Part 1 - Hands-On Test Automation Project with Java + Cucumber + Selenium + Spring + Docker

Soraia Reis Fernandes
Senior Software Engineer @ Avenue Code
Published Aug 5, 2020

+ Follow

Success in test automation requires careful design work. Before starting an automation project it is necessary to invest time creating a structure that can support test creation. A well-structured project can bring significant benefits: ease of scripting, scalability, modularity, understandability, process definition, re-usability, maintenance, etc. In this article, we are gonna go over some steps to structure an automation project using Java, Cucumber, Selenium, Spring, and Docker.

What are the requirements?

differences, so keep this in mind.

- **Java Development Kit (JDK):** The JDK is a software development environment used for developing Java applications. It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), and other tools needed in Java development. The version I used while writing this article was **Java 8**. Install Java JDK 8 from: https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html.

- **Gradle:** Gradle is a build automation tool for multi-language software development. It controls the development process in the tasks of compilation and packaging to testing, deployment, and publishing. The Gradle version used while writing this article was **6.4.1**. Install Gradle from: https://gradle.org/install/.

- **Source-code editor for Java:** I use IntelliJ, a source-code editor developed by JetBrains. You can get IntelliJ from: https://www.jetbrains.com/idea/download/. There are some plugins that are required to be enabled on IntelliJ while following this article: Cucumber Java, Gherkin, and Lombok.

- **Firefox:** Firefox because it comes pre-bundled in the Selenium package. Its driver is included in the *selenium-server-stanalone.jar*, so I preferred to use it in the article. The Firefox version used was **79.0**. Install Firefox from: https://www.mozilla.org/.

- **Docker:** Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. The Docker version used while writing this article was **19.03.8**, which is part of the *Docker Desktop* community version **2.3.0.3**. Install Docker from: https://www.docker.com/get-started.

To check or confirm if you have Java JDK installed, run this command in your terminal:

To check or confirm if you have Gradle installed, run this command in your terminal:

```
gradle -v
```

To check or confirm if you have Docker installed, run this command in your terminal:

```
docker -v
```

Who is this article for?

This article does not intend to go deeper in Selenium nor on Cucumber, Spring, or Docker. It will be focused on how to structure a test automation project using them. However, you are not required to know them before following this article. Even never working with any of them before you should be able to follow and practice this article content. But if you are interested in deep learning them I will suggest a few links while you read the article.

How is this article divided?

This article has 5 parts that will be published separated. This page contains Part 1 only.

- **Part 1 - Basic Structure:** shows how to set up the basic structure of a Java project with Gradle, TestNG, Cucumber, and Selenium.

- **Part 3 - Configurations:** shows more details on ways of running the tests and doing some configurations related to it.
- **Part 4 - Reporting, Screenshots and Logging:** shows how to add custom reporting, embed screenshots to the report, and add logs with customized outputs.
- **Part 5 - Parallel Testing and Docker:** shows how to run the tests in parallel, in a Docker container, and different browsers.

Are you ready? :)

Part 1 - Basic Structure with Selenium and Cucumber

I have many years of experience in automation with Java. I have done different projects using this language for different purposes. I have created architectures in many different ways as well, because each application requires a different set of tools and libraries to automate. In this article, I want to share a high-level view on points I like to cover in test automation architectures.

Now that I gave you an introduction to my experience with Java automation projects, we will start creating a basic structure. Once you have the required setup (as mentioned on *What are the requirements?*) you can continue the practice presented in this article.

The first step is to create the basic structure. I highly recommend to use a build automation system in any project you create. It will help mainly with **dependency management** and **test execution**, which makes life easy. In this article, we are going to use **Gradle**. Once you checked that you have Java and Gradle installed on your machine you can follow the steps below:

```
mkdir demo-spring-selenium && cd demo-spring-selenium
```

- Start a new Gradle Java project executing the command below and select all default options (press RETURN until the end):

```
gradle init --type java-library
```

- Open the project on the IDE (for example, IntelliJ) and you will be ready to go.

When you use **gradle init** command it creates the project with a sample class (Library.java) and unit test class (LibraryTest.java). Those files are not necessary in our project (just samples), so delete them. Beside those files, you may know or have noticed that Java projects are structured inside a **src** directory. The **src** directory contains all of the source material for building the project. It contains a subdirectory for each type: **main** for the main build artifact and **test** for the unit test code and resources. Since our automation project is going to be used only for test automation purposes we just need the **test** folder on our project, and then delete the **main** directory.

The project structure should look like this:

No alt text provided for this image

After creating our project structure we should define our **version control system** and where we are going to **host** the project. We are going to use **Git** and save our project on a **GitHub** repository. But before, we need to understand that there are files we do not want to check into our repository. Some people think they need to and/or are not necessary. It is best practice not to commit unnecessary files. To do so we can add a list of unwanted files and folders to the **.gitignore** file, which is a simple file that lists what Git should ignore making undesirable files invisible.

- On the root of our project we should already have the file **.gitignore** (created by gradle init command) or create one if it is not there;
- On the **.gitignore** file add the list of files and folders we want to ignore (see example below).

```
.gradle
.idea
*.iml
build
out
.DS_Store
```

The **.gradle** folder contains Gradle settings, but is not part of the Java project. Whenever someone clones and opens the project locally this folder will be recreated. The **.idea** folder and the ***.iml** files are specific to IntelliJ IDE. Files related to IDEs should not be pushed to a git repository. Although they do not appear once you create the project, you should also exclude the folders **build** and **out**. Both folders contain the output of your project when you build/compile. Note that **out** folder is an IntelliJ IDE specific folder. And if you are a Mac user like me you will need to add **.DS_Store**, which is a file that stores custom attributes of its containing folder, such as the position of icons or the choice of a background image.

Use the following link to add your new project on your GitHub account.

Adding an existing project to GitHub using the command line example, **JUnit** or **TestNG**. As a testing framework, we can use **Cucumber**, **JBehave**, **Behat**, which are BDD test frameworks. And as an automation library, we can use **Selenium** for web application automation, or **REST Assured** for API.

In this demo project, we will be using **TestNG**, **Cucumber,** and **Selenium**. Follow the steps below to configure the project with those libraries:

- On the root of our project, we can see the file **build.gradle**.
- Open this file and clean all the unnecessary comments and dependencies.
- Add the IDE plugins to integrate our gradle java project to the IDE. For example, when using IntelliJ IDE we need to add: id 'idea'.
- Add as a dependency the following libraries:

```
cucumber-java
cucumber-testng
selenium-java
selenium-api
```

The build.gradle file should look like this:

```
plugins {
    id 'java'
    id 'idea'
}


repositories {
    jcenter()
}


ext {
    cucumber = '6.4.0'
    selenium = '3.141.59'
}
```

```
        testImplementation "org.seleniumhq.selenium:selenium-java:$selenium"
        testImplementation "org.seleniumhq.selenium:selenium-api:$selenium"
    }
```

For organization purposes, I like to keep the version of the libraries as a property, so they can be easily updated when a new version comes up. Gradle allows you to add properties using the "ext" block (as shown above).

We are going to use an example application built by Dave Haeffner which is perfect for practicing automated tests. I forked from his main project to guarantee the page does not change and the examples continue to work after time. The base URL for our samples will be: https://soraia.herokuapp.com.

We will create our first test for the Login: https://soraia.herokuapp.com/login. This feature consists of a simple user/password form as shown in the image below.

No alt text provided for this image

When we start any test automation project we need to consider applying a very well known design pattern called **Page Objects**. This pattern was developed by Martin Fowler and became popular among UI automation projects because it enhances test maintenance and reduces code duplication. The Page Object encapsulates the mechanics required to find and manipulate the interface elements.

The purpose of this pattern is to create an object for each page of the application and, using Object Orientation principle, encapsulate in each class the attributes and methods, such as fields and actions of each page. The test classes itself will then use the methods of the encapsulated classes.

No alt text provided for this image

methods called “typeUsername”, "typePassword" and "clickOnLogin", passing necessary information as parameters, and we use those methods in the test class.

If you want to read more about the pattern, you can start reading Martin Fowler's article: https://martinfowler.com/bliki/PageObject.html. In case you want a more deep understanding, there are also many other articles you can search for that explain the Page Objects pattern.

We will use Page Objects and Cucumber in our project so we need to define the organization of packages and folders inside the project. Our project will have mainly three layers:

1. Page Objects;
2. Step Definitions; and
3. Feature Files

For the first two we will create separate packages under our main package in the **src** folder, and for the feature files, we will store them in the **resources** folder.

- Create the following packages under **demo.spring.selenium** package: **pages** and **stepdefinitions**.
- Create a folder called **features** under the **resources** folder.

To support the PageObject pattern, WebDriver's support library contains a factory class. The **PageFactory** Class in Selenium is an extension of the Page Object design pattern. It is a way to initialize the web elements of the Page Object when you instantiate it. Using the PageFactory class we use the annotation **@FindBy** to find an element or list of elements, and we use the static *initElements* method to

No alt text provided for this image

We will be using PageFactory in our project. We first need to create a class that will open the Login feature page. Inside the "pages" package create the **HomePage.java** class.

On this page, we only need to click on the link that will open the Login form page. Inside this class, we are going to create a WebElement variable for the link (**formAuthentication**) and associate its HTML locator using the annotation @FindBy. Then create the method **clickFormAuthentication()** that will click on the "Form Authentication" link using the Selenium command **<webElement>.click()**.

All Page Object classes will have a constructor that will receive the WebDriver as a parameter and invoke the PageFactory *initElements* static method.

```
public class HomePage {

  @FindBy(linkText = "Form Authentication")
  WebElement formAuthentication;

  public HomePage(WebDriver driver) {
    PageFactory.initElements(driver, this);
  }


  public void clickFormAuthentication() {
    formAuthentication.click();
  }
}
```

Now let’s create a class that will have all the elements and action methods necessary to fill the Login form and click on the Login button. Inside the "pages" package create the **LoginPage.java** class.

variable for each web element of the page (**usernameInput**, **passwordInput**, and **loginButton**) and associate its HTML locator using the annotation @FindBy.

Then create the methods **typeUsername(String username)** and **typePassword(String password)** that will type the values on username and password fields using Selenium command **<webElement>.sendKeys(<text>)**.

Finally, create the method **clickLogin()** that will click on the Login button using the Selenium command **<webElement>.click()**. And remember that we also need a constructor that will invoke the PageFactory *initElements* static method.

Our class should look like this:

```
public class LoginPage {

  @FindBy(id = "username")
  WebElement usernameInput;


  @FindBy(id = "password")
  WebElement passwordInput;


  @FindBy(className = "radius")
  WebElement loginButton;


  public LoginPage(WebDriver driver) {
    PageFactory.initElements(driver, this);
  }


  public void typeUsername(String username) {
    this.usernameInput.sendKeys(username);
  }


  public void typePassword(String password) {
    this.passwordInput.sendKeys(password);
  }


  public void clickLogin() {
    loginButton.click();
  }
}
```

simply use TestNG or JUnit to create test scenarios. In the project we are building in this article I want to show an approach using Cucumber, however, I want to mention that in the case you decide for an approach without BDD, I still recommend you create a similar layered architecture that between Page Objects and your test scenarios you have a layer that will encapsulate execution steps somehow.

No alt text provided for this image

In summary, it does not matter the tools you use for test automation, an automation project needs to have an appropriate abstraction layer. An abstraction layer is a way of hiding the working details, allowing the separation of concerns.

For our test project, we are going to use Cucumber as our BDD testing framework. We need to create our test scenario by writing it in **Gherkin** language. **Gherkin** is a simple text language designed to be easy to learn by non-programmers, but structured enough to allow for a concise description of examples to illustrate business rules. It allows us to describe the behavior of the software without detailing how this behavior is implemented.

The Login test scenario in Gherkin language would be like (ignore the tags @login and @smoke which will be explained later) the following. Create the file **login.feature** inside the "features" folder with:

```
@login
Feature: Login


  @smoke
  Scenario: Login successfully
    Given I open Login Page
    When I fill the username with "tomsmith"
    And I fill the password with "SuperSecretPassword!"
    And I click on Login
```

would like to introduce an important concept called **Hooks**. Hooks are blocks of code that can run at various points in the Cucumber execution cycle. They are typically used for setup and teardown before and after each scenario. They can be used to perform tasks that are not part of business functionality such as open a browser, connect to a database, etc. As you can see our Gherkin does not specify any step to open or close the browser, which is needed, respectively, before and after our scenario.

It is very simple! In our project, we will create a class called **Hooks.java** inside the package "stepdefinitions" like the following. It will hold a WebDriver static variable, so that all the step definition classes will be able to access the WebDriver, and the methods that will open and close the browser we annotate with @Before and @After respectively.

```
public class Hooks {

  public static WebDriver driver;

  @Before
  public void openBrowser() {
    driver = new FirefoxDriver();
    driver.get("https://soraia.herokuapp.com");
  }

  @After
  public void closeBrowser() {
    driver.quit();
  }
}
```

Note: Using a static variable for WebDriver is not recommended in an architecture, especially if we want to run parallel test scenarios. I am using just to simplify the explanation of a basic structure. Later in this same article, I will provide a solution for this when we will introduce the Dependency Injection concept.

that will open the Login feature page. The Cucumber annotations **@Given, @When, and @Then** are used to associate a method to a step. The HomeSteps.java class should look like the following:

```
public class HomeSteps {

  private HomePage homePage = new HomePage(Hooks.driver);

  @Given("^I open Login Page$")
  public void iOpenLoginPage() {
    homePage.clickFormAuthentication();
  }
}
```

Now create a **LoginSteps.java** class inside the "stepdefinitions" package. In this class, we are going to instantiate the LoginPage class to be able to execute the screen actions calling LoginPage methods. The LoginSteps.java class should look like the following:

```
public class LoginSteps {

  private LoginPage loginPage = new LoginPage(Hooks.driver);

  @When("^I fill the username with \"(.*)\"$")
  public void iFillTheUsernameInputWith(String username) {
    this.loginPage.typeUsername(username);
  }

  @When("^I fill the password with \"(.*)\"$")
  public void iFillThePasswordInputWith(String password) {
    this.loginPage.typePassword(password);
  }

  @When("^I click on Login$")
  public void iClickOnLoginButton() {
    this.loginPage.clickLogin();
  }
}
```

At this point, we should be able to execute our test scenario by right-clicking in the **login.feature** file, and choosing the option **Run Feature: login'**.

No alt text provided for this image

A test is not complete if it does not validate the result. As you can see, after login the page "secure" is opened with a successful message "You logged into a secure area!". To add a validation into our project we will create a **SecurePage.java** class inside the "pages" package following the same strategy mentioned for **LoginPage.java**.

Create a WebElement variable **messageDiv** and associate its HTML locator using the annotation @FindBy. Then create the method **getMessage()** that will get the message and return the text using Selenium command **<webElement>.getText()**.

```
public class SecurePage {

  @FindBy(id = "flash")
  WebElement messageDiv;

  public SecurePage(WebDriver driver) {
    PageFactory.initElements(driver, this);
  }

  public String getMessage() {
    return this.messageDiv.getText().split("\n")[0];
  }
}
```

Then we add a new step to our **Login successfully** scenario that will validate the message. Our feature file should look like the following now.

```
@login
Feature: Login

  @smoke
  Scenario: Login successfully
    Given I open Login Page
    When I fill the username with "tomsmith"
    And I fill the password with "SuperSecretPassword!"
    And I click on Login
    Then I see "You logged into a secure area!" message
```

SecureSteps.java class inside the "stepdefinitions" package. In this class, we are going to instantiate the SecurePage.java class to be able to call the **getMessage()** method.

To validate the message we can use the TestNG assertion library since we already have TestNG in our project. However, I would like to introduce **Hamcrest**. Hamcrest is a framework for writing matcher objects allowing 'match' rules to be defined declaratively. I like Hamcrest because it is more code readable and provides more detailed failure messages. *Are you curious about Hamcrest?* Check this link: http://hamcrest.org/JavaHamcrest/tutorial.

Add the following dependency and version variable to our **build.gradle** file. Inside the "ext" block add **hamcrest = '2.2'** and then inside "dependencies" add:

```
testImplementation "org.hamcrest:hamcrest-library:$hamcrest"
```

We will use the static method *assertThat* to validate if the message matches the expected value and our **SecureSteps.java** class should look like the following:

```
public class SecureSteps {

  private SecurePage securePage = new SecurePage(Hooks.driver);

  @Then("^I see \"(.*)\" message$")
  public void iSeeMessage(String message) {
    assertThat(this.securePage.getMessage(), is(message));
  }
}
```

Now we can run the test again.

Let’s add a new scenario that will reuse all the implementation we have done so far. This new scenario will test an invalid user login. Once you add it, execute the new test scenario and you should see it passing.

```
@login
Feature: Login

  @smoke
  Scenario: Login successfully
    Given I open Login Page
    When I fill the username with "tomsmith"
    And I fill the password with "SuperSecretPassword!"
    And I click on Login
    Then I see "You logged into a secure area!" message

  Scenario: Login with invalid user
    Given I open Login Page
    When I fill the username with "invalid"
    And I fill the password with "SuperSecretPassword!"
    And I click on Login
    Then I see "Your username is invalid!" message
```

Last but not least, we should create a readme file and document everything that will help users of your framework. On the root of our project create the file **README.md**. Here is a sample of how I would create it.

```
# Demo Spring Selenium Automation Project

## Table of Contents

- [Authors](#authors)
- [Pre-requisites](#pre-requisites)
- [Libraries](#libraries)

## Authors

* [Soraia Reis](https://github.com/soraiareis)

## Pre-requisites

You should download and install these properly on your system. Visit the we

* [Java](https://www.java.com/en/download/)
* [Gradle](https://gradle.org/)
* [Firefox](https://www.mozilla.org/)

## Libraries
```

Setting up a basic structure to support a web-based application test automation is done here. In the next parts of this article, I will introduce the Dependency Injection concept, integrate it with Spring, show some configuration features, add reporting, embed screenshots, provide logging mechanisms, run the test in parallel and integrate with Docker.

Remember to commit your changes to Git:

```
git add .
git commit -m "Basic structure with Selenium and Cucumber"
git push
```

The implementation of this part can be found in my **demo-spring-selenium** project tagged as v1.0. You can download this release to check how it looks and/or use it as a starting point for Part 2 of this article.

Thanks for reading! I hope you found it helpful.

If you enjoyed it, I appreciate your help in spreading it by sharing it with a friend.

55 · 2 Comments

Like Comment Share

Ana Paula dos Santos 1y
Ana Paula dos Santos
Valéria Silva
Like Reply | 2 Likes

Bharath S 1y
Bharath S
Soraia Reis Fernandes Thanks for the next article series !! Excited to learn from you !!
Like Reply | 1 Like

See more comments

Soraia Reis Fernandes

More articles by this author

See all

Part 5 - Hands-On Test Automation Project with Java + Cucumber + Selenium + Spring + Docker

Part 5 - Hands-On Test Automation…

Sep 2, 2020

Part 4 - Hands-On Test Automation Project with Java + Cucumber + Selenium + Spring + Docker

Part 4 - Hands-On Test Automation…

Aug 26, 2020

Part 3 - Hands-On Test Automation Project with Java + Cucumber + Selenium + Spring + Docker

Part 3 - Hands-On Test Automation…

Aug 19, 2020