Testing using Serenity BDD and Cucumber in Java

- Overview
- Requirements
- Description
  - Using Jira and Xray as master
  - Using Git or other VCS as master
- References

Overview

Serenity BDD is a framework for assisting in automated acceptance testing using BDD.

It provides the ability to write executable specifications, run them and produce comprehensive reports.

In this tutorial, we will create some tests using Serenity BDD along with Cucumber. The specification will be done using standard Cucumber .feature files, where each test is written as a Scenario or Scenario Outline. The corresponding steps implementation will be done in Java.

The tutorial details slightly different instructions depending on where you want to perform the edition of your features and corresponding scenarios (please check the possible workflows).

Requirements

- serenity-bdd
- cucumber
- chromedriver (a version that supports your current Chrome version)
- maven
- Chrome

Description

This tutorial is highly based on an existing Serenity+Cucumber quick start project with some minor changes.

The business-readable tests aim to validate a search engine using some examples that interact with it using Selenium WebDriver and Chrome.

Please note

The code used for this tutorial can be found here; you may also find the original unchanged project here.

The serenity configuration file can be used as such but it can be updated to customize certain Serenity behaviours.

serenity.properties

```
serenity.project.name=Serenity and Cucumber Quick Start
```

Even though you could follow the page-objects pattern, Serenity favors the Screenplay pattern. Thus, instead of abstracting every single page as a class using the page-objects pattern, users are advised to implement classes that abstract an actor/personna that interacts with the application.

These actors can perform business-understandable actions/tasks, also known as steps. In code they should have the @Step annotation, so they can be understood as such and appear in the reports, for example.

One can see each actor/personna related class as a step library. A step library adds a layer of abstraction between the "what" and the "how" of our acceptance tests.

Multiple step libraries can be used to provide the building blocks for writing the our executable test specification.

Please note

Steps should be focused in the "what" we are aiming to achieve and not not on the "how". A step can, in turn, invoke other more technical methods that implement the "how".

Whenever using Cucumber along with Serenity, Cucumber step definitions are used as an additional layer of abstraction on top of standard step libraries.

Methods implementing them use the typical Gherkin @Given, @When, @Then annotations from the Cucumber library.

```
   @Steps
   NavigateTo navigateTo;
...
   @When("^s?he searches for \"(.*)\"")
    public void i_search_for(String term) {
        searchFor.term(term);  // this
    }
```

In this tutorial, all steps are defined and referenced from within a class. Some variables have the @Steps annotation, so their respective class has also business-related steps.

src/test/java/starter/stepdefinitions/SearchOnDuckDuckGoStepDefinitions.java

```
package starter.stepdefinitions;

import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import net.thucydides.core.annotations.Steps;
import starter.navigation.NavigateTo;
import starter.search.SearchFor;
import starter.search.SearchResult;

import static org.assertj.core.api.Assertions.assertThat;
import static starter.matchers.TextMatcher.textOf;

public class SearchOnDuckDuckGoStepDefinitions {

    @Steps
    NavigateTo navigateTo;

    @Steps
    SearchFor searchFor;

    @Steps
    SearchResult searchResult;

    @Given("^(?:.*) is on the DuckDuckGo home page")
    public void i_am_on_the_DuckDuckGo_home_page() {
        navigateTo.theDuckDuckGoHomePage();
    }

    @When("^s?he searches for \"(.*)\"")
    public void i_search_for(String term) {
        searchFor.term(term);
    }

    @Then("^all the result titles should contain the word \"(.*)\"")
    public void all_the_result_titles_should_contain_the_word(String term) {
        assertThat(searchResult.titles())
                .matches(results -> results.size() > 0)
                .allMatch(title -> textOf(title).containsIgnoringCase(term));
    }
}
```

By default, standard Cucumber .feature files live in the `src/test/resources/features` directory.

However, this can be customized as a option to the runner class.

src/test/java/starter/CucumberTestSuite.java

```
...
@RunWith(CucumberWithSerenity.class)
@CucumberOptions(
        features = "features/",
        plugin =    {
                "pretty", "html:target/serenity-reports//serenity-html-report",
                "json:target/serenity-reports/cucumber_report.json",
                "rerun:target/serenity-reports/rerun.txt"
        }
)
...
```

It can also be enforced whenever running maven from the command line using a system property (e.g. `-Dcucumber.features="features/"`).

We will also configure the runner to generate a Cucumber JSON report containing test results that can be processed by Xray.

We've updated slightly the feature from the upstream project, to make the two scenarios a bit more different. You can also see a tag before the "Feature", which gives the ability to automatically link the scenarios to some existing story/requirement in Jira.

src/test/resources/features/search/search_by_keyword.feature

```
@REQ_CALC-6399
Feature: Search by keyword

  @cucumber @green
  Scenario: Searching for a food term
    Given Sergey is on the DuckDuckGo home page
    When he searches for "cucumber food"
    Then all the result titles should contain the word "recipes"

  @cucumber @brown
  Scenario: Searching for a gherkin
    Given Sergey is on the DuckDuckGo home page
    When he searches for "cucumber"
    Then all the result titles should contain the word "cucumber"
```

Remember that we need to manage:

- features (declarative specifications, usually stored in .feature files)
- their implementation

Besides that, you need to decide is which workflow we'll use: do we want to use Xray/Jira as the master for writing the declarative specification or do we want to manage those in Git, for example?

Learn more

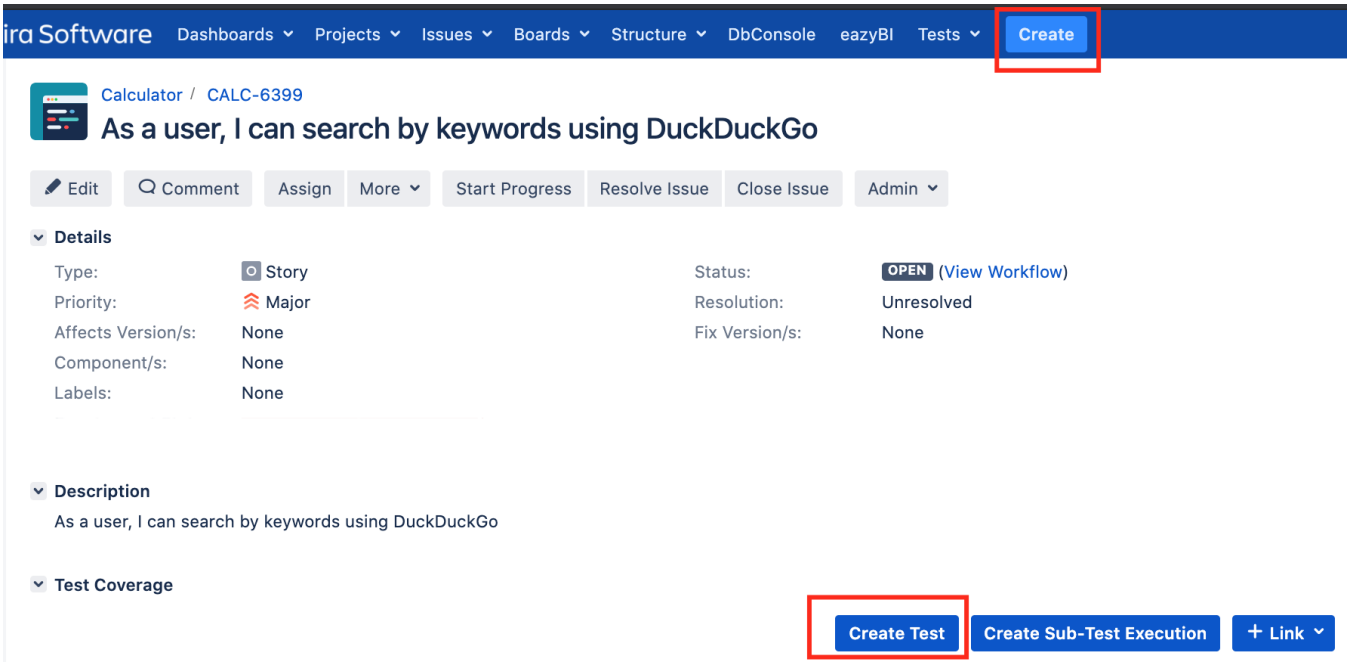
Please see Testing in BDD with Gherkin based frameworks (e.g. Cucumber) for an overview of the possible workflows.

Using Jira and Xray as master

This section assumes using Xray as master, i.e. the place that you'll be using to edit the specifications (e.g. the scenarios that are part of .feature files).

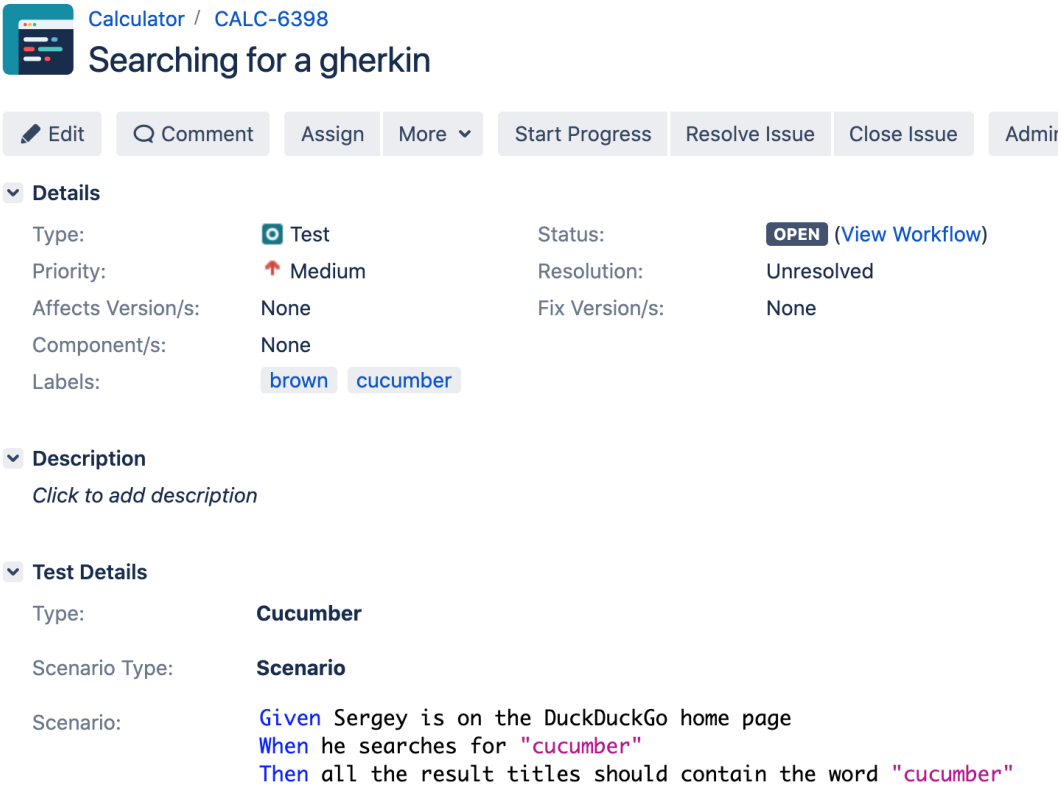
The first step is to create "Cucumber" Tests, of Cucumber Type "Scenario", in Jira.

Test can be created from the user story of using the standard Jira's issue create button/action.



The specification would be exactly the same as the one provided in one of the scenarios in the the original repository.

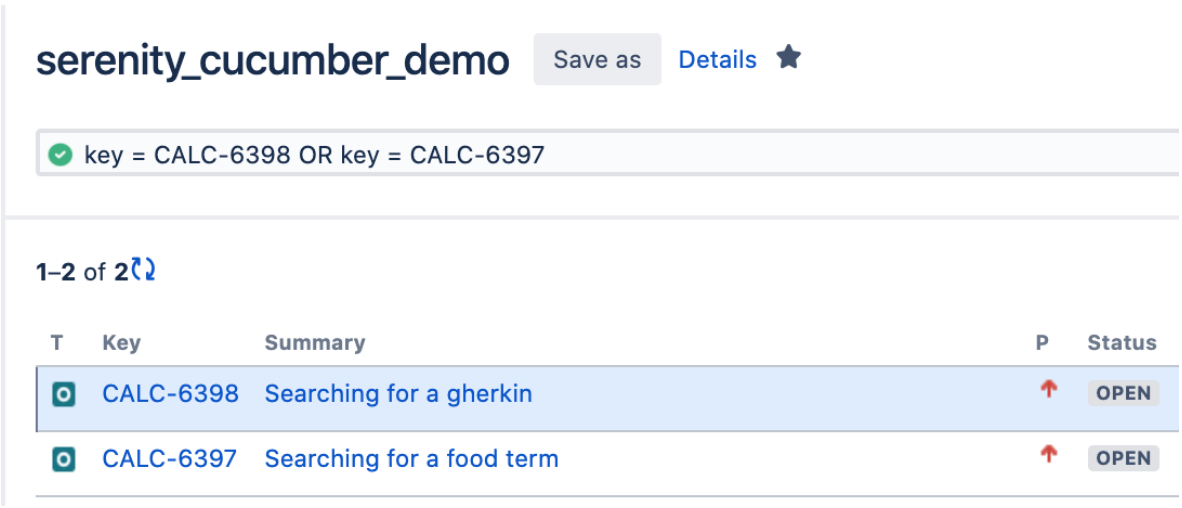
The test is quite self-explanatory, which is the ultimate purpose of using this approach: a browser is open on the "DuckDuckGo" home page, search by “cucumber” and then we check if all results contain the word "cucumber" in the title.



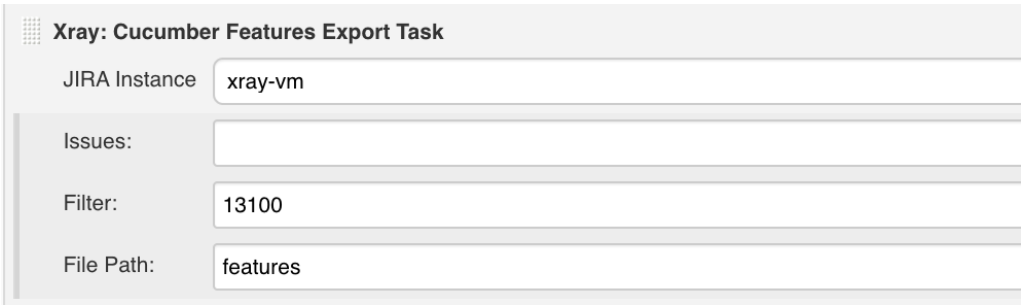
We would repeat this for every Scenario/Scenario we would like to specify.

Then, we need to export these executable scenarios as .feature file(s) in order to run them (locally or in the CI environment). This may be done via the REST API, or the **Export to Cucumber** UI action from within the Test/Test Execution issue or even based on an existing saved filter.

In this case, we are going to use a saved filter in Jira. The filter can contain Test issues, to user stories, Test Plans, Test Executions; Xray will always find out the related Test issues.



A plugin for your CI tool of choice (e.g. Jenkins) can be used to ease this task.



You could also do it from the command line.

example of exporting features from the command line

```
curl -u admin:admin  "http://jiraserver.example.com/rest/raven/1.0/export/test?filter=13100&fz=true" -o
features.zip
rm -rf features/*
unzip -o features.zip  -d features
```

We will store the exported .feature(s) in a temporary folder (e.g. `features/`), that we need to clean before the export process.

After being exported, the created .feature file will be similar to the original one but will contain the references to the Test issue key and the covered requirement issue key.

```
@REQ_CALC-6399
Feature: As a user, I can search by keywords using DuckDuckGo
        #As a user, I can search by keywords using DuckDuckGo


        @TEST_CALC-6398 @brown @cucumber @src/test/resources/features/search/search_by_keyword.feature
        Scenario: Searching for a gherkin
                Given Sergey is on the DuckDuckGo home page
                When he searches for "cucumber"
                Then all the result titles should contain the word "cucumber"


        @TEST_CALC-6397 @cucumber @green @src/test/resources/features/search/search_by_keyword.feature
        Scenario: Searching for a food term
                Given Sergey is on the DuckDuckGo home page
                When he searches for "cucumber food"
                Then all the result titles should contain the word "recipes"
```

Tests can be run using Maven; we need to tell the runner to pick the .feature files from the "features/" folder using the "cucumber.features" system property.

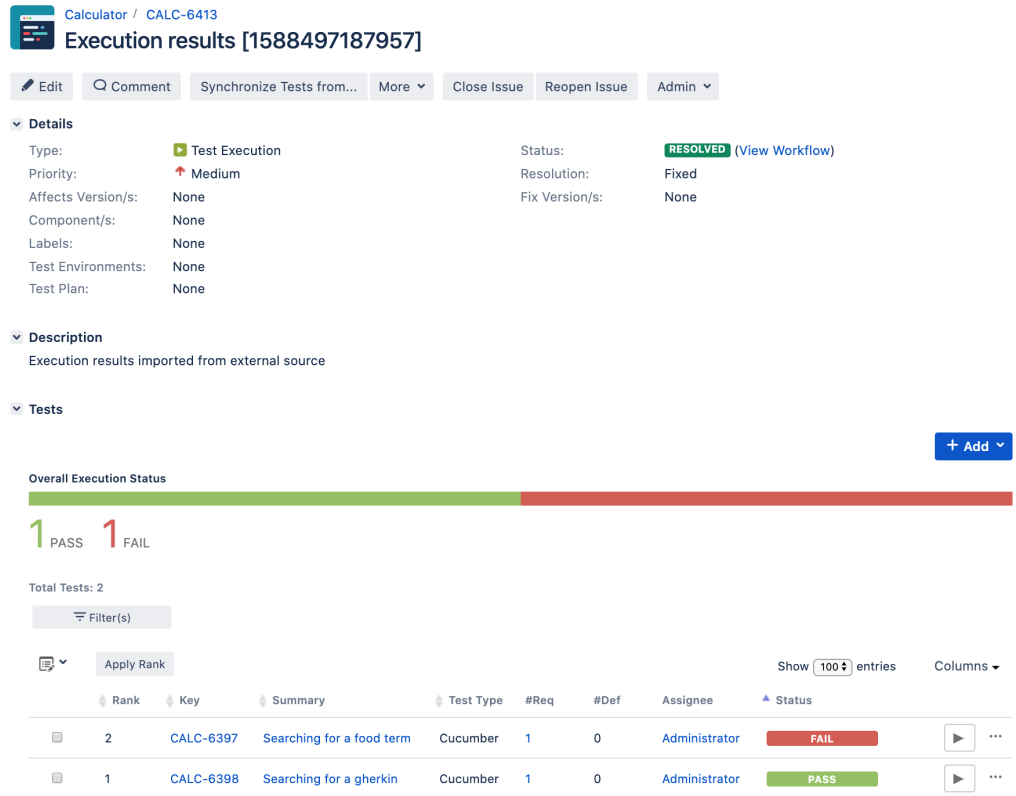
```
rm -r target/serenity-reports/*
mvn clean verify -Denvironment=staging -Dcucumber.features="features/"
```

After running the tests and generating the Cucumber JSON report (e.g., cucumber_report.json), it can be imported to Xray via the REST API or the **Import Execution Results** action within the Test Execution or by using one of available plugins for CI tools.

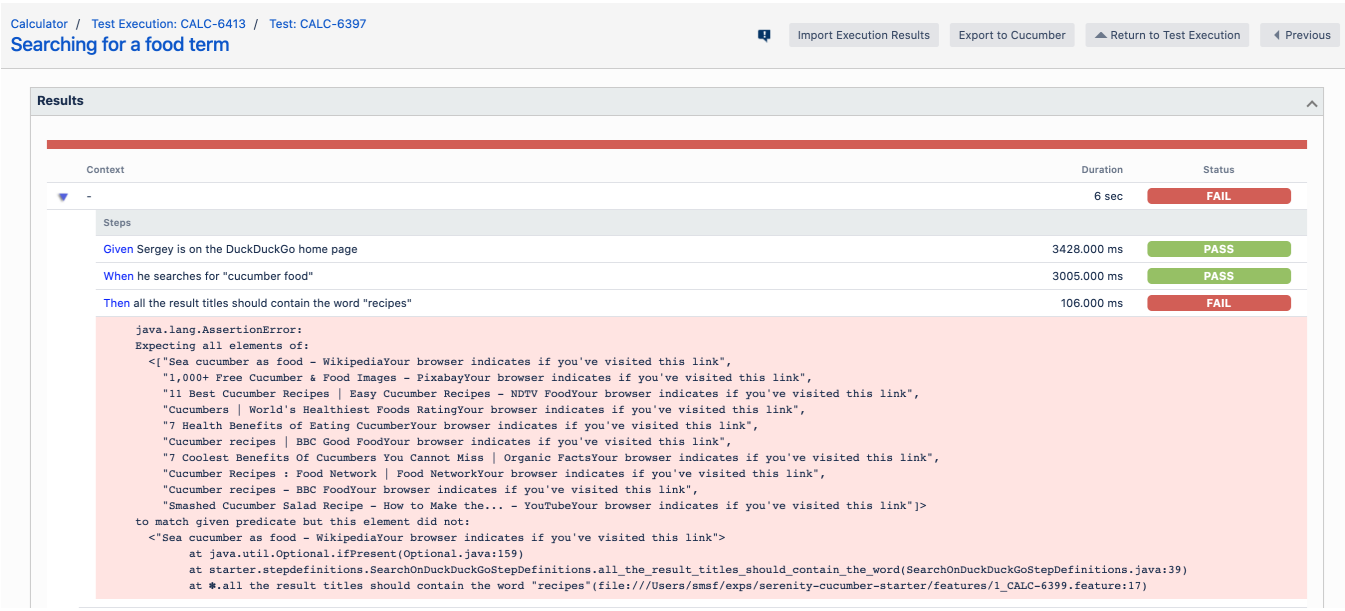
example of importing results from the command line

```
curl -H "Content-Type: application/json" -X POST -u admin:admin --data @"target/serenity-reports
/cucumber_report.json" http://jiraserver.example.com/rest/raven/1.0/import/execution/cucumber
```

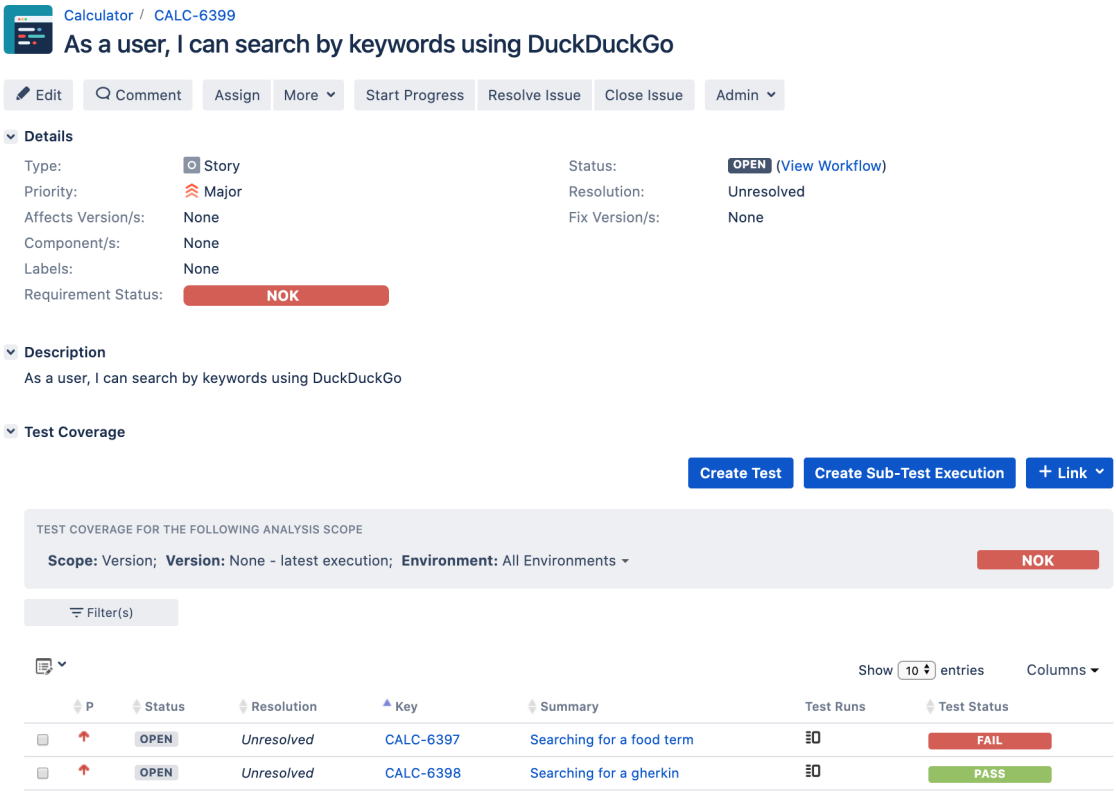
A Test Execution containing the results for each test scenario will be created.



The execution screen details will provide information on the test run result that includes step-level information including duration; in this case we can only see the Gherkin-level keywords.



On the “requirement”/user story side (i.e the “feature”) we can also see how this result impacting on the coverage.



Using Git or other VCS as master

You can edit your .feature outside of Jira/Xray (eventually storing them in your VCS using Git, for example).

In our example, the feature file can be found at `src/test/resources/features/search/search_by_keyword.feature`.

src/test/resources/features/search/search_by_keyword.feature

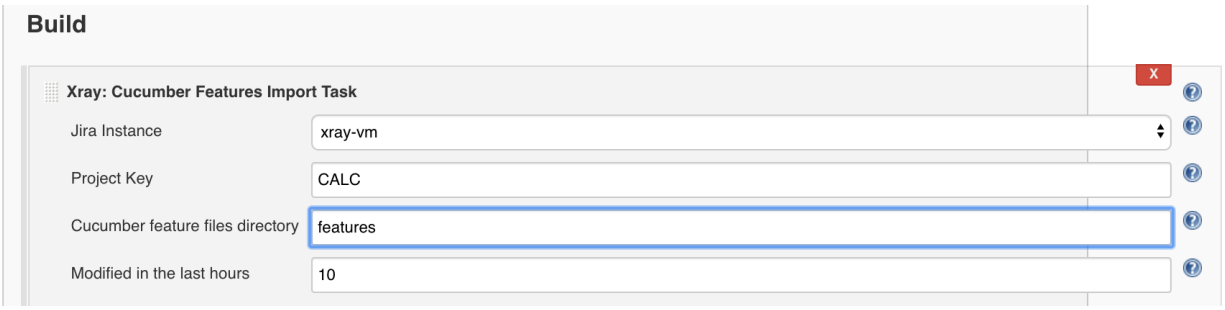
```
@REQ_CALC-6399
Feature: Search by keyword

  @cucumber @green
  Scenario: Searching for a food term
    Given Sergey is on the DuckDuckGo home page
    When he searches for "cucumber food"
    Then all the result titles should contain the word "recipes"

  @cucumber @brown
  Scenario: Searching for a gherkin
    Given Sergey is on the DuckDuckGo home page
    When he searches for "cucumber"
    Then all the result titles should contain the word "cucumber"
```

Note: we can link the tests/scenarios to an existing user story/requirement in Jira/Xray by adding a tag before the "Feature" element.

In any case, you'll need to synchronize your .feature files to Jira/Xray so that you can have visibility of them and report results against them.

Thus, you need to import your .feature files to Xray/Jira which will create (or update) Test and Pre-Condition entities in Xray. The process is idem-potent.

You can invoke the REST API directly, or use one of the available plugins for well-known CI tools (e.g. Jenkins), and choose the destination project.



Sample shell script

```
rm features.zip
zip -r features.zip src/test/resources/features/ -i \*.feature
curl -H "Content-Type: multipart/form-data" -u admin:admin -F "file=@features.zip" "http://jiraserver.example.com/rest/raven/1.0/import/feature?projectKey=CALC"
```

The tests will be created (or updated if they already exist); we may notice that there's a specific label being added to keep track of the original .feature where the scenario came from.

Calculator / CALC-6397

Searching for a food term

Edit | Comment | Assign | More | Start Progress | Resolve Issue | Close Issue | Admin

Details

Type:	Test	Status:	OPEN (View Workflow)
Priority:	Medium	Resolution:	Unresolved
Affects Version/s:	None	Fix Version/s:	None
Component/s:	None		
Labels:	cucumber green src/test/resources/features/search/search_by_keyword.feature		

Description

Click to add description

Test Details

Type:	**Cucumber**
Scenario Type:	**Scenario**
Scenario:	Given Sergey is on the DuckDuckGo home page When he searches for "cucumber food" Then all the result titles should contain the word "recipes"

Calculator / CALC-6398

Searching for a gherkin

Edit | Comment | Assign | More | Start Progress | Resolve Issue | Close Issue | Admin

Details

Type:	Test	Status:	OPEN (View Workflow)
Priority:	Medium	Resolution:	Unresolved
Affects Version/s:	None	Fix Version/s:	None
Component/s:	None		
Labels:	brown cucumber src/test/resources/features/search/search_by_keyword.feature		

Description

Click to add description

Test Details

Type:	**Cucumber**
Scenario Type:	**Scenario**
Scenario:	Given Sergey is on the DuckDuckGo home page When he searches for "cucumber" Then all the result titles should contain the word "cucumber"

Please note

In simple terms, each Scenario of each .feature will be created as a Test issue that contains unique identifiers, so that if you import once again then Xray can update the existent Test and don't create any duplicated tests; each Background will be created as a Pre-Condition.

More info in Importing Cucumber Tests - REST.

Afterward, you can export those features out of Jira based on some criteria, so they are properly tagged.

As an example, we can export the tests based on the covered issue; you could use also a saved Jira filter using its filter id.

Below you can see an example using Xray Jenkins plugin.

You could also do it from the command line.

example of exporting features from the command line

```
curl -u admin:admin  "http://jiraserver.example.com/rest/raven/1.0/export/test?keys=CALC-6399&fz=true" -o
features.zip
rm -rf features/*
unzip -o features.zip  -d features
```

This will produce a .feature file with the Scenario(s)/Scenario Outline(s) tagged with the respective Test issue keys.

features/1_CALC-6399.feature

```
@REQ_CALC-6399
Feature: As a user, I can search by keywords using DuckDuckGo
        #As a user, I can search by keywords using DuckDuckGo


        @TEST_CALC-6398 @brown @cucumber @src/test/resources/features/search/search_by_keyword.feature
        Scenario: Searching for a gherkin
                Given Sergey is on the DuckDuckGo home page
                When he searches for "cucumber"
                Then all the result titles should contain the word "cucumber"


        @TEST_CALC-6397 @cucumber @green @src/test/resources/features/search/search_by_keyword.feature
        Scenario: Searching for a food term
                Given Sergey is on the DuckDuckGo home page
                When he searches for "cucumber food"
                Then all the result titles should contain the word "recipes"
```

Tests can be run using Maven; we need to tell the runner to pick the .feature files from the "features/" folder using the "cucumber.features" system property.

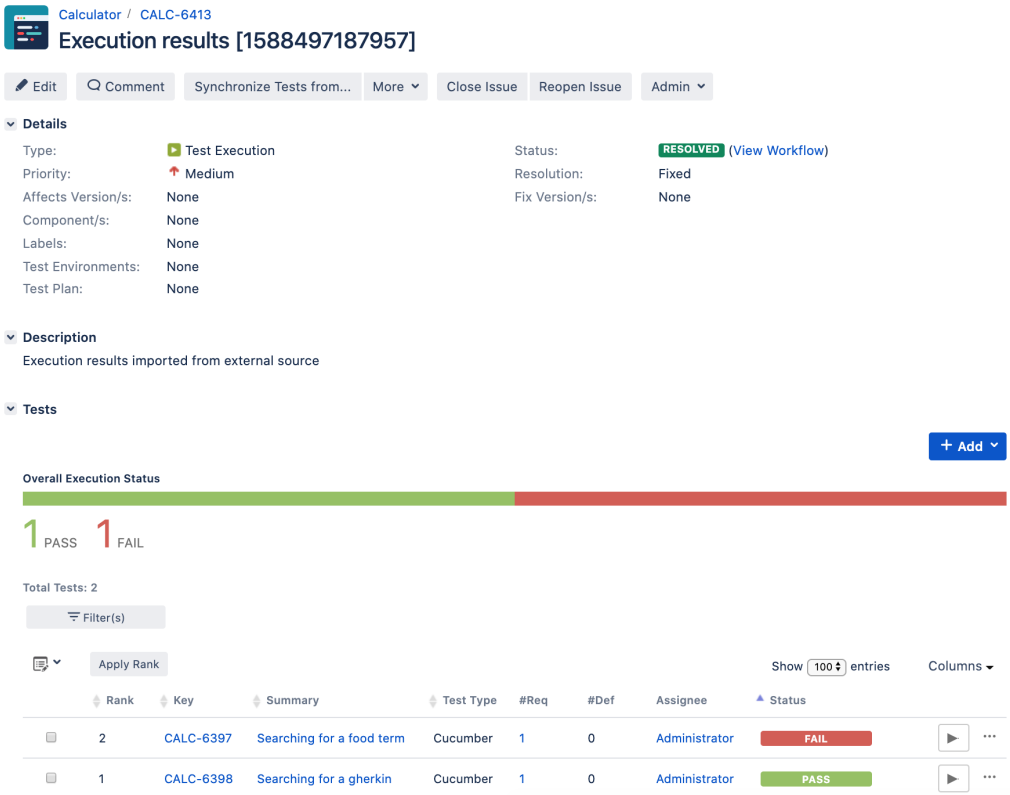
```
rm -r target/serenity-reports/*
mvn clean verify -Denvironment=staging -Dcucumber.features="features/"
```

After running the tests and generating the Cucumber JSON report (e.g., cucumber_report.json), it can be imported to Xray via the REST API or the **Import Execution Results** action within the Test Execution or by using one of available plugins for CI tools.

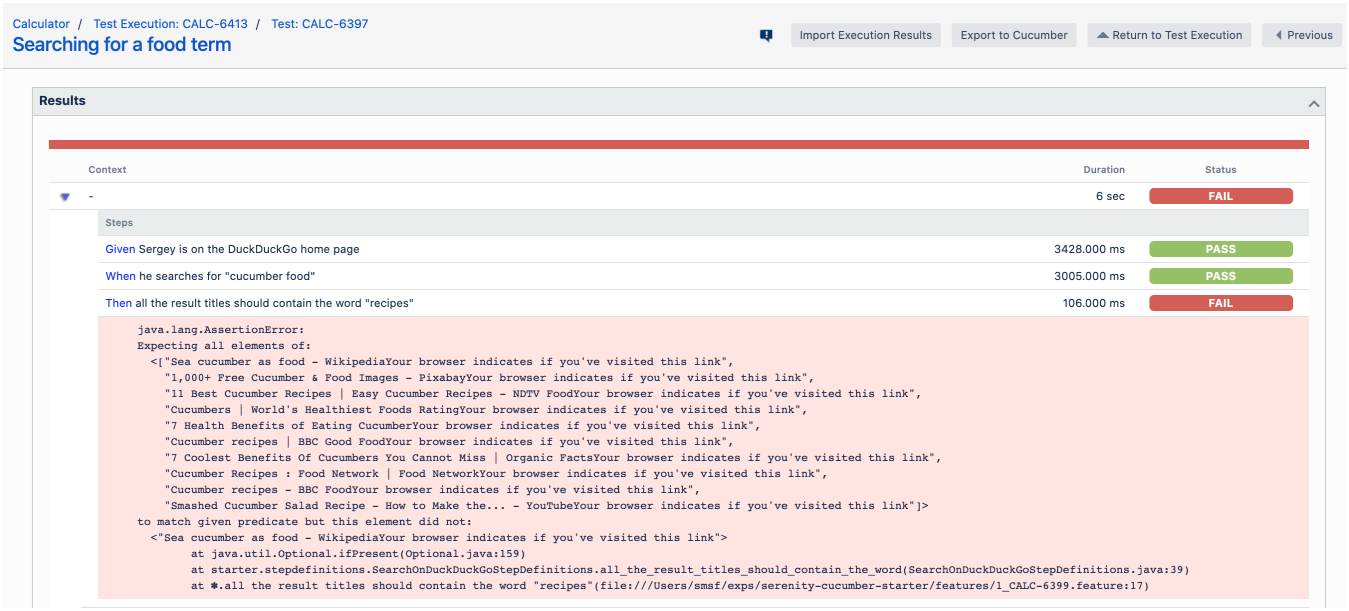
example of importing results from the command line

```
curl -H "Content-Type: application/json" -X POST -u admin:admin --data @"target/serenity-reports
/cucumber_report.json" http://jiraserver.example.com/rest/raven/1.0/import/execution/cucumber
```

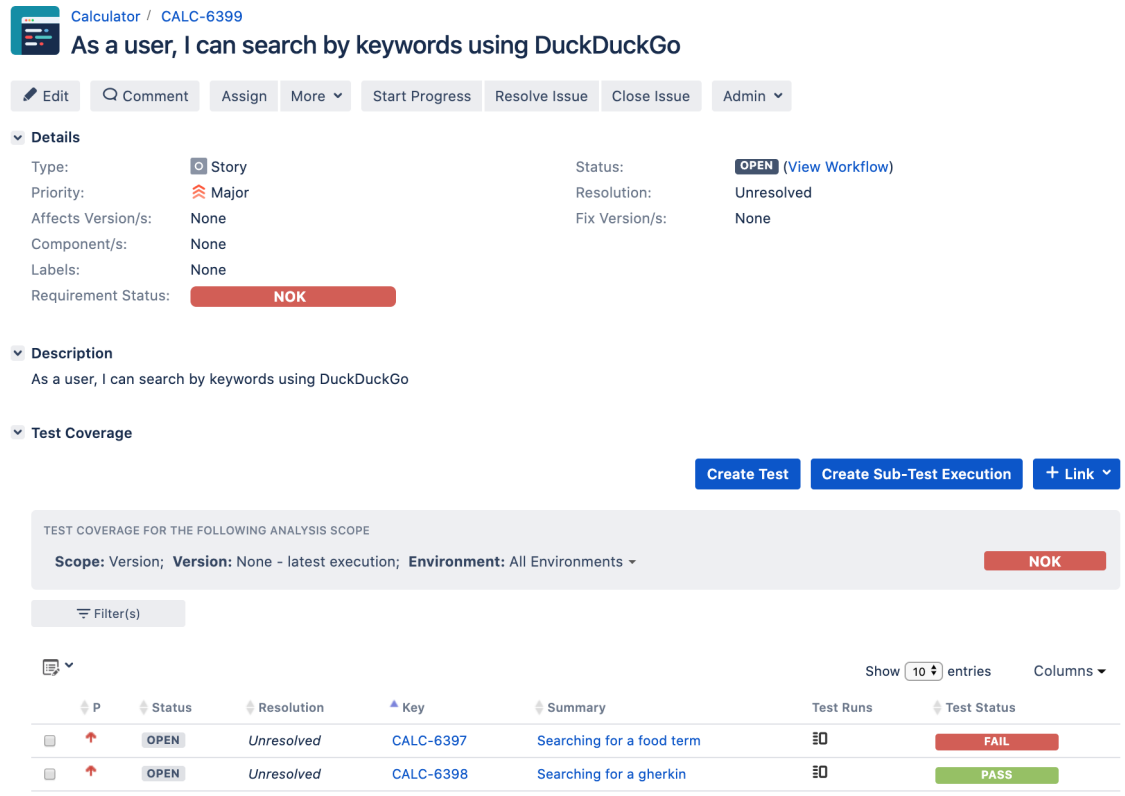
A Test Execution containing the results for each test scenario will be created.



The execution screen details will provide information on the test run result that includes step-level information including duration; in this case we can only see the Gherkin-level keywords.



On the “requirement”/user story side (i.e the “feature”) we can also see how this result impacting on the coverage.



If we change the specification (i.e. the Gherkin scenarios), we need to import the .feature(s) once again.

Therefore, in the CI we always need to start by importing the .feature file(s) to keep Jira/Xray on synch.

References

- Serenity BDD (formerly Thucydides)
- https://johnfergusonsmart.com/serenity-bdd/
- Sample project using Serenity and Cucumber
- Step libraries article
- An Introduction to BDD Test Automation with Serenity and Cucumber-JVM