entries that are either used to

manipulate the endpoint behavior or as validation object. These Kafka specific headers are stored with a header key prefix `citrus_kafka_*`. You can set or verify those headers in send and receive actions as follows:

Java

```
send(helloKafkaEndpoint)
    .message()
    .header(KafkaMessageHeaders.TOPIC, "my.very.special.topic")
    .header(KafkaMessageHeaders.MESSAGE_KEY, "myKey")
    .header(KafkaMessageHeaders.PARTITION, 1);
```

XML

```
<header>
    <element name="citrus_kafka_topic" value="my.very.special.topic"/>
    <element name="citrus_kafka_messageKey" value="myKey"/>
    <element name="citrus_kafka_partition" value="1" />
</header>
```

The header entries above are used in a send operation in order to overwrite the topic destination, to set the record key and to specify the target partition of the producer record. These settings do only apply for the very specific send operation. Default values on the Kafka endpoint are overwritten respectively.

Typing of message header entries may also be of interest in order to meet the Kafka standards. For instance the following message key is of type `java.lang.Integer` and is therefore transferred via Kafka's key-serializer as an integer value. You need to set the header type to `integer` and use a `org.apache.kafka.common.serialization.IntegerSerializer` as key-serializer on the Kafka endpoint configuration.

Java

```
send(helloKafkaEndpoint)
    .message()
    .header(KafkaMessageHeaders.MESSAGE_KEY, 1L);
```

XML

```
<header>
    <element name="citrus_kafka_messageKey" value="1" type="integer"/>
</header>
```

In case of a receiving operation message headers are valuable validation objects that can be used to verify the message content with an expected behavior.

Java

```
receive(helloKafkaEndpoint)
    .message()
    .header(KafkaMessageHeaders.TIMESTAMP, Matchers.greaterThan(0))
    .header(KafkaMessageHeaders.TOPIC, "my.expected.topic")
    .header(KafkaMessageHeaders.MESSAGE_KEY, "myKey")
    .header(KafkaMessageHeaders.PARTITION, 1)
    .header(KafkaMessageHeaders.OFFSET, Matchers.greaterThanOrEqualTo(0));
```

XML

```
<header>
    <element name="citrus_kafka_timestamp" value="@assertThat(greaterThan(0))@"/>
    <element name="citrus_kafka_topic" value="my.expected.topic"/>
    <element name="citrus_kafka_messageKey" value="myKey"/>
    <element name="citrus_kafka_partition" value="1"/>
    <element name="citrus_kafka_offset"
value="@assertThat(greaterThanOrEqualTo(0))@"/>
</header>
```

These are the available Kafka message headers in Citrus:

Header	Name	Type	Description
KafkaMessageHeaders.TIMESTAMP	citrus_kafka_timestamp	java.lang.Long	Record timestamp value
KafkaMessageHeaders.TOPIC	citrus_kafka_topic	java.lang.String	Topic name
KafkaMessageHeaders.MESSAGE_KEY	citrus_kafka_messageKey	java.lang.Object	Record key
KafkaMessageHeaders.PARTITION	citrus_kafka_partition	java.lang.Integer	Topic partition id
KafkaMessageHeaders.OFFSET	citrus_kafka_offset	java.lang.Long	Record offset on partition

13.4. Kafka Message

Citrus also provides a Kafka message implementation that you can use on any send and receive operation. This enables you to set special message headers in a more comfortable way when using the Java fluent API:

Use message objects

```
send(helloKafkaEndpoint)
    .message(new KafkaMessage("sayHello")
                    .topic("my.very.special.topic")
                    .messageKey("myKey")
                    .partition(1));
```

The message implementation provides fluent API builder methods for each Kafka specific header.

Additionally, when receiving messages, you might want to use Kafka message selectors.

13.5. Kafka Message Selector

The Kafka Message Selector feature allows you to selectively receive messages from a Kafka topic based on specific criteria. This powerful functionality enables you to filter Kafka messages by different criteria, e.g. based on headers. Additionally, the defined time window for message retrieval significantly improves the performance. Imagine a large Kafka topic with thousands of events. Looking through all of these would require an immense amount of resources and time. Instead, selective message consumption starts at an offset `Ox = OT-n`. Where `T` is the current timestamp and `n` is the maximum timespan in which the wanted event is expected to have been published.

13.5.1. Basic Usage

The Kafka Message Selector can be used in various ways, depending on your preferred syntax and test framework.

Java

```
then(
    receive(kafkaEndpoint)
        .selector(
            kafkaMessageFilter()
                .eventLookbackWindow(Duration.ofSeconds(1L))
                .kafkaMessageSelector(kafkaHeaderEquals("key", "value"))
                .build()
        )
);
```

Java 2

```
then(
    kafkaEndpoint.findKafkaEventHeaderEquals(Duration.ofSeconds(1L), "key", "value")
);
```

XML

```
<receive endpoint="helloKafkaEndpoint">
    <description>Receive selective Kafka message</description>
    <selector>
      <element name="header-filter-key" value="key"/>
      <element name="header-filter-value" value="value"/>
      <element name="event-lookback-window" value="PT1S"/>
    </selector>
</receive>
```

13.5.2. Configuration

Java DSL	XML DSL	Description
`eventLookbackWindow`	`event-lookback-window`	This defines how far back in time the selector should search for messages. When using XML configuration, the event lookback window must be specified as an ISO-8601 duration string. For example, `PT1S` represents a duration of 1 second.
`kafkaMessageSelector`	See <<kafka-message-selector-types,Selector Types	This specifies the criteria for message selection. In the examples, we're using `kafkaHeaderEquals("key", "value")`, which selects messages where a header with the `key` "key" exactly matches the `value` "value".
`pollTimeout`	`poll-timeout`	The timeout duration for each poll operation when consuming messages from Kafka. This value determines how long the consumer will wait for new records in each poll cycle. It is not the overall receive action timeout! When using XML configuration, the poll timeout must be specified as an ISO-8601 duration string. For example, `PT0.100S` represents a duration of 1 millisecond.

13.5.3. Selector Types

Message Header

The framework provides two main types of message header selectors. From within the Java DSL, these two can be easily invoked using statically provided methods:

1. `kafkaHeaderEquals`: Matches messages where the specified header `key` exactly equals the given `value`.
2. `kafkaHeaderContains`: Matches messages where the specified header `key` contains the given `value` as a substring.

More advanced users might want to do pre- or suffix matching. That is also possible.

Java

```
then(
    receive(kafkaWithRandomConsumerGroupEndpoint)
        .selector(
            kafkaMessageFilter()
                .eventLookbackWindow(Duration.ofSeconds(1L))
                .kafkaMessageSelector(
                    KafkaMessageByHeaderSelector.builder()
                        .key("key")
                        .value("prefix")
                        .matchingMechanism(STARTS_WITH)
                        .build()
                )
                .build()
        )
);
```

Note that if the specified `key` is `null`, all headers in the record will be matched against the `value`. If the `value` is `null` however, all headers with the exact `key` match.

Java DSL	XML DSL	Description
`key`	`header-filter-key`	Key-filter being applied to Kafka messages. Matches exact if specified, all keys if `null` or empty.
`value`	`header-filter-value`	Value-filter being applied to Kafka messages. Matches all values if `null` or empty. Otherwise matches as specified by strategy.

Java DSL	XML DSL	Description
`valueMatchingStrategy`	`header-filter-comparator`	Specifies how the `value` is being matched. Must be one of `EQUALS`, `CONTAINS`, `STARTS_WITH` or `ENDS_WITH`. It defaults to `EQUALS`, if not specified.

13.5.4. Best Practices

Set Appropriate Lookback Window: Choose a lookback window that balances between finding the desired message and performance. A larger window might find older messages but could impact performance.

Combine with Other Citrus Features: The Kafka Message Selector can be combined with other Citrus testing features for comprehensive Kafka integration testing.

13.6. Dynamic Kafka Endpoints

As we have seen before the topic name can be overwritten in each send and receive operation by specifying the `citrus_kafka_topic` message header. In addition to that you can make use of completely dynamic Kafka endpoints, too.

The dynamic endpoint is created on the fly with respective settings. So you can use the `kafka` endpoint component in your test as follows:

Java

```
send("kafka:hello")
    .message()
    .body("foo")
    .header(KafkaMessageHeaders.MESSAGE_KEY, 1);
```

XML

```
<send endpoint="kafka:hello">
    <message>
        ...
    </message>
    <header>
        <element name="citrus_kafka_messageKey" value="1"/>
    </header>
</send>
```

This action above will create a dynamic Kafka endpoint and publish the message to the `hello` topic. The dynamic endpoint url uses the `kafka:` scheme and gives the topic name as resource path. In addition to that the dynamic endpoint url is able to set multiple parameters such as `server`. Let's have a look at this in a small example.

Java

```
send("kafka:hello?server=localhost:9091")
    .message(new KafkaMessage("foo"));
```

XML

```
<send endpoint="kafka:hello?server=localhost:9091">
    <message>
        ...
    </message>
</send>
```

You can add multiple parameters to the endpoint url in order to set properties on the dynamic endpoint. You can read more about dynamic endpoints in chapter dynamic endpoints.

13.7. Embedded Kafka Server

The Kafka message broker is composed of a Zookeeper server and a Kafka server. Citrus provides an embedded server (**for testing purpose only!**) that is able to start within your integration test environment. The server cluster is configured with one single Zookeeper server and a single Kafka server. You can define server ports and broker properties such as topics, number of partitions and broker ids. Given topics are automatically added via admin client on the Kafka server with given amount of partitions.

You can add the embedded server component to the Spring application context as normal Spring bean. The server will automatically start and stop within the application context lifecycle. The Zookeeper log directory is located in the Java temp directory and is automatically deleted on JVM exit.

See the following configuration how to use the embedded server component:

Java

```
@Bean
public EmbeddedKafkaServer kafkaServer() {
    return new EmbeddedKafkaServerBuilder()
                    .topics("foo", "bar")
                    .kafkaServerPort(9091)
                    .build();
}
```

XML

```
<citrus-kafka:embedded-server id="kafkaServer"
                              topics="foo,bar"
                              kafka-server-port="9091"/>
```

The embedded server component provides following properties to set:

Name	Type	Description
topics	java.lang.String	Comma delimited list of topic names that automatically will be created on the server.
kafka-server-port	java.lang.Integer	Port of the embedded Kafka server
zookeeper-port	java.lang.Integer	Zookeeper server port. By default, a random port is used.
broker-properties	java.util.Map	Map of broker property key-value pairs that overwrite the default broker properties. For a list of available properties please review the official Kafka documentation.
partitions	java.lang.Integer	Number of partitions to create for each topic
log-dir-path	java.lang.String	Path to Zookeeper log directory. The Zookeeper server will create its data directory in this directory. By default, the Java temp directory is used.
auto-delete-logs	java.lang.Boolean	Auto delete Zookeeper log directories on exit. Default is true.

Chapter 14. Http REST support

REST APIs have gained more and more significance regarding client-server interfaces with message exchange over Http. Http is a synchronous protocol by nature so the client blocks in order to wait for the server response synchronously. Citrus is able to connect with Http services and REST APIs on both client and server side with a powerful JSON message data support.

In the next sections you will learn how to invoke Http services as a client and how to handle REST Http requests in a test case. The chapter deals with setting up an Http server in order to accept client requests and provide proper Http responses with GET, PUT, DELETE or POST request method.

The http components in Citrus are kept in a separate Maven module. So you should add the module as Maven dependency to your project accordingly.

Http module dependency

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-http</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

As Citrus provides a customized Http configuration schema for the Spring application context configuration files. Simply include the http-config namespace in the configuration XML files as follows.

Spring bean definition namespace

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:citrus="http://www.citrusframework.org/schema/config"
   xmlns:citrus-http="http://www.citrusframework.org/schema/http/config"
   xsi:schemaLocation="
   http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.citrusframework.org/schema/config
   http://www.citrusframework.org/schema/config/citrus-config.xsd
   http://www.citrusframework.org/schema/http/config
   http://www.citrusframework.org/schema/http/config/citrus-http-config.xsd">

    [...]

</beans>
```

Now we are ready to use the customized Citrus Http configuration elements with the citrus-http namespace prefix.

14.1. Http REST client

On the client side Citrus uses a simple Http message client component connecting to the server. The **request-url** attribute defines the Http server endpoint URL to connect to. You can reference this client in your test case in order to send and receive messages. Citrus as client waits for the response message from the server. After that the response message goes through the validation process as usual.

Java

```
@Bean
public HttpClient httpClient() {
    return new HttpClientBuilder()
        .requestUrl("http://localhost:8080/hello")
        .requestMethod(HttpMethod.GET)
        .contentType("application/xml")
        .charset("UTF-8")
        .timeout(60000L)
        .build();
}
```

XML

```
<citrus-http:client id="httpClient"
          request-url="http://localhost:8080/hello"
          request-method="GET"
          content-type="application/xml"
          charset="UTF-8"
          timeout="60000"/>
```

The **request-method** defines the Http method to use. In addition to that we can specify the content-type of the request we are about to send. The charset is also added to the content-type header. In case you do not want to set the charset at all please specify an empty string as the default value is *UTF-8*.

The client builds the Http request and sends it to the Http server. While the client is waiting for the synchronous Http response to arrive we are able to poll several times for the response message in our test case. As usual you can use the same client endpoint in your test case to send and receive messages synchronously. In case the reply message comes in too late according to the timeout settings a respective timeout error is raised.

Http defines several request methods that a client can use to access Http server resources. In the example client above we are using **GET** as default request method. Of course you can overwrite this setting in a test case action by setting the Http request method inside the sending test action. The Http client component can be used as normal endpoint in a sending test action. Use something like this in your test:

Java

```
send("httpClient")
    .message()
    .body("Hello HttpServer")
    .header(HttpMessageHeaders.HTTP_REQUEST_METHOD, HttpMethod.POST.name())
```

XML

```
<send endpoint="httpClient">
    <message>
        <payload>
            <TestMessage>
                <Text>Hello HttpServer</Text>
            </TestMessage>
        </payload>
    </message>
    <header>
        <element name="citrus_http_method" value="POST"/>
    </header>
</send>
```

Citrus uses the Spring REST template mechanism for sending out Http requests. This means you have great customizing opportunities with a special REST template configuration. You can think of basic Http authentication, read timeouts and special message factory implementations. Just use the custom REST template attribute in client configuration like this:

Java

```
@Bean
public HttpClient httpClient() {
    return new HttpClientBuilder()
        .requestUrl("http://localhost:8080/hello")
        .requestMethod(HttpMethod.GET)
        .contentType("text/plain")
        .restTemplate(customizedRestTemplate())
        .timeout(60000L)
        .build();
}

@Bean
public RestTemplate customizedRestTemplate() {
    RestTemplate restTemplate = new RestTemplate();

    StringHttpMessageConverter converter = new StringHttpMessageConverter();
    converter.setSupportedMediaTypes(Collections.singletonList("text/plain"));

    restTemplate.setMessageConverters(Collections.singletonList(converter));

    restTemplate.setErrorHandler(customErrorHandler());

    HttpComponentsClientHttpRequestFactory requestFactory = new
HttpComponentsClientHttpRequestFactory();
    requestFactory.setReadTimeout(9000L);

    restTemplate.setRequestFactory(requestFactory);

    return restTemplate;
}
```

XML

```
<citrus-http:client id="httpClient"
                    request-url="http://localhost:8080/hello"
                    request-method="GET"
                    content-type="text/plain"
                    rest-template="customizedRestTemplate"/>

<!-- Customized rest template -->
<bean name="customizedRestTemplate"
class="org.springframework.web.client.RestTemplate">
  <property name="messageConverters">
    <util:list id="converter">
      <bean class="org.springframework.http.converter.StringHttpMessageConverter">
        <property name="supportedMediaTypes">
          <util:list id="types">
            <value>text/plain</value>
          </util:list>
        </property>
      </bean>
    </util:list>
  </property>
  <property name="errorHandler">
    <!-- Custom error handler -->
    <ref bean ="customErrorHandler"/>
  </property>
  <property name="requestFactory">
    <bean
class="org.springframework.http.client.HttpComponentsClientHttpRequestFactory">
      <property name="readTimeout" value="9000"/>
    </bean>
  </property>
</bean>
```

Up to now we have used a generic **send** test action to send Http requests as a client. This is completely valid strategy as the Citrus Http client is a normal message endpoint.

In order to simplify the Http usage in a test case Citrus also provides special test action implementations for Http.

These Http specific actions are located in a separate XML namespace. In case you are writing XML test cases you need to add this namespace to our test case XML first.

Add Citrus Http action namespace

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:http="http://www.citrusframework.org/schema/http/testcase"
        xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.citrusframework.org/schema/http/testcase
        http://www.citrusframework.org/schema/http/testcase/citrus-http-testcase.xsd">

      [...]

    </beans>
```

The test case is now ready to use the specific Http test actions.

Java

```
http().client("httpClient")
        .send()
        .post("/customer")
        .message()
        .body("<customer>" +
                "<id>citrus:randomNumber()</id>" +
                "<name>testuser</name>" +
              "</customer>")
        .contentType(MediaType.APPLICATION_XML_VALUE)
        .accept(MediaType.APPLICATION_XML_VALUE)
        .header("X-CustomHeaderId", "${custom_header_id}");
```

XML

```
<http:send-request client="httpClient">
  <http:POST path="/customer">
    <http:headers content-type="application/xml" accept="application/xml">
      <http:header name="X-CustomHeaderId" value="${custom_header_id}"/>
    </http:headers>
    <http:body>
      <http:data>
        <![CDATA[
          <customer>
            <id>citrus:randomNumber()</id>
            <name>testuser</name>
          </customer>
        ]]>
      </http:data>
    </http:body>
  </http:POST>
</http:send-request>
```

The action above uses several Http specific settings such as the request method **POST** as well as the **content-type** and **accept** headers. As usual the send action needs a target Http client endpoint component. We can specify a request **path** attribute that added as relative path to the base uri used on the client.

When using a **GET** request we can specify some request uri parameters.

Java

```
http().client("httpClient")
        .send()
        .get("/customer/${custom_header_id}")
        .message()
        .contentType(MediaType.APPLICATION_XML_VALUE)
        .accept(MediaType.APPLICATION_XML_VALUE)
        .queryParam("type", "active");
```

XML

```
<http:send-request client="httpClient">
  <http:GET path="/customer/${custom_header_id}">
    <http:params content-type="application/xml" accept="application/xml">
      <http:param name="type" value="active"/>
    </http:params>
  </http:GET>
</http:send-request>
```

The send action above uses a **GET** request on the endpoint uri `http://localhost:8080/customer/1234?type=active`.

Of course when sending Http client requests we are also interested in receiving Http response messages. We want to validate the success response with Http status code.

Java

```
http().client("httpClient")
        .receive()
        .response(HttpStatus.OK)
        .message()
        .body("<customer>" +
                "<id>citrus:randomNumber()</id>" +
                "<name>testuser</name>" +
              "</customer>")
        .header("X-CustomHeaderId", "${custom_header_id}");
```

XML

```
<http:receive-response client="httpClient">
  <http:headers status="200" reason-phrase="OK" version="HTTP/1.1">
    <http:header name="X-CustomHeaderId" value="${custom_header_id}"/>
  </http:headers>
  <http:body>
    <http:data>
      <![CDATA[
          <customerResponse>
            <success>true</success>
          </customerResponse>
      ]]>
    </http:data>
  </http:body>
</http:receive-response>
```

The **receive-response** test action also uses a client component. We can expect response status code information such as **status** and **reason-phrase** . Of course Citrus will raise a validation exception in case Http status codes mismatch.

By default, the client component will add the **Accept** http header and set its value to a list of all supported encodings on the host operating system. This list can get quite big so you may want to not set this default accept header. The setting is done in the Spring RestTemplate:

Java

```
@Bean
public RestTemplate customizedRestTemplate() {
    RestTemplate restTemplate = new RestTemplate();

    StringHttpMessageConverter converter = new StringHttpMessageConverter();
    converter.setWriteAcceptCharset(false);

    restTemplate.setMessageConverters(Collections.singletonList(converter));

    return restTemplate;
}
```

XML

```
<bean name="customizedRestTemplate"
class="org.springframework.web.client.RestTemplate">
    <property name="messageConverters">
        <util:list id="converter">
            <bean
class="org.springframework.http.converter.StringHttpMessageConverter">
                <property name="writeAcceptCharset" value="false"/>
            </bean>
        </util:list>
    </property>
</bean>
```

Add this custom RestTemplate configuration and set it to the client component with **rest-template** property. Fortunately the Citrus client component provides a separate setting **default-accept-header** which is a Boolean setting. By default, this setting is set to **true** so the default accept header is automatically added to all requests. If you set this flag to **false** the header is not set:

Java

```
@Bean
public HttpClient httpClient() {
    return new HttpClientBuilder()
        .requestUrl("http://localhost:8080/hello")
        .requestMethod(HttpMethod.GET)
        .contentType("text/plain")
        .defaultAcceptHeader(false)
        .timeout(60000L)
        .build();
}
```

XML

```
<citrus-http:client id="httpClient"
                    request-url="http://localhost:8080/hello"
                    request-method="GET"
                    content-type="text/plain"
                    default-accept-header="false"/>
```

Of course, you can set the **Accept** header on each send operation in order to tell the server what kind of content types are supported in response messages.

Now we can send and receive messages as Http client with specific test actions. Now let's move on to the Http server.

14.2. Http client interceptors

The client component is able to add custom interceptors that participate in the request/response processing. The interceptors need to implement the common interface **org.springframework.http.client.ClientHttpRequestInterceptor**.

Java

```
@Bean
public HttpClient httpClient() {
    return new HttpClientBuilder()
        .requestUrl("http://localhost:8080/hello")
        .requestMethod(HttpMethod.GET)
        .interceptor(new LoggingClientInterceptor())
        .build();
}
```

XML

```
<citrus-http:client id="httpClient"
                   request-url="http://localhost:8080/hello"
                   request-method="GET"
                   interceptors="clientInterceptors"/>

<util:list id="clientInterceptors">
  <bean class="org.citrusframework.http.interceptor.LoggingClientInterceptor"/>
</util:list>
```

The sample above adds the Citrus logging client interceptor that logs requests and responses exchanged with that client component. You can add custom interceptor implementations here in order to participate in the request/response message processing.

14.3. Http REST server

Receiving Http requests requires an Http server listening on a port on your local machine. Citrus offers an embedded Http server which is capable of handling incoming Http requests. The server accepts client connections and must provide a proper Http response. In the next section you will see how to simulate server side Http REST service with Citrus.

Java

```
@Bean
public HttpServer httpServer() {
    return new HttpServerBuilder()
        .port(8080)
        .autoStart(true)
        .build();
}
```

XML

```
<citrus-http:server id="httpServer"
                port="8080"
                auto-start="true"/>
```

Citrus uses an embedded Jetty server that will automatically start when the Citrus context is loaded (auto-start="true"). The basic connector is listening on port **8080** for requests. Test cases can interact with this server instance via message channels by default. The server provides an inbound channel that holds incoming request messages. The test case can receive those requests from the channel with a normal receive test action. In a second step the test case can provide a synchronous response message as reply which will be automatically sent back to the Http client as response.

The figure above shows the basic setup with inbound channel and reply channel. You as a tester should not worry about this too much. By default, you as a tester just use the server as synchronous endpoint in your test case. This means that you simply receive a message from the server and send a response back.

Java

```
receive("httpServer")
    .message()
    .body("...");

send("httpServer")
    .message()
    .body("...")
    .header(HttpMessageHeaders.HTTP_STATUS_CODE, 200);
```

XML

```
<receive endpoint="httpServer">
    <message>
        <data>
          [...]
        </data>
    </message>
</receive>

<send endpoint="httpServer">
    <message>
        <data>
          [...]
        </data>
    </message>
</send>
```

As you can see we reference the server id in both receive and send actions. The Citrus server instance will automatically send the response back to the calling Http client. In most cases this is exactly what we want to do - send back a response message that is specified inside the test. The Http server component by default uses a channel endpoint adapter in order to forward all incoming requests to an in-memory message channel. This is done completely behind the scenes. The Http server component provides some more customization possibilities when it comes to endpoint adapter implementations. This topic is discussed in a separate section endpoint-adapter. Up to now we keep it simple by synchronously receiving and sending messages in the test case.

> The default channel endpoint adapter automatically creates an inbound message channel where incoming messages are stored too internally. So if you need to clean up a server that has already stored some incoming messages you can do this easily by purging the internal message channel. The message channel follows a naming convention **{serverName}.inbound** where **{serverName}** is the Spring bean name of the Citrus server endpoint component. If you purge this internal channel in a before test nature you are sure that obsolete messages on a server instance get purged before each test is executed.

So let's get back to our mission of providing response messages as server to connected clients. As

you might know Http REST works with some characteristic properties when it comes to send and receive messages. For instance a client can send different request methods GET, POST, PUT, DELETE, HEAD and so on. The Citrus server may verify this method when receiving client requests. Therefore we have introduced special Http test actions for server communication. Have a look at a simple example:

Java

```
http().server("httpServer")
    .receive()
    .post("/test")
    .message()
    .contentType("application/xml")
    .accept("application/xml")
    .body("<testRequestMessage>" +
            "<text>Hello HttpServer</text>" +
          "</testRequestMessage>")
    .header("Authorization", "Basic c29tZVVzZXJuYW1lOnNvbWVQYXNzd29yZA==")
    .header("X-CustomHeaderId", "${custom_header_id}")
    .extract(fromHeaders()
                .header("X-MessageId", "message_id"));

http().server("httpServer")
    .send()
    .response(HttpStatus.OK)
    .message()
    .contentType("application/xml")
    .body("<testResponseMessage>" +
            "<text>Hello Citrus</text>" +
          "</testResponseMessage>")
    .header("X-CustomHeaderId", "${custom_header_id}")
    .header("X-MessageId", "${message_id}");
```

XML

```
<http:receive-request server="helloHttpServer">
  <http:POST path="/test">
    <http:headers content-type="application/xml" accept="application/xml, */*">
      <http:header name="X-CustomHeaderId" value="${custom_header_id}"/>
      <http:header name="Authorization" value="Basic
c29tZVVzZXJuYW1lOnNvbWVQYXNzd29yZA==="/>
    </http:headers>
    <http:body>
    <http:data>
      <![CDATA[
        <testRequestMessage>
          <text>Hello HttpServer</text>
        </testRequestMessage>
      ]]>
    </http:data>
    </http:body>
  </http:POST>
  <http:extract>
    <http:header name="X-MessageId" variable="message_id"/>
  </http:extract>
</http:receive-request>

<http:send-response server="helloHttpServer">
  <http:headers status="200" reason-phrase="OK" version="HTTP/1.1">
    <http:header name="X-MessageId" value="${message_id}"/>
    <http:header name="X-CustomHeaderId" value="${custom_header_id}"/>
    <http:header name="Content-Type" value="application/xml"/>
  </http:headers>
  <http:body>
  <http:data>
    <![CDATA[
      <testResponseMessage>
        <text>Hello Citrus</text>
      </testResponseMessage>
    ]]>
  </http:data>
  </http:body>
</http:send-response>
```

We receive a client request and validate that the request method is **POST** on request path **/test** . Now we can validate special message headers such as **content-type** . In addition to that we can check custom headers and basic authorization headers. As usual the optional message body is compared to an expected message template. The custom **X-MessageId** header is saved to a test variable **message_id** for later usage in the response.

The response message defines Http typical entities such as **status** and **reason-phrase**. Here the tester can simulate **404 NOT_FOUND** errors or similar other status codes that get sent back to the client. In our example everything is **OK** and we send back a response body and some custom

header entries.

That is basically how Citrus simulates Http server operations. We receive the client request and validate the request properties. Then we send back a response with an Http status code.

This completes the server actions on Http message transport. Now we continue with some more Http specific settings and features.

14.4. Http headers

When dealing with Http request/response communication we always deal with Http specific headers. The Http protocol defines a group of header attributes that both client and server need to be able to handle. You can set and validate these Http headers in Citrus quite easy. Let us have a look at a client operation in Citrus where some Http headers are explicitly set before the request is sent out.

Java

```
http().client("httpClient")
    .send()
    .post()
    .message()
    .contentType("application/xml")
    .accept("application/xml")
    .body("<testRequestMessage>" +
            "<text>Hello HttpServer</text>" +
          "</testRequestMessage>")
    .header("X-CustomHeaderId", "${custom_header_id}");
```

XML

```
<http:send-request client="httpClient">
  <http:POST>
    <http:headers>
        <http:header name="X-CustomHeaderId" value="${custom_header_id}"/>
        <http:header name="Content-Type" value="application/xml"/>
        <http:header name="Accept" value="application/xml"/>
    </http:headers>
    <http:body>
        <http:payload>
            <testRequestMessage>
                <text>Hello HttpServer</text>
            </testRequestMessage>
        </http:payload>
    </http:body>
  </http:POST>
</http:send-request>
```

We are able to set custom headers (**X-CustomHeaderId**) that go directly into the Http header

section of the request. In addition to that testers can explicitly set Http reserved headers such as **Content-Type** . Fortunately you do not have to set all headers on your own. Citrus will automatically set the required Http headers for the request. So we have the following Http request which is sent to the server:

Sample Http request

```
POST /test HTTP/1.1
Accept: application/xml
Content-Type: application/xml
X-CustomHeaderId: 123456789
Accept-Charset: macroman
User-Agent: Jakarta Commons-HttpClient/3.1
Host: localhost:8091
Content-Length: 175
<testRequestMessage>
    <text>Hello HttpServer</text>
</testRequestMessage>
```

On server side testers are interested in validating the Http headers. Within Citrus receive action you simply define the expected header entries. The Http specific headers are automatically available for validation as you can see in this example:

Java

```
http().server("httpServer")
    .receive()
    .post()
    .message()
    .contentType("application/xml")
    .accept("application/xml")
    .body("<testRequestMessage>" +
            "<text>Hello HttpServer</text>" +
          "</testRequestMessage>")
    .header("X-CustomHeaderId", "${custom_header_id}");
```

XML

```
<http:receive-request server="httpServer">
  <http:POST>
    <http:headers>
        <http:header name="X-CustomHeaderId" value="${custom_header_id}"/>
        <http:header name="Content-Type" value="application/xml"/>
        <http:header name="Accept" value="application/xml"/>
    </http:headers>
    <http:body>
        <http:payload>
            <testRequestMessage>
                <text>Hello HttpServer</text>
            </testRequestMessage>
        </http:payload>
    </http:body>
  </http:POST>
</http:receive-request>
```

The test checks on custom headers and Http specific headers to meet the expected values.

Now that we have accepted the client request and validated the contents we are able to send back a proper Http response message. Same thing here with Http specific headers. The Http protocol defines several headers marking the success or failure of the server operation. In the test case you can set those headers for the response message with conventional Citrus header names. See the following example to find out how that works for you.

Java

```
http().server("httpServer")
    .send()
    .response(HttpStatus.OK)
    .message()
    .contentType("application/xml")
    .body("<testResponseMessage>" +
            "<text>Hello Citrus</text>" +
          "</testResponseMessage>")
    .header("X-CustomHeaderId", "${custom_header_id}");
```

XML

```
<http:send-response server="httpServer">
    <http:headers status="200" reason-phrase="OK">
        <http:header name="X-CustomHeaderId" value="${custom_header_id}"/>
        <http:header name="Content-Type" value="application/xml"/>
    </http:headers>
    <http:body>
        <http:payload>
            <testResponseMessage>
                <text>Hello Citrus</text>
            </testResponseMessage>
        </http:payload>
    </http:body>
</http:send-response>
```

Once more we set the custom header entry (**X-CustomHeaderId**) and an Http reserved header (**Content-Type**) for the response message. On top of this we are able to set the response status for the Http response. We use the reserved header names **status** in order to mark the success of the server operation. With this mechanism we can easily simulate different server behaviour such as Http error response codes (e.g. 404 - Not found, 500 - Internal error). Let us have a closer look at the generated response message:

Sample Http response

```
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8
Accept-Charset: macroman
Content-Length: 205
Server: Jetty(7.0.0.pre5)
<testResponseMessage>
    <text>Hello Citrus Client</text>
</testResponseMessage>
```

You do not have to set the reason phrase all the time. It is sufficient to only set the Http status code. Citrus will automatically add the proper reason phrase for well known Http status codes.

The only thing that is missing right now is the validation of Http status codes when receiving the server response in a Citrus test case. It is very easy as you can use the Citrus reserved header names for validation, too.

Java

```
http().client("httpClient")
    .receive()
    .response(HttpStatus.OK)
    .message()
    .contentType("application/xml")
    .body("<testResponseMessage>" +
            "<text>Hello Citrus</text>" +
          "</testResponseMessage>")
    .header("X-CustomHeaderId", "${custom_header_id}");
```

XML

```
<http:receive-response client="httpClient">
    <http:headers status="200" reason-phrase="OK" version="HTTP/1.1">
        <http:header name="X-CustomHeaderId" value="${custom_header_id}"/>
    </http:headers>
    <http:body>
        <http:payload>
            <testResponseMessage>
                <text>Hello Citrus</text>
            </testResponseMessage>
        </http:payload>
    </http:body>
</http:receive-response>
```

Be aware of the slight differences in request URI and context path. The context path gives you the web application context path within the servlet container for your web application. The request URI always gives you the complete path that was called for this request.

As you can see we are able to validate all parts of the initial request endpoint URI the client was calling. This completes the Http header processing within Citrus. On both client and server side Citrus is able to set and validate Http specific header entries which is essential for simulating Http communication.

14.5. Http query parameter

Up to now we have used some of the basic Citrus reserved Http header names (status, version, reason-phrase). In Http RESTful services some other header names are essential for validation. These are request attributes like query parameters, context path and request URI. The Citrus server side REST message controller will automatically add all this information to the message header for you. So all you need to do is validate the header entries in your test.

The next example receives an Http GET method request on server side. Here the GET request does not have any message payload, so the validation just works on the information given in the message header. We assume the client to call `http://localhost:8080/app/users?id=123456789`. As a

tester we need to validate the request method, request URI, context path and the query parameters.

Java

```
http()
    .server(httpServer)
    .receive()
    .get("/api/users")
    .message()
    .queryParam("id", "123456789")
    .contentType(MediaType.APPLICATION_XML_VALUE)
    .accept(MediaType.APPLICATION_XML_VALUE)
    .header("Host", "localhost:8080");
```

XML

```
<http:receive-request server="httpServer">
  <http:GET path="/api/users" context-path="/app">
    <http:params>
        <http:param name="id" value="123456789"/>
    </http:params>
    <http:headers content-type="application/xml" accept="application/xml">
        <http:header name="Host" value="localhost:8080"/>
    </http:headers>
    <http:body>
        <http:data></http:data>
    </http:body>
  </http:GET>
</http:receive-request>
```

The http server is able to validate incoming Http query parameters. You can add as many parameters as you would like to verify. Each parameter value is able to use test variables and validation matcher expressions as usual.

On the client side we are able to send query parameters in our request.

Java

```
http()
    .client(httpClient)
    .send()
    .get("/api/users")
    .message()
    .queryParam("id", "123456789")
    .contentType(MediaType.APPLICATION_XML_VALUE)
    .accept(MediaType.APPLICATION_XML_VALUE);
```

XML

```
<http:send-request client="httpClient">
  <http:GET path="/app/users">
    <http:params>
        <http:param name="id" value="123456789"/>
    </http:params>
    <http:headers content-type="application/xml" accept="application/xml"/>
    <http:body>
        <http:data></http:data>
    </http:body>
  </http:GET>
</http:send-request>
```

The parameters are automatically added to the request URL that is configured on the `httpClient` component.

14.6. Http server interceptors

The server component is able to add custom interceptors that participate in the request/response processing. The interceptors need to implement the common interface **org.springframework.web.servlet.HandlerInterceptor**.

Java

```
@Bean
public HttpServer httpServer() {
    return new HttpServerBuilder()
        .port(8080)
        .autoStart(true)
        .interceptor(new LoggingHandlerInterceptor())
        .build();
}
```

XML

```
<citrus-http:server id="httpServer"
                   port="8080"
                   auto-start="true"
                   interceptors="serverInterceptors"/>

<util:list id="serverInterceptors">
  <bean class="org.citrusframework.http.interceptor.LoggingHandlerInterceptor"/>
</util:list>
```

The sample above adds the Citrus logging handler interceptor that logs requests and responses exchanged with that server component. You can add custom interceptor implementations here in order to participate in the request/response message processing.

14.7. Http form urlencoded data

HTML form data can be sent to the server using different methods and content types. One of them is a POST method with **x-www-form-urlencoded** body content. The form data elements are sent to the server using key-value pairs POST data where the form control name is the key and the control data is the url encoded value.

Form urlencoded form data content could look like this:

Form urlencoded data

```
password=s%21cr%21t&username=foo
```

As you can see the form data is automatically encoded. In the example above we transmit two form controls **password** and **username** with respective values **scrt** and **foo** . In case we would validate this form data in Citrus we are able to do this with plaintext message validation.

Java

```
receive("httpServer")
    .message()
    .type(MessageType.PLAINTEXT)
    .body("password=s%21cr%21t&username=${username}")
    .header(HttpMessageHeaders.HTTP_REQUEST_METHOD, HttpMethod.POST.name())
    .header(HttpMessageHeaders.HTTP_REQUEST_URI, "/form-test")
    .header(HttpMessageHeaders.HTTP_CONTENT_TYPE, "application/x-www-form-
urlencoded");

send("httpServer")
    .message()
    .header(HttpMessageHeaders.HTTP_STATUS_CODE, 200);
```

XML

```
<receive endpoint="httpServer">
  <message type="plaintext">
    <data>
      <![CDATA[
        password=s%21cr%21t&username=${username}
      ]]>
    </data>
  </message>
  <header>
    <element name="citrus_http_method" value="POST"/>
    <element name="citrus_http_request_uri" value="/form-test"/>
    <element name="Content-Type" value="application/x-www-form-urlencoded"/>
  </header>
</receive>

<send endpoint="httpServer">
  <message>
    <data></data>
  </message>
  <header>
    <element name="citrus_http_status_code" value="200"/>
  </header>
</send>
```

Obviously validating these key-value pair character sequences can be hard especially when having HTML forms with lots of form controls. This is why Citrus provides a special message validator for **x-www-form-urlencoded** contents. First of all we have to add **citrus-http** module as dependency to our project if we have not done so yet. After that we can add the validator implementation to the list of message validators used in Citrus.

Java

```
@Bean
public FormUrlEncodedMessageValidator formUrlEncodedMessageValidator() {
    return new FormUrlEncodedMessageValidator();
}
```

XML

```
<citrus:message-validators>
  <citrus:validator
class="org.citrusframework.http.validation.FormUrlEncodedMessageValidator"/>
</citrus:message-validators>
```

Now we are able to receive the urlencoded form data message in a test.

Java

```
receive("httpServer")
    .message()
    .type("x-www-form-urlencoded")
    .body("<form-data xmlns=\"http://www.citrusframework.org/schema/http/message\">\n"
+
            "<content-type>application/x-www-form-urlencoded</content-type>\n" +
            "<action>/form-test</action>\n" +
            "<controls>\n" +
                "<control name=\"password\">\n" +
                    "<value>${password}</value>\n" +
                "</control>\n" +
                "<control name=\"username\">\n" +
                    "<value>${username}</value>\n" +
                "</control>\n" +
            "</controls>\n" +
            "</form-data>")
    .header(HttpMessageHeaders.HTTP_REQUEST_METHOD, HttpMethod.POST.name())
    .header(HttpMessageHeaders.HTTP_REQUEST_URI, "/form-test")
    .header(HttpMessageHeaders.HTTP_CONTENT_TYPE, "application/x-www-form-
urlencoded");
```

XML

```
<receive endpoint="httpServer">
  <message type="x-www-form-urlencoded">
    <payload>
      <form-data xmlns="http://www.citrusframework.org/schema/http/message">
        <content-type>application/x-www-form-urlencoded</content-type>
        <action>/form-test</action>
        <controls>
          <control name="password">
            <value>${password}</value>
          </control>
          <control name="username">
            <value>${username}</value>
          </control>
        </controls>
      </form-data>
    </payload>
  </message>
  <header>
    <element name="citrus_http_method" value="POST"/>
    <element name="citrus_http_request_uri" value="/form-test"/>
    <element name="Content-Type" value="application/x-www-form-urlencoded"/>
  </header>
</receive>
```

We use a special message type **x-www-form-urlencoded** so the new message validator will take

action. The form url encoded message validator is able to handle a special XML representation of the form data. This enables the very powerful XML message validation capabilities of Citrus such as ignoring elements and usage of test variables inline.

Each form control is translated to a control element with respective name and value properties. The form data is validated in a more comfortable way as the plaintext message validator would be able to offer.

14.8. Http error handling

So far we have received response messages with Http status code **200 OK** . How to deal with server errors like **404 Not Found** or **500 Internal server error** ? The default Http message client error strategy is to propagate server error response messages to the receive action for validation. We simply check on Http status code and status text for error validation.

Java

```
http().client("httpClient")
        .send()
        .post()
        .message()
        .body("<testRequestMessage>" +
                "<text>Hello HttpServer</text>" +
              "</testRequestMessage>");

http().client("httpClient")
        .receive()
        .response(HttpStatus.FORBIDDEN);
```

XML

```
<http:send-request client="httpClient">
    <http:POST>
        <http:body>
            <http:payload>
                <testRequestMessage>
                    <text>Hello HttpServer</text>
                </testRequestMessage>
            </http:payload>
        </http:body>
    </http:POST>
</http:send-request>

<http:receive-response client="httpClient">
    <http:headers status="403" reason-phrase="FORBIDDEN"/>
    <http:body>
        <http:data><![CDATA[]]></http:data>
    </http:body>
</http:receive-response>
```

The message data can be empty depending on the server logic for these error situations. If we receive additional error information as message payload just add validation assertions as usual.

Instead of receiving such empty messages with checks on Http status header information we can change the error strategy in the message sender component in order to automatically raise exceptions on response messages other than **200 OK** . Therefore we go back to the Http message sender configuration for changing the error strategy.

Java

```
@Bean
public HttpClient httpClient() {
    return new HttpClientBuilder()
        .requestUrl("http://localhost:8080/test")
        .errorHandlingStrategy(ErrorHandlingStrategy.THROWS_EXCEPTION)
        .build();
}
```

XML

```
<citrus-http:client id="httpClient"
                  request-url="http://localhost:8080/test"
                  error-strategy="throwsException"/>
```

Now we expect an exception to be thrown because of the error response. Following from that we have to change our test case. Instead of receiving the error message with receive action we assert the client exception and check on the Http status code and status text.

Java

```
assertException()
    .exception(HttpClientErrorException.class)
    .message("403 Forbidden")
    .when(
        http().client("httpClient")
            .send()
            .post()
            .message()
            .body("<testRequestMessage>" +
                    "<text>Hello HttpServer</text>" +
                  "</testRequestMessage>")
    );
```

XML

```
<assert exception="org.springframework.web.client.HttpClientErrorException"
           message="403 Forbidden">
    <when>
        <http:send-request client="httpClient">
            <http:body>
                <http:payload>
                    <testRequestMessage>
                        <text>Hello HttpServer</text>
                    </testRequestMessage>
                </http:payload>
            </http:body>
        </http:send-request>
    </when>
</assert>
```

Both ways of handling Http error messages on client side are valid for expecting the server to raise Http error codes. Choose the preferred way according to your test project requirements.

14.9. Http client basic authentication

As a client you may have to authenticate properly in order to access a resource on the server. In many cases this will be basic authentication with username/password where the credentials are transmitted in the request header section with base64 encoding.

The easiest approach to set the **Authorization** header for a basic authentication Http request would be to explicitly set the header in the send action definition. Of course, you have to use the correct basic authentication header syntax with base64 encoding for the username:password phrase. See how it works in this simple example.

Java

```
http().client("httpClient")
    .send()
    .get()
    .message()
    .header("Authorization", "Basic c29tZVVzZXJuYW1lOnNvbWVQYXNzd29yZA==");
```

XML

```
<http:send-request client="httpClient">
  <http:GET>
    <http:headers>
      <http:header name="Authorization" value="Basic
c29tZVVzZXJuYW1lOnNvbWVQYXNzd29yZA==”/>
    </http:headers>
  </http:GET>
</http:send-request>
```

Citrus will add this header to the Http requests and the server will read the **Authorization** username and password. For more convenient base64 encoding you can also use a Citrus function `citrus:encodeBase64(username:password)`. See functions-encode-base64 for more details.

Instead of adding the Http **Authorization** header manually you can also enable the basic authentication on the Http client endpoint. Simply configure the basic authentication credentials on the Citrus Http client component. Just see the following example and learn how to do that.

Java

```
@Bean
public HttpClient httpClient() {
    return new HttpClientBuilder()
        .requestUrl("http://localhost:8080/test")
        .authentication(HttpAuthentication.basic("username", "password"))
        .build();
}
```

XML

```
<citrus-http:client id="httpClient"
                    request-url="http://localhost:8080/test"
                    request-factory="clientRequestFactory"/>

<bean id="clientRequestFactory"
    class="org.citrusframework.http.security.HttpClientRequestFactoryBean">
    <constructor-arg>
        <bean class="org.citrusframework.http.security.BasicAuthentication">
            <constructor-arg value="username"/>
            <constructor-arg value="password"/>
        </bean>
    </constructor-arg>
</bean>
```

In case you need a more powerful configuration of the basic authentication you can also directly use the request factory as a configuration item for basic authentication.

Java

```
@Bean
public HttpClient httpClient() {
    return new HttpClientBuilder()
        .requestUrl("http://localhost:8080/test")
        .requestFactory(basicAuthFactory())
        .build();
}

@Bean
public BasicAuthClientHttpRequestFactory basicAuthFactory() {
    BasicAuthClientHttpRequestFactory factory = new
BasicAuthClientHttpRequestFactory();

    AuthScope scope = new AuthScope("localhost", "8080", "", "basic");
    factory.setAuthScope(scope);

    UsernamePasswordCredentials credentials = new
UsernamePasswordCredentials("username", "password");
    factory.setCredentials(credentials);

    return factory;
}
```

XML

```
<citrus-http:client id="httpClient"
                    request-url="http://localhost:8080/test"
                    request-factory="basicAuthFactory"/>

<bean id="basicAuthFactory"
    class="org.citrusframework.http.client.BasicAuthClientHttpRequestFactory">
  <property name="authScope">
      <bean class="org.apache.hc.client5.http.auth.AuthScope">
        <constructor-arg value="localhost"/>
        <constructor-arg value="8080"/>
      </bean>
  </property>
  <property name="credentials">
    <bean class="org.apache.hc.client5.http.auth.UsernamePasswordCredentials">
        <constructor-arg value="username"/>
        <constructor-arg value="password"/>
    </bean>
  </property>
</bean>
```

The advantage of setting the basic authentication on the Http client is obvious. All test actions that reference the client component will automatically use the basic authentication.

The above configuration results in Http client requests with authentication headers properly set for basic authentication. The client request factory takes care of adding the proper basic authentication header to each request that is sent with this Citrus message sender. Citrus uses preemptive authentication. The message sender only sends a single request to the server with all authentication information set in the message header. The request which determines the authentication scheme on the server is skipped. This is why you have to add some auth scope in the client request factory so Citrus can set up an authentication cache within the Http context in order to have preemptive authentication.

As a result of the basic auth client request factory the following example request that is created by the Citrus Http client has the **Authorization** header set. This is done now automatically for all requests with this Http client.

Sample request with basic authentication

```
POST /test HTTP/1.1
Accept: application/xml
Content-Type: application/xml
Accept-Charset: iso-8859-1, us-ascii, utf-8
Authorization: Basic c29tZVVzZXJuYW1lOnNvbWVQYXNzd29yZA==
User-Agent: Jakarta Commons-HttpClient/3.1
Host: localhost:8080
Content-Length: 175
<testRequestMessage>
  <text>Hello HttpServer</text>
</testRequestMessage>
```

14.10. Http server basic authentication

The Citrus Http server is able to configure basic authentication so clients need to authenticate properly when accessing server resources.

Java

```
@Bean
public HttpServer httpServer() {
    return new HttpServerBuilder()
        .port(8080)
        .autoStart(true)
        .authentication("/foo/*", HttpAuthentication.basic("username", "password"))
        .build();
}
```

XML

```
<citrus-http:server id="basicAuthHttpServer"
                port="8080"
                auto-start="true"
                security-handler="securityHandlerFactory"/>

<bean id="securityHandlerFactory"
class="org.citrusframework.http.security.HttpSecurityHandlerFactoryBean">
    <constructor-arg>
        <bean class="org.citrusframework.http.security.BasicAuthentication">
            <constructor-arg value="username"/>
            <constructor-arg value="password"/>
            <property name="resourcePath" value="/foo/*"/>
        </bean>
    </constructor-arg>
</bean>
```

In case you need a more powerful server configuration you can also set the security handler directly.

Java

```
@Bean
public HttpServer httpServer() {
    return new HttpServerBuilder()
        .port(8080)
        .autoStart(true)
        .securityHandler(securityHandler())
        .build();
}

@Bean
public SecurityHandlerFactory securityHandler() {
    SecurityHandlerFactory factory = new SecurityHandlerFactory();

    User user = new User();
    user.setName("citrus");
    user.setPassword("secret");
    user.setRoles("CitrusRole");
    factory.setUsers(Collections.singletonList(user));

    factory.setConstraints(Collections.singletonMap("/foo/*",
                                    new BasicAuthConstraint("CitrusRole")));

    return factory;
}
```

XML

```
<citrus-http:server id="basicAuthHttpServer"
                port="8080"
                auto-start="true"
                security-handler="securityHandler"/>

<bean id="securityHandler"
class="org.citrusframework.http.security.SecurityHandlerFactory">
    <property name="users">
        <list>
            <bean class="org.citrusframework.http.security.User">
                <property name="name" value="citrus"/>
                <property name="password" value="secret"/>
                <property name="roles" value="CitrusRole"/>
            </bean>
        </list>
    </property>
    <property name="constraints">
        <map>
            <entry key="/foo/*">
                <bean class="org.citrusframework.http.security.BasicAuthConstraint">
                    <constructor-arg value="CitrusRole"/>
                </bean>
            </entry>
        </map>
    </property>
</bean>
```

We have set a security handler on the server web container with a constraint on all resources with `/foo/*`. Following from that the server requires basic authentication for these resources. The granted users and roles are specified within the security handler bean definition. Connecting clients have to set the basic auth Http header properly using the correct user and role for accessing the Citrus server now.

You can customize the security handler for your very specific needs (e.g. load users and roles with JDBC from a database). Just have a look at the code base and inspect the settings and properties offered by the security handler interface.

This mechanism is not restricted to basic authentication only. With other settings you can also set up digest or form-based authentication constraints very easy.

14.11. Http client SSL

You can use SSL when establishing a client connection with the server. The Citrus Http client supports a secure connection with these settings.

Java

```
@Bean
public HttpClient httpClient() {
    return new HttpClientBuilder()
        .requestUrl("https://localhost:8080/test")
        .secured(HttpSecureConnection.ssl())
        .build();
}
```

XML

```
<citrus-http:client id="httpClient"
                    request-url="http://localhost:8080/test"
                    connection-manager="sslConnectionManager"/>

<bean id="sslConnectionManager"
    class="org.citrusframework.http.security.HttpClientConnectionManagerFactory">
    <constructor-arg>
        <bean class="org.citrusframework.http.security.SSLConnection"/>
    </constructor-arg>
</bean>
```

The example Http client uses SSL secure connection. By default, the client will accept any server certificates and hostnames often referred to as trust-all-strategy.

You may want to also use a proper trust store to explicitly accept server certificates. The client by default allows self-signed certificates.

Java

```
@Bean
public HttpClient sslHttpClient() {
    return new HttpClientBuilder()
        .requestUrl("https://localhost:8080/test")
        .secured(HttpSecureConnection.ssl()
            .trustStore("path/to/server.jks", "trustStorePassword"))
        .build();
}
```

XML

```
<citrus-http:client id="sslHttpClient"
                    request-url="http://localhost:8080/test"
                    connection-manager="sslConnectionManager"/>

<bean id="sslConnectionManager"
    class="org.citrusframework.http.security.HttpClientConnectionManagerFactory">
    <constructor-arg>
        <bean class="org.citrusframework.http.security.SSLConnection">
            <constructor-arg value="path/to/server.jks"/>
            <constructor-arg value="trustStorePassword"/>
        </bean>
    </constructor-arg>
</bean>
```

You need to provide a key store file (.jks) and a password to access the certificates located in the store. The client now will do proper SSL handshake with the server in order to use secure connections.

14.12. Http server SSL

The Citrus Http server is able to configure a SSL connector so clients need to use secure connections to access the server resources. The connector

Java

```
@Bean
public HttpServer sslHttpServer() {
    return new HttpServerBuilder()
        .port(8080)
        .autoStart(true)
        .secured(8443, HttpSecureConnection.ssl()
            .keyStore("path/to/server.jks", "keyStorePassword"))
        .build();
}
```

XML

```
<citrus-http:server id="sslHttpServer"
                port="8080"
                auto-start="true"
                connector="serverConnectorFactory"/>

<bean id="serverConnectorFactory"
class="org.citrusframework.http.security.HttpServerConnectorFactory">
    <constructor-arg>
        <bean class="org.citrusframework.http.security.SSLConnection">
            <constructor-arg value="path/to/server.jks"/>
            <constructor-arg value="trustStorePassword"/>
            <property name="securePort" value="8443"/>
        </bean>
    </constructor-arg>
</bean>
```

The server now uses a `securePort` and a server certificate. In most cases the certificate in a test environment is self-signed so please make sure to allow self-signed certificates in your clients.

14.13. Http cookies

Cookies hold any kind of information and are saved as test information on the client side. Http servers are able to instruct the client (browser) to save a new cookie with name, value and some attributes. This is usually done with a *"Set-Cookie"* message header set on the server response message. Citrus is able to add those cookie information in a server response.

Java

```
Cookie cookie = new Cookie("Token", "${messageId}");
cookie.setPath("/test/cookie.py");
cookie.setSecure(false);
cookie.setDomain("citrusframework.org");
cookie.setMaxAge(86400);

http().server("httpServer")
    .receive()
    .post()
    .message()
    .body("Some request data")
    .header("Operation", "sayHello");

http().server("httpServer")
    .send()
    .response(HttpStatus.OK)
    .message()
    .body("Some response body")
    .header("Operation", "sayHello")
    .cookie(cookie);
```

XML

```
<http:receive-request server="httpServer">
    <http:POST>
        <http:headers>
            <http:header name="Operation" value="getCookie"/>
        </http:headers>
    <http:body>
        <http:data>
            <![CDATA[
            Some request data
            ]]>
        </http:data>
    </http:body>
    </http:POST>
</http:receive-request>

<http:send-response server="httpServer">
    <http:headers status="200" reason-phrase="OK" version="HTTP/1.1">
        <http:header name="Operation" value="getCookie"/>
        <http:cookie name="Token"
                     value="${messageId}"
                     secure="false"
                     domain="citrusframework.org"
                     path="/test/cookie.py"
                     max-age="86400"/>
    </http:headers>
    <http:body>
        <http:data>
            <![CDATA[
            Some response body
            ]]>
        </http:data>
    </http:body>
</http:send-response>
```

The sample above receives an Http request with method POST and some request data. The server response is specified with *Http 200 OK* and some additional cookie information. The cookie is part of the message header specification and gets a name and value as well as several other attributes. This response will result in an Http response with the *"Set-Cookie"* header set:

Set cookie header

```
Set-Cookie:Token=5877643571;Path=/test/cookie.py;Domain=citrusframework.org;Max-
Age=86400
```

As you can see test variables are replaced before the cookie is added to the response. The client now is able to receive the cookie information for validation:

Java

```
Cookie cookie = new Cookie("Token", "${messageId}");
cookie.setPath("/test/cookie.py");
cookie.setSecure(false);
cookie.setDomain("citrusframework.org");
cookie.setMaxAge(86400);

http().client("echoHttpClient")
    .receive()
    .response(HttpStatus.OK)
    .message()
    .body("Some response body")
    .header("Operation", "sayHello")
    .cookie(cookie);
```

XML

```
<http:receive-response server="echoHttpClient">
  <http:headers status="200" reason-phrase="OK" version="HTTP/1.1">
  <http:header name="Operation" value="getCookie"/>
  <http:cookie name="Token"
               value="${messageId}"
               secure="false"
               domain="citrusframework.org"
               path="/test/cookie.py"
               max-age="86400"/>
  </http:headers>
  <http:body>
    <http:data>
      <![CDATA[
        Some response body
      ]]>
    </http:data>
  </http:body>
</http:receive-response>
```

Once again the cookie information is added to the header specification. The Citrus message validation will make sure that the cookie information is present with all specified attributes.

In all further actions the client is able to continue to send the cookie information with name and value:

Java DSL

```
http().client("echoHttpClient")
    .send()
    .post()
    .fork(true)
    .message()
    .body("Some other request data")
    .header("Operation", "sayHello")
    .cookie(new Cookie("Token", "${messageId}"));
```

XML

```
<http:send-request client="echoHttpClient" fork="true">
  <http:POST>
    <http:headers>
      <http:header name="Operation" value="sayHello"/>
      <http:cookie name="Token" value="${messageId}"/>
    </http:headers>
    <http:body>
      <http:data>
        <![CDATA[
          Some other request data
        ]]>
      </http:data>
    </http:body>
  </http:POST>
</http:send-request>
```

The cookie now is only specified with name and value as the cookie now goes to the *"Cookie"* request message header.

Cookie token

```
Cookie:Token=5877643571
```

Of course the Citrus Http server can now also validate the cookie information in a request validation:

Java

```
http().server("httpServer")
    .receive()
    .post()
    .message()
    .body("Some other request data")
    .header("Operation", "sayHello")
    .cookie(new Cookie("Token", "${messageId}"));
```

XML

```
<http:receive-request client="httpServer">
  <http:POST>
    <http:headers>
      <http:header name="Operation" value="sayHello"/>
      <http:cookie name="Token" value="${messageId}"/>
    </http:headers>
    <http:body>
      <http:data>
        <![CDATA[
          Some other request data
        ]]>
      </http:data>
    </http:body>
  </http:POST>
</http:receive-request>
```

The Citrus message validation will make sure that the cookie is set in the request with respective name and value.

14.14. Http Gzip compression

Gzip is a very popular compression mechanism for optimizing the message transportation for large content. The Citrus http client and server components support gzip compression out of the box. This means that you only need to set the specific encoding headers in your http request/response message.

Accept-Encoding=gzip

Setting for clients when requesting gzip compressed response content. The Http server must support gzip compression then in order to provide the response as zipped byte stream. The Citrus http server component automatically recognizes this header in a request and applies gzip compression to the response.

Content-Encoding=gzip

When an http server sends compressed message content to the client this header is set to **gzip** in order to mark the compression. The Http client must support gzip compression then in order to unzip the message content. The Citrus http client component automatically recognizes this header in a response and applies gzip unzip logic before passing the message to the test case.

The Citrus client and server automatically take care of gzip compression when those headers are set. In the test case you do not need to zip or unzip the content then as it is automatically done before.

This means that you can request gzipped content from a server with just adding the message header **Accept-Encoding** in your http request operation.

Java

```
http().client("gzipClient")
    .send()
    .post()
    .message()
    .body("Some other request data")
    .contentType("text/html")
    .header("Accept-Encoding", "gzip")
    .header("Accept", "text/plain");

http().client("gzipClient")
    .receive()
    .response(HttpStatus.OK)
    .message()
    .type(MessageType.PLAINTEXT)
    .contentType("text/plain")
    .body("${text}");
```

XML

```
<http:send-request client="gzipClient">
  <http:POST>
    <http:headers content-type="text/html">
      <http:header name="Accept-Encoding" value="gzip"/>
      <http:header name="Accept" value="text/plain"/>
    </http:headers>
  </http:POST>
</http:send-request>

<http:receive-response client="gzipClient">
    <http:headers status="200" reason-phrase="OK">
      <http:header name="Content-Type" value="text/plain"/>
    </http:headers>
    <http:body type="plaintext">
      <http:data>${text}</http:data>
    </http:body>
</http:receive-response>
```

On the server side if we receive a message and the response should be compressed with Gzip we just have to set the **Content-Encoding** header in the response operation.

Java

```
http().server("httpServer")
    .receive()
    .post()
    .message()
    .body("Some other request data")
    .contentType("text/html")
    .header("Accept-Encoding", "gzip")
    .header("Accept", "text/plain");

http().server("httpServer")
    .send()
    .response(HttpStatus.OK)
    .message()
    .contentType("text/plain")
    .body("${text}");
```

XML

```
<http:receive-request server="httpServer">
    <http:POST path="/echo">
      <http:headers>
        <http:header name="Content-Type" value="text/html"/>
        <http:header name="Accept-Encoding" value="gzip"/>
        <http:header name="Accept" value="text/plain"/>
      </http:headers>
    </http:POST>
</http:receive-request>

<http:send-response server="httpServer">
    <http:headers status="200" reason-phrase="OK">
      <http:header name="Content-Encoding" value="gzip"/>
      <http:header name="Content-Type" value="text/plain"/>
    </http:headers>
    <http:body>
      <http:data>${text}</http:data>
    </http:body>
</http:send-response>
```

So the Citrus server will automatically add gzip compression to the response for us.

Of course you can also send gzipped content as a client. Then you would just set the **Content-Encoding** header to **gzip** in your request. The client will automatically apply compression for you.

14.15. Http servlet filters

The Citrus http server component supports custom servlet filters that take part in handling an incoming request/response communication. This might be useful when customizing the basic

server behavior such as custom zip/unzip mechanisms. The custom servlet filters are referenced in the http server component as follows:

Java

```
@Bean
public HttpServer httpServer() {
    return new HttpServerBuilder()
        .port(8080)
        .filters(filters())
        .filterMappings(filterMappings())
        .build();
}

public Map<String, Filter> filters() {
    Map<String, Filter> filters = new HashMap<>();

    filters.put("request-caching-filter", new RequestCachingServletFilter());
    filters.put("gzip-filter", new GzipServletFilter());

    return filters;
}

public Map<String, String> filterMappings() {
    Map<String, String> filterMappings = new HashMap<>();

    filterMappings.put("request-caching-filter", "/*");
    filterMappings.put("gzip-filter", "/gzip/*");

    return filterMappings;
}
```

XML

```
<citrus-http:server id="httpServer"
                    port="8080"
                    filters="filters"
                    filter-mappings="filterMappings"/>

<util:map id="filters">
    <entry key="request-caching-filter">
        <bean class="org.citrusframework.http.servlet.RequestCachingServletFilter"/>
    </entry>
    <entry key="gzip-filter">
        <bean class="org.citrusframework.http.servlet.GzipServletFilter"/>
    </entry>
</util:map>

<util:map id="filterMappings">
    <entry key="request-caching-filter" value="/*"/>
    <entry key="gzip-filter" value="/gzip/*"/>
</util:map>
```

The map of filters are specified as normal Spring configuration entries. The server component uses the attribute `filters` to reference a set of custom servlet filters. The map holds one to many servlet filter beans each given a name that is also referenced in the respective servlet mappings. The servlet mappings specify when to apply those filters.

This way you can set a very custom servlet filter chain for each request/response communication. As usual the filter implementations can participate in the request and response handling process.

Citrus provides several default servlet implementations that are automatically added to each http server component these implementations are:

org.citrusframework.http.servlet.RequestCachingServletFilter

caches incoming request data so input streams can be read multiple times during request processing (important when request logging is enabled)

org.citrusframework.http.servlet.GzipServletFilter

applies Gzip compression when according headers are set and client explicitly asks for compressed request/response communication

By the time you define some custom servlet filters or mappings to the server component Citrus will not apply default servlet filters. This means you always need to construct the whole servlet filter chain including default servlet filters mentioned above.

14.16. Http servlet context customization

The Citrus Http server uses Spring application context loading on startup. For high customizations you can provide a custom servlet context file which holds all custom configurations as Spring beans for the server. Here is a sample servlet context with some basic Spring MVC components and the

central HttpMessageController which is responsible for handling incoming requests (GET, PUT, DELETE, POST, etc.).

Java

```
@Bean
public RequestMappingHandlerMapping citrusHandlerMapping() {
    return new RequestMappingHandlerMapping();
}

@Bean
public RequestMappingHandlerAdapter citrusMethodHandlerAdapter() {
    RequestMappingHandlerAdapter adapter = new RequestMappingHandlerAdapter();


adapter.setMessageConverters(Collections.singletonList(citrusHttpMessageConverter()));

    return adapter;
}

@Bean
public DelegatingHttpEntityMessageConverter citrusHttpMessageConverter() {
    return new DelegatingHttpEntityMessageConverter();
}

@Bean
public HttpMessageController citrusHttpMessageController() {
    HttpMessageController controller = new HttpMessageController();

    controller.setEndpointAdapter(new EmptyResponseEndpointAdapter());

    return controller;
}
```

XML

```
<bean id="citrusHandlerMapping"
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapp
ing"/>

<bean id="citrusMethodHandlerAdapter"
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdap
ter">
  <property name="messageConverters">
    <util:list id="converters">
      <ref bean="citrusHttpMessageConverter"/>
    </util:list>
  </property>
</bean>

<bean id="citrusHttpMessageConverter"
class="org.citrusframework.http.message.DelegatingHttpEntityMessageConverter"/>

<bean id="citrusHttpMessageController"
class="org.citrusframework.http.controller.HttpMessageController">
  <property name="endpointAdapter">
      <bean
       class="org.citrusframework.endpoint.adapter.EmptyResponseEndpointAdapter"/>
  </property>
</bean>
```

The beans above are responsible for proper Http server configuration. In general you do not need to adjust those beans, but we have the possibility to do so which gives us a great customization and extension points. The important part is the endpoint adapter definition inside the HttpMessageController. Once a client request was accepted the adapter is responsible for generating a proper response to the client.

You can add the custom servlet context as file resource to the Citrus Http server component. Just use the **context-config-location** attribute as follows:

Java

```
@Bean
public HttpServer httpServer() {
    return new HttpServerBuilder()
        .port(8080)
        .autoStart(true)
        .contextConfigLocation("classpath:org/citrusframework/http/custom-servlet-
context.xml")
        .build();
}
```

XML

```
<citrus-http:server id="helloHttpServer"
      port="8080"
      auto-start="true"
      context-config-location="classpath:org/citrusframework/http/custom-servlet-
context.xml"/>
```

Chapter 15. SOAP WebServices

SOAP Web Services over HTTP is a widely used communication scenario in modern enterprise applications. A SOAP Web Service client is posting a SOAP request via HTTP to a server. SOAP via HTTP is a synchronous message protocol by default so the client is waiting synchronously for the response message. Citrus provides both SOAP client and server components in order to meet both directions of this scenario. The components used are very similar to the HTTP components that were discussed in the sections before.

The SOAP WebService components in Citrus are kept in a separate Maven module. So you should add the module as Maven dependency to your project accordingly.

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-ws</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

In order to use the SOAP WebService support you need to include the specific XML configuration schema provided by Citrus. See the following XML definition to find out how to include the citrus-ws namespace.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:citrus="http://www.citrusframework.org/schema/config"
       xmlns:citrus-ws="http://www.citrusframework.org/schema/ws/config"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.citrusframework.org/schema/config
       http://www.citrusframework.org/schema/config/citrus-config.xsd
       http://www.citrusframework.org/schema/ws/config
       http://www.citrusframework.org/schema/ws/config/citrus-ws-config.xsd">

    [...]

</beans>
```

Now you are ready to use the customized soap configuration elements - all using the citrus-ws prefix - in your Spring configuration.

15.1. SOAP client

Citrus is able to form a proper SOAP request in order to pass it to the server via HTTP and validate the respective SOAP response message. Let us see how a message client for SOAP looks like in the Spring configuration:

```
<citrus-ws:client id="soapClient"
                        request-url="http://localhost:8090/test"
                        timeout="60000"/>
```

The client component uses the **request-url** in order to access the server resource. The client will automatically build a proper SOAP request message including the SOAP envelope, SOAP header and the message payload as SOAP body. This means that you as a tester do not care about SOAP envelope specific logic in the test case. The client endpoint component saves the synchronous SOAP response so the test case can receive this message with a normal receive test action.

In detail you as a tester just send and receive using the same client endpoint reference just as you would do with a synchronous JMS or channel communication. In case no response message is available in time according to the timeout settings Citrus raises a timeout error and the test will fail.

The SOAP client component uses a SoapMessageFactory implementation in order to create the SOAP messages. This is a Spring bean added to the Citrus Spring application context. Spring offers several reference implementations as message factories so you can choose one of them (e.g. for SOAP 1.1 or 1.2 implementations).

```
<!-- Default SOAP Message Factory (SOAP 1.1) -->
<bean id="messageFactory"
class="org.springframework.ws.soap.saaj.SaajSoapMessageFactory"/>

<!-- SOAP 1.2 Message Factory -->
<bean id="soap12MessageFactory"
class="org.springframework.ws.soap.saaj.SaajSoapMessageFactory">
  <property name="soapVersion">
    <util:constant static-field="org.springframework.ws.soap.SoapVersion.SOAP_12"/>
  </property>
</bean>
```

By default Citrus will search for a bean with id **'messageFactory'** . In case you intend to use different identifiers you need to tell the SOAP client component which message factory to use:

```
<citrus-ws:client id="soapClient"
         request-url="http://localhost:8090/test"
         message-factory="soap12MessageFactory"/>
```

Up to now we have used a static endpoint request url for the SOAP message sender. Besides that we can use dynamic endpoint uri in configuration. We just use an endpoint uri resolver instead of the static request url like this:

```
<citrus-ws:client id="soapClient"
             endpoint-resolver="dynamicEndpointResolver"
             message-factory="soap12MessageFactory"/>

<bean id="dynamicEndpointResolver"
       class="org.citrusframework.endpoint.resolver.DynamicEndpointUriResolver"/>
```

The **dynamicEndpointResolver** bean must implement the EndpointUriResolver interface in order to resolve dynamic endpoint uri values. Citrus offers a default implementation, the **DynamicEndpointUriResolver**, which uses a specific message header for setting the dynamic endpoint uri for each message. The message header needs to specify the header **citrus_endpoint_uri** with a valid request uri. Just like this:

```
<header>
  <element name="citrus_endpoint_uri"
           value="http://localhost:${port}/${context}" />
</header>
```

As you can see you can use dynamic test variables then in order to build the request uri to use. The SOAP client evaluates the endpoint uri header and sends the message to this server resource. You can use a different uri value then in different test cases and send actions.

15.2. SOAP client interceptors

The client component is able to add custom interceptors that participate in the request/response processing. The interceptors need to implement the common interface **org.springframework.ws.client.support.interceptor.ClientInterceptor**.

```
<citrus-ws:client id="secureSoapClient"
                  request-url="http://localhost:8080/services/ws/todolist"
                  interceptors="clientInterceptors"/>

<util:list id="clientInterceptors">
  <bean class="org.springframework.ws.soap.security.wss4j.Wss4jSecurityInterceptor">
    <property name="securementActions" value="Timestamp UsernameToken"/>
    <property name="securementUsername" value="admin"/>
    <property name="securementPassword" value="secret"/>
  </bean>
  <bean class="org.citrusframework.ws.interceptor.LoggingClientInterceptor"/>
</util:list>
```

The sample above adds Wss4J WsSecurity interceptors in order to add security constraints to the request messages.

When customizing the interceptor chain all default interceptors (like logging interceptor) are lost. You need to add these interceptors explicitly as shown with the *org.citrusframework.ws.interceptor.LoggingClientInterceptor* which is able to log request/response messages during communication.

15.3. SOAP server

Every client needs a server to talk to. When receiving SOAP messages we require a web server instance listening on a port. Citrus is using an embedded Jetty server instance in combination with the Spring Web Service API in order to accept SOAP request calls as a server. See how the Citrus SOAP server is configured in the Spring configuration.

```
<citrus-ws:server id="helloSoapServer"
             port="8080"
             auto-start="true"
             resource-base="src/it/resources"/>
```

The server component is able to start automatically when application starts up. In the example above the server is listening for requests on port ***8080*** . This setup uses the standard connector configuration for the Jetty server. For detailed customization the Citrus Jetty server configuration also supports explicit connector configurations (@connector and @connectors attributes). For more information please see the Jetty connector documentation.

Test cases interact with this server instance via message channels by default. The server component provides an inbound channel that holds incoming request messages. The test case can receive those requests from the channel with a normal receive test action. In a second step the test case can provide a synchronous response message as reply which will be automatically sent back to the calling SOAP client as response.

The figure above shows the basic setup with inbound channel and reply channel. You as a tester should not worry about this too much. By default you as a tester just use the server as synchronous endpoint in your test case. This means that you simply receive a message from the server and send a response back.

```
<testcase name="soapServerTest">
    <actions>
        <receive endpoint="helloSoapServer">
            <message>
                <data>
                  [...]
                </data>
            </message>
        </receive>

        <send endpoint="helloSoapServer">
            <message>
                <data>
                  [...]
                </data>
            </message>
        </send>
    </actions>
</testcase>
```

As you can see we reference the server id in both receive and send actions. The Citrus server instance will automatically send the response back to the calling client. In most cases this is what you need to simulate a SOAP server instance in Citrus. Of course we have some more customization possibilities that we will go over later on. These customizations are optional so you can also skip the next description on endpoint adapters if you are happy with just what you have learned about the SOAP server component in Citrus.

Just like the HTTP server component the SOAP server component by default uses the channel endpoint adapter in order to forward all incoming requests to an in-memory message channel. This is done completely behind the scenes. The Citrus configuration has become a lot easier here so you do not have to configure this by default. When nothing else is set the test case does not worry about that setting on the server and just uses the server id reference as synchronous endpoint.

> The default channel endpoint adapter automatically creates an inbound message channel where incoming messages are stored too internally. So if you need to clean up a server that has already stored some incoming messages you can do this easily by purging the internal message channel. The message channel follows a naming convention **{serverName}.inbound** where **{serverName}** is the Spring bean name of the Citrus server endpoint component. If you purge this internal channel in a before test nature you are sure that obsolete messages on a server instance get purged before each test is executed.

However, we do not want to lose the great extendability and customizing capabilities of the Citrus server component. This is why you can optionally define the endpoint adapter implementation used by the Citrus SOAP server. We provide several message endpoint adapter implementations for different simulation strategies. With these endpoint adapters you should be able to generate proper SOAP response messages for the client in various ways. Before we have a closer look at the different adapter implementations we want to show how you can set a custom endpoint adapter on the

server component.

```
<citrus-ws:server id="helloSoapServer"
        port="8080"
        auto-start="true"
        endpoint-adapter="emptyResponseEndpointAdapter"
        resource-base="src/it/resources"/>

        <citrus:empty-response-adapter id="emptyResponseEndpointAdapter"/>
```

With this endpoint adapter configuration above we change the Citrus server behavior from scratch. Now the server automatically sends back an empty SOAP response message every time. Setting a custom endpoint adapter implementation with custom logic is as easy as defining a custom endpoint adapter Spring bean and reference it in the server attribute. You can read more about endpoint adapters in endpoint-adapter.

15.4. SOAP send and receive

Citrus provides test actions for sending and receiving messages of all kind. Different message content and different message transports are available to these send and receive actions. When using SOAP message transport we might need to set special information on those messages. These are special SOAP headers, SOAP faults and so on. So we have created a special SOAP namespace for all your SOAP related send and receive operations in an XML DSL test:

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
          xmlns:spring="http://www.springframework.org/schema/beans"
          xmlns:ws="http://www.citrusframework.org/schema/ws/testcase"
          xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd
          http://www.citrusframework.org/schema/testcase
          http://www.citrusframework.org/schema/testcase/citrus-testcase.xsd
          http://www.citrusframework.org/schema/ws/testcase
          http://www.citrusframework.org/schema/ws/testcase/citrus-ws-testcase.xsd">
```

Once you have added the **ws** namespace from above to your test case you are ready to use special send and receive operations in the test.

XML DSL

```
<ws:send endpoint="soapClient" soap-action="MySoapService/sayHello">
    <message>
        [...]
    </message>
</ws:send>

<ws:receive endpoint="soapServer" soap-action="MySoapService/sayHello">
    <message>
        [...]
    </message>
</ws:receive>
```

The special namespace contains the following elements:

send	Special send operation for sending out SOAP message content.
receive	Special receive operation for validating SOAP message content.
send-fault	Special send operation for sending out SOAP fault message content.
assert-fault	Special assertion operation for expecting a SOAP fault message as response.

The special SOAP related send and receive actions can coexist with normal Citrus actions. In fact you can mix those action types as you want inside a test case. All test actions that work with SOAP message content on client and server side should use this special namespace.

In Java DSL we have something similar to that. The Java DSL provides special SOAP related features when calling the **soap()** method. With a fluent API you are able to then send and receive SOAP message content as client and server.

Java DSL

```
@CitrusTest
public void soapTest() {

    soap().client("soapClient")
        .send()
        .soapAction("MySoapService/sayHello")
        .payload("...");

    soap().client("soapClient")
        .receive()
        .payload("...");
}
```

In the following sections the SOAP related capabilities are discussed in more detail.

15.5. SOAP headers

SOAP defines several header variations that we discuss in the following sections. First of all we deal with the special **SOAP action** header. In case we need to set this SOAP action header we simply need to use the special ***soap-action*** attribute in our test. The special header key in combination with an underlying SOAP client endpoint component constructs the SOAP action in the SOAP message.

XML DSL

```
<ws:send endpoint="soapClient" soap-action="MySoapService/sayHello">
    <message>
        [...]
    </message>
</ws:send>

<ws:receive endpoint="soapServer" soap-action="MySoapService/sayHello">
    <message>
        [...]
    </message>
</ws:receive>
```

Java DSL

```
@CitrusTest
public void soapActionTest() {

    soap().client("soapClient")
        .send()
        .soapAction("MySoapService/sayHello")
        .payload("...");

    soap().server("soapClient")
        .receive()
        .soapAction("MySoapService/sayHello")
        .payload("...");
}
```

The SOAP action header is added to the message before sending and validated when used in a receive operation.

The **soap-action** attribute is defined in the special SOAP namespace in Citrus. We recommend to use this namespace for all your send and receive operations that deal with SOAP message content. However you can also set the special SOAP action header when not using the special SOAP namespace: Just set this header in your test action:

```
<header>
    <element name="citrus_soap_action" value="sayHello"/>
</header>
```

Secondly a SOAP message is able to contain customized SOAP headers. These are key-value pairs where the key is a qualified name (QName) and the value a normal String value.

```
<header>
    <element name="{http://citrusframework.org/sayHello}h1:Operation"
value="sayHello"/>
    <element name="{http://citrusframework.org/sayHello}h1:Request"
value="HelloRequest"/>
</header>
```

The key is defined as qualified QName character sequence which has a mandatory XML namespace and a prefix along with a header name. Last but not least a SOAP header can contain whole XML fragment values. The next example shows how to set these XML fragments as SOAP header in Citrus:

```
<header>
    <data>
      <![CDATA[
          <User xmlns="http://citrusframework.org/schemas/sayHello">
              <UserId>123456789</UserId>
              <Handshake>S123456789</Handshake>
          </User>
      ]]>
    </data>
</header>
```

You can also use external file resources to set this SOAP header XML fragment as shown in this last example code:

```
<header>
    <resource file="classpath:request-soap-header.xml"/>
</header>
```

This completes the SOAP header possibilities for sending SOAP messages with Citrus. Of course you can also use these variants in SOAP message header validation. You define expected SOAP headers, SOAP action and XML fragments and Citrus will match incoming requests to that. Just use **citrus_soap_action** header key in your receiving message action and you validate this SOAP header accordingly.

When validating SOAP header XML fragments you need to define the whole XML header fragment as expected header data like this:

```
<receive endpoint="soapMessageEndpoint">
    <message>
        <data>
          <![CDATA[
            <ResponseMessage xmlns="http://citrusframework.org/schema">
              <resultCode>OK</resultCode>
            </ResponseMessage>
          ]]>
        </data>
    </message>
    <header>
        <data>
            <![CDATA[
                <SOAP-ENV:Header
                    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
                    <customHeader xmlns="http://citrusframework.org/headerschema">
                        <correlationId>${correlationId}</correlationId>
                        <applicationId>${applicationId}</applicationId>
                        <trackingId>${trackingId}</trackingId>
                        <serviceId>${serviceId}</serviceId>
                        <interfaceVersion>1.0</interfaceVersion>
                        <timestamp>@ignore@</timestamp>
                    </customHeader>
                </SOAP-ENV:Header>
            ]]>
        </data>
        <element name="citrus_soap_action" value="doResponse"/>
    </header>
</receive>
```

As you can see the SOAP XML header validation can combine header element and XML fragment validation. This is also likely to be used when dealing with WS-Security message headers.

15.6. SOAP HTTP mime headers

Besides the SOAP specific header elements the HTTP mime headers (e.g. Content-Type, Content-Length, Authorization) might be candidates for validation, too. When using HTTP as transport layer the SOAP message may define those mime headers. The tester is able to send and validate these headers inside the test case, although these HTTP headers are located outside the SOAP envelope. Let us first of all speak about validating the HTTP mime headers. This feature is not enabled by default. We have enabled this in our SOAP server configuration.

```
<citrus-ws:server id="helloSoapServer"
        port="8080"
        auto-start="true"
        handle-mime-headers="true"
        resource-base="src/it/resources"/>
```

With this configuration Citrus will handle all available mime headers and pass those to the test case for normal header validation.

```
<ws:receive endpoint="helloSoapServer">
    <message>
        <payload>
            <SoapMessageRequest xmlns="http://citrusframework.org/schemas/sample.xsd">
                <Operation>Validate mime headers</Operation>
            </SoapMessageRequest>
        </payload>
    </message>
    <header>
        <element name="Content-Type" value="text/xml; charset=utf-8"/>
    </header>
</ws:receive>
```

The validation of these HTTP mime headers is as usual now that we have enabled the mime header handling in the SOAP server. The transport HTTP headers are available in the header just like the normal SOAP header elements do. So you can validate the headers as usual.

So much for receiving and validating HTTP mime message headers with SOAP communication. Now we want to send special mime headers on client side. We overwrite or add mime headers to our sending action. We mark some headers with following prefix ***"citrus_http"***_ . This tells the SOAP client to add these headers to the HTTP header section outside the SOAP envelope. Keep in mind that header elements without this prefix go right into the SOAP header section by default.

```
<ws:send endpoint="soapClient">
  [...]
  <header>
    <element name="citrus_http_operation" value="foo"/>
  </header>
  [...]
</ws:send>
```

The listing above defines an HTTP mime header **operation** . The header prefix ***citrus_http***_ is cut off before the header goes into the HTTP header section. With this feature we can decide where exactly our header information is located in our resulting client message.

15.7. SOAP Envelope handling

By default Citrus will remove the SOAP envelope in message converter. Following from that the Citrus test case is independent from SOAP message formats and is not bothered with handling of SOAP envelope at all. This is great in most cases but sometimes it might be mandatory to also see the whole SOAP envelope inside the test case receive action. Therefore you can keep the SOAP envelope for incoming messages by configuration on the SOAP server side.

```
<citrus-ws:server id="helloSoapServer"
        port="8080"
        auto-start="true"
        keep-soap-envelope="true"/>
```

With this configuration Citrus will handle all available mime headers and pass those to the test case for normal header validation.

```
<ws:receive endpoint="helloSoapServer">
<message>
  <payload>
    <SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
      <SOAP-ENV:Header/>
      <SOAP-ENV:Body>
        <SoapMessageRequest xmlns="http://citrusframework.org/schemas/sample.xsd">
        <Operation>Validate mime headers</Operation>
        </SoapMessageRequest>
      </SOAP-ENV:Body>
    </SOAP-ENV:Envelope>
  </payload>
  </message>
</ws:receive>
```

So now you are able to validate the whole SOAP envelope as is. This might be of interest in very special cases. As mentioned by default the Citrus server will automatically remove the SOAP envelope and translate the SOAP body to the message payload for straight forward validation inside the test cases.

15.8. SOAP server interceptors

The Citrus SOAP server supports the concept of interceptors in order to add custom logic to the request/response processing steps. The interceptors need to implement a common interface: **org.springframework.ws.server.EndpointInterceptor**. We are able to customize the interceptor chain on the server component as follows:

```
<citrus-ws:server id="secureSoapServer"
                  port="8080"
                  auto-start="true"
                  interceptors="serverInterceptors"/>

<util:list id="serverInterceptors">
  <bean
class="org.citrusframework.ws.interceptor.SoapMustUnderstandEndpointInterceptor">
    <property name="acceptedHeaders">
      <list>
        <value>{http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
secext-1.0.xsd}Security</value>
      </list>
    </property>
  </bean>
  <bean class="org.citrusframework.ws.interceptor.LoggingEndpointInterceptor"/>
  <bean class="org.springframework.ws.soap.security.wss4j.Wss4jSecurityInterceptor">
    <property name="validationActions" value="Timestamp UsernameToken"/>
    <property name="validationCallbackHandler">
      <bean id="passwordCallbackHandler"
class="org.springframework.ws.soap.security.wss4j.callback.SimplePasswordValidationCal
lbackHandler">
        <property name="usersMap">
          <map>
            <entry key="admin" value="secret"/>
          </map>
        </property>
      </bean>
    </property>
  </bean>
</util:list>
```

The custom interceptors are used to enable WsSecurity features on the soap server component via Wss4J.

> When customizing the interceptor chain of the soap server component all default interceptors (like logging interceptors) are lost. You can see that we had to add the *org.citrusframework.ws.interceptor.LoggingEndpointInterceptor* explicitly in order to log request/response messages for the server communication.

15.9. SOAP 1.2

By default Citrus components use SOAP 1.1 version. Fortunately SOAP 1.2 is supported the same way. As we already mentioned before the Citrus SOAP components do use a SOAP message factory for creating messages in SOAP format.

```
<!-- SOAP 1.1 Message Factory -->
<bean id="soapMessageFactory"
class="org.springframework.ws.soap.saaj.SaajSoapMessageFactory">
  <property name="soapVersion">
    <util:constant static-field="org.springframework.ws.soap.SoapVersion.SOAP_11"/>
  </property>
</bean>

<!-- SOAP 1.2 Message Factory -->
<bean id="soap12MessageFactory"
class="org.springframework.ws.soap.saaj.SaajSoapMessageFactory">
  <property name="soapVersion">
    <util:constant static-field="org.springframework.ws.soap.SoapVersion.SOAP_12"/>
  </property>
</bean>
```

As you can see the SOAP message factory can either create SOAP 1.1 or SOAP 1.2 messages. This is how Citrus can create both SOAP 1.1 and SOAP 1.2 messages. Of course you can have multiple message factories configured in your project. Just set the message factory on a WebService client or server component in order to define which version should be used.

```
<citrus-ws:client id="soap12Client"
            request-url="http://localhost:8080/echo"
            message-factory="soap12MessageFactory"
            timeout="1000"/>

<citrus-ws:server id="soap12Server"
          port="8080"
          auto-start="true"
          root-parent-context="true"
          message-factory="soap12MessageFactory"/>
```

By default Citrus components do connect with a message factory called **messageFactory** no matter what SOAP version this factory is using.

15.10. SOAP faults

SOAP faults describe a failed communication in SOAP WebServices world. Citrus is able to send and receive SOAP fault messages. On server side Citrus can simulate SOAP faults with fault-code, fault-reason, fault-actor and fault-detail. On client side Citrus is able to handle and validate SOAP faults in response messages. The next section describes how to deal with SOAP faults in Citrus.

15.11. Send SOAP faults

As Citrus simulates SOAP server endpoints you also need to think about sending a SOAP fault to the calling client. In case Citrus receives a SOAP request as a server you can respond with a proper SOAP fault if necessary.

Please keep in mind that we use the citrus-ws extension for sending SOAP faults in our test case, as shown in this very simple example:

XML DSL

```
<ws:send-fault endpoint="helloSoapServer">
    <ws:fault>
        <ws:fault-code>{http://citrusframework.org/faults}citrus:TEC-1000</ws:fault-
code>
        <ws:fault-string>Invalid request</ws:fault-string>
        <ws:fault-actor>SERVER</ws:fault-actor>
        <ws:fault-detail>
            <![CDATA[
                <FaultDetail xmlns="http://citrusframework.org/schemas/sayHello.xsd">
                    <MessageId>${messageId}</MessageId>
                    <CorrelationId>${correlationId}</CorrelationId>
                    <ErrorCode>TEC-1000</ErrorCode>
                    <Text>Invalid request</Text>
                </FaultDetail>
            ]]>
        </ws:fault-detail>
    </ws:fault>
    <ws:header>
        <ws:element name="citrus_soap_action" value="sayHello"/>
    </ws:header>
</ws:send-fault>
```

The example generates a simple SOAP fault that is sent back to the calling client. The fault-actor and the fault-detail elements are optional. Same with the soap action declared in the special Citrus header ***citrus_soap_action*** . In the sample above the fault-detail data is placed inline as XML data. As an alternative to that you can also set the fault-detail via external file resource. Just use the ***file*** attribute as fault detail instead of the inline CDATA definition.

XML DSL

```
<ws:send-fault endpoint="helloSoapServer">
    <ws:fault>
        <ws:fault-code>{http://citrusframework.org/faults}citrus:TEC-1000</ws:fault-
code>
        <ws:fault-string>Invalid request</ws:fault-string>
        <ws:fault-actor>SERVER</ws:fault-actor>
        <ws:fault-detail file="classpath:myFaultDetail.xml"/>
    </ws:fault>
    <ws:header>
        <ws:element name="citrus_soap_action" value="sayHello"/>
    </ws:header>
</ws:send-fault>
```

The generated SOAP fault looks like follows:

```
HTTP/1.1 500 Internal Server Error
Accept: text/xml, text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
SOAPAction: "sayHello"
Content-Type: text/xml; charset=utf-8
Content-Length: 680
Server: Jetty(7.0.0.pre5)

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header/>
    <SOAP-ENV:Body>
        <SOAP-ENV:Fault>
            <faultcode xmlns:citrus="http://citrusframework.org/faults">citrus:TEC-
1000</faultcode>
            <faultstring xml:lang="en">Invalid request</faultstring>
            <detail>
                <FaultDetail xmlns="http://citrusframework.org/schemas/sayHello.xsd">
                    <MessageId>9277832563</MessageId>
                    <CorrelationId>4346806225</CorrelationId>
                    <ErrorCode>TEC-1000</ErrorCode>
                    <Text>Invalid request</Text>
                </FaultDetail>
            </detail>
        </SOAP-ENV:Fault>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Notice that the send action uses a special XML namespace (ws:send). This ws namespace belongs to the Citrus WebService extension and adds SOAP specific features to the normal send action. When you use such ws extensions you need to define the additional namespace in your test case. This is usually done in the root `<spring:beans>` element where we simply declare the citrus-ws specific namespace like follows.

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
    xmlns:spring="http://www.springframework.org/schema/beans"
    xmlns:ws="http://www.citrusframework.org/schema/ws/testcase"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.citrusframework.org/schema/testcase
    http://www.citrusframework.org/schema/testcase/citrus-testcase.xsd
    http://www.citrusframework.org/schema/ws/testcase
    http://www.citrusframework.org/schema/ws/testcase/citrus-ws-testcase.xsd">
```

15.12. Receive SOAP faults

In case you receive SOAP response messages as a client endpoint you may need to handle and validate SOAP faults in error situations. Citrus can validate SOAP faults with fault-code, fault-actor,

fault-string and fault-detail values.

As a client we send out a request and receive a SOAP fault as response. By default the client sending action in Citrus throws a specific exception when the SOAP response is a SOAP fault element. This exception is called ***SoapFaultClientException*** coming from the Spring API. You as a tester can assert this kind of exception in a test case in order to expect the SOAP error.

XML DSL

```
<assert class="org.springframework.ws.soap.client.SoapFaultClientException">
    <send endpoint="soapClient">
        <message>
            <payload>
                <SoapFaultForcingRequest
                  xmlns="http://citrusframework.org/schemas/soap">
                    <Message>This is invalid</Message>
                </SoapFaultForcingRequest>
            </payload>
        </message>
    </send>
</assert>
```

The SOAP message sending action is surrounded by a simple assert action. The asserted exception class is the ***SoapFaultClientException*** that we have mentioned before. This means that the test expects the exception to be thrown during the communication. In case the exception is missing the test fails.

So far we have used the Citrus core capabilities of asserting an exception. This basic assertion test action is not able to offer direct access to the SOAP fault-code and fault-string values for validation. The basic assert action simply has no access to the actual SOAP fault elements. Fortunately we can use the **citrus-ws** namespace again which offers a special assert action implementation especially designed for SOAP faults in this case.

XML DSL

```
<ws:assert-fault fault-code="{http://citrusframework.org/faults}TEC-1001"
           fault-string="Invalid request">
           fault-actor="SERVER">
  <ws:when>
    <send endpoint="soapClient">
        <message>
            <payload>
                <SoapFaultForcingRequest
                  xmlns="http://citrusframework.org/schemas/soap">
                    <Message>This is invalid</Message>
                </SoapFaultForcingRequest>
            </payload>
        </message>
    </send>
  </ws:when>
</ws:assert-fault>
```

The special assert action offers several attributes to validate the expected SOAP fault. Namely these are **"fault-code"**, **"fault-string"** and **"fault-actor"** . The **fault-code** is defined as a QName string and is mandatory for the validation. The fault assertion also supports test variable replacement as usual.

The time you use SOAP fault validation you need to tell Citrus how to validate the SOAP faults. Citrus needs an instance of a ***SoapFaultValidator*** that we need to add to the Spring application context. By default Citrus is searching for a bean with the id **'soapFaultValidator'** .

```
<bean id="soapFaultValidator"
class="org.citrusframework.ws.validation.SimpleSoapAttachmentValidator"/>
```

Citrus offers several reference implementations for these SOAP fault validators. These are:

- org.citrusframework.ws.validation.SimpleSoapAttachmentValidator
- org.citrusframework.ws.validation.SimpleSoapFaultValidator
- org.citrusframework.ws.validation.XmlSoapFaultValidator

Please see the API documentation for details on the available reference implementations. Of course you can also define your own SOAP validator logic (would be great if you could share your ideas!). In the test case you can explicitly choose the validator to use:

XML DSL

```
<ws:assert-fault fault-code="{http://citrusframework.org/faults}TEC-1001"
              fault-string="Invalid request"
              fault-validator="mySpecialSoapFaultValidator">
   [...]
</ws:assert-fault>
```

> Another important thing to notice when asserting SOAP faults is the fact, that Citrus needs to have a ***SoapMessageFactory*** available in the Spring application context. If you deal with SOAP messaging in general you will already have such a bean in the context.

```
<bean id="messageFactory"
class="org.springframework.ws.soap.saaj.SaajSoapMessageFactory"/>
```

Choose one of Spring's reference implementations or some other implementation as SOAP message factory. Citrus will search for a bean with id **'messageFactory'** by default. In case you have other beans with different identifiers please choose the messageFactory in the test case assert action:

XML DSL

```
<ws:assert-fault fault-code="{http://citrusframework.org/faults}TEC-1001"
              fault-string="Invalid request"
              message-factory="mySpecialMessageFactory">
   [...]
</ws:assert-fault>
```

> Notice the ws specific namespace that belongs to the Citrus WebService extensions. As the **ws:assert** action uses SOAP specific features we need to refer to the citrus-ws namespace. You can find the namespace declaration in the root element in your test case.

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
    xmlns:spring="http://www.springframework.org/schema/beans"
    xmlns:ws="http://www.citrusframework.org/schema/ws/testcase"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.citrusframework.org/schema/testcase
    http://www.citrusframework.org/schema/testcase/citrus-testcase.xsd
    http://www.citrusframework.org/schema/ws/testcase
    http://www.citrusframework.org/schema/ws/testcase/citrus-ws-testcase.xsd">
```

Citrus is also able to validate SOAP fault details. See the following example for understanding how to do it:

XML DSL

```
<ws:assert-fault fault-code="{http://citrusframework.org/faults}TEC-1001"
           fault-string="Invalid request">
    <ws:fault-detail>
      <![CDATA[
          <FaultDetail xmlns="http://citrusframework.org/schemas/soap">
              <ErrorCode>TEC-1000</ErrorCode>
              <Text>Invalid request</Text>
          </FaultDetail>
      ]]>
    </ws:fault-detail>
    <ws:when>
        <send endpoint="soapClient">
            <message>
                <payload>
                    <SoapFaultForcingRequest
                      xmlns="http://citrusframework.org/schemas/soap">
                        <Message>This is invalid</Message>
                    </SoapFaultForcingRequest>
                </payload>
            </message>
        </send>
    </ws:when>
</ws:assert-fault>
```

The expected SOAP fault detail content is simply added to the **ws:assert** action. The ***SoapFaultValidator*** implementation defined in the Spring application context is responsible for checking the SOAP fault detail with validation algorithm. The validator implementation checks the detail content to meet the expected template. Citrus provides some default ***SoapFaultValidator*** implementations. Supported algorithms are pure String comparison (**org.citrusframework.ws.validation.SimpleSoapFaultValidator**) as well as XML tree walk-through (**org.citrusframework.ws.validation.XmlSoapFaultValidator**).

When using the XML validation algorithm you have the complete power as known from normal message validation in receive actions. This includes schema validation or ignoring elements for instance. On the fault-detail element you are able to add some validation settings such as **schema-validation=enabled/disabled**, custom **schema-repository** and so on.

XML DSL

```
<ws:assert-fault fault-code="{http://citrusframework.org/faults}TEC-1001"
           fault-string="Invalid request">
    <ws:fault-detail schema-validation="false">
      <![CDATA[
          <FaultDetail xmlns="http://citrusframework.org/schemas/soap">
              <ErrorCode>TEC-1000</ErrorCode>
              <Text>Invalid request</Text>
          </FaultDetail>
      ]]>
    </ws:fault-detail>
    <ws:when>
        <send endpoint="soapClient">
            [...]
        </send>
    </ws:when>
</ws:assert-fault>
```

Please see also the Citrus API documentation for available validator implementations and validation algorithms.

So far we have used assert action wrapper in order to catch SOAP fault exceptions and validate the SOAP fault content. Now we have an alternative way of handling SOAP faults in Citrus. With exceptions the send action aborts and we do not have a receive action for the SOAP fault. This might be inadequate if we need to validate the SOAP message content (SOAPHeader and SOAPBody) coming with the SOAP fault. Therefore the web service message sender component offers several fault strategy options. In the following we discuss the propagation of SOAP fault as messages to the receive action as we would do with normal SOAP messages.

```
<citrus-ws:client id="soapClient"
                  request-url="http://localhost:8090/test"
                  fault-strategy="propagateError"/>
```

We have configured a fault strategy **propagateError** so the message sender will not raise client exceptions but inform the receive action with SOAP fault message contents. By default the fault strategy raises client exceptions (fault-strategy= **throwsException**).

So now that we do not raise exceptions we can leave out the assert action wrapper in our test. Instead we simply use a receive action and validate the SOAP fault like this.

```
<send endpoint="soapClient">
    <message>
        <payload>
            <SoapFaultForcingRequest
xmlns="http://citrusframework.org/schemas/sample.xsd">
                <Message>This is invalid</Message>
            </SoapFaultForcingRequest>
        </payload>
    </message>
</send>

<receive endpoint="soapClient" timeout="5000">
    <message>
        <payload>
            <SOAP-ENV:Fault xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/">
                <faultcode
xmlns:CITRUS="http://citrus.org/soap">CITRUS:${soapFaultCode}</faultcode>
                <faultstring xml:lang="en">${soapFaultString}</faultstring>
            </SOAP-ENV:Fault>
        </payload>
    </message>
</receive>
```

So choose the preferred way of handling SOAP faults either by asserting client exceptions or propagating fault messages to the receive action on a SOAP client.

15.13. Multiple SOAP fault details

SOAP fault messages can hold multiple SOAP fault detail elements. In the previous sections we have used SOAP fault details in sending and receiving actions as single element. In order to meet the SOAP specification Citrus is also able to handle multiple SOAP fault detail elements in a message. You just use multiple fault-detail elements in your test action like this:

```
<ws:send-fault endpoint="helloSoapServer">
    <ws:fault>
        <ws:fault-code>{http://citrusframework.org/faults}citrus:TEC-1000</ws:fault-
code>
        <ws:fault-string>Invalid request</ws:fault-string>
        <ws:fault-actor>SERVER</ws:fault-actor>
        <ws:fault-detail>
            <![CDATA[
                <FaultDetail xmlns="http://citrusframework.org/schemas/sayHello.xsd">
                    <MessageId>${messageId}</MessageId>
                    <CorrelationId>${correlationId}</CorrelationId>
                    <ErrorCode>TEC-1000</ErrorCode>
                    <Text>Invalid request</Text>
                </FaultDetail>
            ]]>
        </ws:fault-detail>
        <ws:fault-detail>
            <![CDATA[
                <ErrorDetail xmlns="http://citrusframework.org/schemas/sayHello.xsd">
                    <ErrorCode>TEC-1000</ErrorCode>
                </ErrorDetail>
            ]]>
        </ws:fault-detail>
    </ws:fault>
    <ws:header>
        <ws:element name="citrus_soap_action" value="sayHello"/>
    </ws:header>
</ws:send-fault>
```

This will result in following SOAP envelope message:

```
HTTP/1.1 500 Internal Server Error
Accept: text/xml, text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
SOAPAction: "sayHello"
Content-Type: text/xml; charset=utf-8
Content-Length: 680
Server: Jetty(7.0.0.pre5)

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header/>
    <SOAP-ENV:Body>
        <SOAP-ENV:Fault>
            <faultcode xmlns:citrus="http://citrusframework.org/faults">citrus:TEC-
1000</faultcode>
            <faultstring xml:lang="en">Invalid request</faultstring>
            <detail>
                <FaultDetail xmlns="http://citrusframework.org/schemas/sayHello.xsd">
                    <MessageId>9277832563</MessageId>
                    <CorrelationId>4346806225</CorrelationId>
                    <ErrorCode>TEC-1000</ErrorCode>
                    <Text>Invalid request</Text>
                </FaultDetail>
                <ErrorDetail xmlns="http://citrusframework.org/schemas/sayHello.xsd">
                    <ErrorCode>TEC-1000</ErrorCode>
                </ErrorDetail>
            </detail>
        </SOAP-ENV:Fault>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Of course we can also expect several fault detail elements when receiving a SOAP fault.

XML DSL

```
<ws:assert-fault fault-code="{http://citrusframework.org/faults}TEC-1001"
           fault-string="Invalid request">
    <ws:fault-detail schema-validation="false">
      <![CDATA[
          <FaultDetail xmlns="http://citrusframework.org/schemas/soap">
              <ErrorCode>TEC-1000</ErrorCode>
              <Text>Invalid request</Text>
          </FaultDetail>
      ]]>
    </ws:fault-detail>
    <ws:fault-detail>
      <![CDATA[
          <ErrorDetail xmlns="http://citrusframework.org/schemas/soap">
              <ErrorCode>TEC-1000</ErrorCode>
          </ErrorDetail>
      ]]>
    </ws:fault-detail>
    <ws:when>
        <send endpoint="soapClient">
            [...]
        </send>
    </ws:when>
</ws:assert-fault>
```

As you can see we can individually use validation settings for each fault detail. In the example above we disabled schema validation for the first fault detail element.

15.14. Send HTTP error codes with SOAP

The SOAP server logic in Citrus is able to simulate pure HTTP error codes such as 404 "Not found" or 500 "Internal server error". The good thing is that the Citrus server is able to receive a request for proper validation in a receive action and then simulate HTTP errors on demand.

The mechanism on HTTP error code simulation is not different to the usual SOAP request/response handling in Citrus. We receive the request as usual and we provide a response. The HTTP error situation is simulated according to the special HTTP header **citrus_http_status** in the Citrus SOAP response definition. In case this header is set to a value other than 200 OK the Citrus SOAP server sends an empty SOAP response with HTTP error status code set accordingly.

```
<receive endpoint="helloSoapServer">
  <message>
      <payload>
          <Message xmlns="http://citrusframework.org/schemas/sample.xsd">
              <Text>Hello SOAP server</Text>
          </Message>
      </payload>
  </message>
</receive>

<send endpoint="helloSoapServer">
    <message>
        <data></data>
    </message>
    <header>
        <element name="citrus_http_status_code" value="500"/>
    </header>
</send>
```

The SOAP response must be empty and the HTTP status code is set to a value other than 200, like 500. This results in an HTTP error sent to the calling client with error 500 "Internal server error".

15.15. SOAP attachment support

Citrus is able to add attachments to a SOAP request on client and server side. As usual you can validate the SOAP attachment content on a received SOAP message. The next chapters describe how to handle SOAP attachments in Citrus.

15.16. Send SOAP attachments

As client Citrus is able to add attachments to the SOAP message. I think it is best to go straight into an example in order to understand how it works.

```
<ws:send endpoint="soapClient">
    <message>
        <payload>
            <SoapMessageWithAttachment
xmlns="http://citrusframework.org/schemas/sample.xsd">
                <Operation>Read the attachment</Operation>
            </SoapMessageWithAttachment>
        </payload>
    </message>
    <ws:attachment content-id="MySoapAttachment" content-type="text/plain">
        <ws:resource file="classpath:org/citrusframework/ws/soapAttachment.txt"/>
    </ws:attachment>
</ws:send>
```

In the previous chapters you may have already noticed the **citrus-ws** namespace that stands for the SOAP extensions in Citrus. Please include the **citrus-ws** namespace in your test case as described earlier in this chapter so you can use the attachment support.

The special send action of the SOAP extension namespace is aware of SOAP attachments. The attachment content usually consists of a **content-id**, a **content-type** and the actual content as plain text or binary content. Inside the test case you can use external file resources or inline CDATA sections for the attachment content. As you are familiar with Citrus you may know this already from other actions.

Citrus will construct a SOAP message with the SOAP attachment. Currently only one attachment per message is supported.

15.17. Receive SOAP attachments

When Citrus calls SOAP WebServices as a client we may receive SOAP responses with attachments. The tester can validate those received SOAP messages with attachment content quite easy. As usual let us have a look at an example first.

```
<ws:receive endpoint="soapClient">
    <message>
        <payload>
            <SoapMessageWithAttachmentRequest
xmlns="http://citrusframework.org/schemas/sample.xsd">
                <Operation>Read the attachment</Operation>
            </SoapMessageWithAttachmentRequest>
        </payload>
    </message>
    <ws:attachment content-id="MySoapAttachment"
                   content-type="text/plain"
                   validator="mySoapAttachmentValidator">
        <ws:resource file="classpath:org/citrusframework/ws/soapAttachment.txt"/>
    </ws:attachment>
</ws:receive>
```

Again we use the Citrus SOAP extension namespace with the specific receive action that is aware of SOAP attachment validation. The tester can validate the **content-id**, the **content-type** and the attachment content. Instead of using the external file resource you could also define an expected attachment template directly in the test case as inline CDATA section.

The **ws:attachment** element specifies a validator instance. This validator determines how to validate the attachment content. SOAP attachments are not limited to XML content. Plain text content and binary content is possible, too. So each SOAP attachment validating action can use a different ***SoapAttachmentValidator*** instance which is responsible for validating and comparing received attachments to expected template attachments. In the Citrus configuration the validator is set as normal Spring bean with the respective identifier.

```
<bean id="soapAttachmentValidator"
class="org.citrusframework.ws.validation.SimpleSoapAttachmentValidator"/>
<bean id="mySoapAttachmentValidator"
class="com.company.ws.validation.MySoapAttachmentValidator"/>
```

You can define several validator instances in the Citrus configuration. The validator with the general id **"soapAttachmentValidator"** is the default validator for all actions that do not explicitly set a validator instance. Citrus offers a set of reference validator implementations. The ***SimpleSoapAttachmentValidator*** will use a simple plain text comparison. Of course you are able to add individual validator implementations, too.

15.18. SOAP MTOM support

MTOM (Message Transmission Optimization Mechanism) enables you to send and receive large SOAP message content using streamed data handlers. This optimizes the resource allocation on server and client side where not all data is loaded into memory when marshalling/unmarshalling the message payload data. In detail MTOM-enabled messages do have a XOP package inside the message payload replacing the actual large content data. The content is then streamed aas separate attachment. Server and client can operate with a data handler providing access to the streamed content. This is very helpful when using large binary content inside a SOAP message for instance.

Citrus is able to both send and receive MTOM enabled SOAP messages on client and server. Just use the **mtom-enabled** flag when sending a SOAP message:

```
<ws:send endpoint="soapMtomClient" mtom-enabled="true">
  <message>
    <data>
      <![CDATA[
        <image:addImage xmlns:image="http://citrusframework.org/imageService/">
          <image>cid:IMAGE</image>
        </image:addImage>
      ]]>
    </data>
  </message>
  <ws:attachment content-id="IMAGE" content-type="application/octet-stream">
    <ws:resource file="classpath:org/citrusframework/hugeImageData.png"/>
  </ws:attachment>
</ws:send>
```

As you can see the example above sends a SOAP message that contains a large binary image content. The actual binary image data is referenced with a content id marker **cid:IMAGE** inside the message payload. The actual image content is added as attachment with a separate file resource. Important is here the **content-id** which matches the id marker in the SOAP message payload (**IMAGE**).

Citrus builds a proper SOAP MTOM enabled message automatically adding the XOP package inside the message. The binary data is sent as separate SOAP attachment accordingly. The resulting SOAP message looks like this:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header></SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <image:addImage xmlns:image="http://citrusframework.org/imageService/">
      <image><xop:Include xmlns:xop="http://www.w3.org/2004/08/xop/include"
href="cid:IMAGE"/></image>
    </image:addImage>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

On the server side Citrus is also able to handle MTOM enabled SOAP messages. In a server receive action you can specify the MTOM SOAP attachment content as follows.

```
<ws:receive endpoint="soapMtomServer" mtom-enabled="true">
  <message schema-validation="false">
    <data>
      <![CDATA[
        <image:addImage xmlns:image="http://citrusframework.org/imageService/">
          <image><xop:Include xmlns:xop="http://www.w3.org/2004/08/xop/include"
href="cid:IMAGE"/></image>
        </image:addImage>
      ]]>
    </data>
  </message>
  <ws:attachment content-id="IMAGE" content-type="application/octet-stream">
    <ws:resource file="classpath:org/citrusframework/hugeImageData.png"/>
  </ws:attachment>
</ws:receive>
```

We define the MTOM attachment content as separate SOAP attachment. The **content-id** is referenced somewhere in the SOAP message payload data. At runtime Citrus will add the XOP package definition automatically and perform validation on the message and its streamed MTOM attachment data.

Next thing that we have to talk about is inline MTOM data. This means that the content should be added as either **base64Binary** or **hexBinary** encoded String data directly to the message content. See the following example that uses the **mtom-inline** setting:

```
<ws:send endpoint="soapMtomClient" mtom-enabled="true">
  <message>
    <data>
      <![CDATA[
        <image:addImage xmlns:image="http://citrusframework.org/imageService/">
          <image>cid:IMAGE</image>
          <icon>cid:ICON</icon>
        </image:addImage>
      ]]>
    </data>
  </message>
  <ws:attachment content-id="IMAGE" content-type="application/octet-stream"
            mtom-inline="true" encoding-type="base64Binary">
    <ws:resource file="classpath:org/citrusframework/image.png"/>
  </ws:attachment>
  <ws:attachment content-id="ICON" content-type="application/octet-stream"
            mtom-inline="true" encoding-type="hexBinary">
    <ws:resource file="classpath:org/citrusframework/icon.ico"/>
  </ws:attachment>
</ws:send>
```

The listing above defines two inline MTOM attachments. The first attachment **cid:IMAGE** uses the encoding type **base64Binary** which is the default. The second attachment **cid:ICON** uses

hexBinary encoding. Both attachments are added as inline data before the message is sent. The final SOAP message looks like follows:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header></SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <image:addImage xmlns:image="http://citrusframework.org/imageService/">

<image>VGhpcyBpcyBhIGJpbmFyeSBpbWFnZSBhdHRhY2htZW50IQpWYXJpYWJsZXMgJXt0ZXN0fSBzaG91bGQ
gbm90IGJlIHJlcGxhY2VkIQ==</image>

<icon>546869732069732061206269 6E6172792069636F6E20617474616368 6D656E74210A566172696162
6C657320257B746573747D2073686F756C64206E6F74206265207265706C6163656421</icon>
    </image:addImage>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The image content is a base64Binary String and the icon a hexBinary String. Of course this mechanism also is supported in receive actions on the server side where the expected message content is added as inline MTOM data before validation takes place.

15.19. SOAP client basic authentication

As a SOAP client you may have to use basic authentication in order to access a server resource. Basic authentication via HTTP stands for username/password authentication where the credentials are transmitted in the HTTP request header section as base64 encoded entry. As Citrus uses the Spring WebService stack we can use the basic authentication support there. We set the user credentials on the HttpClient message sender which is used inside the Spring ***WebServiceTemplate*** .

Citrus provides a comfortable way to set the HTTP message sender with basic authentication credentials on the ***WebServiceTemplate*** . Just see the following example and learn how to do that.

```
<citrus-ws:client id="soapClient"
                              request-url="http://localhost:8090/test"
                              message-sender="basicAuthClient"/>

<bean id="basicAuthClient"
class="org.springframework.ws.transport.http.HttpComponentsMessageSender">
  <property name="authScope">
      <bean class="org.apache.hc.client5.http.auth.AuthScope">
        <constructor-arg value="http"/>
        <constructor-arg value="localhost"/>
        <constructor-arg value="8090"/>
        <constructor-arg>
          <null/>
        </constructor-arg>
        <constructor-arg value="basic"/>
      </bean>
  </property>
  <property name="credentials">
    <bean class="org.apache.hc.client5.http.auth.UsernamePasswordCredentials">
        <constructor-arg value="someUsername"/>
        <constructor-arg value="somePassword"/>
    </bean>
  </property>
</bean>
```

The above configuration results in SOAP requests with authentication headers properly set for basic authentication. The special message sender takes care of adding the proper basic authentication header to each request that is sent with this Citrus message sender. By default preemptive authentication is used. The message sender only sends a single request to the server with all authentication information set in the message header. The request which determines the authentication scheme on the server is skipped. This is why you have to add some auth scope so Citrus can setup an authentication cache within the HTTP context in order to have preemptive authentication.

> You can also skip the message sender configuration and set the **Authorization** header on each request in your send action definition on your own. Be aware of setting the header as HTTP mime header using the correct prefix and take care of using the correct basic authentication with base64 encoding for the **username:password** phrase.

```
<header>
    <element name="citrus_http_Authorization" value="Basic
c29tZVVzZXJuYW1lOnNvbWVQYXNzd29yZA=="/>
</header>
```

For base64 encoding you can also use a Citrus function, see functions-encode-base64

15.20. SOAP server basic authentication

When providing SOAP WebService server functionality Citrus can also set basic authentication so all clients need to authenticate properly when accessing the server resource.

```
<citrus-ws:server id="simpleSoapServer"
             port="8080"
             auto-start="true"
             resource-base="src/it/resources"
             security-handler="basicSecurityHandler"/>

<bean id="securityHandler"
class="org.citrusframework.ws.security.SecurityHandlerFactory">
    <property name="users">
        <list>
            <bean class="org.citrusframework.ws.security.User">
                <property name="name" value="citrus"/>
                <property name="password" value="secret"/>
                <property name="roles" value="CitrusRole"/>
            </bean>
        </list>
    </property>
    <property name="constraints">
        <map>
            <entry key="/foo/*">
                <bean class="org.citrusframework.ws.security.BasicAuthConstraint">
                    <constructor-arg value="CitrusRole"/>
                </bean>
            </entry>
        </map>
    </property>
</bean>
```

We have set a security handler on the server web container with a constraint on all resources with `/foo/*`. Following from that the server requires basic authentication for these resources. The granted users and roles are specified within the security handler bean definition. Connecting clients have to set the basic auth HTTP header properly using the correct user and role for accessing the Citrus server now.

You can customize the security handler for your very specific needs (e.g. load users and roles with JDBC from a database). Just have a look at the code base and inspect the settings and properties offered by the security handler interface.

This mechanism is not restricted to basic authentication only. With other settings you can also set up digest or form-based authentication constraints very easy.

15.21. WS-Addressing support

The web service stack offers a lot of different technologies and standards within the context of SOAP WebServices. We speak of WS-* specifications in particular. One of these specifications deals with addressing. On client side you may add wsa header information to the request in order to give the server instructions how to deal with SOAP faults for instance.

In Citrus WebService client you can add those header information using the common configuration like this:

```
<citrus-ws:client id="soapClient"
                          request-url="http://localhost:8090/test"
                          message-converter="wsAddressingMessageConverter"/>

<bean id="wsAddressingMessageConverter"
class="org.citrusframework.ws.message.converter.WsAddressingMessageConverter">
  <constructor-arg>
    <bean id="wsAddressing200408"
class="org.citrusframework.ws.addressing.WsAddressingHeaders">
        <property name="version" value="VERSION200408"/>
        <property name="action" value="http://citrus.sample/sayHello"/>
        <property name="to" value="http://citrus.sample/server"/>
        <property name="from">
            <bean
class="org.springframework.ws.soap.addressing.core.EndpointReference">
                <constructor-arg value="http://citrus.sample/client"/>
            </bean>
        </property>
        <property name="replyTo">
            <bean
class="org.springframework.ws.soap.addressing.core.EndpointReference">
                <constructor-arg value="http://citrus.sample/client"/>
            </bean>
        </property>
        <property name="faultTo">
            <bean
class="org.springframework.ws.soap.addressing.core.EndpointReference">
                <constructor-arg value="http://citrus.sample/fault/resolver"/>
            </bean>
        </property>
    </bean>
  </constructor-arg>
</bean>
```

The WsAddressing header values will be used for all request messages that are sent with the soap client component *soapClient*. You can overwrite the WsAddressing header in each send test action in your test though. Just set the special WsAddressing message header on your request. You can use the following message header names in order to overwrite the default addressing headers specified in the message converter configuration (also see the class

org.citrusframework.ws.addressing.WsAddressingMessageHeaders).

citrus_soap_ws_addressing_messageId	addressing message id as URI
citrus_soap_ws_addressing_from	addressing from endpoint reference as URI
citrus_soap_ws_addressing_to	addressing to URI
citrus_soap_ws_addressing_action	addressing action URI
citrus_soap_ws_addressing_replyTo	addressing reply to endpoint reference as URI
citrus_soap_ws_addressing_faultTo	addressing fault to endpoint reference as URI

When using these message headers you are able to explicitly overwrite the WsAddressing headers. Test variables are supported of course when specifying the values. Most of the values are parsed to a URI value at the end so please make sure to use correct URI String representations.

The WS-Addressing specification knows several versions. Supported versions are:

VERSION10	WS-Addressing 1.0 May 2006
VERSION200408	August 2004 edition of the WS-Addressing specification

The addressing headers find a place in the SOAP message header with respective namespaces and values. A possible SOAP request with WS addressing headers looks like follows:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing">
        <wsa:To SOAP-ENV:mustUnderstand="1">http://citrus.sample/server</wsa:To>
        <wsa:From>
            <wsa:Address>http://citrus.sample/client</wsa:Address>
        </wsa:From>
        <wsa:ReplyTo>
            <wsa:Address>http://citrus.sample/client</wsa:Address>
        </wsa:ReplyTo>
        <wsa:FaultTo>
            <wsa:Address>http://citrus.sample/fault/resolver</wsa:Address>
        </wsa:FaultTo>
        <wsa:Action>http://citrus.sample/sayHello</wsa:Action>
        <wsa:MessageID>urn:uuid:4c4d8af2-b402-4bc0-a2e3-ad33b910e394</wsa:MessageID>
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
        <cit:HelloRequest xmlns:cit="http://citrus/sample/sayHello">
            <cit:Text>Hello Citrus!</cit:Text>
        </cit:HelloRequest>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

By default when not set explicitly on the message headers the WsAddressing message id property is automatically generated for each request. You can set the message id generation strategy in the Spring application context message converter configuration:

```
<bean id="wsAddressingMessageConverter"
class="org.citrusframework.ws.message.converter.WsAddressingMessageConverter">
  <property name="messageIdStrategy">
    <bean
class="org.springframework.ws.soap.addressing.messageid.UuidMessageIdStrategy"/>
  </property>
</bean>
```

By default the strategy will create a new Java UUID for each request. The strategy also uses a common resource name prefix *urn:uuid:*. You can overwrite the message id any time for each request explicitly by setting the message header *citrus_soap_ws_addressing_messageId* with a respective value on the message in your test.

15.22. SOAP client fork mode

SOAP over HTTP uses synchronous communication by nature. This means that sending a SOAP message in Citrus over HTTP will automatically block further test actions until the synchronous HTTP response has been received. In test cases this synchronous blocking might cause problems for several reasons. A simple reason would be that you need to do further test actions in parallel to the synchronous HTTP SOAP communication (e.g. simulate another backend system in the test case).

You can separate the SOAP send action from the rest of the test case by using the **"fork"** mode. The SOAP client will automatically open a new Java Thread for the synchronous communication and the test is able to continue with execution although the synchronous HTTP SOAP response has not arrived yet.

```
<ws:send endpoint="soapClient" fork="true">
  <message>
    <payload>
        <SoapRequest xmlns="http://citrusframework.org/schemas/sample.xsd">
          <Operation>Read the attachment</Operation>
        </SoapRequest>
    </payload>
  </message>
</ws:send>
```

With the **"fork"** mode enabled the test continues with execution while the sending action waits for the synchronous response in a separate Java Thread. You could reach the same behaviour with a complex <parallel>/<sequential> container construct, but forking the send action is much more straight forward.

> It is highly recommended to use a proper **"timeout"** setting on the SOAP receive action when using fork mode. The forked send operation might take some time and the corresponding receive action might run into failure as the response was has not been received yet. The result would be a broken test because of the missing response message. A proper **"timeout"** setting for the receive action solves this problem as the action waits for this time period and occasionally repeatedly asks for the SOAP response message. The following listing sets the receive timeout to 10 seconds, so the action waits for the forked send action to deliver the SOAP response in time.

```
<ws:receive endpoint="soapClient" timeout="10000">
  <message>
    <payload>
        <SoapResponse xmlns="http://citrusframework.org/schemas/sample.xsd">
          <Operation>Did something</Operation>
          <Success>true</Success>
        </SoapResponse>
    </payload>
  </message>
</ws:receive>
```

15.23. SOAP servlet context customization

For highly customized SOAP server components in Citrus you can define a full servlet context configuration file. Here you have the full power to add Spring endpoint mappings and custom endpoint implementations. You can set the custom servlet context as external file resource on the

server component:

```
<citrus-ws:client id="soapClient"
          context-config-location="classpath:citrus-ws-servlet.xml"
          message-factory="soap11MessageFactory"/>
```

Now let us have a closer look at the context-config-location attribute. This configuration defines the Spring application context file for endpoints, request mappings and other SpringWS specific information. Please see the official SpringWS documentation for details on this Spring based configuration. You can also just copy the following example application context which should work for you in general.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="loggingInterceptor"

class="org.springframework.ws.server.endpoint.interceptor.PayloadLoggingInterceptor">
      <description>
          This interceptor logs the message payload.
      </description>
  </bean>

  <bean id="helloServicePayloadMapping"

class="org.springframework.ws.server.endpoint.mapping.PayloadRootQNameEndpointMapping"
>
      <property name="mappings">
          <props>
              <prop
                  key="{http://citrusframework.org/schemas/sayHello}HelloRequest">
                  helloServiceEndpoint
              </prop>
          </props>
      </property>
      <property name="interceptors">
          <list>
              <ref bean="loggingInterceptor"/>
          </list>
      </property>
  </bean>

  <bean id="helloServiceEndpoint"
class="org.citrusframework.ws.server.WebServiceEndpoint">
      <property name="endpointAdapter" ref="staticResponseEndpointAdapter"/>
  </bean>
```

```
  <citrus:static-response-adapter id="staticResponseEndpointAdapter">
      <citrus:payload>
          <![CDATA[
              <HelloResponse xmlns="http://citrusframework.org/schemas/sayHello">
                  <MessageId>123456789</MessageId>
                  <CorrelationId>CORR123456789</CorrelationId>
                  <User>WebServer</User>
                  <Text>Hello User</Text>
              </HelloResponse>
          ]]>
      </citrus:payload>
      <citrus:header>
          <citrus:element
name="{http://citrusframework.org/schemas/samples/sayHello.xsd}ns0:Operation"
                  value="sayHelloResponse"/>
          <citrus:element
name="{http://citrusframework.org/schemas/samples/sayHello.xsd}ns0:Request"
                  value="HelloRequest"/>
          <citrus:element name="citrus_soap_action"
                  value="sayHello"/>
      </citrus:header>
  </citrus:static-response-adapter>
</beans>
```

The program listing above describes a normal SpringWS request mapping with endpoint configurations. The mapping is responsible to forward incoming requests to the endpoint which will handle the request and provide a proper response message. First of all we add a logging interceptor to the context so all incoming requests get logged to the console first. Then we use a payload mapping (PayloadRootQNameEndpointMapping) in order to map all incoming ***'HelloRequest'*** SOAP messages to the ***'helloServiceEndpoint'*** . Endpoints are of essential nature in Citrus SOAP WebServices implementation. They are responsible for processing a request in order to provide a proper response message that is sent back to the calling client. Citrus uses the endpoint in combination with a message endpoint adapter implementation.

The endpoint works together with the message endpoint adapter that is responsible for providing a response message for the client. The various message endpoint adapter implementations in Citrus were already discussed in endpoint-adapter.

In this example the ***'helloServiceEndpoint'*** uses the ***'static-response-adapter'*** which is always returning a static response message. In most cases static responses will not fit the test scenario and you will have to respond more dynamically.

Regardless of which message endpoint adapter setup you are using in your test case the endpoint transforms the response into a proper SOAP message. You can add as many request mappings and

endpoints as you want to the server context configuration. So you are able to handle different request types with one single Jetty server instance.

That's it for connecting with SOAP WebServices! We saw how to send and receive SOAP messages with Jetty and Spring WebServices. Have a look at the samples coming with your Citrus archive in order to learn more about the SOAP message handling.

Chapter 16. Apache Camel support

Apache Camel is a fantastic Open Source integration framework that empowers you to quickly and easily integrate various systems consuming or producing data.

Camel implements the enterprise integration patterns for building mediation and routing rules in your software. With the Camel support in Citrus you are able to directly interact with the 300+ Camel components through route definitions. You can call Camel routes and receive synchronous response messages from a Citrus test. You can also simulate the Camel route endpoint with receiving messages and providing simulated response messages.

The camel components in Citrus are located in a separate Maven module. So you should add the module as Maven dependency to your project accordingly.

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-camel</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

For XML configurations Citrus provides a special Camel configuration schema that is used in Spring configuration files. You have to include the `citrus-camel` namespace in your Spring configuration XML files as follows.

XML

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:citrus="http://www.citrusframework.org/schema/config"
      xmlns:citrus-camel="http://www.citrusframework.org/schema/camel/config"
      xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.citrusframework.org/schema/config
      http://www.citrusframework.org/schema/config/citrus-config.xsd
      http://www.citrusframework.org/schema/camel/config
      http://www.citrusframework.org/schema/camel/config/citrus-camel-config.xsd">

      [...]

      </beans>
```

Now you are ready to use the Camel configuration elements using the citrus-camel namespace prefix.

The next sections explain the Citrus capabilities while working with Apache Camel.

16.1. Camel endpoint

Camel and Citrus both use the endpoint pattern in order to define message destinations. Users can interact with these endpoints when creating the mediation and routing logic. The Citrus endpoint component for Camel interaction is defined as follows in your Citrus Spring configuration.

Java

```
@Bean
public CamelEndpoint helloCamelEndpoint() {
    return new CamelEndpointBuilder()
        .endpointUri("direct:hello")
        .build();
}
```

XML

```
<citrus-camel:endpoint id="helloCamelEndpoint"
      endpoint-uri="direct:hello"/>
```

The endpoint defines an endpoint uri that will be called when messages are sent/received using the endpoint producer or consumer. The endpoint uri refers to a Camel route that is located inside a Camel context. The Camel route should use the respective endpoint uri as `from` definition.

Java

```
@Bean
public CamelContext camelContext() {
    CamelContext context = new DefaultCamelContext();

    context.addRoutes(new RouteBuilder() {
        @Override
        public void configure() throws Exception {
            from("direct:hello")
                .routeId("helloRoute")
                .to("log:org.citrusframework.camel?level=INFO")
                .to("seda:greetings");
        }
    });

    return context;
}
```

XML

```
<camelContext id="camelContext" xmlns="http://camel.apache.org/schema/spring">
  <route id="helloRoute">
    <from uri="direct:hello"/>
    <to uri="log:org.citrusframework.camel?level=INFO"/>
    <to uri="seda:greetings-feed"/>
  </route>
</camelContext>
```

In the example above the Camel context is placed as Spring bean. The context defines a Camel route the Citrus test case can interact with.

The Camel context is automatically referenced in the Citrus Camel endpoint. This is because Citrus will automatically look for a Camel context in the Spring bean configuration.

In case you have multiple Camel context instances in your configuration you can explicitly link the endpoint to a context with `camel-context="camelContext"`.

Java

```
@Bean
public CamelEndpoint helloCamelEndpoint() {
    return new CamelEndpointBuilder()
        .camelContext(specialCamelContext)
        .endpointUri("direct:hello")
        .build();
}
```

XML

```
<citrus-camel:endpoint id="helloCamelEndpoint"
      camel-contxt="specialCamelContext"
      endpoint-uri="direct:hello"/>
```

This explicitly binds the endpoint to the context named "*specialCamelContext*". This configuration would be the easiest setup to use Camel with Citrus as you can add the Camel context straight to the Spring bean application context and interact with it in Citrus. Of course, you can also import your Camel context and routes from other Spring bean context files, or you can start the Camel context routes with Java code.

In the example the Camel route is listening on the route endpoint uri `direct:hello`. Incoming messages will be logged to the console using a `log` Camel component. After that the message is forwarded to a `seda` Camel component which is a simple queue in memory.

The Citrus endpoint can interact with this sample route definition sending messages to the `direct:hello` endpoint. The endpoint configuration holds the endpoint uri information that tells Citrus how to access the Camel route. This endpoint uri can be any Camel endpoint uri that is used in a Camel route.

The Camel routes support asynchronous and synchronous message communication patterns. By default, Citrus uses asynchronous communication with Camel routes. This means that the Citrus producer sends the exchange message to the route endpoint uri and is finished immediately. There is no synchronous response to await. In contrary to that the synchronous endpoint will send and receive a synchronous message on the Camel destination route. This message exchange pattern is discussed in a later section in this chapter.

For now, we have a look on how to use the Citrus Camel endpoint in a test case in order to send a message to the Camel route:

Java

```
send(helloCamelEndpoint)
    .message()
    .body("Hello from Citrus!");
```

XML

```
<send endpoint="helloCamelEndpoint">
  <message type="plaintext">
    <payload>Hello from Citrus!</payload>
  </message>
</send>
```

You can use the very same Citrus Camel endpoint component to receive messages in your test case, too. In this situation you would receive a message from the route endpoint. This is especially designed for queueing endpoint routes such as the Camel seda component. In our example Camel route above the seda Camel component is called with the endpoint uri **seda:greetings-feed**.

This means that the Camel route is sending a message to the `seda` component. Citrus is able to receive this route message with an endpoint component like this:

Java

```
@Bean
public CamelEndpoint greetingsFeed() {
    return new CamelEndpointBuilder()
        .endpointUri("seda:greetings-feed")
        .build();
}
```

XML

```
<citrus-camel:endpoint id="greetingsFeed"
    endpoint-uri="seda:greetings-feed"/>
```

You can use the Citrus camel endpoint in your test case receive action in order to consume the message on the seda component.

Java

```
receive(greetingsFeed)
    .message()
    .type(MessageType.PLAINTEXT)
    .body("Hello from Citrus!");
```

XML

```
<receive endpoint="greetingsFeed">
  <message type="plaintext">
    <payload>Hello from Citrus!</payload>
  </message>
</receive>
```

> Instead of defining a static Citrus camel component you could also use the dynamic endpoint components in Citrus. This would enable you to send your message directly using the endpoint uri **direct:news** in your test case. Read more about this in dynamic-endpoint-components.

Citrus is able to send and receive messages with Camel route endpoint uri. This enables you to invoke a Camel route. The Camel components used is defined by the endpoint uri as usual. When interacting with Camel routes you might need to send back some response messages in order to simulate boundary applications. We will discuss the synchronous communication in the next section.

16.2. Synchronous Camel endpoint

The synchronous Camel producer sends a message to a route and waits synchronously for the response to arrive. In Camel this communication is represented with the exchange pattern **InOut**. The basic configuration for a synchronous Camel endpoint component looks like follows:

Java

```
@Bean
public CamelSyncEndpoint helloCamelEndpoint() {
    return new CamelSyncEndpointBuilder()
        .endpointUri("direct:hello")
        .timeout(1000L)
        .pollingInterval(300L)
        .build();
}
```

XML

```
<citrus-camel:sync-endpoint id="helloCamelEndpoint"
      endpoint-uri="direct:hello"
      timeout="1000"
      polling-interval="300"/>
```

Synchronous endpoints poll for synchronous reply messages to arrive. The poll interval is an optional setting in order to manage the amount of reply message handshake attempts. Once the endpoint was able to receive the reply message synchronously the test case can receive the reply. In case the reply message is not available in time we raise some timeout error and the test will fail.

In a first test scenario we write a test case that sends a message to the synchronous endpoint and waits for the synchronous reply message to arrive. So we have two actions on the same Citrus endpoint, first send then receive.

Java

```
send(helloCamelEndpoint)
    .message()
    .type(MessageType.PLAINTEXT)
    .body("Hello from Citrus!");

receive(helloCamelEndpoint)
    .message()
    .type(MessageType.PLAINTEXT)
    .body("This is the reply from Apache Camel!");
```

XML

```
<send endpoint="helloCamelEndpoint">
  <message type="plaintext">
    <payload>Hello from Citrus!</payload>
  </message>
</send>

<receive endpoint="helloCamelEndpoint">
  <message type="plaintext">
    <payload>This is the reply from Apache Camel!</payload>
  </message>
</receive>
```

The next variation deals with the same synchronous communication, but send and receive roles are switched. Now Citrus receives a message from a Camel route and has to provide a reply message. We handle this synchronous communication with the same synchronous Apache Camel endpoint component. Only difference is that we initially start the communication by receiving a message from the endpoint. Knowing this Citrus is able to send a synchronous response back. Again just use the same endpoint reference in your test case. So we have again two actions in our test case, but this time first receive then send.

Java

```
receive(helloCamelEndpoint)
    .message()
    .type(MessageType.PLAINTEXT)
    .body("Hello from Apache Camel!");

send(helloCamelEndpoint)
    .message()
    .type(MessageType.PLAINTEXT)
    .body("This is the reply from Citrus!");
```

XML

```
<receive endpoint="helloCamelEndpoint">
  <message type="plaintext">
    <payload>Hello from Apache Camel!</payload>
  </message>
</receive>

<send endpoint="helloCamelEndpoint">
  <message type="plaintext">
    <payload>This is the reply from Citrus!</payload>
  </message>
</send>
```

This is pretty simple. Citrus takes care of setting the Camel exchange pattern **InOut** while using synchronous communications. The Camel routes do respond and Citrus is able to receive the synchronous messages accordingly. With this pattern you can interact with Camel routes where Citrus simulates synchronous clients and consumers.

16.3. Camel exchange headers

Camel uses exchanges when sending and receiving messages to and from routes. These exchanges hold specific information on the communication outcome. Citrus automatically converts these exchange information to special message header entries. You can validate those exchange headers then easily in your test case:

Java

```
receive(greetingsFeed)
    .message()
    .type(MessageType.PLAINTEXT)
    .body("Hello from Camel!")
    .header("citrus_camel_route_id", "greetings")
    .header("citrus_camel_exchange_id", "ID-local-50532-1402653725341-0-3")
    .header("citrus_camel_exchange_failed", false)
    .header("citrus_camel_exchange_pattern", "InOnly")
    .header("CamelCorrelationId", "ID-local-50532-1402653725341-0-1")
    .header("CamelToEndpoint", "seda://greetings-feed");
```

XML

```
<receive endpoint="greetingsFeed">
  <message type="plaintext">
    <payload>Hello from Camel!</payload>
  </message>
  <header>
    <element name="citrus_camel_route_id" value="greetings"/>
    <element name="citrus_camel_exchange_id" value="ID-local-50532-1402653725341-0-
3"/>
    <element name="citrus_camel_exchange_failed" value="false"/>
    <element name="citrus_camel_exchange_pattern" value="InOnly"/>
    <element name="CamelCorrelationId" value="ID-local-50532-1402653725341-0-1"/>
    <element name="CamelToEndpoint" value="seda://greetings-feed"/>
  </header>
</receive>
```

In addition to the Camel specific exchange information the Camel exchange does also hold some custom properties. These properties such as **CamelToEndpoint** or **CamelCorrelationId** are also added automatically to the Citrus message header so can expect them in a `receive` message action.

16.4. Camel exception handling

Let us suppose the following route definition:

Java

```
@Bean
public CamelContext camelContext() {
    CamelContext context = new DefaultCamelContext();

    context.addRoutes(new RouteBuilder() {
        @Override
        public void configure() throws Exception {
            from("direct:hello")
                .routeId("helloRoute")
                .to("log:org.citrusframework.camel?level=INFO")
                .to("seda:greetings-feed")
                .onException(CitrusRuntimeException.class)
                    .to("seda:exceptions");
        }
    });

    return context;
}
```

XML

```
<camelContext id="camelContext" xmlns="http://camel.apache.org/schema/spring">
  <route id="helloRoute">
    <from uri="direct:hello"/>
    <to uri="log:org.citrusframework.camel?level=INFO"/>
    <to uri="seda:greetings-feed"/>
    <onException>
      <exception>org.citrusframework.exceptions.CitrusRuntimeException</exception>
      <to uri="seda:exceptions"/>
    </onException>
  </route>
</camelContext>
```

The route has an exception handling block defined that is called as soon as the exchange processing ends up in some error or exception. With Citrus you can also simulate an exchange exception when sending back a synchronous response to a calling route.

Java

```
send(helloCamelEndpoint)
    .message()
    .type(MessageType.PLAINTEXT)
    .body("Something went wrong!")
    .header("citrus_camel_exchange_exception", CitrusRuntimeException.class)
    .header("citrus_camel_exchange_exception_message", "Something went wrong!")
    .header("citrus_camel_exchange_failed", true);
```

XML

```
<send endpoint="greetingsFeed">
  <message type="plaintext">
    <payload>Something went wrong!</payload>
  </message>
  <header>
    <element name="citrus_camel_exchange_exception"
                value="org.citrusframework.exceptions.CitrusRuntimeException"/>
    <element name="citrus_camel_exchange_exception_message" value="Something went
wrong!"/>
    <element name="citrus_camel_exchange_failed" value="true"/>
  </header>
</send>
```

This message as response to the **seda:greetings-feed** route would cause Camel to enter the exception handling in the route definition. The exception handling is activated and calls the error handling route endpoint **seda:exceptions** . Of course Citrus would be able to receive such an exception exchange validating the exception handling outcome.

In such failure scenarios the Camel exchange holds the exception information (**CamelExceptionCaught**) such as causing exception class and error message. These headers are present in an error scenario and can be validated in Citrus when receiving error messages as follows:

Java

```
receive(errorCamelEndpoint)
    .message()
    .type(MessageType.PLAINTEXT)
    .body("Something went wrong!")
    .header("citrus_camel_route_id", "helloRoute")
    .header("citrus_camel_exchange_failed", true)
    .header("CamelExceptionCaught",
"org.citrusframework.exceptions.CitrusRuntimeException: Something went wrong!");
```

XML

```
<receive endpoint="errorCamelEndpoint">
  <message type="plaintext">
    <payload>Something went wrong!</payload>
  </message>
  <header>
    <element name="citrus_camel_route_id" value="helloRoute"/>
    <element name="citrus_camel_exchange_failed" value="true"/>
    <element name="CamelExceptionCaught"
        value="org.citrusframework.exceptions.CitrusRuntimeException: Something went
wrong!"/>
  </header>
</receive>
```

This completes the basic exception handling in Citrus when using the Camel endpoints.

16.5. Camel context handling

In the previous samples we have used the Camel context as Spring bean context that is automatically loaded when Citrus starts up. Now when using a single Camel context instance Citrus is able to automatically pick this Camel context for route interaction. If you use more than one Camel context you have to tell the Citrus endpoint component which context to use. The endpoint offers an optional attribute called `camel-context`.

Java

```
@Bean
public CamelEndpoint newsCamelEndpoint() {
    return new CamelEndpointBuilder()
        .camelContext(newsContext())
        .endpointUri("direct:news")
        .build();
}

@Bean
public CamelContext newsContext() {
    CamelContext context = new DefaultCamelContext();

    context.addRoutes(new RouteBuilder() {
        @Override
        public void configure() throws Exception {
            from("direct:news")
                .routeId("newsRoute")
                .to("log:org.citrusframework.camel?level=INFO")
                .to("seda:news-feed");
        }
    });

    return context;
}

@Bean
public CamelContext helloContext() {
    CamelContext context = new DefaultCamelContext();

    context.addRoutes(new RouteBuilder() {
        @Override
        public void configure() throws Exception {
            from("direct:hello")
                .routeId("helloRoute")
                .to("log:org.citrusframework.camel?level=INFO")
                .to("seda:greetings");
        }
    });

    return context;
}
```

XML

```
<citrus-camel:endpoint id="newsCamelEndpoint"
    camel-context="newsContext"
    endpoint-uri="direct:news"/>

<camelContext id="newsContext" xmlns="http://camel.apache.org/schema/spring">
    <route id="newsRoute">
      <from uri="direct:news"/>
      <to uri="log:org.citrusframework.camel?level=INFO"/>
      <to uri="seda:news-feed"/>
    </route>
</camelContext>

<camelContext id="helloContext" xmlns="http://camel.apache.org/schema/spring">
  <route id="helloRoute">
    <from uri="direct:hello"/>
    <to uri="log:org.citrusframework.camel?level=INFO"/>
    <to uri="seda:greetings"/>
  </route>
</camelContext>
```

In the example above we have two Camel context instances loaded. The endpoint has to pick the context to use with the attribute **camel-context** which resides to the Spring bean id of the Camel context.

16.6. Camel route actions

Since Citrus 2.4 we introduced some Camel specific test actions that enable easy interaction with Camel routes and the Camel context.

In XML the Camel route test actions do follow a specific XML namespace. This means you have to add this namespace to the test case when using the actions.

XML

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:camel="http://www.citrusframework.org/schema/camel/testcase"
      xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.citrusframework.org/schema/camel/testcase
      http://www.citrusframework.org/schema/camel/testcase/citrus-camel-testcase.xsd">

  [...]

</beans>
```

Once you have added the special Camel namespace with prefix `camel` you are ready to start using the Camel test actions in your test case.

Create Camel routes

You can create a new Camel route as part of the test using this test action.

Java

```
public class CamelRouteActionIT extends TestNGCitrusSpringSupport {

    @Autowired
    private CamelContext camelContext;

    @Test
    @CitrusTest
    public void createCamelRoute() {
        $(camel().camelContext(camelContext)
            .route()
            .create(new RouteBuilder() {
                @Override
                public void configure() throws Exception {
                    from("direct:messages")
                        .routeId("message-tokenizer")
                        .split().tokenize(" ")
                        .to("seda:words");
                }
            }));
    }
}
```

XML

```
<testcase name="CamelRouteIT">
  <actions>
      <camel:create-routes>
        <routeContext xmlns="http://camel.apache.org/schema/spring">
          <route id="message-tokenizer">
            <from uri="direct:messages"/>
            <split>
              <tokenize token=" "/>
              <to uri="seda:words"/>
            </split>
          </route>
        </routeContext>
      </camel:create-routes>
  </actions>
</testcase>
```

In the example above we have used the **camel:create-route** test action that will create new Camel

routes at runtime in the Camel context. The target Camel context is referenced with an automatic context lookup.

> The default Camel context name in this lookup is "*citrusCamelContext*". You can set this default context name via environment variables (*CITRUS_CAMEL_CONTEXT_NAME*) or system properties (*citrus.camel.context.name*).

If no specific settings are set Citrus will automatically try to look up the Camel context with name "*citrusCamelContext*" in the Spring bean configuration. All route operations will target this Camel context then.

In addition to that you can skip this lookup and directly reference a target Camel context with the action attribute **camel-context** (used in the second action above).

Remove Camel routes

You can remove routes from the Camel context as part of the test.

Java

```
public class CamelRouteActionIT extends TestNGCitrusSpringSupport {

    @Autowired
    private CamelContext camelContext;

    @Test
    @CitrusTest
    public void removeCamelRoutes() {
        $(camel().camelContext(camelContext)
            .route()
            .remove("route_1", "route_2", "route_3"));
    }
}
```

XML

```
<testcase name="CamelRouteIT">
  <actions>
      <camel:remove-routes camel-context="camelContext">
        <route id="route_1"/>
        <route id="route_2"/>
        <route id="route_3"/>
      </camel:remove-routes>
  </actions>
</testcase>
```

Start/stop routes

Next operation we will discuss is the start and stop of existing Camel routes:

Java

```
public class CamelRouteActionIT extends TestNGCitrusSpringSupport {

    @Autowired
    private CamelContext camelContext;

    @Test
    @CitrusTest
    public void startThenStopCamelRoutes() {
        $(camel().camelContext(camelContext)
            .route()
            .start("route_1"));

        $(camel().camelContext(camelContext)
            .route()
            .stop("route_2", "route_3"));
    }
}
```

XML

```
<testcase name="CamelRouteIT">
  <actions>
      <camel:start-routes camel-context="camelContext">
        <route id="route_1"/>
      </camel:start-routes>

      <camel:stop-routes camel-context="camelContext">
        <route id="route_2"/>
        <route id="route_3"/>
      </camel:stop-routes>
  </actions>
</testcase>
```

Starting and stopping Camel routes at runtime is important when temporarily Citrus need to receive a message on a Camel endpoint URI. We can stop a route, use a Citrus camel endpoint instead for validation and start the route after the test is done. This way we can also simulate errors and failure scenarios in a Camel route interaction.

16.7. Camel controlbus actions

The Camel controlbus component is a good way to access route statistics and route status information within a Camel context. Citrus provides controlbus test actions to easily access the controlbus operations at runtime.

Java

```
public class CamelControlBusIT extends TestNGCitrusSpringSupport {

    @Autowired
    private CamelContext camelContext;

    @Test
    @CitrusTest
    public void createCamelRoute() {
        $(camel().camelContext(camelContext)
            .controlbus()
            .route("route_1")
            .status()
            .result(ServiceStatus.Stopped));

        $(camel().camelContext(camelContext)
            .controlbus()
            .route("route_1")
            .start());

        $(camel().camelContext(camelContext)
            .controlbus()
            .route("route_1")
            .status()
            .result(ServiceStatus.Started));
    }
}
```

XML

```
<testcase name="CamelControlBusIT">
  <actions>
    <camel:control-bus camel-context="camelContext">
      <camel:route id="route_1" action="status"/>
      <camel:result>Stopped</camel:result>
    </camel:control-bus>

    <camel:control-bus>
      <camel:route id="route_1" action="start"/>
    </camel:control-bus>

    <camel:control-bus camel-context="camelContext">
      <camel:route id="route_1" action="status"/>
      <camel:result>Started</camel:result>
    </camel:control-bus>

    <camel:control-bus>
      <camel:language type="simple">${camelContext.stop()}</camel:language>
    </camel:control-bus>

    <camel:control-bus camel-context="camelContext">
      <camel:language
type="simple">${camelContext.getRouteController().getRouteStatus('route_3')}</camel:la
nguage>
      <camel:result>Started</camel:result>
    </camel:control-bus>
  </actions>
</testcase>
```

The example test case shows the controlbus access. As already mentioned you can explicitly reference a target Camel context with `camel-context="camelContext"`. In case no specific context is referenced Citrus will automatically lookup a target Camel context with the default context name "*citrusCamelContext*".

Camel provides two different ways to specify operations and parameters. The first option is the use of an **action** attribute. The Camel route id has to be specified as mandatory attribute. As a result the controlbus action will be executed on the target route during test runtime. This way we can also start and stop Camel routes in a Camel context.

In case a controlbus operation has a result such as the **status** action we can specify a control result that is compared. Citrus will raise validation exceptions when the results differ.

The second option for executing a controlbus action is the language expression. We can use Camel language expressions on the Camel context for accessing a controlbus operation. Also, here we can define an optional outcome as expected result.

Java

```
public class CamelControlBusIT extends TestNGCitrusSpringSupport {

    @Autowired
    private CamelContext camelContext;

    @Test
    @CitrusTest
    public void createCamelRoute() {
        $(camel().camelContext(camelContext)
            .controlbus()
            .language("simple", "${camelContext.getRouteStatus('my_route')}")
            .result(ServiceStatus.Stopped));

        $(camel().camelContext(camelContext)
            .controlbus()
            .language("simple", "${camelContext.stop()}"));
    }
}
```

XML

```
<testcase name="CamelControlBusIT">
  <actions>
    <camel:control-bus camel-context="camelContext">
      <camel:language
type="simple">${camelContext.getRouteStatus('my_route')}</camel:language>
      <camel:result>Started</camel:result>
    </camel:control-bus>

    <camel:control-bus>
      <camel:language type="simple">${camelContext.stop()}</camel:language>
    </camel:control-bus>
  </actions>
</testcase>
```

16.8. Camel infrastructure services

The Apache Camel project maintains a huge set of test infrastructure services that you can also start and stop during a Citrus test. See this documentation about the Camel test infra services.

The Camel infra services usually start Docker/Podman containers and/or use Testcontainers.

See this list of supported infrastructure services in Apache Camel:

- arangodb
- artemis (amqp, mqtt, persistent)

- aws (cloud-watch, config, dynamo-db, dynamodb, ec2, event-bridge, iam, kinesis, kms, lambda, s3, secrets-manager, sns, sqs, sts)
- azure
- cassandra
- chat-script
- couchbase
- couchdb
- elasticsearch
- fhir
- ftp
- ftps
- google (pub-sub)
- hashicorp (vault)
- hazelcast
- hive-mq (sparkplug)
- kafka (redpanda, strimzi)
- microprofile (lra)
- milvus
- minio
- mongodb
- mosquitto
- nats
- ollama
- openldap
- postgres
- pulsar
- qdrant
- rabbitmq
- redis
- rocketmq
- sftp
- smb
- solr
- torch-serve
- xmpp

- zookeeper

You can start/stop such an infra service via test actions in Citrus.

Java

```
public class CamelInfraIT extends TestNGCitrusSpringSupport {

    @Test
    @CitrusTest
    public void createInfraService() {
        $(camel()
            .infra()
            .run()
            .service("postgres"));
    }
}
```

YAML

```
name: "CamelInfraTest"
actions:
  - camel:
      infra:
        run:
          service: postgres
```

XML

```
<test name="CamelInfraIT"
      xmlns="http://citrusframework.org/schema/xml/testcase"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://citrusframework.org/schema/xml/testcase
http://citrusframework.org/schema/xml/testcase/citrus-testcase.xsd">
  <actions>
    <camel>
      <infra>
        <run service="postgres"/>
      </infra>
    </camel>
  </actions>
</testcase>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

Once the infrastructure service is started Citrus exposes connection settings as test variables. You can use the exposed connection settings to create proper clients that connect to the services.

The exposed connection settings follow a naming pattern that looks like this:

- `CITRUS_CAMEL_INFRA_SERVICE_<SERVICE_NAME>_<PROPERTY_NAME>`
- `CITRUS_CAMEL_INFRA_SERVICE_<SERVICE_NAME>_<IMPLEMENTATION>_<PROPERTY_NAME>`

For instance a infrastructure service called `foo` that exposes the settings `url` and `port` expose the following test variables.

Variable
CITRUS_CAMEL_INFRA_SERVICE_FOO_URL
CITRUS_CAMEL_INFRA_SERVICE_FOO_PORT

You can now use these test variables when connecting to the service. Please see the individual Apache Camel infrastructure services documentation to see which properties get exposed by the service.

Some infrastructure services provide multiple implementations (e.g. Kafka provides both Redpanda and Strimzi).

You may need to choose the implementation as follows:

Java

```
public class CamelInfraIT extends TestNGCitrusSpringSupport {

    @Test
    @CitrusTest
    public void createInfraService() {
        $(camel()
            .infra()
            .run()
            .service("kafka")
            .implementation("strimzi"));
    }
}
```

YAML

```
name: "CamelInfraTest"
actions:
  - camel:
      infra:
        run:
          service: kafka
          implementation: strimzi
```

XML

```
<test name="CamelInfraIT"
      xmlns="http://citrusframework.org/schema/xml/testcase"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://citrusframework.org/schema/xml/testcase
http://citrusframework.org/schema/xml/testcase/citrus-testcase.xsd">
  <actions>
    <camel>
      <infra>
        <run service="kafka" implementation="strimzi"/>
      </infra>
    </camel>
  </actions>
</testcase>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

By default, the infrastructure services get automatically removed after the test. You can disable the auto removal when running the service (`auto-remove` setting).

To explicitly stop a running infrastructure service use the following test action.

Java

```
public class CamelInfraIT extends TestNGCitrusSpringSupport {

    @Test
    @CitrusTest
    public void createInfraService() {
        $(camel()
            .infra()
            .stop()
            .service("postgres"));
    }
}
```

YAML

```
name: "CamelInfraTest"
actions:
  - camel:
      infra:
        stop:
          service: postgres
```

XML

```
<test name="CamelInfraIT"
      xmlns="http://citrusframework.org/schema/xml/testcase"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://citrusframework.org/schema/xml/testcase
http://citrusframework.org/schema/xml/testcase/citrus-testcase.xsd">
  <actions>
    <camel>
      <infra>
        <stop service="postgres"/>
      </infra>
    </camel>
  </actions>
</testcase>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

16.9. Camel endpoint DSL

Since Camel 3 the endpoint DSL provides a convenient way to construct an endpoint uri. In Citrus you can use the Camel endpoint DSL to send/receive messages in a test.

Java

```
$(send(camel().endpoint(seda("test")::getUri))
        .message()
        .body("Citrus rocks!"));
```

The fluent endpoint DSL in Camel allows to build the endpoint uri. The `camel().endpoint(seda("test")::getUri)` builds the endpoint uri `seda:test`. The endpoint DSL provides all settings and properties that you can set for a Camel endpoint component.

16.10. Camel processor support

Camel implements the concept of processors as enterprise integration pattern. A processor is able to add custom logic to a Camel route. Each processor is able to access the Camel exchange that is being processed in the current route. The processor is able to change the message content (body, headers) as well as the exchange information.

The send/receive operations in Citrus also implement the processor concept. With the Citrus Camel support you can use the very same Camel processor also in a Citrus test action.

Message processor on send

```
public class CamelMessageProcessorIT extends TestNGCitrusSpringSupport {

    @Autowired
    private CamelContext camelContext;

    @Test
    @CitrusTest
    public void shouldProcessMessages() {
        $(send(camel().endpoint(seda("test")::getUri))
                .message()
                .body("Citrus rocks!")
                .process(camel(camelContext)
                    .process(exchange -> exchange
                            .getMessage()

.setBody(exchange.getMessage().getBody(String.class).toUpperCase())))
        );
    }
}
```

The example above uses a Camel processor to change the exchange and the message content before the message is sent to the endpoint. This way you can apply custom changes to the message as part of the test action.

Message processor on receive

```
public class CamelMessageProcessorIT extends TestNGCitrusSpringSupport {

    @Autowired
    private CamelContext camelContext;

    @Test
    @CitrusTest
    public void shouldProcessMessages() {
        $(send(camel().endpoint(seda("test")::getUri))
                .message()
                .body("Citrus rocks!"));

        $(receive(camel().endpoint(seda("test")::getUri))
                .process(camel(camelContext)
                        .process(exchange -> exchange
                                .getMessage()

.setBody(exchange.getMessage().getBody(String.class).toUpperCase())))
                .message()
                .type(MessageType.PLAINTEXT)
                .body("CITRUS ROCKS!"));
    }
}
```

The Camel processors are very powerful. In particular, you can apply transformations of multiple kind.

Transform processor

```
public class CamelTransformIT extends TestNGCitrusSpringSupport {

    @Autowired
    private CamelContext camelContext;

    @Test
    @CitrusTest
    public void shouldTransformMessageReceived() {
        $(send(camel().endpoint(seda("hello")::getUri))
                .message()
                .body("{\"message\": \"Citrus rocks!\"}")
        );

        $(receive(camel().endpoint(seda("hello")::getUri))
                .transform(
                    camel()
                        .camelContext(camelContext)
                        .transform()
                        .jsonpath("$.message"))
                .message()
                .type(MessageType.PLAINTEXT)
                .body("Citrus rocks!"));
    }
}
```

The transform pattern is able to change the message content before a message is received/sent in Citrus. The example above applies a JsonPath expression as part of the message processing. The JsonPath expression evaluates `$.message` on the Json payload and saves the result as new message body content. The following message validation expects the plaintext value `Citrus rocks!`.

The message processor is also able to apply a complete route logic as part of the test action.

Route processor

```
public class CamelRouteProcessorIT extends TestNGCitrusSpringSupport {

    @Autowired
    private CamelContext camelContext;

    @Test
    @CitrusTest
    public void shouldProcessRoute() {
        CamelRouteProcessor.Builder beforeReceive = camel(camelContext).route(route ->
                route.choice()
                    .when(jsonpath("$.greeting[?(@.language == 'EN')]"))
                        .setBody(constant("Hello!"))
                    .when(jsonpath("$.greeting[?(@.language == 'DE')]"))
                        .setBody(constant("Hallo!"))
                    .otherwise()
                        .setBody(constant("Hi!")));

        $(send(camel().endpoint(seda("greetings")::getUri))
                .message()
                .body("{" +
                        "\"greeting\": {" +
                            "\"language\": \"EN\"" +
                        "}" +
                      "}")
        );

        $(receive("camel:" + camel().endpoints().seda("greetings").getUri())
                .process(beforeReceive)
                .message()
                .type(MessageType.PLAINTEXT)
                .body("Hello!"));
    }
}
```

With the complete route logic you have the full power of Camel ready to be used in your send/receive test action. This enables many capabilities as Camel implements the enterprise integration patterns such as split, choice, enrich and many more.

16.11. Camel data format support

Camel uses the concept of data format to transform message content in form of marshal/unmarshal operations. You can use the data formats supported in Camel in Citrus, too.

Data format marshal/unmarshal

```
public class CamelDataFormatIT extends TestNGCitrusSpringSupport {

    @Autowired
    private CamelContext camelContext;

    @Test
    @CitrusTest
    public void shouldApplyDataFormat() {
        when(send(camel().endpoint(seda("data")::getUri))
                .message()
                .body("Citrus rocks!")
                .transform(camel(camelContext)
                        .marshal()
                        .base64())
        );

        then(receive("camel:" + camel().endpoints().seda("data").getUri())
                .transform(camel(camelContext)
                        .unmarshal()
                        .base64())
                .transform(camel(camelContext)
                        .convertBodyTo(String.class))
                .message()
                .type(MessageType.PLAINTEXT)
                .body("Citrus rocks!"));
    }
}
```

The example above uses the `base64` data format provided in Camel to marshal/unmarshal the message content to/from a base64 encoded String. Camel provides support for many data formats as you can see in the documentation on data formats.

Chapter 17. Message channel support

Message channels represent the in memory messaging solution in Citrus. Producer and consumer components are linked via channels exchanging messages in memory. The transport comes from the Spring Integration project.

This opens up a lot of great possibilities to interact with the Spring Integration transport adapters for FTP, TCP/IP and so on. In addition to that the message channel support provides us a good way to exchange messages in memory.

The message channel configuration components use the "citrus-si" configuration namespace and schema definition. Include this namespace into your Spring configuration in order to use the Citrus configuration elements. The namespace URI and schema location are added to the Spring configuration XML file as follows.

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:citrus-si="http://www.citrusframework.org/schema/spring-
integration/config"
        xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.citrusframework.org/schema/spring-integration/config
        http://www.citrusframework.org/schema/spring-integration/config/citrus-spring-
integration-config.xsd">

    [...]

</beans>
```

Right now you are able to use customized Citrus XML elements in order to define the Spring Integration endpoint components.

17.1. Channel endpoint

Citrus offers a channel endpoint component that is able to create producers and consumers. Producer and consumer send and receive messages both to and from a channel endpoint. By default, the endpoint is asynchronous when configured in the Citrus context. With this component you are able to access message channels directly:

Java

```
@Bean
public ChannelEndpoint helloEndpoint() {
    return new ChannelEndpointBuilder()
        .channel("helloChannel")
        .build();
}

@Bean
public MessageSelectingQueueChannel helloChannel() {
    return new MessageSelectingQueueChannel();
}
```

XML

```
<citrus-si:channel-endpoint id="helloEndpoint" channel="helloChannel"/>

<citrus-si:channel id="helloChannel"/>
```

The Citrus channel endpoint references a Spring Integration channel directly. Inside your test case you can reference the Citrus endpoint as usual to send and receive messages.

The Citrus channel endpoint also supports a customized message channel template that will actually send the messages. The customized template might give you access to special configuration possibilities.

Java

```
@Bean
public ChannelEndpoint helloEndpoint() {
    return new ChannelEndpointBuilder()
        .channel("helloChannel")
        .messagingTemplate(messagingTemplate())
        .build();
}
```

XML

```
<citrus-si:channel-endpoint id="helloEndpoint"
                            channel="helloChannel"
                            message-channel-template="myMessageChannelTemplate"/>
```

The message sender is now ready to publish messages on the defined channel. The communication is supposed to be asynchronous, so the producer is not able to process any reply message. We will deal with synchronous communication and reply messages later in this chapter. You can reference the id of the endpoint in a send and receive test action.

Java

```
when(send("helloEndpoint")
        .message()
        .body("<v1:HelloRequest
xmlns:v1=\"http://citrusframework.org/schemas/HelloService.xsd\">" +
                "<v1:Text>Hello World!</v1:Text>" +
            "</v1:HelloRequest>"));

then(receive("helloEndpoint")
        .message()
        .body("<v1:HelloResponse
xmlns:v1=\"http://citrusframework.org/schemas/HelloService.xsd\">" +
                "<v1:Text>Hello Citrus!</v1:Text>" +
            "</v1:HelloResponse>"));
```

XML

```
<send endpoint="helloEndpoint">
    <message>
        <payload>
            <v1:HelloRequest
xmlns:v1="http://citrusframework.org/schemas/HelloService.xsd">
                <v1:Text>Hello World!</v1:Text>
            </v1:HelloRequest>
        </payload>
    </message>
</send>

<receive endpoint="helloEndpoint">
    <message>
        <payload>
            <v1:HelloResponse
xmlns:v1="http://citrusframework.org/schemas/HelloService.xsd">
                <v1:Text>Hello Citrus!</v1:Text>
            </v1:HelloResponse>
        </payload>
    </message>
</receive>
```

You can send and receive messages from the same Spring Integration message channel endpoint. As usual the receiver connects to the message destination and waits for messages to arrive. The user can set a receive timeout which is set to 5000 milliseconds by default. In case no message was received in this time frame the receiver raises timeout errors and the test fails.

17.2. Synchronous channel endpoints

The synchronous channel producer publishes messages and waits synchronously for the response to arrive on a reply channel destination. The reply channel name is set in the message headers. The

counterpart in this communication must send its reply to that channel. The basic configuration for a synchronous channel endpoint component looks like follows:

Java

```
@Bean
public ChannelSyncEndpoint helloEndpoint() {
    return new ChannelSyncEndpointBuilder()
        .channel("helloChannel")
        .pollingInterval(1000)
        .build();
}
```

XML

```
<citrus-si:channel-sync-endpoint id="helloSyncEndpoint"
                             channel="helloChannel"
                             polling-interval="1000"/>
```

Synchronous message channel endpoints usually do poll for synchronous reply messages for processing the reply messages. The poll interval is an optional setting in order to manage the amount of reply message handshake attempts. When the endpoint was able to receive the reply message synchronously the test case can verify the reply.

In case all polling attempts have failed the action raises a timeout error, and the test will fail.

By default, the channel endpoint uses temporary reply channel destinations. The temporary reply channels are only used once for a single communication handshake. The temporary reply channel is deleted automatically.

When sending a message to this endpoint in the first place the producer will wait synchronously for the response message to arrive on the reply channel. You can receive the reply message in your test case using the same endpoint component. So we have two actions on the same endpoint, first send then receive.

Java

```
when(send("helloSyncEndpoint")
        .message()
        .body("<v1:HelloRequest
xmlns:v1=\"http://citrusframework.org/schemas/HelloService.xsd\">" +
                "<v1:Text>Hello World!</v1:Text>" +
            "</v1:HelloRequest>"));

then(receive("helloSyncEndpoint")
        .message()
        .body("<v1:HelloResponse
xmlns:v1=\"http://citrusframework.org/schemas/HelloService.xsd\">" +
                "<v1:Text>Hello Citrus!</v1:Text>" +
            "</v1:HelloResponse>"));
```

XML

```
<send endpoint="helloSyncEndpoint">
    <message>
        <payload>
            <v1:HelloRequest
xmlns:v1="http://citrusframework.org/schemas/HelloService.xsd">
                <v1:Text>Hello World!</v1:Text>
            </v1:HelloRequest>
        </payload>
    </message>
</send>

<receive endpoint="helloSyncEndpoint">
    <message>
        <payload>
            <v1:HelloResponse
xmlns:v1="http://citrusframework.org/schemas/HelloService.xsd">
                <v1:Text>Hello Citrus!</v1:Text>
            </v1:HelloResponse>
        </payload>
    </message>
</receive>
```

This is how you handle synchronous communication as a sender. You publish messages to a channel and wait for reply messages on a temporary reply channel. The next section deals with the same synchronous communication, but now Citrus will receive a request and send a synchronous reply message to a temporary reply channel.

As usual the reply channel name is stored in the message headers. Citrus handles this synchronous communication with the same synchronous channel endpoint component. The handling of temporary reply destinations is done automatically behind the scenes.

So we have again two actions in our test case, but this time first receive then send.

Java

```
when(receive("helloSyncEndpoint")
        .message()
        .body("<v1:HelloRequest
xmlns:v1=\"http://citrusframework.org/schemas/HelloService.xsd\">" +
                "<v1:Text>Hello World!</v1:Text>" +
            "</v1:HelloRequest>"));

then(send("helloSyncEndpoint")
        .message()
        .body("<v1:HelloResponse
xmlns:v1=\"http://citrusframework.org/schemas/HelloService.xsd\">" +
                "<v1:Text>Hello Citrus!</v1:Text>" +
            "</v1:HelloResponse>"));
```

XML

```
<receive endpoint="helloSyncEndpoint">
    <message>
        <payload>
            <v1:HelloRequest
xmlns:v1="http://citrusframework.org/schemas/HelloService.xsd">
                <v1:Text>Hello World!</v1:Text>
            </v1:HelloRequest>
        </payload>
    </message>
</receive>

<send endpoint="helloSyncEndpoint">
    <message>
        <payload>
            <v1:HelloResponse
xmlns:v1="http://citrusframework.org/schemas/HelloService.xsd">
                <v1:Text>Hello Citrus!</v1:Text>
            </v1:HelloResponse>
        </payload>
    </message>
</send>
```

17.3. Message selectors

A channel can hold multiple messages at the same time. Usually you receive messages using first-in-first-out pattern. Message selectors enable you to select messages form that channel so you can pick messages form a channel based on a selector evaluation.

Citrus introduces a special queue message channel implementation that support message selectors.

Java

```
@Bean
public MessageSelectingQueueChannel helloChannel() {
    return new MessageSelectingQueueChannel();
}
```

XML

```
<citrus-si:channel id="orderChannel" capacity="5"/>
```

The Citrus message channel implementation extends the queue channel implementation from Spring Integration. So we can add a capacity attribute for this channel. A receive test action makes use of message selectors on header values as described in message-selector.

In addition to that we have implemented other message filter possibilities on message channels that we discuss in the next sections.

17.4. Payload matching selector

You can select messages based on the payload content. Either you define the expected payload as an exact match in the selector or you make use of Citrus validation matchers which is more adequate in most scenarios.

Assume there are two different plain text messages living on a message channel waiting to be picked up by a consumer.

```
Hello, welcome!
```

```
GoodBye, see you next time!
```

The tester would like to pick up the message starting with **GoodBye** in our test case. The other messages should be left on the channel as we are not interested in it right now. We can define a payload matching selector in the receive action like this:

Java

```
when(receive("orderChannelEndpoint")
        .selector(Collections.singletonMap("payload", "@startsWith(GoodBye)@"))
        .message()
        .body("GoodBye, see you next time!"));
```

XML

```
<receive endpoint="orderChannelEndpoint">
    <selector>
        <element name="payload" value="@startsWith(GoodBye)@"/>
    </selector>
    <message>
        <payload>GoodBye, see you next time!</payload>
    </message>
</receive>
```

The Citrus receiver picks up the **GoodBye** from the channel selected via the payload matching expression defined in the selector element. Of course, you can also combine message header selectors and payload matching selectors as shown in this example below where a message header **sequenceId** is added to the selection logic.

Java

```
Map<String, String> selectorMap = new HashMap<>();
selectorMap.put("payload", "@startsWith(GoodBye)@");
selectorMap.put("sequenceId", "1234");

when(receive("orderChannelEndpoint")
        .selector(selector)
        .message()
        .body("GoodBye, see you next time!"));
```

XML

```
<selector>
    <element name="payload" value="@startsWith(GoodBye)@"/>
    <element name="sequenceId" value="1234"/>
</selector>
```

17.5. Root QName selector

As a special payload matching selector you can use the XML root QName of your message as selection criteria when dealing with XML message content. Let's see how this works in a small example:

We have two different XML messages on a message channel waiting to be picked up by a consumer.

```
<HelloMessage xmlns="http://citrusframework.org/schema">Hello Citrus</HelloMessage>
```

```
<GoodbyeMessage xmlns="http://citrusframework.org/schema">Goodbye
Citrus</GoodbyeMessage>
```

We would like to pick up the **GoodbyeMessage** in our test case. The **HelloMessage** should be left on the message channel as we are not interested in it right now. We can define a root qname message selector in the receive action like this:

Java

```
when(receive("orderChannelEndpoint")
        .selector(Collections.singletonMap("root-qname", "GoodbyeMessage"))
        .message()
        .body("<GoodbyeMessage xmlns=\"http://citrusframework.org/schema\">Goodbye
Citrus</GoodbyeMessage>"));
```

XML

```
<receive endpoint="orderChannelEndpoint">
    <selector>
        <element name="root-qname" value="GoodbyeMessage"/>
    </selector>
    <message>
        <payload>
            <GoodbyeMessage xmlns="http://citrusframework.org/schema">Goodbye
Citrus</GoodbyeMessage>
        </payload>
    </message>
</receive>
```

The Citrus receiver picks up the **GoodbyeMessage** from the channel selected via the root qname of the XML message payload. Of course, you can also combine message header selectors and root qname selectors as shown in this example below where a message header **sequenceId** is added to the selection logic.

Java

```
Map<String, String> selectorMap = new HashMap<>();
selectorMap.put("root-qname", "GoodbyeMessage");
selectorMap.put("sequenceId", "1234");

when(receive("orderChannelEndpoint")
        .selector(selector)
        .message()
        .body("GoodBye, see you next time!"));
```

XML

```
<selector>
    <element name="root-qname" value="GoodbyeMessage"/>
    <element name="sequenceId" value="1234"/>
</selector>
```

As we deal with XML qname values, we can also use namespaces in our selector root qname selection.

Java

```
when(receive("orderChannelEndpoint")
        .selector(Collections.singletonMap("root-qname",
"{http://citrusframework.org/schema}GoodbyeMessage"))
        .message()
        .body("<GoodbyeMessage xmlns=\"http://citrusframework.org/schema\">Goodbye
Citrus</GoodbyeMessage>"));
```

XML

```
<selector>
    <element name="root-qname"
value="{http://citrusframework.org/schema}GoodbyeMessage"/>
</selector>
```

17.6. Xpath selector

It is also possible to evaluate some XPath expression on the message payload in order to select a message from a message channel. The XPath expression outcome must match an expected value and only then the message is consumed from the channel.

The syntax for the XPath expression is to be defined as the element name like this:

Java

```
when(receive("orderChannelEndpoint")
        .selector(Collections.singletonMap("xpath://Order/status", "pending"))
        .message()
        .body("<Order><status>pending</status></Order>"));
```

XML

```
<selector>
    <element name="xpath://Order/status" value="pending"/>
</selector>
```

The message selector looks for order messages with **status="pending"** in the message payload. This means that following messages would get accepted/declined by the message selector.

```
<Order><status>pending</status></Order> <!-- ACCEPTED -->
<Order><status>finished</status></Order> <!-- NOT ACCEPTED -->
```

Of course, you can also use XML namespaces in your XPath expressions when selecting messages

from channels.

Java

```
when(receive("orderChannelEndpoint")
        .selector(Collections.singletonMap("xpath://ns1:Order/ns1:status", "pending"))
        .message()
        .body("<Order><status>pending</status></Order>"));
```

XML

```
<selector>
    <element name="xpath://ns1:Order/ns1:status" value="pending"/>
</selector>
```

Namespace prefixes must match the incoming message - otherwise the XPath expression will not work as expected. In our example the message should look like this:

```
<ns1:Order
xmlns:ns1="http://citrus.org/schema"><ns1:status>pending</ns1:status></ns1:Order>
```

Knowing the correct XML namespace prefix is not always easy. If you are not sure which namespace prefix to choose Citrus ships with a dynamic namespace replacement for XPath expressions. The XPath expression looks like this and is most flexible:

Java

```
when(receive("orderChannelEndpoint")
        .selector(Collections.singletonMap(

"xpath://{http://citrus.org/schema}:Order/{http://citrus.org/schema}:status",
"pending"))
        .message()
        .body("<Order><status>pending</status></Order>"));
```

XML

```
<selector>
    <element
name="xpath://{http://citrus.org/schema}:Order/{http://citrus.org/schema}:status"
                value="pending"/>
</selector>
```

This will match all incoming messages regardless the XML namespace prefix that is used.

17.7. JsonPath selector

It is also possible to evaluate some JsonPath expression on the message payload in order to select a message from a message channel. The JsonPath expression outcome must match an expected value and only then the message is consumed from the channel.

The syntax for the JsonPath expression is to be defined as the element name like this:

Java

```
when(receive("orderChannelEndpoint")
        .selector(Collections.singletonMap("jsonPath:$.order.status", "pending"))
        .message()
        .body("{ \"order\": { \"status\": \"pending\" } }"));
```

XML

```
<selector>
    <element name="jsonPath:$.order.status" value="pending"/>
</selector>
```

The message selector looks for order messages with **status="pending"** in the message payload. This means that following messages would get accepted/declined by the message selector.

```
{ "order": { "status": "pending" } } //ACCEPTED
{ "order": { "status": "finished" } } //NOT ACCEPTED
```

Chapter 18. WebSocket support

The WebSocket message protocol builds on top of Http standard and brings bidirectional communication to the Http client-server world. Citrus is able to send and receive messages with WebSocket connections as client and server. The Http server implementation is now able to define multiple WebSocket endpoints. The new Citrus WebSocket client is able to publish and consumer messages via bidirectional WebSocket protocol.

The new WebSocket support is located in the module **citrus-websocket** . Therefore we need to add this module to our project as dependency when we are about to use the WebSocket features in Citrus.

```
<dependency>
    <groupId>org.citrusframework</groupId>
    <artifactId>citrus-websocket</artifactId>
    <version>${citrus.version}</version>
</dependency>
```

As Citrus provides a customized WebSocket configuration schema for the Spring application context configuration files we have to add name to the top level **beans** element. Simply include the websocket-config namespace in the configuration XML files as follows.

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:citrus="http://www.citrusframework.org/schema/config"
      xmlns:citrus-websocket="http://www.citrusframework.org/schema/websocket/config"
      xsi:schemaLocation="
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.citrusframework.org/schema/config
            http://www.citrusframework.org/schema/config/citrus-config.xsd
            http://www.citrusframework.org/schema/websocket/config
            http://www.citrusframework.org/schema/websocket/config/citrus-websocket-
config.xsd">

    [...]

</beans>
```

Now our project is ready to use the Citrus WebSocket support. First of all let us send a message via WebSocket connection to some server.

18.1. WebSocket client

On the client side Citrus offers a client component that goes directly to the Spring bean application context. The client needs a server endpoint uri. This is a WebSocket protocol endpoint uri.

```
<citrus-websocket:client id="helloWebSocketClient"
    url="http://localhost:8080/hello"
    timeout="5000"/>
```

The **url** defines the endpoint to send messages to. The server has to be a WebSocket ready web server that supports Http connection upgrade for WebSocket protocols. WebSocket by its nature is an asynchronous bidirectional protocol. This means that the connection between client and server remains open and both server and client can send and receive messages. So when the Citrus client is waiting for a message we need a timeout that stops the asynchronous waiting. The receiving test action and the test case will fail when such a timeout is raised.

The WebSocket client will automatically open a connection to the server and ask for a connection upgrade to WebSocket protocol. This handshake is done once when the connection to the server is established. After that the client can push messages to the server and on the other side the server can push messages to the client. Now let's first push some messages to the server:

```
<send endpoint="helloWebSocketClient">
  <message>
      <payload>
          <TestMessage>
              <Text>Hello WebSocketServer</Text>
          </TestMessage>
      </payload>
  </message>
</send>
```

The connection handshake and the connection upgrade is done automatically by the client. After that the message is pushed to the server. As WebSocket is a bidirectional protocol we can also receive messages on the WebSocket client. These messages are pushed from server to all connected clients.

```
<receive endpoint="helloWebSocketClient">
  <message>
      <payload>
          <TestMessage>
              <Text>Hello WebSocketClient</Text>
          </TestMessage>
      </payload>
  </message>
</receive>
```

We just use the very same client endpoint component in a message receive action. The client will wait for messages from the server and once received perform the well known message validation. Here we expect some XML message payload. This completes the client side as we are able to push and consumer messages via WebSocket connections.

Up to now we have used static WebSocket endpoint URIs in our client component configurations. This can be done with a more powerful dynamic endpoint URI in WebSocket client. Similar to the endpoint resolving mechanism in SOAP you can dynamically set the called endpoint uri at test runtime through message header values. By default Citrus will check a specific header entry for dynamic endpoint URI which is simply defined for each message sending action inside the test.

The **dynamicEndpointResolver** bean must implement the EndpointUriResolver interface in order to resolve dynamic endpoint uri values. Citrus offers a default implementation, the **DynamicEndpointUriResolver**, which uses a specific message header for setting dynamic endpoint uri. The message header needs to specify the header **citrus_endpoint_uri** with a valid request uri.

```
<header>
  <element name="citrus_endpoint_uri"
value="ws://localhost:8080/customers/${customerId}"/>
</header>
```

The specific send action above will send its message to the dynamic endpoint (ws://localhost:8080/customers/${customerId}[ws://localhost:8080/customers/${customerId}]) which is set in the header **citrus_endpoint_uri** .

18.2. WebSocket server endpoints

On the server side Citrus has a Http server implementation that we can easily start during test runtime. The Http server accepts connections from clients and also supports WebSocket upgrade strategies. This means clients can ask for an upgrade to the WebSocket standard. In this handshake the server will upgrade the connection to WebSocket and afterwards client and server can exchange messages over this connection. This means the connection is kept alive and multiple messages can be exchanged. Let's see how WebSocket endpoints are added to a Http server component in Citrus.

```
<citrus-websocket:server id="helloHttpServer"
        port="8080"
        auto-start="true"
        resource-base="src/it/resources">
    <citrus-websocket:endpoints>
        <citrus-websocket:endpoint ref="websocket1"/>
        <citrus-websocket:endpoint ref="websocket2"/>
    </citrus-websocket:endpoints>
</citrus-websocket:server>

<citrus-websocket:endpoint id="websocket1" path="/test1"/>
<citrus-websocket:endpoint id="websocket2" path="/test2" timeout="10000"/>
```

The embedded Jetty WebSocket server component in Citrus now is able to define multiple

WebSocket endpoints. The WebSocket endpoints match to a request path on the server and are referenced by a unique id. Each WebSocket endpoint can follow individual timeout settings. In a test we can use these endpoints directly to receive messages.

```
<testcase name="httpWebSocketServerTest">
    <actions>
        <receive endpoint="websocket1">
            <message>
                <data>
                  [...]
                </data>
            </message>
        </receive>

        <send endpoint="websocket1">
            <message>
                <data>
                  [...]
                </data>
            </message>
        </send>
    </actions>
</testcase>
```

As you can see we reference the endpoint id in both receive and send actions. Each WebSocket endpoint holds one or more open connections to its clients. Each message that is sent is pushed to all connected clients. Each client can send messages to the WebSocket endpoint.

The WebSocket endpoint component handles connection handshakes automatically and caches all open sessions in memory. By default all connected clients will receive the messages pushed from server. This is done completely behind the scenes. The Citrus server is able to handle multiple WebSocket endpoints with different clients connected to it at the same time. This is why we have to choose the WebSocket endpoint on the server by its identifier when sending and receiving messages.

With this WebSocket endpoints we change the Citrus server behavior so that clients can upgrade to WebSocket connection. Now we have a bidirectional connection where the server can push messages to the client and vice versa.

18.3. WebSocket headers

The WebSocket standard defines some default headers to use during connection upgrade. These headers are made available to the test case in both directions. Citrus will handle these header values with special care when WebSocket support is activated on a server or client. Now WebSocket messages can also be split into multiple pieces. Each message part is pushed separately to the server but still is considered to be a single message payload. The server has to collect and aggregate all messages until a special message header **isLast** is set in one of the message parts.

The Citrus WebSocket client can slice messages into several parts.

```
<send endpoint="webSocketClient">
    <message type="json">
        <data>
        [
            {
                "event" : "client_message_1",
                "timestamp" : "citrus:currentDate()"
            },
        </data>
    </message>
    <header>
        <element name="citrus_websocket_is_last" value="false"/>
    </header>
</send>

<sleep milliseconds="500"/>

<send endpoint="webSocketClient">
    <message type="json">
        <data>
            {
                "event" : "client_message_2",
                "timestamp" : "citrus:currentDate()"
            }
          ]
        </data>
    </message>
    <header>
        <element name="citrus_websocket_is_last" value="true"/>
    </header>
</send>
```

The test above has two separate send operations both sending to a WebSocket endpoint. The first sending action sets the header **citrus_websocket_is_last** to **false** which indicates that the message is not complete yet. The 2nd send action pushes the rest of the message to the server and set the **citrus_websocket_is_last** header to **true** . Now the server is able to aggregate the message pieces to a single message payload. The result is a valida JSON array with both events in it.

```
[
  {
    "event" : "client_message_1",
    "timestamp" : "2015-01-01"
  },
  {
    "event" : "client_message_2",
    "timestamp" : "2015-01-01"
  }
]
```

Now the server part in Citrus is able to handle these sliced messages, too. The server will automatically aggregate those message parts before passing it to the test case for validation.

Chapter 19. Mail support

Sending and receiving mails is the next interest we are going to talk about. When dealing with mail communication you most certainly need to interact with some sort of IMAP or POP mail server. But in Citrus we do not want to manage mails in a personal inbox. We just need to be able to exchange mail messages the persisting in a user inbox is not part of our business.

This is why Citrus provides **just** an SMTP mail server which accepts mail messages from clients. Once the SMTP server has accepted an incoming mail it forwards those data to the running test case. In the test case you can receive the incoming mail message and perform message validation as usual. The mail sending part is easy as Citrus offers a mail client that connects to some SMTP server for sending mails to the outside world.

The mail components in Citrus are kept in a separate Maven module. So you should check that the module is available as Maven dependency in your project

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-mail</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

As usual Citrus provides a customized mail configuration schema that is used in Spring configuration files. Simply include the citrus-mail namespace in the configuration XML files as follows.

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:citrus="http://www.citrusframework.org/schema/config"
      xmlns:citrus-mail="http://www.citrusframework.org/schema/mail/config"
      xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.citrusframework.org/schema/config
      http://www.citrusframework.org/schema/config/citrus-config.xsd
      http://www.citrusframework.org/schema/mail/config
      http://www.citrusframework.org/schema/mail/config/citrus-mail-config.xsd">

      [...]

      </beans>
```

Now you are ready to use the customized Http configuration elements with the citrus-mail namespace prefix.

Read the next section in order to find out more about the mail message support in Citrus.

19.1. Mail client

The mail sending part is quite easy and straight forward. We just need to send a mail message to some SMTP server. So Citrus provides a mail client that sends out mail messages.

```
<citrus-mail:client id="simpleMailClient"
      host="localhost"
      port="25025"/>
```

This is how a Citrus mail client component is defined in the Spring application context. You can use this client referenced by its id or name in your test case in a message sending action. The client defines a host and port attribute which should connect the client to some SMTP server instance.

We all know mail message contents. The mail message has some general properties set by the user:

from The message sender mail address

to The message recipient mail address. You can add multiple recipients by using a comma separated list.

cc Copy recipient mail address. You can add multiple recipients by using a comma separated list.

bcc Blind copy recipient mail address. You can add multiple recipients by using a comma separated list.

subject Some subject used as mail headline.

As a tester you are able to set these properties in your test case. Citrus defines an XML mail message representation that you can use inside your send action. Let us have a look at this:

```
<send endpoint="simpleMailClient">
    <message>
        <payload>
            <mail-message xmlns="http://www.citrusframework.org/schema/mail/message">
                <from>christoph@citrusframework.com</from>
                <to>dev@citrusframework.com</to>
                <cc></cc>
                <bcc></bcc>
                <subject>This is a test mail message</subject>
                <body>
                    <contentType>text/plain; charset=utf-8</contentType>
                    <content>Hello Citrus mail server!</content>
                </body>
            </mail-message>
        </payload>
    </message>
</send>
```

The basic XML mail message representation defines a list of basic mail properties such as **from**, **to** or **subject** . In addition to that we define a text body which is either plain text or HTML. You can specify the content type of the mail body very easy (e.g. text/plain or text/html). By default Citrus uses **text/plain** content type.

Now when dealing with mail messages you often come to use multipart structures for attachments. In Citrus you can define attachment content as base64 character sequence. The Citrus mail client will automatically create a proper multipart mail mime message using the content types and body parts specified.

```
<send endpoint="simpleMailClient">
    <message>
        <payload>
            <mail-message xmlns="http://www.citrusframework.org/schema/mail/message">
                <from>christoph@citrusframework.com</from>
                <to>dev@citrusframework.com</to>
                <cc></cc>
                <bcc></bcc>
                <subject>This is a test mail message</subject>
                <body>
                    <contentType>text/plain; charset=utf-8</contentType>
                    <content>Hello Citrus mail server!</content>
                    <attachments>
                        <attachment>
                            <contentType>text/plain; charset=utf-8</contentType>
                            <content>This is attachment data</content>
                            <fileName>attachment.txt</fileName>
                        </attachment>
                    </attachments>
                </body>
            </mail-message>
        </payload>
    </message>
</send>
```

That completes the basic mail client capabilities. But wait we have not talked about error scenarios where mail communication results in error. When running into mail error scenarios we have to handle the error respectively with exception handling. When the mail server responded with errors Citrus will raise mail exceptions automatically and your test case fails accordingly.

As a tester you can catch and assert these mail exceptions verifying your error scenario.

```
<assert exception="org.springframework.mail.MailSendException">
    <when>
        <send endpoint="simpleMailClient">
            <message>
                <payload>
                    <mail-message
xmlns="http://www.citrusframework.org/schema/mail/message">
                        [...]
                    </mail-message>
                </payload>
            </message>
        </send>
    </when>
<assert/>
```

We assert the ***MailSendException*** from Spring to be thrown while sending the mail message to the

SMTP server. With exception message validation you are able to expect very specific mail send errors on the client side. This is how you can handle some sort of error situation returned by the mail server. Speaking of mail servers we need to also talk about providing a mail server endpoint in Citrus for clients. This is part of our next section.

19.2. Mail server

Consuming mail messages is a more complicated task as we need to have some sort of server that clients can connect to. In your mail client software you typically point to some IMAP or POP inbox and receive mails from that endpoint. In Citrus we do not want to manage a whole personal mail inbox such as IMAP or POP would provide. We just need an SMTP server endpoint for clients to send mails to. The SMTP server accepts mail messages and forwards those to a running test case for further validation.

> We have no user inbox where incoming mails are stored. The mail server just forwards incoming mails to the running test for validation. After the test the incoming mail message is gone.

And this is exactly what the Citrus mail server is capable of. The server is a very lightweight SMTP server. All incoming mail client connections are accepted by default and the mail data is converted into a Citrus XML mail interface representation. The XML mail message is then passed to the running test for validation.

Let us have a look at the Citrus mail server component and how you can add it to the Spring application context.

```
<citrus-mail:server id="simpleMailServer"
      port="25025"
      auto-start="true"/>
```

The mail server component receives several properties such as **port** or **auto-start** . Citrus starts an in-memory SMTP server that clients can connect to.

In your test case you can then receive the incoming mail messages on the server in order to perform the well known XML validation mechanisms within Citrus. The message header and the payload contain all mail information so you can verify the content with expected templates as usual:

```
<receive endpoint="simpleMailServer">
    <message>
        <payload>
            <mail-message xmlns="http://www.citrusframework.org/schema/mail/message">
                <from>christoph@citrusframework.com</from>
                <to>dev@citrusframework.com</to>
                <cc></cc>
                <bcc></bcc>
                <subject>This is a test mail message</subject>
                <body>
                    <contentType>text/plain; charset=utf-8</contentType>
                    <content>Hello Citrus mail server!</content>
                </body>
            </mail-message>
        </payload>
    </message>
    <header>
        <element name="citrus_mail_from" value="christoph@citrusframework.com"/>
        <element name="citrus_mail_to" value="dev@citrusframework.com"/>
        <element name="citrus_mail_subject" value="This is a test mail message"/>
        <element name="citrus_mail_content_type" value="text/plain; charset=utf-8"/>
    </header>
</receive>
```

The general mail properties such as **from**, **to**, **subject** are available as elements in the mail payload and in the message header information. The message header names do start with a common Citrus mail prefix **citrus_mail** . Following from that you can verify these special mail message headers in your test as shown above. Citrus offers following mail headers:

- citrus_mail_from
- citrus_mail_to
- citrus_mail_cc
- citrus_mail_bcc
- citrus_mail_subject
- citrus_mail_replyTo
- citrus_mail_date

In addition to that Citrus converts the incoming mail data to a special XML mail representation which is passed as message payload to the test. The mail body parts are represented as body and optional attachment elements. As this is plain XML you can verify the mail message content as usual using Citrus variables, functions and validation matchers.

Regardless of how the mail message has passed the validation the Citrus SMTP mail server will automatically respond with success codes (SMTP 250 OK) to the calling client. This is the basic Citrus mail server behavior where all client connections are accepted an all mail messages are responded with SMTP 250 OK response codes.

Now in more advanced usage scenarios the tester may want to control the mail communication outcome. User can force some error scenarios where mail clients are not accepted or mail communication should fail with some SMTP error state for instance.

By using a more advanced mail server setup the tester gets more power to sending back mail server response codes to the mail client. Just use the advanced mail adapter implementation in your mail server component configuration:

```
<citrus-mail:server id="advancedMailServer"
      auto-accept="false"
      split-multipart="true"
      port="25025"
      auto-start="true"/>
```

We have disabled the **auto-accept** mode on the mail server. This means that we have to do some additional steps in your test case to accept the incoming mail message first. So we can decide in our test case whether to accept or decline the incoming mail message for a more powerful test. You accept/decline a mail message with a special XML accept request/response exchange in your test case:

```
<receive endpoint="advancedMailServer">
    <message>
        <payload>
            <accept-request
xmlns="http://www.citrusframework.org/schema/mail/message">
                <from>christoph@citrusframework.com</from>
                <to>dev@citrusframework.com</to>
            </accept-request>
        </payload>
    </message>
</receive>
```

So before receiving the actual mail message we receive this simple accept-request in our test. The accept request gives us the message **from** and **to** resources of the mail message. Now the test decides to also decline a mail client connection. You can simulate that the server does not accept the mail client connection by sending back a negative accept response.

```
<send endpoint="advancedMailServer">
    <message>
        <payload>
            <accept-response
xmlns="http://www.citrusframework.org/schema/mail/message">
                <accept>true</accept>
            </accept-response>
        </payload>
    </message>
</send>
```

Depending on the accept outcome the mail client will receive an error response with proper error codes. If you accept the mail message with a positive accept response the next step in your test receives the actual mail message as we have seen it before in this chapter.

Now besides not accepting a mail message in the first place you can als simulate another error scenario with the mail server. In this scenario the mail server should respond with some sort of SMTP error code after accepting the message. This is done with a special mail response message like this:

```
<receive endpoint="advancedMailServer">
    <message>
        <payload>
            <mail-message xmlns="http://www.citrusframework.org/schema/mail/message">
                <from>christoph@citrusframework.com</from>
                <to>dev@citrusframework.com</to>
                <cc></cc>
                <bcc></bcc>
                <subject>This is a test mail message</subject>
                <body>
                    <contentType>text/plain; charset=utf-8</contentType>
                    <content>Hello Citrus mail server!</content>
                </body>
            </mail-message>
        </payload>
    </message>
</receive>

<send endpoint="advancedMailServer">
    <message>
        <payload>
            <mail-response xmlns="http://www.citrusframework.org/schema/mail/message">
                <code>443</code>
                <message>Failed!</message>
            </mail-response>
        </payload>
    </message>
</send>
```

As you can see from the example above we first accept the connection and receive the mail content as usual. Now the test returns a negative mail response with some error code reason set. The Citrus SMTP communication will then fail and the calling mail client receives the respective error.

If you skip the negative mail response the server will automatically respond with positive SMTP response codes to the calling client.

Chapter 20. FTP support

With Citrus it is possible to start your own ftp server for accepting incoming client requests. You can also use Citrus as an FTP client to send FTP commands. The next sections deal with FTP connectivity.

The FTP components in Citrus are maintained in their own Maven module. So you should add the module as Maven dependency to your project accordingly.

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-ftp</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

As Citrus provides a customized FTP configuration schema for the Spring application context configuration files we have to add name to the top level **beans** element. Simply include the ftp-config namespace in the configuration XML files as follows.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:citrus="http://www.citrusframework.org/schema/config"
    xmlns:citrus-ftp="http://www.citrusframework.org/schema/ftp/config"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.citrusframework.org/schema/config
    http://www.citrusframework.org/schema/config/citrus-config.xsd
    http://www.citrusframework.org/schema/http/config
    http://www.citrusframework.org/schema/ftp/config/citrus-ftp-config.xsd">

      [...]

      </beans>
```

Now we are ready to use the customized Citrus FTP configuration elements with the citrus-ftp namespace prefix.

20.1. FTP client

We want to use Citrus to connect to some FTP server as a client sending commands such as creating a directory or listing files. Citrus offers a client component doing exactly this FTP client connection.

Spring Bean

```
@Bean
public FtpClient ftpClient() {
    return CitrusEndpoints.ftp()
            .client()
            .port(2222)
            .username("citrus")
            .password("admin")
            .timeout(10000L)
            .build();
}
```

Spring XML

```
<citrus-ftp:client id="ftpClient"
      host="localhost"
      port="2222"
      username="admin"
      password="admin"
      timeout="10000"/>
```

The configuration above describes a Citrus ftp client connected to a ftp server with `ftp://localhost:22222`. For authentication username and password are defined as well as the global connection timeout. The client will automatically send username and password for proper authentication to the server when opening a new connection.

20.1.1. FTP client commands

In a test case you are now able to use the client to push commands to the server.

Java

```
@CitrusTest
public void ftpMessageTest() {
    $(send(ftpClient)
        .message(FtpMessage.command(FTPCmd.MKD).arguments("test"))
    );

    CommandResult result = new CommandResult();
    result.setSuccess(true);
    result.setReplyCode(String.valueOf(257));
    result.setReplyString("@contains(\"/test\" created)@");

    $(receive(ftpClient)
        .message(FtpMessage.result(result))
    );
}
```

XML

```
<test name="FtpMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <send endpoint="ftpClient">
            <message>
                <body>
                    <payload>
                        <ftp:command
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:signal>MKD</ftp:signal>
                            <ftp:arguments>test</ftp:arguments>
                        </ftp:command>
                    </payload>
                </body>
            </message>
        </send>

        <receive endpoint="ftpClient">
            <message>
                <body>
                    <payload>
                        <ftp:command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:success>true</ftp:success>
                            <ftp:reply-code>257</ftp:reply-code>
                            <ftp:reply-string>257 "/test" created.</ftp:reply-string>
                        </ftp:command-result>
                    </payload>
                </body>
            </message>
        </receive>
    </actions>
</test>
```

YAML

```
name: FtpMessageTest
actions:
  - send:
      endpoint: "ftpClient"
      message:
        body: |
          <ftp:command xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
            <ftp:signal>MKD</ftp:signal>
            <ftp:arguments>test</ftp:arguments>
          </ftp:command>
  - receive:
      endpoint: "ftpClient"
      message:
        body: |
          <ftp:command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
            <ftp:success>true</ftp:success>
            <ftp:reply-code>257</ftp:reply-code>
            <ftp:reply-string>257 "/test" created.</ftp:reply-string>
          </ftp:command-result>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="FtpMessageTest">
        <actions>
            <send endpoint="ftpClient">
                <message>
                    <payload>
                        <ftp:command
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:signal>MKD</ftp:signal>
                            <ftp:arguments>test</ftp:arguments>
                        </ftp:command>
                    </payload>
                </message>
            </send>

            <receive endpoint="ftpClient">
                <message>
                    <payload>
                        <ftp:command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:success>true</ftp:success>
                            <ftp:reply-code>257</ftp:reply-code>
                            <ftp:reply-string>257 "/test" created.</ftp:reply-string>
                        </ftp:command-result>
                    </payload>
                </message>
            </receive>
        </actions>
    </testcase>
</spring:beans>
```

As you can see most of the ftp communication parameters are specified in a ftp command message. Citrus automatically converts those information to proper FTP commands and response messages.

20.1.2. Store files

The client is able to store files on the server using file transfer.

Java

```
@CitrusTest
public void ftpMessageTest() {
    $(send(ftpClient)
        .message(FtpMessage.put("test/hello.txt", DataType.ASCII).arguments(""))
    );

    PutCommandResult result = new PutCommandResult();
    result.setSuccess(true);
    result.setReplyCode(String.valueOf(226));
    result.setReplyString("@contains(Transfer complete)@");

    $(receive(ftpClient)
        .message(FtpMessage.result(result))
    );
}
```

XML

```
<test name="FtpMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <send endpoint="ftpClient">
            <message>
                <body>
                    <payload>
                        <ftp:put-command
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:file path="test/hello.txt" type="ASCII"/>
                            <ftp:target path="/test/hello.txt"/>
                        </ftp:put-command>
                    </payload>
                </body>
            </message>
        </send>

        <receive endpoint="ftpClient">
            <message>
                <body>
                    <payload>
                        <ftp:put-command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:success>true</ftp:success>
                            <ftp:reply-code>226</ftp:reply-code>
                            <ftp:reply-string>@contains('Transfer
complete')@</ftp:reply-string>
                        </ftp:put-command-result>
                    </payload>
                </body>
            </message>
        </receive>
    </actions>
</test>
```

YAML

```
name: FtpMessageTest
actions:
  - send:
      endpoint: "ftpClient"
      message:
        body: |
          <ftp:put-command
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
            <ftp:file path="test/hello.txt" type="ASCII"/>
            <ftp:target path="/test/hello.txt"/>
          </ftp:put-command>
  - receive:
      endpoint: "ftpClient"
      message:
        body: |
          <ftp:put-command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
            <ftp:success>true</ftp:success>
            <ftp:reply-code>226</ftp:reply-code>
            <ftp:reply-string>@contains('Transfer complete')@</ftp:reply-string>
          </ftp:put-command-result>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="FtpMessageTest">
        <actions>
            <send endpoint="ftpClient">
                <message>
                    <payload>
                        <ftp:put-command
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:file path="test/hello.txt" type="ASCII"/>
                            <ftp:target path="/test/hello.txt"/>
                        </ftp:put-command>
                    </payload>
                </message>
            </send>

            <receive endpoint="ftpClient">
                <message>
                    <payload>
                        <ftp:put-command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:success>true</ftp:success>
                            <ftp:reply-code>226</ftp:reply-code>
                            <ftp:reply-string>@contains('Transfer
complete')@</ftp:reply-string>
                        </ftp:put-command-result>
                    </payload>
                </message>
            </receive>
        </actions>
    </testcase>
</spring:beans>
```

The file store operation uses the put command as message payload when sending the file request. The file content is loaded from external file resource. You can choose the transfer type `ASCII` and `BINARY`. When the file is stored on server side we receive a success result message with respective reply code and string for validation.

20.1.3. Retrieve files

We are able to retrieve files from an FTP server. We need to specify the target file path that we want to get on the server user home directory.

Java

```
@CitrusTest
public void ftpMessageTest() {
    $(send(ftpClient)
        .message(FtpMessage.get("test/hello.txt", "target/test/hello.txt",
DataType.ASCII))
    );

    $(receive(ftpClient)

.message(FtpMessage.result(getRetrieveFileCommandResult("target/test/hello.txt", new
ClassPathResource("test/hello.txt"))))
    );
}

private GetCommandResult getRetrieveFileCommandResult(String path, Resource content)
throws IOException {
    GetCommandResult result = new GetCommandResult();
    result.setSuccess(true);
    result.setReplyCode(String.valueOf(226));
    result.setReplyString("@contains('Transfer complete')@");

    GetCommandResult.File entryResult = new GetCommandResult.File();
    entryResult.setPath(path);
    entryResult.setData(FileUtils.readToString(content));
    result.setFile(entryResult);

    return result;
}
```

XML

```
<test name="FtpMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <send endpoint="ftpClient">
            <message>
                <body>
                    <payload>
                        <ftp:get-command
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:file path="test/hello.txt" type="ASCII"/>
                            <ftp:target path="target/test/hello.txt"/>
                        </ftp:get-command>
                    </payload>
                </body>
            </message>
        </send>

        <receive endpoint="ftpClient">
            <message>
                <body>
                    <payload>
                        <ftp:get-command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:success>true</ftp:success>
                            <ftp:reply-code>226</ftp:reply-code>
                            <ftp:reply-string>@contains('Transfer
complete')@</ftp:reply-string>
                            <ftp:file path="target/test/hello.txt">

<ftp:data>citrus:readFile('classpath:test/hello.txt')</ftp:data>
                            </ftp:file>
                        </ftp:get-command-result>
                    </payload>
                </body>
            </message>
        </receive>
    </actions>
</test>
```

YAML

```
name: FtpMessageTest
actions:
  - send:
      endpoint: "ftpClient"
      message:
        body: |
          <ftp:get-command
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
            <ftp:file path="test/hello.txt" type="ASCII"/>
            <ftp:target path="target/test/hello.txt"/>
          </ftp:get-command>
  - receive:
      endpoint: "ftpClient"
      message:
        body: |
          <ftp:get-command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
            <ftp:success>true</ftp:success>
            <ftp:reply-code>226</ftp:reply-code>
            <ftp:reply-string>@contains('Transfer complete')@</ftp:reply-string>
            <ftp:file path="target/test/hello.txt">
              <ftp:data>citrus:readFile('classpath:test/hello.txt')</ftp:data>
            </ftp:file>
          </ftp:get-command-result>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="FtpMessageTest">
        <actions>
            <send endpoint="ftpClient">
                <message>
                    <payload>
                        <ftp:get-command
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:file path="test/hello.txt" type="ASCII"/>
                            <ftp:target path="target/test/hello.txt"/>
                        </ftp:get-command>
                    </payload>
                </message>
            </send>

            <receive endpoint="ftpClient">
                <message>
                    <payload>
                        <ftp:get-command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:success>true</ftp:success>
                            <ftp:reply-code>226</ftp:reply-code>
                            <ftp:reply-string>@contains('Transfer
complete')@</ftp:reply-string>
                            <ftp:file path="target/test/hello.txt">

<ftp:data>citrus:readFile('classpath:test/hello.txt')</ftp:data>
                            </ftp:file>
                        </ftp:get-command-result>
                    </payload>
                </message>
            </receive>
        </actions>
    </testcase>
</spring:beans>
```

When file transfer is complete we are able to verify the file content in a command result. The file content is provided as data string.

20.1.4. List files

Listing files on the server is possible with the list command.

Java

```
@CitrusTest
public void ftpMessageTest() {
    $(send(ftpClient)
        .message(FtpMessage.list("test"))
    );

    $(receive(ftpClient)
        .message(FtpMessage.result(getListCommandResult("hello.txt")))
    );
}

private ListCommandResult getListCommandResult(String ... fileNames) {
    ListCommandResult result = new ListCommandResult();
    result.setSuccess(true);
    result.setReplyCode(String.valueOf(226));
    result.setReplyString("@contains('Closing data connection')@");

    ListCommandResult.Files expectedFiles = new ListCommandResult.Files();

    for (String fileName : fileNames) {
        ListCommandResult.Files.File entry = new ListCommandResult.Files.File();
        entry.setPath(fileName);
        expectedFiles.getFiles().add(entry);
    }

    result.setFiles(expectedFiles);

    return result;
}
```

XML

```
<test name="FtpMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <send endpoint="ftpClient">
            <message>
                <body>
                    <payload>
                        <ftp:list-command
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:target path="test" />
                        </ftp:list-command>
                    </payload>
                </body>
            </message>
        </send>

        <receive endpoint="ftpClient">
            <message>
                <body>
                    <payload>
                        <ftp:list-command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:success>true</ftp:success>
                            <ftp:reply-code>226</ftp:reply-code>
                            <ftp:reply-string>@contains('Closing data
connection')@</ftp:reply-string>
                            <ftp:files>
                              <ftp:file path="hello.txt"/>
                            </ftp:files>
                        </ftp:list-command-result>
                    </payload>
                </body>
            </message>
        </receive>
    </actions>
</test>
```

YAML

```
name: FtpMessageTest
actions:
  - send:
      endpoint: "ftpClient"
      message:
        body: |
          <ftp:list-command
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
            <ftp:target path="test" />
          </ftp:list-command>
  - receive:
      endpoint: "ftpClient"
      message:
        body: |
          <ftp:list-command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
            <ftp:success>true</ftp:success>
            <ftp:reply-code>226</ftp:reply-code>
            <ftp:reply-string>@contains('Closing data connection')@</ftp:reply-string>
            <ftp:files>
              <ftp:file path="hello.txt"/>
            </ftp:files>
          </ftp:list-command-result>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="FtpMessageTest">
        <actions>
            <send endpoint="ftpClient">
                <message>
                    <payload>
                        <ftp:list-command
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:target path="test" />
                        </ftp:list-command>
                    </payload>
                </message>
            </send>

            <receive endpoint="ftpClient">
                <message>
                    <payload>
                        <ftp:list-command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:success>true</ftp:success>
                            <ftp:reply-code>226</ftp:reply-code>
                            <ftp:reply-string>@contains('Closing data
connection')@</ftp:reply-string>
                            <ftp:files>
                              <ftp:file path="hello.txt"/>
                            </ftp:files>
                        </ftp:list-command-result>
                    </payload>
                </message>
            </receive>
        </actions>
    </testcase>
</spring:beans>
```

Listing files results in a command result that gives us the list of files on the server directory. We are able to verify that list with respective file paths.

20.2. FTP server

Now that we are able to access FTP as a client we might also want to simulate the server side. Therefore Citrus offers a server component that is listening on a port for incoming FTP connections. The server has a default home directory on the local file system specified. But you can also define home directories per user. For now let us have a look at the server configuration component:

XML

```
<citrus-ftp:server id="ftpServer">
      port="2222"
      auto-start="true"
      auto-handle-commands="MKD,PORT,TYPE"
      user-manager-properties="classpath:ftp.server.properties"/>
```

Java

```
@Bean
public FtpServer ftpListServer() {
    return CitrusEndpoints.ftp()
            .server()
            .port(2222)
            .autoLogin(true)
            .autoStart(true)
            .autoHandleCommands(Stream.of(FTPCmd.MKD.getCommand(),
                                          FTPCmd.PORT.getCommand(),

FTPCmd.TYPE.getCommand()).collect(Collectors.joining(",")))
            .userManagerProperties(new
ClassPathResource("citrus.ftp.user.properties"))
            .build();
```

The ftp server configuration is quite simple. The server starts automatically and binds to a port. With `autoLogin` and `autoHandleCommands` we can specify the behavior of the server. When `autoLogin` is enabled the server will automatically accept user login requests. With `autoHandleCommands` we can set a list of commands that should also be handled automatically so we do not have to verify those commands in a test case. The server will automatically respond with a positive command result then.

The user configuration is read from a **user-manager-property** file. Let us have a look at the content of this user management file:

```
# Password is "admin"
ftpserver.user.admin.userpassword=c7ad44cbad762a5da0a452f9e854fdc1e0e7a52a38015f23f3ea
b1d80b931dd472634dfac71cd34ebc35d16ab7fb8a90c81f975113d6c7538dc69dd8de9077ec
ftpserver.user.admin.homedirectory=target/ftp/user/admin
ftpserver.user.admin.enableflag=true
ftpserver.user.admin.writepermission=true
ftpserver.user.admin.maxloginnumber=0
ftpserver.user.admin.maxloginperip=0
ftpserver.user.admin.idletime=0
ftpserver.user.admin.uploadrate=0
ftpserver.user.admin.downloadrate=0

ftpserver.user.anonymous.userpassword=
ftpserver.user.anonymous.homedirectory=target/ftp/user/anonymous
ftpserver.user.anonymous.enableflag=true
ftpserver.user.anonymous.writepermission=false
ftpserver.user.anonymous.maxloginnumber=20
ftpserver.user.anonymous.maxloginperip=2
ftpserver.user.anonymous.idletime=300
ftpserver.user.anonymous.uploadrate=4800
ftpserver.user.anonymous.downloadrate=4800
```

The FTP server defines two accounts `citrus` and `anonymous`. Clients may authenticate to the server using these credentials. Based on the user account we can set a user workspace home directory. The server will save incoming stored files to this directory and the server will read retrieved files from that home directory.

In case you want to setup some files in that directory in order to provide it to clients, please copy those files to that home directory prior to the test.

The ftp-client connects to the server using the user credentials and is then able to store and retrieve files in a test.

You are able to define as many user for the ftp server as you like. In addition to that you have plenty of configuration possibilities per user. Citrus uses the Apache ftp server implementation. So for more details on configuration capabilities please consult the official Apache ftp server documentation.

The following listings show how to handle incoming commands representing different file operation such as store and retrieve. In the test we indicate the server response that we would link the server to respond with. Positive command results accept the client command and execute the command. As we have a fully qualified ftp server running the client can store, retrieve files and create and change directories. All incoming commands result in a file system change in the user home directory. So stored files are stored in that working directory and retrieved files are read form that directory. In the test case we only receive the commands for validation purpose and to indicate server success or failure response.

20.2.1. FTP server commands

Now we would like to use the server in a test case. Each operation that arrives on the server is automatically forwarded to the test case for validation. This means that we can verify any command on the server by using a normal receive action in our test.

Java

```
@CitrusTest
public void ftpMessageTest() {
    $(receive(ftpServer)
        .message(FtpMessage.command(FTPCmd.MKD).arguments("test"))
    );

    $(send(ftpServer)
        .message(FtpMessage.success())
    );
}
```

XML

```
<test name="FtpMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <receive endpoint="ftpServer">
            <message>
                <body>
                    <payload>
                        <ftp:command
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:signal>MKD</ftp:signal>
                            <ftp:arguments>/test</ftp:arguments>
                        </ftp:command>
                    </payload>
                </body>
            </message>
        </receive>

        <send endpoint="ftpServer">
            <message>
                <body>
                    <payload>
                        <ftp:command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:success>true</ftp:success>
                        </ftp:command-result>
                    </payload>
                </body>
            </message>
        </send>
    </actions>
</test>
```

YAML

```
name: FtpMessageTest
actions:
  - receive:
      endpoint: "ftpServer"
      message:
        body: |
          <ftp:command xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
            <ftp:signal>MKD</ftp:signal>
            <ftp:arguments>/test</ftp:arguments>
          </ftp:command>
  - send:
      endpoint: "ftpServer"
      message:
        body: |
          <ftp:command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
            <ftp:success>true</ftp:success>
          </ftp:command-result>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="FtpMessageTest">
        <actions>
            <receive endpoint="ftpServer">
                <message>
                    <payload>
                        <ftp:command
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:signal>MKD</ftp:signal>
                            <ftp:arguments>/test</ftp:arguments>
                        </ftp:command>
                    </payload>
                </message>
            </receive>

            <send endpoint="ftpServer">
                <message>
                    <payload>
                        <ftp:command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:success>true</ftp:success>
                        </ftp:command-result>
                    </payload>
                </message>
            </send>
        </actions>
    </testcase>
</spring:beans>
```

The receive action uses the command signal and argument for validation. In the sample above we receive a `MKD` signal with argument `/test` which implies a create directory command. The server respectively the test case is now able to simulate the response for this command. We respond with a success command result. Following from that the Citrus FTP server implementation will create that directory in the user home directory and respond to the client with a proper success message.

Of course you can also simulate error scenarios here. Just respond in the test with a negative command result.

20.2.2. Store files

Clients are able to store files on the server component. Each file store operation is executed in the user home directory when the command result is successful. In a test you can verify the `STOR` signal coming from the client.

Java

```
@CitrusTest
public void ftpMessageTest() {
    $(receive(ftpServer)
        .message(FtpMessage.command(FTPCmd.STOR).arguments("/test/hello.txt"))
    );

    $(send(ftpServer)
        .message(FtpMessage.success())
    );
}
```

XML

```
<test name="FtpMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <receive endpoint="ftpServer">
            <message>
                <body>
                    <payload>
                        <ftp:command
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:signal>STOR</ftp:signal>
                            <ftp:arguments>/test/hello.txt</ftp:arguments>
                        </ftp:command>
                    </payload>
                </body>
            </message>
        </receive>

        <send endpoint="ftpServer">
            <message>
                <body>
                    <payload>
                        <ftp:command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:success>true</ftp:success>
                        </ftp:command-result>
                    </payload>
                </body>
            </message>
        </send>
    </actions>
</test>
```

YAML

```
name: FtpMessageTest
actions:
  - receive:
      endpoint: "ftpServer"
      message:
        body: |
          <ftp:command xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
            <ftp:signal>STOR</ftp:signal>
            <ftp:arguments>/test/hello.txt</ftp:arguments>
          </ftp:command>
  - send:
      endpoint: "ftpServer"
      message:
        body: |
          <ftp:command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
            <ftp:success>true</ftp:success>
          </ftp:command-result>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="FtpMessageTest">
        <actions>
            <receive endpoint="ftpServer">
                <message>
                    <payload>
                        <ftp:command
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:signal>STOR</ftp:signal>
                            <ftp:arguments>/test/hello.txt</ftp:arguments>
                        </ftp:command>
                    </payload>
                </message>
            </receive>

            <send endpoint="ftpServer">
                <message>
                    <payload>
                        <ftp:command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:success>true</ftp:success>
                        </ftp:command-result>
                    </payload>
                </message>
            </send>
        </actions>
    </testcase>
</spring:beans>
```

After that you should find a new file in the user home directory with the given file path. The file transfer is automatically handled by the Citrus FTP server component.

20.2.3. Retrieve files

Clients should be able to get files from the server by using get/retrieve commands. In the request the client needs to give the target file path based on the user home directory.

Java

```
@CitrusTest
public void ftpMessageTest() {
    $(receive(ftpServer)
        .message(FtpMessage.command(FTPCmd.RETR).arguments("/test/hello.txt"))
    );

    $(send(ftpServer)
        .message(FtpMessage.success())
    );
}
```

XML

```
<test name="FtpMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <receive endpoint="ftpServer">
            <message>
                <body>
                    <payload>
                        <ftp:command
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:signal>RETR</ftp:signal>
                            <ftp:arguments>/test/hello.txt</ftp:arguments>
                        </ftp:command>
                    </payload>
                </body>
            </message>
        </receive>

        <send endpoint="ftpServer">
            <message>
                <body>
                    <payload>
                        <ftp:command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:success>true</ftp:success>
                        </ftp:command-result>
                    </payload>
                </body>
            </message>
        </send>
    </actions>
</test>
```

YAML

```
name: FtpMessageTest
actions:
  - receive:
      endpoint: "ftpServer"
      message:
        body: |
          <ftp:command xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
            <ftp:signal>RETR</ftp:signal>
            <ftp:arguments>/test/hello.txt</ftp:arguments>
          </ftp:command>
  - send:
      endpoint: "ftpServer"
      message:
        body: |
          <ftp:command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
            <ftp:success>true</ftp:success>
          </ftp:command-result>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="FtpMessageTest">
        <actions>
            <receive endpoint="ftpServer">
                <message>
                    <payload>
                        <ftp:command
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:signal>RETR</ftp:signal>
                            <ftp:arguments>/test/hello.txt</ftp:arguments>
                        </ftp:command>
                    </payload>
                </message>
            </receive>

            <send endpoint="ftpServer">
                <message>
                    <payload>
                        <ftp:command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:success>true</ftp:success>
                        </ftp:command-result>
                    </payload>
                </message>
            </send>
        </actions>
    </testcase>
</spring:beans>
```

The file request is verified with proper signal and arguments. When the server command result is positive the Citrus FTP server will transfer the file content to the calling client.

20.2.4. List files

When clients request for listing files on the server we get a list command on the server.

Java

```
@CitrusTest
public void ftpMessageTest() {
    $(receive(ftpServer)
        .message(FtpMessage.command(FTPCmd.LIST).arguments("test"))
    );

    $(send(ftpServer)
        .message(FtpMessage.success())
    );
}
```

XML

```
<test name="FtpMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <receive endpoint="ftpServer">
            <message>
                <body>
                    <payload>
                        <ftp:command
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:signal>LIST</ftp:signal>
                            <ftp:arguments>test</ftp:arguments>
                        </ftp:command>
                    </payload>
                </body>
            </message>
        </receive>

        <send endpoint="ftpServer">
            <message>
                <body>
                    <payload>
                        <ftp:command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:success>true</ftp:success>
                        </ftp:command-result>
                    </payload>
                </body>
            </message>
        </send>
    </actions>
</test>
```

YAML

```
name: FtpMessageTest
actions:
  - receive:
      endpoint: "ftpServer"
      message:
        body: |
          <ftp:command xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
            <ftp:signal>LIST</ftp:signal>
            <ftp:arguments>test</ftp:arguments>
          </ftp:command>
  - send:
      endpoint: "ftpServer"
      message:
        body: |
          <ftp:command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
            <ftp:success>true</ftp:success>
          </ftp:command-result>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="FtpMessageTest">
        <actions>
            <receive endpoint="ftpServer">
                <message>
                    <payload>
                        <ftp:command
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:signal>LIST</ftp:signal>
                            <ftp:arguments>test</ftp:arguments>
                        </ftp:command>
                    </payload>
                </message>
            </receive>

            <send endpoint="ftpServer">
                <message>
                    <payload>
                        <ftp:command-result
xmlns:ftp="http://www.citrusframework.org/schema/ftp/message">
                            <ftp:success>true</ftp:success>
                        </ftp:command-result>
                    </payload>
                </message>
            </send>
        </actions>
    </testcase>
</spring:beans>
```

As you can see the list command is verified with proper signal and arguments that specifies the target folder to list the files for. When the command result is positive the FTP server implementation will send back a proper list command result for that given directory in the user home directory.

Chapter 21. SFTP/SCP support

With Citrus it is possible to start your own sftp server for accepting incoming client requests. You can also use Citrus as a SFTP client to send FTP commands. The next sections deal with SFTP connectivity.

The SFTP components in Citrus are maintained in their own Maven module. So you should add the module as Maven dependency to your project accordingly.

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-ftp</artifactId>
  <version>2.7.6</version>
</dependency>
```

As Citrus provides a customized SFTP configuration schema for the Spring application context configuration files we have to add name to the top level **beans** element. Simply include the sftp-config namespace in the configuration XML files as follows.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:citrus="http://www.citrusframework.org/schema/config"
    xmlns:citrus-sftp="http://www.citrusframework.org/schema/sftp/config"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.citrusframework.org/schema/config
    http://www.citrusframework.org/schema/config/citrus-config.xsd
    http://www.citrusframework.org/schema/http/config
    http://www.citrusframework.org/schema/sftp/config/citrus-sftp-config.xsd">

      [...]

      </beans>
```

Now we are ready to use the customized Citrus SFTP configuration elements with the citrus-sftp namespace prefix.

21.1. SFTP client

We want to use Citrus to connect to some FTP server using secure file transfer and authentication. Citrus offers a client component doing exactly this SFTP client connection.

XML

```
<citrus-sftp:client id="sftpClient"
           strict-host-checking="false"
           port="2222"
           username="citrus"
           private-key-path="classpath:ssh/citrus.priv"
           timeout="10000"/>
```

Java

```
@Bean
public SftpClient sftpClient() {
    return CitrusEndpoints.sftp()
            .client()
            .strictHostChecking(false)
            .port(2222)
            .username("citrus")
            .privateKeyPath("classpath:ssh/citrus.priv")
            .build();
}
```

The configuration above describes a Citrus sftp client. The sftp-client connects to the server using the user credentials and/or the private key authentication. The client will automatically authenticate to the server when opening a new connection. As the Citrus stfp client supports various authentication methods, the following order is default: `publickey,password,keyboard-interactive`. You're able to configure the default order with the `preferred-authentications` attribute in XML or the `preferredAuthentications` method in Java.

XML

```
<citrus-sftp:client id="sftpClient"
           strict-host-checking="false"
           port="2222"
           username="citrus"
           private-key-path="classpath:ssh/citrus.priv",
           preferred-authentications="publickey,password,gssapi-with-mic,keyboard-
interactive"
           timeout="10000"/>
```

Java

```
@Bean
public SftpClient sftpClient() {
    return CitrusEndpoints.sftp()
            .client()
            .strictHostChecking(false)
            .port(2222)
            .username("citrus")
            .privateKeyPath("classpath:ssh/citrus.priv")
            .preferredAuthentications("publickey,password,gssapi-with-mic,keyboard-
interactive")
            .build();
}
```

21.1.1. SFTP client commands

In a test case you are now able to use the client to push commands to the server.

XML DSL

```
<send endpoint="sftpClient">
  <message>
    <payload>
      <ftp:command>
        <ftp:signal>MKD</ftp:signal>
        <ftp:arguments>test</ftp:arguments>
      </ftp:command>
    </payload>
  </message>
</send>

<receive endpoint="sftpClient">
  <message>
    <payload>
      <ftp:command-result>
        <ftp:success>true</ftp:success>
        <ftp:reply-code>257</ftp:reply-code>
        <ftp:reply-string>257 Pathname created</ftp:reply-string>
      </ftp:command-result>
    </payload>
  </message>
</receive>
```

Java DSL

```
send(sftpClient)
    .message(FtpMessage.command(FTPCmd.MKD).arguments("test"));

CommandResult result = new CommandResult();
result.setSuccess(true);
result.setReplyCode(String.valueOf(257));
result.setReplyString("257 Pathname created");

receive(sftpClient)
    .message(FtpMessage.result(result));
```

As you can see most of the sftp communication parameters are specified in a ftp command message. Citrus automatically converts those information to proper FTP commands and response messages.

21.1.2. Store files

The client is able to store files on the server using file transfer.

XML DSL

```
<send endpoint="sftpClient">
  <message>
    <payload>
      <ftp:put-command>
        <ftp:file path="test/hello.txt" type="ASCII"/>
        <ftp:target path="/test/hello.txt"/>
      </ftp:put-command>
    </payload>
  </message>
</send>

<receive endpoint="sftpClient">
  <message>
    <payload>
      <ftp:put-command-result>
        <ftp:success>true</ftp:success>
        <ftp:reply-code>226</ftp:reply-code>
        <ftp:reply-string>@contains('Transfer complete')@</ftp:reply-string>
      </ftp:put-command-result>
    </payload>
  </message>
</receive>
```

Java DSL

```
send(sftpClient)
        .message(FtpMessage.put("test/hello.txt", DataType.ASCII).arguments(""));

PutCommandResult result = new PutCommandResult();
        result.setSuccess(true);
        result.setReplyCode(String.valueOf(226));
        result.setReplyString("@contains(Transfer complete)@");

receive(sftpClient)
        .message(FtpMessage.result(result));
```

The file store operation uses the put command as message payload when sending the file request. The file content is loaded from external file resource. You can choose the transfer type `ASCII` and `BINARY`. When the file is stored on server side we receive a success result message with respective reply code and string for validation.

21.1.3. Retrieve files

We are able to retrieve files from a SFTP server. We need to specify the target file path that we want to get on the server user home directory.

XML DSL

```
<send endpoint="sftpClient">
  <message>
    <payload>
      <ftp:get-command>
        <ftp:file path="test/hello.txt" type="ASCII"/>
        <ftp:target path="target/test/hello.txt"/>
      </ftp:get-command>
    </payload>
  </message>
</send>

<receive endpoint="sftpClient">
  <message>
    <payload>
      <ftp:get-command-result>
        <ftp:success>true</ftp:success>
        <ftp:reply-code>226</ftp:reply-code>
        <ftp:reply-string>@contains('Transfer complete')@</ftp:reply-string>
        <ftp:file path="target/test/hello.txt">
          <ftp:data>citrus:readFile('classpath:test/hello.txt')</ftp:data>
        </ftp:file>
      </ftp:get-command-result>
    </payload>
  </message>
</receive>
```

Java DSL

```
send(sftpClient)
        .message(FtpMessage.get("test/hello.txt", "target/test/hello.txt",
DataType.ASCII));

receive(sftpClient)

.message(FtpMessage.result(getRetrieveFileCommandResult("target/test/hello.txt", new
ClassPathResource("test/hello.txt"))));
```

```
private GetCommandResult getRetrieveFileCommandResult(String path, Resource content)
throws IOException {
    GetCommandResult result = new GetCommandResult();
    result.setSuccess(true);
    result.setReplyCode(String.valueOf(226));
    result.setReplyString("@contains('Transfer complete')@");

    GetCommandResult.File entryResult = new GetCommandResult.File();
    entryResult.setPath(path);
    entryResult.setData(FileUtils.readToString(content));
    result.setFile(entryResult);

    return result;
}
```

When file transfer is complete we are able to verify the file content in a command result. The file content is provided as data string.

21.1.4. List files

Listing files on the server is possible with the list command.

XML

```
<send endpoint="sftpClient">
  <message>
    <payload>
      <ftp:list-command>
        <ftp:target path="test" />
      </ftp:list-command>
    </payload>
  </message>
</send>

<receive endpoint="sftpClient">
  <message>
    <payload>
      <ftp:list-command-result>
        <ftp:success>true</ftp:success>
        <ftp:reply-code>150</ftp:reply-code>
        <ftp:reply-string>List files complete</ftp:reply-string>
        <ftp:files>
          <ftp:file path="."/>
          <ftp:file path=".."/>
          <ftp:file path="hello.txt"/>
        </ftp:files>
      </ftp:list-command-result>
    </payload>
  </message>
</receive>
```

Java

```
send(sftpClient)
        .message(FtpMessage.list("test"));

receive(sftpClient)
        .message(FtpMessage.result(getListCommandResult("hello.txt")));
```

```
private ListCommandResult getListCommandResult(String ... fileNames) {
    ListCommandResult result = new ListCommandResult();
    result.setSuccess(true);
    result.setReplyCode(String.valueOf(226));
    result.setReplyString("@contains('Closing data connection')@");

    ListCommandResult.Files.File currentDir = new ListCommandResult.Files.File();
    currentDir.setPath(".");
    expectedFiles.getFiles().add(currentDir);

    ListCommandResult.Files.File parentDir = new ListCommandResult.Files.File();
    parentDir.setPath("..");
    expectedFiles.getFiles().add(parentDir);

    ListCommandResult.Files expectedFiles = new ListCommandResult.Files();

    for (String fileName : fileNames) {
        ListCommandResult.Files.File entry = new ListCommandResult.Files.File();
        entry.setPath(fileName);
        expectedFiles.getFiles().add(entry);
    }

    result.setFiles(expectedFiles);

    return result;
}
```

Listing files results in a command result that gives us the list of files on the server directory. We are able to verify that list with respective file paths.

21.2. SFTP server

Now that we are able to access SFTP as a client we might also want to simulate the server side. Therefore Citrus offers a server component that is listening on a port for incoming SFTP connections. The server has a default home directory on the local file system specified. But you can also define home directories per user. For now let us have a look at the server configuration component:

XML

```
<citrus-sftp:server id="sftpServer"
                    port="2222"
                    auto-start="true"
                    user="citrus"
                    password="admin"
                    allowed-key-path="classpath:ssh/citrus_pub.pem"/>
```

Java

```
@Bean
public SftpServer sftpServer() {
    return CitrusEndpoints.sftp()
            .server()
            .port(2222)
            .autoStart(true)
            .user("citrus")
            .password("admin")
            .allowedKeyPath("classpath:ssh/citrus_pub.pem")
            .build();
}
```

The **sftpServer** is a small but fully qualified SFTP server implementation in Citrus. The server receives a `user` that defines the user account and its home directory. All commands will be performed in this user home directory. You can set the user home directory using the `userHomePath` attribute on the server. By default this is a directory located in `${user.dir}/target/{serverName}/home/{user}`.

In case you want to setup some files in that directory in order to provide it to clients, please copy those files to that home directory prior to the test. The server adds the public key to the list of allowed keys.

The following listings show how to handle incoming commands representing different file operation such as store and retrieve. In the test we indicate the server response that we would link the server to respond with. Positive command results accept the client command and execute the command. As we have a fully qualified sftp server running the client can store, retrieve files and create and change directories. All incoming commands result in a file system change in the user home directory. So stored files are stored in that working directory and retrieved files are read form that directory. In the test case we only receive the commands for validation purpose and to indicate server success or failure response.

21.2.1. SFTP server commands

Now we would like to use the server in a test case. Each operation that arrives on the server is automatically forwarded to the test case for validation. This means that we can verify any command on the server by using a normal receive action in our test.

XML DSL

```
<receive endpoint="sftpServer">
  <message>
    <payload>
      <ftp:command>
        <ftp:signal>MKD</ftp:signal>
        <ftp:arguments>/test</ftp:arguments>
      </ftp:command>
    </payload>
  </message>
</receive>

<send endpoint="sftpServer">
  <message>
    <payload>
      <ftp:command-result>
        <ftp:success>true</ftp:success>
      </ftp:command-result>
    </payload>
  </message>
</send>
```

Java DSL

```
receive(sftpServer)
    .message(FtpMessage.command(FTPCmd.MKD).arguments("test"));

send(sftpServer)
    .message(FtpMessage.success());
```

The receive action uses the command signal and argument for validation. In the sample above we receive a `MKD` signal with argument `/test` which implies a create directory command. The server respectively the test case is now able to simulate the response for this command. We respond with a success command result. Following from that the Citrus SFTP server implementation will create that directory in the user home directory and respond to the client with a proper success message.

Of course you can also simulate error scenarios here. Just respond in the test with a negative command result.

21.2.2. Store files

Clients are able to store files on the server component. Each file store operation is executed in the user home directory when the command result is successful. In a test you can verify the `STOR` signal coming from the client.

XML DSL

```
<echo>
  <message>Store file on server</message>
</echo>

<receive endpoint="sftpServer">
  <message>
    <payload>
      <ftp:put-command>
        <ftp:signal>STOR</ftp:signal>
        <ftp:file path="@ignore@" type="ASCII"/>
        <ftp:target path="/test/hello.txt"/>
      </ftp:put-command>
    </payload>
  </message>
</receive>

<send endpoint="sftpServer">
  <message>
    <payload>
      <ftp:command-result>
        <ftp:success>true</ftp:success>
      </ftp:command-result>
    </payload>
  </message>
</send>
```

Java DSL

```
receive(sftpServer)
        .message(put("@ignore@","/test/hello.txt", DataType.ASCII));

send(sftpServer)
        .message(FtpMessage.success());
```

After that you should find a new file in the user home directory with the given file path. The file transfer is automatically handled by the Citrus SFTP server component.

21.2.3. Retrieve files

Clients should be able to get files from the server by using get/retrieve commands. In the request the client needs to give the target file path based on the user home directory.

XML DSL

```
<echo>
  <message>Retrieve file from server</message>
</echo>

<receive endpoint="sftpServer">
  <message>
    <payload>
      <ftp:get-command>
        <ftp:signal>RETR</ftp:signal>
        <ftp:file path="/test/hello.txt" type="ASCII"/>
        <ftp:target path="@ignore@"/>
      </ftp:get-command>
    </payload>
  </message>
</receive>

<send endpoint="sftpServer">
  <message>
    <payload>
      <ftp:command-result>
        <ftp:success>true</ftp:success>
      </ftp:command-result>
    </payload>
  </message>
</send>
```

Java DSL

```
receive(sftpServer)
        .message(FtpMessage.get("/test/hello.txt", "@ignore@", DataType.ASCII));

send(sftpServer)
        .message(FtpMessage.success());
```

The file request is verified with proper signal and arguments. When the server command result is positive the Citrus SFTP server will transfer the file content to the calling client.

21.2.4. List files

When clients request for listing files on the server we get a list command on the server.

XML DSL

```
<receive endpoint="sftpServer">
  <message>
    <payload>
      <ftp:command>
        <ftp:signal>LIST</ftp:signal>
        <ftp:arguments>test</ftp:arguments>
      </ftp:command>
    </payload>
  </message>
</receive>

<send endpoint="sftpServer">
  <message>
    <payload>
      <ftp:command-result>
        <ftp:success>true</ftp:success>
      </ftp:command-result>
    </payload>
  </message>
</send>
```

Java DSL

```
receive(sftpServer)
        .message(FtpMessage.command(FTPCmd.LIST).arguments("test"));

send(sftpServer)
        .message(FtpMessage.success());
```

As you can see the list command is verified with proper signal and arguments that specifies the target folder to list the files for. When the command result is positive the SFTP server implementation will send back a proper list command result for that given directory in the user home directory.

21.3. SCP client

As Citrus provides a customized SCP configuration schema for the Spring application context configuration files we have to add name to the top level **beans** element. Simply include the scp-config namespace in the configuration XML files as follows.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:citrus="http://www.citrusframework.org/schema/config"
    xmlns:citrus-scp="http://www.citrusframework.org/schema/scp/config"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.citrusframework.org/schema/config
    http://www.citrusframework.org/schema/config/citrus-config.xsd
    http://www.citrusframework.org/schema/http/config
    http://www.citrusframework.org/schema/scp/config/citrus-scp-config.xsd">

      [...]

      </beans>
```

Now we are ready to use the customized Citrus SCP configuration elements with the citrus-scp namespace prefix.

We want to use Citrus to connect to some FTP server using secure file copy with SCP. Citrus offers a client component doing exactly this SCP client connection.

XML

```
<citrus-scp:client id="scpClient"
               port="2222"
               username="citrus"
               password="admin"
               private-key-path="classpath:ssh/citrus.priv"/>
```

Java

```
@Bean
public ScpClient scpClient() {
    return CitrusEndpoints.scp()
            .client()
            .port(2222)
            .username("citrus")
            .password("admin")
            .privateKeyPath("classpath:ssh/citrus.priv")
            .build();
}
```

The configuration above describes a Citrus scp client. The scp-client connects to the server using the user credentials and/or the private key authentication. The client will automatically authenticate to the server when opening a new connection.

21.3.1. Store files

The client is able to store files on the server using file transfer.

XML DSL

```
<send endpoint="scpClient">
  <message>
    <payload>
      <ftp:put-command>
        <ftp:file path="test/hello.txt" type="ASCII"/>
        <ftp:target path="/test/hello.txt"/>
      </ftp:put-command>
    </payload>
  </message>
</send>

<receive endpoint="scpClient">
  <message>
    <payload>
      <ftp:put-command-result>
        <ftp:success>true</ftp:success>
        <ftp:reply-code>226</ftp:reply-code>
        <ftp:reply-string>@contains('Transfer complete')@</ftp:reply-string>
      </ftp:put-command-result>
    </payload>
  </message>
</receive>
```

Java DSL

```
send(scpClient)
        .message(FtpMessage.put("test/hello.txt", DataType.ASCII).arguments(""));

PutCommandResult result = new PutCommandResult();
        result.setSuccess(true);
        result.setReplyCode(String.valueOf(226));
        result.setReplyString("@contains(Transfer complete)@");

receive(scpClient)
        .message(FtpMessage.result(result));
```

The file store operation uses the put command as message payload when sending the file request. The file content is loaded from external file resource. You can choose the transfer type `ASCII` and `BINARY`. When the file is stored on server side we receive a success result message with respective reply code and string for validation.

21.3.2. Retrieve files

We are able to retrieve files from a SFTP server. We need to specify the target file path that we want to get on the server user home directory.

XML DSL

```
<send endpoint="scpClient">
  <message>
    <payload>
      <ftp:get-command>
        <ftp:file path="test/hello.txt" type="ASCII"/>
        <ftp:target path="target/test/hello.txt"/>
      </ftp:get-command>
    </payload>
  </message>
</send>

<receive endpoint="scpClient">
  <message>
    <payload>
      <ftp:get-command-result>
        <ftp:success>true</ftp:success>
        <ftp:reply-code>226</ftp:reply-code>
        <ftp:reply-string>@contains('Transfer complete')@</ftp:reply-string>
        <ftp:file path="target/test/hello.txt">
          <ftp:data>citrus:readFile('classpath:test/hello.txt')</ftp:data>
        </ftp:file>
      </ftp:get-command-result>
    </payload>
  </message>
</receive>
```

Java DSL

```
send(scpClient)
        .message(FtpMessage.get("test/hello.txt", "target/test/hello.txt",
DataType.ASCII));

receive(scpClient)

.message(FtpMessage.result(getRetrieveFileCommandResult("target/test/hello.txt", new
ClassPathResource("test/hello.txt"))));
```

```
private GetCommandResult getRetrieveFileCommandResult(String path, Resource content)
throws IOException {
    GetCommandResult result = new GetCommandResult();
    result.setSuccess(true);
    result.setReplyCode(String.valueOf(226));
    result.setReplyString("@contains('Transfer complete')@");

    GetCommandResult.File entryResult = new GetCommandResult.File();
    entryResult.setPath(path);
    entryResult.setData(FileUtils.readToString(content));
    result.setFile(entryResult);

    return result;
}
```

When file transfer is complete we are able to verify the file content in a command result. The file content is provided as data string.

Chapter 22. File support

In chapter message-channels we have discussed the native Spring Integration channel support which enables Citrus to interact with all Spring Integration messaging adapter implementations. This is a fantastic way to extend Citrus for additional transports. This interaction now comes handy when writing and reading files from the file system in Citrus.

22.1. Write files

We want to use the Spring Integration file adapter for both reading and writing files with a local directory. Citrus can easily connect to this file adapter implementation with its message channel support. Citrus message sender and receiver speak to message channels that are connected to the Spring Integration file adapters.

```
<citrus-si:channel-endpoint id="fileEndpoint" channel="fileChannel"/>

<file:outbound-channel-adapter id="fileOutboundAdapter"
      channel="fileChannel"
      directory="file:${some.directory.property}"/>

<si:channel id="fileChannel"/>
```

The configuration above describes a Citrus message channel endpoint connected to a Spring Integration outbound file adapter that writes messages to a storage directory. With this combination you are able to write files to a directory in your Citrus test case. The test case uses the channel endpoint in its send action and the endpoint interacts with the Spring Integration file adapter so sending out the file.

The Spring Integration file adapter configuration components add a new namespace to our Spring application context. See this template which holds all necessary namespaces and schema locations:

```
<?xml version="1.0" encoding="UTF-8"?>
        <beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:citrus="http://www.citrusframework.org/schema/config"
        xmlns:si="http://www.springframework.org/schema/integration"
        xmlns:file="http://www.springframework.org/schema/integration/file"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.citrusframework.org/schema/config
        http://www.citrusframework.org/schema/config/citrus-config.xsd
        http://www.springframework.org/schema/integration
        http://www.springframework.org/schema/integration/spring-integration.xsd
        http://www.springframework.org/schema/integration/file
        http://www.springframework.org/schema/integration/file/spring-integration-
file.xsd">
        </beans>
```

22.2. Read files

The next program listing shows a possible inbound file communication. So the Spring Integration file inbound adapter will read files from a storage directory and publish the file contents to a message channel. Citrus can then receive those files as messages in a test case via the channel endpoint and validate the file contents for instance.

```
<file:inbound-channel-adapter  id="fileInboundAdapter"
        channel="fileChannel"
        directory="file:${some.directory.property}">
    <si:poller fixed-rate="100"/>
</file:inbound-channel-adapter>

<si:channel id="fileChannel">
    <si:queue capacity="25"/>
    <si:interceptors>
        <bean
class="org.springframework.integration.transformer.MessageTransformingChannelIntercept
or">
            <constructor-arg>
                <bean
class="org.springframework.integration.file.transformer.FileToStringTransformer"/>
            </constructor-arg>
        </bean>
    </si:interceptors>
</si:channel>

<citrus-si:channel-endpoint id="fileEndpoint" channel="fileChannel"/>
```

The file inbound adapter constructs Java file objects as the message payload by default. Citrus can only work on String message payloads. So we need a file transformer that converts the file objects to String payloads representing the file's content.

This file adapter example shows how easy Citrus can work hand in hand with Spring Integration adapter implementations. The message channel support is a fantastic way to extend the transport and protocol support in Citrus by connecting with the very good Spring Integration adapter implementations. Have a closer look at the Spring Integration project for more details and other adapter implementations that you can use with Citrus integration testing.

Chapter 23. Selenium support

Selenium is a very popular tool for testing user interfaces with browser automation. Citrus is able to integrate with the Selenium Java API in order to execute Selenium commands.

> The Selenium test components in Citrus are kept in a separate Maven module. If not already done, you have to include the module as Maven dependency to your project

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-selenium</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

Citrus provides a "citrus-selenium" configuration namespace and schema definition for Selenium related components and actions. Include this namespace into your Spring configuration in order to use the Citrus Selenium configuration elements. The namespace URI and schema location are added to the Spring configuration XML file as follows.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:citrus-selenium="http://www.citrusframework.org/schema/selenium/config"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.citrusframework.org/schema/selenium/config
       http://www.citrusframework.org/schema/selenium/config/citrus-selenium-
config.xsd">

    [...]

</beans>
```

After that you are able to use customized Citrus XML elements in order to define the Spring beans.

23.1. Selenium browser

Selenium uses browser automation in order to simulate the user interact with web applications. You can configure the Selenium browser and web driver as Spring bean.

```
<citrus-selenium:browser id="seleniumBrowser"
                type="firefox"
                start-page="http://citrusframework.org"/>
```

The Selenium browser component supports different browser types for the commonly used browsers out in the wild.

- **htmlunit**
- **firefox**
- **safari**
- **chrome**
- **googlechrome**
- **internet explorer**
- **edge**
- **custom**

Html unit is the default browser type and represents a headless browser that executed without displaying the graphical user interface. In case you need a totally different browser or you need to customize the Selenium web driver you can use the *type="custom"* in combination with a web driver reference:

```
<citrus-selenium:browser id="mySeleniumBrowser"
                type="custom"
                web-driver="operaWebDriver"/>

<bean id="operaWebDriver" class="org.openqa.selenium.opera.OperaDriver"/>
```

Now Citrus is using the customized Selenium web driver implementation.

When using Firefox as browser you may also want to set the optional properties **firefox-profile** and **version**.

```
<citrus-selenium:browser id="mySeleniumBrowser"
                type="firefox"
                firefox-profile="firefoxProfile"
                version="FIREFOX_38"
                start-page="http://citrusframework.org"/>

<bean id="firefoxProfile" class="org.openqa.selenium.firefox.FirefoxProfile"/>
```

Now Citrus is able to execute Selenium operations as a user.

23.2. Selenium actions

We have several Citrus test actions each representing a Selenium command. These actions can be part of a Citrus test case. As a prerequisite we have to enable the Selenium specific test actions in our XML test as follows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:selenium="http://www.citrusframework.org/schema/selenium/testcase"
        xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.citrusframework.org/schema/selenium/testcase
        http://www.citrusframework.org/schema/selenium/testcase/citrus-selenium-
testcase.xsd">

    [...]

</beans>
```

We added a special selenium namespace with prefix **selenium:** so now we can start to add Selenium test actions to the test case:

XML DSL

```
<testcase name="SeleniumCommandIT">
    <actions>
      <selenium:start browser="webBrowser"/>

      <selenium:navigate page="http://localhost:8080"/>

      <selenium:find>
        <selenium:element tag-name="h1" text="Welcome!">
          <selenium:styles>
            <selenium:style name="font-size" value="40px"/>
          </selenium:styles>
        </selenium:element>
      </selenium:find>

      <selenium:click>
        <selenium:element id="ok-button"/>
      </selenium:click>
    </actions>
</testcase>
```

In this very simple example we first start the Selenium browser instance. After that we can continue to use Selenium commands without browser attribute explicitly set. Citrus knows which browser instance is currently active and will automatically use this opened browser instance. Next in this example we find some element on the displayed page by its tag-name and text. We also validate the element style *font-size* to meet the expected value *40px* in this step.

In addition to that the example performs a click operation on the element with the id *ok-button*. Selenium supports element find operations on different properties:

id	finds element based on the *id* attribute
name	finds element based on the *name* attribute
tag-name	finds element based on the *tag name*
class-name	finds element based on the css *class name*
link-text	finds link element based on the *link-text*
xpath	finds element based on XPath evaluation in the DOM

Based on that we can execute several Selenium commands in a test case and validate the results such as web elements. Citrus supports the following Selenium commands with respective test actions:

selenium:start	Start the browser instance
selenium:find	Finds element on current page and validates element properties
selenium:click	Performs click operation on element
selenium:hover	Performs hover operation on element
selenium:navigate	Navigates to new page url (including history back, forward and refresh)
selenium:set-input	Finds input element and sets value
selenium:check-input	Finds checkbox element and sets/unsets value
selenium:dropdown-select	Finds dropdown element and selects single or multiple value/s
selenium:fill-form	Fills form with given values and optionally submits form afterwards. Supports setting multiple fields at the same time.
selenium:page	Instantiate page object with dependency injection and execute page action with verification
selenium:open	Open new window
selenium:close	Close window by given name

selenium:switch	Switch focus to window with given name
selenium:wait-until	Wait for element to be *hidden* or *visible*
selenium:alert	Access current alert dialog (with action *access* or *dismiss*)
selenium:screenshot	Makes screenshot of current page
selenium:store-file	Store file to temporary browser directory
selenium:get-stored-file	Gets stored file from temporary browser directory
selenium:javascript	Execute Javascript code in browser
selenium:clear-cache	Clear browser cache and all cookies
selenium:stop	Stops the browser instance

Up to now we have only used the Citrus XML DSL. Of course all Selenium commands are also available in Java DSL as the next example shows.

Java DSL

```
@Autowired
private SeleniumBrowser seleniumBrowser;

@CitrusTest
public void seleniumTest() {
    selenium().start(seleniumBrowser);

    selenium().navigate("http://localhost:8080");

    selenium().find().element(By.id("header"));
                .tagName("h1")
                .enabled(true)
                .displayed(true)
                .text("Welcome!")
                .style("font-size", "40px");

    selenium().click().element(By.linkText("Click Me!"));
}
```

Now let’s have a closer look at the different Selenium test actions supported in Citrus.

23.3. Start/stop browser

You can start and stop the browser instance with a test action. This instantiates a new browser window and prepares everything for interacting with the web interface.

XML DSL

```
<selenium:start browser="seleniumBrowser"/>

<!-- Do something in browser -->

<selenium:stop browser="seleniumBrowser"/>
```

Java DSL

```
selenium().start(seleniumBrowser);

// do something in browser

selenium().stop(seleniumBrowser);
```

After starting a browser instance Citrus will automatically use this very same browser instance in all further Selenium actions. This mechanism is based on a test variable (**selenium_browser**) that is automatically set. All other test actions are able to load the current browser instance by reading this test variable before execution. In case you need to explicitly use a different browser instance than the active instance you can add the **browser** attribute to all Selenium test actions.

> It is a good idea to start and stop the browser instance before each test case. This makes sure that tests are also executable in single run and it always sets up a new browser instance so tests will not influence each other.

23.4. Find

The find element test action searches for an element on the current page. The element is specified by one of the following settings:

id	finds element based on the *id* attribute
name	finds element based on the *name* attribute
tag-name	finds element based on the *tag name*
class-name	finds element based on the css *class name*
link-text	finds link element based on the *link-text*

xpath finds element based on XPath evaluation in the DOM

The find element action will automatically fail in case there is no such element on the current page. In case the element is found you can add additional attributes and properties for further element validation:

XML DSL

```
<selenium:find>
  <selenium:element tag-name="h1" text="Welcome!">
    <selenium:styles>
      <selenium:style name="font-size" value="40px"/>
    </selenium:styles>
  </selenium:element>
</selenium:find>

<selenium:find>
  <selenium:element id="ok-button" text="Ok" enabled="true" displayed="true">
    <selenium:attributes>
      <selenium:attribute name="type" value="submit"/>
    </selenium:attributes>
  </selenium:element>
</selenium:find>
```

Java DSL

```
selenium().find().element(By.tagName("h1"))
        .text("Welcome!")
        .style("font-size", "40px");

selenium().find().element(By.id("ok-button"))
        .tagName("button")
        .enabled(true)
        .displayed(true)
        .text("Ok")
        .style("color", "red")
        .attribute("type", "submit");
```

The example above finds the **h1** element by its tag name and validates the text and css style attributes. Secondly the **ok-button** is validated with expected enabled, displayed, text, style and attribute values. The elements must be present on the current page and all expected element properties have to match. Otherwise the test action and the test case is failing with validation errors.

23.5. Click

The action performs a click operation on the element.

XML DSL

```
<selenium:click>
  <selenium:element link-text="Click Me!"/>
</selenium:click>
```

Java DSL

```
selenium().click().element(By.linkText("Click Me!"));
```

23.6. Hover

The action performs a hover operation on the element.

XML DSL

```
<selenium:hover>
  <selenium:element link-text="Find Me!"/>
</selenium:hover>
```

Java DSL

```
selenium().hover().element(By.linkText("Find Me!"));
```

23.7. Form input actions

The following actions are used to access form input elements such as text fields, checkboxes and dropdown lists.

XML DSL

```
<selenium:set-input value="Citrus">
  <selenium:element name="username"/>
</selenium:set-input>

<selenium:check-input checked="true">
  <selenium:element xpath="//input[@type='checkbox']"/>
</selenium:check-input>

<selenium:dropdown-select option="happy">
  <selenium:element id="user-mood"/>
</selenium:dropdown-select>
```

Java DSL

```
selenium().setInput("Citrus").element(By.name("username"));
selenium().checkInput(true).element(By.xpath("//input[@type='checkbox']"));

selenium().select("happy").element(By.id("user-mood"));
```

The actions above select dropdown options and set user input on text fields and checkboxes. As usual the form elements are selected by some properties such as ids, names or xpath expressions.

23.8. Fill form

The fill form action sets multiple form fields with given values and optionally submits the form afterwards.

XML DSL

```
<selenium:fill-form submit="save">
  <selenium:fields>
    <selenium:field id="username" value="foo_user"/>
    <selenium:field id="password" value="${secret}"/>
  </selenium:fields>
</selenium:fill-form>
```

Java DSL

```
selenium().fillForm()
        .field("username", "foo_user")
        .field("password", "${secret}")
        .submit("save");
```

The actions above set the form fields identified by its id and sets the given values. The actions supports input fields, checkboxes and drop-down select controls. Optionally the form gets submitted by clicking on the given submit button once all fields are set.

The user also has the opportunity to set the form fields via Json object where each property in the Json object represents a form field.

XML DSL

```
<selenium:fill-form submit="save">
    <selenium:json>
      {
        "username": "foo_user",
        "password": "secret"
      }
    </selenium:json>
</selenium:fill-form>
```

Java DSL

```
selenium().fillForm()
        .fromJson("""
            {
              "username": "foo_user",
              "password": "secret"
            }
        """)
        .submit(By.id("save");
```

This will set the form fields `username` and `password` with the given values before submitting the form.

23.9. Page actions

Page objects are a well known pattern when using Selenium. The page objects define elements that the page is working with. In addition to that the page objects define actions that can be executed from outside. This object-oriented approach for accessing pages and their elements is a very good idea. Let's have a look at a sample page object.

```
public class UserFormPage implements WebPage {

    @FindBy(id = "userForm")
    private WebElement form;

    @FindBy(id = "username")
    private WebElement userName;

    /**
     * Sets the user name.
     */
    public void setUserName(String value, TestContext context) {
        userName.clear();
        userName.sendKeys(value);
    }

    /**
     * Submits the form.
     * @param context
     */
    public void submit(TestContext context) {
        form.submit();
    }
}
```

As you can see the page object is a Java POJO that implements the **WebPage** interface. The page defines **WebElement** members. These are automatically injected by Citrus and Selenium based on the **FindBy** annotation. Now the test case is able to load that page object and execute some action

methods on the page such as *setUserName* or *submit*.

XML DSL

```
<selenium:page type="org.citrusframework.selenium.pages.UserFormPage"
               action="setUserName">
  <selenium:arguments>
    <selenium:argument>Citrus</selenium:argument>
  </selenium:arguments>
</selenium:page>

<selenium:page type="org.citrusframework.selenium.pages.UserFormPage"
               action="submit"/>
```

Java DSL

```
selenium().page(UserFormPage.class).argument("Citrus").execute("setUserName");

selenium().page(UserFormPage.class).execute("submit");
```

The page object class is automatically loaded and instantiated with dependency injection for all *FindBy* annotated web elements. After that the action method is executed. The action methods can also have method parameters as seen in *setUserName*. The value parameter is automatically set when calling the method.

Methods can also use the optional parameter *TestContext*. With this context you can access the current test context with all test variables for instance. This method parameter should always be the last parameter.

23.10. Page validation

We can also use page object for validation purpose. The page object is loaded and instantiated as described in previous section. Then the page validator is called. The validator performs assertions and validation operations with the page object. Let's see a sample page validator:

```
public class UserFormValidator implements PageValidator<UserFormPage> {

    @Override
    public void validate(UserFormPage webPage, SeleniumBrowser browser, TestContext
context) {
        Assert.isTrue(webPage.getUserName() != null);

Assert.isTrue(StringUtils.hasText(webPage.getUserName().getAttribute("value")));
    }
}
```

The page validator is called with the web page instance, the browser and the test context. The validator should assert page objects and web elements for validation purpose. In a test case we can

call the validator to validate the page.

XML DSL

```
<bean id ="userFormValidator"
class="org.citrusframework.selenium.pages.UserFormValidator"/>

<selenium:page type="org.citrusframework.selenium.pages.UserFormPage"
               action="validate"
               validator="userFormValidator"/>
```

Java DSL

```
@Autowired
private UserFormValidator userFormValidator;

selenium().page(UserFormPage.class).execute("validate").validator(userFormValidator);
```

Instead of using a separate validator class you can also put the validation method to the page object itself. Then page object and validation is done within the same class:

```
public class UserFormPage implements WebPage, PageValidator<UserFormPage> {

    @FindBy(id = "userForm")
    private WebElement form;

    @FindBy(id = "username")
    private WebElement userName;

    /**
     * Sets the user name.
     */
    public void setUserName(String value, TestContext context) {
        userName.clear();
        userName.sendKeys(value);
    }

    /**
     * Submits the form.
     * @param context
     */
    public void submit(TestContext context) {
        form.submit();
    }

    @Override
    public void validate(UserFormPage webPage, SeleniumBrowser browser, TestContext
context) {
        Assert.isTrue(userName != null);
        Assert.isTrue(StringUtils.hasText(userName.getAttribute("value")));
        Assert.isTrue(form != null);
    }
}
```

XML DSL

```
<selenium:page type="org.citrusframework.selenium.pages.UserFormPage"
               action="validate"/>
```

Java DSL

```
selenium().page(UserFormPage.class).execute("validate");
```

23.11. Wait

Sometimes it is required to wait for an element to appear or disappear on the current page. The wait action will wait a given time for the element status to be *visible* or *hidden*.

XML DSL

```
<selenium:wait until="hidden">
    <selenium:element id="info-dialog"/>
</selenium:wait>
```

Java DSL

```
selenium().waitUntil().hidden().element(By.id("info-dialog"));
```

The example waits for the element *info-dialog* to disappear. The time to wait is 5000 milliseconds by default. You can set the timeout on the action. Due to Selenium limitations the minimum wait time is 1000 milliseconds.

23.12. Navigate

The action navigates to a new page either by using a new relative path or a complete new Http URL.

XML DSL

```
<selenium:navigate page="http://localhost:8080"/>

<selenium:navigate page="help"/>
```

Java DSL

```
selenium().navigate("http://localhost:8080");

selenium().navigate("help");
```

The sample above describes a new page with new Http URL. The browser will navigate to this new page. All further Selenium actions are performed on this new page. The second navigation action opens the relative page *help* so the new page URL is `http://localhost:8080/help`.

Navigation is always done on the active browser window. You can manage the opened windows as described in next section.

23.13. Window actions

Selenium is able to manage multiple windows. So you can open, close and switch active windows in a Citrus test.

XML DSL

```
<selenium:open-window name="my_window"/>
<selenium:switch-window name="my_window"/>
<selenium:close-window name="my_window"/>
```

Java DSL

```
selenium().open().window("my_window");
selenium().focus().window("my_window");
selenium().close().window("my_window");
```

When a new window is opened Selenium creates a window handle for us. This window handle is saved as test variable using a given window name. So after opening the window you can access the window by its name in further actions. All upcoming Selenium actions will take place in this new active window. Of course the test actions will fail as soon as the window with that given name is missing. Citrus uses default window names that are automatically used as test variables:

selenium_active_window	the active window handle
selenium_last_window	the last window handle when switched to other window

23.14. Alert

We are able to access the alert dialog on the current page. Citrus will validate the displayed dialog text and accept or dismiss of the dialog.

XML DSL

```
<selenium:alert accept="true">
  <selenium:alert-text>Hello!</selenium:alert-text>
</selenium:alert>
```

Java DSL

```
selenium().alert().text("Hello!").accept();
```

The alert dialog text is validated when expected text is given on the test action. The user can decide to accept or dismiss the dialog. After that the dialog should be closed. In case the test action fails to find an open alert dialog the test action raises runtime errors and the test will fail.

23.15. Make screenshot

You can execute this action in case you want to take a screenshot of the current page. This action only works with browsers that actually display the user interface. The action will not have any effect when executed with Html unit web driver in headless mode.

XML DSL

```
<selenium:screenshot/>

<selenium:screenshot output-dir="target"/>
```

Java DSL

```
selenium().screenhsot();

selenium().screenhsot("target");
```

The test action has an optional parameter *output-dir* which represents the output directory where the screenshot is saved to.

[[temporary-storage-(firefox)]] == Temporary storage (Firefox)

Important This action only works with Firefox web driver! Other browsers are not working with the temporary download storage.

The browser uses a temporary storage for downloaded files. We can access this temporary storage during a test case.

XML DSL

```
<selenium:store-file file-path="classpath:download/file.txt"/>
<selenium:get-stored-file file-name="file.txt"/>
```

Java DSL

```
selenium().store("classpath:download/file.txt");
selenium().getStored("file.txt");
```

As you can see the test case is able to store new files to the temporary browser storage. We have to give the file path as classpath or file system path. When reading the temporary file storage we need to specify the file name that we want to access in the temporary storage. The temporary storage is not capable of subdirectories all files are stored directly to the storage in one single directory.

In case the stored file is not found by that name the test action fails with respective errors. On the other hand when the file is found in temporary storage Citrus will automatically create a new test variable **selenium_download_file** which contains the file name as value.

23.16. Clear browser cache

When clearing the browser cache all cookies and temporary files will be deleted.

XML DSL

```
<selenium:clear-cache/>
```

Java DSL

```
selenium().clearCache();
```

Chapter 24. Vert.x event bus support

Vert.x is an application platform for the JVM that provides a network event bus for lightweight scalable messaging solutions. The Citrus Vert.x components do participate on that event bus messaging as producer or consumer. With these components you can access Vert.x instances available in your network in order to test those Vert.x applications in some integration test scenario.

The Vert.x components in Citrus are kept in a separate Maven module. So you should add the module as Maven dependency to your project accordingly.

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-vertx</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

Citrus provides a special Vert.x configuration schema that is used in our Spring configuration files. You have to include the citrus-vertx namespace in your Spring configuration XML files as follows.

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:citrus="http://www.citrusframework.org/schema/config"
      xmlns:citrus-vertx="http://www.citrusframework.org/schema/vertx/config"
      xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.citrusframework.org/schema/config
      http://www.citrusframework.org/schema/config/citrus-config.xsd
      http://www.citrusframework.org/schema/vertx/config
      http://www.citrusframework.org/schema/vertx/config/citrus-vertx-config.xsd">

      [...]

      </beans>
```

Now you are ready to use the Citrus Vert.x configuration elements using the citrus-vertx namespace prefix.

The next sections discuss sending and receiving operations on the Vert.x event bus with Citrus.

24.1. Vert.x endpoint

As usual Citrus uses an endpoint component in order to specify some message destination to send and receive messages to and from. The Vert.x endpoint component is defined as follows in your Citrus Spring configuration.

```
<citrus-vertx:endpoint id="simpleVertxEndpoint"
      host="localhost"
      port="5001"
      pubSubDomain="false"
      address="news-feed"/>

<bean id="vertxInstanceFactory"
class="org.citrusframework.vertx.factory.CachingVertxInstanceFactory"/>
```

The endpoint holds some general information how to access the Vert.x event bus. Host and port values define the Vert.x Hazelcast cluster hostname and port. Citrus starts a new Vert.x instance using this cluster. So all other Vert.x instances connected to this cluster host will receive the event bus messages from Citrus during the test. In your test case you can use this endpoint component referenced by its id or name in order to send and receive messages on the event bus address **news-feed** . In Vert.x the event bus address defines the destination for event consumers to listen on. As already mentioned cluster hostname and port are optional, so Citrus will use **localhost** and a new random port on the cluster host if nothing is specified.

The Vert.x event bus supports publish-subscribe and point-to-point message communication patterns. By default the **pubSubDomain** in Citrus is false so the event bus sender will initiate a point-to-point communication on the event bus address. This means that only one single consumer on the event bus address will receive the message. If there are more consumers on the address the first to come wins and receives the message. In contrary to that the publish-subscribe scenario would deliver the message to all available consumers on the event bus address simultaneously. You can enable the **pubSubDomain** on the Vert.x endpoint component for this communication pattern.

The Vert.x endpoint needs an instance factory implementation in order to create the embedded Vert.x instance. By default the bean name **vertxInstanceFactory** is recognized by all Vert.x endpoint components. We will talk about Vert.x instance factories in more detail later on in this chapter.

As message content you can send and receive JSON objects or simple character sequences to the event bus. Let us have a look at a simple sample sending action that uses the new Vert.x endpoint component:

```
<send endpoint="simpleVertxEndpoint">
  <message type="plaintext">
    <payload>Hello from Citrus!</payload>
  </message>
</send>
```

As the Vert.x Citrus endpoint is bidirectional you can also receive messages from the event bus.

```
<receive endpoint="simpleVertxEndpoint">
  <message type="plaintext">
    <payload>Hello from Vert.x!</payload>
  </message>
  <header>
    <element name="citrus_vertx_address" value="news-feed"/>
  </header>
</receive>
```

Citrus automatically adds some special message headers to the message, so you can validate the Vert.x event bus address. This completes the simple send and receive operations on a Vert.x event bus. Now let's move on to synchronous endpoints where Citrus waits for a reply on the event bus.

24.2. Synchronous Vert.x endpoint

The synchronous Vert.x event bus producer sends a message and waits synchronously for the response to arrive on some reply address destination. The reply address name is generated automatically and set in the request message header attributes so the receiving counterpart in this communication can send its reply to that event bus address. The basic configuration for a synchronous Vert.x endpoint component looks like follows:

```
<citrus-vertx:sync-endpoint id="vertxSyncEndpoint"
      address="hello"
      timeout="1000"
      polling-interval="300"/>
```

Synchronous endpoints poll for synchronous reply messages to arrive on the event bus reply address. The poll interval is an optional setting in order to manage the amount of reply message handshake attempts. Once the endpoint was able to receive the reply message synchronously the test case can receive the reply. In case all message handshake attempts do fail because the reply message is not available in time we raise some timeout error and the test will fail.

> The Vert.x endpoint uses temporary reply address destinations. The temporary reply address in generated and is only used once for a single communication handshake. After that the reply address is dismissed again.

When sending a message to the synchronous Vert.x endpoint the producer will wait synchronously for the response message to arrive on the reply address. You can receive the reply message in your test case using the same endpoint component. So we have two actions on the same endpoint, first send then receive.

```
<send endpoint="vertxSyncEndpoint">
  <message type="plaintext">
    <payload>Hello from Citrus!</payload>
  </message>
</send>

<receive endpoint="vertxSyncEndpoint">
  <message type="plaintext">
    <payload>This is the reply from Vert.x!</payload>
  </message>
</receive>
```

In the last section we saw that synchronous communication is based on reply messages on temporary reply event bus address. We saw that Citrus is able to send messages to event bus address and wait for reply messages to arrive. This next section deals with the same synchronous communication, but send and receive roles are switched. Now Citrus receives a message and has to send a reply message to a temporary reply address.

We handle this synchronous communication with the same synchronous Vert.x endpoint component. Only difference is that we initially start the communication by receiving a message from the endpoint. Knowing this Citrus is able to send a synchronous response back. Again just use the same endpoint reference in your test case. The handling of the temporary reply address is done automatically behind the scenes. So we have again two actions in our test case, but this time first receive then send.

```
<receive endpoint="vertxSyncEndpoint">
  <message type="plaintext">
    <payload>Hello from Vert.x!</payload>
  </message>
</receive>

<send endpoint="vertxSyncEndpoint">
  <message type="plaintext">
    <payload>This is the reply from Citrus!</payload>
  </message>
</send>
```

The synchronous message endpoint for Vert.x event bus communication will handle all reply address destinations and provide those behind the scenes.

24.3. Vert.x instance factory

Citrus starts an embedded Vert.x instance at runtime in order to participate in the Vert.x cluster. Within this cluster multiple Vert.x instances are connected via the event bus. For starting the Vert.x event bus Citrus uses a cluster hostname and port definition. You can customize this cluster host in order to connect to a very special cluster in your network.

Now Citrus needs to manage the Vert.x instances created during the test run. By default Citrus will look for an instance factory bean named **vertxInstanceFactory** . You can choose the factory implementation to use in your project. By default you can use the caching factory implementation that caches the Vert.x instances so we do not connect more than one Vert.x instance to the same cluster host. Citrus offers following instance factory implementations:

org.citrusframework.vertx.factory.CachingVertxInstanceFactory

default implementation that reuses the Vert.x instance based on given cluster host and port. With this implementation we ensure to connect a single Citrus Vert.x instance to a cluster host.

org.citrusframework.vertx.factory.SingleVertxInstanceFactory

creates a single Vert.x instance and reuses this instance for all endpoints. You can also set your very custom Vert.x instance via configuration for custom Vert.x instantiation.

The instance factory implementations do implement the ***VertxInstanceFactory*** interface. So you can also provide your very special implementation. By default Citrus looks for a bean named **vertxInstanceFactory** but you can also define your very special factory implementation onm an endpoint component. The Vert.x instance factory is set on the Vert.x endpoint as follows:

```
<citrus-vertx:endpoint id="vertxHelloEndpoint"
      address="hello"
      vertx-factory="singleVertxInstanceFactory"/>

<bean id="singleVertxInstanceFactory"
      class="org.citrusframework.vertx.factory.SingleVertxInstanceFactory"/>
```

Chapter 25. JDBC support

Database communication is an essential part of many applications, when persistent data storage is required. May it be orders, customer data, product recommendations or product information, if persistent storage is in place, the data contains a certain business value. Therefore it's important that your software handles your persistent storage the right way. To ensure that, Citrus offers a JDBC server endpoint that allows you to verify the communication between your application and a real database server which is accessible via the Citrus-JDBC-Driver.

To enable the JDBC support for your test project, you'll have to add the following dependency.

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-jdbc</artifactId>
  <version>${citrus.version}</version>
  <scope>test</scope>
</dependency>
```

25.1. The Citrus-JDBC-Driver

The Citrus-JDBC-Driver is a JDBC conform driver that realizes the communication with the Citrus-JDBC-Server. To be able to use Citrus-JDBC within a CI/CD setup, we recommend to ensure that your software is able to exchange the JDBC driver by configuration. The Citrus-JDBC-Driver is available on maven central under `org.citrusframework.citrus-db-driver`. After the driver has been downloaded and configured to be used by your application, please make sure that the driver configuration matches the Citrus-JDBC-Server configuration in your tests.

Example jdbc driver configuration

```
<systemProperties>
    <systemProperty>
      <name>todo.persistence.type</name>
      <value>jdbc</value>
    </systemProperty>
    <systemProperty>
      <name>todo.jdbc.driverClassName</name>
      <value>org.citrusframework.db.driver.JdbcDriver</value>
    </systemProperty>
    <systemProperty>
      <name>todo.jdbc.url</name>
      <value>jdbc:citrus:http://localhost:3306/testdb</value>
    </systemProperty>
</systemProperties>
```

Citrus-JDBC-Server matching the driver config

```
@Bean
public JdbcServer jdbcServer() {
    return CitrusEndpoints.jdbc()
        .server()
        .host("localhost")
        .databaseName("testdb")
        .port(3306)
        .build();
}
```

25.2. The Citrus-JDBC-Server

To setup a JDBC endpoint within your tests, just configure it as any other endpoint via e.g. XML, Spring Bean or by Citrus Annotations.

Spring Bean

```
@Bean
public JdbcServer jdbcServer() {
    return CitrusEndpoints.jdbc()
        .server()
        .host("localhost")
        .databaseName("testdb")
        .port(4567)
        .timeout(10000L)
        .autoStart(true)
        .autoTransactionHandling(false)
        .build();
}
```

Spring XML

```
<citrus-jdbc:server id="testServer"
                      host="citrus-jdbc-test-server"
                      port="4567"
                      database-name="test-db"
                      max-connections="50"
                      auto-start="true"/>
```

Citrus Annotations

```
@CitrusEndpoint
@JdbcServerConfig(
            databaseName = "testdb",
            autoStart = true,
            port = 4567)
private JdbcServer jdbcServer;
```

After that configuration has been done, you'll be able to use the Server within your tests to receive and send messages from or to your system under test.

```
@Test
@CitrusTest
public void testAddTodoEntry() {
    variable("todoName", "citrus:concat('todo_', citrus:randomNumber(4))");
    variable("todoDescription", "Description: ${todoName}");

    http()
        .client(todoClient)
        .send()
        .post("/todolist")
        .fork(true)
        .contentType("application/x-www-form-urlencoded")
        .payload("title=${todoName}&description=${todoDescription}");

    receive(jdbcServer)
        .messageType(MessageType.JSON)
        .message(JdbcMessage.execute(
            "@startsWith('INSERT INTO todo_entries (id, title, description, done)
VALUES (?, ?, ?, ?)')@"));①

    send(jdbcServer)
        .message(JdbcMessage.result().rowsUpdated(1));②

    http()
        .client(todoClient)
        .receive()
        .response(HttpStatus.FOUND);

    http()
        .client(todoClient)
        .send()
        .get("/todolist")
        .fork(true)
        .accept("text/html");

    receive(jdbcServer)
            .message(JdbcMessage.execute("SELECT id, title, description FROM
todo_entries"));③
```

```
    send(jdbcServer)
        .messageType(MessageType.JSON)
        .message(JdbcMessage.result().dataSet("[ {" +
                    "\"id\": \"" + UUID.randomUUID().toString() + "\"," +
                    "\"title\": \"${todoName}\"," +
                    "\"description\": \"${todoDescription}\"," +
                    "\"done\": \"false\"" +
                "} ]"));④

    http()
        .client(todoClient)
        .receive()
        .response(HttpStatus.OK)
        .messageType(MessageType.XHTML)
        .xpath("(//xh:li[@class='list-group-item']/xh:span)[last()]", "${todoName}");
}
```

① Expects a `INSERT` statement matching the given expression.

② Responds with a result set stating, that one row has been updated/created.

③ Expects a `SELECT` statement matching the given statement.

④ Responds with the DataSet specified as JSON string.

25.2.1. Transactions

When it comes to complex modifications of your database, transactions are commonly used. Citrus is able to verify the behavior of your system under test concerning start, commit and rollback actions of transactions. The verification of transactions has to be enabled in the server Citrus-JDBC-Server configuration. For more information, please have a look at the Configuration section.

Verifiying transaction commit

```
receive(jdbcServer)
    .message(JdbcMessage.startTransaction());①

receive(jdbcServer)
    .message(JdbcMessage.execute("@startsWith('INSERT INTO todo_entries (id, title,
description, done) VALUES (?, ?, ?, ?)')@"));

send(jdbcServer)
    .message(JdbcMessage.result().rowsUpdated(1));

receive(jdbcServer)
    .message(JdbcMessage.commitTransaction());②
```

① Verify, that the transaction has been started.

② Verify, that the modification of the database has been committed.

It is also possible to simulate an erroneous modification including the verification of a rollback.

Verifiying transaction rollback

```
receive(jdbcServer)
    .message(JdbcMessage.startTransaction());①

receive(jdbcServer)
    .message(JdbcMessage.execute("@startsWith('INSERT INTO todo_entries (id, title,
description, done) VALUES (?, ?, ?, ?)')@"));

send(jdbcServer)
    .message(JdbcMessage.result().exception("Could not execute something"));

receive(jdbcServer)
    .message(JdbcMessage.rollbackTransaction());②
```

① Verify, that the transaction has been started.

② Verify, that a rollback occurred after the database exception has been send.

25.2.2. Prepared statements

Because prepared statements work slightly different than simple database queries, the validation of those is also slightly different. Currently, Citrus offers you the possibility to verify that your application has created the correct prepared statement, that it was executed with the correct parameters and that it has been closed.

Verifying prepared statement

```
receive(jdbcServer)
    .message(JdbcMessage.createPreparedStatement("INSERT INTO todo_entries (id, title,
description, done) VALUES (?, ?, ?, ?)"));①

receive(jdbcServer)
    .message(JdbcMessage.execute(
        "INSERT INTO todo_entries (id, title, description, done) VALUES (?, ?, ?, ?) -
(1,sample,A sample todo,false)"));②

receive(jdbcServer)
    .message(JdbcMessage.closeStatement());③
```

① Verify that the given prepared statement has been created.

② Verify that the statement has been executed with the parameters `1,sample,A sample todo,false`.

③ Verify that the statement has been closed.

Please notice, that the verification of `createPreparedStatement` and `closeStatement` messages has to be activated via configuration. For more information, please have a look at the Configuration section.

25.2.3. Callable statements / stored procedures

As well as prepared statements, callable statements are different from simple queries. Callable statements are used on jdbc level to access stored procedures, functions, etc. on the database server.

Verifying callable statement

```
receive(jdbcServer)
    .message(JdbcMessage.createCallableStatement("{CALL limitedToDoList(?)}"));①

receive(jdbcServer)
    .message(JdbcMessage.execute("{CALL limitedToDoList(?)} - (1)"));②

send(jdbcServer)
    .messageType(MessageType.XML)
    .message(JdbcMessage.result().dataSet("" +
            "<dataset>" +
                "<row>" +
                    "<id>1</id>"+
                    "<title>sample</title>"+
                    "<description>A sample todo</description>" +
                    "<done>false</done>" +
                 "</row>" +
                "</dataset>"));

receive(jdbcServer)
    .message(JdbcMessage.closeStatement());③
```

① Verify that the given callable statement has been created.

② Verify that the statement has been executed with the parameter 1.

③ Verify that the statement has been closed.

As you might have noticed, callable statements and prepared statements have nearly the same workflow in Citrus. The only difference is the creation of the statement itself. It is also the case that the verification of `createCallableStatement` and `closeStatement` messages has to be activated via configuration. For more information, please have a look at the Configuration section.

25.2.4. Configuration

As already mentioned, you're able to configure the JDBC endpoint in different ways (XML, Spring Bean, etc.). The following properties are available to configure the server for your test scenario.

Property	Mandatory	Default	Description
id	Yes		Only required for XML configuration.

Property	Mandatory	Default	Description
auto connect	No	true	Determines whether the server should automatically accept connection related messages or validate them. This includes `openConnection` and `closeConnection`.
auto create statement	No	true	Determines whether the server should automatically accept statement related messages or validate them. This includes `createStatement`, `createPreparedStatement`, `createCallableStatement` and `closeStatement`.
auto transaction handling	No	true	Determines whether the server should automatically accept transaction related messages or validate them. This includes `startTransaction`, `commitTransaction` and `rollbackTransaction`.
auto handle queries	No	Collection of system queries for different databases	Determines whether the server should automatically respond with a positive answer for matching queries, e.g. `SELECT USER FROM DUAL`. You can override the currently defined validation queries when setting `citrus.jdbc.auto.handle.query` system property within the citrus-application.properties. The property value is expected to be a semicolon separated list of regex patterns. Every query can be specified as a regular expression, e.g. `SELECT.*FROM DUAL;SELECT \\w;`.
host	Yes		The hostname of the server. There has to be a valid route between the test suite, the system under test and the database server.
port	No	4567	The port the server listens to.
database name	Yes		The database name to work on
max connections	No	20	The maximum amount of open connections to be accepted by the server.
polling interval	No	500	Polling interval when waiting for synchronous reply message to arrive.

Property	Mandatory	Default	Description
timeout	No	5000	Send/receive timeout setting
debug logging	No	false	Determines whether the inbound channel debug logging should be enabled.

In addition, there are advanced configuration possibilities to customize the behavior of the JDBC server.

Property	Mandatory	Default	Description
correlator	No	DefaultMessageCorrelator	A MessageCorrelator implementation to identify messages.
endpoint adapter	No	JdbcEndpointAdapterController	An Endpoint adapter implementation creating the messages for validation.

25.3. JdbcMessage

The JdbcMessage class is the central location to specifying your expected inbound and outbound communication for the JDBC endpoint.

Message	receive/send	Description
`JdbcMessage.openConnection(Properties properties)`	receive	States that a connection has been opened with the given properties. The evaluation of connections has to be enabled via the endpoint configuration.
`JdbcMessage.closeConnection()`	receive	States that the connection has been closed. The evaluation of connections has to be enabled via the endpoint configuration.
`JdbcMessage.createStatement()`	receive	States that a statement has been created. The evaluation of statement handling has to be enabled via the endpoint configuration.
`JdbcMessage.createPreparedStatement(String sql)`	receive	States that a SQL statement matching the given expression has been created. The evaluation of statement handling has to be enabled via the endpoint configuration.

Message	receive/send	Description
`JdbcMessage.createCallableStatement(String sql)`	receive	States that a callable statement referencing a function or procedure that is matching the given expression has been created. The evaluation of statement handling has to be enabled via the endpoint configuration.
`JdbcMessage.closeStatement()`	receive	States that a statement has been closed. The evaluation of statement handling has to be enabled via the endpoint configuration.
`JdbcMessage.execute(String sql)`	receive	States that a SQL statement matching the given expression has been executed.
`JdbcMessage.startTransaction()`	receive	States that a transaction start has been received. The evaluation of transaction handling has to be enabled via the endpoint configuration.
`JdbcMessage.commitTransaction()`	receive	States that a commit for a transaction has been received. The evaluation of transaction handling has to be enabled via the endpoint configuration.
`JdbcMessage.rollbackTransaction()`	receive	States that a rollback of the transaction has been received. The evaluation of transaction handling has to be enabled via the endpoint configuration.
`JdbcMessage.result()`	send	Sends a positive empty result to the system under test.
`JdbcMessage.result(boolean success)`	send	Sends empty result to the system under test which is a success or a failure based on the given boolean value.
`JdbcMessage.exception(String exceptionText)`	send	Sends an exception to the system under test. Regarding the driver documentation, the error will be an SQLException.

Message	receive/send	Description
`JdbcMessage.rowsUpdated(int number)`	send	Sends a positive result to the system under test where the payload is the number of updated rows.
`JdbcMessage.dataSet(DataSet dataSet)`	send	Sends a positive result to the system under test where the payload is the specified DataSet.
`JdbcMessage.dataSet(String dataSet)`	send	Sends a positive result to the system under test where the payload is the specified DataSet. To use this, you have to specify the format of the dataSet String. Please refer to the section DataSet parsing.
`JdbcMessage.dataSet(Resource dataSet)`	send	Sends a positive result to the system under test where the payload is the content of the specified resource. To use this, you have to specify the format of the dataSet String. Please refer to the section DataSet parsing.
`JdbcMessage.success()`	send	Sends a positive empty result to the system under test.
`JdbcMessage.error()`	send	Sends an empty error result to the system under test.

25.3.1. DataSet parsing

Citrus provides different ways to prepare the response DataSets for your system under test. You can specify your DataSets as Java Objects, as XML or JSON Strings or as resource file containing your XML or JSON DataSet as text.

Java dataset creation example

```
Row sheldon = new Row();
sheldon.getValues().put("id", "1");
sheldon.getValues().put("name", "Sheldon");
sheldon.getValues().put("profession", "physicist");

Row leonard = new Row();
leonard.getValues().put("id", "2");
leonard.getValues().put("name", "Leonard");
leonard.getValues().put("profession", "physicist");
leonard.getValues().put("email", "leo@bigbangtheory.org");

Row penny = new Row();
penny.getValues().put("id", "3");
penny.getValues().put("name", "Penny");
penny.getValues().put("profession", "this_and_that");

Table table = new Table("user");
table.getRows().add(sheldon);
table.getRows().add(leonard);
table.getRows().add(penny);

DataSet userDataSet = new TableDataSetProducer(table).produce();

send(jdbcServer).message(JdbcMessage.result().dataSet(userDataSet));
```

If you use the XML or JSON notation as string or within a resource, you'll have to specify that in your test setup.

Java json dataset creation example

```
receive(jdbcServer)
                .message(JdbcMessage.execute("SELECT id, title, description FROM
todo_entries"));

send(jdbcServer)
        .messageType(MessageType.JSON)①
        .message(JdbcMessage.result().dataSet("[ {" +
                    "\"id\": \"" + UUID.randomUUID().toString() + "\"," +
                    "\"title\": \"${todoName}\"," +
                    "\"description\": \"${todoDescription}\"," +
                    "\"done\": \"false\"" +
                "} ]"));
```

① Tells Citrus that the response has to be interpreted as JSON.

Java xml dataset creation example

```
receive(jdbcServer)
        .message(JdbcMessage.execute("SELECT id, title, description FROM
todo_entries"));
send(jdbcServer)
        .messageType(MessageType.XML)①
        .message(JdbcMessage.result().dataSet("" +
                "<dataset>" +
                    "<row>" +
                        "<id>${todoId}</id>"+
                        "<title>${todoName}</title>"+
                        "<description>${todoDescription}</description>" +
                        "<done>false</done>" +
                     "</row>" +
                "</dataset>"));
```

① Tells Citrus that the response has to be interpreted as XML.

Technically it is not required to specify the messages as `MessageType.XML`, because the default message type in citrus currently **is** XML. Nevertheless we highly recommend to specify the message type. This will ensure that your tests sustain future changes.

Chapter 26. Docker support

Citrus provides configuration components and test actions for interaction with a Docker daemon. The Citrus docker client component will execute Docker commands for container management such as start, stop, build, inspect and so on. The Docker client by default uses the Docker remote REST API. As a user you can execute Docker commands as part of a Citrus test and validate possible command results.

> The Docker test components in Citrus are kept in a separate Maven module. If not already done, you have to include the module as Maven dependency to your project

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-docker</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

Citrus provides a "citrus-docker" configuration namespace and schema definition for Docker related components and actions. Include this namespace into your Spring configuration in order to use the Citrus Docker configuration elements. The namespace URI and schema location are added to the Spring configuration XML file as follows.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:citrus-docker="http://www.citrusframework.org/schema/docker/config"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.citrusframework.org/schema/docker/config
       http://www.citrusframework.org/schema/docker/config/citrus-docker-config.xsd">

    [...]

</beans>
```

After that you are able to use customized Citrus XML elements in order to define the Spring beans.

26.1. Docker client

Citrus operates with the Docker remote REST API in order to interact with the Docker daemon. The Docker client is defined as Spring bean component in the configuration as follows:

```
<citrus-docker:client id="dockerClient"/>
```

The Docker client component above is using all default configuration values. By default Citrus is searching the system properties as well as environment variables for default Docker settings such as:

DOCKER_HOST	tcp://localhost:2376
DOCKER_CERT_PATH	~/.docker/machine/machines/default
DOCKER_TLS_VERIFY	1
DOCKER_MACHINE_NAME	default

In case these settings are not settable in your environment you can also use explicit settings in the Docker client component:

```
<citrus-docker:client id="dockerClient"
            url="tcp://localhost:2376"
            version="1.20"
            username="user"
            password="s!cr!t"
            email="user@foo.bar"
            registry="https://index.docker.io/v1/"
            cert-path="/path/to/some/cert/directory"
            config-path="/path/to/some/config/directory"/>
```

Now Citrus is able to access the Docker remote API for executing commands such as start, stop, build, inspect and so on.

26.2. Docker commands

We have several Citrus test actions each representing a Docker command. These actions can be part of a test case where you can manage Docker containers inside the test. As a prerequisite we have to enable the Docker specific test actions in our XML test as follows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:docker="http://www.citrusframework.org/schema/docker/testcase"
        xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.citrusframework.org/schema/docker/testcase
        http://www.citrusframework.org/schema/docker/testcase/citrus-docker-
testcase.xsd">

    [...]

</beans>
```

We added a special docker namespace with prefix **docker:** so now we can start to add Docker test actions to the test case:

XML DSL

```
<testcase name="DockerCommandIT">
    <actions>
          <docker:ping></docker:ping>

          <docker:version>
            <docker:expect>
              <docker:result>
                <![CDATA[
                  {
                    "Version":"1.8.3",
                    "ApiVersion":"1.21",
                    "GitCommit":"@ignore@",
                    "GoVersion":"go1.4.2",
                    "Os":"darwin",
                    "Arch":"amd64",
                    "KernelVersion":"@ignore@"
                  }
                ]]>
              </docker:result>
            </docker:expect>
          </docker:version>
    </actions>
</testcase>
```

In this very simple example we first ping the Docker daemon to make sure we have connectivity up and running. After that we get the Docker version information. The second action shows an important concept when executing Docker commands in Citrus. As a tester we might be interested in validating the command result. So we can specify an optional **docker:result** which is usually in JSON data format. As usual we can use test variables here and ignore some values explicitly such as the **GitCommit** value.

Based on that we can execute several Docker commands in a test case:

XML DSL

```
<testcase name="DockerCommandIT">
    <variables>
      <variable name="imageId" value="busybox"></variable>
      <variable name="containerName" value="citrus_box"></variable>
    </variables>

    <actions>
        <docker:pull image="${imageId}"
                        tag="latest"/>

        <docker:create image="${imageId}"
                          name="${containerName}"
                          cmd="top">
            <docker:expect>
                <docker:result>
                    <![CDATA[
                      {"Id":"@variable(containerId)@","Warnings":null}
                    ]]>
                </docker:result>
            </docker:expect>
        </docker:create>

        <docker:start container="${containerName}"/>
    </actions>
</testcase>
```

In this example we pull a Docker image, build a new container out of this image and start the container. As you can see each Docker command action offers attributes such as **container**, **image** or **tag** . These are command settings that are available on the Docker command specification. Read more about the Docker commands and the specific settings in official Docker API reference guide.

Citrus supports the following Docker commands with respective test actions:

- **docker:pull**
- **docker:build**
- **docker:create**
- **docker:start**
- **docker:stop**
- **docker:wait**
- **docker:ping**
- **docker:version**
- **docker:inspect**

- **docker:remove**
- **docker:info**

Some of the Docker commands can be executed both on container and image targets such as **docker:inspect** or **docker:remove** . The command action then offers both **container** and **image** attributes so the user can choose the target of the command operation to be a container or an image.

Up to now we have only used the Citrus XML DSL. Of course all Docker commands are also available in Java DSL as the next example shows.

Java DSL

```
@CitrusTest
public void dockerTest() {
    docker().version()
        .validateCommandResult(new CommandResultCallback<Version>() {
            @Override
            public void doWithCommandResult(Version version, TestContext context) {
                Assert.assertEquals(version.getApiVersion(), "1.20");
            }
    });

    docker().ping();

    docker().start("my_container");
}
```

The Java DSL Docker commands provide an optional **CommandResultCallback** that is called with the unmarshalled command result object. In the example above the *Version* model object is passed as argument to the callback. So the tester can access the command result and validate its properties with assertions.

By default Citrus tries to find a Docker client component within the Citrus Spring application context. If not present Citrus will instantiate a default docker client with all default settings. You can also explicitly set the docker client instance when using the Java DSL Docker command actions:

Java DSL

```
@Autowired
private DockerClient dockerClient;

@CitrusTest
public void dockerTest() {
    docker().client(dockerClient).version()
        .validateCommandResult(new CommandResultCallback<Version>() {
            @Override
            public void doWithCommandResult(Version version, TestContext context) {
                Assert.assertEquals(version.getApiVersion(), "1.20");
            }
    });

    docker().client(dockerClient).ping();

    docker().client(dockerClient).start("my_container");
}
```

Chapter 27. SSH support

In the spirit of other Citrus mock services, there is support for simulating an external SSH server as well as for connecting to SSH servers as a client during the test execution. Citrus translates SSH requests and responses to simple XML documents for better validation with the common Citrus mechanisms.

This means that the Citrus test case does not deal with pure SSH protocol commands. Instead of this we use the powerful XML validation capabilities in Citrus when dealing with the simple XML documents that represent the SSH request/response data.

Let us clarify this with a little example. Once the real SSH server daemon is fired up within Citrus we accept an SSH EXEC request for instance. The request is translated into an XML message of the following format:

```
<ssh-request xmlns="http://www.citrusframework.org/schema/ssh/message">
  <command>cat - | sed -e 's/Hello/Hello SSH/'</command>
  <stdin>Hello World</stdin>
</ssh-request>
```

This message can be validated with the usual Citrus mechanism in a receive test action. If you do not know how to do this, please read one of the sections about XML message validation in this reference guide first. Now after having received this request message the respective SSH response should be provided as appropriate answer. This is done with a message sending action on a reply handler as it is known from synchronous http message communication in Citrus for instance. The SSH XML representation of a response message looks like this:

```
<ssh-response xmlns="http://www.citrusframework.org/schema/ssh/message">
  <stdout>Hello SSH World</stdout>
  <stderr></stderr>
  <exit>0</exit>
</ssh-response>
```

Besides simulating a full-featured SSH server, Citrus also provides SSH client functionality. This client uses the same request message pattern, which is translated into a real SSH call to an SSH server. The SSH response received is also translated into an XML message as shown above so we can validate it with known validation mechanisms in Citrus.

Similar to the other Citrus modules (http, soap), a Citrus SSH server and client is configured in Citrus Spring application context. There is a dedicated **ssh** namespace available for all ssh Citrus components. The namespace declaration goes into the context top-level element as usual:

```
<beans
    [...]
    xmlns:citrus-ssh="http://www.citrusframework.org/schema/ssh/config"
    [...]
    xsi:schemaLocation="
        [...]
        http://www.citrusframework.org/schema/ssh/config
        http://www.citrusframework.org/schema/ssh/config/citrus-ssh-config.xsd
        [...] ">
   [...]
</beans>
```

Both, SSH server and client along with their configuration options are described in the following two sections.

27.1. SSH Client

A Citrus SSH client is useful for testing against a real SSH server. So Citrus is able to invoke SSH commands on the external server and validate the SSH response accordingly. The test case does not deal with the pure SSH protocol within this communication. The Citrus SSH client component expects a customized XML representation and automatically translates these request messages into a real SSH call to a specific host. Once the synchronous SSH response was received the result gets translated back to the XML response message representation. On this translated response we can easily apply the validation steps by the usual Citrus means.

The SSH client components receive its configuration in the Spring application context as usual. We can use the special SSH module namespace for easy configuration:

```
<citrus-ssh:client id="sshClient"
           port="9072"
           user="roland"
           private-key-path="classpath:org/citrusframework/ssh/test_user.priv"
           strict-host-checking="false"
           host="localhost"/>
```

The SSH client receives several attributes, these are:

id	Id identifying the bean and used as reference from with test descriptions. (e.g. id="sshClient")
host	Host to connect to for sending an SSH Exec request. Default is 'localhost' (e.g. host="localhost") port: Port to use. Default is 2222 (e.g. port="9072")

private-key-path	Path to a private key, which can be either a plain file path or a class resource if prefixed with 'classpath' (e.g. private-key-path="classpath:test_user.priv")
private-key-password	Optional password for the private key (e.g. password="s!cr!t")
user	User used for connecting to the SSH server (e.g. user="roland")
password	Password used for password based authentication. Might be combined with "private-key-path" in which case both authentication mechanism are tried (e.g. password="ps!st)
strict-host-checking	Whether the host key should be verified by looking it up in a 'known_hosts' file. Default is false (e.g. strict-host-checking="true")
known-hosts-path	Path to a known hosts file. If prefixed with 'classpath:' this file is looked up as a resource in the classpath (e.g. known-hosts-path="/etc/ssh/known_hosts")
command-timeout	Timeout in milliseconds for how long to wait for the SSH command to complete. Default is 5 minutes (e.g. command-timeout="300000")
connection-timeout	Timeout in milliseconds for how long to for a connectiuon to connect. Default is 1 minute (e.g. connection-timeout="60000")
actor	Actor used for switching groups of actions (e.g. actor="ssh-mock")

Once defines as client component in the Spring application context test cases can reference the client in every send test action.

```
<send endpoint="sshClient">
  <message>
    <payload>
        <ssh-request xmlns="http://www.citrusframework.org/schema/ssh/message">
          <command>shutdown</command>
          <stdin>input</stdin>
        </ssh-request>
    </payload>
  </message>
</send>

<receive endpoint="sshClient">
  <message>
    <payload>
        <ssh-response xmlns="http://www.citrusframework.org/schema/ssh/message">
            <stdout>Hello Citrus</stdout>
            <stderr/>
            <exit>0</exit>
        </ssh-response>
    </payload>
  </message>
</receive>
```

As you can see we use usual send and receive test actions. The XML SSH representation helps us to specify the request and response data for validation. This way you can call SSH commands against an external SSH server and validate the response data.

27.2. SSH Server

Now that we have used Citrus on the client side we can also use Citrus SSH server module in order to provide a full stacked SSH server daemon. We can accept SSH client connections and provide proper response messages as an answer.

Given the above SSH module namespace declaration, adding a new SSH server is quite simple:

```
<citrus-ssh:server id="sshServer"
             allowed-key-path="classpath:org/citrusframework/ssh/test_user_pub.pem"
             user="roland"
             port="9072"
             auto-start="true"
             endpoint-adapter="sshEndpointAdapter"/>
```

The **endpoint-adapter** is the handler which receives the SSH request as messages (in the request format described above). Endpoint adapter implementations are fully described in http-serverAll adapters described there are supported in SSH server module, too.

The `<citrus-ssh:server>` supports the following attributes:

SSH Server Attributes:

id	Name of the SSH server which identifies it unique within the Citrus Spring context (e.g. id="sshServer")
host-key-path	Path to PEM encoded key pair (public and private key) which is used as host key. By default, a standard, pre-generate, fixed keypair is used. The path can be specified either as a file path, or, if prefixed with **classpath:** is looked up from within the classpath. The path the is relative from to the top-level package, so no leading slash should be used (e.g. hist-key-path="/etc/citrus_ssh_server.pem)
user-home-path	Path to user home directory. If not set ${user.dir}/target/{serverName}/home/{user} is used by default.
user	User which is allowed to connect (e.g. user="roland")
allowed-key-path	Path to an SSH public key stored in PEM format. These are the keys, which are allowed to connect to the SSH server when publickey authentication is used. It seves the same purpose as authorized_keys for standard SSH installations. The path can be specified either as a file path, or, if prefixed with **classpath:** is looked up from within the classpath. The path the is relative from to the top-level package, so no leading slash should be used (e.g. allowed-key-path="classpath:test_user_pub.pem)
password	Password which should be used when password authentication is used. Both publickey authentication and password based authentication can be used together in which case both methods are tried in turn (e.g. password="s!cr!t")
host	Host address (e.g. localhost)
port	Port on which to listen. The SSH server will bind on localhost to this port (e.g. port="9072")
auto-start	Whether to start this SSH server automatically. Default is **true** . If set to **false**, a test action is responsible for starting/stopping the server (e.g. auto-start="true")
endpoint-adapter	Bean reference to an endpoint adapter which processes the incoming SSH request. The message format for the request and response are described above (e.g. endpoint-adapter="sshEndpointAdapter")

Once the SSH server component is added to the Spring application context with a proper endpoint adapter like the MessageChannel forwarding adapter we can receive incoming requests in a test case and provide a respone message for the client.

```
<receive endpoint="sshServer">
  <message>
    <payload>
        <ssh-request xmlns="http://www.citrusframework.org/schema/ssh/message">
           <command>shutdown</command>
           <stdin>input</stdin>
        </ssh-request>
    </payload>
  </message>
</receive>

<send endpoint="sshServer">
  <message>
    <payload>
        <ssh-response xmlns="http://www.citrusframework.org/schema/ssh/message">
            <stdout>Hello Citrus</stdout>
            <exit>0</exit>
        </ssh-response>
    </payload>
  </message>
</send>
```

Chapter 28. RMI support

RMI stands for Remote Method Invocation and is a standard way of calling Java method interfaces where caller and callee (client and server) are not located within the same JVM. So the object passed to the method as argument as well as the method return value are transmitted over the wire.

As a client Citrus is able to connect to some RMI registry that exposes some remote interfaces. As a server Citrus implements such an RMI registry and handles incoming method calls with providing the respective return value.

> The RMI components in Citrus are kept in a separate Maven module. So you should check that the module is available as Maven dependency in your project

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-rmi</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

As usual Citrus provides a customized rmi configuration schema that is used in Spring configuration files. Simply include the citrus-rmi namespace in the configuration XML files as follows.

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:citrus="http://www.citrusframework.org/schema/config"
      xmlns:citrus-rmi="http://www.citrusframework.org/schema/rmi/config"
      xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.citrusframework.org/schema/config
      http://www.citrusframework.org/schema/config/citrus-config.xsd
      http://www.citrusframework.org/schema/rmi/config
      http://www.citrusframework.org/schema/rmi/config/citrus-rmi-config.xsd">

      [...]

      </beans>
```

Now you are ready to use the customized Http configuration elements with the citrus-rmi namespace prefix.

Read the next section in order to find out more about the RMI message support in Citrus.

28.1. RMI client

On the client side we want to call e remote interface. We need to specify the method to call as well as all method arguments. The respective method return value is receivable within the test case for validation. Citrus provides a client component for RMI that sends out service invocation calls.

```
<citrus-rmi:client id="rmiClient1"
      host="localhost"
      port="1099"
      binding="newsService"/>

      <citrus-rmi:client id="rmiClient2"
        server-url="rmi://localhost:1099/newsService"/>
```

The client component in the Spring application context receives host and port configuration of a valid RMI service registry. Either by specifying a proper server url or by giving host, port and binding properties. The service binding is the name of the service that we would like to address in the registry. Now we are ready to use this client referenced by its id or name in a test case for a message sending action.

Java

```
@CitrusTest
public void rmiMessageTest() {
    $(send(rmiClient)
        .message(RmiMessage.invocation(NewsService.class, "getNews"))
    );
}
```

XML

```
<test name="RmiMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <send endpoint="rmiClient">
            <message>
                <body>
                    <payload>
                        <service-invocation
xmlns="http://www.citrusframework.org/schema/rmi/message">
                          <remote>org.citrusframework.rmi.remote.NewsService</remote>
                          <method>getNews</method>
                        </service-invocation>
                    </payload>
                </body>
            </message>
        </send>
    </actions>
</test>
```

YAML

```
name: RmiMessageTest
actions:
  - send:
      endpoint: "rmiClient"
      message:
        body: |
          <service-invocation
xmlns="http://www.citrusframework.org/schema/rmi/message">
              <remote>org.citrusframework.rmi.remote.NewsService</remote>
              <method>getNews</method>
          </service-invocation>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="RmiMessageTest">
        <actions>
            <send endpoint="rmiClient">
                <message>
                    <payload>
                        <service-invocation
xmlns="http://www.citrusframework.org/schema/rmi/message">
                          <remote>org.citrusframework.rmi.remote.NewsService</remote>
                          <method>getNews</method>
                        </service-invocation>
                    </payload>
                </message>
            </send>
        </actions>
    </testcase>
</spring:beans>
```

We are using the usual Citrus send message action referencing the **rmiClient** as endpoint. The message payload is a special Citrus message that defines the service invocation. We define the **remote** interface as well as the **method** to call. Citrus RMI client component will be able to interpret this message content and call the service method.

The method return value is receivable for validation using the very same client endpoint.

Java

```
@CitrusTest
public void rmiMessageTest() {
    $(receive(rmiClient)
        .message(RmiMessage.result("This is news from RMI!"))
    );
}
```

XML

```
<test name="RmiMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <receive endpoint="rmiClient">
            <message>
                <body>
                    <payload>
                        <service-result
xmlns="http://www.citrusframework.org/schema/rmi/message">
                          <object type="java.lang.String" value="This is news from
RMI!"/>
                        </service-result>
                    </payload>
                </body>
            </message>
        </receive>
    </actions>
</test>
```

YAML

```
name: RmiMessageTest
actions:
  - receive:
      endpoint: "rmiClient"
      message:
        body: |
          <service-result xmlns="http://www.citrusframework.org/schema/rmi/message">
              <object type="java.lang.String" value="This is news from RMI!"/>
          </service-result>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="RmiMessageTest">
        <actions>
            <receive endpoint="rmiClient">
                <message>
                    <payload>
                        <service-result
xmlns="http://www.citrusframework.org/schema/rmi/message">
                          <object type="java.lang.String" value="This is news from
RMI!"/>
                        </service-result>
                    </payload>
                </message>
            </receive>
        </actions>
    </testcase>
</spring:beans>
```

In the sample above we receive the service result and expect a **java.lang.String** object return value. The return value content is also validated within the service result payload.

Of course we can also deal with method arguments.

Java

```
@CitrusTest
public void rmiMessageTest() {
    $(send(rmiClient)
        .message(RmiMessage.invocation(NewsService.class, "setNews")
              .argument("This is breaking news!"))
    );
}
```

XML

```
<test name="RmiMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <send endpoint="rmiClient">
            <message>
                <body>
                    <payload>
                        <service-invocation
xmlns="http://www.citrusframework.org/schema/rmi/message">

<remote>org.citrusframework.rmi.remote.NewsService</remote>
                            <method>setNews</method>
                            <args>
                              <arg value="This is breaking news!"/>
                            </args>
                        </service-invocation>
                    </payload>
                </body>
            </message>
        </send>
    </actions>
</test>
```

YAML

```
name: RmiMessageTest
actions:
  - send:
      endpoint: "rmiClient"
      message:
        body: |
          <service-invocation
xmlns="http://www.citrusframework.org/schema/rmi/message">
                <remote>org.citrusframework.rmi.remote.NewsService</remote>
                <method>setNews</method>
                <args>
                  <arg value="This is breaking news!"/>
                </args>
          </service-invocation>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="RmiMessageTest">
        <actions>
            <send endpoint="rmiClient">
                <message>
                    <payload>
                        <service-invocation
xmlns="http://www.citrusframework.org/schema/rmi/message">

<remote>org.citrusframework.rmi.remote.NewsService</remote>
                            <method>setNews</method>
                            <args>
                              <arg value="This is breaking news!"/>
                            </args>
                        </service-invocation>
                    </payload>
                </message>
            </send>
        </actions>
    </testcase>
</spring:beans>
```

This completes the basic remote service call. Citrus invokes the remote interface method and validates the method return value. As a tester you might also face errors and exceptions when calling the remote interface method. You can catch and assert these remote exceptions verifying your error scenario.

Java

```
@CitrusTest
public void rmiMessageTest() {
    $(assertException()
        .exception(java.rmi.RemoteException.class)
        .when(
            send(rmiClient)
                .message(RmiMessage.invocation(NewsService.class, "setNews")
                        .argument("This is breaking news!"))
        )
    );
}
```

XML

```
<test name="RmiMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <assert exception="java.rmi.RemoteException">
            <when>
                <send endpoint="rmiClient">
                    <message>
                        <body>
                            <payload>
                                <service-invocation
xmlns="http://www.citrusframework.org/schema/rmi/message">

<remote>org.citrusframework.rmi.remote.NewsService</remote>
                                    <method>setNews</method>
                                    <args>
                                      <arg value="This is breaking news!"/>
                                    </args>
                                </service-invocation>
                            </payload>
                        </body>
                    </message>
                </send>
            </when>
        </assert>
    </actions>
</test>
```

YAML

```
name: RmiMessageTest
actions:
  - assert:
      exception: "java.rmi.RemoteException"
      when:
        - send:
            endpoint: "rmiClient"
            message:
              body: |
                <service-invocation
xmlns="http://www.citrusframework.org/schema/rmi/message">
                      <!-- ... -->
                </service-invocation>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="RmiMessageTest">
        <actions>
            <assert exception="java.rmi.RemoteException">
                <when>
                    <send endpoint="rmiClient">
                        <message>
                            <payload>
                                <service-invocation
xmlns="http://www.citrusframework.org/schema/rmi/message">
                                    <!-- ... -->
                                </service-invocation>
                            </payload>
                        </message>
                    </send>
                </when>
            </assert>
        </actions>
    </testcase>
</spring:beans>
```

We assert the ***RemoteException*** to be thrown while calling the remote service method. This is how you can handle some sort of error situation while calling remote services. In the next section we will handle RMI communication where Citrus provides the remote interfaces.

28.2. RMI server

On the server side Citrus needs to provide remote interfaces with methods callable for clients. This means that Citrus needs to support all your remote interfaces with method arguments and return values. The Citrus RMI server is able to bind your remote interfaces to a service registry. All incoming RMI client method calls are automatically accepted and the method arguments are converted into a Citrus XML service invocation representation. The RMI method call is then passed to the running test for validation.

Let us have a look at the Citrus RMI server component and how you can add it to the Spring application context.

```
<citrus-rmi:server id="rmiServer"
      host="localhost"
      port="1099"
      interface="org.citrusframework.rmi.remote.NewsService"
      binding="newService"
      create-registry="true"
      auto-start="true"/>
```

The RMI server component uses properties such as **host** and **port** to define the service registry. By default Citrus will connect to this service registry and bind its remote interfaces to it. With the attribute **create-registry** Citrus can also create the registry for you.

You have to give Citrus the fully qualified remote interface name so Citrus can bind it to the service registry and handle incoming method calls properly. In your test case you can then receive the incoming method calls on the server in order to perform validation steps.

Java

```
@CitrusTest
public void rmiMessageTest() {
    $(receive(rmiServer)
        .message(RmiMessage.invocation(NewsService.class, "getNews"))
    );
}
```

XML

```
<test name="RmiMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <receive endpoint="rmiServer">
            <message>
                <body>
                    <payload>
                        <service-invocation
xmlns="http://www.citrusframework.org/schema/rmi/message">
                          <remote>org.citrusframework.rmi.remote.NewsService</remote>
                          <method>getNews</method>
                        </service-invocation>
                    </payload>
                </body>
                <headers>
                    <header name="citrus_rmi_interface"
value="org.citrusframework.rmi.remote.NewsService"/>
                    <header name="citrus_rmi_method" value="getNews"/>
                </headers>
            </message>
        </receive>
    </actions>
</test>
```

YAML

```
name: RmiMessageTest
actions:
  - receive:
      endpoint: "rmiServer"
      message:
        body: |
          <service-invocation
xmlns="http://www.citrusframework.org/schema/rmi/message">
              <remote>org.citrusframework.rmi.remote.NewsService</remote>
              <method>getNews</method>
          </service-invocation>
        headers:
          - name: citrus_rmi_interface
            value: "org.citrusframework.rmi.remote.NewsService"
          - name: citrus_rmi_method
            value: "getNews"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="RmiMessageTest">
        <actions>
            <receive endpoint="rmiServer">
                <message>
                    <payload>
                        <service-invocation
xmlns="http://www.citrusframework.org/schema/rmi/message">
                          <remote>org.citrusframework.rmi.remote.NewsService</remote>
                          <method>getNews</method>
                        </service-invocation>
                    </payload>
                </message>
                <header>
                    <element name="citrus_rmi_interface"
value="org.citrusframework.rmi.remote.NewsService"/>
                    <element name="citrus_rmi_method" value="getNews"/>
                </header>
            </receive>
        </actions>
    </testcase>
</spring:beans>
```

As you can see Citrus converts the incoming service invocation to a special XML representation which is passed as message payload to the test. As this is plain XML you can verify the RMI message content as usual using Citrus variables, functions and validation matchers.

Since we have received the method call we need to provide some return value for the client. As

usual we can specify the method return value with some XML representation.

Java

```
@CitrusTest
public void rmiMessageTest() {
    $(send(rmiServer)
        .message(RmiMessage.result("This is news from RMI!"))
    );
}
```

XML

```
<test name="RmiMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <send endpoint="rmiServer">
            <message>
                <body>
                    <payload>
                        <service-result
xmlns="http://www.citrusframework.org/schema/rmi/message">
                            <object type="java.lang.String" value="This is news from
RMI!"/>
                        </service-result>
                    </payload>
                </body>
            </message>
        </send>
    </actions>
</test>
```

YAML

```
name: RmiMessageTest
actions:
  - send:
      endpoint: "rmiServer"
      message:
        body: |
          <service-result xmlns="http://www.citrusframework.org/schema/rmi/message">
            <object type="java.lang.String" value="This is news from RMI!"/>
          </service-result>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="RmiMessageTest">
        <actions>
            <send endpoint="rmiServer">
                <message>
                    <payload>
                        <service-result
xmlns="http://www.citrusframework.org/schema/rmi/message">
                            <object type="java.lang.String" value="This is news from
RMI!"/>
                        </service-result>
                    </payload>
                </message>
            </send>
        </actions>
    </testcase>
</spring:beans>
```

The service result is defined as object with a **type** and **value** . The Citrus RMI remote interface method will return this value to the calling client. This would complete the successful remote service invocation. At this point we also have to think of choosing to raise some remote exception as service outcome.

Java

```
@CitrusTest
public void rmiMessageTest() {
    $(send(rmiServer)
        .message(RmiMessage.exception("Something went wrong"))
    );
}
```

XML

```
<test name="RmiMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <send endpoint="rmiServer">
            <message>
                <body>
                    <payload>
                        <service-result
xmlns="http://www.citrusframework.org/schema/rmi/message">
                            <exception>Something went wrong</exception>
                        </service-result>
                    </payload>
                </body>
            </message>
        </send>
    </actions>
</test>
```

YAML

```
name: RmiMessageTest
actions:
  - send:
      endpoint: "rmiServer"
      message:
        body: |
          <service-result xmlns="http://www.citrusframework.org/schema/rmi/message">
            <exception>Something went wrong</exception>
          </service-result>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="RmiMessageTest">
        <actions>
            <send endpoint="rmiServer">
                <message>
                    <payload>
                        <service-result
xmlns="http://www.citrusframework.org/schema/rmi/message">
                            <exception>Something went wrong</exception>
                        </service-result>
                    </payload>
                </message>
            </send>
        </actions>
    </testcase>
</spring:beans>
```

In the example above Citrus will not return some object as service result but raise a **java.rmi.RemoteException** with respective error message as specified in the test case. The calling client will receive the exception accordingly.

Chapter 29. JMX support

JMX is a standard Java API for making beans accessible to others in terms of management and remote configuration. JMX is the short term for Java Management Extensions and is often used in JEE application servers to manage bean attributes and operations from outside (e.g. another JVM). A managed bean server hosts multiple managed beans for JMX access. Remote connections to JMX can be realized with RMI (Remote method invocation) capabilities.

Citrus is able to connect to JMX managed beans as client and server. As a client Citrus can invoke managed bean operations and read write managed bean attributes. As a server Citrus is able to expose managed beans as mbean server. Clients can access those Citrus managed beans and get proper response objects as result. Doing so you can use the JVM platform managed bean server or some RMI registry for providing remote access.

The JMX components in Citrus are kept in a separate Maven module. So you should check that the module is available as Maven dependency in your project

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-jmx</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

As usual Citrus provides a customized jmx configuration schema that is used in Spring configuration files. Simply include the citrus-jmx namespace in the configuration XML files as follows.

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:citrus="http://www.citrusframework.org/schema/config"
      xmlns:citrus-jmx="http://www.citrusframework.org/schema/jmx/config"
      xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.citrusframework.org/schema/config
      http://www.citrusframework.org/schema/config/citrus-config.xsd
      http://www.citrusframework.org/schema/jmx/config
      http://www.citrusframework.org/schema/jmx/config/citrus-jmx-config.xsd">

      [...]

      </beans>
```

Now you are ready to use the customized Http configuration elements with the citrus-jmx namespace prefix.

Next sections describe the JMX message support in Citrus in more detail.

29.1. JMX client

On the client side we want to call some managed bean by either accessing managed attributes with read/write or by invoking a managed bean operation. For proper mbean server connectivity we should specify a client component for JMX that sends out mbean invocation calls.

```
<citrus-jmx:client id="jmxClient"
      server-url="platform"/>
```

The client component specifies the target managed bean server that we want to connect to. In this example we are using the JVM platform mbean server. This means we are able to access all JVM managed beans such as Memory, Threading and Logging. In addition to that we can access all custom managed beans that were exposed to the platform mbean server.

In most cases you may want to access managed beans on a different JVM or application server. So we need some remote connection to the foreign mbean server.

```
<citrus-jmx:client id="jmxClient"
      server-url="service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi"
      username="user"
      password="s!cr!t"
      auto-reconnect="true"
      delay-on-reconnect="5000"/>
```

In this example above we connect to a remote mbean server via RMI using the default RMI registry **localhost:1099** and the service name **jmxrmi** . Citrus is able to handle different remote transport protocols. Just define those in the **server-url** .

Now that we have setup the client component we can use it in a test case to access a managed bean.

Java

```
@CitrusTest
public void jmxMessageTest() {
    $(send(jmxClient)
        .message(JmxMessage.invocation("java.lang:type=Memory")
            .attribute("Verbose"))
    );
}
```

XML

```
<test name="JmxMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <send endpoint="jmxClient">
            <message>
                <body>
                    <payload>
                        <mbean-invocation
xmlns="http://www.citrusframework.org/schema/jmx/message">
                          <mbean>java.lang:type=Memory</mbean>
                          <attribute name="Verbose"/>
                        </mbean-invocation>
                    </payload>
                </body>
            </message>
        </send>
    </actions>
</test>
```

YAML

```
name: JmxMessageTest
actions:
  - send:
      endpoint: "jmxClient"
      message:
        body: |
          <mbean-invocation xmlns="http://www.citrusframework.org/schema/jmx/message">
              <mbean>java.lang:type=Memory</mbean>
              <attribute name="Verbose"/>
          </mbean-invocation>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="JmxMessageTest">
        <actions>
            <send endpoint="jmxClient">
                <message>
                    <payload>
                        <mbean-invocation
xmlns="http://www.citrusframework.org/schema/jmx/message">
                          <mbean>java.lang:type=Memory</mbean>
                          <attribute name="Verbose"/>
                        </mbean-invocation>
                    </payload>
                </message>
            </send>
        </actions>
    </testcase>
</spring:beans>
```

As you can see we just used a normal send action referencing the jmx client component that we have just added. The message payload is an XML representation of the managed bean access. This is a special Citrus XML representation. Citrus will convert this XML payload to the actuel managed bean access. In the example above we try to access a managed bean with object name **java.lang:type=Memory** . The object name is defined in JMX specification and consists of a key **java.lang:type** and a value **Memory** . So we identify the managed bean on the server by its type.

Now that we have access to the managed bean we can read its managed attributes such as **Verbose** . This is a boolean type attribute so the mbean invocation result will be a respective Boolean object. We can validate the managed bean attribute access in a receive action.

Java

```
@CitrusTest
public void jmxMessageTest() {
    $(receive(jmxClient)
        .message(JmxMessage.result(false))
    );
}
```

XML

```
<test name="JmxMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <receive endpoint="jmxClient">
            <message>
                <body>
                    <payload>
                        <mbean-result
xmlns="http://www.citrusframework.org/schema/jmx/message">
                          <object type="java.lang.Boolean" value="false"/>
                        </mbean-result>
                    </payload>
                </body>
            </message>
        </receive>
    </actions>
</test>
```

YAML

```
name: JmxMessageTest
actions:
  - receive:
      endpoint: "jmxClient"
      message:
        body: |
          <mbean-result xmlns="http://www.citrusframework.org/schema/jmx/message">
              <object type="java.lang.Boolean" value="false"/>
          </mbean-result>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="JmxMessageTest">
        <actions>
            <receive endpoint="jmxClient">
                <message>
                    <payload>
                        <mbean-result
xmlns="http://www.citrusframework.org/schema/jmx/message">
                          <object type="java.lang.Boolean" value="false"/>
                        </mbean-result>
                    </payload>
                </message>
            </receive>
        </actions>
    </testcase>
</spring:beans>
```

In the sample above we receive the mbean result and expect a **java.lang.Boolean** object return value. The return value content is also validated within the mbean result payload.

Some managed bean attributes might also be settable for us. So we can define the attribute access as write operation by specifying a value in the send action payload.

Java

```
@CitrusTest
public void jmxMessageTest() {
    $(send(jmxClient)
        .message(JmxMessage.invocation("java.lang:type=Memory")
            .attribute("Verbose", true))
    );
}
```

XML

```
<test name="JmxMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <send endpoint="jmxClient">
            <message>
                <body>
                    <payload>
                        <mbean-invocation
xmlns="http://www.citrusframework.org/schema/jmx/message">
                          <mbean>java.lang:type=Memory</mbean>
                          <attribute name="Verbose" value="true"
type="java.lang.Boolean"/>
                        </mbean-invocation>
                    </payload>
                </body>
            </message>
        </send>
    </actions>
</test>
```

YAML

```
name: JmxMessageTest
actions:
  - send:
      endpoint: "jmxClient"
      message:
        body: |
          <mbean-invocation xmlns="http://www.citrusframework.org/schema/jmx/message">
              <mbean>java.lang:type=Memory</mbean>
              <attribute name="Verbose" value="true" type="java.lang.Boolean"/>
          </mbean-invocation>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="JmxMessageTest">
        <actions>
            <send endpoint="jmxClient">
                <message>
                    <payload>
                        <mbean-invocation
xmlns="http://www.citrusframework.org/schema/jmx/message">
                          <mbean>java.lang:type=Memory</mbean>
                          <attribute name="Verbose" value="true"
type="java.lang.Boolean"/>
                        </mbean-invocation>
                    </payload>
                </message>
            </send>
        </actions>
    </testcase>
</spring:beans>
```

Now we have write access to the managed attribute **Verbose** . We do specify the value and its type **java.lang.Boolean** . This is how we can set attribute values on managed beans.

Last not least we are able to access managed bean operations.

Java

```
@CitrusTest
public void jmxMessageTest() {
    $(send(jmxClient)
        .message(JmxMessage.invocation("org.citrusframework.jmx.mbean:type=HelloBean")
            .operation("sayHello")
            .parameter("Hello JMX!"))
    );
}
```

XML

```
<test name="JmxMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <send endpoint="jmxClient">
            <message>
                <body>
                    <payload>
                        <mbean-invocation
xmlns="http://www.citrusframework.org/schema/jmx/message">
                          <mbean>org.citrusframework.jmx.mbean:type=HelloBean</mbean>
                          <operation name="sayHello">
                            >parameter>
                              >param type="java.lang.String" value="Hello JMX!"/>
                            >/parameter>
                          >/operation>
                        </mbean-invocation>
                    </payload>
                </body>
            </message>
        </send>
    </actions>
</test>
```

YAML

```
name: JmxMessageTest
actions:
  - send:
      endpoint: "jmxClient"
      message:
        body: |
          <mbean-invocation xmlns="http://www.citrusframework.org/schema/jmx/message">
              <mbean>org.citrusframework.jmx.mbean:type=HelloBean</mbean>
              <operation name="sayHello">
                >parameter>
                  >param type="java.lang.String" value="Hello JMX!"/>
                >/parameter>
              >/operation>
          </mbean-invocation>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="JmxMessageTest">
        <actions>
            <send endpoint="jmxClient">
                <message>
                    <payload>
                        <mbean-invocation
xmlns="http://www.citrusframework.org/schema/jmx/message">
                          <mbean>org.citrusframework.jmx.mbean:type=HelloBean</mbean>
                          <operation name="sayHello">
                            >parameter>
                              >param type="java.lang.String" value="Hello JMX!"/>
                            >/parameter>
                          >/operation>
                        </mbean-invocation>
                    </payload>
                </message>
            </send>
        </actions>
    </testcase>
</spring:beans>
```

In the example above we access a custom managed bean and invoke its operation **sayHello** . We are also using operation parameters for the invocation. This should call the managed bean operation and return its result if any as usual.

This completes the basic JMX managed bean access as client. Now we also want to discuss the server side were Citrus is able to provide managed beans for others

29.2. JMX server

The server side is always a little bit more tricky because we need to simulate custom managed bean access as a server. First of all Citrus provides a server component that specifies the connection properties for clients such as transport protocols, ports and mbean object names. Let’s create a new server that accepts incoming requests via RMI on a remote registry **localhost:1099** .

```
<citrus-jmx:server id="jmxServer"
      server-url="service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi">
    <citrus-jmx:mbeans>
        <citrus-jmx:mbean type="org.citrusframework.jmx.mbean.HelloBean"/>
        <citrus-jmx:mbean type="org.citrusframework.jmx.mbean.NewsBean"
objectDomain="org.citrusframework.news" objectName="name=News"/>
    </citrus-jmx:mbeans>
</citrus-jmx:server>
```

As usual we define a **server-url** that controls the JMX connector access to the mbean server. In this example above we open a JMX RMI connector for clients using the registry **localhost:1099** and the service name **jmxrmi** By default Citrus will not attempt to create this registry automatically so the registry has to be present before the server start up. With the optional server property **create-registry** set to **true** you can auto create the registry when the server starts up. These properties do only apply when using a remote JMX connector server.

Besides using the whole server-url as property we can also construct the connection by host, port, protocol and binding properties.

```
<citrus-jmx:server id="jmxServer"
      host="localhost"
      port="1099"
      protocol="rmi"
      binding="jmxrmi">
    <citrus-jmx:mbeans>
        <citrus-jmx:mbean type="org.citrusframework.jmx.mbean.HelloBean"/>
        <citrus-jmx:mbean type="org.citrusframework.jmx.mbean.NewsBean"
objectDomain="org.citrusframework.news" objectName="name=News"/>
    </citrus-jmx:mbeans>
</citrus-jmx:server>
```

On last thing to mention is that we could have also used **platform** as server-url in order to use the JVM platform mbean server instead.

Now that we clarified the connectivity we need to talk about how to define the managed beans that are available on our JMX mbean server. This is done as nested **mbean** configuration elements. Here the managed bean definitions describe the managed bean with its objectDomain, objectName, operations and attributes. The most convenient way of defining such managed bean definitions is to give a bean type which is the fully qualified class name of the managed bean. Citrus will use the package name and class name for proper objectDomain and objectName construction.

Let's have a closer look at the first mbean definition in the example above. So the first managed bean is defined by its class name **org.citrusframework.jmx.mbean.HelloBean** and therefore is accessible using the objectName **org.citrusframework.jmx.mbean:type=HelloBean** . In addition to that Citrus will read the class information such as available methods, getters and setters for constructing a proper MBeanInfo. In the second managed bean definition in our example we have used additional custom objectDomain and objectName values. So the **NewsBean** will be accessible with **org.citrusframework.news:name=News** on the managed bean server.

This is how we can define the bindings of managed beans and what clients need to search for when finding and accessing the managed beans on the server. When clients try to find the managed beans they have to use proper objectNames accordingly. ObjectNames that are not defined on the server will be rejected with managed bean not found error.

Right now we have to use the qualified class name of the managed bean in the definition. What happens if we do not have access to that mbean class or if there is not managed bean interface available at all? Citrus provides a generic managed bean that is able to handle any managed bean interaction. The generic bean implementation needs to know the managed operations and

attributes though. So let's define a new generic managed bean on our server:

```
<citrus-jmx:server id="jmxServer"
server-url="service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi">
    <citrus-jmx:mbeans>
        <citrus-jmx:mbean name="fooBean" objectDomain="foo.object.domain"
objectName="type=FooBean">
            <citrus-jmx:operations>
                <citrus-jmx:operation name="fooOperation">
                    <citrus-jmx:parameter>
                        <citrus-jmx:param type="java.lang.String"/>
                        <citrus-jmx:param type="java.lang.Integer"/>
                    </citrus-jmx:parameter>
                </citrus-jmx:operation>
                <citrus-jmx:operation name="barOperation"/>
            </citrus-jmx:operations>
            <citrus-jmx:attributes>
                <citrus-jmx:attribute name="fooAttribute" type="java.lang.String"/>
                <citrus-jmx:attribute name="barAttribute" type="java.lang.Boolean"/>
            </citrus-jmx:attributes>
        </citrus-jmx:mbean>
    </citrus-jmx:mbeans>
</citrus-jmx:server>
```

The generic bean definition needs to define all operations and attributes that are available for access. Up to now we are restricted to using Java base types when defining operation parameter and attribute return types. There is actually no way to define more complex return types. Nevertheless Citrus is now able to expose the managed bean for client access without having to know the actual managed bean implementation.

Now we can use the server component in a test case to receive some incoming managed bean access.

Java

```
@CitrusTest
public void jmxMessageTest() {
    $(receive(jmxServer)
        .message(JmxMessage.invocation("org.citrusframework.jmx.mbean:type=HelloBean")
            .operation("sayHello")
            .parameter("Hello JMX!"))
    );
}
```

XML

```
<test name="JmxMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <receive endpoint="jmxServer">
            <message>
                <body>
                    <payload>
                        <mbean-invocation
xmlns="http://www.citrusframework.org/schema/jmx/message">
                          <mbean>org.citrusframework.jmx.mbean:type=HelloBean</mbean>
                          <operation name="sayHello">
                            >parameter>
                              >param type="java.lang.String" value="Hello JMX!"/>
                            >/parameter>
                          </operation>
                        </mbean-invocation>
                    </payload>
                </body>
            </message>
        </receive>
    </actions>
</test>
```

YAML

```
name: JmxMessageTest
actions:
  - receive:
      endpoint: "jmxServer"
      message:
        body: |
          <mbean-invocation xmlns="http://www.citrusframework.org/schema/jmx/message">
              <mbean>org.citrusframework.jmx.mbean:type=HelloBean</mbean>
              <operation name="sayHello">
                >parameter>
                  >param type="java.lang.String" value="Hello JMX!"/>
                >/parameter>
              </operation>
          </mbean-invocation>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="JmxMessageTest">
        <actions>
            <receive endpoint="jmxServer">
                <message>
                    <payload>
                        <mbean-invocation
xmlns="http://www.citrusframework.org/schema/jmx/message">
                          <mbean>org.citrusframework.jmx.mbean:type=HelloBean</mbean>
                          <operation name="sayHello">
                            >parameter>
                              >param type="java.lang.String" value="Hello JMX!"/>
                            >/parameter>
                          </operation>
                        </mbean-invocation>
                    </payload>
                </message>
            </receive>
        </actions>
    </testcase>
</spring:beans>
```

In this very first example we expect a managed bean access to the bean **org.citrusframework.jmx.mbean:type=HelloBean** . We further expect the operation **sayHello** to be called with respective parameter values. Now we have to define the operation result that will be returned to the calling client as operation result.

Java

```
@CitrusTest
public void jmxMessageTest() {
    $(send(jmxServer)
        .message(JmxMessage.result("Hello from JMX!"))
    );
}
```

XML

```
<test name="JmxMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <send endpoint="jmxServer">
            <message>
                <body>
                    <payload>
                        <mbean-result
xmlns="http://www.citrusframework.org/schema/jmx/message">
                            <object type="java.lang.String" value="Hello from JMX!"/>
                        </mbean-result>
                    </payload>
                </body>
            </message>
        </send>
    </actions>
</test>
```

YAML

```
name: JmxMessageTest
actions:
  - send:
      endpoint: "jmxServer"
      message:
        body: |
          <mbean-result xmlns="http://www.citrusframework.org/schema/jmx/message">
            <object type="java.lang.String" value="Hello from JMX!"/>
          </mbean-result>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="JmxMessageTest">
        <actions>
            <send endpoint="jmxServer">
                <message>
                    <payload>
                        <mbean-result
xmlns="http://www.citrusframework.org/schema/jmx/message">
                            <object type="java.lang.String" value="Hello from JMX!"/>
                        </mbean-result>
                    </payload>
                </message>
            </send>
        </actions>
    </testcase>
</spring:beans>
```

The operation returns a String **Hello from JMX!** . This is how we can expect operation calls on managed beans. Now we already have seen that managed beans also expose attributes. The next example is handling incoming attribute read access.

Java

```
@CitrusTest
public void jmxMessageTest() {
    $(receive(jmxServer)
        .message(JmxMessage.invocation("org.citrusframework.news:name=News")
            .attribute("newsCount"))
    );

    $(send(jmxServer)
        .message(JmxMessage.result(100))
    );
}
```

XML

```
<test name="JmxMessageTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <receive endpoint="jmxServer">
            <message>
                <body>
                    <payload>
                        <mbean-invocation
xmlns="http://www.citrusframework.org/schema/jmx/message">
                          <mbean>org.citrusframework.news:name=News</mbean>
                            >attribute name="newsCount"/>
                        </mbean-invocation>
                    </payload>
                </body>
            </message>
        </receive>
        <send endpoint="jmxServer">
            <message>
                <body>
                    <payload>
                        <mbean-result
xmlns="http://www.citrusframework.org/schema/jmx/message">
                            <object type="java.lang.Integer" value="100"/>
                        </mbean-result>
                    </payload>
                </body>
            </message>
        </send>
    </actions>
</test>
```

YAML

```
name: JmxMessageTest
actions:
  - receive:
      endpoint: "jmxServer"
      message:
        body: |
          <mbean-invocation xmlns="http://www.citrusframework.org/schema/jmx/message">
              <mbean>org.citrusframework.news:name=News</mbean>
                >attribute name="newsCount"/>
          </mbean-invocation>
  - send:
      endpoint: "jmxServer"
      message:
        body: |
          <mbean-result xmlns="http://www.citrusframework.org/schema/jmx/message">
            <object type="java.lang.Integer" value="100"/>
          </mbean-result>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="JmxMessageTest">
        <actions>
            <receive endpoint="jmxServer">
                <message>
                    <payload>
                        <mbean-invocation
xmlns="http://www.citrusframework.org/schema/jmx/message">
                          <mbean>org.citrusframework.news:name=News</mbean>
                            >attribute name="newsCount"/>
                        </mbean-invocation>
                    </payload>
                </message>
            </receive>

            <send endpoint="jmxServer">
                <message>
                    <payload>
                        <mbean-result
xmlns="http://www.citrusframework.org/schema/jmx/message">
                            <object type="java.lang.Integer" value="100"/>
                        </mbean-result>
                    </payload>
                </message>
            </send>
        </actions>
    </testcase>
</spring:beans>
```

The receive action expects read access to the **NewsBean** attribute **newsCount** and returns a result object of type **java.lang.Integer** . This way we can expect all attribute access to our managed beans. Write operations will have an attribute value specified.

This completes the JMX server capabilities with managed bean access on operations and attributes.

Chapter 30. Zookeeper support

Citrus provides configuration components and test actions for interacting with Zookeeper. The Citrus Zookeeper client component executes commands like create-node, check node-exists, delete-node, get node-data or set node-data. As a user you can execute Zookeeper commands as part of a Citrus test and validate possible command results.

> The Zookeeper test components in Citrus are kept in a separate Maven module. If not already done, you have to include the module as Maven dependency to your project

```
<dependency>
      <groupId>org.citrusframework</groupId>
      <artifactId>citrus-zookeeper</artifactId>
      <version>${citrus.version}</version>
      </dependency>
```

Citrus provides a "citrus-zookeeper" configuration namespace and schema definition for Zookeeper related components and actions. Include this namespace into your Spring configuration in order to use the Citrus zookeeper configuration elements. The namespace URI and schema location are added to the Spring configuration XML file as follows.

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:citrus-zookeeper="http://www.citrusframework.org/schema/zookeeper/config"
      xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.citrusframework.org/schema/zookeeper/config
      http://www.citrusframework.org/schema/zookeeper/config/citrus-zookeeper-
config.xsd">

      [...]

      </beans>
```

After that you are able to use customized Citrus XML elements in order to define the Spring beans.

30.1. Zookeeper client

Before you can interact with a Zookeeper server you have to configure the Zookeeper client. A sample configuration is provided below describing the configuration options available:

```
<citrus-zookeeper:client id="zookeeperClient"
                                 url="http://localhost:21118"
                                 timeout="2000"/>
```

This is a typical client configuration for connecting to a Zookeeper server. Now you are able to execute several commands. These commands will be sent to the Zookeeper server for execution.

30.2. Zookeeper commands

See below all available Zookeeper commands that a Citrus client is able to execute.

```
info: Retrieves the current state of the client connection
create: Creates a znode in a specified path of the ZooKeeper namespace
delete: Deletes a znode from a specified path of the ZooKeeper namespace
exists: Checks if a znode exists in the path
children: Gets a list of children of a znode
get: Gets the data associated with a znode
set: Sets/writes data into the data field of a znode
```

Before we see some of these commands in action we have to add a new test namespace to our test case when using the XML DSL.

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:zookeeper="http://www.citrusframework.org/schema/zookeeper/testcase"
      xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.citrusframework.org/schema/zookeeper/testcase
      http://www.citrusframework.org/schema/zookeeper/testcase/citrus-zookeeper-
testcase.xsd">

      [...]

      </beans>
```

We added the Zookeeper namespace with prefix **zookeeper:** so now we can start to add special test actions to the test case:

XML DSL

```
<zookeeper:create zookeeper-client="zookeeperClient" path="/${randomString}"
acl="OPEN_ACL_UNSAFE" mode="PERSISTENT">
  <zookeeper:data>foo</zookeeper:data>
  <zookeeper:expect>
    <zookeeper:result>
      <![CDATA[
        {
          "responseData":{
              "path":"/${randomString}"
          }
        }
      ]]>
    </zookeeper:result>
  </zookeeper:expect>
</zookeeper:create>

<zookeeper:get zookeeper-client="zookeeperClient" path="/${randomString}">
  <zookeeper:expect>
    <zookeeper:result>
      <![CDATA[
      {
        "responseData":{
          "data":"foo"
        }
      }
      ]]>
    </zookeeper:result>
  </zookeeper:expect>
</zookeeper:getData>

<zookeeper:set zookeeper-client="zookeeperClient" path="/${randomString}">
  <zookeeper:data>bar</zookeeper:data>
</zookeeper:setData>
```

When using the Java DSL we can directly configure the commands with a fluent API.

Java DSL

```
@CitrusTest
public void testZookeeper() {
    variable("randomString", "citrus:randomString(10)");

    zookeeper()
        .create("/${randomString}", "foo")
        .acl("OPEN_ACL_UNSAFE")
        .mode("PERSISTENT")
        .validateCommandResult(new CommandResultCallback<ZooResponse>() {
            @Override
            public void doWithCommandResult(ZooResponse result, TestContext context) {
                Assert.assertEquals(result.getResponseData().get("path"),
context.replaceDynamicContentInString("/${randomString}"));
            }
        });

    zookeeper()
        .get("/${randomString}")
        .validateCommandResult(new CommandResultCallback<ZooResponse>() {
            @Override
            public void doWithCommandResult(ZooResponse result, TestContext context) {
                Assert.assertEquals(result.getResponseData().get("version"), 0);
            }
        });

    zookeeper()
        .set("/${randomString}", "bar");
}
```

The examples above create a new znode in Zookeeper using a **randomString** as path. We can get and set the data with expecting and validating the result of the Zookeeper server. This is basically the idea of integrating Zookepper operations to a Citrus test. This opens the gate to manage Zookeeper related entities within a Citrus test. We can manipulate and validate the znodes on the Zookeeper instance.

Zookeeper keeps its nodes in a hierarchical storage. This means a znode can have children and we can add and remove those. In Citrus you can get all children of a znode and manage those within the test:

XML DSL

```
<zookeeper:create zookeeper-client="zookeeperClient" path="/${randomString}/child1"
acl="OPEN_ACL_UNSAFE" mode="EPHEMERAL">
  <zookeeper:data></zookeeper:data>
  <zookeeper:expect>
    <zookeeper:result>
      <![CDATA[
        {
          "responseData":{
              "path":"/${randomString}/child1"
          }
        }
      ]]>
    </zookeeper:result>
  </zookeeper:expect>
</zookeeper:create>

<zookeeper:create zookeeper-client="zookeeperClient" path="/${randomString}/child2"
acl="OPEN_ACL_UNSAFE" mode="EPHEMERAL">
  <zookeeper:data></zookeeper:data>
  <zookeeper:expect>
    <zookeeper:result>
      <![CDATA[
        {
          "responseData":{
              "path":"/${randomString}/child2"
          }
        }
      ]]>
    </zookeeper:result>
  </zookeeper:expect>
</zookeeper:create>

<zookeeper:children zookeeper-client="zookeeperClient" path="/${randomString}">
  <zookeeper:expect>
    <zookeeper:result>
      <![CDATA[
        {
          "responseData":{
              "children":["child1","child2"]
          }
        }
      ]]>
    </zookeeper:result>
  </zookeeper:expect>
</zookeeper:children>
```

Java DSL

```
zookeeper()
    .create("/${randomString}/child1", "")
    .acl("OPEN_ACL_UNSAFE")
    .mode("PERSISTENT")
    .validateCommandResult(new CommandResultCallback<ZooResponse>() {
        @Override
        public void doWithCommandResult(ZooResponse result, TestContext context) {
            Assert.assertEquals(result.getResponseData().get("path"),
context.replaceDynamicContentInString("/${randomString}/child1"));
        }
    });

zookeeper()
    .create("/${randomString}/child2", "")
    .acl("OPEN_ACL_UNSAFE")
    .mode("PERSISTENT")
    .validateCommandResult(new CommandResultCallback<ZooResponse>() {
        @Override
        public void doWithCommandResult(ZooResponse result, TestContext context) {
            Assert.assertEquals(result.getResponseData().get("path"),
context.replaceDynamicContentInString("/${randomString}/child2"));
        }
    });

zookeeper()
    .children("/${randomString}")
    .validateCommandResult(new CommandResultCallback<ZooResponse>() {
        @Override
        public void doWithCommandResult(ZooResponse result, TestContext context) {
            Assert.assertEquals(result.getResponseData().get("children").toString(),
"[child1, child2]");
        }
    });
```

Chapter 31. Spring Restdocs support

Spring Restdocs project helps to easily generate API documentation for RESTful services. While messages are exchanged the Restdocs library generates request/response snippets and API documentation. You can add the Spring Restdocs documentation to the Citrus client components for Http **and** SOAP endpoints.

> The Spring Restdocs support components in Citrus are kept in a separate Maven module. If not already done, you have to include the module as Maven dependency to your project

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-restdocs</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

For easy configuration Citrus has created a separate namespace and schema definition for Spring Restdocs related documentation. Include this namespace into your Spring configuration in order to use the Citrus Restdocs configuration elements. The namespace URI and schema location are added to the Spring configuration XML file as follows.

```
<spring:beans xmlns:spring="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns="http://www.citrusframework.org/schema/cucumber/testcase"
     xsi:schemaLocation="
     http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.citrusframework.org/schema/restdocs/config
     http://www.citrusframework.org/schema/restdocs/config/citrus-restdocs-
config.xsd">

    [...]

</spring:beans>
```

After that you are able to use customized Citrus XML elements in order to define the Spring beans.

31.1. Spring Restdocs using Http

First of all we concentrate on adding the Spring Restdocs feature to Http client communication. The next sample configuration uses the new Spring Restdocs components in Citrus:

```
<citrus-restdocs:documentation id="restDocumentation"
                                                output-directory="target/citrus-
docs/generated-snippets"
                                                identifier="rest-docs/{method-name}"/>
```

The above component adds a new documentation configuration. Behind the scenes the component creates a new restdocs configurer and a client interceptor. We can reference the new restdocs component in **citrus-http** client components like this:

```
<citrus-http:client id="httpClient"
          request-url="http://localhost:8080/test"
          request-method="POST"
          interceptors="restDocumentation"/>
```

The Spring Restdocs documentation component acts as a client interceptor. Every time the client component is used to send and receive a message the restdocs interceptor will automatically create its API documentation. The configuration **identifier** attribute describes the output format **rest-docs/{method-name}** which results in a folder layout like this:

```
target/citrus-docs
  |- rest-docs
    |- test-a
      |- curl-request.adoc
      |- http-request.adoc
      |- http-response.adoc
    |- test-b
      |- curl-request.adoc
      |- http-request.adoc
      |- http-response.adoc
    |- test-c
      |- curl-request.adoc
      |- http-request.adoc
      |- http-response.adoc
```

The example above is the result of three test cases each of them performing a client Http request/response communication. Each test message exchange is documented with separate files:

curl-request.adoc

```
$ curl 'http://localhost:8080/test' -i -X POST -H 'Accept: application/xml' -H
'CustomHeaderId: 123456789' -H 'Content-Type: application/xml;charset=UTF-8' -H
'Accept-Charset: utf-8' -d '<testRequestMessage>
    <text>Hello HttpServer>/text>
</testRequestMessage>'
```

The curl file represents the client request as curl command and can be seen as a sample to

reproduce the request.

http-request.adoc

```
POST /test HTTP/1.1
Accept: application/xml
CustomHeaderId: 123456789
Content-Type: application/xml;charset=UTF-8
Content-Length: 118
Accept-Charset: utf-8
Host: localhost

<testRequestMessage>
    <text>Hello HttpServer>/text>
</testRequestMessage>
```

The **http-request.adoc** file represents the sent message data for the client request. The respective **http-response.adoc** represents the response that was sent to the client.

http-response.adoc

```
HTTP/1.1 200 OK
Date: Tue, 07 Jun 2016 12:10:46 GMT
Content-Type: application/xml;charset=UTF-8
Accept-Charset: utf-8
Content-Length: 122
Server: Jetty(9.2.15.v20160210)

<testResponseMessage>
    <text>Hello Citrus!>/text>
</testResponseMessage>
```

Nice work! We have automatically created snippets for the RESTful API by just adding the interceptor to the Citrus client component. Spring Restdocs components can be combined manually. See the next configuration that uses this approach.

```
<citrus-restdocs:configurer id="restDocConfigurer" output-directory="target/citrus-
docs/generated-snippets"/>
<citrus-restdocs:client-interceptor id="restDocClientInterceptor" identifier="rest-
docs/{method-name}"/>

<util:list id="restDocInterceptors">
    <ref bean="restDocConfigurer"/>
    <ref bean="restDocClientInterceptor"/>
</util:list>
```

```
<citrus-http:client id="httpClient"
          request-url="http://localhost:8080/test"
          request-method="POST"
          interceptors="restDocInterceptors"/>
```

What exactly is the difference to the **citrus-restdocs:documentation** that we have used before? In general there is no difference. Both configurations are identical in its outcome. Why should someone use the second approach then? It is more verbose as we need to also define a list of interceptors. The answer is easy. If you want to combine the restdocs interceptors with other client interceptors in a list then you should use the manual combination approach. We can add basic authentication interceptors for instance to the list of interceptors then. The more comfortable **citrus-restdocs:documentation** component only supports exclusive restdocs interceptors.

31.2. Spring Restdocs using SOAP

You can use the Spring Restdocs features also for SOAP clients in Citrus. This is a controversy idea as SOAP endpoints are different to RESTful concepts. But at the end SOAP Http communication is Http communication with request and response messages. Why should we miss out the fantastic documentation feature here just because of ideology reasons.

The concept of adding the Spring Restdocs documentation as interceptor to the client is still the same.

```
<citrus-restdocs:documentation id="soapDocumentation"
                                               type="soap"
                                               output-directory="target/citrus-
docs/generated-snippets"
                                               identifier="soap-docs/{method-name}"/>
```

We have added a **type** setting with value **soap** . And that is basically all we need to do. Now Citrus knows that we would like to add documentation for a SOAP client:

```
<citrus-ws:client id="soapClient"
      request-url="http://localhost:8080/test"
      interceptors="soapDocumentation"/>
```

Following from that the **soapClient** is enabled to generate Spring Restdocs documentation for each request/response. The generated snippets then do represent the SOAP request and response messages.

http-request.adoc

```
POST /test HTTP/1.1
SOAPAction: "test"
Accept: application/xml
CustomHeaderId: 123456789
Content-Type: application/xml;charset=UTF-8
Content-Length: 529
Accept-Charset: utf-8
Host: localhost

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
    <Operation xmlns="http://citrusframework.org/test">sayHello>/Operation>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <testRequestMessage>
      <text>Hello HttpServer>/text>
    </testRequestMessage>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

http-response.adoc

```
HTTP/1.1 200 OK
Date: Tue, 07 Jun 2016 12:10:46 GMT
Content-Type: application/xml;charset=UTF-8
Accept-Charset: utf-8
Content-Length: 612
Server: Jetty(9.2.15.v20160210)

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
    >Operation xmlns="http://citrusframework.org/test">sayHello>/Operation>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <testResponseMessage>
      <text>Hello Citrus!>/text>
    </testResponseMessage>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The file names are still using **http-request** and **http-response** but the content is clearly the SOAP request/response message data.

31.3. Spring Restdocs in Java DSL

How can we use Spring Restdocs in Java DSL? Of course we have special support in Citrus Java DSL for the Spring Restdocs configuration, too.

Java DSL

```
public class RestDocConfigurationIT extends TestNGCitrusTestDesigner {

    @Autowired
    private TestListeners testListeners;

    private HttpClient httpClient;

    @BeforeClass
    public void setup() {
        CitrusRestDocConfigurer restDocConfigurer =
CitrusRestDocsSupport.restDocsConfigurer(new
ManualRestDocumentation("target/generated-snippets"));
        RestDocClientInterceptor restDocInterceptor =
CitrusRestDocsSupport.restDocsInterceptor("rest-docs/{method-name}");

        httpClient = CitrusEndpoints.http()
            .client()
            .requestUrl("http://localhost:8073/test")
            .requestMethod(HttpMethod.POST)
            .contentType("text/xml")
            .interceptors(Arrays.asList(restDocConfigurer, restDocInterceptor))
            .build();

        testListeners.addTestListener(restDocConfigurer);
    }

    @Test
    @CitrusTest
    public void testRestDocs() {
        http().client(httpClient)
            .send()
            .post()
            .payload("<testRequestMessage>" +
                      "<text>Hello HttpServer</text>" +
                  "</testRequestMessage>");

        http().client(httpClient)
            .receive()
            .response(HttpStatus.OK)
            .payload("<testResponseMessage>" +
                      "<text>Hello TestFramework</text>" +
                  "</testResponseMessage>");
    }
}
```

The mechanism is quite similar to the XML configuration. We add the Restdocs configurer and interceptor to the list of interceptors for the Http client. If we do this all client communication is automatically documented. The Citrus Java DSL provides some convenient configuration methods

in class **CitrusRestDocsSupport** for creating the configurer and interceptor objects.

> The configurer must be added to the list of test listeners. This is a mandatory step in order to enable the configurer for documentation preparations before each test. Otherwise we would not be able to generate proper documentation. If you are using the XML configuration this is done automatically for you.

Chapter 32. Dynamic endpoint components

Endpoints represent the central components in Citrus to send or receive a message on some destination. Usually endpoints get defined in the basic Citrus Spring application context configuration as Spring bean components. In some cases this might be over engineering as the tester just wants to send or receive a message. In particular this is done when doing sanity checks in server endpoints while debugging a certain scenario.

With endpoint components you are able to create the Citrus endpoint for sending and receiving a message at test runtime. There is no additional configuration or Spring bean component needed. You just use the endpoint uri in a special naming convention and Citrus will create the endpoint for you. Let us see a first example of this scenario:

```
<testcase name="DynamicEndpointTest">
    <actions>
        <send endpoint="jms:Hello.Queue?timeout=10000">
            <message>
                <payload>
                [...]
                </payload>
            </message>
        </send>

        <receive endpoint="jms:Hello.Response.Queue?timeout=5000">
            <message>
                <payload>
                [...]
                </payload>
            </message>
        </receive>
    </actions>
</testcase>
```

As you can see the endpoint uri just goes into the test case action in substitution to the usual endpoint reference name. Instead of referencing a bean id that points to the previously configured Citrus endpoint we use the endpoint uri directly. The endpoint uri should give all information to create the endpoint at runtime. In the example above we use a keyword **jms:** which tells Citrus that we need to create a JMS message endpoint. Secondly we give the JMS destination name **Hello.Queue** which is a mandatory part of the endpoint uri when using the JMS component. The optional timeout parameter completed the uri. Citrus is able to create the JMS endpoint at runtime sending the message to the defined destination via JMS.

Of course this mechanism is not limited to JMS endpoints. We can use all default Citrus message transports in the endpoint uri. Just pick the right keyword that defines the message transport to use. Here is a list of supported keywords:

jms	Creates a JMS endpoint for sending and receiving message to a queue or topic

channel Creates a channel endpoint for sending and receiving messages using an in memory Spring Integration message channel

http Creates an HTTP client for sending a request to some server URL synchronously waiting for the response message

ws Creates a Web Socket client for sending messages to or receiving messages from a Web Socket server

soap Creates a SOAP WebService client that send a proper SOAP message to the server URL and waits for the synchronous response to arrive

ssh Creates a new ssh client for publishing a command to the server

mail or smtp: Creates a new mail client for sending a mail mime message to an SMTP server

camel Creates a new Apache Camel endpoint for sending and receiving Camel exchanges both to and from Camel routes.

vertx or eventbus: Creates a new Vert.x instance sending and receiving messages with the network event bus

rmi Creates a new RMI client instance sending and receiving messages for method invocation on remote interfaces

jmx Creates a new JMX client instance sending and receiving messages to and from a managed bean server.

Depending on the message transport we have to add mandatory parameters to the endpoint uri. In the JMS example we had to specify the destination name. The mandatory parameters are always part of the endpoint uri. Optional parameters can be added as key value pairs to the endpoint uri. The available parameters depend on the endpoint keyword that you have chosen. See these example endpoint uri expressions:

```
    jms:queuename?connectionFactory=specialConnectionFactory&timeout=10000
    jms:topic:topicname?connectionFactory=topicConnectionFactory

jms:sync:queuename?connectionFactory=specialConnectionFactory&pollingInterval=100&repl
yDestination=myReplyDestination

    channel:channelName
    channel:sync:channelName
    channel:channelName?timeout=10000&channelResolver=myChannelResolver

    http:localhost:8088/test
```

```
    http://localhost:8088/test

http:localhost:8088?requestMethod=GET&timeout=10000&errorHandlingStrategy=throwsExcept
ion&requestFactory=myRequestFactory
    http://localhost:8088/test?requestMethod=DELETE&customParam=foo

    websocket:localhost:8088/test
    websocket://localhost:8088/test
    ws:localhost:8088/test
    ws://localhost:8088/test

    soap:localhost:8088/test

soap:localhost:8088?timeout=10000&errorHandlingStrategy=propagateError&messageFactory=
myMessageFactory

    mail:localhost:25000
    smtp://localhost:25000

smtp://localhost?timeout=10000&username=foo&password=1234&mailMessageMapper=myMapper

    ssh:localhost:2200

ssh://localhost:2200?timeout=10000&strictHostChecking=true&user=foo&password=12345678

    rmi://localhost:1099/someService
    rmi:localhost/someService&timeout=10000

    jmx:rmi:///jndi/rmi://localhost:1099/someService
    jmx:platform&timeout=10000

    camel:direct:address
    camel:seda:address
    camel:jms:queue:someQueue?connectionFactory=myConnectionFactory

camel:activemq:queue:someQueue?concurrentConsumers=5&destination.consumer.prefetchSize
=50
    camel:controlbus:route?routeId=myRoute&action=status

    vertx:addressName
    vertx:addressName?port=10105&timeout=10000&pubSubDomain=true
    vertx:addressName?vertxInstanceFactory=vertxFactory
```

The optional parameters get directly set as endpoint configuration. You can use primitive values as well as Spring bean id references. Citrus will automatically detect the target parameter type and resolve the value to a Spring bean in the application context if necessary. If you use some unknown parameter Citrus will raise an exception at runtime as the endpoint could not be created properly.

In synchronous communication we have to reuse endpoint components in order to receive synchronous messages on reply destinations. This is a problem when using dynamic endpoints as

the endpoints get created at runtime. Citrus uses a caching of endpoints that get created at runtime. Following from that we have to use the exact same endpoint uri in your test case in order to get the cached endpoint instance. With this little trick synchronous communication will work just as it is done with static endpoint components. Have a look at this sample test:

```
<testcase name="DynamicEndpointTest">
    <actions>
        <send endpoint="jms:sync:Hello.Sync.Queue">
            <message>
                <payload>
                [...]
                </payload>
            </message>
        </send>

        <receive endpoint="jms:sync:Hello.Sync.Queue">
            <message>
                <payload>
                [...]
                </payload>
            </message>
        </receive>
    </actions>
</testcase>
```

As you can see we used the exact dynamic endpoint uri in both send and receive actions. Citrus is then able to reuse the same dynamic endpoint and the synchronous reply will be received as expected. However the reuse of exactly the same endpoint uri might get annoying as we also have to copy endpoint uri parameters and so on.

```
<testcase name="DynamicEndpointTest">
    <actions>
        <send endpoint="http://localhost:8080/HelloService?user=1234567">
            <message>
                <payload>
                [...]
                </payload>
            </message>
        </send>

        <receive endpoint="http://localhost:8080/HelloService?user=1234567">
            <message>
                <payload>
                [...]
                </payload>
            </message>
        </receive>
    </actions>
</testcase>
```

We have to use the exact same endpoint uri when receiving the synchronous service response. This is not very straight forward. This is why Citrus also supports dynamic endpoint names. With a special endpoint uri parameter called **endpointName** you can name the dynamic endpoint. In a corresponding receive action you just use the endpoint name as reference which makes life easier:

```
<testcase name="DynamicEndpointTest">
    <actions>
        <send endpoint="http://localhost:8080/HelloService?endpointName=myHttpClient">
            <message>
                <payload>
                [...]
                </payload>
            </message>
        </send>

        <receive endpoint="http://localhost?endpointName=myHttpClient">
            <message>
                <payload>
                [...]
                </payload>
            </message>
        </receive>
    </actions>
</testcase>
```

So we can reference the dynamic endpoint with the given name. The internal **endpointName** uri parameter is automatically removed before sending out messages. Once again the dynamic endpoint uri mechanism provides a fast way to write test cases in Citrus with less configuration.

But you should consider to use the static endpoint components defined in the basic Spring bean application context for endpoints that are heavily reused in multiple test cases.

Chapter 33. Endpoint adapter

Endpoint adapter help to customize the behavior of a Citrus server such as HTTP or SOAP web servers. As the servers get started with the Citrus context they are ready to receive incoming client requests. Now there are different ways to process these incoming requests and to provide a proper response message. By default the server will forward the incoming request to an in-memory message channel where a test can receive the message and provide a synchronous response. This message channel handling is done automatically behind the scenes so the tester does not care about these things. The tester just uses the server directly as endpoint reference in the test case. This is the default behaviour. In addition to that you can define custom endpoint adapters on the Citrus server in order to change this default behavior.

You set the custom endpoint adapter directly on the server configuration as follows:

```
<citrus-http:server id="helloHttpServer"
    port="8080"
    auto-start="true"
    endpoint-adapter="emptyResponseEndpointAdapter"
    resource-base="src/it/resources"/>

    <citrus:empty-response-adapter id="emptyResponseEndpointAdapter"/>
```

Now let us have a closer look at the provided endpoint adapter implementations.

33.1. Empty response endpoint adapter

This is the simplest endpoint adapter you can think of. It simply provides an empty success response using the HTTP response code **200** . The adapter does not need any configurations or properties as it simply responds with an empty HTTP response.

```
<citrus:empty-response-adapter id="emptyResponseEndpointAdapter"/>
```

33.2. Static response endpoint adapter

The next more complex endpoint adapter will always return a static response message.

```
<citrus:static-response-adapter id="endpointAdapter">
    <citrus:payload>
        <![CDATA[
          <HelloResponse
            xmlns="http://citrusframework.org/schemas/samples/sayHello.xsd">
              <MessageId>123456789</MessageId>
              <CorrelationId>Cx1x123456789</CorrelationId>
              <Text>Hello User</Text>
          </HelloResponse>
        ]]>
    </citrus:payload>
    <citrus:header>
        <citrus:element
name="{http://citrusframework.org/schemas/samples}h1:Operation"
                  value="sayHello"/>
        <citrus:element
name="{http://citrusframework.org/schemas/samples}h1:MessageId"
                  value="123456789"/>
    </citrus:header>
 </citrus:static-response-adapter>
```

The endpoint adapter is configured with a static message payload and static response header values. The response to the client is therefore always the same. You can add dynamic values by using Citrus functions such as **randomString** or **randomNumber**. Also we are able to use values of the actual request message that has triggered the response adapter. The request is available via the local message store. In combination with Xpath or JsonPath functions we can map values from the actual request.

```
<citrus:static-response-adapter id="endpointAdapter">
    <citrus:payload>
        <![CDATA[
          <HelloResponse
            xmlns="http://citrusframework.org/schemas/samples/sayHello.xsd">
              <MessageId>citrus:randomNumber(10)</MessageId>
              <CorrelationId>citrus:xpath(citrus:message(request.body()),
'/hello:HelloRequest/hello:CorrelationId')</CorrelationId>
              <Text>Hello User</Text>
          </HelloResponse>
        ]]>
    </citrus:payload>
    <citrus:header>
        <citrus:element
name="{http://citrusframework.org/schemas/samples}h1:Operation"
                  value="sayHello"/>
        <citrus:element
name="{http://citrusframework.org/schemas/samples}h1:MessageId"
                  value="citrus:randomNumber(10)"/>
    </citrus:header>
 </citrus:static-response-adapter>
```

The example above maps the **CorrelationId** of the **HelloRequest** message to the response with Xpath function. The local message store automatically has the message named **request** stored so we can access the payload with this message name.

> XML is namespace specific so we need to use the namespace prefix **hello** in the Xpath expression. The namespace prefix should evaluate to a global namespace entry in the global Citrus xpath-namespace.

33.3. Request dispatching endpoint adapter

The idea behind the request dispatching endpoint adapter is that the incoming requests are dispatched to several other endpoint adapters. The decision which endpoint adapter should handle the actual request is done depending on some adapter mapping. The mapping is done based on the payload or header data of the incoming request. A mapping strategy evaluates a mapping key using the incoming request. You can think of an XPath expression that evaluates to the mapping key for instance. The endpoint adapter that maps to the mapping key is then called to handle the request.

So the request dispatching endpoint adapter is able to dynamically call several other endpoint adapters based on the incoming request message at runtime. This is very powerful. The next example uses the request dispatching endpoint adapter with a XPath mapping key extractor.

```
<citrus:dispatching-endpoint-adapter id="dispatchingEndpointAdapter"
         mapping-key-extractor="mappingKeyExtractor"
         mapping-strategy="mappingStrategy"/>

<bean id="mappingStrategy"
  class="org.citrusframework.endpoint.adapter.mapping.SimpleMappingStrategy">
    <property name="adapterMappings">
      <map>
          <entry key="sayHello" ref="helloEndpointAdapter"/>
      </map>
    </property>
</bean>

<bean id="mappingKeyExtractor"

class="org.citrusframework.endpoint.adapter.mapping.XPathPayloadMappingKeyExtractor">
    <property name="xpathExpression" value="//TestMessage/Operation/*"/>
</bean>

<citrus:static-response-adapter id="helloEndpointAdapter">
    <citrus:payload>
        <![CDATA[
            <HelloResponse
                xmlns="http://citrusframework.org/schemas/samples/sayHello.xsd">
                <MessageId>123456789</MessageId>
                <Text>Hello User</Text>
            </HelloResponse>
        ]]>
    </citrus:payload>
</citrus:static-response-adapter>
```

The XPath mapping key extractor expression decides for each request which mapping key to use in order to find a proper endpoint adapter through the mapping strategy. The endpoint adapters available in the application context are mapped via their bean id. For instance an incoming request with a matching element ***//TestMessage/Operation/sayHello*** would be handled by the endpoint adapter bean that is registered in the mapping strategy as "sayHello" key. The available endpoint adapters are configured in the same Spring application context.

Citrus provides several default mapping key extractor implementations.

HeaderMappingKeyExtractor	Reads a special header entry and uses its value as mapping key
SoapActionMappingKeyExtractor	Uses the soap action header entry as mapping key
XPathPayloadMappingKeyExtractor	Evaluates a XPath expression on the request payload and uses the result as mapping key

In addition to that we need a mapping strategy. Citrus provides following default implementations.

SimpleMappingStrategy	Simple key value map with endpoint adapter references
BeanNameMappingStrategy	Loads the endpoint adapter Spring bean with the given id matching the mapping key
ContextLoadingMappingStrategy	Same as BeanNameMappingStrategy but loads a separate application context defined by external file resource

33.4. Channel endpoint adapter

The channel connecting endpoint adapter is the default adapter used in all Citrus server components. Indeed this adapter also provides the most flexibility. This adapter forwards incoming requests to a channel destination. The adapter is waiting for a proper response on a reply destination synchronously. With the channel endpoint components you can read the requests on the channel and provide a proper response on the reply destination.

```
<citrus-si:channel-endpoint-adapter id="channelEndpointAdapter"
              channel-name="inbound.channel"
              timeout="2500"/>
```

33.5. JMS endpoint adapter

Another powerful endpoint adapter is the JMS connecting adapter implementation. This adapter forwards incoming requests to a JMS destination and waits for a proper response on a reply destination. A JMS endpoint can access the requests internally and provide a proper response on the reply destination. So this adapter is very flexible to provide proper response messages.

This special adapter comes with the **citrus-jms** module. So you have to add the module and the special XML namespace for this module to your configuration files. The Maven module for **citrus-jms** goes to the Maven POM file as normal project dependency. The **citrus-jms** namespace goes to the Spring bean XML configuration file as follows:

Citrus provides a "citrus-jms" configuration namespace and schema definition for JMS related components and features. Include this namespace into your Spring configuration in order to use the Citrus JMS configuration elements. The namespace URI and schema location are added to the Spring configuration XML file as follows.

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:citrus-jms="http://www.citrusframework.org/schema/jms/config"
        xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.citrusframework.org/schema/jms/config
        http://www.citrusframework.org/schema/jms/config/citrus-jms-config.xsd">

    [...]

</beans>
```

After that you are able to use the adapter implementation in the Spring bean configuration.

```
<citrus-jms:endpoint-adapter id="jmsEndpointAdapter"
              destination-name="JMS.Queue.Requests.In"
              reply-destination-name="JMS.Queue.Response.Out"
              connection-factory="jmsConnectionFactory"
              timeout="2500"/>

<bean id="jmsConnectionFactory"
class="org.apache.activemq.artemis.jms.client.ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>
```

Chapter 34. Connectors

Connectors are quite similar to endpoints. The modules connect Citrus to a certain technology or framework rather than implementing a message transport (client and server) like endpoints usually do.

Connectors typically provide a client side only implementation that enable Citrus to interact with a service or framework (e.g. Docker, Kubernetes, Knative, Selenium, OpenAPI specification).

Chapter 35. OpenAPI support

OpenAPI is a popular specification language to describe HTTP APIs and its exposure to clients. Citrus is able to leverage the specification to auto generate client and server request/response message data. The generated message data follows the rules of a given operation in the specification. In particular, the message body is generated from the given Json schema rules in that specification. This way users may do contract-driven testing where the client and the server ensure the conformity with the contract to obey to the same specification rules.

> The OpenAPI support in Citrus get enabled by adding a separate Maven module as dependency to your project

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-openapi</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

35.1. OpenAPI specification

The OpenAPI test actions in Citrus uses a specification which usually is a json or yaml document shared between the components.

Sometimes the specification gets exposed by a server application via HTTP endpoint. You can directly load the specification from the HTTP URL. Or you may just point the OpenAPI components to a local specification file.

Citrus supports OpenAPI on both client and server components so the next sections will describe the usage for each of those.

35.2. OpenAPI client

On the client side Citrus uses the OpenAPI specification to generate a proper HTTP request that is sent to the server. The user just gives a valid operationId from the specification every thing else is automatically generated. The Citrus client message will use the proper request path (e.g. `/petstore/v3/pet`) and Content-Type (e.g. `applicaiton/json`) according to the specification rules.

Of course, you can also validate the HTTP response message with auto generated validation. The user just gives the expected HTTP status code that is also described in the specification (e.g. 200 OK). The response data used as expected message content is then also generated from the specification.

As an example the following OpenAPI specification defines the operation `getPetById`.

petstore-v3.yaml

```yaml
openapi: 3.0.2
info:
  title: Petstore
  version: 1.0.1
servers:
  - url: 'http://localhost/petstore/v3/'
paths:
  '/pet/{petId}':
    get:
      operationId: getPetById
      parameters:
        - name: petId
          description: ID of pet to return
          schema:
            format: int64
            type: integer
          in: path
          required: true
      responses:
        '200':
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Pet'
        '404':
          description: Pet not found
      summary: Find pet by ID
      description: Returns a single pet
# ...
```

The operation defines the HTTP GET request on `/pet/{petId}` and the response `200` OK that delivers the `#/components/schemas/Pet` Json object to the calling client as a response.

The Json schema for the pet defines all properties on the object.

Pet Json schema

```
Pet:
  required:
    - category
    - name
    - status
  type: object
  properties:
    id:
      format: int64
      type: integer
    category:
      $ref: '#/components/schemas/Category'
    name:
      type: string
      example: doggie
    photoUrls:
      type: array
      items:
        type: string
    tags:
      type: array
      items:
        $ref: '#/components/schemas/Tag'
    status:
      description: pet status in the store
      enum:
        - available
        - pending
        - sold
      type: string
# ...
```

In a testcase Citrus is able to leverage this information in order to send a proper request and validate the response based on the OpenAPI specification.

Java

```
private final HttpClient httpClient = new HttpClientBuilder()
            .requestUrl("http://localhost:%d".formatted(port))
            .build();

private final OpenApiSpecification petstoreSpec = OpenApiSpecification.from(
            Resources.create("classpath:org/citrusframework/openapi/petstore/petstore-
v3.json"));

@CitrusTest
public void openApiClientTest() {
    when(openapi(petstoreSpec)
                .client(httpClient)
                .send("getPetById"));

    then(openapi(petstoreSpec)
                .client(httpClient)
                .receive("getPetById", HttpStatus.OK));
}
```

XML

```
<test name="OpenApiClientTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <variables>
        <variable name="petstoreSpec"
value="classpath:org/citrusframework/openapi/petstore/petstore-v3.json"/>
    </variables>
    <actions>
        <openapi specification="${petstoreSpec}" client="httpClient">
          <send-request operation="getPetById"/>
        </openapi>

        <openapi specification="${petstoreSpec}" client="httpClient">
          <receive-response operation="getPetById" status="200"/>
        </openapi>
    </actions>
</test>
```

YAML

```
name: OpenApiClientTest
variables:
  - name: petstoreSpec
    value: classpath:org/citrusframework/openapi/petstore/petstore-v3.yaml
actions:
  - openapi:
      specification: ${petstoreSpec}
      client: "httpClient"
      sendRequest:
        operation: getPetById
  - openapi:
      specification: ${petstoreSpec}
      client: "httpClient"
      receiveResponse:
        operation: getPetById
        status: 200
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

In this very first example The client uses the OpenAPI specification to generate a proper GET HTTP request for the `getPetById` operation. The request is sent to the server using the request URL path `/petstore/v3/pet/${petId}` as declared in the OpenAPI specification.

The resulting HTTP response from the server is verified on the client by giving the operationId and the expected status `200`. The OpenAPI client generates the expected control message from the given Json schema in the OpenAPI specification.

The generated control message contains validation matchers and expressions as follows.

Generated control message body

```
{
  "id": "@isNumber()@",
  "name": "@notEmpty()@",
  "category": {
    "id": "@isNumber()@",
    "name": "@notEmpty()@"
  },
  "photoUrls": "@notEmpty()@",
  "tags":  "@ignore@",
  "status": "@matches(sold|pending|available)@"
}
```

This control message meets the rules defined by the OpenAPI Json schema specification for the pet object. For instance the enum field `status` is validated with a matching expression. In case the OpenAPI specification changes the generated control message will change accordingly.

This completes the client side OpenAPI support. Now let's have a closer look at the server side OpenAPI support in the next section.

35.3. OpenAPI server

On the server side Citrus is able to verify incoming requests based on the OpenAPI specification. The expected request message content as well as the expected resource URL path and the Content-Type are automatically validated.

Java

```
private final HttpServer httpServer = new HttpServerBuilder()
            .port(port)
            .timeout(5000L)
            .autoStart(true)
            .defaultStatus(HttpStatus.NO_CONTENT)
            .build();

private final OpenApiSpecification petstoreSpec = OpenApiSpecification.from(
            Resources.create("classpath:org/citrusframework/openapi/petstore/petstore-
v3.json"));

@CitrusTest
public void openApiClientTest() {
    when(openapi(petstoreSpec)
                .server(httpServer)
                .receive("addPet"));

    then(openapi(petstoreSpec)
                .server(httpServer)
                .send("addPet", HttpStatus.CREATED));
}
```

XML

```
<test name="OpenApiClientTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <variables>
        <variable name="petstoreSpec"
value="classpath:org/citrusframework/openapi/petstore/petstore-v3.json"/>
    </variables>
    <actions>
        <openapi specification="${petstoreSpec}" server="httpServer">
          <receive-request operation="addPet"/>
        </openapi>

        <openapi specification="${petstoreSpec}" server="httpServer">
          <send-response operation="addPet" status="200"/>
        </openapi>
    </actions>
</test>
```

YAML

```
name: OpenApiClientTest
variables:
  - name: petstoreSpec
    value: classpath:org/citrusframework/openapi/petstore/petstore-v3.yaml
actions:
  - openapi:
      specification: ${petstoreSpec}
      server: "httpServer"
      receiveRequest:
        operation: addPet
  - openapi:
      specification: ${petstoreSpec}
      server: "httpServer"
      sendResponse:
        operation: addPet
        status: 200
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

The example above uses the `addPet` operation defined in the OpenAPI specification. The operation expects a HTTP POST request with a pet object as message payload. The OpenAPI server generates an expected Json message body according to the specification. This ensures that the incoming client request meets the Json schema rules for the pet object. Also, the server will verify the HTTP request method, the Content-Type header as well as the used resource path `/petstore/v3/pet`.

The given HTTP status code defines the response that should be sent by the server. The server will generate a proper response according to the OpenAPI specification. This also includes a potential response message body (e.g. pet object).

Chapter 36. JBang support

With JBang you can run your `.java` files directly in your shell - instantly without tedious setup.

The JBang support in Citrus gets enabled by adding a separate Maven module as a dependency to your project

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-jbang-connector</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

You may wonder why is there another module named `citrus-jbang`! What is the difference compared to the `citrus-jbang-connector` module? The `citrus-jbang` module represents the Citrus JBang app that you can use in JBang to run Citrus tests without any prior setup. The `citrus-jbang-connector` module provides the JBang support to use JBang as part of a test case, for instance in the form of a test action that runs JBang scripts.

36.1. JBang action

The JBang test action runs a script or JBang application with a spawned process. You can call any JBang CLI command and run your JBang app. JBang is called with the Java process API so the JBang CLI binary is executed and the command output is saved for later reference. You can verify the command output with an expected output and you can also verify the exit code of the spawned process.

Java

```
@CitrusTest
public void jBangScriptTest() {
    when(jbang()
            .app("myApp")
            .command("getUsers")
            .arg("--username", "FooUser"));
}
```

XML

```
<test name="JBangScriptTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <jbang app="myApp" command="getUsers">
          <args>
            <arg name="--username" value="FooUser"/>
          </args>
        </jbang>
    </actions>
</test>
```

YAML

```
name: JBangScriptTest
actions:
  - jbang:
      app: "myApp"
      command: "getUsers"
      args:
        - name: "--username"
          value: "FooUser"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

The test above calls the JBang app `myApp` with the `getUsers` command. The call is similar to a command line statement like this:

```
$ jbang myApp getUsers --username=FooUser
```

The app `myApp` represents a JBang application that has been installed previously in JBang (for instance from a GitHub repository that holds a `jbang-catalog.json`):

```
$ jbang app install myApp@github-user/repository
```

Chapter 37. Kubernetes support

Kubernetes is one of the hottest management platforms for containerized applications these days. Kubernetes lets you deploy, scale and manage your Docker containers on the platform. You get features like auto-scaling, self-healing, service discovery and load balancing.

Citrus provides interaction with the Kubernetes REST API to access the Kubernetes cluster and its resources within a Citrus test case.

> The Kubernetes test components in Citrus are kept in a separate Maven module. If not already done, you have to include the module as Maven dependency to your project

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-kubernetes</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

37.1. Kubernetes client

Citrus operates with the Kubernetes remote REST API in order to interact with the Kubernetes platform. The Kubernetes client component holds the configuration to connect to a Kubernetes cluster:

Citrus Bean

```
@BindToRegistry
public KubernetesClient k8sClient() {
    return new KubernetesClientBuilder()
                .build();
}
```

Spring Bean

```
@Bean
public KubernetesClient k8sClient() {
    return new KubernetesClientBuilder()
                .build();
}
```

Spring XML

```
<citrus-k8s:client id="k8sClient"/>
```

The Kubernetes client is based on the Fabric8 Java Kubernetes client implementation. Following

from that the component can be configured in various ways. By default, the client reads the system properties as well as environment variables for default Kubernetes settings such as:

- **kubernetes.master / KUBERNETES_MASTER**
- **kubernetes.api.version / KUBERNETES_API_VERSION**
- **kubernetes.trust.certificates / KUBERNETES_TRUST_CERTIFICATES**

If you set these properties in your environment the client component will automatically pick up the configuration settings. Also, when using `kubectl` command line locally the client may automatically use the stored user authentication settings from there.

For a complete list of settings and explanation of those please refer to the Fabric8 client documentation.

In case you need to set the client configuration explicitly on your environment you can also use explicit settings on the Kubernetes client component:

Citrus Bean

```
@BindToRegistry
public KubernetesClient k8sClient() {
    return new KubernetesClientBuilder()
                .client(fabric8Client)
                .url("http://localhost:8843")
                .version("v1")
                .username("user")
                .password("s!cr!t")
                .namespace("user_namespace")
                .messageConverter(messageConverter)
                .objectMapper("objectMapper")
                .build();
}
```

Spring Bean

```
@Bean
public KubernetesClient k8sClient() {
    return new KubernetesClientBuilder()
                .client(fabric8Client)
                .url("http://localhost:8843")
                .version("v1")
                .username("user")
                .password("s!cr!t")
                .namespace("user_namespace")
                .messageConverter(messageConverter)
                .objectMapper("objectMapper")
                .build();
}
```

Spring XML

```
<citrus-k8s:client id="k8sClient"
              client="fabric8Client"
              url="http://localhost:8843"
              version="v1"
              username="user"
              password="s!cr!t"
              namespace="user_namespace"
              message-converter="messageConverter"
              object-mapper="objectMapper"/>
```

You have the option to set an `oauthToken` for the client instead of using `username` and `password`.

Now Citrus is able to access the Kubernetes remote API for executing commands such as list-pods, watch-services and so on. Citrus provides a set of actions that perform a Kubernetes command via REST. The results usually get validated in the Citrus test as usual.

37.2. Kubernetes actions

Citrus provides a set of test actions specifically designed to manage resources on the Kubernetes cluster. Each test action requires a reference to the Kubernetes client. Usually this client is created once in the bean registry and used by many test actions.

All resources created on the Kubernetes cluster will be automatically removed by Citrus after the test. You can disable this default behavior via system property setting or environment variables (`citrus.kubernetes.auto.remove.resources=false` or `CITRUS_KUBERNETES_AUTO_REMOVE_RESOURCES=false`)

37.3. Create Labels

Resources in Kubernetes are able to expose labels as part of the resource metadata. You can set labels on existing Kubernetes resources with this test action:

Java

```
@CitrusTest
public void createLabelsTest() {
    given(kubernetes()
            .client(k8sClient)
            .pods()
            .addLabel("my-pod")
            .label("test", "citrus")
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="CreateLabelsTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
      <kubernetes client="k8sClient" namespace="my-namespace">
        <create-labels type="POD" resource="my-pod">
          <labels>
            <label name="test" value="citrus"/>
          </labels>
        </create-labels>
      </kubernetes>
    </actions>
</test>
```

YAML

```
name: CreateLabelsTest
actions:
  - kubernetes:
      client: "k8sClient"
      namespace: "my-namespace"
      createLabels:
        type: "POD"
        resource: "my-pod"
        labels:
          - name: "test"
            value: "citrus"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

37.4. Create Annotations

Resources in Kubernetes are able to expose annotations as part of the resource metadata. You can set annotations on existing Kubernetes resources with this test action:

Java

```
@CitrusTest
public void createAnnotationsTest() {
    given(kubernetes()
            .client(k8sClient)
            .pods()
            .addAnnotation("my-pod")
            .annotation("test", "citrus")
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="CreateAnnotationsTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
      <kubernetes client="k8sClient" namespace="my-namespace">
        <create-annotations type="POD" resource="my-pod">
          <annotations>
            <annotation name="test" value="citrus"/>
          </annotations>
        </create-annotations>
      </kubernetes>
    </actions>
</test>
```

YAML

```
name: CreateAnnotationsTest
actions:
  - kubernetes:
      client: "k8sClient"
      namespace: "my-namespace"
      createAnnotations:
        type: "POD"
        resource: "my-pod"
        labels:
          - name: "test"
            value: "citrus"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

37.5. Create Service

You can create a Kubernetes service resource with this test action:

Java

```
@CitrusTest
public void createServiceTest() {
    given(kubernetes()
            .client(k8sClient)
            .services()
            .create("my-service")
            .portMapping(80, 8080)
            .withPodSelector(Collections.singletonMap("test", "citrus"))
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="CreateServiceTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
      <kubernetes client="k8sClient" namespace="my-namespace">
        <create-service name="my-service">
          <ports>
            <port-mapping port="80" target-port="8080"/>
          </ports>
          <selector>
            <label name="test" value="citrus"/>
          </selector>
        </create-service>
      </kubernetes>
    </actions>
</test>
```

YAML

```
name: CreateServiceTest
actions:
  - kubernetes:
      client: "k8sClient"
      namespace: "my-namespace"
      createService:
        name: "my-service"
        ports:
          - port: "80"
            targetPort: "8080"
        selector:
          labels:
            - name: "test"
              value: "citrus"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

The service is able to define port mappings as well as pod selectors based on labels. This exposes the service as a Kubernetes resource on the connected cluster in the given namespace.

> The Kubernetes service test action will automatically bind this service to a local HttpServer instance on the given port mapping. As a result the HttpServer instance is created and started as part of the test action. The HttpServer instance is added to the bean registry for further reference. The default HttpServer bean name is `citrus-k8s-service` or the service name chosen for the Kubernetes service. You can reference the HttpServer in following test actions to receive requests from the Kubernetes service.

> The behavior to auto create the HttpServer instance can be disabled via system property or environment variable setting (`citrus.kubernetes.auto.create.server.binding=false` or `CITRUS_KUBERNETES_AUTO_CREATE_SERVER_BINDING=false`) You can also customize the HttpServer instance by referencing a previously created server from the bean registry. In this case the auto creation is skipped for this test action and the custom server reference is used.

37.6. Connect with Service

Every Kubernetes Deployment is able to expose a service to other resources on the cluster so clients are able to call the service endpoint. Usually Kubernetes service exposes this endpoint to internal

clients only (e.g. service type is ClusterIP). Internal client means that the client connects from within the same cluster (e.g. from another pod). Following from that your test that runs on your local machine may not have access to the Kubernetes service endpoint.

The solution for that is to connect to the Kubernetes service as part of the Citrus test. This enables the access to such a Kubernetes service. The service connection uses a port forward to the Kubernetes service and the underlying Kubernetes pod. You can then use a local port on your machine to access the Kubernetes service.

Java

```
@CitrusTest
public void connectServiceTest() {
    given(kubernetes()
            .client(k8sClient)
            .services()
            .connect("my-service")
            .portMapping(8080, 31234)
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="ServiceConnectTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
  <actions>
    <kubernetes client="k8sClient" namespace="my-namespace">
      <connect>
        <service name="my-service" port="8080" local-port="31234"/>
      </connect>
    </kubernetes>
  </actions>
</test>
```

YAML

```
name: ServiceConnectTest
actions:
  - kubernetes:
      client: "k8sClient"
      namespace: "my-namespace"
      connect:
        service:
          name: "my-service"
          port: 8080
          localPort: 31234
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

The service connect test action creates a new port forward using a local port on your machine connecting to the exposed port on the service. The user is able to define the port mapping with a fixed local port and the target exposed port on the Kubernetes service.

> You can also skip the local port setting to choose a random local port.

This opens the connection of your local machine to the Kubernetes service so sending requests to the local endpoint will invoke the service on the Kubernetes cluster. The port forward is kept open for the rest of your test or until you explicitly disconnect.

The connect test action automatically creates a new Http client instance and stores this in the Citrus registry. You can use the client in following test actions to connect to the new port forward service endpoint. As usual, you can reference the client by its name in the Citrus registry. The name of the Http client instance that connects to the new port forward service uses a default pattern: `<service-name>.client`.

In our example the client is stored in the Citrus registry with the name `my-service.client`. You can use the client in a normal Http test action to call the new port forward service.

Java

```
@CitrusTest
public void connectServiceTest() {
    when(http()
        .client("my-service.client")
        .send()
        .get()
    );

    then(http()
        .client("my-service.client")
        .receive()
        .response(HttpStatus.OK)
    );
}
```

XML

```
<test name="ServiceConnectTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
  <actions>
    <http client="my-service.client">
      <send-request>
        <GET/>
      </send-request>
    </http>

    <http client="my-service.client">
      <receive-response>
        <response status="200" reason-phrase="OK"/>
      </receive-response>
    </http>
  </actions>
</test>
```

YAML

```
name: ServiceConnectTest
actions:
  - http:
      client: "my-service.client"
      sendRequest:
        GET: {}
  - http:
      client: "httpClient"
      receiveResponse:
        status: 200
        reasonPhrase: OK
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

Instead of using the default generated Http client name you can also explicitly specify a client name. Citrus will search create the client and add it to the Citrus registry with that name.

The Kubernetes service port forward may require special permissions on your Kubernetes cluster. Please make sure to connect with a proper user that is able to create the port forward. On Kubernetes platforms such as Minikube or Kind this is given by default. On other cluster types you may need to set up special permissions for your local Kubernetes user.

The port forward connection is automatically closed when the test ends. This is done with an explicit disconnect test action that is run in the finally section of the current test. In case you need to keep the port forward connection open you can disable the `auto-remove` option on the connect test action.

37.7. Disconnect from Service

By default, the port forward service connection is closed automatically when the test ends. In case you need to explicitly close the port forward connection you can do so with the following test action.

Java

```
@CitrusTest
public void connectServiceTest() {
    given(kubernetes()
            .client(k8sClient)
            .services()
            .disconnect("my-service")
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="ServiceConnectTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
  <actions>
    <kubernetes client="k8sClient" namespace="my-namespace">
      <disconnect>
        <service name="my-service"/>
      </disconnect>
    </kubernetes>
  </actions>
</test>
```

YAML

```
name: ServiceConnectTest
actions:
  - kubernetes:
      client: "k8sClient"
      namespace: "my-namespace"
      disconnect:
        service:
          name: "my-service"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

This closes the port forward to the Kubernetes service immediately.

37.8. Create ConfigMap

You can create a Kubernetes config map with this test action:

Java

```
@CitrusTest
public void createConfigMapTest() {
    given(kubernetes()
            .client(k8sClient)
            .configMaps()
            .create("my-config-map")
            .properties(Collections.singletonMap("username", "foo"))
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="CreateConfigMapTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
      <kubernetes client="k8sClient" namespace="my-namespace">
        <create-config-map name="my-config-map">
          <properties>
            <property name="username" value="foo"/>
          </properties>
        </create-config-map>
      </kubernetes>
    </actions>
</test>
```

YAML

```
name: CreateConfigMapTest
actions:
  - kubernetes:
      client: "k8sClient"
      namespace: "my-namespace"
      createConfigMap:
        name: "my-config-map"
        properties:
          - name: username
            value: foo
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

It is also possible to load the properties from a file resource:

Java

```
@CitrusTest
public void createConfigMapTest() {
    given(kubernetes()
            .client(k8sClient)
            .configMaps()
            .create("my-config-map")
            .fromFile(Resources.create("application.properties"))
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="CreateConfigMapTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
      <kubernetes client="k8sClient" namespace="my-namespace">
        <create-config-map name="my-config-map" file="application.properties"/>
      </kubernetes>
    </actions>
</test>
```

YAML

```
name: CreateConfigMapTest
actions:
  - kubernetes:
      client: "k8sClient"
      namespace: "my-namespace"
      createConfigMap:
        name: "my-config-map"
        file: "application.properties"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

37.9. Create Secret

You can create a Kubernetes secret with this test action:

Java

```
@CitrusTest
public void createSecretTest() {
    given(kubernetes()
            .client(k8sClient)
            .secrets()
            .create("my-secret")
            .properties(Collections.singletonMap("password", "top_secret"))
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="CreateSecretTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
      <kubernetes client="k8sClient" namespace="my-namespace">
        <create-secret name="my-secret">
          <properties>
            <property name="password" value="top_secret"/>
          </properties>
        </create-secret>
      </kubernetes>
    </actions>
</test>
```

YAML

```
name: CreateSecretTest
actions:
  - kubernetes:
      client: "k8sClient"
      namespace: "my-namespace"
      createSecret:
        name: "my-secret"
        properties:
          - name: password
            value: top_secret
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

It is also possible to load the properties from a file resource:

Java

```
@CitrusTest
public void createSecretTest() {
    given(kubernetes()
            .client(k8sClient)
            .secrets()
            .create("my-secret")
            .fromFile(Resources.create("credentials.properties"))
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="CreateSecretTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
      <kubernetes client="k8sClient" namespace="my-namespace">
        <create-secret name="my-secret" file="credentials.properties"/>
      </kubernetes>
    </actions>
</test>
```

YAML

```
name: CreateSecretTest
actions:
  - kubernetes:
      client: "k8sClient"
      namespace: "my-namespace"
      createSecret:
        name: "my-secret"
        file: "credentials.properties"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

37.10. Create Resources

You can create standards Kubernetes resources such as Pods as part of the test. The easiest way to define the resource specification is to give a YAML file that represents the resource.

Of course, you can use Citrus test variables in the resource specification.

pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    name: my-pod
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
```

Java

```
@CitrusTest
public void createPodResourceTest() {
    given(kubernetes()
            .client(k8sClient)
            .resources()
            .create()
            .resource(Resources.create("pod.yaml"))
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="CreatePodResourceTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
      <kubernetes client="k8sClient" namespace="my-namespace">
        <create-resource file="pod.yaml"/>
      </kubernetes>
    </actions>
</test>
```

YAML

```
name: CreatePodResourceTest
actions:
  - kubernetes:
      client: "k8sClient"
      namespace: "my-namespace"
      createResource:
        file: "pod.yaml"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

37.11. Delete Resources

You can delete any resource on the Kubernetes cluster. Of course your client connection needs to have sufficient permissions to perform the delete operation. The Kubernetes resource that should be deleted is identified by its name and an optional namespace.

The next example deletes a service resource identified by its name.

Java

```
@CitrusTest
public void deleteServiceTest() {
    given(kubernetes()
            .client(k8sClient)
            .services()
            .delete("my-service")
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="DeleteServiceTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
      <kubernetes client="k8sClient" namespace="my-namespace">
        <delete-service name="my-service"/>
      </kubernetes>
    </actions>
</test>
```

YAML

```
name: DeleteServiceTest
actions:
  - kubernetes:
      client: "k8sClient"
      namespace: "my-namespace"
      deleteService:
        name: "my-service"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

As an alternative to that you can also use a resource specification to delete the resource.

Java

```
@CitrusTest
public void deletePodResourceTest() {
    given(kubernetes()
            .client(k8sClient)
            .resources()
            .delete()
            .resource(Resources.create("pod.yaml"))
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="DeletePodResourceTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
      <kubernetes client="k8sClient" namespace="my-namespace">
        <delete-resource file="pod.yaml"/>
      </kubernetes>
    </actions>
</test>
```

YAML

```
name: DeletePodResourceTest
actions:
  - kubernetes:
      client: "k8sClient"
      namespace: "my-namespace"
      deleteResource:
        file: "pod.yaml"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

37.12. Create Custom Resource

Given the following Kubernetes custom resource:

foo.yaml

```
apiVersion: citrus.dev/v1
kind: Foo
metadata:
  name: my-foo
  labels:
    test: citrus
spec:
  message: Hello
```

You can create the Kubernetes custom resource with this test action:

Java

```
@CitrusTest
public void createCustomResourceTest() {
    given(kubernetes()
            .client(k8sClient)
            .customResources()
            .create()
            .kind("Foo")
            .apiVersion("citrus.dev/v1")
            .file("foo.yaml")
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="CreateCustomResourceTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
      <kubernetes client="k8sClient" namespace="my-namespace">
        <create-custom-resource kind="Foo" api-version="citrus.dev/v1" file="foo.yaml"
/>
      </kubernetes>
    </actions>
</test>
```

YAML

```
name: CreateCustomResourceTest
actions:
  - kubernetes:
      client: "k8sClient"
      namespace: "my-namespace"
      createCustomResource:
        kind: "Foo"
        apiVersion: "citrus.dev/v1"
        file: "foo.yaml"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

When you have access to a Fabric8 Java model that represents the custom resource you can use the Java type to specify the Kind and ApiVersion while creating the resource. Also, you can use a Java object instance directly to create the resource with the Fabric8 Kubernetes client.

37.13. Delete Custom Resource

When you want to delete a Kubernetes custom resource you need to specify the Kind as well as the ApiVersion in combination with the resource name on the cluster.

Java

```
@CitrusTest
public void createCustomResourceTest() {
    given(kubernetes()
            .client(k8sClient)
            .customResources()
            .delete()
            .kind("Foo")
            .apiVersion("citrus.dev/v1")
            .name("foo")
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="DeleteCustomResourceTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
      <kubernetes client="k8sClient" namespace="my-namespace">
        <delete-custom-resource kind="Foo" api-version="citrus.dev/v1" name="foo" />
      </kubernetes>
    </actions>
</test>
```

YAML

```
name: DeleteCustomResourceTest
actions:
  - kubernetes:
      client: "k8sClient"
      namespace: "my-namespace"
      deleteCustomResource:
        kind: "Foo"
        apiVersion: "citrus.dev/v1"
        name: "foo"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

When you have access to a Fabric8 Java model that represents the custom resource you can use the Java type to specify the Kind and ApiVersion while deleting the resource.

37.14. Verify Custom Resource

You can verify the ready condition status for a Kubernetes custom resource. Usually the custom resource exposes a Status sub-resource with that expose a ready condition. You can verify this condition as part of a test like this:

Java

```
@CitrusTest
public void verifyCustomResourceTest() {
    given(kubernetes()
            .client(k8sClient)
            .customResources()
            .verify("foo")
            .kind("Foo")
            .apiVersion("citrus.dev/v1")
            .isReady()
            .maxAttempts(2)
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="VerifyCustomResourceTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
      <kubernetes client="k8sClient" namespace="my-namespace">
        <verify-custom-resource kind="Foo" api-version="citrus.dev/v1" name="foo"
condition="Ready" />
      </kubernetes>
    </actions>
</test>
```

YAML

```
name: VerifyCustomResourceTest
actions:
  - kubernetes:
      client: "k8sClient"
      namespace: "my-namespace"
      verifyCustomResource:
        kind: "Foo"
        apiVersion: "citrus.dev/v1"
        name: "foo"
        condition: "Ready"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

Instead of specifying the resource name you can also use a label expression (name=value) to identify the custom resource on the cluster namespace. Citrus tries to find the resource with the matching label.

37.15. Verify Pod Status

Each Pod in Kubernetes exposes its Ready state condition as well as the phase (Running, Error, Initializing, Terminating, ...). You can verify this Pod status in your test case:

Java

```
@CitrusTest
public void verifyPodTest() {
    given(kubernetes()
            .client(k8sClient)
            .pods()
            .verify("my-pod")
            .isRunning()
            .maxAttempts(2)
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="VerifyPodTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
      <kubernetes client="k8sClient" namespace="my-namespace">
        <verify-pod name="my-pod" phase="Running" />
      </kubernetes>
    </actions>
</test>
```

YAML

```
name: VerifyPodTest
actions:
  - kubernetes:
      client: "k8sClient"
      namespace: "my-namespace"
      verifyPod:
        name: "my-pod"
        phase: "Running"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

When asserting the `Running` phase the verification also checks for all containers in the Pod to expose the condition `Ready`.

37.16. Verify Pod Logs

You can read and verify the Pod logs as part of the test.

Java

```
@CitrusTest
public void verifyPodTest() {
    given(kubernetes()
            .client(k8sClient)
            .pods()
            .verify("my-pod")
            .isRunning()
            .waitForLogMessage("Something interesting")
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="VerifyPodTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
      <kubernetes client="k8sClient" namespace="my-namespace">
        <verify-pod name="my-pod" phase="Running" log-message="Something
interesting"/>
      </kubernetes>
    </actions>
</test>
```

YAML

```
name: VerifyPodTest
actions:
  - kubernetes:
      client: "k8sClient"
      namespace: "my-namespace"
      verifyPod:
        name: "my-pod"
        phase: "Running"
        logMessage: "Something interesting"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

This action scans the Pod logs for a message `Something interesting`. As soon as the log message has been located in the Pod logs the test action finishes with success. The action waits for the log message to appear in the Pod logs for given amount of time with a specified polling attempt configuration.

37.17. Kubernetes Citrus agent

The Citrus agent is a way to run tests as part of the Kubernetes platform. You can deploy the Citrus agent on the Kubernetes cluster. Usually the agent exposes a REST service endpoint so clients are then able to inject Citrus tests for execution.

The Citrus agent runs as part of the Kubernetes cluster, so it has access to Kubernetes resources on the same namespace. This means the tests that run with the agent are able to access resources such as Knative eventing brokers, Services, ConfigMaps, Secrets and many more.

37.17.1. Agent connect

You can connect to a Citrus agent service as part of your test like follows:

Java

```
@CitrusTest
public void agentConnectTest() {
    given(kubernetes()
            .client(k8sClient)
            .agent()
            .connect()
            .service("citrus-agent")
            .localPort(12345)
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="AgentConnectTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
      <kubernetes client="k8sClient" namespace="my-namespace">
        <agent>
          <connect name="citrus-agent" local-port="12345"/>
        </agent>
      </kubernetes>
    </actions>
</test>
```

YAML

```
name: AgentConnectTest
actions:
  - kubernetes:
      client: "k8sClient"
      namespace: "my-namespace"
      agent:
        connect:
          name: "citrus-agent"
          localPort: 12345
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

The agent connect test action connects to the Citrus agent service on the given Kubernetes namespace. In case the Citrus agent application has been deployed already and the exposed service is available the test action connects to the service via local port forwarding (see also the section about connecting with a Kubernetes Service). This means that you will be able to trigger test runs

on the agent with calling REST operations on the local endpoint.

In case the Citrus agent application is not present on the Kubernetes cluster the connect test action will automatically create the deployment on the given namespace. The deployment includes a ConfigMap that holds the test case sources (e.g. in the form of a `test-jar` Java archive). The agent deployment specification will mount the test sources into the agent Pod via volume mounts, so you can run the tests from the `test-jar` Java archive.

Also, the action creates a new Kubernetes service that exposes the agent REST API and the Web UI. Once the connection is done you can start triggering test actions and test runs on the Citrus agent.

37.17.2. Agent run

When the local test is connected to the Kubernetes Citrus agent you can trigger the test execution on the agent.

Java

```
@CitrusTest
public void agentRunTest() {
    given(agent()
            .run()
            .sourceFile("classpath:my-test.groovy"));
}
```

XML

```
<test name="AgentRunTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
      <agent>
        <run>
          <source file="classpath:my-test.groovy"/>
        </run>
      </agent>
    </actions>
</test>
```

YAML

```
name: AgentRunTest
actions:
  - agent:
      source:
        file: "classpath:my-test.groovy"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

The test source code can be any of the supported domain specific languages (e.g. XML, YAML, Java, Groovy, Cucumber). The example above loads a Groovy test source code file. The test actions will be sent to the Citrus agent and executed as a test within the agent runtime. The test results are validated for success outcome and in case the remote test execution fails the local Citrus agent run action will also fail accordingly.

When using XML and YAML tests you can also leverage the possibility to use inline test actions as part of the Citrus agent run action

YAML

```
name: AgentRunTest
actions:
  - agent:
      run:
        actions:
          - echo:
              message: Citrus rocks!
```

XML

```
<test name="AgentRunTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
      <agent>
        <run>
          <actions>
            <echo message="Citrus rocks!"/>
          </actions>
        </run>
      </agent>
    </actions>
</test>
```

All nested test actions within the Citrus agent run action will be run on the connected remote Citrus agent.

37.17.3. Agent disconnect

You can explicitly disconnect from the Citrus agent service. This terminates the local port forward and also removes the Citrus agent deployment and all dependent resources from the current Kubernetes namespace.

Java

```
@CitrusTest
public void agentDisconnectTest() {
    given(kubernetes()
            .client(k8sClient)
            .agent()
            .disconnect()
            .service("citrus-agent")
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="AgentDisconnectTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
      <kubernetes client="k8sClient" namespace="my-namespace">
        <agent>
          <disconnect name="citrus-agent"/>
        </agent>
      </kubernetes>
    </actions>
</test>
```

YAML

```
name: AgentDisconnectTest
actions:
  - kubernetes:
      client: "k8sClient"
      namespace: "my-namespace"
      agent:
        disconnect:
          name: "citrus-agent"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

37.18. Spring bean config

Citrus provides a "citrus-kubernetes" configuration namespace and schema definition for Kubernetes related components and actions. Include this namespace into your Spring configuration in order to use the Citrus Kubernetes configuration elements.

37.18.1. Spring XML config namespace

The namespace URI and schema location are added to the Spring configuration XML file as follows.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:citrus-k8s="http://www.citrusframework.org/schema/kubernetes/config"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.citrusframework.org/schema/kubernetes/config
       http://www.citrusframework.org/schema/kubernetes/config/citrus-kubernetes-
config.xsd">

    [...]

</beans>
```

After that you are able to use customized Citrus XML elements in order to define the Spring beans.

37.18.2. Spring XML testcase namespace

We have several Citrus test actions each representing a Kubernetes command. These actions can be part of a test case where you can manage Kubernetes pods inside the test. As a prerequisite we have to enable the Kubernetes specific test actions in our XML test as follows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:k8s="http://www.citrusframework.org/schema/kubernetes/testcase"
        xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.citrusframework.org/schema/kubernetes/testcase
        http://www.citrusframework.org/schema/kubernetes/testcase/citrus-kubernetes-
testcase.xsd">

    [...]

</beans>
```

We added a special kubernetes namespace with prefix **k8s:** so now we can start to add Kubernetes test actions to the test case:

Spring XML

```
<testcase name="KubernetesCommandIT">
    <actions>
      <k8s:info client="myK8sClient">
        <k8s:validate>
          <k8s:result>{
            "result": {
              "clientVersion": "1.4.27",
              "apiVersion": "v1",
              "kind":"Info",
              "masterUrl": "${masterUrl}",
              "namespace": "test"
            }
          }</k8s:result>
        </k8s:validate>
      </k8s:info>
    </actions>
</testcase>
```

In this very simple example we first ping the Kubernetes REST API to make sure we have connectivity up and running. The info command connects the REST API and returns a list of status information of the Kubernetes client.

Based on that we can execute several Kubernetes commands in a test case and validate the Json results.

Citrus supports the following Kubernetes API commands with respective test actions:

- **k8s:info**
- **k8s:list-pods**
- **k8s:get-pod**
- **k8s:delete-pod**
- **k8s:list-services**
- **k8s:get-service**
- **k8s:delete-service**
- **k8s:list-namespaces**
- **k8s:list-events**
- **k8s:list-endpoints**
- **k8s:list-nodes**
- **k8s:list-replication-controllers**
- **k8s:watch-pods**
- **k8s:watch-services**
- **k8s:watch-namespaces**

- **k8s:watch-nodes**
- **k8s:watch-replication-controllers**

We will discuss these commands in detail in the following sections.

37.19. Kubernetes commands

The Citrus test may use Kubernetes commands that get run as test actions.

Java

```
@CitrusTest
public void kubernetesTest() {
    kubernetes().info()
        .validate(new CommandResultCallback<InfoResult>() {
            @Override
            public void doWithCommandResult(InfoResult info, TestContext context) {
                Assert.assertEquals(info.getApiVersion(), "v1");
            }
    });

    kubernetes().pods()
                .list()
                .withoutLabel("running")
                .label("app", "myApp");
}
```

Spring XML

```
<testcase name="KubernetesCommandIT">
    <actions>
      <k8s:info>
        <k8s:validate>
          <k8s:result>{
            "result": {
              "clientVersion": "1.4.27",
              "apiVersion": "v1",
              "kind":"Info",
              "masterUrl": "${masterUrl}",
              "namespace": "test"
            }
          }</k8s:result>
        </k8s:validate>
      </k8s:info>

      <k8s:list-pods>
        <k8s:validate>
          <k8s:result>{
            "result": {
              "apiVersion":"v1",
              "kind":"PodList",
              "metadata":"@ignore@",
              "items":[]
            }
          }</k8s:result>
          <k8s:element path="$.result.items.size()" value="0"/>
        </k8s:validate>
      </k8s:list-pods>
    </actions>
</testcase>
```

In this very simple example we first ping the Kubernetes REST API to make sure we have connectivity up and running. The info command connects the REST API and returns a list of status information of the Kubernetes client. After that we get the list of available Kubernetes pods.

As a tester we might be interested in validating the command results. So we can specify an optional **k8s:result** which is usually in Json format. With that we can apply the full Citrus Json validation power to the Kubernetes results. As usual, we can use test variables here and ignore some values explicitly such as the **metadata** value. Also, JsonPath expression validation and Json test message validation features in Citrus come in here to validate the results.

The Java DSL Kubernetes commands provide an optional **CommandResultCallback** that is automatically called with the unmarshalled command result object. In the example above the *InfoResult* model object is passed as argument to the callback. So the tester can access the command result and validate its properties with assertions.

The option of Lamdba expressions allows us to use some syntactical sugar in Java when using the

CommandResultCallback.

Java

```
@CitrusTest
public void kubernetesTest() {
    kubernetes().info()
                .validate((info, context) -> Assert.assertEquals(info.getApiVersion(),
"v1"));
}
```

By default, Citrus tries to find a Kubernetes client component within the Citrus Spring application context. If not present Citrus will instantiate a default kubernetes client with all default settings. You can also explicitly set the kubernetes client instance when using the Java DSL Kubernetes command actions:

Java

```
@Autowired
private KubernetesClient k8sClient;

@CitrusTest
public void kubernetesTest() {
    kubernetes().client(k8sClient)
                .info()
                .validate((info, context) -> Assert.assertEquals(info.getApiVersion(),
"v1"));
}
```

Spring XML

```
<testcase name="KubernetesCommandIT">
    <actions>
      <k8s:info client="k8sClient">
        <k8s:validate>
          <k8s:result>{
            "result": {
              "clientVersion": "1.4.27",
              "apiVersion": "v1",
              "kind":"Info",
              "masterUrl": "${masterUrl}",
              "namespace": "test"
            }
          }</k8s:result>
        </k8s:validate>
      </k8s:info>
    </actions>
</testcase>
```

37.19.1. Info command

The info command just gets the client connection settings and provides them as a Json result to the action.

Java

```
@CitrusTest
public void infoTest() {
    kubernetes().info()
                .validate((info, context) -> Assert.assertEquals(info.getApiVersion(),
"v1"));
}
```

Spring XML

```
<k8s:info client="myK8sClient">
  <k8s:validate>
    <k8s:result>{
      "result": {
        "clientVersion": "1.4.27",
        "apiVersion": "v1",
        "kind":"Info",
        "masterUrl": "${masterUrl}",
        "namespace": "test"
      }
    }</k8s:result>
  </k8s:validate>
</k8s:info>
```

37.19.2. List resources

We can list Kubernetes resources such as pods, services, endpoints and replication controllers. The list can be filtered by several properties such as

- label
- namespace

The test action is able to define respective filters to the list, so we get only pods the match the given attributes:

Java

```
@CitrusTest
public void listPodsTest() {
    kubernetes()
        .client(k8sClient)
        .pods()
        .list()
        .label("app=todo")
        .validate("$..status.phase", "Running")
        .validate((pods, context) -> {
            Assert.assertFalse(CollectionUtils.isEmpty(pods.getResult().getItems()));
        });
}
```

Spring XML

```
<k8s:list-pods label="app=todo">
    <k8s:validate>
      <k8s:result>{
        "result": {
          "apiVersion":"${apiVersion}",
          "kind":"PodList",
          "metadata":"@ignore@",
          "items":"@ignore@"
        }
      }</k8s:result>
      <k8s:element path="$.result.items.size()" value="1"/>
      <k8s:element path="$..status.phase" value="Running"/>
    </k8s:validate>
</k8s:list-pods>
```

As you can see we are able to give the pod label that is searched for in list of all pods. The list returned is validated either by giving an expected Json message or by adding JsonPath expressions with expected values to check.

In Java DSL we can add a validation result callback that is provided with the unmarshalled result object for validation. Besides *label* filtering we can also specify the *namespace* and the pod *name* to search for.

You can also define multiple labels as comma delimited list:

Java

```
@CitrusTest
public void listServicesTest() {
    kubernetes()
        .client(k8sClient)
        .services()
        .list()
        .label("stage!=test,provider=fabric8")
        .namespace("default");
}
```

Spring XML

```
<k8s:list-services label="stage!=test,provider=fabric8" namespace="default"/>
```

As you can see we have combined to label filters `stage!=test` and `provider=fabric8` on pods in namespace `default`. The first label filter is negated so the label `stage` should **not** be `test` here.

37.19.3. List nodes and namespaces

Nodes and namespaces are special resources that are not filtered by their namespace as they are more global resources. The rest is pretty similar to listing pods or services. We can add filters such as `name` and `label`.

Java

```
@CitrusTest
public void listPodsTest() {
    kubernetes()
        .client(k8sClient)
        .namespaces()
        .list()
        .label("provider=citrus")
        .validate((pods, context) -> {
            Assert.assertFalse(CollectionUtils.isEmpty(pods.getResult().getItems()));
        });
}
```

Spring XML

```
<k8s:list-namespaces label="provider=citrus">
    <k8s:validate>
      <k8s:element path="$.result.items.size()" value="1"/>
    </k8s:validate>
</k8s:list-namespaces>
```

37.19.4. Get resources

We can get a very special Kubernetes resource such as a pod or service for detailed validation of that resource. We need to specify a resource name in order to select the resource from list of available resources in Kubernetes.

Java

```
@CitrusTest
public void getPodsTest() {
    kubernetes()
        .client(k8sClient)
        .pods()
        .get("citrus_pod")
        .validate("$..status.phase", "Running")
        .validate((pod, context) -> {
            Assert.assertEquals(pods.getResult().getStatus().getPhase(), "Running");
        });
}
```

Spring XML

```
<k8s:get-pod name="citrus_pod">
    <k8s:validate>
      <k8s:result>{
      "result": {
        "apiVersion":"${apiVersion}",
        "kind":"Pod",
        "metadata": {
            "annotations":"@ignore@",
            "creationTimestamp":"@ignore@",
            "finalizers":[],
            "generateName":"@startsWith('hello-minikube-')@",
            "labels":{
              "pod-template-hash":"@ignore@",
              "run":"hello-minikube"
            },
            "name":"${podName}",
            "namespace":"default",
            "ownerReferences":"@ignore@",
            "resourceVersion":"@ignore@",
            "selfLink":"/api/${apiVersion}/namespaces/default/pods/${podName}",
            "uid":"@ignore@"
        },
        "spec": {
          "containers": [{
            "args":[],
            "command":[],
            "env":[],
            "image":"gcr.io/google_containers/echoserver:1.4",
```

```
            "imagePullPolicy":"IfNotPresent",
            "name":"hello-minikube",
            "ports":[{
              "containerPort":8080,
              "protocol":"TCP"
            }],
            "resources":{},
            "terminationMessagePath":"/dev/termination-log",
            "volumeMounts":"@ignore@"
          }],
          "dnsPolicy":"ClusterFirst",
          "imagePullSecrets":"@ignore@",
          "nodeName":"minikube",
          "restartPolicy":"Always",
          "securityContext":"@ignore@",
          "serviceAccount":"default",
          "serviceAccountName":"default",
          "terminationGracePeriodSeconds":30,
          "volumes":"@ignore@"
        },
        "status": "@ignore@"
      }
      }</k8s:result>
      <k8s:element path="$..status.phase" value="Running"/>
    </k8s:validate>
</k8s:get-pod>
```

As you can see we are able to get the complete pod information from Kubernetes. The result is validated with Json message validator in Citrus. This means we can use *@ignore@* as well as test variables and JsonPath expressions.

37.19.5. Create resources

We can create new Kubernetes resources within a Citrus test. This is very important in case we need to set up new pods or services for the test run. You can create new resources by giving a `.yaml` file holding all information how to create the new resource. See the following sample YAML for a new pod and service:

pod.yaml

```
kind: Pod
apiVersion: v1
metadata:
  name: hello-netty-${randomId}
  namespace: default
  labels:
    server: hello-netty
spec:
  containers:
    - name: hello-netty
      image: netty
      imagePullPolicy: IfNotPresent
      ports:
        - containerPort: 8080
          protocol: TCP
  restartPolicy: Always
  terminationGracePeriodSeconds: 30
  dnsPolicy: ClusterFirst
  serviceAccountName: default
  serviceAccount: default
  nodeName: minikube
```

This YAML file specifies a new resource of kind *Pod*. We define the metadata as well as all containers that are part of this pod. The container is build from *jetty:9.3* Docker image that should be pulled automatically from Docker Hub registry. We also expose port 8080 as *containerPort* so the upcoming service configuration can provide this port to clients as Kubernetes service.

service.yaml

```
kind: Service
apiVersion: v1
metadata:
  name: hello-netty
  namespace: default
  labels:
    service: hello-netty
spec:
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
      nodePort: 31123
  selector:
    server: hello-netty
  type: NodePort
  sessionAffinity: None
```

The service resource maps the port *8080* and selects all pods with label *server=hello-netty*. This

makes the jetty container available to clients. The service type is *NodePort* which means that clients outside of Kubernetes are also able to access the service by using the port *nodePort=31123*. We can use Citrus functions such as *randomNumber* in the YAML files.

In the test case we can use these YAML files to create the resources in Kubernetes:

Java

```
@CitrusTest
public void createPodsTest() {
    kubernetes()
        .pods()
        .create(new ClassPathResource("templates/hello-netty-pod.yml"))
        .namespace("default");

    kubernetes()
        .services()
        .create(new ClassPathResource("templates/hello-service.yml"))
        .namespace("default");
}
```

Spring XML

```
<testcase name="KubernetesIT">
  <actions>
    <k8s:create-pod namespace="default">
      <k8s:template file="classpath:templates/hello-netty-pod.yml"/>
    </k8s:create-pod>

    <k8s:create-service namespace="default">
      <k8s:template file="classpath:templates/hello-service.yml"/>
    </k8s:create-service>
  </actions>
</testcase>
```

Creating new resources may take some time to finish. Kubernetes will have to pull images, build containers and start up everything. The `create` action is not waiting synchronously for all that to have happened. Therefore, we might add a list-pods action that waits for the new resources to appear.

Spring XML

```
<repeat-onerror-until-true condition="@assertThat(greaterThan(9))@" auto-sleep="1000">
  <k8s:list-pods label="server=hello-netty">
    <k8s:validate>
      <k8s:element path="$.result.items.size()" value="1"/>
      <k8s:element path="$..status.phase" value="Running"/>
    </k8s:validate>
  </k8s:list-pods>
</repeat-onerror-until-true>
```

With this repeat on error action we wait for the new *server=hello-netty* labeled pod to be in state *Running*.

37.19.6. Delete resources

With that command we are able to delete a resource in Kubernetes. Up to now deletion of pods and services is supported. We have to give a name of the resource that we want to delete.

Java

```
@CitrusTest
public void deletePodsTest() {
    kubernetes()
        .pods()
        .delete("citrus_pod")
        .validate((result, context) ->
Assert.assertTrue(result.getResult().getSuccess()));
}
```

Spring XML

```
<k8s:delete-pod name="citrus_pod">
    <k8s:validate>
      <k8s:element path="$.result.success" value="true"/>
    </k8s:validate>
</k8s:delete-pod>
```

37.19.7. Watch resources

The watch operation is still in experimental state and may face severe adjustments and improvements in near future.

When using a watch command we add a subscription to change events on a Kubernetes resources. So we can watch resources such as pods, services for future changes. Each change on that resource triggers a new watch event result that we can expect and validate.

Java

```
@CitrusTest
public void listPodsTest() {
    kubernetes()
        .pods()
        .watch()
        .label("provider=citrus")
        .validate((watchEvent, context) -> {
            Assert.assertFalse(watchEvent.hasError());
            Assert.assertEquals(((WatchEventResult) watchEvent).getAction(),
Watcher.Action.DELETED);
        });
}
```

Spring XML

```
<k8s:watch-pods label="provider=citrus">
    <k8s:validate>
      <k8s:element path="$.action" value="DELETED"/>
    </k8s:validate>
</k8s:watch-pods>
```

The watch command may be triggered several times for multiple changes on the respective Kubernetes resource. The watch action will always handle one single event result. The first event trigger is forwarded to the action validation. All further watch events on that same resource are ignored. This means that you may need multiple watch actions in your test case in case you expect multiple watch events to be triggered.

37.20. Kubernetes command bus

We have seen how to access the Kubernetes remote REST API by using special Citrus test actions in a test. As an alternative to that we can also use more generic send/receive actions in Citrus for accessing the Kubernetes API.

The next example shows this in action:

Java

```
@CitrusTest
public void kubernetesSendReceiveTest() {
    send(k8sClient)
        .message()
        .body("""
        { "command": "info" }
        """);

    receive(k8sClient)
        .message()
        .body("""
        {
          "command": "info",
          "result": {
            "clientVersion": "1.4.27",
            "apiVersion": "v1",
            "kind":"Info",
            "masterUrl": "${masterUrl}",
            "namespace": "test"
          }
        }
        """);

    send(k8sClient)
        .message()
        .body("""
        { "command": "list-pods" }
        """);

    receive(k8sClient)
        .message()
        .body("""
        {
          "command": "list-pods",
          "result": {
            "apiVersion":"v1",
            "kind":"PodList",
            "metadata":"@ignore@",
            "items":[]
          }
        }
        """)
        .validate(json().expression("$.result.items.size()", 0));
}
```

Spring XML

```
<testcase name="KubernetesSendReceiveIT">
```

```
    <actions>
      <send endpoint="k8sClient">
        <message>
          <data>
            { "command": "info" }
          </data>
        </message>
      </send>

      <receive endpoint="k8sClient">
        <message type="json">
          <data>
          {
            "command": "info",
            "result": {
                "clientVersion": "1.4.27",
                "apiVersion": "v1",
                "kind":"Info",
                "masterUrl": "${masterUrl}",
                "namespace": "test"
            }
          }
          </data>
        </message>
      </receive>

      <echo>
        <message>List all pods</message>
      </echo>

      <send endpoint="k8sClient">
        <message>
          <data>
            { "command": "list-pods" }
          </data>
        </message>
      </send>

      <receive endpoint="k8sClient">
        <message type="json">
          <data>
          {
            "command": "list-pods",
            "result": {
                  "apiVersion":"v1",
                  "kind":"PodList",
                  "metadata":"@ignore@",
                  "items":[]
            }
          }
          </data>
```

```
          <validate path="$.result.items.size()" value="0"/>
        </message>
      </receive>
    </actions>
</testcase>
```

As you can see we can use the send/receive actions to call Kubernetes API commands and receive the respective results in Json format, too. This gives us the well known Json validation mechanism in Citrus in order to validate the results from Kubernetes. This way you can load Kubernetes resources verifying its state and properties. Of course JsonPath expressions also come in here in order to validate Json elements explicitly.

Chapter 38. Knative support

Knative lets you build Serverless and Event Driven applications on the cloud. The project knows the two main components Knative Serving and Knative Eventing.

Citrus as a test framework is able to interact with Knative to manage and leverage Serving and Eventing resources on a Kubernetes cluster.

The Knative support in Citrus is enabled by adding a separate Maven module as a dependency to your project

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-knative</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

38.1. Knative client

Citrus interacts with Knative resources on the Kubernetes cluster. This means we need to connect as a client to a Kubernetes cluster as part of the test. Citrus uses the Fabric8 Knative client implementation for this connection. You can create and configure the client once and reuse the client instance in multiple test cases.

Citrus Bean

```
@BindToRegistry
public KubernetesClient kubernetesClient() {
    return new KubernetesClientBuilder().build();
}

@BindToRegistry
public KnativeClient knativeClient() {
    return kubernetesClient().adapt(KnativeClient.class);
}
```

Spring Bean

```
@Bean
public KubernetesClient kubernetesClient() {
    return new KubernetesClientBuilder().build();
}

@Bean
public KnativeClient knativeClient() {
    return kubernetesClient().adapt(KnativeClient.class);
}
```

The Fabric8 Knative client is based on the Kubernetes client implementation. You can bind the client instances as beans to the Citrus bean registry. You may then reference and inject the client to your tests using the `@CitrusResource` annotation. Spring beans may be autowired to the test case as usual with `@Autowired`.

38.2. Knative broker

Citrus is able to create Knative Eventing broker resources as part of the test.

Java

```
@CitrusTest
public void knativeBrokerTest() {
    given(knative()
            .client(knativeClient)
            .brokers()
            .create("my-broker")
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="KnativeBrokerTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <knative client="knativeClient" namespace="test">
          <create-broker name="my-broker"/>
        </knative>
    </actions>
</test>
```

YAML

```
name: KnativeBrokerTest
actions:
  - knative:
      client: "knativeClient"
      namespace: "test"
      createBroker:
        name: "my-broker"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

The test above creates a new Knative broker resource in the given namespace. In the following you can connect to the broker to produce and consume events.

> Citrus automatically removes the created Knative resources after the test. This is the default behavior that makes Citrus delete resources such as the Knative broker even when the test has finished in failure state. However, you can disable the auto removal by setting this system property or environment variable: `citrus.knative.auto.remove.resources=false` or `CITRUS_KNATIVE_AUTO_REMOVE_RESOURCES=false`

You can also disable the auto removal of the Knative broker on the test action, too:

Java

```
@CitrusTest
public void knativeBrokerTest() {
    given(knative()
            .client(knativeClient)
            .brokers()
            .create("my-broker")
            .autoRemoveResources(false)
            .inNamespace("my-namespace"));
}
```

38.3. Verify broker status

The test may also verify that a broker is running and ready on a given Kubernetes namespace.

Java

```
@CitrusTest
public void knativeBrokerTest() {
    given(knative()
            .client(knativeClient)
            .brokers()
            .verify("my-broker")
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="KnativeBrokerTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <knative client="knativeClient" namespace="test">
          <verify-broker name="my-broker"/>
        </knative>
    </actions>
</test>
```

YAML

```
name: KnativeBrokerTest
actions:
  - knative:
      client: "knativeClient"
      namespace: "test"
      verifyBroker:
        name: "my-broker"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

38.4. Send events

Once a Knative broker is available we can start to produce some events with a Citrus test. Knative uses the CloudEvent data format out of the box. The test is able to set the CloudEvent attributes as well as the event data when pushing the event to the broker.

Java

```
@CitrusTest
public void produceEventTest() {
    when(knative()
            .event()
            .send()
            .brokerUrl("http://my-cluster.svc.cluster.local/my-namespace/my-broker")
            .eventData("""
            { "message": "Hello Knative event!" }
            """));
}
```

XML

```
<test name="ProduceEventTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <knative namespace="test">
          <send-event broker="http://my-cluster.svc.cluster.local/my-namespace/my-
broker">
            <event>
              <data>{ "message": "Hello Knative event!" }</data>
            </event>
          </send-event>
        </knative>
    </actions>
</test>
```

YAML

```
name: ProduceEventTest
actions:
    - knative:
      namespace: "test"
      sendEvent:
        broker: "my-broker"
        event:
          data: |
            { "message": "Hello Knative event!" }
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

This action produces a new Knative event with some CloudEvent attributes default values set. The

Http request to the Knative broker looks like this:

```
POST http://my-cluster.svc.cluster.local/my-namespace/my-broker
Accept:text/plain, application/json, application/*+json, */*
Host:broker-ingress.knative-eventing.svc.cluster.local
Content-Type:application/json
Ce-Id:2818d613-bc75-4b25-b570-b825bbe33378
Ce-Type:org.citrusframework.event.test
Ce-Specversion:1.0
Ce-Source:citrus-test
Content-Length:54

{ "message": "Hello Knative event!" }
```

You can see the CloudEvent attributes set with `Ce-*` headers. You can customize these attributes on the send event test action.

Java

```
@CitrusTest
public void produceEventTest() {
    when(knative()
            .event()
            .send()
            .brokerUrl("http://my-cluster.svc.cluster.local/my-namespace/my-broker")
            .attribute("ce-type", "org.citrusframework.knative.event")
            .eventData("""
            { "message": "Hello Knative event!" }
            """));
}
```

XML

```
<test name="ProduceEventTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <knative namespace="test">
          <send-event broker="http://my-cluster.svc.cluster.local/my-namespace/my-
broker">
            <event>
              <ce-attributes>
                <ce-attribute name="ce-type"
value="org.citrusframework.knative.event"/>
              </ce-attributes>
              <data>{ "message": "Hello Knative event!" }</data>
            </event>
          </send-event>
        </knative>
    </actions>
</test>
```

YAML

```
name: ProduceEventTest
actions:
    - knative:
      namespace: "test"
      sendEvent:
        broker: "my-broker"
        event:
          data: |
            { "message": "Hello Knative event!" }
          attributes:
            - name: ce-type
              value: "org.citrusframework.knative.event"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

38.5. Receive events

The Knative broker dispatches events by invoking services. Usually this service is an arbitrary Kubernetes Service resource or any addressable resource on the Kubernetes cluster. The events that should be consumed by a service gets specified in a Knative Trigger resource.

This means we need to consume Knative events with a Kubernetes service resource. Citrus provides

special Kubernetes support that enables you to create a Kubernetes service resource on the cluster. Read more about it in chapter Kubernetes support.

You can create the Kubernetes service as follows:

Java

```
@CitrusTest
public void createServiceTest() {
    given(kubernetes()
            .client(k8sClient)
            .service()
            .create("my-service")
            .portMapping(80, 8080)
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="CreateServiceTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <kubernetes client="k8sClient" namespace="my-namespace">
          <create-service name="my-service">
            <ports>
              <port-mapping port="80" target-port="8080"/>
            </ports>
          </create-service>
        </kubernetes>
    </actions>
</test>
```

YAML

```
name: CreateServiceTest
actions:
  - kubernetes:
      client: "k8sClient"
      namespace: "my-namespace"
      createService:
        name: "my-service"
        ports:
          - port: "80"
            targetPort: "8080"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

This creates the Kubernetes service named `my-service` in the namespace `my-namespace`.

> The test requires a local HttServer instance to actually receive the requests on the given port mapping. Citrus automatically creates this local HttpServer instance. Please make sure to choose a free port on your host machine when defining the port mapping.

> Before creating the HttpServer instance as part of the Kubernetes test action Citrus tries to resolve the HttpServer instance from the Citrus bean registry. This means you can also predefine the HttpServer instance as a usual bean in the registry. Just give the server name as an additional attribute to the Kubernetes service create test action so Citrus knows which server instance should be resolved via the bean registry.

This creates a proper Kubernetes service and binds it to the local port where the local HttpServer instance is listening for incoming requests. Before we can start to actually consume events we need to create a Knative Trigger resource. You can create the Trigger resource as part of the test, too:

Java

```
@CitrusTest
public void createTriggerTest() {
    given(knative()
            .client(knativeClient)
            .trigger()
            .create("my-trigger")
            .broker("my-broker")
            .service("my-service")
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="CreateTriggerTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
      <knative client="knativeClient" namespace="my-namespace">
        <create-trigger name="my-trigger" broker="my-broker" service="my-service"/>
      </knative>
    </actions>
</test>
```

YAML

```
name: CreateTriggerTest
actions:
  - knative:
      client: "knativeClient"
      namespace: "my-namespace"
      createTrigger:
        name: "my-trigger"
        broker: "my-broker"
        service: "my-service"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

The Trigger resource receives the broker name that should be used as a source of events and the service name as a subject to being invoked when an event gets dispatched. The trigger name and the namespace defines where the Trigger resource is created on the Kubernetes cluster.

You add a filter to the Trigger resource to filter events on CloudEvent attributes.

Java

```
@CitrusTest
public void createTriggerTest() {
    given(knative()
            .client(knativeClient)
            .trigger()
            .create("my-trigger")
            .broker("my-broker")
            .service("my-service")
            .filter("type", "my.very.specific.event.type")
            .inNamespace("my-namespace"));
}
```

XML

```
<test name="CreateTriggerTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
      <knative client="knativeClient" namespace="my-namespace">
        <create-trigger name="my-trigger" broker="my-broker" service="my-service">
          <filter>
            <attribute name="type" value="my.very.specific.event.type"/>
          </filter>
        </create-trigger>
      </knative>
    </actions>
</test>
```

YAML

```
name: CreateTriggerTest
actions:
  - knative:
      client: "knativeClient"
      namespace: "my-namespace"
      createTrigger:
        name: "my-trigger"
        broker: "my-broker"
        service: "my-service"
        filter:
          attributes:
            - name: type
              value: "my.very.specific.event.type"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

The filter is set on the CloudeEvent type attribute so the trigger will only dispatch events with this specific type. The created Trigger resource looks like follows:

Knative Trigger

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: my-trigger
spec:
  broker: default
  filter:
    attributes:
      type: my.very.specific.event.type
  subscriber:
    ref:
      apiVersion: v1
      kind: Service
      name: my-service
```

This completes all resources that we need to consume events from the Knative eventing broker. The following test action receives and verifies a Knative event with an expected CloudEvent message holding expected attributes and event data.

Java

```
@CitrusTest
public void consumeEventTest() {
    when(knative()
            .event()
            .receive()
            .serviceName("my-service")
            .attribute("ce-type", "org.citrusframework.knative.event")
            .eventData("""
            {
                "message": "Hello Knative event!"
            }
            """));
}
```

XML

```
<test name="ConsumeEventTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <knative namespace="my-namespace">
          <receive-event service="my-service">
            <event>
              <ce-attributes>
                <ce-attribute name="ce-type"
value="org.citrusframework.knative.event"/>
              </ce-attributes>
              <data>{ "message": "Hello Knative event!" }</data>
            </event>
          </receive-event>
        </knative>
    </actions>
</test>
```

YAML

```
name: ConsumeEventTest
actions:
    - knative:
      namespace: "my-namespace"
      receiveEvent:
        service: "my-service"
        event:
          data: |
            { "message": "Hello Knative event!" }
          attributes:
            - name: ce-type
              value: "org.citrusframework.knative.event"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

38.6. Knative channels

TODO

38.7. Knative subscription

TODO

Chapter 39. Testcontainers support

Testcontainers leverage Docker containers in your tests to provide lightweight infrastructure instances such as databases, message brokers and web browsers. Anything that you can run in a Docker container is subject to be managed by Testcontainers in your test. Citrus as a framework is able to start/stop your Testcontainers instances as part of the test.

> The Testcontainers support is enabled by adding a separate Maven module as a project dependency

```
<dependency>
  <groupId>org.citrusframework</groupId>
  <artifactId>citrus-testcontainers</artifactId>
  <version>${citrus.version}</version>
</dependency>
```

39.1. Start and stop Testcontainers

The Testcontainers test action runs a container and exposes its connection settings as test variables. You can start any Docker container image as a Testcontainers instance. The Testcontainers project provides many modules that represent different test infrastructure technologies.

Each of those modules can be started in Citrus with a special test action:

Java

```
@CitrusTest
public void testcontainersTest() {
    given(testcontainers()
            .start()
            .image("busybox:latest")
            .containerName("busybox-container")
            .withCommand("echo", "Hello", "World")
            .withLabel("app", "citrus"));
}
```

XML

```
<test name="TestcontainersTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <testcontainers>
          <start>
            <container name="busybox-container" image="busybox:latest" command="echo
Hello World">
              <labels>
                <label name="app" value="citrus"/>
              </labels>
            </container>
          </start>
        </testcontainers>
    </actions>
</test>
```

YAML

```
name: TestcontainersTest
actions:
  - testcontainers:
      start:
        container:
          name: "busybox-container"
          image: "busybox:latest"
          command: "echo Hello World"
          labels:
            - name: app
              value: citrus
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

In the test example uses the Docker image name `busybox:latest` to start the Testcontainer instance. You can set labels, annotations as well as the command that is run as entry point when starting the container.

You can also leverage the Testcontainers API to specify the container details:

Java

```
@CitrusTest
public void testcontainersTest() {
    GenericContainer<?> busyBox = new GenericContainer("busybox:latest")
                .withCommand("echo", "Hello World");

    given(testcontainers()
            .start()
            .container("my-container", busyBox));
}
```

The Citrus test automatically exposes connection details of the running Testcontainers instance. The connection settings get exposed in the form of test variables.

These are the test variables that get exposed representing connection settings for the Testcontainers instance.

Variable
CITRUS_TESTCONTAINERS_<CONTAINER_NAME>_HOST
CITRUS_TESTCONTAINERS_<CONTAINER_NAME>_PORT
CITRUS_TESTCONTAINERS_<CONTAINER_NAME>_CONTAINER_IP
CITRUS_TESTCONTAINERS_<CONTAINER_NAME>_CONTAINER_ID
CITRUS_TESTCONTAINERS_<CONTAINER_NAME>_CONTAINER_NAME

You can use these test variables in the test to access the details of the running Testcontainers instance. Special containers may expose additional connection settings such as user credentials, endpoint URLs, managed ports and so on. Read about this in the individual supported Testcontainers module sections that follow in this guide.

Each Testcontainers instance that is started from Citrus is automatically removed after the test. This is the default behavior. However, you can disable the auto removal by setting this system property or environment variable: `citrus.testcontainers.auto.remove.resources=false` or `CITRUS_TESTCONTAINERS_AUTO_REMOVE_RESOURCES=false`

You can also disable the auto removal of the started Testcontainers started by Citrus:

Java

```
@CitrusTest
public void testcontainersTest() {
    GenericContainer<?> busyBox = new GenericContainer("busybox:latest")
                .withCommand("echo", "Hello World");

    given(testcontainers()
            .start()
            .autoRemove(false)
            .container("my-container", busyBox));
}
```

This keeps the started Testcontainers instance running even after the test has finished. When the whole test suite is finished the Testcontainers library may automatically stop and remove the container.

You can explicitly stop a running Testcontainers instance by its name with this test action:

Java

```
@CitrusTest
public void testcontainersTest() {
    given(testcontainers()
            .stop()
            .containerName("busybox-container"));
}
```

XML

```
<test name="TestcontainersTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <testcontainers>
          <stop name="busybox-container"/>
        </testcontainers>
    </actions>
</test>
```

YAML

```
name: TestcontainersTest
actions:
  - testcontainers:
      stop:
        name: "busybox-container"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

This is how you can manage any Docker image as a Testcontainers instance. Citrus also provides special support for test infrastructure Testcontainers modules such as:

- PostgreSQL
- MongoDB
- LocalStack
- Kafka
- Redpanda

The following sections deal with these special Testcontainers modules.

39.2. PostgreSQL

You can start a PostgreSQL instance with the following test action.

Java

```
@CitrusTest
public void testcontainersTest() {
    given(testcontainers()
            .postgreSQL()
            .start()
            .initScript(Resources.create("db.init.sql")));
}
```

XML

```
<test name="TestcontainersTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <testcontainers>
          <start>
            <postgresql>
              <init-script file="classpath:db.init.sql"/>
            </postgresql>
          </start>
        </testcontainers>
    </actions>
</test>
```

YAML

```
name: TestcontainersTest
actions:
  - testcontainers:
      start:
        postgresql:
          initScript:
            file: "db.init.sql"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

In the test example the PostgreSQL Testcontainers module is started. The given init script is used to initialize the database on startup.

Citrus exposes special test variables that represent connection settings for the PostgreSQL container:

Variable
CITRUS_TESTCONTAINERS_POSTGRESQL_HOST
CITRUS_TESTCONTAINERS_POSTGRESQL_CONTAINER_IP
CITRUS_TESTCONTAINERS_POSTGRESQL_CONTAINER_ID
CITRUS_TESTCONTAINERS_POSTGRESQL_CONTAINER_NAME
CITRUS_TESTCONTAINERS_POSTGRESQL_SERVICE_PORT
CITRUS_TESTCONTAINERS_POSTGRESQL_PORT
CITRUS_TESTCONTAINERS_POSTGRESQL_LOCAL_URL
CITRUS_TESTCONTAINERS_POSTGRESQL_SERVICE_LOCAL_URL
CITRUS_TESTCONTAINERS_POSTGRESQL_SERVICE_NAME
CITRUS_TESTCONTAINERS_POSTGRESQL_SERVICE_URL
CITRUS_TESTCONTAINERS_POSTGRESQL_URL
CITRUS_TESTCONTAINERS_POSTGRESQL_DRIVER
CITRUS_TESTCONTAINERS_POSTGRESQL_DB_NAME
CITRUS_TESTCONTAINERS_POSTGRESQL_USERNAME
CITRUS_TESTCONTAINERS_POSTGRESQL_PASSWORD

In addition to that the Citrus test action exposes a DataSource that connects to the PostgreSQL database and binds it as abean to the Citrus registry. This means that you can resolve the data

source bean via the test context reference resolver.

> You can set the name of the exposed data source bean on each Citrus Testcontainers test action for PostgreSQL. By default, Citrus exposed the data source bean with the name `postgreSQL`.

39.3. MongoDB

You can start a MongoDB instance with the following test action.

Java

```
@CitrusTest
public void testcontainersTest() {
    given(testcontainers()
            .mongodb()
            .start());
}
```

XML

```
<test name="TestcontainersTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <testcontainers>
          <start>
            <mongodb/>
          </start>
        </testcontainers>
    </actions>
</test>
```

YAML

```
name: TestcontainersTest
actions:
  - testcontainers:
      start:
        mongodb: {}
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

In the test example the MongoDB Testcontainers module is started. Citrus exposes special test variables that represent connection settings for the MongoDB container:

Variable
CITRUS_TESTCONTAINERS_MONGODB_HOST
CITRUS_TESTCONTAINERS_MONGODB_CONTAINER_IP
CITRUS_TESTCONTAINERS_MONGODB_CONTAINER_ID
CITRUS_TESTCONTAINERS_MONGODB_CONTAINER_NAME
CITRUS_TESTCONTAINERS_MONGODB_LOCAL_URL
CITRUS_TESTCONTAINERS_MONGODB_SERVICE_PORT
CITRUS_TESTCONTAINERS_MONGODB_PORT
CITRUS_TESTCONTAINERS_MONGODB_SERVICE_LOCAL_URL
CITRUS_TESTCONTAINERS_MONGODB_SERVICE_NAME
CITRUS_TESTCONTAINERS_MONGODB_SERVICE_URL
CITRUS_TESTCONTAINERS_MONGODB_URL
CITRUS_TESTCONTAINERS_MONGODB_CONNECTION_STRING

You can use these test variables to connect to the MongoDB Testcontainers instance.

39.4. LocalStack

The LocalStack Testcontainers module allows you to start lightweight Amazon WebServices such as AWS S3, AWS SQS, AWS SNS or AWS Kinesis. You can start a LocslStack instance with the following test action.

Java

```
@CitrusTest
public void testcontainersTest() {
    given(testcontainers()
            .localstack()
            .start()
            .withService(LocalStackContainer.Service.S3));
}
```

XML

```
<test name="TestcontainersTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <testcontainers>
          <start>
            <localstack services="S3"/>
          </start>
        </testcontainers>
    </actions>
</test>
```

YAML

```
name: TestcontainersTest
actions:
  - testcontainers:
      start:
        localstack:
          services:
            - "S3"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

Each LocalStack instance receives a list of services that should be enabled. In the example above the AWS S3 service is enabled.

39.4.1. Exposed connection settings

Citrus exposes special test variables that represent connection settings for the LocalStack container:

Variable
CITRUS_TESTCONTAINERS_LOCALSTACK_HOST
CITRUS_TESTCONTAINERS_LOCALSTACK_CONTAINER_IP
CITRUS_TESTCONTAINERS_LOCALSTACK_CONTAINER_ID
CITRUS_TESTCONTAINERS_LOCALSTACK_CONTAINER_NAME
CITRUS_TESTCONTAINERS_LOCALSTACK_REGION
CITRUS_TESTCONTAINERS_LOCALSTACK_ACCESS_KEY
CITRUS_TESTCONTAINERS_LOCALSTACK_SECRET_KEY

Variable
CITRUS_TESTCONTAINERS_LOCALSTACK_SERVICE_PORT
CITRUS_TESTCONTAINERS_LOCALSTACK_SERVICE_LOCAL_URL
CITRUS_TESTCONTAINERS_LOCALSTACK_SERVICE_NAME
CITRUS_TESTCONTAINERS_LOCALSTACK_SERVICE_URL

For each enabled service on the LocalStack instance these variables are exposed:

Variable
CITRUS_TESTCONTAINERS_LOCALSTACK_<SERVICE_NAME>_URL
CITRUS_TESTCONTAINERS_LOCALSTACK_<SERVICE_NAME>_LOCAL_URL
CITRUS_TESTCONTAINERS_LOCALSTACK_<SERVICE_NAME>_PORT

39.4.2. Service client auto creation

The Citrus LocalStack container action is able to automatically create clients for the enabled services.

By default, Citrus creates the client instances and stores them in the bean Citrus registry. You can then reference the clients by their name.

The following services support auto creation of clients:

- S3 ("s3Client")
- SNS ("snsClient")
- SQS ("sqsClient")
- KINESIS (kinesisClient)
- DYNAMODB (dnyamodbClient)
- EVENT_BRIDGE("eventbridgeClient")

For instance the S3 service creates a client instance called `s3Client`. The clients are automatically configured with the container service settings such as endpointUri, accessKey, secretKey and region for S3.

The underlying client factory is able to handle specific options. For the S3 client factory for instance this is a comma separated list of bucket names to auto create S3 buckets.

Java

```
@CitrusTest
public void testcontainersTest() {
    given(testcontainers()
            .localstack()
            .start()
            .withService(LocalStackContainer.Service.S3))
            .withOption("buckets", "my_bucket_name,another_bucket_name");
}
```

XML

```
<test name="TestcontainersTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <testcontainers>
          <start>
            <localstack services="S3">
              <options>
                <option name="buckets" value="my_bucket_name,another_bucket_name"/>
              </options>
            </localstack>
          </start>
        </testcontainers>
    </actions>
</test>
```

YAML

```
name: TestcontainersTest
actions:
  - testcontainers:
      start:
        localstack:
          services:
            - "S3"
          options:
            buckets: "my_bucket_name,another_bucket_name"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

Please see this list of supported options on the individual client factories:

Client	Option	Description
s3Client	buckets	Comma separated list of bucket names that get auto created
sqsClient	queues	Comma separated list of queue names that get auto created
snsClient	topics	Comma separated list of topic names that get auto created
kinesisClient	streams	Comma separated list of stream names that get auto created
kinesisClient	<stream_name>.shard.count	Shard count set on the given stream
dynamodbClient	tables	Comma separated list of table names that get auto created.
dynamodbClient	<table_name>.id	Id attribute name set on the table as a primary id.
dynamodbClient	<table_name>.view.type	View type set on the table stream specification (default=NEW_AND_OLD_IMAGES).
dynamodbClient	<table_name>.read.capacity	Read capacity set as provisioned throughput (default=1).
dynamodbClient	<table_name>.write.capacity	Write capacity set as provisioned throughput (default=1).
eventbridgeClient	eventBusNames	Comma separated list of event bus names that get auto created

39.4.3. Custom service clients

You can use the exposed connections settings in the test variables to connect to the LocalStack Testcontainers instance.

As an example this S3 client connects to the LocalStack Testcontainers instance and creates a test bucket:

Java

```
private S3Client createS3Client(TestContext context) {
    S3Client s3 = S3Client
            .builder()

.endpointOverride(URI.create(context.getVariable("${CITRUS_TESTCONTAINERS_LOCALSTACK_S
3_URL}")))
            .credentialsProvider(StaticCredentialsProvider.create(
                    AwsBasicCredentials.create(

context.getVariable("${CITRUS_TESTCONTAINERS_LOCALSTACK_ACCESS_KEY}"),

context.getVariable("${CITRUS_TESTCONTAINERS_LOCALSTACK_SECRET_KEY}"))
            ))
            .forcePathStyle(true)

.region(Region.of(context.getVariable("${CITRUS_TESTCONTAINERS_LOCALSTACK_REGION}")))
            .build();

    s3.createBucket(b -> b.bucket(context.getVariable("${aws.s3.bucketNameOrArn}")));

    return s3;
}
```

You can access the Citrus TesContext via `@CitrusResource` injection in your test class or method.

39.5. Kafka

You can start a Kafka instance with the following test action.

Java

```
@CitrusTest
public void testcontainersTest() {
    given(testcontainers()
            .kafka()
            .start());
}
```

XML

```
<test name="TestcontainersTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <testcontainers>
          <start>
            <kafka/>
          </start>
        </testcontainers>
    </actions>
</test>
```

YAML

```
name: TestcontainersTest
actions:
  - testcontainers:
      start:
        kafka: {}
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

In the test example the Kafka Testcontainers module is started. Citrus exposes special test variables that represent connection settings for the Kafka container:

Variable
CITRUS_TESTCONTAINERS_KAFKA_HOST
CITRUS_TESTCONTAINERS_KAFKA_CONTAINER_IP
CITRUS_TESTCONTAINERS_KAFKA_CONTAINER_ID
CITRUS_TESTCONTAINERS_KAFKA_CONTAINER_NAME
CITRUS_TESTCONTAINERS_KAFKA_LOCAL_BOOTSTRAP_SERVERS
CITRUS_TESTCONTAINERS_KAFKA_SERVICE_PORT
CITRUS_TESTCONTAINERS_KAFKA_PORT
CITRUS_TESTCONTAINERS_KAFKA_SERVICE_LOCAL_BOOTSTRAP_SERVERS
CITRUS_TESTCONTAINERS_KAFKA_SERVICE_NAME
CITRUS_TESTCONTAINERS_KAFKA_SERVICE_BOOTSTRAP_SERVERS
CITRUS_TESTCONTAINERS_KAFKA_BOOTSTRAP_SERVERS

You can use these test variables to connect to the Kafka Testcontainers instance.

39.6. Redpanda

You can start a Redpanda instance with the following test action.

Java

```
@CitrusTest
public void testcontainersTest() {
    given(testcontainers()
            .redpanda()
            .start());
}
```

XML

```
<test name="TestcontainersTest"
xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <testcontainers>
          <start>
            <redpanda/>
          </start>
        </testcontainers>
    </actions>
</test>
```

YAML

```
name: TestcontainersTest
actions:
  - testcontainers:
      start:
        redpanda: {}
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

In the test example the Redpanda Testcontainers module is started. Citrus exposes special test variables that represent connection settings for the Redpanda container:

Variable
CITRUS_TESTCONTAINERS_REDPANDA_HOST
CITRUS_TESTCONTAINERS_REDPANDA_CONTAINER_IP
CITRUS_TESTCONTAINERS_REDPANDA_CONTAINER_ID
CITRUS_TESTCONTAINERS_REDPANDA_CONTAINER_NAME
CITRUS_TESTCONTAINERS_REDPANDA_LOCAL_BOOTSTRAP_SERVERS
CITRUS_TESTCONTAINERS_REDPANDA_SERVICE_PORT
CITRUS_TESTCONTAINERS_REDPANDA_PORT
CITRUS_TESTCONTAINERS_REDPANDA_SERVICE_LOCAL_BOOTSTRAP_SERVERS
CITRUS_TESTCONTAINERS_REDPANDA_SERVICE_NAME
CITRUS_TESTCONTAINERS_REDPANDA_SERVICE_BOOTSTRAP_SERVERS
CITRUS_TESTCONTAINERS_REDPANDA_BOOTSTRAP_SERVERS

You can use these test variables to connect to the Redpanda Testcontainers instance.

39.7. Docker compose

The Testcontainers project provides the opportunity to interact with Docker compose. In Citrus you can start a Docker compose specification (`compose.yaml`) as part of the test.

By default, the Testcontainers compose container uses your local Docker compose executable binary. This means you need to have Docker compose installed on your machine. You can verify the installation with (`docker compose version`).

If for some reason you are not able to install Docker compose you may also disable the `useComposeBinary` setting in Citrus (e.g. via System property or environment variable). The Testcontainers library will then use arbitrary Docker containers in combination with an ambassador container that stars and manages the services defined in the Docker compose specification.

A Docker compose specification may look like this:

compose.yaml

```
services:
  web:
    image: nginx:latest
    ports:
      - '8080:80'
    volumes:
      - ./html:/usr/share/nginx/html:ro
```

The specification defines one or more services that are started as part of the `compose up` command:

Java

```
@CitrusTest
public void composeTest() {
    given(testcontainers()
            .compose()
            .up()
            .file("compose.yaml"));
}
```

XML

```
<test name="ComposeTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <testcontainers>
          <compose>
            <up file="compose.yaml"/>
          </compose>
        </testcontainers>
    </actions>
</test>
```

YAML

```
name: ComposeTest
actions:
  - testcontainers:
      compose:
        up:
          file: "compose.yaml"
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

The Docker containers started with Docker compose will be automatically stopped and removed after the test. You can change this behavior by setting one of the System property or environment variable settings: `citrus.testcontainers.compose.auto.remove.resources=false` or `CITRUS_TESTCONTAINERS_COMPOSE_AUTO_REMOVE_RESOURCES=false`

You may explicitly stop all Docker containers with the `compose down` command:

Java

```
@CitrusTest
public void composeTest() {
    given(testcontainers()
            .compose()
            .down());
}
```

XML

```
<test name="ComposeTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <testcontainers>
          <compose>
            <down/>
          </compose>
        </testcontainers>
    </actions>
</test>
```

YAML

```
name: ComposeTest
actions:
  - testcontainers:
      compose:
        down: {}
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <!-- NOT SUPPORTED -->
</spring:beans>
```

The `compose down` test action searches for a single Testcontainers compose container that has been started before. In case you are operating with multiple `compose up` commands in a test you may identify each of those compose containers with a unique name and reference this name in the `compose down` test action.

Chapter 40. Functions

The test framework will offer several functions that are useful throughout the test execution. The functions will always return a string value that is ready for use as variable value or directly inside a text message.

A set of functions is usually combined to a function library. The library has a prefix that will identify the functions inside the test case. The default test framework function library uses a default prefix (citrus). You can write your own function library using your own prefix in order to extend the test framework functionality whenever you want.

The library is built in the Spring configuration and contains a set of functions that are of public use.

```
<citrus:function-library id="testLibrary" prefix="foo:">
          <citrus:function name="randomNumber"
class="org.citrusframework.functions.RandomNumberFunction"/>
          <citrus:function name="randomString"
class="org.citrusframework.functions.RandomStringFunction"/>
          <citrus:function name="customFunction" ref="customFunctionBean"/>
          ...
</citrus:function-library>
```

As you can see the library defines one-to-many functions either referenced as normal Spring bean or by its implementing Java class name. Citrus constructs the library and you are able to use the functions in your test case with the leading library prefix just like this:

```
foo:randomNumber()
foo:randomString()
foo:customFunction()
```

You can add custom function implementations and custom function libraries. Just use a custom prefix for your library. The default Citrus function library uses the **citrus:** prefix.In the next chapters the default functions offered by the framework will be described in detail.

40.1. concat()

The function will combine several string tokens to a single string value. This means that you can combine a static text value with a variable value for instance. A first example should clarify the usage:

```
<testcase name="concatFunctionTest">
    <variables>
        <variable name="date" value="citrus:currentDate(yyyy-MM-dd)" />
        <variable name="text" value="Hello Test Framework!" />
    </variables>
    <actions>
        <echo>
            <message>
                citrus:concat('Today is: ', ${date}, ' right!?')
            </message>
        </echo>
        <echo>
            <message>
                citrus:concat('Text is: ', ${text})
            </message>
        </echo>
    </actions>
</testcase>
```

Please do not forget to mark static text with single quote signs. There is no limitation for string tokens to be combined.

```
citrus:concat('Text1', 'Text2', 'Text3', ${text}, 'Text5', …, 'TextN')
```

The function can be used wherever variables can be used. For instance when validating XML elements in the receive action.

```
<message>
    <validate path="//element/element" value="citrus:concat('Cx1x', ${generatedId})"/>
</message>
```

40.2. substring()

The function will have three parameters.

1. String to work on
2. Starting index
3. End index (optional)

Let us have a look at a simple example for this function:

```
<echo>
    <message>
        citrus:substring('Hello Test Framework', 6)
    </message>
</echo>
<echo>
    <message>
        citrus:substring('Hello Test Framework', 0, 5)
    </message>
</echo>
```

Function output:

```
Test Framework
Hello
```

40.3. stringLength()

The function will calculate the number of characters in a string representation and return the number.

```
<echo>
    <message>citrus:stringLength('Hello Test Framework')</message>
</echo>
```

Function output:

20

40.4. translate()

This function will replace regular expression matching values inside a string representation with a specified replacement string.

```
<echo>
    <message>
        citrus:translate('H.llo Test Fr.mework', '\.', 'a')
    </message>
</echo>
```

Note that the second parameter will be a regular expression. The third parameter will be a simple replacement string value.

Function output:

40.5. substringBefore()

The function will search for the first occurrence of a specified string and will return the substring before that occurrence. Let us have a closer look in a simple example:

```
<echo>
    <message>
        citrus:substringBefore('Test/Framework', '/')
    </message>
</echo>
```

In the specific example the function will search for the '/' character and return the string before that index.

Function output:

Test

40.6. substringAfter()

The function will search for the first occurrence of a specified string and will return the substring after that occurrence. Let us clarify this with a simple example:

```
<echo>
    <message>
        citrus:substringAfter('Test/Framework', '/')
    </message>
</echo>
```

Similar to the substringBefore function the '/' character is found in the string. But now the remaining string is returned by the function meaning the substring after this character index.

Function output:

Framework

40.7. round()

This is a simple mathematical function that will round decimal numbers representations to their nearest non-decimal number.

```
<echo>
    <message>citrus:round('3.14')</message>
</echo>
```

Function output:

3

40.8. floor()

This function will round down decimal number values.

```
<echo>
    <message>citrus:floor('3.14')</message>
</echo>
```

Function output:

3.0

40.9. ceiling()

Similar to floor function, but now the function will round up the decimal number values.

```
<echo>
    <message>citrus:ceiling('3.14')</message>
</echo>
```

Function output:

4.0

40.10. randomNumber()

The random number function will provide you the opportunity to generate random number strings containing positive number letters. There is a singular Boolean parameter for that function describing whether the generated number should have exactly the amount of digits. Default value for this padding flag will be true.

Next example will show the function usage:

```
<variables>
    <variable name="rndNumber1" value="citrus:randomNumber(10)"/>
    <variable name="rndNumber2" value="citrus:randomNumber(10, true)"/>
    <variable name="rndNumber2" value="citrus:randomNumber(10, false)"/>
    <variable name="rndNumber3" value="citrus:randomNumber(3, false)"/>
</variables>
```

Function output:

```
8954638765
5003485980
6387650
65
```

40.11. randomNumberGenerator()

This function is the big brother of randomNumber it generates a random numeric value with customizable decimal precision and range. It provides options to control the minimum and maximum values, inclusivity, and numerical multiples of the generated number.

The function accepts up to six parameters, which influence the generated number:

Parameters

1. Decimal places (optional, default: 0)
    - Defines the number of decimal places in the generated number.
    - A value of 0 results in an integer output.
2. Minimum value (optional, default: -1,000,000)
    - The lower bound for the generated number.
3. Maximum value (optional, default: 1,000,000)
    - The upper bound for the generated number.
4. Exclude minimum (optional, default: false)
    - If true, the generated number will never be exactly the minimum value.
5. Exclude maximum (optional, default: false)
    - If true, the generated number will never be exactly the maximum value.
6. Multiple of (optional)
    - If provided, the generated number will be a multiple of this value.
    - If no valid multiple exists within the given range, the function may return Infinity.
7. Format pattern (optional)
    - If no pattern is specified a plain string representation of the number will be returned.

- e.g. ",.000", " .000", " .#E0"

XML Example

```
<variables>
    <variable name="rndNum1" value="${citrus:randomNumberGenerator()}"/>
    <variable name="rndNum2" value="${citrus:randomNumberGenerator(2)}"/>
    <variable name="rndNum3" value="${citrus:randomNumberGenerator(3, -10, 10)}"/>
    <variable name="rndNum4" value="${citrus:randomNumberGenerator(0, 100, 200, true,
false)}"/>
    <variable name="rndNum5" value="${citrus:randomNumberGenerator(2, 0, 100, false,
true, 5)}"/>
    <variable name="rndNum5" value="${citrus:randomNumberGenerator(0, 10000000,
1000000000, false, false, null, #.###E0)}"/>
</variables>
```

Function Output Examples

```
157
23.78
-5.462
101
15.00
```

If no parameters are provided, the function generates a random integer between -1,000,000 and 1,000,000.

If decimal places are specified, the number will be formatted accordingly.

If multiple of is set, the generated number will always be a multiple of the given value within the range.

If minimum or maximum values are excluded, the function ensures that the output never equals these boundaries.

40.12. randomString()

This function will generate a random string representation with a defined length.

Parameters

1. Length of the generated string. (Required, no default value)
2. Case of the generated letters. (Optional, default: MIXED)
    - Possible values: UPPERCASE, LOWERCASE, MIXED.
3. Digits characters (0-9) included. (Optional, default: false)
    - If set to true, digits will be part of the output.
4. Minimum number of characters in the generated string. (Optional, default: length)

- This value must be positive or zero and less than or equal to the total length.
- If smaller than the length, the generated string will have a random length between the minimum and the specified length.

```
<variables>
    <variable name="rndString0" value="${citrus:randomString(10)}"/>
    <variable name="rndString1" value="citrus:randomString(10)"/>
    <variable name="rndString2" value="citrus:randomString(10, UPPERCASE)"/>
    <variable name="rndString3" value="citrus:randomString(10, LOWERCASE)"/>
    <variable name="rndString4" value="citrus:randomString(10, MIXED)"/>
    <variable name="rndString4" value="citrus:randomString(10, MIXED, true)"/>
    <variable name="rndString4" value="citrus:randomString(10, MIXED, true, 2)"/>
</variables>
```

Function output:

```
HrGHOdfAer
AgSSwedetG
JSDFUTTRKU
dtkhirtsuz
Vt567JkA32
Vt56
```

40.13. randomPattern()

This function generates a random string that matches the given regular expression pattern. It utilizes the Generex library to produce valid strings based on the provided regex.

The function takes one parameter, which is the regular expression pattern to generate the string from.

```
<variables>
    <variable name="rndPattern1" value="${citrus:randomPattern('[A-Z]{5}')}"/>
    <variable name="rndPattern2" value="${citrus:randomPattern('[0-9]{3}-[A-
Z]{2}')}"/>
    <variable name="rndPattern3" value="${citrus:randomPattern('[a-z]{4}[0-9]{2}')}"/>
    <variable name="rndPattern4" value="${citrus:randomPattern('[A-Za-z0-
9]{8,12}')}"/>
</variables>
```

Function output:

```
ABCDE
735-RT
bcda42
TgH56yJk12
```

The generated string strictly adheres to the provided regex pattern. The function supports character ranges, quantifiers, and grouping as per standard Java regex syntax. If an invalid regex is provided, an error will be thrown. An error might also be thrown for very complex regex, which cannot be handled by the Generex library.

40.14. randomEnumValue()

This function returns one of its supplied arguments. Furthermore, you can specify a custom function with a configured list of values (the enumeration). The function will randomly return an entry when called without arguments. This promotes code reuse and facilitates refactoring.

In the next sample the function is used to set a httpStatusCode variable to one of the given HTTP status codes (200, 401, 500)

```
<variable name="httpStatusCode" value="citrus:randomEnumValue('200', '401', '500')" />
```

As mentioned before you can define a custom function for your very specific needs in order to easily manage a list of predefined values like this:

```
<citrus:function-library id="myCustomFunctionLibrary" prefix="custom:">
    <citrus-function name="randomHttpStatusCode" ref="randomHttpStatusCodeFunction"/>
</citrus:function-library>

<bean id="randomHttpStatusCodeFunction"
class="org.citrusframework.functions.core.RandomEnumValueFunction">
  <property name="values">
    <list>
      <value>200</value>
      <value>500</value>
      <value>401</value>
    </list>
  </property>
</bean>
```

We have added a custom function library with a custom function definition. The custom function "randomHttpStatusCode" randomly chooses an HTTP status code each time it is called. Inside the test you can use the function like this:

```
<variable name="httpStatusCode" value="custom:randomHttpStatusCode()" />
```

40.15. currentDate()

This function will definitely help you when accessing the current date. Some examples will show the usage in detail:

```
<echo><message>citrus:currentDate()</message></echo>
<echo><message>citrus:currentDate('yyyy-MM-dd')</message></echo>
<echo><message>citrus:currentDate('yyyy-MM-dd HH:mm:ss')</message></echo>
<echo><message>citrus:currentDate('yyyy-MM-dd'T'hh:mm:ss')</message></echo>
<echo><message>citrus:currentDate('yyyy-MM-dd HH:mm:ss', '+1y')</message></echo>
<echo><message>citrus:currentDate('yyyy-MM-dd HH:mm:ss', '+1M')</message></echo>
<echo><message>citrus:currentDate('yyyy-MM-dd HH:mm:ss', '+1d')</message></echo>
<echo><message>citrus:currentDate('yyyy-MM-dd HH:mm:ss', '+1h')</message></echo>
<echo><message>citrus:currentDate('yyyy-MM-dd HH:mm:ss', '+1m')</message></echo>
<echo><message>citrus:currentDate('yyyy-MM-dd HH:mm:ss', '+1s')</message></echo>
<echo><message>citrus:currentDate('yyyy-MM-dd HH:mm:ss', '-1y')</message></echo>
```

Note that the currentDate function provides two parameters. First parameter describes the date format string. The second will define a date offset string containing year, month, days, hours, minutes or seconds that will be added or subtracted to or from the actual date value.

Function output:

```
01.09.2009
2009-09-01
2009-09-01 12:00:00
2009-09-01T12:00:00
```

40.16. upperCase()

This function converts any string to upper case letters.

```
<echo>
    <message>citrus:upperCase('Hello Test Framework')</message>
</echo>
```

Function output:

HELLO TEST FRAMEWORK

40.17. lowerCase()

This function converts any string to lower case letters.

```
<echo>
    <message>citrus:lowerCase('Hello Test Framework')</message>
</echo>
```

Function output:

hello test framework

40.18. average()

The function will sum up all specified number values and divide the result through the number of values.

```
<variable name="avg" value="citrus:average('3', '4', '5')"/>
```

avg = **4.0**

40.19. minimum()

This function returns the minimum value in a set of number values.

```
<variable name="min" value="citrus:minimum('3', '4', '5')"/>
```

min = **3.0**

40.20. maximum()

This function returns the maximum value in a set of number values.

```
<variable name="max" value="citrus:maximum('3', '4', '5')"/>
```

max = **5.0**

40.21. sum()

The function will sum up all number values. The number values can also be negative.

```
<variable name="sum" value="citrus:sum('3', '4', '5')"/>
```

sum = **12.0**

40.22. absolute()

The function will return the absolute number value.

```
<variable name="abs" value="citrus:absolute('-3')"/>
```

abs = **3.0**

40.23. mapValue()

This function implementation maps string keys to string values. This is very helpful when the used key is randomly chosen at runtime and the corresponding value is not defined during the design time.

The following function library defines a custom function for mapping HTTP status codes to the corresponding messages:

```
<citrus:function-library id="myCustomFunctionLibrary" prefix="custom:">
      <citrus-function name="getHttpStatusMessage"
ref="getHttpStatusMessageFunction"/>
</citrus:function-library>

<bean id="getHttpStatusMessageFunction"
class="org.citrusframework.functions.core.MapValueFunction">
  <property name="values">
    <map>
      <entry key="200" value="OK" />
      <entry key="401" value="Unauthorized" />
      <entry key="500" value="Internal Server Error" />
    </map>
  </property>
</bean>
```

In this example the function sets the variable httpStatusMessage to the 'Internal Server Error' string dynamically at runtime. The test only knows the HTTP status code and does not care about spelling and message locales.

```
<variable name="httpStatusCodeMessage" value="custom:getHttpStatusMessage('500')" />
```

40.24. randomUUID()

The function will generate a random Java UUID.

```
<variable name="uuid" value="citrus:randomUUID()"/>
```

uuid = **98fbd7b0-832e-4b85-b9d2-e0113ee88356**

40.25. encodeBase64()

The function will encode a string to binary data using base64 hexadecimal encoding.

```
<variable name="encoded" value="citrus:encodeBase64('Hallo Testframework')"/>
```

encoded = **VGVzdCBGcmFtZXdvcms=**

It also has an optional charset parameter that is used for encoding the input string, with UTF-8 as its default value.

```
<variable name="encoded" value="citrus:encodeBase64('Hallo Testframework', 'UTF-8')"/>
```

40.26. decodeBase64()

The function will decode binary data to a character sequence using base64 hexadecimal decoding.

```
<variable name="decoded" value="citrus:decodeBase64('VGVzdCBGcmFtZXdvcms=')"/>
```

decoded = **Hallo Testframework**

It also has an optional charset parameter that is used for encoding the input string, with UTF-8 as its default value.

```
<variable name="decoded" value="citrus:decodeBase64('VGVzdCBGcmFtZXdvcms=', 'UTF-
8')"/>
```

40.27. escapeXml()

If you want to deal with escaped XML in your test case you may want to use this function. It automatically escapes all XML special characters.

```
<echo>
    <message>
        <![CDATA[
            citrus:escapeXml('<Message>Hallo Test Framework</Message>')
        ]]>
    </message>
</echo>
```

<Message>Hallo Test Framework</Message>

40.28. escapeJson()

If you want to deal with escaped JSON in your test case you may want to use this function. It automatically escapes all JSON special characters (especially double quotes).

```
<echo>
    <message>
        <![CDATA[
            {
                "kafkaClientConfiguration": {
                    "bootstrapServers": ["${kafkaBootstrapServer}"]
                },
                "messageBody": "citrus:escapeJson("{"exampleJson": "exampleValue"}")"
            }
        ]]>
    </message>
</echo>
```

40.29. cdataSection()

Usually we use CDATA sections to define message payload data inside a testcase. We might run into problems when the payload itself contains CDATA sections as nested CDATA sections are prohibited by XML nature. In this case the next function ships very usefull.

```
<variable name="cdata" value="citrus:cdataSection('payload')"/>
```

cdata = `<![CDATA[payload]]>`

40.30. digestAuthHeader()

Digest authentication is a commonly used security algorithm, especially in Http communication and SOAP WebServices. Citrus offers a function to generate a digest authentication principle used in the Http header section of a message.

```
<variable name="digest"
  value="citrus:digestAuthHeader('username', 'password', 'authRealm', 'acegi',
                            'POST', 'http://127.0.0.1:8080', 'citrus', 'md5')"/>
```

A possible digest authentication header value looks like this:

```
<Digest username=foo,realm=arealm,nonce=MTMzNT,
uri=http://127.0.0.1:8080,response=51f98c,opaque=b29a30,algorithm=md5>
```

You can use these digest headers in messages sent by Citrus like this:

```
<header>
  <element name="citrus_http_Authorization"
    value="vflig:digestAuthHeader('${username}','${password}','${authRealm}',

'${nonceKey}','POST','${uri}','${opaque}','${algorithm}')"/>
</header>
```

This will set a Http Authorization header with the respective digest in the request message. So your test is ready for client digest authentication.

40.31. localHostAddress()

Test cases may use the local host address for some reason (e.g. used as authentication principle). As the tests may run on different machines at the same time we can not use static host addresses. The provided function localHostAddress() reads the local host name dynamically at runtime.

```
<variable name="address" value="citrus:localHostAddress()"/>
```

A possible value is either the host name as used in DNS entry or an IP address value:

address = `<192.168.2.100>`

40.32. changeDate()

This function works with date values and manipulates those at runtime by adding or removing a date value offset. You can manipulate several date fields such as: year, month, day, hour, minute or second.

Let us clarify this with a simple example for this function:

```
<echo>
    <message>citrus:changeDate('01.01.2000', '+1y+1M+1d')</message>
</echo>
<echo>
    <message>citrus:changeDate(citrus:currentDate(), '-1M')</message>
</echo>
```

Function output:

```
02.02.2001
13.04.2013
```

As you can see the change date function works on static date values or dynamic variable values or functions like **citrus:currentDate()** . By default the change date function requires a date format such as the current date function ('dd.MM.yyyy'). You can also define a custom date format:

```
<echo>
    <message>citrus:changeDate('2000-01-10', '-1M-1d', 'yyyy-MM-dd')</message>
</echo>
```

Function output:

```
1999-12-09
```

With this you are able to manipulate all date values of static or dynamic nature at test runtime.

40.33. readFile()

The **readFile** function reads a file resource from given file path and loads the complete file content as function result. The file path can be a system file path as well as a classpath file resource. The file path can have test variables as part of the path or file name. In addition to that the file content can also have test variable values and other functions.

Let's see this function in action:

```
<echo>
    <message>citrus:readFile('classpath:some/path/to/file.txt')</message>
</echo>
<echo>
    <message>citrus:readFile(${filePath})</message>
</echo>
```

The function reads the file content and places the content at the position where the function has been called. This means that you can also use this function as part of Strings and message payloads for instance. This is a very powerful way to extract large message parts to separate file resources. Just add the **readFile** function somewhere to the message content and Citrus will load the extra file content and place it right into the message payload for you.

This function has a second and a third optional parameter that can be used for the following:

- 2nd parameter: a boolean value to indicate that the returned value should be base64 encoded. Defaults to false.

```
<message>citrus:readFile('classpath:some/path/to/file.txt', true)</message>
```

- 3rd parameter: a boolean value to indicate that a dynamic replacement (Citrus variables, functions, etc.) should be performed before the content is base64 encoded. Defaults to false.

```
<message>citrus:readFile('classpath:some/path/to/file.txt', true, true)</message>
```

40.34. message()

When messages are exchanged in Citrus the content is automatically saved to an in memory storage for further access in the test case. That means that functions and test actions can access the messages that have been sent or received within the test case. The **message** function loads a message content from that message store. The message is identified by its name. Receive and send actions usually define the message name. Now we can load the message payload with that name.

Let's see this function in action:

```
<echo>
    <message>citrus:message(myRequest.body())</message>
</echo>
```

The function above loads the message named **myRequest** from the local memory store. This requires a send or receive action to have handled the message before in the same test case.

XML DSL

```
<send endpoint="someEndpoint">
  <message name="myRequest">
    <payload>Some payload</payload>
  </message>
</send>
```

Java DSL

```
send("someEndpoint")
    .message()
    .name("myRequest")
    .body("Some payload");
```

The name of the message is important. Otherwise the message can not be found in the local message store. Note: a message can either be received or sent with a name in order to be stored in the local message store. The **message** function is then able to access the message by its name. In the first example the **body()** has been loaded. Of course we can also access header information.

```
<echo>
    <message>citrus:message(myRequest.header('Operation'))</message>
</echo>
```

The sample above loads the header **Operation** of the message.

In Java DSL the message store is also accessible over the TestContext.

40.35. xpath()

The **xpath** function evaluates a Xpath expressions on some XML source and returns the expression result as String.

```
<echo>
    <message><![CDATA[citrus:xpath('<message><id>1000</id></text>Some text
content</text></message>', '/message/id')]]></message>
</echo>
```

The XML source is given as first function parameter and can be loaded in different ways. In the example above a static XML source has been used. We could load the XML content from external file or just use a test variable.

```
<echo>
    <message><![CDATA[citrus:xpath(citrus:readFile('some/path/to/file.xml'),
'/message/id')]]></message>
</echo>
```

Also accessing the local message store is valid here:

```
<echo>
    <message><![CDATA[citrus:xpath(citrus:message(myRequest.body()),
'/message/id')]]></message>
</echo>
```

This combination is quite powerful as all previously exchanged messages in the test are automatically stored to the local message store. Reusing dynamic message values from other messages becomes very easy then.

40.36. jsonPath()

The **jsonPath** function evaluates a JsonPath expressions on some JSON source and returns the expression result as String.

```
<echo>
    <message><![CDATA[citrus:jsonPath('{ "message": { "id": 1000, "text": "Some text
content" } }', '$.message.id')]]></message>
</echo>
```

The JSON source is given as first function parameter and can be loaded in different ways. In the example above a static JSON source has been used. We could load the JSON content from external file or just use a test variable.

```
<echo>
    <message><![CDATA[citrus:jsonPath(${jsonSource}, '$.message.id')]]></message>
</echo>
```

The JSON source may also be specified in multiple parameters, in which case the arguments except the last one are concatenated with commas, and will be treated as the JSON source. The last parameter is always treated as the JSON path expression.

```
<echo>
    <message><![CDATA[citrus:jsonPath('{ "message": { "id": 1000', '"text": "Some text
content" } }', '$.message.id')]]></message>
</echo>
```

In the example above, the parts `{ "message": { "id": 1000` and `"text": "Some text content" } }` will form the JSON source as `{ "message": { "id": 1000, "text": "Some text content" } }`.

Also accessing the local message store is valid here:

```
<echo>
    <message><![CDATA[citrus:jsonPath(citrus:message(myRequest.body()),
'$.message.id')]]></message>
</echo>
```

This combination is quite powerful as all previously exchanged messages in the test are automatically stored to the local message store. Reusing dynamic message values from other messages becomes very easy then.

40.37. urlEncode()/urlDecode()

The **urlEncode** function takes a String and performs proper URL encoding. The result is a URL encoded String that is using proper character escaping for Http.

```
<echo>
    <message><![CDATA[citrus:urlEncode('foo@citrusframework', 'UTF-8')]]></message>
</echo>
```

The above function takes the String `foo@citrusframework.org` and performs proper URL encoding resulting in `foo%40citrusframework`.

Same logic applies to the `urlDecode()` function that will read an encoded String replacing all escaped characters to the normal String representation.

```
<echo>
    <message><![CDATA[citrus:urlDecode('foo%40citrusframework', 'UTF-8')]]></message>
</echo>
```

The `UTF-8` charset is used during URL encoding operation and is optional as the default is `UTF-8`.

40.38. systemProperty()

The **systemProperty** function resolves a System property expression at test runtime. The resulting value is returned as function result. In case the System property is not available in the JVM an optional default value is used. In case no default value is given the function will fail with errors.

```
<echo>
    <message><![CDATA[citrus:systemProperty('user.name', 'my-default')]]></message>
</echo>
```

40.39. env()

The **env** function can be used to access an environment specific property at test runtime. The environment property can be a variable set on the underlying operating system. Also the `env()` function is able to access the Spring environment settings (see `org.springframework.core.env.Environment`).

As the Spring environment is also able to resolve System properties you can use this function in this manner, too.

```
<echo>
    <message><![CDATA[citrus:env('USER_NAME', 'my-default')]]></message>
</echo>
```

The default value is optional and provides an error fallback in case the environment setting is not available. In case no default value is provided the function will fail with errors.

40.40. unixTimestamp()

unixTimestamp is a parameterless function that simply returns the current epoch timestamp as seconds.

```
<echo>
    <message><![CDATA[citrus:unixTimestamp()]]></message>
</echo>
```

Chapter 41. Validation Matchers

Message validation in Citrus is essential. The framework offers several validation mechanisms for different message types and formats. With test variables we are able to check for simple value equality. We ensure that message entries are equal to predefined expected values. Validation matcher add powerful assertion functionality on top of that. You just can use the predefined validation matcher functionalities in order to perform more complex assertions like **contains** or **isNumber** in your validation statements.

The following sections describe the Citrus default validation matcher implementations that are ready for usage. The matcher implementations should cover the basic assertions on character sequences and numbers. Of course you can add custom validation matcher implementations in order to meet your very specific validation assertions, too.

First of all let us have a look at a validation matcher statement in action so we understand how to use them in a test case.

```
<message>
    <payload>
        <RequestMessage>
            <MessageBody>
                <Customer>
                    <Id>@greaterThan(0)@</Id>
                    <Name>@equalsIgnoreCase('foo')@</Name>
                </Customer>
            </MessageBody>
        </RequestMessage>
    </payload>
</message>
```

The listing above describes a normal message validation block inside a receive test action. We use some inline message payload template as CDATA. As you know Citrus will compare the actual message payload to this expected template in DOM tree comparison. In addition to that you can simply include validation matcher statements. The message element **Id** is automatically validated to be a number greater than zero and the **Name** character sequence is supposed to match 'foo' ignoring case spelling considerations.

Please note the special validation matcher syntax. The statements are surrounded with '@' markers and are identified by some unique name. The optional parameters passed to the matcher implementation state the expected values to match.

> You can use validation matcher with all validation mechanisms - not only with XML validation. Plaintext, JSON, SQL result set validation are also supported.

A set of validation matcher implementations is usually combined to a validation matcher library. The library has a prefix that will identify the validation matcher inside the test case. The default test framework validation matcher library uses a default prefix (citrus). You can write your own validation matcher library using your own prefix in order to extend the test framework

functionality whenever you want.

The library is built in the Spring configuration and contains a set of validation matchers that are of public use.

```
<citrus:validation matcher-library id="testMatcherLibrary" prefix="foo:">
      <citrus:matcher name="isNumber">
class="org.citrusframework.validation.matcher.core.IsNumberValidationMatcher"/>
      <citrus:matcher name="contains">
class="org.citrusframework.validation.matcher.core.ContainsValidationMatcher"/>
      <citrus:matcher name="customMatcher"> ref="customMatcherBean"/>
      ...
      </citrus:validation matcher-library>
```

As you can see the library defines one to many validation matcher members either referenced as normal Spring bean or by its implementing Java class name. Citrus constructs the library and you are able to use the validation matcher in your test case with the leading library prefix just like this:

```
@foo:isNumber()@
@foo:contains()@
@foo:customMatcher()@
```

You can add custom validation matcher implementations and custom validation matcher libraries. Just use a custom prefix for your library. The default Citrus validation matcher library uses no prefix. See now the following sections describing the default validation matcher in Citrus.

41.1. ignore()

The ignore validation matcher is a special matcher that ignores the value and is always positive in its outcome. You should use the ignore validation matcher when only validating the pure existence of an element. The value is ignored but the element has to be present in the message payload.

```
<message>
    <payload>
        <RequestMessage>
            <MessageBody>
                <Customer>
                    <Id>@ignore()@</Id>
                    <Name>@equalsIgnoreCase('foo')@</Name>
                </Customer>
            </MessageBody>
        </RequestMessage>
    </payload>
</message>
```

The ignore validation matcher is the only validation matcher that is able to skip the function parameter body. So you can use both `@ignore()@` and `@ignore@`.

41.2. matchesXml()

The XML validation matcher implementation is the possibly most exciting one, as we can validate nested XML with full validation power (e.g. ignoring elements, variable support). The matcher checks a nested XML fragment to compare against expected XML. For instance we receive the following XML message payload for validation:

```
<GetCustomerMessage>
      <CustomerDetails>
          <Id>5</Id>
          <Name>Christoph</Name>
          <Configuration><![CDATA[
            <config>
                <premium>true</premium>
                <last-login>2012-02-24T23:34:23</last-login>
                <link>http://citrusframework.org/customer/5</link>
            </config>
          ]]></Configuration>
      </CustomerDetails>
</GetCustomerMessage>
```

As you can see the message payload contains some configuration as nested XML data in a CDATA section. We could validate this CDATA section as static character sequence comparison, true. But the <last-login> timestamp changes its value continuously. This breaks the static validation for CDATA elements in XML. Fortunately the new XML validation matcher provides a solution for us:

```
<message>
    <payload>
        <GetCustomerMessage>
            <CustomerDetails>
                <Id>5</Id>
                <Name>Christoph</Name>
                <Configuration>citrus:cdataSection('@matchesXml('<config>
                    <premium>${isPremium}</premium>
                    <last-login>@ignore@</last-login>
                    <link>http://citrusframework.org/customer/5</link>
                  </config>')@')</Configuration>
            </CustomerDetails>
        </GetCustomerMessage>
    </payload>
</message>
```

With the validation matcher you are able to validate the nested XML with full validation power. Ignoring elements is possible and we can also use variables in our control XML.

Nested CDATA elements within other CDATA sections are not allowed by XML standard. This is why we create the nested CDATA section on the fly with the function cdataSection().

41.3. equalsIgnoreCase()

This matcher implementation checks for equality without any case spelling considerations. The matcher expects a single parameter as the expected character sequence to check for.

```
<value>@equalsIgnoreCase('foo')@</value>
```

41.4. contains()

This matcher searches for a character sequence inside the actual value. If the character sequence is not found somewhere the matcher starts complaining.

```
<value>@contains('foo')@</value>
```

The validation matcher also exists in a case-insensitive variant.

```
<value>@containsIgnoreCase('foo')@</value>
```

41.5. startsWith()

The matcher implementation asserts that the given value starts with a character sequence otherwise the matcher will arise some error.

```
<value>@startsWith('foo')@</value>
```

41.6. endsWith()

Ends with matcher validates a value to end with a given character sequence.

```
<value>@endsWith('foo')@</value>
```

41.7. matches()

You can check a value to meet a regular expression with this validation matcher. This is for instance very useful for email address validation.

```
<value>@matches('[a-z0-9]')@</value>
```

41.8. matchesDatePattern()

Date values are always difficult to check for equality. Especially when you have millisecond timestamps to deal with. Therefore the date pattern validation matcher should have some improvement for you. You simply validate the date format pattern instead of checking for total equality.

```
<value>@matchesDatePattern('yyyy-MM-dd')@</value>
```

The example listing uses a date format pattern that is expected. The actual date value is parsed according to this pattern and may cause errors in case the value is not a valid date matching the desired format.

41.9. isNumber()

Checking on values to be of numeric nature is essential. The actual value must be a numeric number otherwise the matcher raises errors. The matcher implementation does not evaluate any parameters.

```
<value>@isNumber()@</value>
```

41.10. lowerThan()

This matcher checks a number to be lower than a given threshold value.

```
<value>@lowerThan(5)@</value>
```

41.11. greaterThan()

The matcher implementation will check on numeric values to be greater than a minimum value.

```
<value>@greaterThan(5)@</value>
```

41.12. isWeekday()

The matcher works on date values and checks that a given date evaluates to the expected day of the week. The user defines the expected day by its name in uppercase characters. The matcher fails in case the given date is another week day than expected.

```
<someDate>@isWeekday('MONDAY')@</someDate>
```

Possible values for the expected day of the week are: MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY or SUNDAY.

The field value has to be a date value otherwise the matcher will fail to parse the date. The matcher requires a date format which is **dd.MM.yyyy** by default. You can change this date format as follows:

```
<someDate>@isWeekday(MONDAY('yyyy-MM-dd'))@</someDate>
```

Now the matcher uses the custom date format in order to parse the date value for evaluation. The validation matcher also works with date time values. In this case you have to give a valid date time format respectively (e.g. FRIDAY('yyyy-MM-dd'T'hh:mm:ss')).

41.13. variable()

This is a very special validation matcher. Instead of performing a validation logic you can save the actual value passed to the validation matcher as new test variable. This comes very handy as you can use the matcher wherever you want: JSON message payloads, XML message payloads, headers and so on.

```
<value>@variable('foo')@</value>
```

The validation matcher creates a new variable **foo** with the actual element value as variable value. When leaving out the control value the field name itself is used as variable name.

```
<date>@variable()@</date>
```

This creates a new variable **date** with the actual element value as variable value.

41.14. dateRange()

The matcher works on date values and checks that a given date is within the expected date range. The user defines the expected date range by specifying a from-date, a to-date and optionally a date format. The matcher fails when the given date lies outside the expected date range.

```
<someDate>@dateRange('01-12-2015', '31-12-2015', 'dd-MM-yyyy')@</someDate>
```

Possible valid values would be 'some date' >= '01-12-2015' and 'some date' <= '31-12-2015'

The date-format is optional and when omitted it is assumed that all dates match the default date format **yyyy-MM-dd** . When specifying a custom date format use Java's date format as a reference for valid date formats. Only dates were used in the example above but we could just as easily used

date and time as shown in the example below

```
<someDate>@dateRange('2015.12.01 07:00:00', '2015.12.01 19:00:00', 'yyyy.MM.dd
HH:mm:ss')@</someDate>
```

41.15. assertThat()

Hamcrest is a very powerful matcher library with extraordinary matcher implementations. You can use Hamcrest matchers also as Citrus validation matchers.

```
<someValue>@assertThat(equalTo(foo))@</someValue>
```

In the listing above we are using the **equalTo()** matcher. All Hamcrest matchers are surrounded by an **assertThat** expression. You are able to combine several Hamcrest matchers then in order to construct very powerful validation logic. See the following examples on what is possible then:

```
<someValue>@assertThat(equalTo(value))@</someValue>
<someValue>@assertThat(not(equalTo(other))@</someValue>
<someValue>@assertThat(is(not(other)))@</someValue>
<someValue>@assertThat(not(is(other)))@</someValue>
<someValue>@assertThat(equalToIgnoringCase(VALUE))@</someValue>
<someValue>@assertThat(containsString(lue))@</someValue>
<someValue>@assertThat(not(containsString(other)))@</someValue>
<someValue>@assertThat(startsWith(val))@</someValue>
<someValue>@assertThat(endsWith(lue))@</someValue>
<someValue>@assertThat(anyOf(startsWith(val), endsWith(lue)))@</someValue>
<someValue>@assertThat(allOf(startsWith(val), endsWith(lue)))@</someValue>
<someValue>@assertThat(isEmptyString())@</someValue>
<someValue>@assertThat(not(isEmptyString()))@</someValue>
<someValue>@assertThat(isEmptyOrNullString())@</someValue>
<someValue>@assertThat(nullValue())@</someValue>
<someValue>@assertThat(notNullValue())@</someValue>
<someValue>@assertThat(empty())@</someValue>
<someValue>@assertThat(not(empty())@</someValue>
<someValue>@assertThat(greaterThan(4))@</someValue>
<someValue>@assertThat(allOf(greaterThan(4), lessThan(6),
not(lessThan(5)))@</someValue>
<someValue>@assertThat(is(not(greaterThan(5))))@</someValue>
<someValue>@assertThat(greaterThanOrEqualTo(5))@</someValue>
<someValue>@assertThat(lessThan(5))@</someValue>
<someValue>@assertThat(not(lessThan(1)))@</someValue>
<someValue>@assertThat(lessThanOrEqualTo(4))@</someValue>
<someValue>@assertThat(hasSize(5))@</someValue>
<someValue>@assertThat(closeTo(9.0))@</someValue>
<someValue>@assertThat(closeTo(9.0, 0.5))@</someValue>
<someValue>@assertThat(isIn(foo, bar))@</someValue>
<someValue>@assertThat(isOneOf(foo, bar))@</someValue>
```

Citrus will automatically perform validation matchers on the element value. Only if all matchers are satisfied the validation will pass.

41.16. ignoreNewLine()

This matcher implementation checks for equality with prior normalization of all new line characters. This includes new line types CR, LF and CRLF as well as multiple new lines in value and control value. So when using this validation matcher all new line characters are removed prior to checking for equality.

Let’s assume that we have a value with new lines that we want to validate using the matcher implementation:

```
<value>This
is
a
value with lots of
new lines</value>
```

You can now skip all new line characters in your control value using the `ignoreNewLine` matcher.

```
<value>@ignoreNewLine('This is a value with lots of new lines')@</value>
```

As you can see the new line characters are not breaking the validation. The other whitespace characters remain untouched though.

41.17. trim()

This trim matcher will remove leading and trailing whitespaces before checking for equality.

Let's assume that we have a value with leading and trailing whitespaces:

```
<value>
This is a value   </value>
```

You can now skip all leading and trailing whitespaces in your control value.

```
<value>@trim('This is a value')@</value>
```

41.18. trimAllWhitespaces()

Sometimes it is necessary to check equality of some value without caring for whitespaces at all. The matcher implementation will remove all whitespaces before checking for equality.

```
<value>   some value   </value>
```

You can now skip all whitespaces in your control value.

```
<value>@trimAllWhitespaces('somevalue')@</value>
```

41.19. isUUIDv4()

This validation matcher checks if a valid UUID version 4 is present. Given the `<variable name="validUuid" value="653ce6fd-dca4-4672-bbc0-16e2b74b8b81"/>`, usage is as following.

Example `isUUIDv4()` validation

```
<actions>
    <send endpoint="helloEndpoint">
        <message>
            <data>Hello Citrus!</data>
        </message>
        <header>
            <element name="message-id" value="${validUuid}"/>
        </header>
    </send>

    <receive endpoint="helloEndpoint">
        <message type="plaintext">
            <data>Hello Citrus!</data>
        </message>
        <header>
            <element name="message-id" value="@isUUIDv4()@"/>
        </header>
    </receive>
</actions>
```

Chapter 42. Citrus agent

The Citrus agent provides a way to run tests as part of the Docker container environment and as part of the Kubernetes platform.

The agent ships as a web application and users are able to run Citrus tests on demand. Usually the agent application runs as a Docker container or Kubernetes deployment so that the agent is able to access internal platform services such as Knative eventing brokers or a service resources exposed only to the current Kubernetes namespace.

The agent application exposes a REST service API for clients to inject tests and trigger test runs.

42.1. Agent REST service API

As a web application the agent exposes several REST endpoints. Http clients are able to trigger test runs and inspect the remote test results.

At service startup the agent will automatically run all tests that can be found according to certain criteria. The agent performs a test scan looking for available tests on multiple sources applying multiple search criteria:

- name pattern: Tests that should be executed must match one of the given class name patterns (e.g. `.*IT`)
- packages: List of packages to scan in the agent classpath. All tests matching the given name patterns will be executed.
- source files: List of source file paths to test sources (e.g. XML, YAML, Cucumber feature). All sources found will be run.
- test-jar: A provided Java archive holding test classes for execution. All previous test filters (e.g. name patterns, package filters) get applied while scanning classes in this JAR.

> A test-jar that holds all tests and its dependencies can be injected into the agent application classpath. When using a Docker container image this injection may be accomplished through Docker volumes. In Kubernetes the test-jar injection may be done via ConfigMaps and volume mounts on the citrus-agent deployment.

The citrus-agent may also run tests on demand rather than on startup. The agent provides a REST service endpoint to trigger a test run. Each call to the service API may use different settings and test run criteria (e.g. name patterns, packages, source files).

The agent REST API provides these operations:

Table 2. REST service operations

Operation	Method	Description
/health	GET	Provides general agent service status as a Json object (UP: true/false)

Operation	Method	Description
/run	GET	Triggers a new test run. The request query parameters may define the test run settings (e.g. name pattern, packages, source files). The operation blocks until the test run is finished and the response represents the test result summary using proper Http response codes for failure (500 INTERNAL_SERVER_ERROR) and success (200 OK).
/run	POST	Triggers a new test run. The request body may be a test configuration object that specifies the test run (e.g. name pattern, packages, source files). The operation blocks until the test run is finished and the response represents the test result summary using proper Http response codes for failure (500 INTERNAL_SERVER_ERROR) and success (200 OK).
/run	PUT	Same as GET/POST run operation but the call is not blocking until the test run is finished. Instead of this clients may inspect the test results calling/polling the `/results` service endpoint.
/results	GET	Get cumulated test results for this agent. This represents the overall test results for all tests that have been run on this agent.

Operation	Method	Description
/results/latest	GET	Get the most recent test results for the latest test run. This also includes results from a pending test run which may deliver partial test results (Http response code partial 206). This way clients can check on when the current test run is finished or still pending.
/results/clear	PUT	Clear the cumulated test results to start fresh.
/results/files	GET	Get test results files that have been created as part of the test run. This includes JUnit test reports and HTML Citrus test suite reports.

42.2. Configuration

The agent reads general settings on startup from system properties and environment variables. The environment settings are well suited for both usual Java runtime environment and containerized runtime environments such as Docker containers or Kubernetes deployments.

The Citrus agent application supports the following settings.

Table 3. System properties

System properties	Description
citrus.agent.name	The name of this agent instance. Default value is `citrus-agent`.
citrus.agent.server.port	The port to listen on for incoming client requests. Default port is `4567`.
citrus.agent.test.engine	The default engine used to run tests on this agent. One of junit, junit5, testng, cucumber. Default is `junit5`.
citrus.agent.time.to.live	The time to wait after the tests have run as part of the startup procedure. Default is `-1` which means indefinite.
citrus.agent.system.exit	Should use system exit to mark test failure (e.g. system exit = -1). Default is `false`.

System properties	Description
citrus.agent.skip.tests	Test execution should bee skipped at server startup. Default is `false` which means that all tests found are automatically run as part of the server startup procedure.
citrus.agent.includes	List of test name patterns to include in test runs. The server searches for tests matching these patterns when running all available tests. Default patterns are `["^.IT$", "^.*ITCase$", "^IT.$"]`
citrus.agent.packages	List of packages to include in test scan.
citrus.agent.default.properties	Comma separated list of key-value pairs that represent default system properties for a test run.
citrus.agent.test.sources	List of test source paths to run. This may be paths to XML, YAML, Cucumber feature files that the agent should run as part of a test run.
citrus.agent.config.class	Bean configuration class to load as part of a test run.
citrus.agent.test.jar	Path to a test Java archive that hold tests for a test run. Default value is `classpath:citrus-agent-tests.jar`. The test-jar execution is skipped when the file does not exist.

Same properties are settable via environment variables.

Table 4. Environment variables

Environment variable	Description
CITRUS_AGENT_NAME	The name of this agent instance. Default value is `citrus-agent`.
CITRUS_AGENT_SERVER_PORT	The port to listen on for incoming client requests. Default port is `4567`.
CITRUS_AGENT_TEST_ENGINE	The default engine used to run tests on this agent. One of junit, junit5, testng, cucumber. Default is `junit5`.
CITRUS_AGENT_TIME_TO_LIVE	The time to wait after the tests have run as part of the startup procedure. Default is `-1` which means indefinite.
CITRUS_AGENT_SYSTEM_EXIT	Should use system exit to mark test failure (e.g. system exit = -1). Default is `false`.

Environment variable	Description
CITRUS_AGENT_SKIP_TESTS	Test execution should bee skipped at server startup. Default is `false` which means that all tests found are automatically run as part of the server startup procedure.
CITRUS_AGENT_INCLUDES	List of test name patterns to include in test runs. The server searches for tests matching these patterns when running all available tests. Default patterns are `["^.IT$", "^.*ITCase$", "^IT.$"]`
CITRUS_AGENT_PACKAGES	List of packages to include in test scan.
CITRUS_AGENT_DEFAULT_PROPERTIES	Comma separated list of key-value pairs that represent default system properties for a test run.
CITRUS_AGENT_TEST_SOURCES	List of test source paths to run. This may be paths to XML, YAML, Cucumber feature files that the agent should run as part of a test run.
CITRUS_AGENT_CONFIG_CLASS	Bean configuration class to load as part of a test run.
CITRUS_AGENT_TEST_JAR	Path to a test Java archive that hold tests for a test run. Default value is `classpath:citrus-agent-tests.jar`. The test-jar execution is skipped when the file does not exist.

42.3. Web UI

The Citrus agent application provides a Web user interface for clients to inspect previous test runs. You can access the user interface pointing your browser to the exposed server port. By default, when running the agent service locally (e.g. via JBang) the exposed Web UI endpoint is http://localhost:4567.

When run on Docker or Kubernetes the exposed port may be mapped to a local port or Service resource that you can access with the browser.

42.4. Docker container image

The Citrus agent also ships as a Docker container image that you can use to run the agent as a Docker container or as part of a Kubernetes deployment. The Docker image is available on this Quay.io repository.

You can use the image in a Kubernetes deployment:

Citrus agent deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/managed-by: citrus
    app.kubernetes.io/name: citrus-agent
  name: citrus-agent
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: citrus-agent
  template:
    metadata:
      labels:
        app.kubernetes.io/managed-by: citrus
        app.kubernetes.io/name: citrus-agent
    spec:
      containers:
      - name: citrus-agent
        env:
        - name: CITRUS_AGENT_SERVER_PORT
          value: "8080"
        image: quay.io/citrusframework/citrus-agent:latest
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

The Deployment may use the environment variables to customize the Citrus agent application. The container exposes the Http port so you can access the Web UI with a Service resource.

You can inject a test jar that contains the tests to run with the agent. Just use a ConfigMap with the archive as binary content and mount the JAR into the Citrus agent pod with volumes.

Test jar ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: citrus-agent-resources
  labels:
    app.kubernetes.io/managed-by: citrus
    app.kubernetes.io/name: citrus-agent
binaryData:
  citrus-agent-tests.jar: |
    [...]
```

Citrus agent with volumes

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/managed-by: citrus
    app.kubernetes.io/name: citrus-agent
  name: citrus-agent
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: citrus-agent
  template:
    metadata:
      labels:
        app.kubernetes.io/managed-by: citrus
        app.kubernetes.io/name: citrus-agent
    spec:
      containers:
      - name: citrus-agent
        env:
        - name: CITRUS_AGENT_TEST_JAR
          value: /deployments/resources/citrus-agent-tests.jar
        - name: CITRUS_AGENT_SERVER_PORT
          value: "8080"
        image: quay.io/citrusframework/citrus-agent:latest
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8080
          name: http
          protocol: TCP
        volumeMounts:
        - mountPath: /deployments/resources
          name: resources
      volumes:
      - name: resources
        configMap:
          defaultMode: 420
          name: citrus-agent-resources
```

The volume mount adds the binary content from the ConfigMap to the Pod and the Citrus agent application will be able to run the tests from that Java archive.

You may also use a ConfigMap that holds other test sources such as XML, YAML, Groovy or Cucumber test files.

42.5. Agent JBang commands

The JBang integration is a perfect way to run the agent locally suited very well for rapid prototyping.

The Citrus JBang command line tooling supports several agent related commands:

Table 5. JBang agent commands

command	Description
agent start	Start the Citrus agent application as a local JBang process.
agent stop	Stop the current agent process.

42.6. Maven plugin

The Citrus agent provides a Maven plugin that you can include into your project build lifecycle. You can use the plugin to build a `test-jar` from your local test sources. The plugin will then start a new Citrus agent service (e.g. in Kubernetes) and run the tests from the `test-jar` artifact. The plugin waits for the remote test execution to finish and provides the test results to the local Maven build.

The Citrus agent Maven plugin provides these goals to execute:

Table 6. Goals

Goal	Build phase	Description
test-jar	package	Builds a test JAR artifact from your test scoped sources
start	pre-integration-test	Create and run a new Citrus agent application instance (locally via JBang, as a Docker container or as a Kubernetes deployment)
run	integration-test	Run the tests from the test-jar artifact in the agent service (e.g. the tests run as part of the Kubernetes platform)
stop	post-integration-test	Stop the agent service
verify	verify	Verify the remote test results and make sure that tests have executed successfully

Each Maven plugin goal is bound to a default Maven build lifecycle phase. Like the Maven failsafe plugin the Citrus agent plugin binds its goals to the Maven `integration-test` phases in the build lifecycle.

The agent plugin makes sure to run the `post-integration-test` phase even when tests are failing.

The plugin propagates the remote test execution results to the local Maven build so the `verify` goal is able to fail the build in case of failed tests.

A typical usage example for the Citrus agent Maven plugin looks like follows:

Maven POM

```
<plugin>
  <groupId>org.citrusframework.mvn</groupId>
  <artifactId>citrus-agent-maven-plugin</artifactId>
  <version>${project.version}</version>
  <configuration>
    <run>
      <async>true</async>
    </run>
    <kubernetes>
      <enabled>true</enabled>
    </kubernetes>
  </configuration>
  <executions>
    <execution>
      <id>citrus-agent-tests</id>
      <goals>
        <goal>test-jar</goal>
        <goal>start</goal>
        <goal>test</goal>
        <goal>stop</goal>
        <goal>verify</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

The example above uses the Citrus agent Maven plugin amd binds its individual goals to the Maven build lifecycle. In this example the Kubernetes support is enabled. This means that the Citrus agent service will be started as a Deployment on the current Kubernetes cluster. Please make sure to connect to the Kubernetes cluster before the Maven build (e.g. via `kubectl`).

The plugin uses the Kubernetes client to create the agent service as a deployment. It creates a ConfigMap from the built test-jar Java archive and makes sure the agent service has access to it (e.g. via volume mounts). It connects to the service and triggers the test run on the agent. It waits for the tes run to finish properly and the remote test results are propagated to the local Maven build. It verifies the test results and makes the local Maven build fail in case of test failures or errors. It automatically removes the Citrus agent service deployment and its related resources from the Kubernetes cluster.

When the local Maven build is successful you can be sure that all tests have run successful on the agent.

Chapter 43. Data dictionaries

Data dictionaries in Citrus provide a new way to manipulate message payload data before a message is sent or received. The dictionary defines a set of keys and respective values. Just like every other dictionary it is used to translate things. In our case we translate message data elements.

You can translate common message elements that are used widely throughout your domain model. As Citrus deals with different types of message data (e.g. XML, JSON) we have different dictionary implementations that are described in the next sections.

43.1. XML data dictionaries

As you would expect XML data dictionaries can be used together with XML formatted message payloads. To do so a dictionary has to be added to the basic Citrus Spring application context which will make the dictionary visible to all test cases:

```
<citrus:xml-data-dictionary id="nodeMappingDataDictionary">
  <citrus:mappings>
    <citrus:mapping path="TestMessage.MessageId" value="${messageId}"/>
    <citrus:mapping path="TestMessage.CorrelationId" value="${correlationId}"/>
    <citrus:mapping path="TestMessage.User" value="Christoph"/>
    <citrus:mapping path="TestMessage.TimeStamp" value="citrus:currentDate()"/>
  </citrus:mappings>
</citrus:xml-data-dictionary>
```

As you can see the dictionary is nothing but a normal Spring bean definition. The **NodeMappingDataDictionary** implementation receives a map of key value pairs where the key is a message element path expression. For XML payloads the message element tree is traversed so the path expression is built for an exact message element inside the payload. When matched the respective value is set according to the value stored within the dictionary.

Alternatively the key-value mappings can be defined in an external file and a reference to the file can be provided:

```
<citrus:xml-data-dictionary id="nodeMappingDataDictionary">
  <citrus:mapping-file path="classpath:org/citrusframework/sample.dictionary"/>
</citrus:xml-data-dictionary>
```

The mapping file content just looks like a normal property file in Java:

```
TestMessage.MessageId=${messageId}
TestMessage.CorrelationId=${correlationId}
TestMessage.User=Christoph
TestMessage.TimeStamp=citrus:currentDate()
```

You can set any message element value inside the XML message payload. The path expression also supports XML attributes. Just use the attribute name as the last part of the path expression. Let us have a closer look at a sample XML message payload with attributes:

```
<TestMessage>
  <User name="Christoph" age="18"/>
</TestMessage>
```

Using this sample XML payload we can access the attributes in the data dictionary as follows:

```
<citrus:mapping path="TestMessage.User.name" value="${userName}"/>
<citrus:mapping path="TestMessage.User.age" value="${userAge}"/>
```

The **NodeMappingDataDictionary** implementation is easy to use and can be used with most basic XML payloads.

For more complex XML payloads where more flexibility is required the XPath data dictionaries may be better suited:

```
<citrus:xpath-data-dictionary id="xpathMappingDataDictionary">
  <citrus:mappings>
    <citrus:mapping path="//TestMessage/MessageId" value="${messageId}"/>
    <citrus:mapping path="//TestMessage/CorrelationId" value="${correlationId}"/>
    <citrus:mapping path="//TestMessage/User" value="Christoph"/>
    <citrus:mapping path="//TestMessage/User/@id" value="123"/>
    <citrus:mapping path="//TestMessage/TimeStamp" value="citrus:currentDate()"/>
  </citrus:mappings>
</citrus:xpath-data-dictionary>
```

As expected XPath mapping expressions are more powerful and can better handle complex scenarios with XML namespaces, attributes and node lists. Just like the node mapping dictionary the XPath mapping dictionary also supports variables, functions and an external mapping file.

XPath works fine with namespaces. In general it is good practice to define a namespace context where you map namespace URI values with prefix values. So your XPath expression is more precise and evaluation is strict. In Citrus the **NamespaceContextBuilder** which is also added as a normal Spring bean to the application context manages namespaces used in your XPath expressions. See our XML and XPath chapters in this documentation for detailed description how to accomplish fail-safe XPath expressions with namespaces.

This completes the XML data dictionary usage in Citrus. Later on we will see some more advanced data dictionary scenarios where we will discuss the usage of dictionary scopes and mapping strategies. But before that let us have a look at other message formats like JSON messages.

43.2. JSON data dictionaries

JSON data dictionaries complement with XML data dictionaries. As usual we have to add the JSON data dictionary to the basic Spring application context first. Once this is done the data dictionary automatically applies for all JSON message payloads in Citrus. This means that all JSON messages sent and received get translated with the JSON data dictionary implementation.

Citrus uses message types in order to evaluate which data dictionary may fit to the message that is currently processed. As usual you can define the message type directly in your test case as attribute inside the sending and receiving message action.

Let us see a simple dictionary for JSON data:

```
<citrus:json-data-dictionary id="jsonMappingDataDictionary">
  <citrus:mappings>
    <citrus:mapping path="TestMessage.MessageId" value="${messageId}"/>
    <citrus:mapping path="TestMessage.CorrelationId" value="${correlationId}"/>
    <citrus:mapping path="TestMessage.User" value="Christoph"/>
    <citrus:mapping path="TestMessage.TimeStamp" value="citrus:currentDate()"/>
  </citrus:mappings>
</citrus:json-data-dictionary>
```

The message path expressions do look very similar to those used in XML data dictionaries. Here the path expression keys do apply to the JSON object graph. See the following sample JSON data which perfectly applies to the dictionary expressions above.

```
{"TestMessage": {
  "MessageId": "1122334455",
  "CorrelationId": "100000001",
  "User": "Christoph",
  "TimeStamp": 1234567890 }
}
```

The path expressions will match a very specific message element inside the JSON object graph. The dictionary will automatically set the message element values then. The path expressions are easy to use as you can traverse the JSON object graph very easy.

Of course the data dictionary does also support test variables, functions. Also very interesting is the usage of JSON arrays. A JSON array element is referenced in a data dictionary like this:

```
<citrus:mapping path="TestMessage.Users[0]" value="Christoph"/>
<citrus:mapping path="TestMessage.Users[1]" value="Julia"/>
```

The **Users** element is a JSON array, so we can access the elements with index. Nesting JSON objects and arrays is also supported so you can also handle more complex JSON data.

The **JsonMappingDataDictionary** implementation is easy to use and fits the basic needs for JSON

data dictionaries. The message element path expressions are very simple and do fit basic needs. However when more complex JSON payloads apply for translation we might reach the boundaries here.

For more complex JSON message payloads JsonPath data dictionaries are very effective:

```
<citrus:json-path-data-dictionary id="jsonMappingDataDictionary">
  <citrus:mappings>
    <citrus:mapping path="$.TestMessage.MessageId" value="${messageId}"/>
    <citrus:mapping path="$..CorrelationId" value="${correlationId}"/>
    <citrus:mapping path="$..Users[0]" value="Christoph"/>
    <citrus:mapping path="$.TestMessage.TimeStamp" value="citrus:currentDate()"/>
  </citrus:mappings>
</citrus:json-path-data-dictionary>
```

JsonPath mapping expressions are way more powerful and can also handle very complex scenarios. You can apply for all elements named *CorrelationId* in one single entry for instance.

43.3. Dictionary scopes

Now that we have learned how to add data dictionaries to Citrus we need to discuss some advanced topics. Data dictionary scopes do define the boundaries where the dictionary may apply. By default data dictionaries are global scope dictionaries. This means that the data dictionary applies to all messages sent and received with Citrus. Of course message types are considered so XML data dictionaries do only apply to XML message types. However global scope dictionaries will be activated throughout all test cases and actions.

You can overwrite the dictionary scope. For instance in order to use an explicit scope. When this is done the dictionary wil not apply automatically but the user has to explicitly set the data dictionary in sending or receiving test action. This way you can activate the dictionary to a very special set of test actions.

```
<citrus:xml-data-dictionary id="specialDataDictionary" global-scope="false">
  <citrus:mapping-file path="classpath:org/citrusframework/sample.dictionary"/>
</citrus:xml-data-dictionary>
```

We set the global scope property to **false** so the dictionary is handled in explicit scope. This means that you have to set the data dictionary explicitly in your test actions:

Java

```
@CitrusTest
public void dictionaryTest() {
    $(send(myEndpoint)
        .message()
        .body("<TestMessage>Hello Citrus</TestMessage>")
        .dictionary("specialDataDictionary"));
}
```

XML

```
<test name="DictionaryTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <send endpoint="myEndpoint">
            <message data-dictionary="specialDataDictionary">
                <body>
                    <data>
                    <![CDATA[
                        <TestMessage>Hello Citrus</TestMessage>
                    ]]>
                    </data>
                </body>
            </message>
        </send>
    </actions>
</test>
```

YAML

```
name: DictionaryTest
actions:
  - send:
      endpoint: "myEndpoint"
      message:
        dataDictionary: specialDataDictionary
        body: |
          <TestMessage>Hello Citrus</TestMessage>
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="DictionaryTest">
        <actions>
            <send endpoint="myEndpoint">
                <message data-dictionary="specialDataDictionary">
                    <data>
                    <![CDATA[
                        <TestMessage>Hello Citrus</TestMessage>
                    ]]>
                    </data>
                </message>
            </send>
        </actions>
    </testcase>
</spring:beans>
```

The sample above is a sending test action with an explicit data dictionary reference set. Before sending the message the dictionary is asked for translation. So all matching message element values will be set by the dictionary accordingly. Other global data dictionaries do also apply for this message but the explicit dictionary will always overwrite the message element values.

43.4. Path mapping strategies

Another advanced topic about data dictionaries is the path mapping strategy. When using simple path expressions the default strategy is always **EXACT** . This means that the path expression has to evaluate exactly to a message element within the payload data. And only this exact message element is translated.

You can set your own path mapping strategy in order to change this behavior. For instance another mapping strategy would be **STARS_WITH** . All elements are translated that start with a certain path expression. Let us clarify this with an example:

```
<citrus:xml-data-dictionary id="nodeMappingDataDictionary" mapping-
strategy="STARTS_WITH">
  <citrus:mappings>
    <citrus:mapping path="TestMessage.Property" value="citrus:randomString()"/>
  </citrus:mappings>
</citrus:xml-data-dictionary>
```

Now with the path mapping strategy set to **STARS_WITH** all message element path expressions starting with **TestMessage.Property** will find translation in this dictionary. Following sample message payload would be translated accordingly:

```
<TestMessage>
    <Property>XXX</Property>
    <PropertyName>XXX</PropertyName>
    <PropertyValue>XXX</PropertyValue>
</TestMessage>
```

All child elements of **TestMessage** starting with **Property** will be translated with this data dictionary. In the resulting message payload Citrus will use a random string as value for these elements as we used the **citrus:randomString()** function in the dictionary mapping.

The next mapping strategy would be **ENDS_WITH** . No surprises here - this mapping strategy looks for message elements that end with a certain path expression. Again a simple example will clarify this for you.

```
<citrus:xml-data-dictionary id="nodeMappingDataDictionary" mapping-
strategy="ENDS_WITH">
  <citrus:mappings>
    <citrus:mapping path="Id" value="citrus:randomNumber()"/>
  </citrus:mappings>
</citrus:xml-data-dictionary>
```

Again let us see some sample message payload for this dictionary usage:

```
<TestMessage>
  <RequestId>XXX</RequestId>
  <Properties>
    <Property>
      <PropertyId>XXX</PropertyId>
      <PropertyValue>XXX</PropertyValue>
    </Property>
    <Property>
      <PropertyId>XXX</PropertyId>
      <PropertyValue>XXX</PropertyValue>
    </Property>
  </Properties>
</TestMessage>
```

In this sample all message elements ending with **Id** would be translated with a random number. No matter where in the message tree the elements are located. This is quite useful but also very powerful. So be careful to use this strategy in global data dictionaries as it may translate message elements that you would not expect in the first place.

Chapter 44. Test actors

The concept of test actors is a way to enable/disable a group of tests actions and message endpoints based on configuration. Test actors help you to use the same tests in different environments where some of the participating services may be available or need to be simulated by Citrus. In particular in end-to-end test scenarios some interface partners may be deployed and ready for participating in the test. In other environments no interface partners are available and Citrus needs to simulate all interfaces accordingly. Test actors allows us to use the very same test definition in both environments by just adjusting the Citrus configuration.

Usually Citrus simulates all interface partners within a test case which is great for continuous integration testing where just the System Under Test is deployed and all surrounding infrastructure and partners get simulated. In end-to-end integration test scenarios some of our interface partners may be deployed and online. Some other interface partners still require Citrus simulation logic.

It would be great if we were able to reuse the Citrus integration tests in all of these described test environments. The Citrus test describes the complete test flow of messages, we only have to remove the simulated send/receive actions for those interface partner applications that are deployed and available in an end-to-end test setup.

With test actors we have the opportunity to link test actions, in particular send/receive message actions, to a named actor. The test actor can be enabled/disabled via configuration and following from that all linked test actions are enabled/disabled, too.

This means the same Citrus test case is runnable with different test actor configurations where some services are deployed as real applications and others may require simulation by Citrus.

44.1. Create test actors

First thing to do is to define one or more test actors in the Citrus configuration. A test actor represents a participating party (e.g. interface partner, backend application, service).

We define the test actors as beans and add them to the Citrus bean registry.

Citrus Bean

```
@BindToRegistry(name = "travel_agency")
public TestActor travelAgencyActor() {
    return new TestActor("travel_agency");
}

@BindToRegistry(name = "royal_airline")
public TestActor royalAirlineActor() {
    return new TestActor("royal_airline");
}

@BindToRegistry(name = "smart_ariline")
public TestActor smartAirlineActor() {
    return new TestActor("smart_ariline");
}
```

Spring Bean

```
@Bean(name = "travel_agency")
public TestActor travelAgencyActor() {
    return new TestActor("travel_agency");
}

@Bean(name = "royal_airline")
public TestActor royalAirlineActor() {
    return new TestActor("royal_airline");
}

@Bean(name = "smart_ariline")
public TestActor smartAirlineActor() {
    return new TestActor("smart_ariline");
}
```

Spring XML

```
<spring:beans xmlns:spring="http://www.springframework.org/schema/beans"
              xmlns:citrus="http://www.citrusframework.org/schema/config">
    <citrus:actor id="travel_agency" name="TRAVEL_AGENCY"/>
    <citrus:actor id="royal_airline" name="ROYAL_AIRLINE"/>
    <citrus:actor id="smart_ariline" name="SMART_AIRLINE"/>
</spring:beans>
```

The listing above defines three test actors participating in our test scenario. A `travel_agency` application which is simulated by Citrus as a calling client, the `smart_airline` application and a `royal_airline` application. Now we have the test actors defined we can link those to message sender/receiver instances and/or test actions within our test case.

44.2. Reference test actors

We need to reference the test actor and set it on a message send or receive action in our test case. We can do this in two different ways. First of all we can set a test actor reference on a message endpoint.

Citrus Bean

```
@BindToRegistry
public JmsSyncEndpoint royalAirlineBookingEndpoint() {
    return JmsEndpoints.jms()
                .synchronous()
                .destination("${royal.airline.request.queue}")
                .actor(royalAirlineActor())
                .build();
}
```

Spring Bean

```
@Bean
public JmsSyncEndpoint royalAirlineBookingEndpoint() {
    return JmsEndpoints.jms()
                .synchronous()
                .destination("${royal.airline.request.queue}")
                .actor(royalAirlineActor())
                .build();
}
```

Spring XML

```
<citrus-jms:sync-endpoint id="royalAirlineBookingEndpoint"
        destination-name="${royal.airline.request.queue}"
        actor="royal_airline"/>
```

Now all test actions that are using this message endpoint are linked to the test actor. In addition to that you can also explicitly link test actions to a test actor.

Java

```
@CitrusTest
public void airlineTest() {
    $(receive("royalAirlineBookingEndpoint")
        .message()
        .body("...")
        .actor("royal_airline")
    );

    $(send("royalAirlineBookingEndpoint")
        .message()
        .body("...")
        .actor("royal_airline")
    );
}
```

XML

```
<test name="AirlineTest" xmlns="http://citrusframework.org/schema/xml/testcase">
    <actions>
        <receive endpoint="royalAirlineBookingEndpoint" actor="royal_airline">
            <message>
                <!-- ... -->
            </message>
        </receive>

        <send endpoint="royalAirlineBookingEndpoint" actor="royal_airline">
            <message>
                <!-- ... -->
            </message>
        </send>
    </actions>
</test>
```

YAML

```
name: AirlineTest
actions:
  - receive:
      endpoint: "royalAirlineBookingEndpoint"
      actor: "royal_airline"
      message: {}
  - send:
      endpoint: "royalAirlineBookingEndpoint"
      actor: "royal_airline"
      message: {}
```

Spring XML

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans">
    <testcase name="AirlineTest">
        <actions>
            <receive endpoint="royalAirlineBookingEndpoint" actor="royal_airline">
                <message>
                    <!-- ... -->
                </message>
            </receive>

            <send endpoint="royalAirlineBookingEndpoint" actor="royal_airline">
                <message>
                    <!-- ... -->
                </message>
            </send>
        </actions>
    </testcase>
</spring:beans>
```

This explicitly links the test actor named `royal_ariline` to test actions. The test actor may be enabled/disabled according to the test environment. All linked test actions and message endpoints will be skipped when the referenced test actor is disabled.

44.3. Disable test actors

By default, test actors are enabled. In some environments it may be required to disable the test actor and all its linked test actions and message endpoints though.

In the airline example usually both airline applications are simulated in the integration tests. Assume that there is a test environment where one of the simulated applications (e.g. the `royal_airline` application) is available as a real application instance deployed. In this scenario the Citrus tests should skip all simulated message interactions for the `royal_airline` application and the real application instance in the test environment should consume the messages instead.

This is easy as we have linked all send/receive actions to one of our test actors. So we can disable the `royal_airline` test actor in our configuration:

Citrus Bean

```
@BindToRegistry(name = "travel_agency")
public TestActor travelAgencyActor() {
    return new TestActor("travel_agency");
}

@BindToRegistry(name = "royal_airline")
public TestActor royalAirlineActor() {
    TestActor royalAirlineActor = new TestActor("royal_airline");
    royalAirlineActor.setDisabled(true);
    return royalAirlineActor;
}

@BindToRegistry(name = "smart_ariline")
public TestActor smartAirlineActor() {
    return new TestActor("smart_ariline");
}
```

Spring Bean

```
@Bean(name = "travel_agency")
public TestActor travelAgencyActor() {
    return new TestActor("travel_agency");
}

@Bean(name = "royal_airline")
public TestActor royalAirlineActor() {
    TestActor royalAirlineActor = new TestActor("royal_airline");
    royalAirlineActor.setDisabled(true);
    return royalAirlineActor;
}

@Bean(name = "smart_ariline")
public TestActor smartAirlineActor() {
    return new TestActor("smart_ariline");
}
```

Spring XML

```
<spring:beans xmlns:spring="http://www.springframework.org/schema/beans"
              xmlns:citrus="http://www.citrusframework.org/schema/config">
    <citrus:actor id="travel_agency" name="TRAVEL_AGENCY"/>
    <citrus:actor id="royal_airline" name="ROYAL_AIRLINE" disabled="true"/>
    <citrus:actor id="smart_ariline" name="SMART_AIRLINE"/>
</spring:beans>
```

Any test action linked to this test actor `royal_airline` is now skipped in the test execution. The real `royal_airline` application available in our test scenario will handle the requests instead. The

`travel_agency` and the `smart_airline` applications still get simulated by Citrus.

This is a perfect way of reusing integration tests in different test scenarios where you enable and disable simulated participating parties in Citrus.

> Server ports may be of special interest when dealing with different test scenarios. You may have to also disable a Citrus embedded Jetty server instance in order to avoid port binding conflicts. You may have to wire endpoint URIs accordingly before executing a test. The real life application may not use the same port and IP address as the Citrus embedded servers for simulation.

44.4. Test actor environment settings

You can enable/disable the test actor also by using System property or environment variable settings. The property and environment variable names follow a specific formatting that includes the test actor name:

- `citrus.test.actor.<name>.enabled=true/false`
- `CITRUS_TEST_ACTOR_<NAME>_ENABLED=true/false`

When the environment setting is available on the test host the test actor with the referenced name is enabled/disabled.

You can set these properties in the `citrus-application.properties` for instance:

citrus-application.properties

```
citrus.test.actor.royal_airline.enabled=false
```

This disables the test actor named `royal_airline` and skips all its linked test actions and message endpoints.

Chapter 45. Test suite actions

A test framework should also provide the functionality to do some work before and after the test run. You could think of preparing/deleting the data in a database or starting/stopping a server in this section before/after a test run. These tasks fit best into the initialization and cleanup phases of Citrus.

> It is important to notice that the Citrus configuration components that we are going to use in the next section belong to a separate XML namespace **citrus-test** . We have to add the namespace declaration to the XML root element of our XML configuration file accordingly.

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
        xmlns:spring="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:citrus-test="http://www.citrusframework.org/schema/testcase"
        xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.citrusframework.org/schema/testcase
        http://www.citrusframework.org/schema/testcase/citrus-testcase.xsd">

        [...]

        </beans>
```

45.1. Before suite

You can influence the behavior of a test run in the initialization phase actually before the tests are executed. See the next code example to find out how it works with actions that take place before the first test is executed:

XML Config

```
<citrus:before-suite id="actionsBeforeSuite">
    <citrus:actions>
        <!-- list of actions before suite -->
    </citrus:actions>
</citrus:before-suite>
```

The Citrus configuration component holds a list of Citrus test actions that get executed before the test suite run. You can add all Citrus test actions here as you would do in a normal test case definition.

XML Config

```
<citrus:before-suite id="actionsBeforeSuite">
    <citrus:actions>
        <citrus-test:sql dataSource="testDataSource"/>
            <citrus-test:statement>CREATE TABLE PERSON (ID integer, NAME
char(250))</citrus-test:statement>
        </citrus-test:sql>
    </citrus:actions>
</citrus:before-suite>
```

Note that we must use the Citrus test case namespace for the nested test action definitions. We access the database and create a table PERSON which is obviously needed in our test cases. You can think of several actions here to prepare the database for instance.

> Citrus offers special startup and shutdown actions that may start and stop server implementations automatically. This might be helpful when dealing with Http servers or WebService containers like Jetty. You can also think of starting/stopping a JMS broker before a test run.

So far we have used XML DSL actions in before suite configuration. Now if you exclusively want to use Java DSL you can do the same with adding a custom class that extends **TestDesignerBeforeSuiteSupport** or **TestRunnerBeforeSuiteSupport** .

Java DSL designer

```
public class MyBeforeSuite extends TestDesignerBeforeSuiteSupport {
    @Override
    public void beforeSuite(TestDesigner designer) {
        designer.echo("This action should be executed before suite");
    }
}
```

The custom implementation extends **TestDesignerBeforeSuiteSupport** and therefore has to implement the method **beforeSuite** . This method add some Java DSL designer logic to the before suite. The designer instance is injected as method argument. You can use all Java DSL methods to this designer instance. Citrus will automatically find and execute the before suite logic. We only need to add this class to the Spring bean application context. You can do this explicitly:

```
<bean id="myBeforeSuite" class="my.company.citrus.MyBeforeSuite"/>
```

Of course you can also use other Spring bean mechanisms such as component-scans here too. The respective test runner implementation extends the **TestRunnerBeforeSuiteSupport** and gets a test runner instance as method argument injected.

Java DSL runner

```
public class MyBeforeSuite extends TestRunnerBeforeSuiteSupport {
    @Override
    public void beforeSuite(TestRunner runner) {
        runner.echo("This action should be executed before suite");
    }
}
```

You can have many before-suite configuration components with different ids in a Citrus project. By default the containers are always executed. But you can restrict the after suite action container execution by defining a suite name, test group names, environment or system properties that should match accordingly:

XML Config

```
<citrus:before-suite id="actionsBeforeSuite" suites="databaseSuite" groups="e2e">
    <citrus:actions>
        <citrus-test:sql dataSource="testDataSource"/>
            <citrus-test:statement>CREATE TABLE PERSON (ID integer, NAME
char(250))</citrus-test:statement>
        </citrus-test:sql>
    </citrus:actions>
</citrus:before-suite>
```

The above before suite container is only executed with the test suite called **databaseSuite** or when the test group **e2e** is defined. Test groups and suite names are only supported when using the TestNG unit test framework. Unfortunately JUnit does not allow to hook into suite execution as easily as TestNG does. This is why after suite action containers are not restricted in execution when using Citrus with the JUnit test framework. You can define multiple suite names and test groups with comma delimited strings as attribute values.

When using the Java DSL before suite support you can set suite names and test group filters by simply calling the respective setter methods in your custom implementation.

```
<bean id="myBeforeSuite" class="my.company.citrus.MyBeforeSuite">
  <property name="suiteNames">
    <list>
      <value>databaseSuite</value>
    </list>
  </property>
  <property name="testGroups">
    <list>
      <value>e2e</value>
    </list>
  </property>
</bean>
```

Environment or system properties are defined as list of key-value pairs. When specified the properties have to be present with respective value. In case the property value is left out in configuration the property must simply exists on the system in order to enable the before suite sequence in that test run.

XML Config

```
<citrus:before-suite id="actionsBeforeSuite" suites="databaseSuite" groups="e2e">
    <citrus:env>
      <citrus:property name="USER"/>
    </citrus:env>
    <citrus:system>
      <citrus:property name="test-stage" value="e2e"/>
    </citrus:system>
    <citrus:actions>
        <citrus-test:sql dataSource="testDataSource"/>
            <citrus-test:statement>CREATE TABLE PERSON (ID integer, NAME
char(250))</citrus-test:statement>
        </citrus-test:sql>
    </citrus:actions>
</citrus:before-suite>
```

In the example above the suite sequence will only apply on environments with *USER* property set and the system property *test-stage* must be set to *e2e*. Otherwise the sequence execution is skipped.

45.2. After suite

A test run may require the test environment to be clean. Therefore it is a good idea to purge all JMS destinations or clean up the database after the test run in order to avoid errors in follow-up test runs. Just like we prepared some data in actions before suite we can clean up the test run in actions after the tests are finished. The Spring bean syntax here is not significantly different to those in before suite section:

XML Config

```
<citrus:after-suite id="actionsAfterSuite">
    <citrus:actions>
        <!-- list of actions after suite -->
    </citrus:actions>
</citrus:after-suite>
```

Again we give the after suite configuration component a unique id within the configuration and put one to many test actions as nested configuration elements to the list of actions executed after the test suite run.

XML Config

```
<citrus:after-suite id="actionsAfterSuite">
    <citrus:actions>
        <citrus-test:sql dataSource="testDataSource"/>
            <citrus-test:statement>DELETE FROM TABLE PERSON</citrus-test:statement>
        </citrus-test:sql>
    </citrus:actions>
</citrus:after-suite>
```

We have to use the Citrus test case XML namespace when defining nested test actions in after suite list. We just remove all data from the database so we do not influence follow-up tests. Quite simple isn't it!?

Of course we can also define Java DSL after suite actions. You can do this by adding a custom class that extends **TestDesignerAfterSuiteSupport** or **TestRunnerAfterSuiteSupport** .

Java DSL designer

```
public class MyAfterSuite extends TestDesignerAfterSuiteSupport {
    @Override
    public void afterSuite(TestDesigner designer) {
        designer.echo("This action should be executed after suite");
    }
}
```

The custom implementation extends **TestDesignerAfterSuiteSupport** and therefore has to implement the method **afterSuite** . This method add some Java DSL designer logic to the after suite. The designer instance is injected as method argument. You can use all Java DSL methods to this designer instance. Citrus will automatically find and execute the after suite logic. We only need to add this class to the Spring bean application context. You can do this explicitly:

```
<bean id="myAfterSuite" class="my.company.citrus.MyAfterSuite"/>
```

Of course you can also use other Spring bean mechanisms such as component-scans here too. The respective test runner implementation extends the **TestRunnerAfterSuiteSupport** and gets a test runner instance as method argument injected.

Java DSL runner

```
public class MyAfterSuite extends TestRunnerAfterSuiteSupport {
    @Override
    public void afterSuite(TestRunner runner) {
        runner.echo("This action should be executed after suite");
    }
}
```

You can have many after-suite configuration components with different ids in a Citrus project. By

default the containers are always executed. But you can restrict the after suite action container execution by defining a suite name, test group names, environment or system properties that should match accordingly:

XML Config

```
<citrus:after-suite id="actionsAfterSuite" suites="databaseSuite" groups="e2e">
    <citrus:actions>
        <citrus-test:sql dataSource="testDataSource"/>
            <citrus-test:statement>DELETE FROM TABLE PERSON</citrus-test:statement>
        </citrus-test:sql>
    </citrus:actions>
</citrus:after-suite>
```

The above after suite container is only executed with the test suite called **databaseSuite** or when the test group **e2e** is defined. Test groups and suite names are only supported when using the TestNG unit test framework. Unfortunately JUnit does not allow to hook into suite execution as easily as TestNG does. This is why after suite action containers are not restricted in execution when using Citrus with the JUnit test framework.

You can define multiple suite names and test groups with comma delimited strings as attribute values.

When using the Java DSL before suite support you can set suite names and test group filters by simply calling the respective setter methods in your custom implementation.

```
<bean id="myAfterSuite" class="my.company.citrus.MyAfterSuite">
  <property name="suiteNames">
    <list>
      <value>databaseSuite</value>
    </list>
  </property>
  <property name="testGroups">
    <list>
      <value>e2e</value>
    </list>
  </property>
</bean>
```

Environment or system properties are defined as list of key-value pairs. When specified the properties have to be present with respective value. In case the property value is left out in configuration the property must simply exist on the system in order to enable the before suite sequence in that test run.

XML Config

```
<citrus:after-suite id="actionsBeforeSuite" suites="databaseSuite" groups="e2e">
    <citrus:env>
      <citrus:property name="USER"/>
    </citrus:env>
    <citrus:system>
      <citrus:property name="test-stage" value="e2e"/>
    </citrus:system>
    <citrus:actions>
        <citrus-test:sql dataSource="testDataSource"/>
            <citrus-test:statement>DELETE FROM TABLE PERSON</citrus-test:statement>
        </citrus-test:sql>
    </citrus:actions>
</citrus:after-suite>
```

In the example above the suite sequence will only apply on environments with *USER* property set and the system property *test-stage* must be set to *e2e*. Otherwise the sequence execution is skipped.

45.3. Before test

Before each test is executed it also might sound reasonable to purge all JMS queues for instance. In case a previous test fails some messages might be left in the JMS queues. Also the database might be in dirty state. The follow-up test then will be confronted with these invalid messages and data. Purging all JMS destinations before a test is therefore a good idea. Just like we prepared some data in actions before suite we can clean up the data before a test starts to execute.

XML Config

```
<citrus:before-test id="defaultBeforeTest">
    <citrus:actions>
        <!-- list of actions before test -->
    </citrus:actions>
</citrus:before-test>
```

The before test configuration component receives a unique id and a list of test actions that get executed before a test case is started. The component receives usual test action definitions just like you would write them in a normal test case definition. See the example below how to add test actions.

XML Config

```
<citrus:before-test id="defaultBeforeTest">
    <citrus:actions>
            <citrus-test:echo>
              <citrus-test:message>This is executed before each test!</citrus-
test:message>
            </citrus-test:echo>
    </citrus:actions>
</citrus:before-test>
```

Note that we must use the Citrus test case XML namespace for the nested test action definitions. You have to declare the XML namespaces accordingly in your configuration root element. The echo test action is now executed before each test in our test suite run. Also notice that we can restrict the before test container execution. We can restrict execution based on the test name, package, test groups and environment or system properties. See following example how this works:

XML Config

```
<citrus:before-test id="defaultBeforeTest" test="*_Ok_Test"
package="org.citrusframework.longrunning.*">
    <citrus:actions>
        <citrus-test:echo>
          <citrus-test:message>This is executed before each test!</citrus-
test:message>
        </citrus-test:echo>
    </citrus:actions>
</citrus:before-test>
```

The above before test component is only executed for test cases that match the name pattern `\*_Ok_Test` and that match the package `org.citrusframework.longrunning.*`. Also we could just use the test name pattern or the package name pattern exclusively. And the execution can be restricted based on the included test groups in our test suite run. This enables us to specify before test actions in various ways. Of course you can have multiple before test configuration components at the same time. Citrus will pick the right containers and put it to execution when necessary.

Environment or system properties are defined as list of key-value pairs. When specified the properties have to be present with respective value. In case the property value is left out in configuration the property must simply exists on the system in order to enable the before suite sequence in that test run.

XML Config

```
<citrus:before-test id="specialBeforeTest">
    <citrus:env>
      <citrus:property name="USER"/>
    </citrus:env>
    <citrus:system>
      <citrus:property name="test-stage" value="e2e"/>
    </citrus:system>
    <citrus:actions>
        <citrus-test:echo>
          <citrus-test:message>This is executed before each test!</citrus-
test:message>
        </citrus-test:echo>
    </citrus:actions>
</citrus:before-test>
```

In the example above the test sequence will only apply on environments with *USER* property set and the system property *test-stage* must be set to *e2e*. Otherwise the sequence execution is skipped.

When using the Java DSL we need to implement the before test logic in a separate class that extends **TestDesignerBeforeTestSupport** or **TestRunnerBeforeTestSupport**

Java DSL designer

```
public class MyBeforeTest extends TestDesignerBeforeTestSupport {
    @Override
    public void beforeTest(TestDesigner designer) {
        designer.echo("This action should be executed before each test");
    }
}
```

As you can see the class implements the method **beforeTest** that is provided with a test designer argument. You simply add the before test actions to the designer instance as usual by calling Java DSL methods on the designer object. Citrus will automatically execute these operations before each test is executed. The same logic applies to the test runner variation that extends **TestRunnerBeforeTestSupport** :

Java DSL runner

```
public class MyBeforeTest extends TestRunnerBeforeTestSupport {
    @Override
    public void beforeTest(TestRunner runner) {
        runner.echo("This action should be executed before each test");
    }
}
```

The before test implementations are added to the Spring bean application context for general activation. You can do this either as explicit Spring bean definition or via package component-scan.

Here is a sample for adding the bean implementation explicitly with some configuration

```
<bean id="myBeforeTest" class="my.company.citrus.MyBeforeTest">
  <property name="packageNamePattern" value="org.citrusframework.e2e"></property>
</bean>
```

We can add filter properties to the before test Java DSL actions so they applied to specific packages or test name patterns. The above example will only apply to tests in package **org.citrusframework.e2e** . Leave these properties empty for default actions that are executed before all tests.

45.4. After test

The same logic that applies to the **before-test** configuration component can be done after each test. The **after-test** configuration component defines test actions executed after each test. Just like we prepared some data in actions before a test we can clean up the data after a test has finished execution.

XML Config

```
<citrus:after-test id="defaultAfterTest">
    <citrus:actions>
        <!-- list of actions after test -->
    </citrus:actions>
</citrus:after-test>
```

The after test configuration component receives a unique id and a list of test actions that get executed after a test case is finished. Notice that the after test actions are executed no matter what result success or failure the previous test case came up to. The component receives usual test action definitions just like you would write them in a normal test case definition. See the example below how to add test actions.

XML Config

```
<citrus:after-test id="defaultAfterTest">
    <citrus:actions>
            <citrus-test:echo>
              <citrus-test:message>This is executed after each test!</citrus-
test:message>
            </citrus-test:echo>
    </citrus:actions>
</citrus:after-test>
```

Please be aware of the fact that we must use the Citrus test case XML namespace for the nested test action definitions. You have to declare the XML namespaces accordingly in your configuration root element. The echo test action is now executed after each test in our test suite run. Of course we can restrict the after test container execution. Supported restrictions are based on the test name,

package, test groups and environment or system properties. See following example how this works:

XML Config

```
<citrus:after-test id="defaultAfterTest" test="*_Error_Test"
package="org.citrusframework.error.*">
    <citrus:actions>
        <citrus-test:echo>
          <citrus-test:message>This is executed after each test!</citrus-test:message>
        </citrus-test:echo>
    </citrus:actions>
</citrus:after-test>
```

The above after test component is obviously only executed for test cases that match the name pattern `\*_Error_Test` and that match the package `org.citrusframework.error.*`. Also we could just use the test name pattern or the package name pattern exclusively. And the execution can be restricted based on the included test groups in our test suite run. This enables us to specify after test actions in various ways. Of course you can have multiple after test configuration components at the same time. Citrus will pick the right containers and put it to execution when necessary.

Environment or system properties are defined as list of key-value pairs. When specified the properties have to be present with respective value. In case the property value is left out in configuration the property must simply exists on the system in order to enable the before suite sequence in that test run.

XML Config

```
<citrus:after-test id="specialAfterTest">
    <citrus:env>
      <citrus:property name="USER"/>
    </citrus:env>
    <citrus:system>
      <citrus:property name="test-stage" value="e2e"/>
    </citrus:system>
    <citrus:actions>
        <citrus-test:echo>
          <citrus-test:message>This is executed after each test!</citrus-test:message>
        </citrus-test:echo>
    </citrus:actions>
</citrus:after-test>
```

In the example above the test sequence will only apply on environments with *USER* property set and the system property *test-stage* must be set to *e2e*. Otherwise the sequence execution is skipped.

When using the Java DSL we need to implement the after test logic in a separate class that extends **TestDesignerAfterTestSupport** or **TestRunnerAfterTestSupport**

Java DSL designer

```
public class MyAfterTest extends TestDesignerAfterTestSupport {
    @Override
    public void afterTest(TestDesigner designer) {
        designer.echo("This action should be executed after each test");
    }
}
```

As you can see the class implements the method **afterTest** that is provided with a test designer argument. You simply add the after test actions to the designer instance as usual by calling Java DSL methods on the designer object. Citrus will automatically execute these operations after each test is executed. The same logic applies to the test runner variation that extends **TestRunnerAfterTestSupport** :

Java DSL runner

```
public class MyAfterTest extends TestRunnerAfterTestSupport {
    @Override
    public void afterTest(TestRunner runner) {
        runner.echo("This action should be executed after each test");
    }
}
```

The after test implementations are added to the Spring bean application context for general activation. You can do this either as explicit Spring bean definition or via package component-scan. Here is a sample for adding the bean implementation explicitly with some configuration

```
<bean id="myAfterTest" class="my.company.citrus.MyAfterTest">
  <property name="packageNamePattern" value="org.citrusframework.e2e"></property>
</bean>
```

We can add filter properties to the after test Java DSL actions so they applied to specific packages or test name patterns. The above example will only apply to tests in package **org.citrusframework.e2e** . Leave these properties empty for default actions that are executed after all tests.

Chapter 46. Customize meta information

Test cases in Citrus are usually provided with some meta information like the author's name or the date of creation. In Citrus you are able to extend this test case meta information with your own very specific criteria.

By default a test case comes shipped with meta information that looks like this:

```
<testcase name="PwdChange_OK_1_Test">
    <meta-info>
        <author>Christoph</author>
        <creationdate>2010-01-18</creationdate>
        <status>FINAL</status>
        <last-updated-by>Christoph</last-updated-by>
        <last-updated-on>2010-01-18T15:00:00</last-updated-on>
    </meta-info>

    [...]
</testcase>
```

You can quite easily add data to this section in order to meet your individual testing strategy. Let us have a simple example to show how it is done.

First of all we define a custom XSD schema describing the new elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:tns="http://www.citrusframework.org/samples/my-testcase-info"
        targetNamespace="http://www.citrusframework.org/samples/my-testcase-info"
        elementFormDefault="qualified">

    <element name="requirement" type="string"/>
    <element name="pre-condition" type="string"/>
    <element name="result" type="string"/>
    <element name="classification" type="string"/>
</schema>
```

We have four simple elements (**requirement**, **pre-condition**, **result** and **classification**) all typed as string. These new elements later go into the test case meta information section.

After we added the new XML schema file to the classpath of our project we need to announce the schema to Spring. As you might know already a Citrus test case is nothing else but a simple Spring configuration file with customized XML schema support. If we add new elements to a test case Spring needs to know the XML schema for parsing the test case configuration file. See the **spring.schemas** file usually placed in the META-INF/spring.schemas in your project.

The file content for our example will look like follows:

```
http://www.citrusframework.org/samples/my-testcase-info/my-testcase-
info.xsd=org/citrusframework/schemas/my-testcase-info.xsd
```

So now we are finally ready to use the new meta-info elements inside the test case:

```
<?xml version="1.0" encoding="UTF-8"?>
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
    xmlns:spring="http://www.springframework.org/schema/beans"
    xmlns:custom="http://www.citrusframework.org/samples/my-testcase-info"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.citrusframework.org/schema/testcase
      http://www.citrusframework.org/schema/testcase/citrus-testcase.xsd
      http://www.citrusframework.org/samples/my-testcase-info
      http://www.citrusframework.org/samples/my-testcase-info/my-testcase-info.xsd">

    <testcase name="PwdChange_OK_1_Test">
        <meta-info>
            <author>Christoph</author>
            <creationdate>2010-01-18</creationdate>
            <status>FINAL</status>
            <last-updated-by>Christoph</last-updated-by>
            <last-updated-on>2010-01-18T15:00:00</last-updated-on>
            <custom:requirement>REQ10001</custom:requirement>
            <custom:pre-condition>Existing user, sufficient rights</custom:pre-
condition>
            <custom:result>Password reset in database</custom:result>
            <custom:classification>PasswordChange</custom:classification>
        </meta-info>

        [...]
    </testcase>
</spring:beans>
```

We use a separate namespace declaration with a custom namespace prefix “custom” in order to declare the new XML schema to our test case. Of course you can pick a namespace url and prefix that fits best for your project.As you see it is quite easy to add custom meta information to your Citrus test case. The customized elements may be precious for automatic reporting. XSL transformations for instance are able to read those meta information elements in order to generate automatic test reports and documentation.

You can also declare our new XML schema in the Eclipse preferences section as user specific XML catalog entry. Then even the schema code completion in your Eclipse XML editor will be available for our customized meta-info elements.

Chapter 47. Tracing incoming/outgoing messages

As we deal with message based interfaces Citrus will send and receive a lot of messages during a test run. Now we may want to see these messages in chronological order as they were processed by Citrus. We can enable message tracing in Citrus in order to save messages to the file system for further investigations.

Citrus offers an easy way to debug all received messages to the file system. You need to enable some specific loggers and interceptors in the Spring application context.

```
<bean class="org.citrusframework.report.MessageTracingTestListener"/>
```

Just add this bean to the Spring configuration and Citrus will listen for sent and received messages for saving those to the file system. You will find files like these in the default test output folder after the test run:

For example:

```
logs/trace/messages/MyTest.msgs
logs/trace/messages/FooTest.msgs
logs/trace/messages/SomeTest.msgs
```

Each Citrus test writes a **.msgs** file containing all messages that went over the wire during the test. By default the debug directory is set to ***logs/trace/messages/*** relative to the project test output directory. But you can set your own output directory in the configuration

```
<bean class="org.citrusframework.report.MessageTracingTestListener">
  <property name="outputDirectory" value="file:/path/to/folder"/>
</bean>
```

> As the file names do not change with each test run message tracing files may be overwritten. So you eventually need to save the generated message debug files before running another group of test cases.

Let's see some sample output for a test case with message communication over SOAP Http:

```
Sending SOAP request:
<?xml version="1.0" encoding="UTF-8"?><SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header>
<Operation xmlns="http://citrusframework.org/test">sayHello</Operation>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
<ns0:HelloRequest xmlns:ns0="http://citrusframework.org/schemas/samples/sayHello.xsd">
    <ns0:MessageId>0857041782</ns0:MessageId>
    <ns0:CorrelationId>6915071793</ns0:CorrelationId>
    <ns0:User>Christoph</ns0:User>
    <ns0:Text>Hello WebServer</ns0:Text>
</ns0:HelloRequest>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

=========================================================================

Received SOAP response:
<?xml version="1.0" encoding="UTF-8"?><SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header/>
<SOAP-ENV:Body>
<ns0:HelloResponse
xmlns:ns0="http://citrusframework.org/schemas/samples/sayHello.xsd">
    <ns0:MessageId>0857041782</ns0:MessageId>
    <ns0:CorrelationId>6915071793</ns0:CorrelationId>
    <ns0:User>WebServer</ns0:User>
    <ns0:Text>Hello Christoph</ns0:Text>
</ns0:HelloResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

For this message tracing to work we need to add logging listeners to our sender and receiver components accordingly.

```
<citrus-ws:client id="webServiceClient"
    request-url="http://localhost:8071"
    message-factory="messageFactory"
    interceptors="clientInterceptors"/>

    <util:list id="clientInterceptors">
        <bean class="org.citrusframework.ws.interceptor.LoggingClientInterceptor"/>
    </util:list>
```

Be aware of adding the Spring **util** XML namespace to the application context when using the **util:list** construct.

Chapter 48. Reporting and Test Results

The framework generates different reports and results after a test run for you. These report and result pages will help you to get an overview of the test cases that were executed and which one were failing. They're being generated into `target/citrus-reports` by default, but that can be changed by setting the `citrus.report.directory` property to the desired location. E.g. it could be set to `citrus.report.directory=build/citrus-reports` for Gradle builds.

In the following chapters, we'll provide an overview of the three standard loggers available in Citrus. It's important to note that in Spring, you have the flexibility to replace these standard loggers with your own implementations. This can be achieved by registering custom implementations, as demonstrated in the following code snippet:

```
@Configuration
public static class OverrideReporterConfiguration {

    @Bean
    public LoggingReporter citrusLoggingReporter() {
        return new OverrideLoggingReporter();
    }

    @Bean
    public JUnitReporter citrusJunitReporter() {
        return new OverrideJUnitReporter();
    }

    @Bean
    public HtmlReporter citrusHtmlReporter() {
        return new OverrideHtmlReporter();
    }
}
```

Note that you need to allow spring bean overriding to load this configuration as such. It's worth noting that bean overriding is not recommended by Spring. Thus the following properties are provided to deactivate standard reporters as needed.

```
citrus.default.logging.reporter.enabled=false
citrus.default.junit.reporter.enabled=false
citrus.default.html.reporter.enabled=false
```

48.1. Console logging

During the test run the framework will provide a huge amount of information that is printed out to the console. This includes current test progress, validation results and error information. This enables the user to quickly supervise the test run progress. Failures in tests will be printed to the console just the time the error occurred. The detailed stack trace information and the detailed error

messages are helpful to get the idea what went wrong.

As the console output might be limited to a defined buffer limit, the user may not be able to follow the output to the very beginning of the test run. Therefore the framework additionally prints all information to a log file according to the logging configuration.

The logging mechanism uses the SLF4J logging framework. SLF4J is independent of logging framework implementations on the market. So in case you use Log4J logging framework the specified log file path as well as logging levels can be freely configured in the respective log4j.xml file in your project. At the end of a test run the combined test results get printed to both console and log file. The overall test results look like following example:

```
CITRUS TEST RESULTS

  [...]
  HelloService_Ok_1             : SUCCESS
  HelloService_Ok_2             : SUCCESS
  EchoService_Ok_1              : SUCCESS
  EchoService_Ok_2              : SUCCESS
  EchoService_TempError_1       : SUCCESS
  EchoService_AutomaticRetry_1 : SUCCESS
  [...]

  Found 175 test cases to execute
  Skipped 0 test cases (0.0%)
  Executed 175 test cases
  Tests failed:         0 (0.0%)
  Tests successfully: 175 (100.0%)
```

Failed tests will be marked as failed in the result list. The framework will give a short description of the error cause while the detailed stack trace information can be found in the log messages that were made during the test run.

```
HelloService_Ok_3 : failed - Exception is Action timed out
```

48.2. JUnit reports

As tests are executed as TestNG test cases, the framework will also generate JUnit compliant XML and HTML reports. JUnit test reports are very popular and find support in many build management and development tools. In general the Citrus test reports give you an overall picture of all tests and tell you which of them were failing.

Build management tools like Jenkins can easily import and display the generated JUnit XML results. Please have a look at the TestNG and JUnit documentation for more information about this topic as well as the build management tools (e.g. Jenkins) to find out how to integrate the tests results.

48.3. HTML reports

Citrus creates HTML reports after each test run. The report provides detailed information on the test run with a summary of all test results. You can find the report after a test run in the directory ***${project.build.directory}/citrus-reports*** .

The report consists of two parts. The test summary on top shows the total number executed tests. The main part lists all test cases with detailed information. With this report you immediately identify all tests that were failing. Each test case is marked in color according to its result outcome.

The failed tests give detailed error information with error messages and Java StackTrace information. In addition to that the report tries to find the test action inside the XML test part that failed in execution. With the failing code snippet you can see where the test stopped.

> JavaScript should be active in your web browser. This is to enable the detailed information which comes to you in form of tooltips like test author or description. If you want to access the tooltips JavaScript should be enabled in your browser.

The HTML reports are customizable by system properties. Use following properties e.g. in your **citrus.properties** file:

citrus.html.report.enabled	Enables/disables HTML report generation (default= ***true***).
citrus.html.report.directory	Output directory path (default= ***${project.build.directory}/citrus-reports***).
citrus.html.report.file	File name for the report file (default= ***citrus-test-results.html***).
citrus.html.report.template	Template HTML file with placeholders for report results.
citrus.html.report.detail.template	Template file for detailed test results.
citrus.html.report.logo	File resource path pointing to an image that is added to top of HTML report.

The HTML report is based on a template file that is customizable to your special needs. The default templates can be found in report-templates sources.

Chapter 49. Configuration options

You have several options to customize your Citrus project. Citrus uses default settings that can be overwritten to some extent. As a framework Citrus internally works with components organized in a central context (e.g. the Spring application context). Citrus registers components in the context in order to share those with the test execution runtime. You can customize the behavior of these components over environment variables and system properties.

49.1. Environment settings

Citrus as an application reads general settings from system properties and environment variables. The Citrus framework settings initialize on the startup and evaluate environment settings in favor of using default values.

The environment settings are well suited for both usual Java runtime environment and containerized runtime environments such as Docker or Kubernetes. The following settings do support this kind of environment configuration.

Table 7. System properties

System properties	Description
citrus.application.properties	File location for application property file that holds other settings. These properties get loaded as system properties on startup. (default="classpath:citrus-application.properties")
citrus.java.config	Class name for custom Java configuration (default=null)
citrus.file.encoding	Default file encoding used in Citrus when reading and writing file content (default=Charset.defaultCharset())
citrus.default.message.type	Default message type for validating payloads (default="XML")
citrus.test.name.variable	Default test name variable that is automatically created for each test (default="citrus.test.name")
citrus.test.package.variable	Default test package variable that is automatically created for each test (default="citrus.test.package")
citrus.default.src.directory	Default test source directory (default="src/test/")
citrus.xml.file.name.pattern	File name patterns used for XML test file package scan (default="/**/*Test.xml,/**/*IT.xml")

System properties	Description
citrus.java.file.name.pattern	File name patterns used for Java test sources package scan (default="/**/*Test.java,/**/*IT.java")

Same properties are settable via environment variables.

Table 8. Environment variables

Environment variable	Description
CITRUS_APPLICATION_PROPERTIES	File location for application property file that holds other settings. These properties get loaded as system properties on startup. (default="classpath:citrus-application.properties")
CITRUS_JAVA_CONFIG	Class name for custom Java configuration (default=null)
CITRUS_FILE_ENCODING	Default file encoding used in Citrus when reading and writing file content (default=Charset.defaultCharset())
CITRUS_DEFAULT_MESSAGE_TYPE	Default message type for validating payloads (default="XML")
CITRUS_TEST_NAME_VARIABLE	Default test name variable that is automatically created for each test (default="citrus.test.name")
CITRUS_TEST_PACKAGE_VARIABLE	Default test package variable that is automatically created for each test (default="citrus.test.package")
CITRUS_DEFAULT_SRC_DIRECTORY	Default test source directory (default="src/test/")
CITRUS_XML_FILE_NAME_PATTERN	File name patterns used for XML test file package scan (default="/**/*Test.xml,/**/*IT.xml")
CITRUS_JAVA_FILE_NAME_PATTERN	File name patterns used for Java test sources package scan (default="/**/*Test.java,/**/*IT.java")

49.2. Spring configuration settings

When spring framework is enabled in Citrus you can set specific settings regarding the Spring application context.

Table 9. System properties

System properties	Description
citrus.spring.application.context	File location for Spring XML configurations (default="classpath*:citrus-context.xml")
citrus.spring.java.config	Class name for Spring Java config (default=null)

Table 10. Environment variables

Environment variable	Description
CITRUS_SPRING_APPLICATION_CONTEXT	File location for Spring XML configurations (default="classpath*:citrus-context.xml")
CITRUS_SPRING_JAVA_CONFIG	Class name for Spring Java config (default=null)

49.3. Property file settings

As mentioned in the previous section Citrus as a framework references some basic settings from system environment properties or variables. You can overwrite these settings in a central property file which is loaded at the very beginning of the Citrus runtime.

The properties in that file are automatically loaded as Java system properties. Just add a property file named **citrus-application.properties** to your project classpath. This property file contains customized settings such as:

citrus-application.properties

```
citrus.spring.application.context=classpath*:citrus-custom-context.xml
citrus.spring.java.config=org.citrusframework.config.MyCustomConfig
citrus.file.encoding=UTF-8
citrus.default.message.type=XML
citrus.xml.file.name.pattern=/**/*Test.xml,/**/*IT.xml
```

Citrus automatically loads these application properties at the startup. All properties are also settable with Java system properties directly. The location of the `citrus-application.properties` file is customizable with the system property `citrus.application.properties or environment variable` `CITRUS_APPLICATION_PROPERTIES`. ` .Custom property file location.

```
CITRUS_APPLICATION_PROPERTIES=file:/custom/path/to/citrus-application.properties
```

You can use **classpath:** and **file:** path prefix in order to give locate a classpath or file-system resource.

Chapter 50. Spring support

The Spring framework provides an awesome set of projects, libraries and tools and is a wide spread and well appreciated framework for Java. The dependency injection and IoC concepts introduced with Spring are awesome.

Citrus is able to work with Spring in terms of loading central components as Spring beans as part of a Spring application context. In the following you can use Spring autowiring and configuration in your tests.

Read the following chapters to know more about how to use Citrus together with Spring.

50.1. Spring XML application context

Citrus supports the Spring framework as IoC container in order to load all components as Spring beans in a central application context. By default, Citrus loads basic components as Spring beans in a Spring Java config class.

With Spring in place it is very easy to change/add custom bean components in the Spring application context. Citrus searches for custom Spring application context files in your project and adds these bean definitions to the Spring application context.

By default, Citrus looks for custom XML Spring application context files in this location: `classpath*:citrus-context.xml`. So you can just add this file named **citrus-context.xml** to your project classpath and Citrus will load all Spring beans automatically.

The location of this file can be customized by setting a System property `citrus.spring.application.context` or the environment variable `CITRUS_SPRING_APPLICATION_CONTEXT`.

Custom Spring XML bean definition context.

```
CITRUS_SPRING_APPLICATION_CONTEXT=file:/custom/path/to/custom-beans.xml
```

custom-beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:citrus="http://www.citrusframework.org/schema/config"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.citrusframework.org/schema/config
            http://www.citrusframework.org/schema/config/citrus-config.xsd
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context.xsd">

  <!-- Add your bean definitions here -->

</beans>
```

The file `custom-beans.xml` should provide a normal Spring bean XML configuration. You can add Spring beans as usual and you can use the Citrus XML components provided by the schemas like `xmlns:citrus="http://www.citrusframework.org/schema/config"`.

Citrus provides several schemas for custom Spring XML components. These are described in more detail in the respective chapters and sections in this reference guide.

> You can also use import statements in this Spring application context in order to load other configuration files. So you are free to modularize your configuration in several files that get loaded by Citrus.

50.2. Spring Java config

You can also use pure Java code to load Spring beans as a configuration. Citrus is able to load the Spring beans from a configuration class. Please define the configuration class with the System property `citrus.spring.java.config` or with the environment variable `CITRUS_SPRING_JAVA_CONFIG`.

Custom Spring Java configuration class.

```
CITRUS_SPRING_JAVA_CONFIG=custom.package.to.MyCustomConfig
```

Citrus loads the given Spring bean configuration class in **MyCustomConfig.class** and adds all defined Spring beans to the application context. See the following example for custom Spring Java configuration:

MyCustomConfig.java

```
import org.citrusframework.TestCase;
import org.citrusframework.report.*;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MyCustomConfig {

    @Bean(name = "plusMinusTestReporter")
    public TestReporter plusMinusTestReporter() {
        return new PlusMinusTestReporter();
    }

    /**
     * Sample test reporter.
     */
    private static class PlusMinusTestReporter extends AbstractTestReporter {

        @Override
        public void generate(TestResults testResults) {
            StringBuilder testReport = new StringBuilder();

            testResults.doWithResults(result -> {
                if (result.isSuccess()) {
                    testReport.append("+");
                } else if (result.isFailed()) {
                    testReport.append("-");
                } else {
                    testReport.append("o");
                }
            });


LoggerFactory.getLogger(PlusMinusTestReporter.class).info(testReport.toString());
        }
    }
}
```

You can mix XML and Java based Spring bean configuration. Citrus loads both sources and adds beans to the Spring bean application context during start.

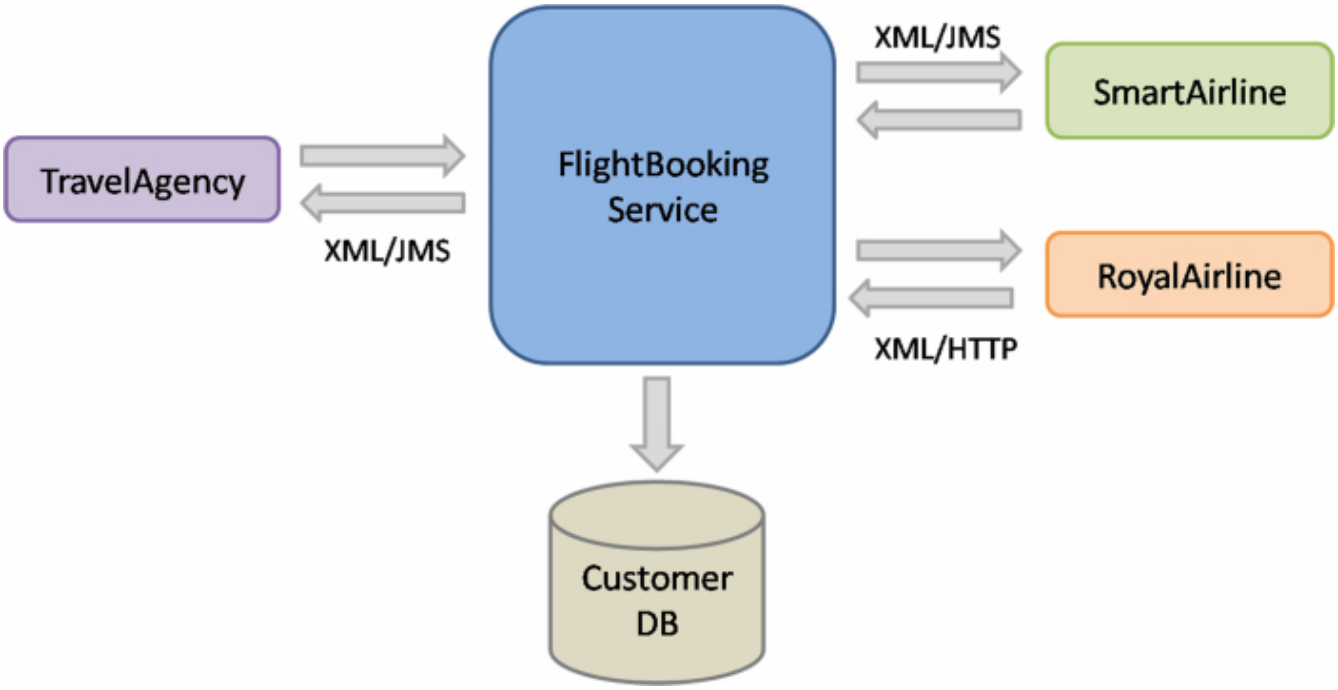
Chapter 51. Samples

This chapter gives some samples where you can see Citrus in action.

51.1. The FlightBooking sample

A simple project example should give you the idea how Citrus works. The system under test is a flight booking service that handles travel requests from a travel agency. A travel request consists of a complete travel route including several flights. The FlightBookingService application will split the complete travel booking into separate flight bookings that are sent to the respective airlines in charge. The booking and customer data is persisted in a database.

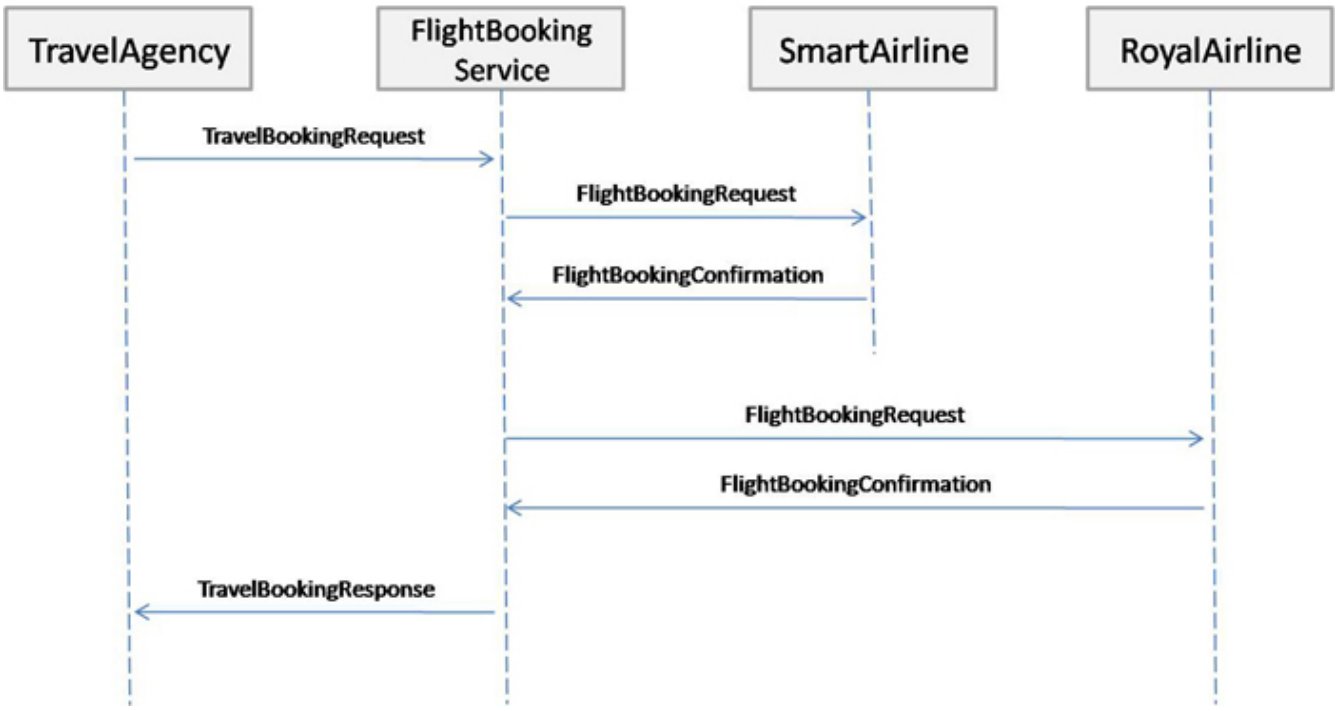
The airlines will confirm or deny the flight bookings. The FlightBookingService application consolidates all incoming flight confirmations and combines them to a complete travel confirmation or denial that is sent back to the travel agency. Next picture tries to put the architecture into graphics:



In our example two different airlines are connected to the FlightBookingService application: the SmartAriline over JMS and the RoyalAirline over Http.

51.1.1. The use case

The use case that we would like to test is quite simple. The test should handle a simple travel booking and expect a positive processing to the end. The test case neither simulates business errors nor technical problems. Next picture shows the use case as a sequence diagram.



The travel agency puts a travel booking request towards the system. The travel booking contains two separate flights. The flight requests are published to the airlines (SmartAirline and RoyalAirline). Both airlines confirm the flight bookings with a positive answer. The consolidated travel booking response is then sent back to the travel agency.

51.1.2. Configure the simulated systems

Citrus simulates all surrounding applications in their behavior during the test. The simulated applications are: TravelAgency, SmartAirline and RoyalAirline. The simulated systems have to be configured in the Citrus configuration first. The configuration is done in Spring XML configuration files, as Citrus uses Spring to glue all its services together.

First of all we have a look at the TravelAgency configuration. The TravelAgency is using JMS to connect to our tested system, so we need to configure this JMS connection in Citrus.

```
<bean id="connectionFactory"
class="org.apache.activemq.artemis.jms.client.ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>

<citrus-jms:endpoint id="travelAgencyBookingRequestEndpoint"
                     destination-name="${travel.agency.request.queue}"/>

<citrus-jms:endpoint id="travelAgencyBookingResponseEndpoint"
                     destination-name="${travel.agency.response.queue}"/>
```

This is all Citrus needs to send and receive messages over JMS in order to simulate the TravelAgency. By default all JMS message senders and receivers need a connection factory. Therefore Citrus is searching for a bean named "connectionFactory". In the example we connect to a ActiveMQ message broker. A connection to other JMS brokers like TIBCO EMS or Apache

ActiveMQ is possible too by simply changing the connection factory implementation.

The identifiers of the message senders and receivers are very important. We should think of suitable ids that give the reader a first hint what the sender/receiver is used for. As we want to simulate the TravelAgency in combination with sending booking requests our id is "travelAgencyBookingRequestEndpoint" for example.

The sender and receivers do also need a JMS destination. Here the destination names are provided by property expressions. The Spring IoC container resolves the properties for us. All we need to do is publish the property file to the Spring container like this.

```
<bean name="propertyLoader"
   class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
        <list>
            <value>citrus.properties</value>
        </list>
    </property>
    <property name="ignoreUnresolvablePlaceholders" value="true"/>
</bean>
```

The citrus.properties file is located in our project's resources folder and defines the actual queue names besides other properties of course:

```
#JMS queues
travel.agency.request.queue=Travel.Agency.Request.Queue
travel.agency.response.queue=Travel.Agency.Response.Queue
smart.airline.request.queue=Smart.Airline.Request.Queue
smart.airline.response.queue=Smart.Airline.Response.Queue
royal.airline.request.queue=Royal.Airline.Request.Queue
```

What else do we need in our Spring configuration? There are some basic beans that are commonly defined in a Citrus application but I do not want to bore you with these details. So if you want to have a look at the Spring application context file in the resources folder and see how things are defined there.

We continue with the first airline to be configured the SmartAirline. The SmartAirline is also using JMS to communicate with the FlightBookingService. So there is nothing new for us, we simply define additional JMS message senders and receivers.

```
<citrus-jms:endpoint id="smartAirlineBookingRequestEndpoint"
                     destination-name="${smart.airline.request.queue}"/>

<citrus-jms:endpoint id="smartAirlineBookingResponseEndpoint"
                     destination-name="${smart.airline.response.queue}"/>
```

We do not define a new JMS connection factory because TravelAgency and SmartAirline are using

the same message broker instance. In case you need to handle multiple connection factories simply define the connection factory with the attribute "connection-factory".

```
<citrus-jms:endpoint id="smartAirlineBookingRequestEndpoint"
                             destination-name="${smart.airline.request.queue}"
                             connection-factory="smartAirlineConnectionFactory"/>

<citrus-jms:endpoint id="smartAirlineBookingResponseEndpoint"
                            destination-name="${smart.airline.response.queue}"
                            connection-factory="smartAirlineConnectionFactory"/>
```

51.1.3. Configure the Http adapter

The RoyalAirline is connected to our system using Http request/response communication. This means we have to simulate a Http server in the test that accepts client requests and provides proper responses. Citrus offers a Http server implementation that will listen on a port for client requests. The adapter forwards incoming request to the test engine over JMS and receives a proper response that is forwarded as a Http response to the client. The next picture shows this mechanism in detail.

The RoyalAirline adapter receives client requests over Http and sends them over JMS to a message receiver as we already know it. The test engine validates the received request and provides a proper response back to the adapter. The adapter will transform the response to Http again and publishes it to the calling client. Citrus offers these kind of adapters for Http and SOAP communication. By writing your own adapters like this you will be able to extend Citrus so it works with protocols that are not supported yet.

Let us define the Http adapter in the Spring configuration:

```
<citrus-http:server id="royalAirlineHttpServer"
                        port="8091"
                        uri="/flightbooking"
                        endpoint-adapter="jmsEndpointAdapter"/>

<citrus-jms:endpoint-adapter id="jmsEndpointAdapter
      destination-name="${royal.airline.request.queue}"/>
      connection-factory="connectionFactory" />
      timeout="2000"/>

<citrus-jms:sync-endpoint id="royalAirlineBookingEndpoint"
                             destination-name="${royal.airline.request.queue}"/>
```

We need to configure a Http server instance with a port, a request URI and the endpoint adapter. We define the JMS endpoint adapter to handle request as described. In Addition to the endpoint adapter we also need synchronous JMS message sender and receiver instances. That's it! We are able to receive Http request in order to simulate the RoyalAirline application. What is missing now? The test case definition itself.

51.1.4. The test case

The test case definition is also a Spring configuration file. Citrus offers a customized XML syntax to define a test case. This XML test defining language is supposed to be easy to understand and more specific to the domain we are dealing with. Next listing shows the whole test case definition. Keep in mind that a test case defines every step in the use case. So we define sending and receiving actions of the use case as described in the sequence diagram we saw earlier.

```
<?xml version="1.0" encoding="UTF-8"?>
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
             xmlns:spring="http://www.springframework.org/schema/beans"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://www.springframework.org/schema/beans
             http://www.springframework.org/schema/beans/spring-beans.xsd
             http://www.citrusframework.org/schema/testcase
             http://www.citrusframework.org/schema/testcase/citrus-testcase.xsd">
    <testcase name="FlightBookingTest">
        <meta-info>
            <author>Christoph Deppisch</author>
            <creationdate>2009-04-15</creationdate>
            <status>FINAL</status>
            <last-updated-by>Christoph Deppisch</last-updated-by>
            <last-updated-on>2009-04-15T00:00:00</last-updated-on>
        </meta-info>
        <description>
            Test flight booking service.
        </description>
        <variables>
            <variable name="correlationId"
                value="citrus:concat('Lx1x', 'citrus:randomNumber(10)')"/>
            <variable name="customerId"
                value="citrus:concat('Mx1x', citrus:randomNumber(10))"/>
        </variables>
        <actions>
            <send endpoint="travelAgencyBookingRequestEndpoint">
                <message>
                    <data>
                      <![CDATA[
                        <TravelBookingRequestMessage
                          xmlns="http://citrusframework.org/schemas/TravelAgency">
                          <correlationId>${correlationId}</correlationId>
                          <customer>
                            <id>${customerId}</id>
                            <firstname>John</firstname>
```

```
                <lastname>Doe</lastname>
              </customer>
              <flights>
                <flight>
                  <flightId>SM 1269</flightId>
                  <airline>SmartAirline</airline>
                  <fromAirport>MUC</fromAirport>
                  <toAirport>FRA</toAirport>
                  <date>2009-04-15</date>
                  <scheduledDeparture>11:55:00</scheduledDeparture>
                  <scheduledArrival>13:00:00</scheduledArrival>
                </flight>
                <flight>
                  <flightId>RA 1780</flightId>
                  <airline>RoyalAirline</airline>
                  <fromAirport>FRA</fromAirport>
                  <toAirport>HAM</toAirport>
                  <date>2009-04-15</date>
                  <scheduledDeparture>16:00:00</scheduledDeparture>
                  <scheduledArrival>17:10:00</scheduledArrival>
                </flight>
              </flights>
            </TravelBookingRequestMessage>
          ]]>
        </data>
    </message>
    <header>
        <element name="correlationId" value="${correlationId}"/>
    </header>
</send>

<receive endpoint="smartAirlineBookingRequestEndpoint">
    <message>
        <data>
          <![CDATA[
            <FlightBookingRequestMessage
              xmlns="http://citrusframework.org/schemas/AirlineSchema">
              <correlationId>${correlationId}</correlationId>
              <bookingId>???</bookingId>
              <customer>
                <id>${customerId}</id>
                <firstname>John</firstname>
                <lastname>Doe</lastname>
              </customer>
              <flight>
                <flightId>SM 1269</flightId>
                <airline>SmartAirline</airline>
                <fromAirport>MUC</fromAirport>
                <toAirport>FRA</toAirport>
                <date>2009-04-15</date>
                <scheduledDeparture>11:55:00</scheduledDeparture>
```

```
                        <scheduledArrival>13:00:00</scheduledArrival>
                      </flight>
                    </FlightBookingRequestMessage>
                  ]]>
                </data>
                <ignore path="//:FlightBookingRequestMessage/:bookingId"/>
            </message>
            <header>
                <element name="correlationId" value="${correlationId}"/>
            </header>
            <extract>
                <message path="//:FlightBookingRequestMessage/:bookingId"
                            variable="${smartAirlineBookingId}"/>
            </extract>
        </receive>

        <send endpoint="smartAirlineBookingResponseEndpoint">
            <message>
                <data>
                  <![CDATA[
                    <FlightBookingConfirmationMessage
                      xmlns="http://citrusframework.org/schemas/AirlineSchema">
                      <correlationId>${correlationId}</correlationId>
                      <bookingId>${smartAirlineBookingId}</bookingId>
                      <success>true</success>
                      <flight>
                        <flightId>SM 1269</flightId>
                        <airline>SmartAirline</airline>
                        <fromAirport>MUC</fromAirport>
                        <toAirport>FRA</toAirport>
                        <date>2009-04-15</date>
                        <scheduledDeparture>11:55:00</scheduledDeparture>
                        <scheduledArrival>13:00:00</scheduledArrival>
                      </flight>
                    </FlightBookingConfirmationMessage>
                  ]]>
                </data>
            </message>
            <header>
                <element name="correlationId" value="${correlationId}"/>
            </header>
        </send>

        <receive endpoint="royalAirlineBookingEndpoint">
            <message>
                <data>
                  <![CDATA[
                    <FlightBookingRequestMessage

xmlns="http://citrusframework.org/schemas/FlightBooking/AirlineSchema">
                      <correlationId>${correlationId}</correlationId>
```

```
                    <bookingId>???</bookingId>
                    <customer>
                        <id>${customerId}</id>
                        <firstname>John</firstname>
                        <lastname>Doe</lastname>
                    </customer>
                    <flight>
                      <flightId>RA 1780</flightId>
                      <airline>RoyalAirline</airline>
                      <fromAirport>FRA</fromAirport>
                      <toAirport>HAM</toAirport>
                      <date>2009-04-15</date>
                      <scheduledDeparture>16:00:00</scheduledDeparture>
                      <scheduledArrival>17:10:00</scheduledArrival>
                    </flight>
                  </FlightBookingRequestMessage>
                ]]>
            </data>
            <ignore path="//:FlightBookingRequestMessage/:bookingId"/>
        </message>
        <header>
            <element name="correlationId" value="${correlationId}"/>
        </header>
        <extract>
            <message path="//:FlightBookingRequestMessage/:bookingId"
                        variable="${royalAirlineBookingId}"/>
        </extract>
    </receive>

    <send endpoint="royalAirlineBookingEndpoint">
        <message>
            <data>
              <![CDATA[
                <FlightBookingConfirmationMessage
                  xmlns="http://citrusframework.org/schemas/AirlineSchema">
                  <correlationId>${correlationId}</correlationId>
                  <bookingId>${royalAirlineBookingId}</bookingId>
                  <success>true</success>
                  <flight>
                    <flightId>RA 1780</flightId>
                    <airline>RoyalAirline</airline>
                    <fromAirport>FRA</fromAirport>
                    <toAirport>HAM</toAirport>
                    <date>2009-04-15</date>
                    <scheduledDeparture>16:00:00</scheduledDeparture>
                    <scheduledArrival>17:10:00</scheduledArrival>
                  </flight>
                </FlightBookingConfirmationMessage>
              ]]>
            </data>
        </message>
```

```
                <header>
                    <element name="correlationid" value="${correlationId}"/>
                </header>
            </send>

            <receive endpoint="travelAgencyBookingResponseEndpoint">
                <message>
                    <data>
                      <![CDATA[
                        <TravelBookingResponseMessage
                          xmlns="http://citrusframework.org/schemas/TravelAgency">
                          <correlationId>${correlationId}</correlationId>
                          <success>true</success>
                          <flights>
                            <flight>
                              <flightId>SM 1269</flightId>
                              <airline>SmartAirline</airline>
                              <fromAirport>MUC</fromAirport>
                              <toAirport>FRA</toAirport>
                              <date>2009-04-15</date>
                              <scheduledDeparture>11:55:00</scheduledDeparture>
                              <scheduledArrival>13:00:00</scheduledArrival>
                            </flight>
                            <flight>
                              <flightId>RA 1780</flightId>
                              <airline>RoyalAirline</airline>
                              <fromAirport>FRA</fromAirport>
                              <toAirport>HAM</toAirport>
                              <date>2009-04-15</date>
                              <scheduledDeparture>16:00:00</scheduledDeparture>
                              <scheduledArrival>17:10:00</scheduledArrival>
                            </flight>
                          </flights>
                        </TravelBookingResponseMessage>
                      ]]>
                    </data>
                </message>
                <header>
                    <element name="correlationId" value="${correlationId}"/>
                </header>
            </receive>

        </actions>
    </testcase>
</spring:beans>
```

Similar to a sequence diagram the test case describes every step of the use case. At the very beginning the test case gets name and its meta information. Following with the variable values that are used all over the test. Here it is the correlationId and the customerId that are used as test variables. Inside message templates header values the variables are referenced several times in the

test

```
<correlationId>${correlationId}</correlationId>
<id>${customerId}</id>
```

The sending/receiving actions use a previously defined message sender/receiver. This is the link between test case and basic Spring configuration we have done before.

```
send endpoint="travelAgencyBookingRequestEndpoint"
```

The sending action chooses a message sender to actually send the message using a message transport (JMS, Http, SOAP, etc.). After sending this first "TravelBookingRequestMessage" request the test case expects the first "FlightBookingRequestMessage" message on the SmartAirline JMS destination. In case this message is not arriving in time the test will fail with errors. In positive case our FlightBookingService works well and the message arrives in time. The received message is validated against a defined expected message template. Only in case all content validation steps are successful the test continues with the action chain. And so the test case proceeds and works through the use case until every message is sent respectively received and validated. The use case is done automatically without human interaction. Citrus simulates all surrounding applications and provides detailed validation possibilities of messages.

Chapter 52. Appendix

Maven archetype

If you start from scratch or in case you would like to have Citrus operating in a separate Maven module you can use the Citrus Maven archetype to create a new Maven project. The archetype will setup a basic Citrus project structure with basic settings and files.

```
mvn archetype:generate -Dfilter=org.citrusframework.mvn:citrus

1: remote -> org.citrusframework.mvn:citrus-quickstart (Citrus quickstart project)
2: remote -> org.citrusframework.mvn:citrus-quickstart-jms (Citrus quickstart project
with JMS consumer and producer)
3: remote -> org.citrusframework.mvn:citrus-quickstart-soap (Citrus quickstart project
with SOAP client and producer)
Choose a number: 1

Define value for groupId: org.citrusframework.samples
Define value for artifactId: citrus-sample
Define value for version: 1.0-SNAPSHOT
Define value for package: org.citrusframework.samples
```

In the sample above we used the Citrus archetype available in Maven central repository. As the list of default archetypes available in Maven central is very long, it has been filtered for official Citrus archetypes.

After choosing the Citrus quickstart archetype you have to define several values for your project: the groupId, the artifactId, the package and the project version. After that we are done! Maven created a Citrus project structure for us which is ready for testing. You should see the following basic project folder structure.

```
citrus-sample
  |   + src
  |   |   + main
  |   |    |   + java
  |   |    |   + resources
  |   |   + test
  |   |    |   + java
  |   |    |   + resources
  pom.xml
```

The Citrus project is absolutely ready for testing. With Maven we can build, package, install and test our project right away without any adjustments. Try to execute the following commands:

```
mvn clean verify
mvn clean verify -Dit.test=MyFirstCitrusTest
```

If you need additional assistance in setting up a Citrus Maven project please visit our Maven setup tutorial on https://citrusframework.org/tutorials.html.