YAKS

Authors: The YAKS Community

Version 0.2.0-SNAPSHOT, 2020-11-27

yaks

Version: 0.2.0-SNAPSHOT

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:

helloworld.feature

```
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:

```
# 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:

```
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:

```
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:

```
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
```

```
NAME                              CREATED AT
tests.yaks.citrusframework.org    2020-11-01T00:00:00Z
```

The following command will list all tests in your namespace:

```
kubectl get tests
```

```
NAME         PHASE    TOTAL   PASSED   FAILED   SKIPPED   ERRORS
helloworld   Passed   1       1        0        0
```

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:

```
test "helloworld" created
+ test-helloworld  test
test-helloworld test INFO   |
test-helloworld test INFO   |
------------------------------------------------------------------------
test-helloworld test INFO   |        .__  __
test-helloworld test INFO   |   ____ |__|/  |________ __ __  ______
test-helloworld test INFO   | _/ ___\|  \   __\_  __ \  |  \/  ___/
test-helloworld test INFO   | \  \___|  ||  |  |  | \/  |  /\___ \
test-helloworld test INFO   |  \___  >__||__|  |__|  |____//____  >
test-helloworld test INFO   |      \/                           \/
test-helloworld test INFO   |
test-helloworld test INFO   | C I T R U S  T E S T S  3.0.0-M2
test-helloworld test INFO   |
test-helloworld test INFO   |
------------------------------------------------------------------------
test-helloworld test INFO   |
test-helloworld test
test-helloworld test Scenario: Print hello message    #
org/citrusframework/yaks/helloworld.feature:3
test-helloworld test   Given print 'Hello from YAKS!' #
org.citrusframework.yaks.standard.StandardSteps.print(java.lang.String)
test-helloworld test INFO   |
------------------------------------------------------------------------
test-helloworld test INFO   |
test-helloworld test INFO   | CITRUS TEST RESULTS
test-helloworld test INFO   |
test-helloworld test INFO   |  Print hello message
..................................... SUCCESS
test-helloworld test INFO   |
test-helloworld test INFO   | TOTAL:    1
test-helloworld test INFO   | FAILED:   0 (0.0%)
test-helloworld test INFO   | SUCCESS:  1 (100.0%)
test-helloworld test INFO   |
test-helloworld test INFO   |
------------------------------------------------------------------------
test-helloworld test
test-helloworld test 1 Scenarios (1 passed)
test-helloworld test 1 Steps (1 passed)
test-helloworld test 0m1.631s
test-helloworld test
test-helloworld test
Test Passed
Test results: Total: 1, Passed: 1, Failed: 0, Skipped: 0
    Print hello message (helloworld.feature:3): Passed
```

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`.

```
yaks test helloworld.feature --logger root=INFO
```

The logging configuration section in thi 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
```

```
NAME         PHASE    TOTAL   PASSED   FAILED   SKIPPED   ERRORS
helloworld   Passed   1       1        0        0
```

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/citrusfrmaework/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

Name	Description	Example
help	Obtain the full list of available commands	`yaks help`
test	Run a test on Kubernetes	`yaks test helloworld.feature`

The list above is not the full list of available commands. You can run `yaks help` to obtain the full list. And each command also takes `--help` as parameter to output more information:

```
yaks test --help
```

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.

```
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.

```
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.

```
@require('org.apache.camel:camel-groovy:@camel.version@')
Feature: Camel route testing

  Background:
    Given Camel route hello.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.`

```
# 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:

```
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:

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

```
{
  "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 a 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:

```yaml
repositories:
  - id: "central"
    name: "Maven Central"
    url: "https://repo.maven.apache.org/maven2/"
    releases:
      enabled: "true"
      updatePolicy: "daily"
    snapshots:
      enabled: "false"
  - id: "jboss-ea"
    name: "JBoss Community Early Access Release Repository"
    url: "https://repository.jboss.org/nexus/content/groups/ea/"
    layout: "default"
```

```json
{
  "repositories": [
      {
        "id": "central",
        "name": "Maven Central",
        "url": "https://repo.maven.apache.org/maven2/",
        "releases": {
          "enabled": "true",
          "updatePolicy": "daily"
        },
        "snapshots": {
          "enabled": "false"
        }
      },
      {
        "id": "jboss-ea",
        "name": "JBoss Community Early Access Release Repository",
        "url": "https://repository.jboss.org/nexus/content/groups/ea/",
        "layout": "default"
      }
    ]
}
```

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:

my-test.feature

```
Feature: JDBC API

  Background:
    Given Database connection
      | url       | ${database.url} |
      | username  | ${database.user} |
      | password  | ${database.password} |
```

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 provide a lot of basic features and predefined steps that you can use to write feature files. Most of the steps aim to leverage capabilities of the underlying test framework Citrus. For instance the steps are able to create Citrus test variables or print messages to the output.

7.1.1. Variable steps

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

create variable

```
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:

json payload

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

You can create multiple variables in one single step using:

creating variables

```
Given variables
  | orderId  | 1001      |
  | name     | Pineapple |
```

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:

Step	Parameter	Description
`print '<any text>'`	any text that should be printed to the output	Printing messages to the output. Supports variables and functions in text.
`log '<any text>'`	any text that should be logged	Log messages to the output. Supports variables and functions in text.

print/log messages

```
Scenario: log messages
  Then print 'YAKS rocks!'

Scenario: multiline log messages
  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.

Step	Parameter	Description
`sleep`	-	Sleep the default time of 5000 milliseconds.
`sleep <time> ms`	time in milliseconds	Sleep given amount of time in milliseconds.

sleep

```
Scenario: sleep time period
  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

ToDo

7.3. Apache Camel K steps

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

For example:

```
...
Given integration xxx is running
Then integration xxx should print Hello world!
```

The Camel K extension library is provided by default in YAKS.

7.4. Groovy steps

The Groovy support in YAKS allows to add framework configuration, bean configuration and test actions via script snippets. In particular you can easily add customized endpoints that send/receive data over various messaging transports.

7.4.1. Framework configuration

You can add endpoints and beans as Citrus framework configuration like follows:

```
Scenario: Endpoint script config
  Given URL: http://localhost:18080
  Given create configuration
  """
  citrus {
      endpoints {
          http {
              server('helloServer') {
                  port = 18080
                  autoStart = true
              }
          }
      }
  }
  """
  When send GET /hello
  Then receive HTTP 200 OK
```

In the example above the scenario 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 component that is automatically started with 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
  Given create configuration
  """
  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.

```
Scenario: Endpoint script config
  Given load configuration citrus.configuration.groovy
  When endpoint hello sends payload Hello from new direct endpoint!
  Then endpoint hello should receive payload Hello from new direct endpoint!
```

citrus.configuration.groovy

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

    endpoints {
        direct {
            asynchronous {
                name = 'hello'
                queue = 'say-hello'
            }
        }
    }
}
```

7.4.2. 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 so you exchange and verify message data. Endpoint

Groovy scripts is one comfortable way to add custom endpoint configurations in a test scenario.

```
Scenario: Create Http endpoint
  Given URL: http://localhost:18081
  Given create endpoint helloServer.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 external file resources.

```
Scenario: Load endpoint
  Given URL: http://localhost:18088
  Given load endpoint fooServer.groovy
  When send GET /hello
  Then receive HTTP 200 OK
`----

.fooServer.groovy
[source]
```

http() .server() .port(18088) .autoStart(true) ``----

7.4.3. 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:

```
Scenario: Custom test actions
  Given create actions basic.groovy
  """
  actions {
    echo('Hello from Groovy script')
    sleep().seconds(1)

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

    echo('Variable foo=${foo}')
  }
  """
  Then apply basic.groovy
```

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 to perform complex actions such as iterations, conditionals and send/receive operations.

```
Scenario: Messaging actions
  Given create actions messaging.groovy
  """
  actions {
    send('direct:myQueue')
      .payload('Hello from Groovy script!')

    receive('direct:myQueue')
      .payload('Hello from Groovy script!')
  }
  """
  Then apply messaging.groovy
```

7.5. 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 during the test.

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

```
Feature: Http client

  Background:
    Given URL: http://localhost:8080

  Scenario: Health check
    Given path /health is healthy

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

The example above sets a base request URL to `http://localhost:8080` and performs a health check on path `/health`. After that we can send any request to the server and verify the response body and status code.

All these steps are part of the core YAKS framework and you can just use them.

On the server side we can start a new Http server instance on a given port and listen for incoming requests. These request can be verified and the test can provide a simulated response message with body and header data.

```
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)`.

Once the request is verified the server responds with a simple Http `201 CREATED`.

7.6. JDBC steps

YAKS provides a library that allows to execute SQL actions on relational DBs (limited to PostgreSQL for this POC).

You can find examples of JDBC steps in the examples file.

There's also an example that uses JDBC and REST together and targets the Syndesis TODO App database.

7.7. JMS steps

ToDo

7.8. Kafka steps

ToDo

7.9. Kubernetes steps

ToDo

7.10. Knative steps

ToDo

7.11. 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 that are part of the REST service. YAKS as a framework is able to use these information in order to generate proper request and response data for your test.

You can find examples of how to use OpenAPI specifications in YAKS in the examples.

Given an OpenAPI specification that you can access via Http URL or local file system you can load all available operations into the test. Once this is completed you can invoke operations by name and verify the response status codes. YAKS will automatically generate proper request/response data for you.

```
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
```

The request/response data is generated from the OpenAPI specification rules and holds randomized values. The following sample shows a generated request for the `addPet` operation where a new pet is transmitted via Http POST.


```
{
  "photoUrls": [
    "XHAGIyFcyh"
  ],
  "name": "mGNTgkfxgg",
  "id": 26866048,
  "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.

```
Feature: Petstore API V3

  Background:
    Given OpenAPI specification: http://localhost:8080/petstore/v3/openapi.json
    Given variable petId is "citrus:randomNumber(5)"
    Given inbound dictionary
      | $.name          | @assertThat(anyOf(is(hasso),is(cutie),is(fluffy)))@ |
      | $.category.name | @assertThat(anyOf(is(dog),is(cat),is(fish)))@ |
    Given outbound dictionary
      | $.name          | citrus:randomEnumValue('hasso','cutie','fluffy') |
      | $.category.name | citrus:randomEnumValue('dog', 'cat', 'fish') |

  [...]
```

With this data dictionaries in place the generated request looks like follows:

```
{
  "photoUrls": [
    "aaKoEDhLYc"
  ],
  "name": "hasso",
  "id": 12337393,
  "category": {
    "name": "cat",
    "id": 23927231
  },
  "tags": [
    {
      "name": "FQxvuCbcqT",
      "id": 58291150
    }
  ],
  "status": "pending"
}
```

You see that we are now using 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. Below is a sample Json payload that verifies the response for the `getPetById` operation.

```
{
  "photoUrls": "@ignore@",
  "name": "@assertThat(anyOf(is(hasso),is(cutie),is(fluffy)))@",
  "id": "@isNumber()@",
  "category": {
    "name": "@assertThat(anyOf(is(dog),is(cat),is(fish)))@",
    "id": "@isNumber()@"
  },
  "tags": "@ignore@",
  "status": "@matches(available|pending|sold)@"
}
```

All mandatory fields need to be in the received json document. Also enumerations and number values are checked to meet the expected values coming form the OpenAPI specification (e.g. `status=@matches(available|pending|sold)@`). This ensures that the response respects the rules defined in the specification.

In case you also want to validate the exact values on each field please use the generic Http steps where you can provide a complete expected Http response with payload and header data.

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

```
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

```
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).

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:

```
$ 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.

```
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.

```
yaks report --fetch --output junit

<?xml version="1.0" encoding="UTF-8"?><testsuite
name="org.citrusframework.yaks.JUnitReport" errors="0" failures="0" skipped="0"
tests="4" time="0">
  <testcase name="helloworld.feature:3"
classname="classpath:org/citrusframework/yaks/helloworld.feature:3"
time="0"></testcase>
  <testcase name="helloworld.feature:7"
classname="classpath:org/citrusframework/yaks/helloworld.feature:7"
time="0"></testcase>
  <testcase name="foo-test.feature:3"
classname="classpath:org/citrusframework/yaks/foo-test.feature:3" time="0"></testcase>
  <testcase name="bar-test.feature:3"
classname="classpath:org/citrusframework/yaks/bar-test.feature:3" time="0"></testcase>
</testsuite>
```

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