<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2.3. Start/stop Camel routes</td>
<td>24</td>
</tr>
<tr>
<td>7.2.4. Send messages via Camel</td>
<td>24</td>
</tr>
<tr>
<td>7.2.5. Receive messages via Camel</td>
<td>25</td>
</tr>
<tr>
<td>7.2.6. Define Camel exchanges</td>
<td>26</td>
</tr>
<tr>
<td>7.2.7. Basic Camel settings</td>
<td>27</td>
</tr>
<tr>
<td>7.2.8. Manage Camel resources</td>
<td>27</td>
</tr>
<tr>
<td>7.3. Apache Camel K steps</td>
<td>28</td>
</tr>
<tr>
<td>7.3.1. Create Camel K integrations</td>
<td>28</td>
</tr>
<tr>
<td>7.3.2. Load Camel K integrations</td>
<td>29</td>
</tr>
<tr>
<td>7.3.3. Delete Camel K integrations</td>
<td>29</td>
</tr>
<tr>
<td>7.3.4. Verify integration is running</td>
<td>29</td>
</tr>
<tr>
<td>7.3.5. Watch Camel K integration logs</td>
<td>30</td>
</tr>
<tr>
<td>7.3.6. Manage Camel K resources</td>
<td>30</td>
</tr>
<tr>
<td>7.4. Kamelet steps</td>
<td>30</td>
</tr>
<tr>
<td>7.4.1. Create Kamelets</td>
<td>31</td>
</tr>
<tr>
<td>7.4.2. Load Kamelets</td>
<td>32</td>
</tr>
<tr>
<td>7.4.3. Delete Kamelets</td>
<td>33</td>
</tr>
<tr>
<td>7.4.4. Verify Kamelet is available</td>
<td>33</td>
</tr>
<tr>
<td>7.5. KameletBinding steps</td>
<td>33</td>
</tr>
<tr>
<td>7.5.1. Create KameletBindings</td>
<td>33</td>
</tr>
<tr>
<td>7.5.2. Load KameletBindings</td>
<td>35</td>
</tr>
<tr>
<td>7.5.3. Delete KameletBindings</td>
<td>35</td>
</tr>
<tr>
<td>7.5.4. Verify KameletBinding is available</td>
<td>35</td>
</tr>
<tr>
<td>7.5.5. Manage Kamelet and KameletBinding resources</td>
<td>35</td>
</tr>
<tr>
<td>7.6. Groovy steps</td>
<td>36</td>
</tr>
<tr>
<td>7.6.1. Framework configuration</td>
<td>36</td>
</tr>
<tr>
<td>7.6.2. Endpoint configuration</td>
<td>39</td>
</tr>
<tr>
<td>7.6.3. Test actions</td>
<td>40</td>
</tr>
<tr>
<td>7.7. Http steps</td>
<td>41</td>
</tr>
<tr>
<td>7.7.1. Http client steps</td>
<td>42</td>
</tr>
<tr>
<td>7.7.2. Send Http requests</td>
<td>42</td>
</tr>
<tr>
<td>7.7.3. Send raw Http request data</td>
<td>44</td>
</tr>
<tr>
<td>7.7.4. Verify Http responses</td>
<td>45</td>
</tr>
<tr>
<td>7.7.5. Verify raw Http response data</td>
<td>46</td>
</tr>
<tr>
<td>7.7.6. Verify response using JsonPath</td>
<td>47</td>
</tr>
<tr>
<td>7.7.7. Http server steps</td>
<td>47</td>
</tr>
<tr>
<td>7.7.8. Http server port</td>
<td>48</td>
</tr>
<tr>
<td>7.7.9. Receive Http requests</td>
<td>49</td>
</tr>
<tr>
<td>7.7.10. Receive raw Http request data</td>
<td>50</td>
</tr>
<tr>
<td>7.7.11. Verify requests using JsonPath</td>
<td>51</td>
</tr>
<tr>
<td>7.7.12. Send Http responses</td>
<td>51</td>
</tr>
</tbody>
</table>
7.13. Open API steps

7.13.1. Load OpenAPI specifications
7.13.2. Invoke operations
7.13.3. Verify operation result
7.13.4. Request fork mode
7.13.5. Inbound/outbound data dictionaries

7.10. Kafka steps

7.10.1. Connection
7.10.2. Topic and endpoint configuration
7.10.3. Send Kafka events
7.10.4. Receive Kafka events
7.10.5. Special configuration

7.9. JMS steps

7.9.1. Connection factory
7.9.2. Destination and endpoint configuration
7.9.3. Send JMS messages
7.9.4. Receive JMS messages

7.8. JDBC steps

7.8.1. Connection configuration
7.8.2. SQL update
7.8.3. SQL query
7.8.4. Result set verification script

7.7. Other steps

7.7.14. Http health checks
7.7.13. Send raw Http response data
7.7.12. Receive events
7.7.11. Cleanup resources
Chapter 1. What is YAKS!?

YAKS is a framework to enable Cloud Native BDD testing on Kubernetes! Cloud Native here means that your tests execute as Kubernetes PODs.

As a user you can run tests by creating a Test custom resource on your favorite Kubernetes based cloud provider. Once the YAKS operator is installed it will listen for custom resources and automatically prepare a test runtime that runs the test as part of the cloud infrastructure.

Tests in YAKS follow the BDD (Behavior Driven Development) concept and represent feature specifications written in Gherkin syntax.

As a framework YAKS provides a set of predefined Cucumber steps which help you to connect with different messaging transports (Http REST, JMS, Kafka, Knative eventing) and verify message data with assertions on the header and body content.

YAKS adds its functionality on top of on Citrus for connecting to different endpoints as a client and/or server.
Chapter 2. Getting started

Assuming you have a Kubernetes playground and that you are connected to a namespace on that cluster just write a `helloworld.feature` BDD file with the following content:

```
Feature: Hello

Scenario: Print hello message
  Given print 'Hello from YAKS!'
```

You can then execute the following command using the YAKS CLI tool:

```
yaks test helloworld.feature
```

This runs the test immediately on the current namespace in your connected Kubernetes cluster. Nothing else is needed.

Continue reading the documentation and learn how to install and get started working with YAKS.
Chapter 3. Installation

YAKS directly runs the test as part of a cloud infrastructure by leveraging the Operator SDK and the concept of custom resources in Kubernetes.

As a user you need to enable YAKS on your infrastructure by installing the operator and creating the required custom resources and roles.

3.1. Requirements

You need access to a Kubernetes or Openshift cluster in order to use YAKS. You have different options to setup/use a Kubernetes or OpenShift cluster.

- Minikube
- Minishift
- Red Hat CodeReady Containers (CRC)
- Google Kubernetes Engine (GKE)
- OpenShift
- IBM Kubernetes Service (IKS)

Obviously the cluster will be the place where the tests will be executed and probably also the place where to run the SUT (System Under Test).

For setting up roles and custom resources you may need to have administrative rights on that cluster.

3.2. Windows prerequisite

For full support of Yaks on Windows please enable "Windows Subsystem for Linux". You can do it manually by heading to Control Panel > Programs > Turn Windows Features On or Off and checking "Windows Subsystem for Linux". Or you can simply execute this command in powershell:

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux
```

This action requires a full reboot of the system.

3.3. Operator install

The YAKS operator will listen for new test resources in order to run those on the cloud infrastructure. The operators is in charge of preparing a proper runtime for each test and it will reconcile the status of a test.

The easiest way to getting started with the YAKS operator installation is to use the YAKS CLI. You can download the CLI from the release page where you will find installation archives for different operating systems.
Download and decompress the archive. The archive holds a binary that will help you to install YAKS and run the tests. To install the `yaks` binary, just make it runnable and move it to a location in your `$PATH`, e.g. on linux:

```bash
# Make executable and move to usr/local/bin
$ chmod a+x yaks-${project.version}-linux-64bit
$ mv yaks-${project.version}-linux-64bit /usr/local/bin/yaks

# Alternatively, set a symbolic link to "yaks"
$ mv yaks-${project.version}-linux-64bit yaks
$ ln -s $(pwd)/yaks /usr/local/bin
```

Once you have the `yaks` CLI available, log into your cluster using the standard `oc` (OpenShift) or `kubectl` (Kubernetes) client tool.

Once you are properly connected to your cluster execute the following command to install YAKS:

```bash
yaks install
```

This will install and run the YAKS operator in the current namespace.

You can specify the target namespace where to run the operator with a `--namespace` option:

```bash
yaks install -n kube-operators
```

The namespace must available on the cluster before running the install command. If the namespace has not been created, yet you can create it with the following command:

```bash
kubectl create namespace kube-operators
```

If not already configured, the command will also setup the YAKS custom resource definitions and roles on the cluster (in this case, the user needs cluster-admin permissions).

Custom Resource Definitions (CRD) are cluster-wide objects and you need admin rights to install them. Fortunately, this operation can be done once per cluster. So, if the `yaks install` operation fails, you'll be asked to repeat it when logged as admin. For Minishift, this means executing `oc login -u system:admin` then `yaks install --cluster-setup` only for the first-time installation.

### 3.3.1. Global mode

By default, the installation is using a global operator mode. This means that the operator only lives once in your cluster watching for tests in all namespaces. A global operator uses cluster-roles in order to manage tests in all namespaces.
When running on OpenShift the default namespace for global operators is `openshift-operators` (it is available by default). Be sure to select this namespace when installing YAKS in the global mode:

```
yaks install -n openshift-operators
```

### 3.3.2. Namespaced mode

You can disable the *global* mode with a CLI setting when running the `install` command:

```
yaks install --global=false
```

In the non global *namespaced* mode the YAKS operator will only have the rights to create new tests in the same namespace as it is running on. The operator will only watch for tests created in that the very same namespace.

Which mode to choose depends on your very specific needs. When you expect to have many tests in different namespaces that will be recreated on a regular basis you may choose the global operator mode because you will not have to reinstall the operator many times.

If you expect to have all tests in a single namespace or if you do not want to use cluster-wide operator permissions for some reason you may want to switch the namespaced mode.

Please also have a look at the *temporary namespaces* section in this guide to make a decision on operator modes.

### 3.4. Verify installation

You can verify the installation by retrieving the custom resource definition provided in YAKS:

```
kubectl get customresourcedefinitions -l app=yaks
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>CREATED AT</th>
</tr>
</thead>
<tbody>
<tr>
<td>tests.yaks.citrusframework.org</td>
<td>2020-11-01T00:00:00Z</td>
</tr>
</tbody>
</table>

The following command will list all tests in your namespace:

```
kubectl get tests
```
<table>
<thead>
<tr>
<th>NAME</th>
<th>PHASE</th>
<th>TOTAL</th>
<th>PASSED</th>
<th>FAILED</th>
<th>SKIPPED</th>
<th>ERRORS</th>
</tr>
</thead>
<tbody>
<tr>
<td>helloworld</td>
<td>Passed</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 4. Running

After completing and verifying the installation you can start running some tests.

You should be connected to your Kubernetes cluster and you should have the YAKS CLI tool available on your machine.

You can verify the proper YAKS CLI setup with:

```
yaks version
```

This will print the YAKS version to the output.

```
YAKS ${project.version}
```

You are now ready to run a first BDD test on the cluster. As a sample create a new feature file that prints some message to the test output.

```
helloworld.feature
```

```
Feature: Hello

    Scenario: Print hello message
        Given print 'Hello from YAKS!'
```

You just need this single file to run the test on the cluster.

```
yaks test helloworld.feature
```

You will be provided with the log output of the test and see the results:
By default, log levels are set to a minimum so you are not bothered with too much boilerplate output. You can increase log levels with the command line option `--logger`.
The logging configuration section in this guide gives you some more details on this topic.

You are now ready to explore the different steps that you can use in a feature file in order to connect with various messaging transports as part of your test.

4.1. Status monitoring

As you run tests with YAKS you add tests to the current namespace. You can review the test status and monitor the test results with the default Kubernetes CLI tool.

The following command will list all tests in your namespace:

```
kubectl get tests
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>PHASE</th>
<th>TOTAL</th>
<th>PASSED</th>
<th>FAILED</th>
<th>SKIPPED</th>
<th>ERRORS</th>
</tr>
</thead>
<tbody>
<tr>
<td>helloworld</td>
<td>Passed</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

The overview includes the test outcome and outline the number of total scenarios that have been executed and the test results for these scenarios (skipped, passed or failed). When a scenario has been failing the error message is also displayed in this overview.

You can get more details of a single test with:

```
kubectl get test helloworld -o yaml
```

This gets you the complete test details as a YAML file. You can then review status and detailed error messages.

Find out more about the individual test results and how to get reports (e.g. JUnit) from a test run in the section about reporting.
Chapter 5. Command line interface (yaks)

The YAKS command line interface (yaks) is the main entry point for installing the operator and for running tests on a Kubernetes cluster.

Releases of the CLI are available on:

- Github Releases: https://github.com/citrusframework/yaks/releases
- Homebrew (Mac and Linux): https://formulae.brew.sh/formula/yaks

5.1. Available Commands

Some of the most used commands are:

Table 1. Useful Commands

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>help</td>
<td>Obtain the full list of available</td>
<td>yaks help</td>
</tr>
<tr>
<td></td>
<td>commands</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Generate</td>
<td></td>
</tr>
<tr>
<td>completion</td>
<td>Generates completion scripts</td>
<td>yaks completion</td>
</tr>
<tr>
<td></td>
<td>(bash, zsh)</td>
<td></td>
</tr>
<tr>
<td>install</td>
<td>Install YAKS operator and setup</td>
<td>yaks install</td>
</tr>
<tr>
<td></td>
<td>cluster (roles, CRDs)</td>
<td></td>
</tr>
<tr>
<td>test</td>
<td>Run a test on Kubernetes</td>
<td>yaks test helloworld.feature</td>
</tr>
<tr>
<td>report</td>
<td>Fetches and generates reports</td>
<td>yaks report --fetch -o</td>
</tr>
<tr>
<td></td>
<td>from test results</td>
<td>junit</td>
</tr>
<tr>
<td>upload</td>
<td>Upload custom artifacts (steps,</td>
<td>yaks upload ./steps/my-cu</td>
</tr>
<tr>
<td></td>
<td>extensions) to Minio storage</td>
<td>stom-steps</td>
</tr>
<tr>
<td>uninstall</td>
<td>Remove YAKS (operator, roles,</td>
<td>yaks uninstall</td>
</tr>
<tr>
<td></td>
<td>CRDs, ...) from the cluster</td>
<td></td>
</tr>
<tr>
<td>version</td>
<td>Print current YAKS version</td>
<td>yaks version</td>
</tr>
</tbody>
</table>

The list above is not the full list of available commands. You can run yaks help to obtain the full list. Each sub-command also takes --help as parameter to output more information on that specific command usage.

Overall help

yaks help
YAKS is a platform to enable Cloud Native BDD testing on Kubernetes.

Usage:

`yaks [command]`

Available Commands:

- `completion` Generates completion scripts
- `help` Help about any command
- `install` Installs YAKS on a Kubernetes cluster
- `report` Generate test report from last test run
- `test` Execute a test on Kubernetes
- `uninstall` Uninstall YAKS from a Kubernetes cluster
- `upload` Upload a local test artifact to the cluster
- `version` Display version information

Flags:

- `--config string` Path to the config file to use for CLI requests
- `-h, --help` help for yaks
- `-n, --namespace string` Namespace to use for all operations

Use "yaks [command] --help" for more information about a command.

---

5.2. install

The command `install` performs the YAKS installation on a target cluster. The command has two separate install steps:

1. Setup cluster resources (CRDs, roles, rolebindings)
2. Install YAKS operator to current namespace (or to the provided namespace in settings)

5.3. test

5.4. report

5.5. upload

5.6. uninstall

5.7. version
Chapter 6. Configuration

There are several runtime options that you can set in order to configure which tests to run for instance. Each test directory can have its own `yaks-config.yaml` configuration file that holds the runtime options for this specific test suite.

```yaml
config:
  runtime:
    cucumber:
      tags:
        - "not @ignored"
      glue:
        - "org.citrusframework.yaks"
        - "com.company.steps.custom"
```

The sample above uses different runtime options for Cucumber to specify a tag filter and some custom glue packages that should be loaded. The given runtime options will be set as environment variables in the YAKS runtime pod.

You can also specify the Cucumber options that get passed to the Cucumber runtime.

```yaml
config:
  runtime:
    cucumber:
      options: "--strict --monochrome --glue org.citrusframework.yaks"
```

Also we can make use of command line options when using the `yaks` binary.

```
yaks test hello-world.feature --tag @regression --glue org.citrusframework.yaks
```

### 6.1. Runtime dependencies

The YAKS testing framework provides a base runtime image that holds all required libraries and artifacts to execute tests. You may need to add additional runtime dependencies though in order to extend the framework capabilities.

For instance when using a Camel route in your test you may need to add additional Camel components that are not part in the basic YAKS runtime (e.g. camel-groovy). You can add the runtime dependency to the YAKS runtime image in multiple ways:

#### 6.1.1. Cucumber tags

You can simply add a tag to your BDD feature specification in order to declare a runtime dependency for your test.
Feature: Camel route testing

Background:
Given Camel route hello.xml

```xml
<route>
  <from uri="direct:hello"/>
  <filter>
    <groovy>request.body.startsWith('Hello')</groovy>
    <to uri="log:org.citrusframework.yaks.camel?level=INFO"/>
  </filter>
  <split>
    <tokenize token="/"/>
    <to uri="seda:tokens"/>
  </split>
</route>
```

Scenario: Hello route
When send to route direct:hello body: Hello Camel!
And receive from route seda:tokens body: Hello
And receive from route seda:tokens body: Camel!

The given Camel route uses the groovy language support and this is not part in the basic YAKS runtime image. So we add the tag `@require('org.apache.camel:camel-groovy:@camel.version@')`. This tag will load the Maven dependency at runtime before the test is executed in the YAKS runtime image.

Note that you have to provide proper Maven artifact coordinates with proper groupId, artifactId and version. You can make use of version properties for these versions available in the YAKS base image:

- citrus.version
- camel.version
- spring.version
- cucumber.version

6.1.2. System property or environment setting

You can add dependencies also by specifying the dependencies as command line parameter when running the test via yaks CLI.

```
yaks test --dependency org.apache.camel:camel-groovy:@camel.version@ camel-route.feature
```

This will add a environment setting in the YAKS runtime container and the dependency will be
loaded automatically at runtime.

6.1.3. Property file

YAKS supports adding runtime dependency information to a property file called `yaks.properties`. The dependency is added through Maven coordinates in the property file using a common property key prefix `yaks.dependency`.

```properties
# include these dependencies
yaks.dependency.foo=org.foo:foo-artifact:1.0.0
yaks.dependency.bar=org.bar:bar-artifact:1.5.0
```

You can add the property file when running the test via `yaks` CLI like follows:

```bash
yaks test --settings yaks.properties camel-route.feature
```

6.1.4. YAKS configuration file

When more dependencies are required to run a test you may consider to add a configuration file as `.yaml` or `.json`.

The configuration file is able to declare multiple dependencies:

```json

dependencies:
  - groupId: org.foo
    artifactId: foo-artifact
    version: 1.0.0
  - groupId: org.bar
    artifactId: bar-artifact
    version: 1.5.0
```

```json
{
    "dependencies": [
        {
            "groupId": "org.foo",
            "artifactId": "foo-artifact",
            "version": "1.0.0"
        },
        {
            "groupId": "org.bar",
            "artifactId": "bar-artifact",
            "version": "1.5.0"
        }
    ]
}
```
You can add the configuration file when running the test via `yaks` CLI like follows:

```
yaks test --settings yaks.settings.yaml camel-route.feature
```

### 6.2. Maven repositories

When adding custom runtime dependencies those artifacts might not be available on the public central Maven repository. Instead you may need to add a custom repository that holds your artifacts.

You can do this with several configuration options:

#### 6.2.1. System property or environment setting

You can add repositories also by specifying the repositories as command line parameter when running the test via `yaks` CLI.

```
yaks test --maven-repository jboss-ea=https://repository.jboss.org/nexus/content/groups/ea/ my.feature
```

This will add an environment setting in the YAKS runtime container and the repository will be added to the Maven runtime project model.

#### 6.2.2. Property file

YAKS supports adding Maven repository information to a property file called `yaks.properties`. The dependency is added through Maven repository id and url in the property file using a common property key prefix `yaks.repository`.

```
# Maven repositories
yaks.repository.central=https://repo.maven.apache.org/maven2/
yaks.repository.jboss-ea=https://repository.jboss.org/nexus/content/groups/ea/
```

You can add the property file when running the test via `yaks` CLI like follows:

```
yaks test --settings yaks.properties my.feature
```

#### 6.2.3. YAKS configuration file

More complex repository configuration might require to add a configuration file as `.yaml` or `.json`.

The configuration file is able to declare multiple repositories:
You can add the configuration file when running the test via `yaks` CLI like follows:

```
yaks test --settings yaks.settings.yaml my.feature
```

### 6.3. Using secrets

Tests usually need to use credentials and connection URLs in order to connect to infrastructure components and services. This might be sensitive data that should not go into the test configuration directly as hardcoded value. You should rather load the credentials from a secret volume source.
To use the implicit configuration via secrets, we first need to create a configuration file holding the properties of a named configuration.

```
mysecret.properties

# Only configuration related to the "mysecret" named config
database.url=jdbc:postgresql://syndesis-db:5432/sampledb
database.user=admin
database.password=special
```

We can create a secret from that file and label it so that it will be picked up automatically by the YAKS operator:

```
# Create the secret from the property file
kubectl create secret generic my-secret --from-file=mysecret.properties
```

Once the secret is created you can bind it to tests by their name. Given the test `my-test.feature` you can bind the secret to the test by adding a label as follows:

```
# Bind secret to the "my-test" test case
kubectl label secret my-secret yaks.citrusframework.org/test=my-test
```

For multiple secrets and variants of secrets on different environments (e.g. dev, test, staging) you can add a secret id and label that one explicitly in addition to the test name.

```
# Bind secret to the named configuration "staging" of the "my-test" test case
kubectl label secret my-secret yaks.citrusframework.org/test=my-test
yaks.citrusframework.org/test.configuration=staging
```

With that in place you just need to set the secret id in your `yaks-config.yaml` for that test.

```
yaks-config.yaml

config:
  runtime:
    secret: staging
```

You can now write a test and use the secret properties as normal test variables:
**Feature:** JDBC API

**Background:**

**Given** Database connection

<table>
<thead>
<tr>
<th>url</th>
<th>${database.url}</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>${database.user}</td>
</tr>
<tr>
<td>password</td>
<td>${database.password}</td>
</tr>
</tbody>
</table>
Chapter 7. Steps

Each line in a BDD feature file is backed by a step implementation that covers the actual runtime logic executed. YAKS provides a set of step implementations that you can just out-of-the-box use in your feature file.

See the following step implementations that enable you to cover various areas of messaging and integration testing.

7.1. Standard steps

The standard steps in YAKS provide a lot of basic functionality that you can just use in your feature files. The functionality is shipped as predefined steps that you add to a feature as you write your test.

Most of the standard steps do leverage capabilities of the underlying test framework Citrus such as creating test variables or printing messages to the log output.

7.1.1. Create variables

Test variables represent the fundamental concept to own test data throughout your test. Once a variable has been created you can reference its value in many places in YAKS and Citrus. You can add a new identifier as a variable and reference its value in many places such as message headers, body content, SQL statements and many more.

@Given("^variable {name} is "{value}"$")

```plaintext
Given variable orderId is "1001"
```

This will create the variable `orderId` in the current test context. All subsequent steps and operations may reference the variable with the expression `${orderId}`. Citrus makes sure to replace the variable placeholder with its actual value before sending out messages and before validating incoming messages. As already mentioned you can use the variable placeholder expression in many places such as message headers and body content:

Variable placeholder in a Json payload

```json
{
  "id": "${orderId}",
  "name": "Watermelon",
  "amount": 10
}
```

You can create multiple variables in one single step using:
7.1.2. Log steps

Logging a message to the output can be helpful in terms of debugging and/or to give information about the context of an operation.

YAKS provides following steps to add log output:

@Then("^(?:log|print) '(text)'$")

Then print 'YAKS provides Cloud native BDD testing!'
And log 'YAKS rocks!'

The steps are printing log messages to the output using INFO level. The text that is printed supports test variables and functions. All placeholders will be replaced before logging.

You can also use multiline log messages as shown in the next example.

@Then("^^(?:log|print)$")

Given print

###

Hello users!

YAKS provides Cloud Native BDD testing on Kubernetes!

###

7.1.3. Sleep

The sleep step lets the test run wait for a given amount of time (in milliseconds). During the sleep no action will be performed and the subsequent steps are postponed respectively.

@Then("^sleep$")

Then sleep

The above step performs a sleep with the default time of 5000 milliseconds.

You can also give a time in milliseconds to sleep.

@Then("^sleep (time) ms$")

Then sleep 2500 ms
The step receives a numeric parameter that represents the amount of time (in milliseconds) to wait.

The Citrus framework also provides a set of BDD step implementations that you can use in a feature file. Read more about the available steps (e.g. for connecting with Selenium) in the official Citrus documentation on BDD testing.

### 7.2. Apache Camel steps

Apache Camel is a very popular enterprise integration library that provides a huge set of ready to use components and endpoints for you to connect with different messaging transports. Also many data formats are supported in Camel so you will be able to incorporate with almost any software interface exchanging data over the wire.

YAKS adds steps to use Apache Camel as part of a test. You are able to send and receive messages with Camel components and make use of the enterprise integration patterns and data formats implemented in Apache Camel.

#### 7.2.1. Create Camel context

The Camel context is a central place to add routes and manage Camel capabilities and services. You can start a new default (empty) Camel context using the following step.

```java
@Given("^(?:Default|New) Camel context$")
```

**Given Default Camel context**

This will setup and start a new Camel context as part of the current test scenario. You can now create and add routes to this context. In case you have special configuration and/or some default routes that you need to initialize as part of the context you can provide a Camel Spring bean configuration in the Camel context step.
Given New Spring Camel context

```xml
<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
                           http://camel.apache.org/schema/spring
                           http://camel.apache.org/schema/spring/camel-spring.xsd">
    <camelContext id="helloContext" xmlns="http://camel.apache.org/schema/spring">
        <route id="helloRoute">
            <from uri="direct:hello"/>
            <to uri="log:org.citrusframework.yaks.camel?level=INFO"/>
            <split>
                <tokenize token="/"/>
                <to uri="seda:tokens"/>
            </split>
        </route>
    </camelContext>
</beans>
```

7.2.2. Create Camel routes

In Apache Camel you need to create routes in order to start producing/consuming data from endpoints. The routes can be defined in XML or Groovy.

@Given("^Camel route {name}\.xml$")

```groovy
Given Camel route hello.xml

```xml
<route>
    <from uri="direct:hello"/>
    <to uri="log:org.citrusframework.yaks.camel?level=INFO"/>
    <split>
        <tokenize token="/"/>
        <to uri="seda:tokens"/>
    </split>
</route>
```

In addition to XML route definitions YAKS also supports the Groovy DSL.
The above steps create the Camel routes and automatically starts them in the current context. The given routes start to consume messages from the endpoint `direct:hello`.

### 7.2.3. Start/stop Camel routes

We are able to explicitly start and stop routes in the current context.

@Then("^start Camel route {name}\$")

Then start Camel route {name}

The given name references a route in the current Camel context. This starts the route and consumes messages from the endpoint URI.

@Then("^stop Camel route {name}\$")

Then stop Camel route {name}

After stopping a route the route will not consume any messages on the given endpoint URI.

In case a Camel route is not needed anymore you can also remove it from the current Camel context.

@Then("^remove Camel route {name}\$")

Then remove Camel route {name}

### 7.2.4. Send messages via Camel

We can send exchanges using any Camel endpoint URI. The endpoint URI can point to an external component or to a route in the current Camel context and trigger its processing logic. The exchange body is given as single or multiline body content.

@When("^send Camel exchange to\("{endpoint_uri}"\) with body: {body}\$")

When send Camel exchange to("direct:hello") with body: Hello Camel!
The step sends an exchange with the body *Hello Camel!*. You can also use multiline body content with the following step:

```java
@When("^send Camel exchange to\("{endpoint_uri}"\) with body\"
     "Hello Camel!
     This is a multiline content!
     \""")
```

In addition to a body content the Camel exchange also defines a set of message headers. You can use a data table to specify message headers when sending a message.

```java
@When("^send Camel exchange to\("{endpoint_uri}"\) with body and headers: {body}$")
```

<table>
<thead>
<tr>
<th>id</th>
<th>1234</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation</td>
<td>sayHello</td>
</tr>
</tbody>
</table>

### 7.2.5. Receive messages via Camel

The YAKS test is able to receive messages from a Camel endpoint URI in order to verify the message content (header and body) with an expected control message.

Once the message is received YAKS makes use of the powerful message validation capabilities of Citrus to make sure that the content is as expected.

```java
@When("^receive Camel exchange from\("{endpoint_uri}"\) with body: {body}$")
```

The step receives an exchange from the endpoint URI `seda:tokens` and verifies the body to be equal to *Hello*. See the next example on how to validate a multiline message body content.

```java
@When("^receive Camel exchange from\("{endpoint_uri}"\) with body$")
```

```java
{  
  "message": "Hello Camel!"
}
```

We can also verify a set of message headers that must be present on the received exchange. Once again we use a data table to define the message headers. This time we provide expected message header values.
When receive Camel exchange from("seda:tokens") with body and headers: Hello
| id         | 1234     |
| operation  | sayHello |

### 7.2.6. Define Camel exchanges

In the previous steps we have seen how to send and receive messages to and from Camel endpoint URIs. We have used the exchange body and header in a single step so far.

In some cases it might be a better option to use multiple steps for defining the complete exchange data upfront. The actual send/receive operation then takes place in a separate step.

The following examples should clarify the usage.

@Given("Camel exchange message header {name}="{value}"$")

```
Camel exchange message header {name}="{value}"
```

This sets a message header on the exchange. We can also use a data table to set multiple headers in one single step:

@Given("Camel exchange message headers$")

```
Camel exchange message headers
| id   | 1234   |
| operation | sayHello |
```

Then we can also set the body in another step.

@Given("Camel exchange body$")

```
Camel exchange body: Hello Camel!
```

Multiline body content is also supported.

@Given("Camel exchange body$")

```
""
{
    "message": "Hello Camel!"
}
""
```

Now that we have specified the exchange headers and body content we can send or receive that
specific exchange in a separate step.

@When("^send Camel exchange to\("{endpoint_uri}"\)$

```java
send Camel exchange to("{endpoint_uri}")
```

@When("^receive Camel exchange from\("{endpoint_uri}"\)$

```java
receive Camel exchange from("{endpoint_uri}")
```

In the previous section we have covered a 2nd approach to send and receive messages with Apache Camel. You specify the exchange in multiple steps first and then send/receive the exchange to/from and endpoint URI in a separate step.

7.2.7. Basic Camel settings

@Given("^Camel consumer timeout is {time}(?: ms| milliseconds)$

```java
Given Camel consumer timeout is {time} milliseconds
```

Sets the default timeout for all Camel components that consume data from messaging transports. After that time the test will fail with a timeout exception when no message has been received.

7.2.8. Manage Camel resources

The Apache Camel steps are able to create resources such as routes. By default these resources get removed automatically after the test scenario.

The auto removal of Camel resources can be turned off with the following step.

@Given("^Disable auto removal of Camel resources$")

```java
Given Disable auto removal of Camel resources
```

Usually this step is a Background step for all scenarios in a feature file. This way multiple scenarios can work on the very same Camel resources and share integrations.

There is also a separate step to explicitly enable the auto removal.

@Given("^Enable auto removal of Camel resources$")

```java
Given Enable auto removal of Camel resources
```

By default, all Camel resources are automatically removed after each scenario.
7.3. Apache Camel K steps

Apache Camel K is a lightweight integration framework built from Apache Camel that runs natively on Kubernetes and is specifically designed for serverless and microservice architectures.

Users of Camel K can instantly run integration code written in Camel DSL on their preferred cloud (Kubernetes or OpenShift).

If the subject under test is a Camel K integration, you can leverage the YAKS Camel K bindings that provide useful steps for managing Camel K integrations.

Working with Camel K integrations

```
@Given("^(?:create|new) Camel-K integration {name}.{type}$")
create Camel-K integration {name}.groovy

<<Camel DSL>>

The YAKS framework provides the Camel K extension library by default. You can create a new Camel K integration and check the status of the integration (e.g. running).

The following sections describe the available Camel K steps in detail.

7.3.1. Create Camel K integrations

```
@Given("^(?:create|new) Camel-K integration {name}.{type}$")
create Camel-K integration {name}.groovy

<<Camel DSL>>

```

```
@Given("^(?:create|new) Camel-K integration {name}.{type} with configuration:$")
create Camel-K integration {name}.groovy with configuration:

| dependencies | mvn:org.foo:foo:1.0,mvn:org.bar:bar:0.9 |
| traits       | quarkus.native=true,quarkus.enabled=true,route.enabled=true |
| properties   | foo.key=value,bar.key=value |
| source       | <<Camel DSL>> |

```

You can add optional configurations to the Camel K integration such as dependencies, traits and properties.

```
**Source**
The route DSL as source for the Camel K integration.

**Dependencies**
List of Maven coordinates that will be added to the integration runtime as a library.

**Traits**
List of trait configuration that will be added to the integration spec. Each trait configuration value must be in the format traitname.key=value.

**Properties**
List of property bindings added to the integration. Each value must be in the format key=value.

### 7.3.2. Load Camel K integrations

@Given("^load Camel-K integration {name}.{type}$")

**Given** load Camel-K integration {name}.groovy

Loads the file {name}.groovy as a Camel K integration.

### 7.3.3. Delete Camel K integrations

@Given("^delete Camel-K integration {name}$")

**Given** delete Camel-K integration {name}

Deletes the Camel K integration with given {name}.

### 7.3.4. Verify integration is running

@Given("^Camel-K integration {name} is running$")

**Given** Camel-K integration {name} is running

Checks that the Camel K integration with given {name} is in state running and that the number of replicas is > 0. The step polls the state of the integration for a given amount of attempts with a given delay between attempts. You can adjust the polling settings with:

**Given Camel-K resource polling configuration**

| maxAttempts | 10 |
| delayBetweenAttempts | 1000 |
7.3.5. Watch Camel K integration logs

@Given("^Camel-K integration {name} should print (.*)$"")

Given Camel-K integration {name} should print {log-message}

Watches the log output of a Camel K integration and waits for given {log-message} to be present in the logs. The step polls the logs for a given amount of time. You can adjust the polling configuration with:

@Given Camel-K resource polling configuration

Given Camel-K resource polling configuration

| maxAttempts | 10   |
| delayBetweenAttempts | 1000 |

You can also wait for a log message to not be present in the output. Just use this step:

@Given("^Camel-K integration {name} should not print (.*)$"")

Given Camel-K integration {name} should not print {log-message}

7.3.6. Manage Camel K resources

The Camel K steps are able to create resources such as integrations. By default these resources get removed automatically after the test scenario.

The auto removal of Camel K resources can be turned off with the following step.

@Given("^Disable auto removal of Camel-K resources$"")

Given Disable auto removal of Camel-K resources

Usually this step is a Background step for all scenarios in a feature file. This way multiple scenarios can work on the very same Camel K resources and share integrations.

There is also a separate step to explicitly enable the auto removal.

@Given("^Enable auto removal of Camel-K resources$"")

Given Enable auto removal of Camel-K resources

By default, all Camel K resources are automatically removed after each scenario.

7.4. Kamelet steps

Kamelets are a form of predefined Camel route templates implemented in Camel K. Usually a
Kamelet encapsulates a certain functionality (e.g. send messages to an endpoint). Additionally, Kamelets define a set of properties that the user needs to provide when using the Kamelet.

YAKS provides steps to manage Kamelets.

### 7.4.1. Create Kamelets

A Kamelet defines a set of properties and specifications that you can set with separate steps in your feature. Each of the following steps set a specific property on the Kamelet. Once you are done with the Kamelet specification you are able to create the Kamelet in the current namespace.

First of all you can specify the media type of the available slots (in, out and error) in the Kamelet.

```java
@Given("^Kamelet type (in|out|error)\(?:=\"{mediaType}\"\)"

```Given Kamelet type in="{mediaType}"

The Kamelet can use a title that you set with the following step.

```java
@Given("^Kamelet title "{title}""

```Given Kamelet title "{title}"

Each flow uses an endpoint uri and defines a set of steps that get called when the Kamelet processing takes place. The following step defines a flow on the current Kamelet.

```java
@Given("^Kamelet flow$"

```Given Kamelet flow

```json
"from:
  uri: timer:tick

parameters:
  period: ":property:period"

steps:
- set-body:
    constant: "{{message}}"
  to: "kamelet:sink"

```The flow uses two properties {{message}} and {{period}}. These placeholders need to be provided by the Kamelet user. The next step defines the property `message` in detail:
The property receives specification such as type, required and an example. In addition to the example you can set a default value for the property.

In addition to using a flow on the Kamelet you can add multiple sources to the Kamelet.

The previous steps defined all properties and Kamelet specifications so now you are ready to create the Kamelet in the current namespace.

The Kamelet requires a unique name. Creating a Kamelet means that a new custom resource of type Kamelet is created. As a variation you can also set the flow when creating the Kamelet.

This creates the Kamelet in the current namespace.

### 7.4.2. Load Kamelets

You can create new Kamelets by giving the complete specification in an external YAML file. The step loads the file content and creates the Kamelet in the current namespace.
Loads the file `{name}.kamelet.yaml` as a Kamelet. At the moment only `kamelet.yaml` source file extension is supported.

### 7.4.3. Delete Kamelets

```java
@Given("^delete Kamelet {name}$")
```

<table>
<thead>
<tr>
<th>Given delete Kamelet {name}</th>
</tr>
</thead>
</table>

Deletes the Kamelet with given `{name}` from the current namespace.

### 7.4.4. Verify Kamelet is available

```java
@Given("^Kamelet {name} is available$")
```

<table>
<thead>
<tr>
<th>Given Kamelet {name} is available$</th>
</tr>
</thead>
</table>

Verifies that the Kamelet custom resource is available in the current namespace.

### 7.5. KameletBinding steps

You can bind a Kamelet as a source to a sink. This concept is described with KameletBindings. YAKS as a framework is able to create and verify KameletBindings in combination with Kamelets.

#### 7.5.1. Create KameletBindings

YAKS provides multiple steps that bind a Kamelet source to a sink. The binding is going to forward all messages processed by the source to the sink.

##### 7.5.1.1. Bind to Http URI

```java
@Given("^bind Kamelet {kamelet} to uri {uri}$")
```

<table>
<thead>
<tr>
<th>Given bind Kamelet {name} to uri {uri}</th>
</tr>
</thead>
</table>

This defines the KameletBinding with the given Kamelet name as source to the given Http URI as a sink.

##### 7.5.1.2. Bind to Kafka topic

You can bind a Kamelet source to a Kafka topic sink. All messages will be forwarded to the topic.

```java
@Given("^bind Kamelet {kamelet} to Kafka topic {topic}$")
```

<table>
<thead>
<tr>
<th>Given bind Kamelet {kamelet} to Kafka topic {topic}</th>
</tr>
</thead>
</table>
7.5.1.3. Bind to Knative channel

Channels are part of the eventing in Knative. Similar to topics in Kafka the channels hold messages for subscribers.

@Given("^bind Kamelet {kamelet} to Knative channel {channel}$")

| Given bind Kamelet {kamelet} to Knative channel {channel} |

Channels can be backed with different implementations. You can explicitly set the channel type to use in the binding.

@Given("^bind Kamelet {kamelet} to Knative channel {channel} of kind {kind}$")

| Given bind Kamelet {kamelet} to Knative channel {channel} of kind {kind} |

7.5.1.4. Specify source/sink properties

The KameletBinding may need to specify properties for source and sink. These properties are defined in the Kamelet source specifications for instance.

You can set properties with values in the following step:

@Given("^KameletBinding source properties$")

| Given KameletBinding source properties | {property} | {value} |

The Kamelet source that we have used in the examples above has defined a property `message`. So you can set the property on the binding as follows.

| Given KameletBinding source properties | message | "Hello world" |

The same approach applies to sink properties.

@Given("^KameletBinding sink properties$")

| Given KameletBinding sink properties | {property} | {value} |

7.5.1.5. Create the binding

The previous steps have defined source and sink of the KameletBinding specification. Now you are ready to create the KameletBinding in the current namespace.
The KameletBinding receives a unique `name` and uses the previously specified source and sink. Creating a KameletBinding means that a new custom resource of type KameletBinding is created in the current namespace.

### 7.5.2. Load KameletBindings

You can create new KameletBindings by giving the complete specification in an external YAML file. The step loads the file content and creates the KameletBinding in the current namespace.

@Given("^(?:load KameletBinding {name}.yaml)$")

**Given** load KameletBinding `{name}.yaml`

Loads the file `{name}.yaml` as a KameletBinding. At the moment YAKS only supports .yaml source files.

### 7.5.3. Delete KameletBindings

@Given("^(?:delete KameletBinding {name}$")

**Given** delete KameletBinding `{name}`

Deletes the KameletBinding with given `{name}` from the current namespace.

### 7.5.4. Verify KameletBinding is available

@Given("^(?:KameletBinding {name} is available$$")

**Given** KameletBinding `{name} is available$

Verifies that the KameletBinding custom resource is available in the current namespace.

### 7.5.5. Manage Kamelet and KameletBinding resources

The described steps are able to create Kamelet resources on the current Kubernetes namespace. By default these resources get removed automatically after the test scenario.

The auto removal of Kamelet resources can be turned off with the following step.

@Given("^(?:Disable auto removal of Kamelet resources$")

**Given** Disable auto removal of Kamelet resources
Usually this step is a **Background** step for all scenarios in a feature file. This way multiple scenarios can work on the very same Kamelet resources and share integrations.

There is also a separate step to explicitly enable the auto removal.

```groovy
@Given("^Enable auto removal of Kamelet resources$")
```

**Given Enable auto removal of Kamelet resources**

By default, all Kamelet resources are automatically removed after each scenario.

### 7.6. Groovy steps

The Groovy support in YAKS adds ways to configure the framework with bean configurations and test actions via Groovy script snippets. In particular, you can add customized endpoints that send/receive data over various messaging transports.

#### 7.6.1. Framework configuration

YAKS uses Citrus components behind the scenes. The Citrus components are configurable through a Groovy domain specific language. You can add endpoints and other components as Citrus framework configuration like follows:

```groovy
@Given("^(?:create|new) configuration$")
```

**Given create configuration**

```groovy
<<<Groovy DSL>>
```

In the next example the step uses a Groovy domain specific language to define a new Http server endpoint.
**Scenario**: Endpoint script config

Given URL: http://localhost:18080

Given create configuration

```groovy
***
citrus {
    endpoints {
        http {
            server('helloServer') {
                port = 18080
                autoStart = true
            }
        }
    }
}
***

When send GET /hello

Then receive HTTP 200 OK

The configuration step creates a new Citrus endpoint named `helloServer` with given properties (`port`, `autoStart`) in form of a Groovy configuration script. The endpoint is a Http server Citrus component that is automatically started listening on the given port. In the following the scenario can send messages to that server endpoint.

The Groovy configuration script adds Citrus components to the test context and supports following elements:

- **endpoints**: Configure Citrus endpoint components that can be used to exchange data over various messaging transports
- **queues**: In memory queues to handle message forwarding for incoming messages
- **beans**: Custom beans configuration (e.g. data source, SSL context, request factory) that can be used in Citrus endpoint components

Let’s quickly have a look at a bean configuration where a new JDBC data source is added to the test suite.
**Scenario**: Bean configuration

```java
Given create configuration
```n
citrus {
    beans {
        dataSource(org.apache.commons.dbcp2.BasicDataSource) {
            driverClassName = "org.h2.Driver"
            url = "jdbc:h2:mem:camel"
            username = "sa"
            password = ""
        }
    }
}
```

The data source will be added as a bean named `dataSource` and can be referenced in all Citrus SQL test actions.

All Groovy configuration scripts that we have seen so far can also be loaded from external file resources, too.

```java
@Given("^load configuration {file_path}.groovy$")
```

```java
Given load configuration {file_path}.groovy
```

The file content is loaded as a Groovy configuration DSL. The next code sample shows such a configuration script.

```java
citrus {
    queues {
        queue('say-hello')
    }
}

endpoints {
    direct {
        asynchronous {
            name = 'hello'
            queue = 'say-hello'
        }
    }
}
```
7.6.2. Endpoint configuration

Endpoints describe an essential part in terms of messaging integration during a test. There are multiple ways to add custom endpoints to a test. Endpoint Groovy scripts is one comfortable way to add custom endpoint configurations in a test scenario. You can do so with the following step.

@Given("^(?:create|new) endpoint \{name\}\.groovy$")

```groovy
Given("^(?:create|new) endpoint \{name\}\.groovy

<<Groovy DSL>>

```The step receives a unique name for the endpoint and a Groovy DSL that specifies the endpoint component with all its properties. In the following sample a new Http server endpoint component will be created.

Create new Http server endpoint

**Scenario**: Create Http endpoint

* Given URL: `http://localhost:18081`
* Given create endpoint helloServer.groovy

```groovy
http()
   .server()
   .port(18081)
   .autoStart(true)
```

* When send GET /hello
* Then receive HTTP 200 OK

The scenario creates a new Http server endpoint named `helloServer`. This server component can be used directly in the scenario to receive and verify messages sent to that endpoint.

You can also load the endpoint configuration from an external file resources.

@Given("^load endpoint \{file_path\}\.groovy$")

```groovy
Given("^load endpoint \{file_path\}\.groovy$")
```

The referenced file should contain the endpoint Groovy DSL.

Create endpoint from file resource

**Scenario**: Load endpoint

* Given URL: `http://localhost:18088`
* Given load endpoint fooServer.groovy
* When send GET /hello
* Then receive HTTP 200 OK
7.6.3. Test actions

YAKS provides a huge set of predefined test actions that users can add to the Gherkin feature files out of the box. However, there might be situations where you want to run a customized test action code as a step in your feature scenario.

With the Groovy script support in YAKS you can add such customized test actions via script snippets:

```groovy
@Given("^(?:create|new) actions \{name\}.groovy\$")

Given create actions {name}.groovy$

<<<Groovy DSL>>

The Groovy test action DSL script receives a unique \{name\}. You can reference this name later in the test in order to apply the defined actions. When applied to the test the defined actions are executed. A sample will show how it is done.

Create test actions with a script

```groovy
Scenario: Custom test actions

Given create actions basic.groovy

```

```groovy
actions {
  echo('Hello from Groovy script')
  sleep().seconds(1)

  createVariables()
    .variable('foo', 'bar')

  echo('Variable foo=${foo}')
}

Then apply basic.groovy

The example above defines the test actions with the Groovy DSL under the name `basic.groovy`. Later in the test the actions are executed with the `apply` step.
Users familiar with Citrus will notice immediately that the action script is using the Citrus actions DSL to describe what should be done when running the Groovy script as part of the test.

The Citrus action DSL is quite powerful and allows you to perform complex actions such as iterations, conditionals and send/receive operations as shown in the next sample.

### Scenario: Messaging actions

```groovy
Given create actions messaging.groovy

```actions
```groovy
send('direct:myQueue')
  .payload('Hello from Groovy script!')

receive('direct:myQueue')
  .payload('Hello from Groovy script!')

```

Then apply messaging.groovy

As an alternative to write the Groovy DSL directly into the test feature file you can also load the test action script from external file resources.

```groovy
@Given("^load actions {file_name}\.groovy$")

```Given load actions {file_name}.groovy$\)

The file name is the name of the action script. So you can use the file name to apply the script in the test for execution.

```groovy
Then apply {file_name}.groovy

```

### 7.7. Http steps

The Http protocol is a widely used communication protocol when it comes to exchanging data between systems. REST Http services are very prominent and producing/consuming those services is a common task in software development these days. YAKS provides ready to use steps that are able to exchange request/response messages via Http as a client and server during the test.

The sample below shows how to use Http communication in a test:
Http communication sample

**Feature:** Http client

**Background:**
- **Given URL:** https://hello-service

**Scenario:** Health check
- **Given** path /health is healthy

**Scenario:** GET request
- **When** send GET /todo
- **Then** verify HTTP response body:
  ```json
  {"id": "@ignore@", "task": "Sample task", "completed": 0}
  ```
- **And** receive HTTP 200 OK

The example above sets a base request URL to https://hello-service and performs a health check on path /health. After that we can send a Http GET request to the endpoint and verify the response status code. All steps shown are part of the YAKS framework so you can use them out of the box. The next sections explore the Http capabilities in more detail.

### 7.7.1. Http client steps

As a client you can specify the server URL and send requests to it.

```
@Given("^(?:URL|url): {url}\$")
```

---

**Given URL:** {url}

The given URL points to a server endpoint. All further client Http steps send the requests to that endpoint.

As an alternative you can reference a Http client component that previously has been added to the framework configuration.

```
@Given("^HTTP client "{name}"\$")
```

---

**Given HTTP client "{name}"**

This step loads a Http client component by its name and uses that for further requests.

Once you have configured the Http endpoint URL or the Http client you can start sending request messages.

### 7.7.2. Send Http requests

Sending requests via Http is as easy as choosing the Http method (GET, POST, PUT, DELETE, ...) to
use.

@When("^send (GET|HEAD|POST|PUT|PATCH|DELETE|OPTIONS|TRACE) {path}$")

| When | send {method} {path} |

The given path resides to a valid resource on the server endpoint. The resource path is added to the base URL and identifies the resource on the server.

Send Http GET request

| When | send GET /todo |

You can choose the Http method that should be used to send the request (e.g. GET). Of course the request can have headers and a message body. You need to set these before sending the request in separate steps.

7.7.2.1. Request headers

@Given("^HTTP request header {name}="{value}"$")

| Given | HTTP request header {name}="{value}" |

The step above adds a Http header to the request. The header is defined with a name and receives a value. You can set multiple headers in a single step, too:

@Given("^HTTP request headers$")

| Given | HTTP request headers |
| {name} | {value} |

The step uses a data table to define multiple message headers with name and value.

Set request headers

| Given | HTTP request headers |
| Accept | application/json |
| Accept-Encoding | gzip |

7.7.2.2. Request body

The Http request can have a body content which is sent as part of the request.

@Given("^HTTP request body: {body}$")

| Given | HTTP request body: {body} |

The step above specifies the Http request body in a single line. When you need to use multiline
7.7.2.3. Request parameters

The Http request is able to use parameters that get added to the request URL. You can set those parameters in a separate step.

@Given("^HTTP request query parameter {name}="{value}"$")

**Given** HTTP request query parameter \{name\}="\{value\}"

7.7.2.4. Request timeouts

In some cases the client waits a long time for the server to respond. As Http is a synchronous communication protocol by its nature the client will synchronously wait for the response before doing any other step. You can specify the time to wait for the server to respond.

@Given("^HTTP request timeout is \{time\} milliseconds$")

**Given** HTTP request timeout is \{time\} milliseconds

This sets the client timeout to the given time in milliseconds.

7.7.2.5. Request fork mode

As seen in the previous section Http is synchronous by default. This can be a problem when the test needs to do multiple things in parallel. By default the Http client step will always block any other step until the server response has been received. You can change this behavior to an asynchronous behavior so the next steps will not be blocked by the Http client step.

@Given("^HTTP request fork mode is \{enabled\|disabled\}$")

**Given** HTTP request fork mode is enabled

This will enable the fork mode so all client request will be non-blocking. By default the fork mode is disabled.

7.7.3. Send raw Http request data

In the previous section several steps have defined the Http request (header, parameter, body) before sending the message in a separate step. As an alternative to this approach you can also
specify the complete Http request data in a single step.

@Given("^^send HTTP request$$")

```
Given send HTTP request
    
    <<request_data>>
    
```

The next example shows the complete Http request data step:

Send raw Http request data

```
Given send HTTP request
    
    GET https://hello-service
    Accept-Charset:utf-8
    Accept:application/json, application/*+json, */*
    Host:localhost:8080
    Content-Type:text/plain;charset=UTF-8
    
```

7.7.4. Verify Http responses

The time you send out a Http request you will be provided with a response from the server. YAKS is able to verify the Http response content in order to make sure that the server has processed the request as expected.

@Then("^^receive HTTP {status_code}(?: {reason_phrase})?$")

```
Then receive HTTP {status_code} {reason_phrase}

```

The most critical part of the Http response is thee status code (e.g. 200, 404, 500). The status code should refer to the success or error of the request. The server can use a wide range of Http status codes that are categorized as follows, also see W3C.

- **1xx** informational response – the request was received, continuing process
- **2xx** successful – the request was successfully received, understood, and accepted
- **3xx** redirection – further action needs to be taken in order to complete the request
- **4xx** client error – the request contains bad syntax or cannot be fulfilled
- **5xx** server error – the server failed to fulfil an apparently valid request

Verify Http status code

```
Then receive HTTP 200 OK

```

The reason phrase **OK** is optional and is also not part of the verification mechanism for the response.
It just gives human readers a better understanding of the status code.

Of course the Http response can also have headers and a message body. YAKS is able to verify those response data, too. Please define the expected headers and body content before verifying the status code.

### 7.7.4.1. Response headers

```gherkin
@Then("expect HTTP response header {name}="{value}"$)
```

Then expect HTTP response header {name}="{value}"

The step above adds a Http header to the response. The header is defined with a name and receives a value. You can set multiple headers in a single step, too:

```gherkin
@Then("expect HTTP response headers$")
```

```plaintext
| {name} | {value} |
```

The step uses a data table to define multiple message headers with name and value.

Verify response headers

```plaintext
<table>
<thead>
<tr>
<th>Encoding</th>
<th>gzip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content-Type</td>
<td>application/json</td>
</tr>
</tbody>
</table>
```

### 7.7.4.2. Response body

```gherkin
@Then("expect HTTP response body: {body}$")
```

Then expect HTTP response body: {body}

The step above specifies the Http response body in a single line. When you need to use multiline body content please use the next step:

```gherkin
@Then("expect HTTP response body$")
```

```plaintext
<<<content>>
```

### 7.7.5. Verify raw Http response data

In the previous section several steps have defined the Http response (header, parameter, body) before verifying the message received. As an alternative to this approach you can also specify the
complete expected Http response data in a single step.

@Then("^receive HTTP response$")

Then receive HTTP response

"""
<<response_data>>
"""

The next example shows the complete Http response data step:

Verify raw Http response data

Then receive HTTP response

"""
HTTP/1.1 200 OK
Content-Type: application/json
X-TodoId:@isNumber()@
Date: @ignore@

{"id": @ignore@, "task": "Sample task", "completed": 0}
"""

7.7.6. Verify response using JsonPath

When verifying Http client responses sent by the server you can use JsonPath expressions to validate the response message body content.

@Then("^(?:expect|verify) HTTP response expression: {expression}="{value}"$")

Then expect HTTP response expression: {expression}="{value}"

The step defines a JsonPath expression (e.g. $.person.age) and an expected value. The expression is evaluated against the received response message body and the value is compared to the expected value. This way you can explicitly verify elements in the Json body.

The very same mechanism also applies to XML message body content. Just use a XPath expression instead of JsonPath.

7.7.7. Http server steps

On the server side YAKS needs to start a Http server instance on a given port and listen for incoming requests. The test is able to verify incoming requests and then provide a simulated response message with response headers and body content.
Http communication sample

**Feature:** Http server

**Background:**
- **Given** HTTP server listening on port 8080

**Scenario:** Expect GET request
- **When** receive GET /todo
- **Then** HTTP response body: `{"id": 1000, "task": "Sample task", "completed": 0}
- **And** send HTTP 200 OK

**Scenario:** Expect POST request
- **Given** expect HTTP request body: `{"id": @isNumber(), "task": "New task", "completed": @matches(0|1)@}`
- **When** receive POST /todo
- **Then** send HTTP 201 CREATED

In the HTTP server sample above we create a new server instance listening on port 8080. Then we expect a GET request on path /todo. The server responds with a Http 200 OK response message and given Json body as payload.

The second scenario expects a POST request with a given body as Json payload. The expected request payload is verified with the powerful Citrus JSON message validator being able to compare JSON tree structures in combination with validation matchers such as @isNumber() or @matches(0|1).

After the request verification has passed the server responds with a simple Http 201 CREATED.

The next sections guide you through the Http server capabilities in YAKS.

**7.7.8. Http server port**

When the test run starts YAKS will initialize the Http server instance and listen on a port for incoming requests. By default this server uses the port 8080, but you can adjust the port with following step.

@Given("^HTTP server listening on port \{port\}$")

**Given** HTTP server listening on port \{port\}

As an alternative to that you can reference a Http server component that can be found in the framework configuration.

@Given("^HTTP server \{"name\}\$")

**Given** HTTP server \{"name\}"

This step loads a Http server component by its name and uses that for server side testing.
7.7.9. Receive Http requests

You can define expected incoming Http requests as part of the test.

@When("^receive \(GET|HEAD|POST|PUT|PATCH|DELETE|OPTIONS|TRACE\) \{path\}$")

When receive \{method\} \{path\}

The incoming request must match the given \{method\} and \{path\}.

Receive Http GET request

When receive GET /todo

Of course, you can also verify headers and the request message body. You need to specify the expected request before receiving the request with the receive steps.

7.7.9.1. Request headers

@Given("^\(?:expect|verify\) HTTP request header \{name\}=\"\{value\}\"\)$")

Given expect HTTP request header \{name\}=\"\{value\}\"

The step above adds the Http header to the request validation. The header must be present in the incoming request and must match the expected value. You can verify multiple headers in a single step, too:

@Given("^\(?:expect|verify\) HTTP request headers$\)

Given expect HTTP request headers

| \{name\} | \{value\} |

The step uses a data table to define the message headers with name and value.

Expect request headers

Given expect HTTP request headers

| Accept | application/json |
| Accept-Encoding | gzip |

7.7.9.2. Request body

Each incoming Http request can have a body and you are able to verify the body content in multiple ways.

@Given("^\(?:expect|verify\) HTTP request body: \{body\}$")

Given expect HTTP request body: \{body\}
The step above specifies the expected Http request body in a single line. Multiline body content must use the next step:

@Given("^(?:expect|verify) HTTP request body\$")

```plaintext
Given expect HTTP request body
```nn```
<<content>>
```nn

7.7.3. Request parameters

The Http request can have parameters on the request URL. You can verify those parameters in a separate step.

@Given("^(?:expect|verify) HTTP request query parameter {name}="{value}"\$")

```plaintext
Given expect HTTP request query parameter {name}="{value}"
```nn

7.7.4. Request timeouts

The test waits for incoming requests but the test may hit request timeouts when no request has been received. By default the server waits for five seconds each time a request is expected. You can adjust the server timeout.

@Given("^HTTP server timeout is {time} milliseconds\$")

```plaintext
Given HTTP server timeout is {time} milliseconds
```nn

This sets the server timeout to the given time in milliseconds.

7.7.10. Receive raw Http request data

In the previous section several steps have defined the expected Http request (header, parameter, body). As an alternative to this approach you can also specify the complete Http request data in a single step.

@Given("^receive HTTP request\$")

```plaintext
Given receive HTTP request
```nn```
<<request_data>>
```nn

The next example shows the complete Http request data step:
Receive raw HTTP request data

```
Given receive HTTP request

GET https://hello-service
Accept-Charset:utf-8
Accept:application/json, application/*+json, */*
Host:localhost:8080
Content-Type:text/plain;charset=UTF-8
```

### 7.7.11. Verify requests using JsonPath

When verifying HTTP client requests you can use JsonPath expressions to validate the request message body content.

@When("^(?:expect|verify) HTTP request expression: {expression}="{value}"$")

```java
When expect HTTP request expression: {expression}="{value}"
```

The step defines a JsonPath expression (e.g. $.person.age) and an expected value. The expression is evaluated against the received request message body and the value is compared to the expected value. This way you can explicitly verify elements in the Json body.

The very same mechanism also applies to XML message body content. Just use a XPath expression instead of JsonPath.

### 7.7.12. Send HTTP responses

The time you have verified a HTTP request as a server you need to provide a proper response to the calling client. YAKS is able to simulate the HTTP response content.

@Then("^send HTTP {status_code}(?: {reason_phrase})?$")

```java
Then send HTTP {status_code} {reason_phrase}
```

The step defines the HTTP response status code (e.g. 200, 404, 500) to return.

Return HTTP status code

```java
Then send HTTP 200 OK
```

The reason phrase OK is optional. It just gives human readers a better understanding of the status code returned.

Of course the HTTP response can also have headers and a message body. YAKS is able to simulate this response data, too.
### 7.7.12.1. Response headers

@Given("^HTTP response header {name}="{value}"$")

Given HTTP response header \{name\}={value}"

The step above adds a Http header to the response. The header is defined with a name and value. You can set multiple headers in a single step, too:

@Given("^HTTP response headers$")

Given HTTP response headers
| {name} | {value} |

The step uses a data table to define multiple message headers with name and value.

Return response headers

| {name} | Encoding | Content-Type | application/json |

### 7.7.12.2. Response body

@Given("^HTTP response body: {body}$")

Given HTTP response body: \{body\}

The step above specifies the Http response body in a single line. When you need to use multiline body content please use the next step:

@Given("^HTTP response body$")

Given HTTP response body
```
<<content>>
```

### 7.7.13. Send raw Http response data

In the previous section several steps have defined the Http response (header, parameter, body). As an alternative to this approach you can also specify the complete Http response data in a single step.
@Then("send HTTP response")

Then send HTTP response

""
<<response_data>>
""

The next example shows the complete HTTP response data step:

Return raw HTTP response data

Then send HTTP response

""
HTTP/1.1 200 OK
Content-Type: application/json
X-TodoId: @isNumber()@
Date: @ignore@

{"id": @ignore@, "task": "Sample task", "completed": 0}
""

7.7.14. HTTP health checks

Often HTTP server provide a health endpoint so clients can check the status of the server to be up and running. The health check is supported with the following steps.

@Given("{URL} is healthy")

Given {URL} is healthy

The step performs a health check on the given {URL} by sending a request to the endpoint and checking for a response status code marking success (200 OK).

Health check

Given https://some-service-url/health is healthy

Instead of specifying the complete health check URL you can make use of the base URL given in the central HTTP step.

@Given("{URL} {path} is healthy")

Given path {path} is healthy

The given path is added to the base URL and should resolve to the health check resource on the server (e.g. /health).
Health path check

| Given URL: https://hello-service |
| Given path /health is healthy |

The steps above perform the health check only a single time. Based on the provided HTTP server response status the step passes or fails. In some cases, it can not make sure that the server has been started yet and the health check might fail occasionally. In these cases, it is a good idea to use the wait health check step.

@Given("^wait for URL {url}$")

| Given wait for URL {url} |

The step will wait for given URL to return a 200 OK response. The step is actively waiting while polling the URL multiple times when the response is not positive. By default, this step uses a HEAD request. You can explicitly choose another HTTP method, too.

@Given("^wait for (GET|HEAD|POST|PUT|PATCH|DELETE|OPTIONS|TRACE) on URL {url}$")

| Given wait for GET on URL {url} |

The sample above uses a GET request for the health checks.

Also, you can explicitly specify the expected return code that must match in order to pass the wait health check.

@Given("^wait for URL {url} to return {status_code}(?: {reason_phrase}?)$")

| Given wait for URL {url} to return {status_code} {reason_phrase} |

Once again, the {reason_phrase} is optional and only for better readability reasons.

Wait for specific status code

| Given wait for URL https://hello-service/health to return 200 OK |

Last not least, you can specify the request method on the wait operation, too.

@Given("^wait for (GET|HEAD|POST|PUT|PATCH|DELETE|OPTIONS|TRACE) on URL {url} to return {status_code}(?: {reason_phrase}?)$")

| Given wait for {method} on URL {url} to return {status_code} {reason_phrase} |

This completes the health check capabilities in the HTTP steps.
7.8. JDBC steps

YAKS provides steps that allow executing SQL actions on relational databases. This includes updates and queries. In case of a database query you are able to verify the result set with expected values.

You can find examples of JDBC steps in examples/jdbc.

7.8.1. Connection configuration

Before running any SQL statement you need to configure a datasource that allows connecting to the database.

@Given("^(?:D|d)atabase connection$")

| Given Database connection | {property} | {value} |

The step configures a new database connection and uses a data table to define connection properties such as connection URL, username and password.

Specify connection properties

| Given Database connection | driver | org.postgresql.Driver |
| | url | jdbc:postgresql://localhost:5432/testdb |
| | username | test |
| | password | secret |

This defines the connection parameters so the test is able to connect to the database.

In addition to that you can also reference an existing datasource that has been added to the framework configuration.

@Given("^(?:D|d)ata source: {name}$")

| Given Data source: {name} |

The name of the datasource should reference a configured component in the test project. You can add components as Spring beans for instance.

7.8.2. SQL update

The test is able to run SQL updates (UPDATE, INSERT, DELETE) on the database.

@When("^(?:execute | perform )?SQL update: {statement}$")

| When execute SQL update: {statement} |
The step executes the given SQL statement using the configured database connection. For multiline statements please use:

```
@When("^(?:execute | perform )?SQL update\$")
```

```
<table>
<thead>
<tr>
<th>When</th>
<th>execute SQL update</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{statement}</td>
</tr>
</tbody>
</table>
```

You can also run multiple statements in a single step by using a data table.

```
@When("^(?:execute | perform )?SQL updates\$")
```

```
<table>
<thead>
<tr>
<th>When</th>
<th>execute SQL updates</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{statement_1}</td>
</tr>
<tr>
<td></td>
<td>{statement_2}</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{statement_x}</td>
</tr>
</tbody>
</table>
```

7.8.3. SQL query

The SQL query obtains data from the database in form of result sets. The YAKS test is able to verify the result sets with an expected set of rows and column values returned.

```
@Given("^SQL query: \{statement\}\$")
```

```
Given SQL query: \{statement\}
```

This step defines the query to execute. Multiline SQL query statements are supported, too.

```
@Given("^SQL query\$")
```

```
<table>
<thead>
<tr>
<th>Given</th>
<th>SQL query</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{statement}</td>
</tr>
</tbody>
</table>
```

You can also run multiple queries in one step. As usual the step uses a data table.

```
@Given("^SQL query statements\$")
```

```
<table>
<thead>
<tr>
<th>When</th>
<th>SQL query statements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{statement_1}</td>
</tr>
<tr>
<td></td>
<td>{statement_2}</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{statement_x}</td>
</tr>
</tbody>
</table>
```
In a next step you can provide the expected outcome in form of column name and value.

7.8.3.1. Verify SQL result set

@Then("^verify column {name}={value}$")

Then verify column {name}={value}

This step executes the query and verifies the column with given name to match the expected value.

You can use multiple verifications on several columns with a data table.

@Then("^verify columns$")

| {column_1} | {value_1_1} | {value_1_2} | 
| {column_2} | {value_2_1} | {value_2_2} | 
| ... |
| {column_x} | {value_x_x} | {value_x_x} |

The data table is able to verify a matrix of rows and columns. Each column can have multiple row values.

Validate multi row result sets

Given SQL query: `SELECT ID, TASK, COMPLETED FROM todo ORDER BY id` Then verify columns

| ID | 1 | 2 | 3 | 4 |
| TASK | Learn some CamelK! | Get some milk | Do laundry | Wash the dog | Test CamelK with YAKS! |
| COMPLETED | 0 | 0 | 0 | 0 |

7.8.4. Result set verification script

For more complex result set validation you can use a Groovy result set verification script.

@Then("^verify result set$")

Then verify result set

```groovy
<<Groovy>>
```

The Groovy script can work with the complete result set and is quite powerful.
7.9. JMS steps

JMS is well-known as transport for point-to-point and publish/subscribe messaging. Users can produce and consume messages on queues and topics on a message broker.

YAKS has support for JMS related messaging on both producer and consumer.

7.9.1. Connection factory

The JMS standard requires clients to open connections over a connection factory. The connection factory is a vendor specific implementation and receives a set of properties such as connection URL, username and password.

```python
@Given("^(?:JMS|jms) connection factory$")
```

| Given JMS connection factory | {property} | {value} |

The configuration step receives a data table that defines the connection settings.

### Connection factory settings

<table>
<thead>
<tr>
<th>Given JMS connection factory</th>
<th>type</th>
<th>org.apache.activemq.ActiveMQConnectionFactory</th>
</tr>
</thead>
<tbody>
<tr>
<td>brokerUrl</td>
<td>tcp://localhost:61616</td>
<td></td>
</tr>
<tr>
<td>username</td>
<td>${activemq.user}</td>
<td></td>
</tr>
<tr>
<td>password</td>
<td>${activemq.password}</td>
<td></td>
</tr>
</tbody>
</table>

The connection factory type is vendor specific and depends on what kind of message broker you are using in your environment. Please make sure to add the respective client library as a project dependency in the YAKS configuration.

Sensitive values such as `username` and `password` can be set with a test variable placeholder. The variable value can be set by a secret in Kubernetes/Openshift. This ensures to not share sensitive data in the public.

As an alternative to defining the connection factory as part of the test steps you can load a predefined connection factory component from the configuration.
@Given("^\(?:JMS|jms\) connection factory \(name\)$")

Given JMS connection factory {name}

The step references a connection factory component that has been added to the framework configuration (e.g. as Spring bean). This way you can share the connection factory in multiple tests.

7.9.2. Destination and endpoint configuration

In addition to the connection factory the test needs to specify the JMS destination (queue or topic) to use.

@Given("^\(?:JMS|jms\) destination: \{name\}$")

Given JMS destination: {name}

This sets the destination name for the next steps. As an alternative to that you can also reference a predefined endpoint component from the configuration.

@Given("^\(?:JMS|jms\) endpoint \"\{name\}\"$")

Given JMS endpoint {name}

The step tries to resolve the JMS endpoint with given {name} in the available configuration. The endpoint has a destination set and references a connection factory on its own.

So now the test is ready to produce and consume messages from JMS destinations.

7.9.3. Send JMS messages

A test can publish messages on a JMS destination. The message consists of message headers and a body content. Before sending a message the tests needs to specify the message content.

7.9.3.1. Message headers

The message headers are key value pairs that are sent as part of the message. You can add a new header with the following step:

@Given("^\(?:JMS|jms\) message header \{name\}(?:=| is )\"\{value\}\"$")

Given JMS message header {name}="{value}"

When using a data table you can set multiple headers in one step.
@Given("^(?:JMS|jms) message headers$")

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>{header_1}</td>
<td>{value_1}</td>
</tr>
<tr>
<td>{header_2}</td>
<td>{value_2}</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>{header_x}</td>
<td>{value_x}</td>
</tr>
</tbody>
</table>

7.9.3.2. Message body

@Given("^(?:JMS|jms) message body: {body}$")

Given JMS message body: {body}

This step can set a single line body content. Of course you can also work with multiline body content.

@Given("^(?:JMS|jms) message body$")

Given JMS message body

```
{body}
```

Now another step can send the message as it has been specified in the previous steps.

@When("^(?:JMS|jms) message$")

When send JMS message

This sends the message to the previously configured JMS destination. You can overwrite this destination in the send step.

@When("^(?:JMS|jms) message to destination {destination}$")

When send JMS message to destination {destination}

The approach described clearly separates message specification and send operation as all of it is done in separate steps. As an alternative you can also specify the message content in one step.

@When("^(?:JMS|jms) message with body: {body}$")

When send JMS message with body: {body}

You can also add some message headers to this step.
When send JMS message with body and headers: {body}:

- {header_1}: {value_1}
- {header_2}: {value_2}
- ...
- {header_x}: {value_x}

The step combines message header and body specification with the actual send operation.

7.9.4. Receive JMS messages

Similar to sending messages to a JMS destination the test can also consume messages from a queue or topic. When the message has been received a validation mechanism makes sure that the message content received matches the expectations.

Users are able to provide expected message headers and body content in order to verify the received message.

7.9.4.1. Message headers

The expected message headers need to be set before receiving the message from the destination.

@Given("^(?:JMS|jms) message header {name}(?:=| is )"{value}"\$")

Given JMS message header {name}="{value}"

When using a data table you can expect multiple headers in one step.

@Given("^(?:JMS|jms) message headers$\")

Given JMS message headers:

- {header_1}: {value_1}
- {header_2}: {value_2}
- ...
- {header_x}: {value_x}

7.9.4.2. Message body

In addition to verify message headers you can also verify the body content. Once again the user specifies the expected message body before the message is received.

@Given("^(?:JMS|jms) message body: {body}$\")

Given JMS message body: {body}

This step can expect a single line body content. Of course you can also work with multiline body content.
@Given("^(?:JMS|jms) message body$")

```java
Given JMS message body
   """
   {body}
   """
```

With the steps above the test has specified the expected message content. With that in place another step can receive the message and perform the validation.

@Then("^receive (?:JMS|jms) message$")

```java
Then receive JMS message
```

The step uses the previously defined JMS destination to consume messages from it. You can use another destination in the step, too.

@Then("^receive (?:JMS|jms) message from destination \{destination\}$")

```java
Then receive JMS message from destination \{destination\}
```

With this approach you have a clean separation of the expected message content specification and the actual receive operation. Of course you can also combine everything in one single step.

@Then(?:receive\|expect\|verify) (?:JMS\|jms) message with body: \{body\}$")

```java
Then receive JMS message with body: \{body\}
```

You can also add some message headers to this step.

@Then(?:receive\|expect\|verify) (?:JMS\|jms) message with body and headers: \{body\}$")

```java
Then receive JMS message with body and headers: \{body\}
| \{header_1\} | \{value_1\} |
| \{header_2\} | \{value_2\} |
...  
| \{header_x\} | \{value_x\} |
```

The step combines message header and body specification with the actual receive operation.

### 7.9.4.3. Consumer timeout

The receive operation takes the first message available on the destination and performs the validation. In case there is no message available the consumer will wait for a given amount of time before a timeout will fail the test. You can adjust the timeout on the JMS consumer.
7.9.4.4. Message selectors

The JMS standard provides a concept of message selectors so consumers can specify which message they want to consume from a destination. The consumer usually evaluates the selector expression on the message headers.

The selector expression defines a key and value that the message must match. The first message to match the selector on the destination it received by the consumer.

Use message selector

7.10. Kafka steps

Apache Kafka is a powerful and widely used event streaming platform. Users are able publish events and subscribe to event streams.

The following sections describe the support for Kafka related event streaming in YAKS.

7.10.1. Connection

First of all the test needs to connect to Kafka bootstrap servers. The connection provides several parameters that you can set with the following step.

The configuration step receives a data table that defines the connection settings.

Connection settings
The most important part of the connection settings is the **url** that points to one or more Kafka bootstrap servers.

In addition to the connection settings there is a set of producer and consumer properties that you can set in order to configure the behavior of producers and consumers connecting with Kafka.

```
@Given("^(?:Kafka|kafka) producer configuration$")
```

```
| Given Kafka producer configuration |
| {property} | {value} |
```

```
@Given("^(?:Kafka|kafka) consumer configuration$")
```

```
| Given Kafka consumer configuration |
| {property} | {value} |
```

The available properties to set here are described in the Apache Kafka documentation. See the following example how to set producer and consumer properties.

```
| Given Kafka producer configuration |
| client.id | yaks_producer |
| request.timeout.ms | 5000 |

| Given Kafka consumer configuration |
| client.id | yaks_consumer |
| max.poll.records | 1 |
```

### 7.10.2. Topic and endpoint configuration

In addition to the connection the test needs to specify the Kafka topic to use.

```
@Given("^(?:Kafka|kafka) topic: {name}$")
```

```
| Given Kafka topic: {name} |
```

This sets the topic name for the next steps. As an alternative to that you can also reference a predefined endpoint component from the configuration.

```
@Given("^(?:Kafka|kafka) endpoint "{name}"$")
```

```
| Given Kafka endpoint {name} |
```

The step tries to resolve the Kafka endpoint with given `{name}` in the available configuration. The endpoint can reference a topic set and connection settings on its own.

So now the test is ready to produce and consume events from Kafka topics.
7.10.3. Send Kafka events

A test can publish events on a Kafka topic. The event consists of message headers and a body content. Before sending an event the tests needs to specify the message content.

7.10.3.1. Message key

Each event on a Kafka event stream has a message key set. You can set this key in a separate step before sending the event.

```java
@Given("^(?:Kafka|kafka) message key: \{key\}$")
Given Kafka message key: \{key\}$")
```

This specifies the message key for the next event that is published.

7.10.3.2. Message headers

The message headers are key value pairs that are sent as part of the message. You can add a new header with the following step:

```java
@Given("^(?:Kafka|kafka) message header \{name\}(?:=| is )"\{value\}"\$")
Given Kafka message header \{name\}="\{value\}""
```

When using a data table you can set multiple headers in one step.

```java
@Given("^(?:Kafka|kafka) message headers$")
Given Kafka message headers

| \{header_1\} | \{value_1\} |
| \{header_2\} | \{value_2\} |
| ... |
| \{header_x\} | \{value_x\} |
```

7.10.3.3. Message body

```java
@Given("^(?:Kafka|kafka) message body: \{body\}$")
Given Kafka message body: \{body\}
```

This step can set a single line body content. Of course you can also work with multiline body content.
Given Kafka message body
   
   `{body}`

Now another step can publish the message as it has been specified in the previous steps.

When send Kafka message

This publishes the message to the previously configured Kafka topic. You can overwrite this topic in the publish step.

When send Kafka message to topic `{topic}`

The approach described clearly separates message specification and send operation as all of it is done in separate steps. As an alternative you can also specify the message content in one step.

When send Kafka message with body: `{body}`

You can also add some message headers to this step.

When send Kafka message with body and headers: `{body}`

| {header_1} | {value_1}  |
| {header_2} | {value_2}  |
| ...         |
| {header_x}  | {value_x}  |

The step combines message header and body specification with the actual send operation.

7.10.4. Receive Kafka events

Similar to publishing events to a Kafka topic the test can also consume events from an event stream. When the event has been received a validation mechanism makes sure that the message content received matches the expectations.

Users are able to provide expected message headers and body content in order to verify the received event.
7.10.4.1. Message headers

The expected message headers need to be set before receiving the event from the topic.

@Given("^(?:Kafka|kafka) message header {name}(?:=| is )"{value}"$")

Given Kafka message header {name}="{value}"

When using a data table you can expect multiple headers in one step.

@Given("^(?:Kafka|kafka) message headers$")

Given Kafka message headers
| {header_1} | {value_1} |
| {header_2} | {value_2} |
|... |
| {header_x} | {value_x} |

7.10.4.2. Message body

In addition to verify message headers you can also verify the body content. Once again the user specifies the expected message body before the message is received.

@Given("^(?:Kafka|kafka) message body: {body}$")

Given Kafka message body: {body}

This step can expect a single line body content. Of course you can also work with multiline body content.

@Given("^(?:Kafka|kafka) message body$")

Given Kafka message body

```
{body}
```

With the steps above the test has specified the expected message content. With that in place another step can receive the message and perform the validation.

@Then("^(?:receive (?:Kafka|kafka) message$")

Then receive Kafka message

The step uses the previously defined Kafka topic to consume events from it. You can use another topic in the step, too.
With this approach you have a clean separation of the expected message content specification and the actual receive operation. Of course you can also combine everything in one single step.

You can also add some message headers to this step.

The step combines message header and body specification with the actual receive operation.

7.10.4.3. Consumer timeout

The receive operation takes the first event available on the topic and performs the validation. In case there is no event available the consumer will wait for a given amount of time before a timeout will fail the test. You can adjust the timeout on the Kafka consumer.

7.10.5. Special configuration

The Kafka standard provides a set of special configuration that you can set as part of the test.

This set the topic partition for all further steps publishing or consuming events from that topic.

7.11. Kubernetes steps

Kubernetes is a famous container management platform that allows automation of deployment, scaling and management of containerized applications.
YAKS uses the Kubernetes client API and is able to create Kubernetes resources (e.g. secrets, services, deployments and so on) as part of the test.

### 7.11.1. Client configuration

@Given("^Kubernetes timeout is {time}(?: ms | milliseconds)$")

This sets the timeout for all Kubernetes client operations.

### 7.11.2. Set namespace

Kubernetes uses the concept of namespaces to separate workloads on the cluster. You can connect to a specific namespace with the following step.

@Given("^Kubernetes namespace {name}$")

### 7.11.3. Kubernetes services

One of the most important features in the YAKS Kubernetes support is the management of services and in particular the automatic deployment of simulated services in Kubernetes.

The user is able to start a local Http server instance and create a service in Kubernetes out of it. This way the test is able to simulate services in Kubernetes and receive and verify incoming requests as part of the test.

First of all we define a new service within the test.

@Given("^Kubernetes service "{name}"$")

This initializes a new Http server that will be used as Kubernetes service. The server is listening on a default port 8888. You can use another port.

@Given("^Kubernetes service port {port}$")

In the following the test is able to create a new Kubernetes service with that Http server.

@Given("^create Kubernetes service {name}$")
The step creates the service in the Kubernetes namespace and exposes the given service port as target port. Clients are now able to connect to that new service. Each requests on the service will reside in a request in the test pod. The test is able to receive the request and verify its content as usual.

This way we can easily simulate Kubernetes services in the current namespace.

In case you need to use another target port you can adjust the port as follows.

Given ^create Kubernetes service \{name\} with target port \{port\}$

Given create Kubernetes service \{name\} with target port \{port\}

This exposes the service with the given target port.

In case you do not need the service anymore you can delete it with this step:

Given ^delete Kubernetes service \{name\}$

Given delete Kubernetes service \{name\}

### 7.11.4. Secrets

Secrets are resources that hold sensitive data. Other resources on the cluster can mount the secret content and access it.

You can create secrets in the current namespace in multiple ways.

Given ^create Kubernetes secret \{name\}$

| {property} | {value} |

The step receives a secret name and a data table holding the property keys and values. These properties build the content of the secret.

Instead of listing all properties in the test itself you can load the secret from an external property file.

Given ^load Kubernetes secret from file \{file\}.properties$

Given load Kubernetes secret from file \{file\}.properties

The step loads the property file and creates the secret from the file content. The file name is used as the name for the secret.

In case you want to cleanup the secret you can delete it with:
7.11.5. Pods, deployments and other resources

In the previous sections the test has creates services and secrets as Kubernetes resources. In addition to that the test is able to apply any resource as a YAML file on the Kubernetes cluster.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
labels:
  name: my-pod
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

The step above creates a new pod resource with the given specification. Instead of adding the resource specification in the test itself you can also load an external YAML file.
In case you need to delete a resource you can do so by specifying the minimal resource as a YAML specification.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
      - containerPort: 80
```

You can also provide the external YAML file when deleting a resource. The step will automatically extract the resource kind and name from the file content.

```ruby
@Given("^delete Kubernetes resource {file_path}$")
```

```ruby
Given delete Kubernetes resource {file_path}
```

Delete resource from file

```ruby
Given delete Kubernetes resource pod.yaml
```
7.11.6. Custom resources

In the previous sections the test has created Kubernetes resources (pods, services, secrets, deployments, ...). The user can also define custom resources in order to extend Kubernetes. YAKS is also able to manage these custom resources.

@Given("^create Kubernetes custom resource in {crd}$")

```
Given create Kubernetes custom resource in {crd}

<<<YAML>>>

apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  name: my-broker

```

Once again the user has to provide a YAML specification of the resource.

Create custom resource

```
Given create Kubernetes custom resource in brokers.eventing.knative.dev

apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  name: my-broker

```

The step needs to know the `{crd}` (custom resource definition) of the custom resource. In the example above the test creates a new resource of kind `Broker` in the custom resource definition `brokers.eventing.knative.dev`.

Of course, you can also load the custom resource from external file resource.

@Given("^load Kubernetes custom resource {file_path} in {crd}$")

```
Given load Kubernetes custom resource {file_path} in {crd}

Load custom resource from file

```

```
Given load Kubernetes custom resource broker.yaml in brokers.eventing.knative.dev

```

Once again the step needs to have the `crd` type and the YAML specification as a file resource.

When deleting a custom resource from Kubernetes the user has to provide a minimal YAML specification that identifies the resource.
Delete custom resource

```yaml
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  name: my-broker
```

As an alternative to that you can use an external file resource that holds the minimal YAML specification.

Delete custom resource from file

```yaml
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  name: my-broker
```

7.11.7. Cleanup Kubernetes resources

The described steps are able to create Kubernetes resources on the current Kubernetes namespace. By default these resources get removed automatically after the test scenario.

The auto removal of Kubernetes resources can be turned off with the following step.

```
@Given("^Disable auto removal of Kubernetes resources$")

Given Disable auto removal of Kubernetes resources
```

Usually this step is a Background step for all scenarios in a feature file. This way multiple scenarios can work on the very same Kubernetes resources.

There is also a separate step to explicitly enable the auto removal.

```
@Given("^Enable auto removal of Kubernetes resources$")

Given Enable auto removal of Kubernetes resources
```
By default, all Kubernetes resources are automatically removed after each scenario.

### 7.12. Knative steps

Knative represents the Kubernetes-based platform to manage serverless workloads. In YAKS you are able to leverage Knative eventing features such as producing and consuming events.

Have a look at the complete example in `examples/knative-eventing.feature`.

The following sections guide you through the Knative eventing capabilities in YAKS.

#### 7.12.1. Client configuration

@Given("^Knative timeout is {time}(?: ms | milliseconds)$")

This sets the timeout for all Knative client operations.

#### 7.12.2. Set namespace

Knative uses the concept of namespaces to separate workloads on the cluster. You can connect to a specific namespace with the following step.

@Given("^Knative namespace {name}$")

This sets the broker name to use in all further steps that publish and consume events. The broker should already be present on the Kubernetes namespace. In case there is no broker yet you can create one.

@Given("^create Knative broker {name}$")

The step creates a new broker with the given `{name}`. The broker uses the default settings given in the Knative platform.
You can verify that the broker is up and running with the following step:

@Given("^Knative broker \{name\} is running$")

    Given Knative broker \{name\} is running

### 7.12.4. Create event consumer service

The Knative broker delivers events to sinks. In order to start consuming events in a test you should create a event consumer service which acts as a sink.

@Given("^create Knative event consumer service \{service\}$")

    Given create Knative event consumer service \{service\}

This step creates a new event consumer service. In particular this step creates a new Kubernetes service. The service instantiates a new local Http server and creates a new Kubernetes service from it. The service exposes a port which is 8080 by default.

You can adjust the service port as follows:

@Given("^Knative service port \{port\}$")

    Given Knative service port \{port\}

By default the Kubernetes service uses the service port as a target port when exposing the service. You can choose another target port, too.

@Given("^create Knative event consumer service \{service\} with target port \{port\}$")

    Given create Knative event consumer service \{service\} with target port \{port\}

### 7.12.5. Manage triggers

Triggers are used to deliver events to services and channels. In YAKS users can create a trigger as part of the test in order to start consuming events.

#### 7.12.5.1. Triggers on services

@Given("^create Knative trigger \{trigger\} on service \{service\}$")

    Given create Knative trigger \{trigger\} on service \{service\}

The step creates a new trigger on the given Knative broker. The trigger watches for events on the broker and forwards these events to the given service.

The service name either references an existing Kubernetes service or a new service that is created as part of the test as described in this guide.
Triggers can use filters on event attributes. This narrows the amount of events handled by the trigger.

@Given("^create Knative trigger \{trigger\} on service \{service\} with filter on attributes$")

<table>
<thead>
<tr>
<th>Given</th>
<th>create Knative trigger {trigger} on service {service} with filter on attributes</th>
</tr>
</thead>
</table>

You need to add one or many attributes with respective value that should be added to the filter. As a result the trigger will only handle events matching the given filters.

### 7.12.5.2. Triggers on channels

Triggers can also forward events to channels. Subscribers are able to start subscriptions on these channels in order to receive the events.

@Given("^create Knative trigger \{trigger\} on channel \{channel\}$")

<table>
<thead>
<tr>
<th>Given</th>
<th>create Knative trigger {trigger} on channel {channel}</th>
</tr>
</thead>
</table>

Of course, you can also add filters on attributes that narrow the amount of events handled by the trigger.

@Given("^create Knative trigger \{trigger\} on channel \{channel\} with filter on attributes$")

<table>
<thead>
<tr>
<th>Given</th>
<th>create Knative trigger {trigger} on channel {channel} with filter on attributes</th>
</tr>
</thead>
</table>

The step uses a data table with attributes and values that should be added to the filter. As a result the trigger will only handle events matching the given filters.

### 7.12.6. Create channels

Channels represent a central concept of Knative eventing. Channels are able to deliver events to multiple subscribers. A test in YAKS is able to create new channels.

@Given("^create Knative channel \{channel\}$")

<table>
<thead>
<tr>
<th>Given</th>
<th>create Knative channel {channel}</th>
</tr>
</thead>
</table>

Once the channel is available you can subscribe a service to the channel.

@Given subscribe service \{service\} to Knative channel \{channel\}$")

| Given | subscribe service \{service\} to Knative channel \{channel\} |
7.12.7. Publish events

The test is able to publish events on the Knative broker. YAKS uses the Knative Http client API to publish events on the broker.

Because of that the test needs to specify a proper broker URL before publishing any events.

7.12.7.1. Knative broker URL

@Given("Knative broker URL: {url}$")

Given Knative broker URL: {url}

The URL points to a Knative broker and uses Http as transport. The test is able to publish events using this broker endpoint.

7.12.7.2. Knative client

As an alternative to that you can also specify a Http client component which connects to the broker.

@Given("Knative client "{name}"$")

Given Knative client "{name}"

The client references a component in the configuration (e.g. Spring bean).

Now the test is ready to publish the event.

7.12.7.3. Create cloud events

@When("create|send Knative event$")

When send Knative event
| {property} | {value} |

The step uses a data table in order to specify the cloud event properties that should be published. The cloud event data structure defines following properties:

- specversion
- type
- source
- subject
- id
- datacontenttype
- data

Following these properties you can specify the cloud event in the send operation.
Send cloud event

<table>
<thead>
<tr>
<th>When</th>
<th>send Knative event</th>
</tr>
</thead>
<tbody>
<tr>
<td>specversion</td>
<td>1.0</td>
</tr>
<tr>
<td>type</td>
<td>greeting</td>
</tr>
<tr>
<td>source</td>
<td><a href="https://github.com/citrusframework/yaks">https://github.com/citrusframework/yaks</a></td>
</tr>
<tr>
<td>subject</td>
<td>hello</td>
</tr>
<tr>
<td>id</td>
<td>say-hello</td>
</tr>
<tr>
<td>datacontenttype</td>
<td>application/json</td>
</tr>
<tr>
<td>data</td>
<td>{&quot;msg&quot;: &quot;Hello Knative!&quot;}</td>
</tr>
</tbody>
</table>

The **data** property defines the cloud event payload which is a Json payload in the example above. This can be any payload and depends on what you want to send as part of the event.

As we are using the Http cloud event model we can also use Http property equivalents as property keys.

Send cloud event via Http properties

<table>
<thead>
<tr>
<th>When</th>
<th>send Knative event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ce-Specversion</td>
<td>1.0</td>
</tr>
<tr>
<td>Ce-Type</td>
<td>greeting</td>
</tr>
<tr>
<td>Ce-Source</td>
<td><a href="https://github.com/citrusframework/yaks">https://github.com/citrusframework/yaks</a></td>
</tr>
<tr>
<td>Ce-Subject</td>
<td>hello</td>
</tr>
<tr>
<td>Ce-Id</td>
<td>say-hello-${id}</td>
</tr>
<tr>
<td>Content-Type</td>
<td>application/json;charset=UTF-8</td>
</tr>
<tr>
<td>data</td>
<td>{&quot;msg&quot;: &quot;Hello Knative!&quot;}</td>
</tr>
</tbody>
</table>

Instead of using a **data** property in the data table you can also specify the event payload in a separate step.

@Given("^Knative event data: {data}$")

**Given Knative event data: {data}**

The step sets a single line event data that is going to represent the payload of the cloud event.

The following step supports multiline event data.

@Given("^Knative event data$")

**Given Knative event data**

```text
""
<<data>>
""
```

With these steps the cloud event data table must not specify the **data** property anymore.
Given Knative event data

```
{
  "msg": "Hello Knative!"
}
```

Then send Knative event

| specversion | 1.0 |
| type         | greeting |
| source       | https://github.com/citrusframework/yaks |
| subject      | hello |
| id           | say-hello |
| datacontenttype | application/json |

7.12.7.4. Create cloud events via Json

The cloud events model supports Json so you can also specify the event with a single step in Json.

@When("^(?:create|send) Knative event as json$")

```
When send Knative event as json

<<<json>>
```

Send cloud event via Json

```
When send Knative event as json

{
  "specversion": "1.0",
  "type": "greeting",
  "source": "https://github.com/citrusframework/yaks",
  "subject": "hello",
  "id": "say-hello",
  "datacontenttype": "application/json",
  "data": "{\"msg\": \"Hello Knative!\"}"
}
```

7.12.7.5. Producer timeouts

The producer connects to the Knative broker in order to publish events. In case the broker is not available a timeout will fail the test. You can adjust the producer timeout.
7.12.8. Receive events

In order to receive events from Knative you should set up a service or channel in combination with a trigger. The trigger watches for events on the broker and forwards these to the service or channel.

The test is able to receive events and verify its content.

7.12.8.1. Receive cloud events

The step uses a data table in order to specify the cloud event properties as expected content. The cloud event data structure defines the following properties:

- specversion
- type
- source
- subject
- id
- datacontenttype
- data

Following these properties you can specify the cloud event in the receive operation.

The data property defines the cloud event payload which is a Json payload in the example above. This can be any payload and depends on what you want to receive as part of the event.

As we are using the Http cloud event model we can also use Http property equivalents as property
keys.

Receive cloud event via Http properties

| Ce-Specversion    | 1.0 |
| Ce-Type           | greeting |
| Ce-Source         | https://github.com/citrusframework/yaks |
| Ce-Subject        | hello |
| Ce-Id             | say-hello-${id} |
| Content-Type      | application/json;charset=UTF-8 |
| data              | {"msg": "Hello Knative!"} |

Instead of using a `data` property in the data table you can also specify the event payload in a separate step.

@Then("^(?:expect|verify) Knative event data: {data}$")

Then expect Knative event data: {data}

The step sets a single line event data that is going to represent the payload of the cloud event.

The following step supports multiline event data.

@Then("^(?:expect|verify) Knative event data$")

Then expect Knative event data

```<<data>>
Hello Knative!
```

With these steps the cloud event data table must not specify the `data` property anymore.

Receive cloud event

| specversion    | 1.0 |
| type           | greeting |
| source         | https://github.com/citrusframework/yaks |
| subject        | hello |
| id             | say-hello |
| datacontenttype | application/json |
7.12.8.2. Receive cloud events via Json

The cloud events model supports Json so you can also specify the event with a single step in Json.

@When("^(?:receive|verify) Knative event as json$")

```java
Then receive Knative event as json
***
<<json>>
***
```

Receive cloud event via Json

```java
Then receive Knative event as json
***
{
    "specversion" : "1.0",
    "type" : "greeting",
    "source" : "https://github.com/citrusframework/yaks",
    "subject" : "hello",
    "id" : "say-hello",
    "datacontenttype" : "application/json",
    "data" : "{"msg": "Hello Knative!"}"
}
***
```

7.12.8.3. Consumer timeouts

The consumer connects to the Knative broker in order to consume events. The consumer will wait for events and in case no event arrives in time a timeout will fail the test. You can adjust this event consumer timeout.

@Given("^Knative event consumer timeout is {time}(?: ms| milliseconds)$")

```java
Given Knative event consumer timeout is {time} milliseconds
```

7.13. Open API steps

OpenAPI documents specify RESTful Http services in a standardized, language-agnostic way. The specifications describe resources, path items, operations, security schemes and many more components. All these components specified are part of a REST Http service.

YAKS as a framework is able to use this information in an OpenAPI document in order to generate proper request and response data for your test.

You can find examples of how to use OpenAPI specifications in examples/openapi.
Feature: Petstore API V3

Background:
Given OpenAPI specification: http://localhost:8080/petstore/v3/openapi.json

Scenario: getPet
When invoke operation: getPetById
Then verify operation result: 200 OK

Scenario: petNotFound
Given variable petId is "0"
When invoke operation: getPetById
Then verify operation result: 404 NOT_FOUND

Scenario: addPet
When invoke operation: addPet
Then verify operation result: 201 CREATED

Scenario: updatePet
When invoke operation: updatePet
Then verify operation result: 200 OK

Scenario: deletePet
When invoke operation: deletePet
Then verify operation result: 204 NO_CONTENT

7.13.1. Load OpenAPI specifications

The test is able to load OpenAPI specifications via Http URL or the local file system. When loaded into the test steps can make use of all available operations in the specification.

@Given("^OpenAPI (?:specification|resource): {url}$")

Given OpenAPI specification {url}

The given url can point to a local file on the file system or to a Http endpoint. The step loads the OpenAPI specification so all operations are ready to be used.

7.13.2. Invoke operations

You can invoke operations by referencing its name/id in the specification. YAKS will automatically generate proper request/response data for you.

The rules in the OpenAPI specification help to generate proper request/response data with randomized values.
When invoke operation: {id}

The step obtains the operation with the given {id} and invokes it on the Http URL that is given in the specification. In case not server URL is given in the specification the step uses the base URL of the OpenAPI endpoint where the document has been loaded from.

The step uses the specification rules for the operation to generate a proper request for you. Request parameters, headers and body content are automatically generated. In case the operation defines a body the step will generate it with randomized values.

Generated body example with randomized values

```
{
    "id": 26866048,
    "name": "mGNTgkfxgg",
    "photoUrls": [
        "XHAGIyFcyh"
    ],
    "category": {
        "name": "konwOUywMo",
        "id": 18676332
    },
    "tags": [
        {
            "name": "KDnoWCfUbn",
            "id": 31444049
        }
    ],
    "status": "sold"
}
```

The generated request should be valid according to the rules in the OpenAPI specification. You can overwrite the randomized values with test variables and inbound/outbound data dictionaries in order to have more human readable test data.

Now that the test has sent the request you can verify the operation result.

### 7.13.3. Verify operation result

The test is able to verify the response status code returned by the server.

Then verify operation result: {status_code} {reason_phrase}

The step expects a {status_code} (e.g. 200, 404, 500) and optionally gives the {reason_phrase} (e.g. OK, NOT_FOUND, INTERNAL_SERVER_ERROR). The reason phrase is optional and is only for better
The operation defines a set of responses in the OpenAPI specification. The step tries to find the response with the given `{status_code}` and reads the given rules such as response body, headers etc. Based on the response definition in the OpenAPI specification the step automatically verifies the server response and makes sure that the response matches the given rules.

In particular the step generates an expected response body (if any is specified) and compares the actual response with the generated one.

**Generated response body example with validations**

```json
{
    "id": "@isNumber()@",
    "name": "@notEmpty()@",
    "photoUrls": "@ignore@",
    "category": {
        "id": "@isNumber()@",
        "name": "@notEmpty()@"
    },
    "tags": "@ignore@",
    "status": "@matches(available|pending|sold)@"
}
```

The generated response makes use of Citrus validation matchers based on the rules in the specification. Id values are validated with `@isNumber()@`, String values should not be empty `@notEmpty()@` and enumeration values are checked with `@matches(value_1|value_2|⋯|value_x)@`.

The received response must match all these validation matchers. In addition to that a Json schema validation is performed on the response.

### 7.13.4. Request fork mode

When the OpenAPI steps fire requests to the server the step synchronously waits for the response. All other steps are in the feature are blocked by the synchronous communication. In some cases this is a problem because you might want to run some steps in parallel to the synchronous communication.

In these cases you can make use of the form mode when sending Http client requests.

@Given("^OpenAPI request fork mode is (enabled|disabled)$")

**Given** OpenAPI request fork mode is enabled

With this in place the step will not block other steps while waiting for the synchronous response from the server. The feature will continue with the next steps when fork mode is enabled. At a later point in time you may verify the response as usual with the separate verification step.
7.13.5. Inbound/outbound data dictionaries

Data dictionaries are a good way to make generated randomized values more human readable. By default YAKS generates random values based on the specifications in the OpenAPI document. You can overwrite the basic generation rules by specifying rules in a data dictionary.

7.13.5.1. Outbound dictionary

Outbound dictionaries are used to customize generated client requests.

@Given("^outbound dictionary\$")

```plaintext
Given outbound dictionary
| {expression} | {value} |
```

The outbound dictionary holds a list of expressions that overwrite values in the generated request body.

Based on the body data format (e.g. Json or XML) you can use JsonPath or XPath expressions in the dictionary. YAKS evaluates the given expressions on the generated request body before the request is sent to the server.

**Outbound dictionary sample**

```plaintext
Given outbound dictionary
| $.name        | citrus:randomEnumValue('hasso','cutie','fluffy') |
| $.category.name| citrus:randomEnumValue('dog', 'cat', 'fish') |
```

You can also load the dictionary rules from an external file resource.

@Given("'^load outbound dictionary {file_path}$'\")

```plaintext
Given load outbound dictionary {file_path}
```

With this outbound data dictionary in place a generated request can look like follows:
You see that the request now uses more human readable values for $.name and $.category.name.

The same mechanism applies for inbound messages that are verified by YAKS. The framework will generate an expected response data structure coming from the OpenAPI specification.

### 7.13.5.2. Inbound dictionary

Inbound dictionaries adjust the generated expected responses which verify incoming messages with expected validation statements.

@Given("^inbound dictionary$")

```regex
| {expression} | {value} |
```

You can also load the dictionary rules from an external file resource.

@Given("^load inbound dictionary {file_path}$")

```regex
Given load inbound dictionary {file_path}
```

The inbound dictionary holds a list of expressions that overwrite values in the generated response body.

Based on the body data format (e.g. Json or XML) you can use JsonPath or XPath expressions in the dictionary. YAKS evaluates the given expressions on the generated response body. This way you can overwrite given values in the body structure before the validation takes place.
Inbound dictionary sample

**Given** inbound dictionary

| $.name    | @assertThat(anyOf(is(hasso),is(cutie),is(fluffy)))@ |
| $.category.name | @assertThat(anyOf(is(dog),is(cat),is(fish)))@ |

Below is a sample Json payload that has been generated with the inbound data dictionary.

**Generated response with inbound dictionary**

```
{
    "id": "@isNumber()@",
    "name": "@assertThat(anyOf(is(hasso),is(cutie),is(fluffy)))@",
    "photoUrls": "@ignore@",
    "category": {
        "name": "@assertThat(anyOf(is(dog),is(cat),is(fish)))@",
        "id": "@isNumber()@"
    },
    "tags": "@ignore@",
    "status": "@matches(available|pending|sold)@"
}
```

The generated response ensures that the rules defined in the OpenAPI specification do match and in addition that the received data meets our expectations in the dictionary.

In case you need to have a more specific response validation where each field gets validated with an expected value please consider using the **Http steps** in YAKS. Here you can provide a complete expected Http response with body and headers.
Chapter 8. Extensions

8.1. Minio upload

Extensions add custom steps to the test runtime so you can use custom step definitions in your feature file.

extension.feature

```plaintext
Scenario: print extended slogan
  Given YAKS does Cloud-Native BDD testing
  Then YAKS can be extended!
```

The step `YAKS can be extended!` is not available in the default step implementations provided by YAKS. The step definition is implemented in a separate custom Maven module and gets uploaded to the Kubernetes cluster using the `container-tools/snap` library.

Snap uses a Minio object storage that is automatically installed in the current namespace. You can build and upload custom Maven modules with:

```
$ yaks upload examples/extensions/steps
```

This will create the Minio storage and perform the upload. After that you can use the custom steps in your feature file. Be sure to add the dependency and the additional glue code in `yaks-config.yaml`.

`yaks-config.yaml`

```yaml
config:
  runtime:
    cucumber:
      glue:
        - "org.citrusframework.yaks"
        - "com.company.steps.custom"
  dependencies:
    - groupId: com.company
      artifactId: steps
      version: "1.0.0-SNAPSHOT"
```

The additional glue code should match the package name where to find the custom step definitions in your custom code.

With that you are all set and can run the test as usual:

```
$ yaks test extension.feature
```
You can also use the upload as part of the test command:

```
$ yaks test extension.feature --upload steps
```

The `--upload` option builds and uploads the custom Maven module automatically before the test.

### 8.2. Jitpack extensions

Jitpack allows you to load custom steps from an external GitHub repository in order to use custom step definitions in your feature file.

**jitpack.feature**

```
Scenario: Use custom steps
  Given My steps are loaded
  Then I can do whatever I want!
```

The steps *My steps are loaded* and *I can do whatever I want!* live in a separate repository on GitHub ([https://github.com/citrusframework/yaks-step-extension](https://github.com/citrusframework/yaks-step-extension)).

We need to add the Jitpack Maven repository, the dependency and the additional glue code in the `yaks-config.yaml`.

**yaks-config.yaml**

```
config:
  runtime:
    cucumber:
      glue:
        - "org.citrusframework.yaks"
        - "dev.yaks.testing.standard"
  settings:
    repositories:
      - id: "central"
        name: "Maven Central"
        url: "https://repo.maven.apache.org/maven2/"
      - id: "jitpack.io"
        name: "JitPack Repository"
        url: "https://jitpack.io"
  dependencies:
    - groupId: com.github.citrusframework
      artifactId: yaks-step-extension
      version: "0.0.1"
```

The additional glue code `dev.yaks.testing.standard` should match the package name where to find the custom step definitions in the library. The Jitpack Maven repository makes sure the library gets resolved at runtime.
With that you are all set and can run the test as usual:

```bash
$ yaks test jitpack.feature
```

In the logs you will see that Jitpack automatically loads the additional dependency before the test.
Chapter 9. Pre/Post scripts

You can run scripts before/after a test group. Just add your commands to the `yaks-config.yaml` configuration for the test group.

```yaml
config:
  namespace:
    temporary: false
    autoRemove: true
  pre:
    - script: prepare.sh
    - run: echo Start!
    - name: Optional name
      timeout: 30m
      run: |
        echo "Multiline"
        echo "Commands are also"
        echo "Supported!"
  post:
    - script: finish.sh
    - run: echo Bye!
```

The section `pre` runs before a test group and `post` is added after the test group has finished. The post steps are run even if the tests or pre steps fail for some reason. This ensures that cleanup tasks are performed also in case of errors.

The `script` option provides a file path to bash script to execute. The user has to make sure that the script is executable. If no absolute file path is given it is assumed to be a file path relative to the current test group directory.

With `run` you can add any shell command. At the moment only single line commands are supported here. You can add multiple `run` commands in a `pre` or `post` section.

Each step can also define a human readable `name` that will be printed before its execution.

By default a step must complete within 30 minutes (`30m`). The timeout can be changed using the `timeout` option in the step declaration (in Golang duration format).

Scripts can leverage the following environment variables that are set automatically by the Yaks runtime:

- `YAKS_NAMESPACE`: always contains the namespace where the tests will be executed, no matter if the namespace is fixed or temporary
Chapter 10. Reporting

After running some YAKS tests you may want to review the test results and generate a summary report. As we are using CRDs on the Kubernetes or OpenShift platform we can review the status of the custom resources after the test run in order to get some test results.

```
oc get tests
NAME    PHASE     TOTAL     PASSED    FAILED    SKIPPED
helloworld    Passed    2         2         0         0
foo-test       Passed    1         1         0         0
bar-test       Passed    1         1         0         0
```

You can also view error details when adding the `wide` option

```
oc get tests -o wide
NAME    PHASE     TOTAL     PASSED    FAILED    SKIPPED    ERRORS
helloworld    Passed    2         1         1         0          [ "helloworld.feature:10 Failed caused by ValidationException - Expected 'foo' but was 'bar'" ]
foo-test       Passed    1         1         0         0
bar-test       Passed    1         1         0         0
```

The YAKS CLI is able to fetch those results in order to generate a summary report locally:

```
yaks report --fetch
Test results: Total: 4, Passed: 4, Failed: 0, Skipped: 0
  classpath:org/citrusframework/yaks/helloworld.feature:3: Passed
  classpath:org/citrusframework/yaks/helloworld.feature:7: Passed
  classpath:org/citrusframework/yaks/foo-test.feature:3: Passed
  classpath:org/citrusframework/yaks/bar-test.feature:3: Passed
```

The report supports different output formats (summary, json, junit). For JUnit style reports use the `junit` output.
The JUnit report is also saved to the local disk in the file `_output/junit-reports.xml`. The `_output` directory is also used to store individual test results for each test executed via the YAKS CLI. So after a test run you can also review the results in that `_output` directory. The YAKS report command can also view those results in `_output` directory in any given output format. Simply leave out the `--fetch` option when generating the report and YAKS will use the test results stored in the local `_output` folder.

```
yaks report
Test results: Total: 5, Passed: 5, Failed: 0, Skipped: 0
  classpath:org/citrusframework/yaks/helloworld.feature:3: Passed
  classpath:org/citrusframework/yaks/helloworld.feature:7: Passed
  classpath:org/citrusframework/yaks/test1.feature:3: Passed
  classpath:org/citrusframework/yaks/test2.feature:3: Passed
  classpath:org/citrusframework/yaks/test3.feature:3: Passed
```
Chapter 11. Contributing

Requirements:

- Go 1.13+
- Operator SDK 0.19.4+
- Maven 3.6.2+
- Git client

You can build the YAKS project and get the `yaks` CLI by running:

```
make build
```

If you want to build the operator image locally for development in Minishift for instance, then:

```
# Build binaries and images
eval $(minishift docker-env)
make clean images-no-test
```

If the operator pod is running, just delete it to let it grab the new image.

```
oc delete pod yaks
```
Chapter 12. Uninstall

In case you really need to remove YAKS and all related resources from Kubernetes or OpenShift you can do so with the following command:

```
yaks uninstall
```

This will remove the YAKS operator from the current namespace along with all related custom resource definitions.

When using the global operator mode you may need to select the proper namespace here.

```
yaks uninstall -n kube-operators
```

By default, the uninstall will **not** remove resources that are possibly shared between namespaces and clusters (e.g. CRDs and roles). Please use the `--all` flag if you need to wipe out these, too.

```
yaks uninstall -n kube-operators --all
```

The `--all` option removes the operator and all related resources such as **CustomResourceDefinitions (CRD)** and **ClusterRole**.

In case the operator has **not** been installed via **Operator Lifecycle Manager (OLM)** you may need to use the option `--olm=false` also when uninstalling. In particular this is the case when installing YAKS from sources on **CRC**.

```
yaks uninstall --olm=false
```

Use this whenever you do not want to use OLM framework for performing the uninstall.
Chapter 13. Samples

ToDo