Testing using Selenium WebDriver and Gauge in Java

Overview

In this tutorial, we will "create" some acceptance tests using Gauge framework and import back the results to Xray.

We'll reuse a sample Java maven-based project having tests that make use of Selenium to drive a browser in a sample shop application.

In Gauge, specifications are used to describe acceptance tests for business cases.

> *A specification is a business test case which describes a particular feature of the application that needs testing. (Gauge team)*

Specification of test scenarios is separate from their actual implementation. While the specification has a structure and proper syntax, the implementation of steps can be done in one of the supported languages (e.g. Java, Ruby, C#, Javascript, Python).

Specifications may be done using steps or concepts (steps may be grouped together as a concept to define a unit of "business intent").

The idea is to have readable test scenarios and promote the reusability of steps/concepts.

Specifications typically live under a `specs/` folder.

A clear Gauge overview may be found in Gauge's documentation site.

Requirements

- Inside the project folder, install Gauge's java plugin

```
gauge install java
```

- Inside the project folder, install the `xml-report` plugin

```
gauge install xml-report
```

- A sample web admin shop must be installed; you can follow the instructions provided in Gauge's sample project to download the .war file. After being downloaded, it may be easily started (it takes a few seconds to do so)

```
java -jar activeadmin-demo.war
```

Description

For this tutorial, we'll use a sample Java maven-based project provided by the Gauge team.

The project contains several tests, as part of distinct specifications.

Next, you may see an example of the `PlaceOrder.spec` file, which contains two scenarios (i.e. tests): "Buy a book" and "Cart retains items until order is placed".

specs/PlaceOrder.spec

```
Place an Order
==============

* Go to active admin store

Buy a book
----------
tags: customer

* Log in with customer name "ScroogeMcduck" and "password"
* Place order for "Beginning Ruby: From Novice to Professional". The cart should now contain "1" items
* Log out

Cart retains items until order is placed
----------------------------------------
tags: customer

* Log in with customer name "ScroogeMcduck" and "password"
* Add "Beginning Ruby: From Novice to Professional" and the cart will now contain "1" item(s)
* Log out

* Log in with customer name "ScroogeMcduck" and "password"
* Add "The Well-Grounded Rubyist" and the cart will now contain "2" item(s)
* Log out
```

The steps implementation is done in different classes (you may freely structure these classes).

The following code shows the associated implementation for the previous specification.

Two things should be mentioned though:

1. some steps (e.g. for login/logout) are implemented in other classes
2. as you can see, the previous specification does not use these "raw" steps as such; instead, it uses concepts detailed in a specific file (PlaceOrder.cpt) as shown ahead

/src/test/java/PlaceOrder.java

```
import com.thoughtworks.gauge.Step;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import utils.driver.Driver;

import java.util.List;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

public class PlaceOrder {

    @Step("Add item <item> to the cart.")
    public void addItemToTheBasket(String item) {
        WebDriver webDriver = Driver.webDriver;
        webDriver.findElement(By.linkText(item)).click();
        webDriver.findElement(By.linkText("Add to Card")).click();
    }

    @Step("Checkout now")
    public void placeTheOrder() {
        WebDriver webDriver = Driver.webDriver;
        webDriver.findElement(By.xpath("//input[@value='Checkout Now!']")).click();
    }

    @Step("Cart now contains <itemCount> number of items")
    public void cartNowContains(int numberOfItems) {
        WebDriver webDriver = Driver.webDriver;
        List<WebElement> products = webDriver.findElements(By.xpath("//table/tbody/tr"));
        assertEquals(numberOfItems,products.size()-2);
    }
}
```

specs/concepts/PlaceOrder.cpt

```
Created by sswaroop on 5/15/17

This is a concept file with following syntax for each concept.
# Checkout
* Checkout now
* Show a message "Thank you for your purchase! We will ship it shortly!"

# Add <item> and the cart will now contain <itemCount> item(s)
* See items available for purchase.
* Add item <item> to the cart.
* Cart now contains <itemCount> number of items

# Place order for <item>. The cart should now contain <itemcount> items
* See items available for purchase.
* Add <item> and the cart will now contain <itemcount> item(s)
* Checkout
```

Since we're using a maven-based project, tests can be run as usual (i.e. "mvn test"); otherwise, you could use the "gauge run ..." command.

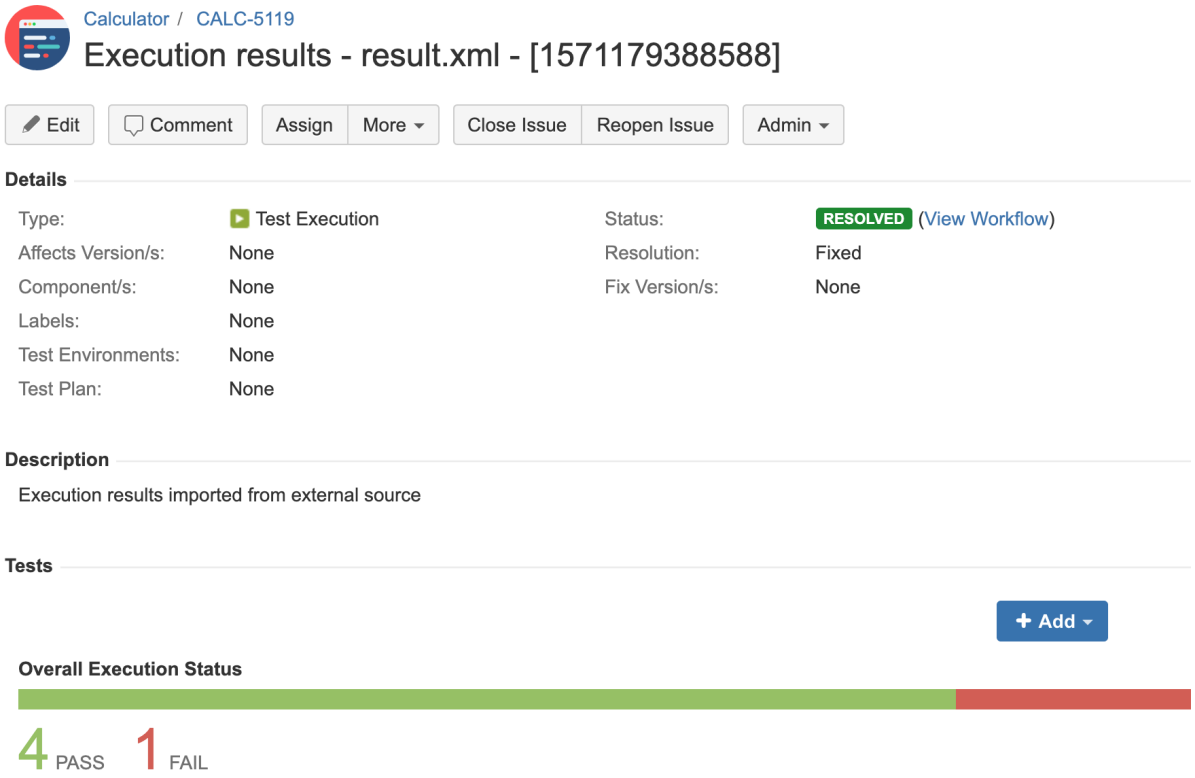
Note that we won't use Surefire to generate the JUnit XML report but we'll use Gauge's "xml-report" plugin instead.

```
mvn test
```

After running the tests and generating the JUnit XML report (e.g., result.xml), it can be imported to Xray (either by the REST API or through **Import Execution Results** action within the Test Execution).

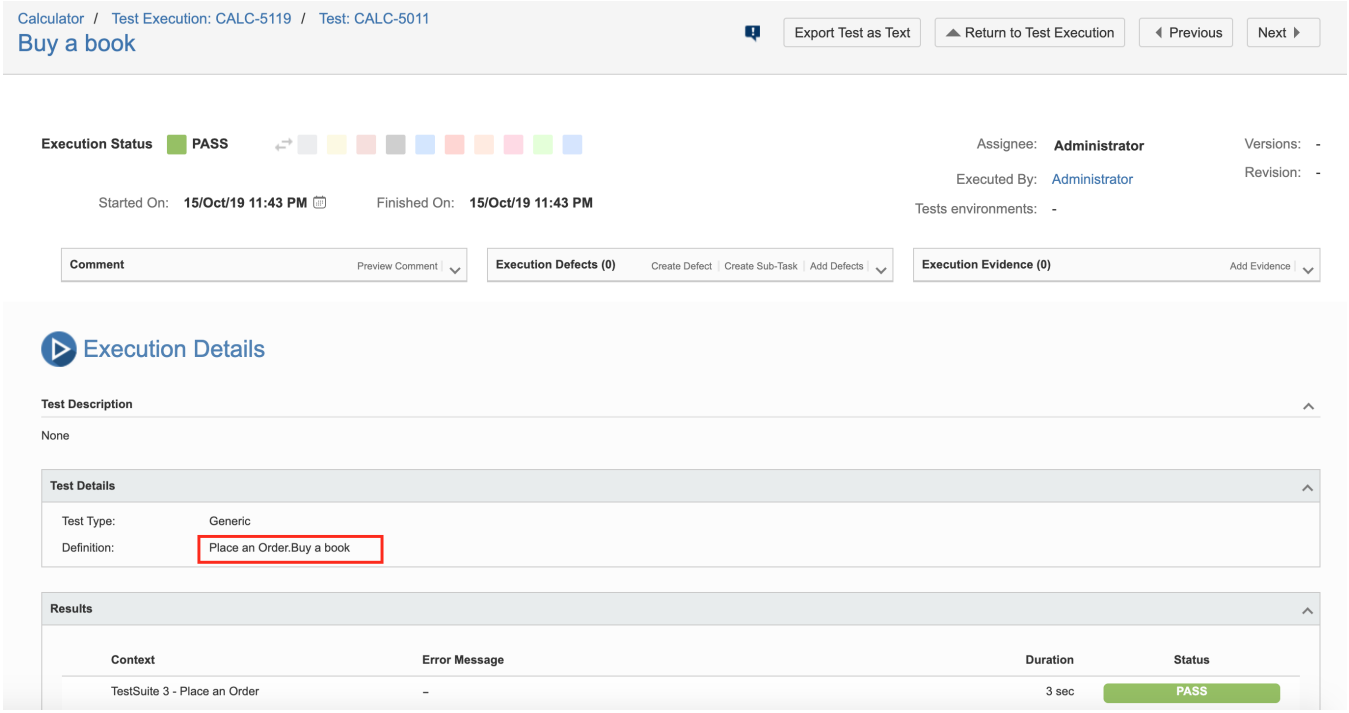
```
curl -H "Content-Type: multipart/form-data" -u admin:admin -F "file=@reports/xml-report/result.xml"
http://jiraserver.example.com/rest/raven/1.0/import/execution/junit?projectKey=CALC
```

A Test Execution will be created containing information about the executed scenarios.



Each scenario is mapped to a Generic Test in Jira, and the **Generic Test Definition** field contains the name of the specification concatenated with the scenario name.

The Execution Details of the Generic Test contains information about the "Test Suite" (as per JUnit format), which in this case corresponds to the scenario name with a prefix.



Notes

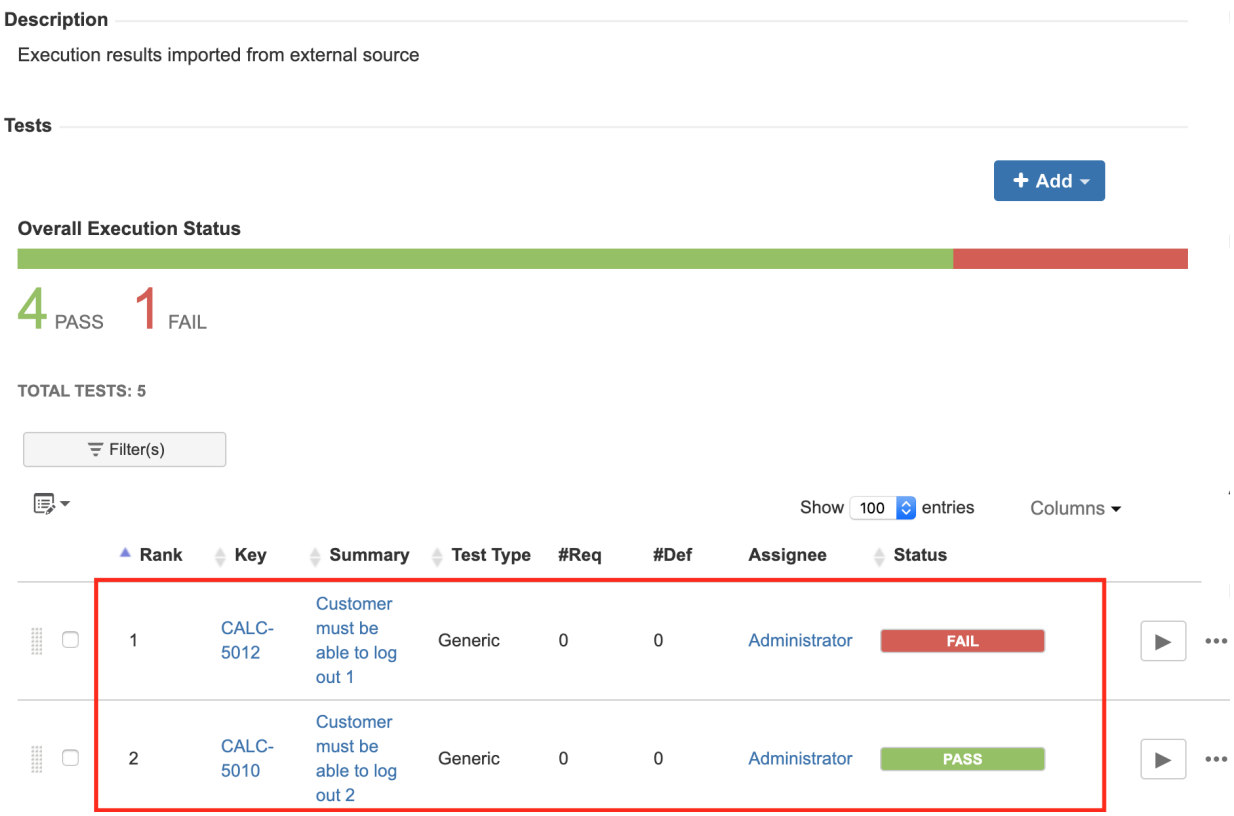
Sometimes you may use data tables in your specifications.

specs/CustomerLogout.spec

```
Customer Log out
================

     |name|email               |password|
     |----|--------------------|--------|
     |John|john.doe@example.com|password|
     |Jane|jane.doe@example.com|password|

* Go to active admin store

Customer must be able to log out
--------------------------------
* Sign up as <name> with email <email> and <password>
* Log out
```

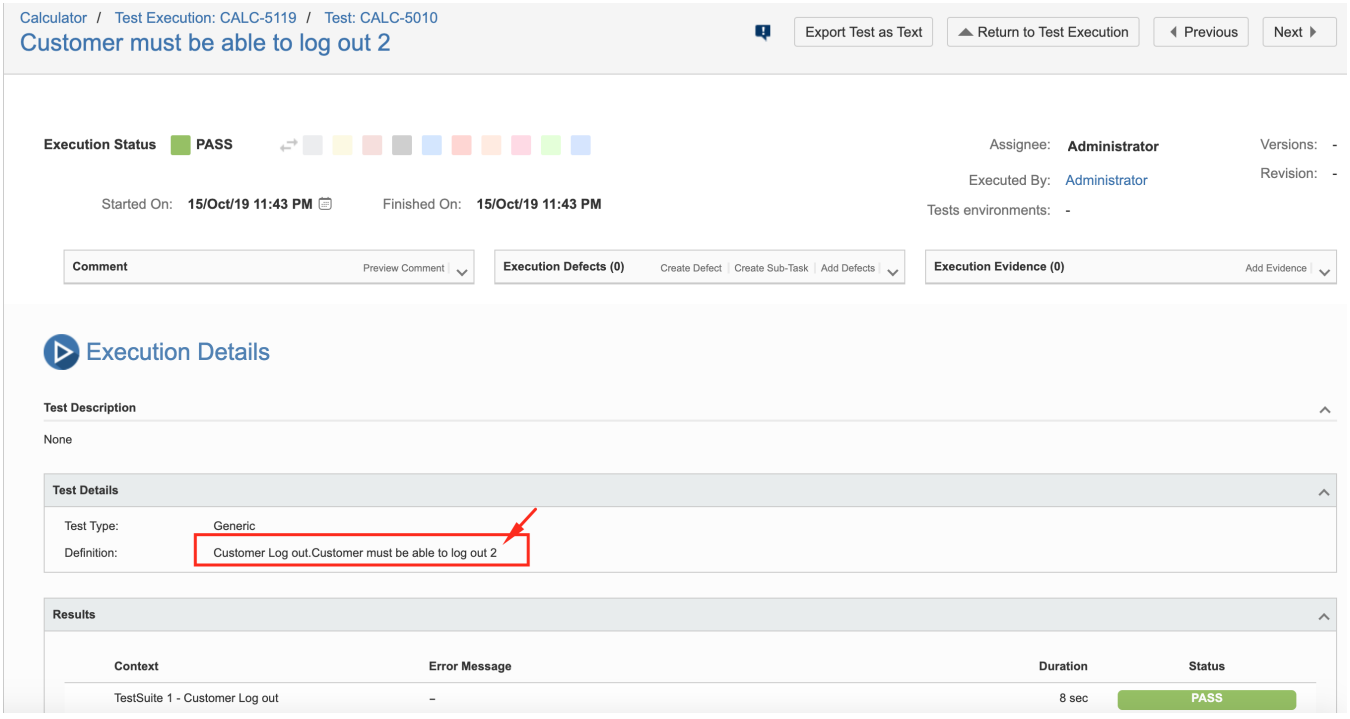
In that case, the scenario will be executed multiple times (once per row) leading to multiple results in the JUnit XML report produced by the "xml-report" plugin.

This, in turn, will lead to multiple Test issues being created.



The unique identifier for the Test will be weak in the sense that it is composed of the specification plus the scenario name followed by a sequential number (i.e. row number).

In other words, if at some point in time you change the table rows order, results can be reported against the wrong existing Test issue. This may be avoided if you manually update the Generic Test Definition field with the proper row index before submitting results. A better way of identifying these scenarios could be perhaps achieved and agreed with the xml-report plugin team.



References

- https://gauge.org/
- https://docs.gauge.org/index.html?os=macos&language=java&ide=vscode
- https://github.com/getgauge/xml-report
- https://github.com/getgauge-examples/java-maven-selenium