## horizontal line

Alphabet Framework

11.08.2021

# Overview

This document will outline our proposal for moving the current testing processes and practcies for ABC inc into the Browserstack pipeline by outlining an easy to maintain framework that can be scaled up with minimal effort.

# Technology Stack

1. Java will be the primary programming language.
2. Selenium will be used for testing the sites and is built into our testing framework by default.
3. SeLion Testing Framework (built on TestNG) will be the foundation framework that everything is built on.
4. Github for version control.
5. Github Actions for CI\CD (Continuous Integration / Continuous Delivery) and scheduling of tests.
6. For reporting, we will also use Github Actions as it has a very pro-active approach to alerting users to passed or failed tests via email and alerts in Github.

## Programming Language

We will be using Java as our programming language as this is the most commonly used and also the most flexible programming language for automation. Python was also an option but there are limitations in Python when it comes to pricing checks and since a large portion of the site deals with eCommerce, we want to side step any easy to avoid pitfalls.

We will be using the latest version of Java and leveraging several open source APIs to make the tests as robust as possible while maintaining efficiency.

## 

## 

## Test Framework

For the framework that can be used, there are so many options that we are literally spoiled for choice. In my own personal opinion, SeLion Framework is one of the best around in terms of features and scalability. I have built and deployed this framework in the past including a full stack deployment on a CI/CD pipeline in Jenkins, so I have a lot of familiarity with this particular framework. It was built by PayPal and is built using TestNG which in itself is one of the best test frameworks available.

With SeLion we can build our tests and run them together, independently, all at once or on a schedule and it integrates very well with BrowserStack enabling testing across all of our real devices and virtual browsers.

More information and documentation on SeLion can be found [here](https://www.symbio.com/solutions/quality-assurance/selion-test-automation/#:~:text=A%20Selenium%20WebDriver-based%20framework%20SeLion%20%28pronounced%20%E2%80%9Csea%20lion%E2%80%9D%29.SeLion,be%20used%20for%20testing%20web%20and%20mobile%20applications.).

## Version Control and CI/CD Actions

For the purposes of this prototype, we will be using GitHub for version control as we want to use GitHub Actions for our CI/CD actions and for scheduling tests to run daily or at specific intervals.

## Reporting

When it comes to reporting, there were a number of options such as Allure; which provides a very aesthetically pleasing report based on test output. However, it requires an extra layer of code to set up and run, and also to maintain. Others such as Surefire, which is built into SeLion by default, is a nice report but is not easy to generate from a remote run.

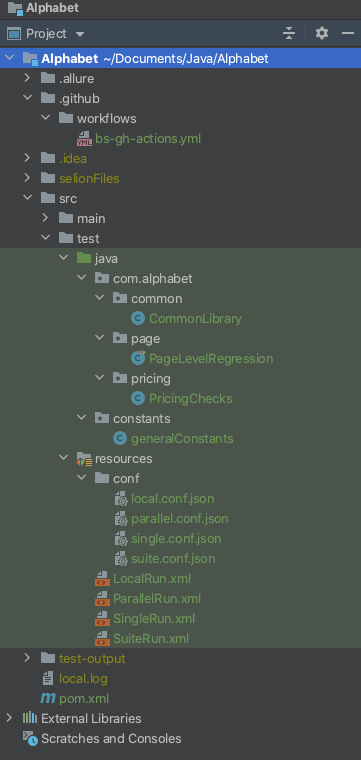
Thankfully, Github Actions serves as basically a report structure in how it manages our actions. When a test passes, you will get an email letting you know and is also easily visible from the code. Same goes for a failure. You will get an email which can take you through to the console output in one click and shows the entire stacktrace. As we are using Maven to manage the Java code, we can also enable full logging in the actions which adds an extra layer of detail to the logs.

I’m a big fan of keeping things simple, so using Github Actions for reporting is killing 2 birds with one stone here.

# Project Setup

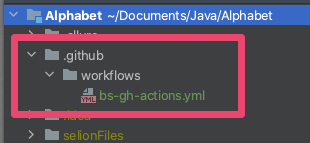
The project is currently hosted on Github at: <https://github.com/GaryBrowserStack/Alphabet>. You can check out the entire project locally using the git link: <https://github.com/GaryBrowserStack/Alphabet.git>.

Once you have checked it out into your IDE, you will see the following structure:



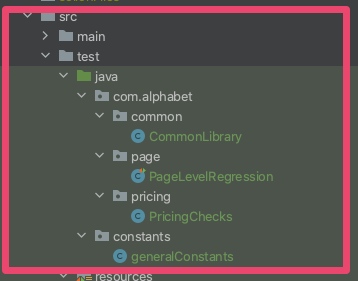
There is quite a bit to unpack here so let’s go through each section.

Inside the .github/workflows folder, you will see a yml file which is the file we use for our Github Actions. It outlines what to test, where to test it and what to use. We will go through this in more detail a bit further down.



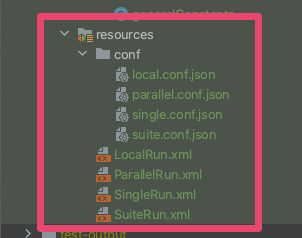
In the Java section, there are a number of things we need to look at:

1. The CommonLibrary is used to set up our BrowserStack specific configurations and allow the tests to run on our grid.
2. The different tests we want to run. For the purposes of this demo, I have set up both page level tests that will check various different things on the page like the products and the filters, and a pricing test that will check the pricing and that it matches between page and cart.
3. The generalConstants file which will contain any Xpaths that we will use along with any commonly used variables to avoid duplication in multiple files.



In the resources section, we have 2 types of files.

1. The JSON files that contain our BrowserStack specific configuration such as username and access key and what environments / browsers we want to run our tests on.
2. The XML files that serve as our “runners” and are used to actually run the automation.



Finally we have the pom.xml file which contains all our plugins and dependencies for the project and will be used to build the project using Maven.



# Project Configuration

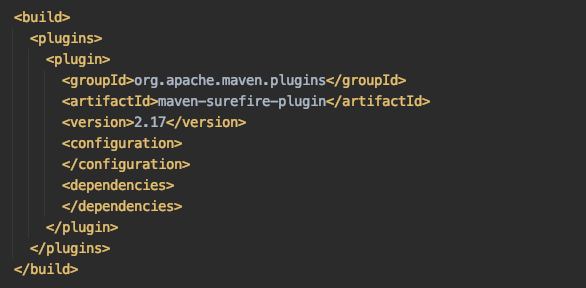
## Config Files

### POM.xml

In order to run anything within this framework, a number of configuration files are required to be in place. Thankfully, this is a single setup and will not require any further changes unless you plan to overhaul the project and move a lot of things around.

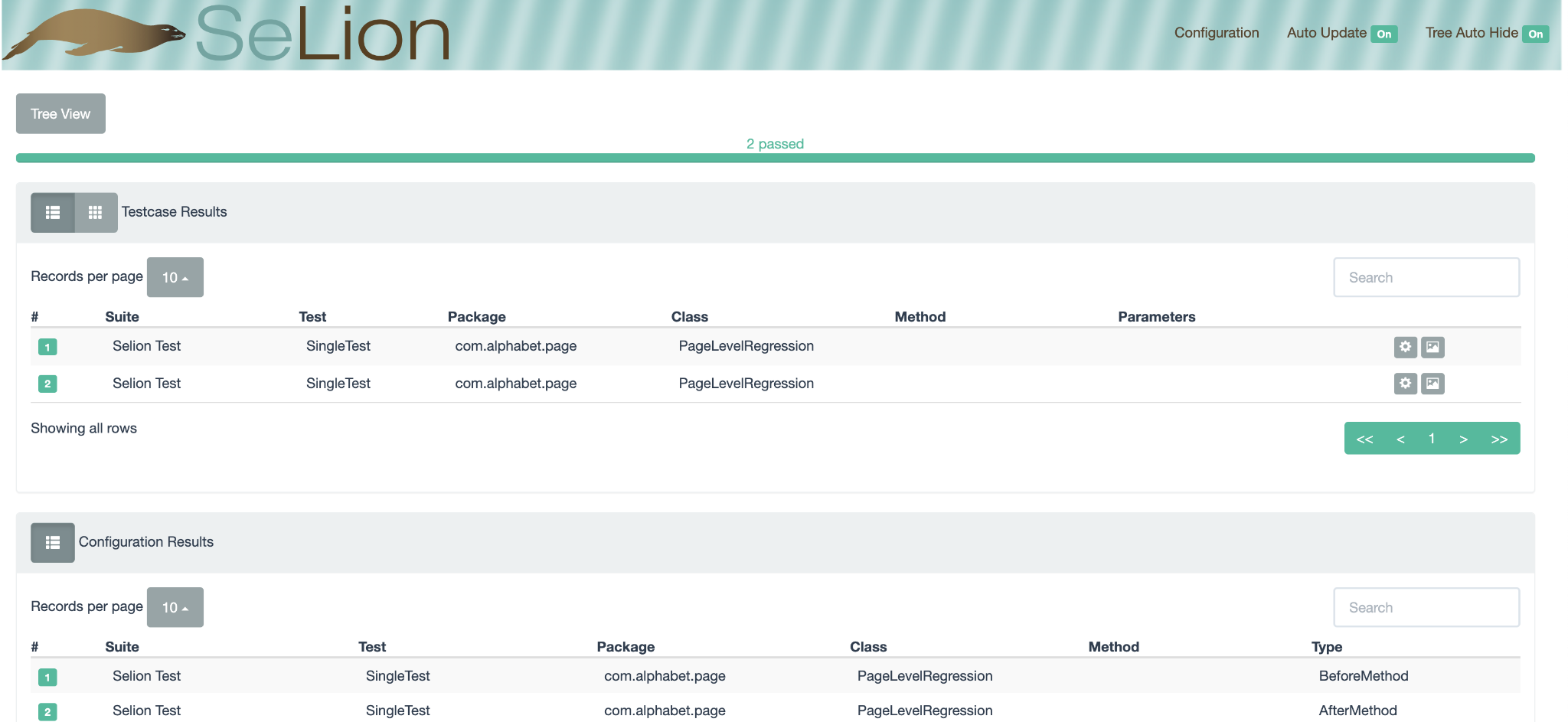
Let’s start with the POM.xml file. Most of this will be boilerplate as it is dictated by the SeLion framework so we don’t need to worry about it. The main sections we will be looking at that will require edits are the **plugins** and **dependencies** sections.

First, plugins. Here we will find the maven-surefire-plugin which handles local reporting



This will generate a report in the directory: *test-output/RuntimeReporter/index.html*

It will look a little like this:



As you can see it looks quite nice and has all the information on tests run. You can also check what tests were run in each by clicking into each method.

Next the dependencies section:



As you will see from the screenshot above, the first dependency is a ***SeLion*** one. This comes as standard in the POM file, as does the second one. The ***Selion-Grid*** is what we use to leverage WebDriver to check the pages.

Next we have a ***selenium-server*** dependency. This is used for local runs outside of BrowserStack. So if you are just running on your local browser, which in reality, you should not need to do, this is what will be used.

Next is the ***browserstack-local-java*** dependency that helps us run local tests on BrowserStack.

The final dependency; ***json-simple*** is used to read the JSON files that we use for BrowserStack configuration.

### BrowserStack Configuration

Speaking of BrowserStack, we need to set up the configuration to read the JSON files and use the BrowserStack URLs to run the tests. This is a boilerplate method and should not require any edits, but I have made some small edits for the purposes of this prototype.

While it may look daunting, all it is doing is reading and parsing the JSON to get all of the information that we need to connect to BrowserStack and perform our tests.

Before we look at this code, let’s take a look at a basic JSON file for a local run.

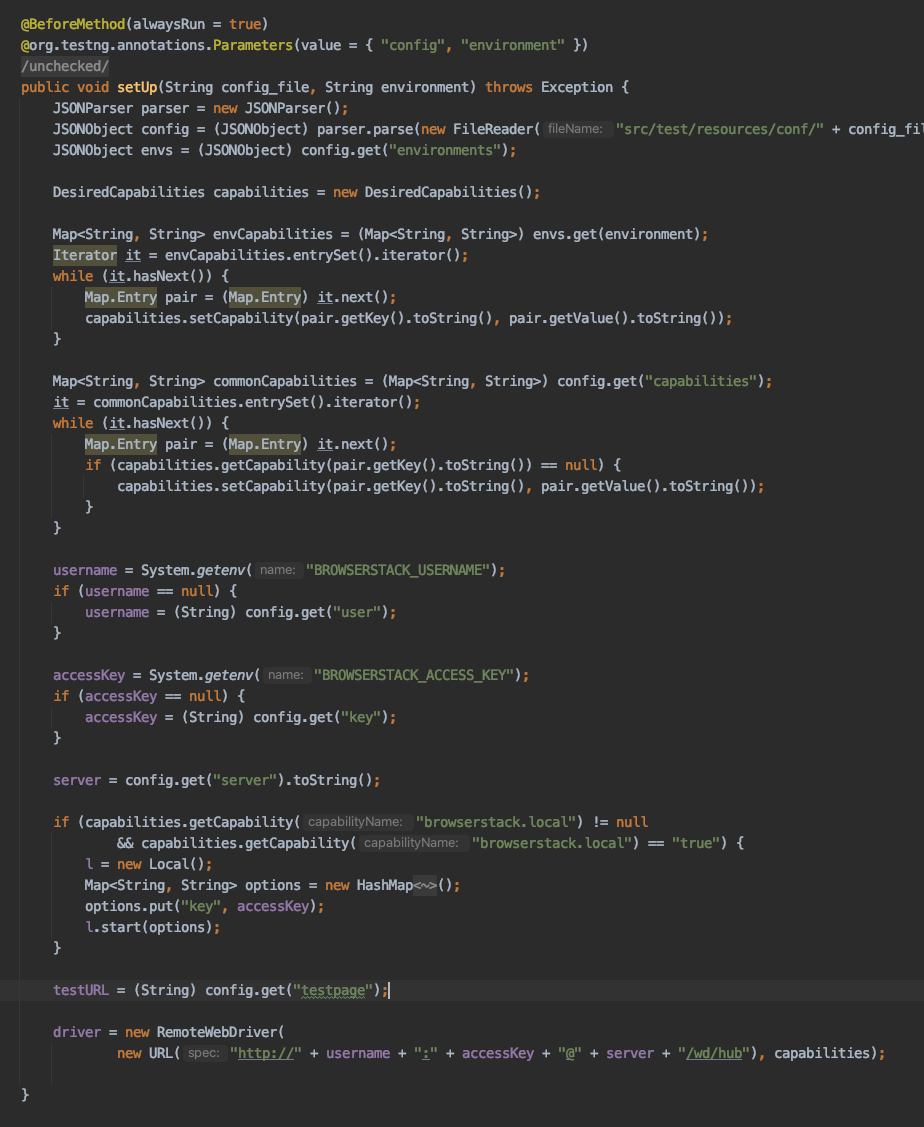


So what you will see here is a server URL which is the same across all JSON files, a username and access key, which is blurred out for obvious privacy reasons.

You will notice that we have a “browserstack.local” variable here that is set to true. This will only be applicable in the Local JSON file.

The environments section is used to set the browser we want to run against. In this example, we are using Chrome.

The next part of configuring the framework to use BrowserStack happens in the Java code, I have placed it in a file called CommonLibrary where we will place any common methods that can be reused across different tests.



There is a lot going on here, but all that really happened is the setup of the BrowserStack connection using our username, access key and server URL. It will also set up the environment parameters that are specified in the JSON.

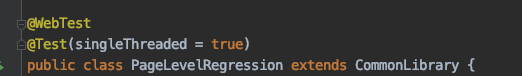
I have also added a new variable to the function called “***testURL***” that will get the testpage parameter from the JSON and we will use this to open the test page.

## Test Classes

I have written a simple test class to show how tests are set up on the Java side.

Selion has a number of identifiers that we need to use in order to mark this file as a test class that can be run in our XML runners (described in more detail below).

At the top of the class, we will define two of these identifiers:



WebTest and Test will tell the runner that this is a valid class to fetch. “***singleThreaded = true***” is used to stop the class from running multiple tests at once which can cause conflicts.



The ***@Test*** parameter at the top of each method tells the runner what code is considered a test within our test class. We may have methods in our test class to perform actions like setting up lists and other information that will not have this parameter.

The priority parameter within the braces denotes the order of the tests. Each number must be unique and in order for the tests to run successfully. We will receive errors for duplicates and missing numbers.

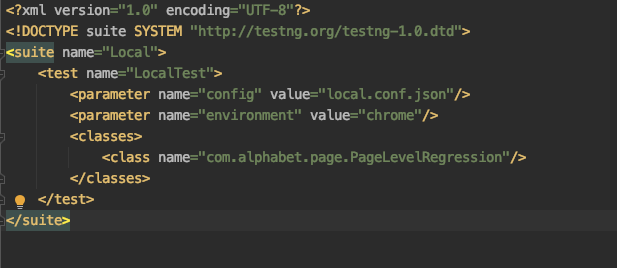
## XML Runners

The XML runners are the most important part of getting the framework to run. For the purposes of this prototype, we have 4 XML files. One for each of the following:

* Local Test
* Single Test
* Suite Test
* Parallel Test

Each one of these can have different tests run as part of them. They can also have the same, it will depend on the testing needs at any given time.

Below is an example of the local test XML:

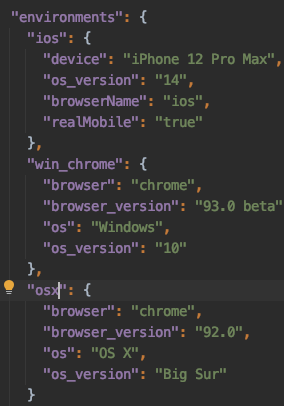


It doesn’t have much going on inside it. There is a config parameter pointing to our local JSON file. And inside the classes, we have our 1 test file. You can have any number of them in here as you wish, and because we set each one as “***singleThreaded***” earlier, they will run one at a time to prevent any overlaps.

Below is a more complex example for the Parallel runner:



The main difference between this and the local test is that we have multiple test sections. These tests, as the name suggests, will run in parallel for the environments specified in the JSON. Below is what those environments are:



# 

# 

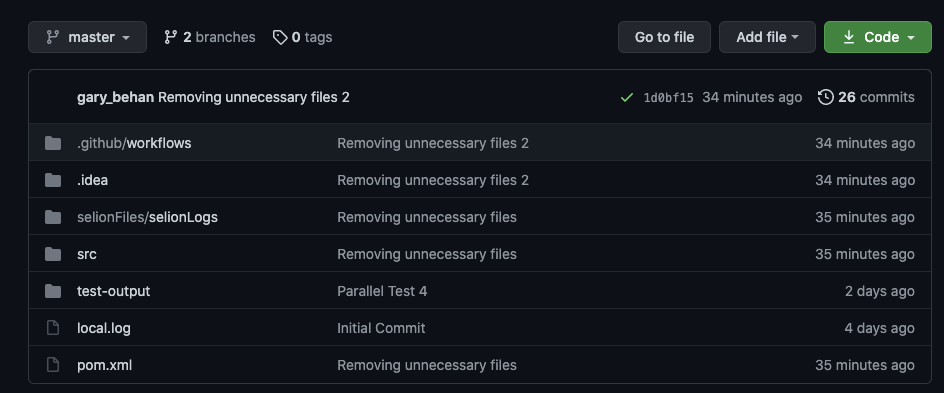
# 

# 

# Version Control and CI / CD

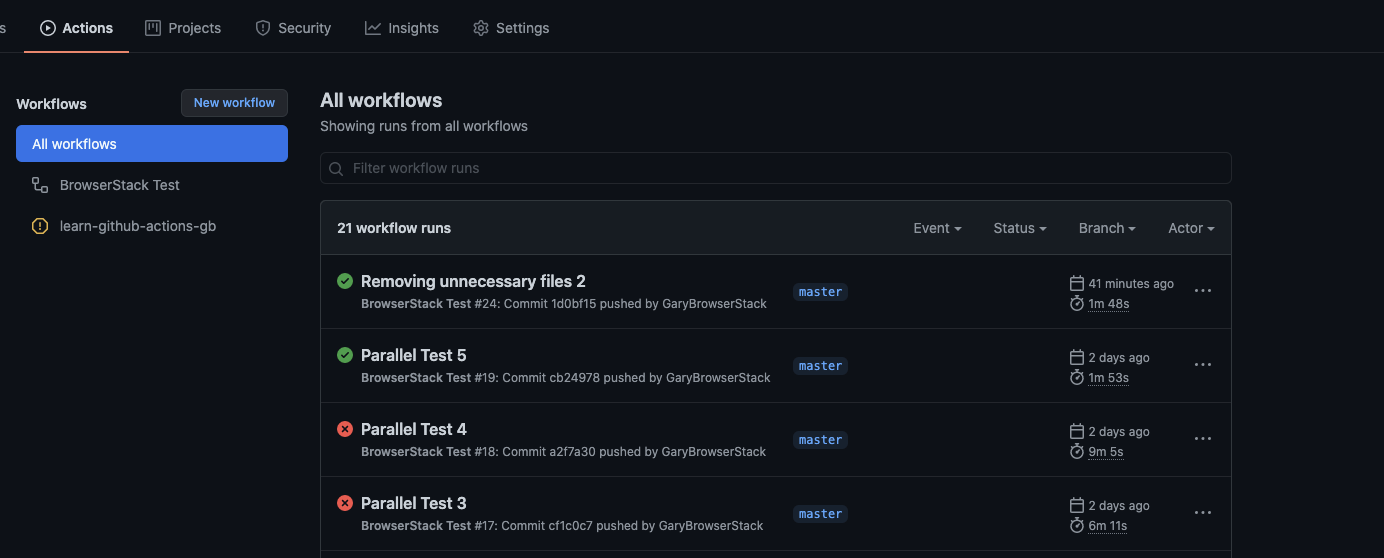
As mentioned previously, we will be using GitHub as our version control system. GitHub is open source but extremely popular. There are plenty of Enterprise version control solutions out there, including some that use Git as a base like BitBucket, but for the purposes of this prototype, we will be using GitHub.

The URL to access the web frontend for the project is: <https://github.com/GaryBrowserStack/Alphabet>



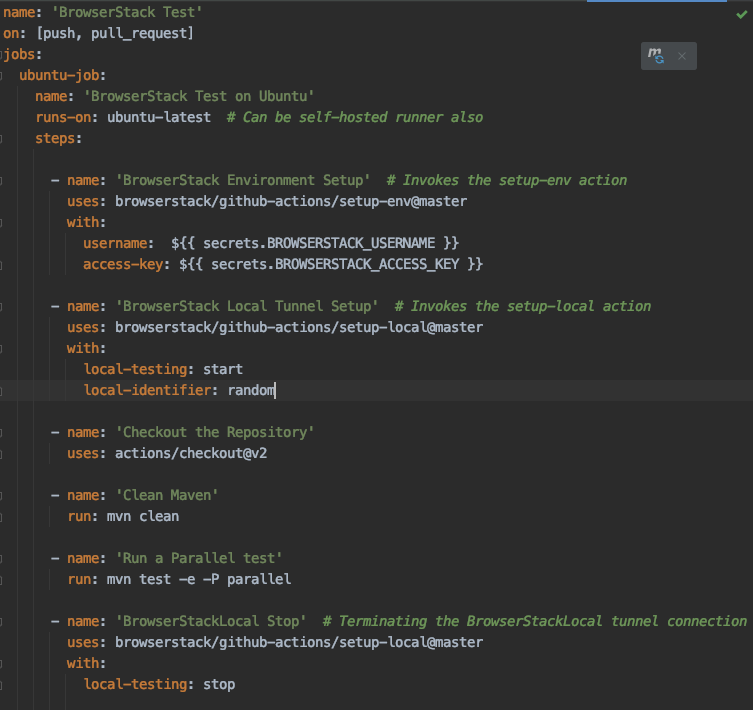
This contains the most recent check in of the code for the project and will allow us to see our actions and their status.

## Actions



The Actions tab will show what workflows have been set up and run on the project. Currently in the prototype there is one YML file that is equal to one workflow. There can be more than one test in a workflow and the workflow can also perform commands and installations.

Here is our YML file:



The first thing to note is that the workflow will run on both a ***pull*** and ***push*** request. So whenever we fetch the latest code from Git, or whenever we check in something new, the workflow will run and all tests will run.

Inside the steps section, there are a number of actions being performed. The first 2, and the last one are BrowserStack specific and will set up the BrowserStack environment for this remote run, open a tunnel to allow the tests to happen and stop the local tunnel at the end.

In between these, we first need to checkout the code from the repository to run the tests. Next step is to do a maven clean to remove any test files from previous runs. Then we will run a parallel test with debug logging to give us the most robust output.

If you remember from our XML runners section, the parallel test is running 3 tests. One for iOS, one for Windows Desktop and one for Mac OSX. This is what is being run by this Workflow file.

# Test Flakiness

It was pointed out to me that previous attempts at automation were hampered by test flakiness when running end to end tests. This can be tricky to deal with as you need data to support the theory that a test is performing in a flaky way.

If a test has failed consistently a number of times, the first thing to do is quarantine the test outside of CI/CD temporarily so that you can troubleshoot what the issue is. It can sometimes be an issue with the code itself, but other times, it can be down to a SeLion issue, a Selenium issue, such as a timeout or the page not loading correctly.

The most common thing I have seen with new automation setups using TestNG is in the way they are performing their Assert statements. The TestNG Assert will automatically fail the entire test so even if you have 100 tests and 99 of them will pass; if one in the middle of all of these were to fail, then the whole test fails and we do not get an accurate report of what is working.

Thankfully, there is a nice feature in SeLion that gets around this. Instead of using “***Assert.assertEquals()***” which is the default TestNG statement, we can use the SeLion default which goes: “***SeLionAsserts.verifyEquals()***”. Using ***verifyEquals*** rather than ***assertEquals*** (which can also be used in ***SeLionAsserts***) will allow a test to continue even if there is a failure on one test.

So to take an example:

If your code says:

***SeLionAsserts.assertEquals(“1”, “1”); // this is correct***

***SeLionAsserts.assertEquals(“2” “5”); // this is incorrect***

***SeLionAsserts.assertEquals(“3”, “3”); // this is correct***

When the test is run, the whole class will fail because of the middle line.

However, if we use verifyEquals, like below:

***SeLionAsserts.verifyEquals(“1”, “1”); // this is correct***

***SeLionAsserts.verifyEquals(“2” “5”); // this is incorrect***

***SeLionAsserts.verifyEquals(“3”, “3”); // this is correct***

Then the test will complete and it will only flag the incorrect one. Technically the test will still fail, but it will only show as a mismatch rather than an outright failure, and will also still run all the tests instead of being stopped in its tracks in the middle.

There is also a very nice extra feature with SeLionAsserts that allows us to enter an optional 3rd parameter to the verify statement that outputs a message with the Assertion result in the console. This allows us to describe what the specific test is doing so that if it fails, there will be an accurate descriptor of what it should be doing.

I believe that all of this will help immensely with flakiness in the E2E tests.