Jedd Hopkins  
Capgemini  
  
  
1. Introduction & Selenium Testing

**\*\*Please note, many method, class and variable names have been changed to generic terms such as ‘thing’ to maintain anonymity\*\***

**Contents**

Page 2 - Introduction

Page 4 - Acceptance Criteria & User Stories

Page 5 - Building the Test Class

Page 7 - Running Tests & Debugging

Page 9 - Peer Review and Merging the Branch

Page 11 - Conclusion

Page 12 - Appendix

**Introduction**

My name is Jedd Hopkins and I am a Junior Software Developer at Capgemini. Capgemini is a large multi-national consulting firm that primarily provides tech consulting services to clients in both the private and public sectors.

Since finishing my classroom training with Makers Academy in March 2019 I have been placed on a secure public sector project with Capgemini. The primary aim of this project is to replace an internal system which was originally written as a single-tiered, monolithic application. We are re-writing the application using a microservices architecture, splitting the application into a set of smaller, interconnected services. Due to the secure nature of the project and to protect confidentiality, I am unable to describe any specific details on the product we are building and throughout my portfolio many images or code samples will be redacted or heavily edited. Within the project there are six lots of work that were all available for bidding, Capgemini won two of these, however I am a developer on ‘Lot1’ which is responsible for building the framework. This will enable all of the other lots to work quicker and more efficiently.

At present, the development team I am part of is made up of four senior and a junior developer (myself). The wider project team includes a project manager, a delivery lead, a number of solution architects, platform engineers and business analysts, a scrum master and a product owner who is part of the client team. I am in regular contact with all members of the team, as well as client stakeholders.

The project runs using both Scaled Agile Framework (SAFe) and Scrum methodologies, and I am in one of eight scrum teams. This means that we operate program increments (PI’s) in three-month cycles. Major project level key ceremonies happen at the PI planning sessions such as PI objectives, key milestones and feature planning for the three months ahead. The features are then broken down into epics which are in turn broken down into stories. All of which I play a very active role in:

* Daily Stand-ups - these take place every morning within our dev team and are when each team member brings the rest of the group up to date with what they have completed, what they are working on and any obstacles that stand in their way. It is an ideal opportunity for individuals to ask questions about tickets they may need to pick up that day. As our team is small, we take turns in facilitating the meeting. This will largely consist of talking through each ticket and then asking anyone else who might have joined the call (BA, PM etc.) if they have any other business or important information to discuss.
* Retrospectives - The retrospective will happen at the end of the sprint and will include every member of the project team. It is an opportunity for the team to inspect itself and create a plan for improvements to be made in the next sprint. It is often a way for members of the client team to also come and sit in, creating a feeling of continuity between everyone. Towards the end of the retrospective, everyone is asked to evaluate how the sprint went. Writing down various things that went well, didn’t go so well and what we would do differently next time. This is then opened to the room for discussion, which is a great opportunity to seek help from my peers or even some recognition that I am doing well within my project.
* Sprint Demos – This is a chance to showcase demos of what we have accomplished in the previous sprint. These demos are presented to the wider teams along with the client team and are a brilliant way to show the value in what we are doing to the stakeholders.
* Backlog Refinement - This session is where user stories are written and estimated. The Product Owner will prioritise stories in the backlog which we will then point ready for the next. We do this by initially talking about the individual ticket and discussing the issues someone may have whilst picking up the ticket. We all then point the ticket, assessing many factors, on its complexity. Generally, if the pointing is close together then we will have a small discussion and a final pointing value will be made. If the pointing varies drastically, then a larger discussion will be had. We then come together to vote again, ideally agreeing. These meetings offer a great opportunity to learn about the different technologies we use at work and are a brilliant platform to ask questions, as there are plenty of senior members of the team in attendance.
* Sprint Planning – This is where we plan the upcoming sprint, taking into account our sprint velocity, the number of stories and the point size of the stories concerned.

With regards to management of source code, we use git with a methodology similar to 'trunk-based development'. This means that we have a main branch called 'master' where the most up to date approved version of the code lives. Developers create feature branches from this main branch, and once development of a feature is complete, they create a merge request for review by the rest of the team. Often this discussion results in a number of changes that need to be made. Once the branch has been approved by at least one member of the development team, it can be merged into master. As my time on this project has progressed, I have become more confident in performing these code reviews for other members of my team and am now regularly reviewing and merging code as the sole reviewer.

Both PI & sprint planning are tracked using Jira. This tool allows us, as a development team, to accurately track our sprint goals and burndowns, which is ideal for making sure that we are meeting our PI plan and sprint goals. Each sprint is projected on a virtual sprint board, along with its associated tickets. Each of the tickets’ subtasks can be seen clearly when viewing the current sprint and these subtasks can be moved and tracked as they are in progress and then completed. Each ticket can be assigned to individuals, making it very easy to see which tickets are currently free or in progress.

Luckily, we follow an agile framework rather than waterfall. This means that all the tools above can be utilised efficiently allowing for continuous iterations of development and testing. This is essential for my current project as the requirements have not yet been fully defined. Operating the way we do would be impossible using a waterfall model as that would leave us no scope to change requirements.

**Acceptance Criteria/User Story**

The development team have been tasked with creating a RESTful web service, and I was tasked with testing the features and user stories.

To do this I was using Selenium to drive the automation within the tests, and Cucumber to create the feature files and scenarios.

Both Selenium and Cucumber can be added into the project as dependencies in the pom.xml file. The dependency for Selenium is shown in Figure 1.1, other dependencies are very similar. Dependencies allow your project to call upon classes and methods provided by the chosen framework.

Figure 1.1 – Selenium dependency



Figure 1.2 – User story

Figure 1.2 shows the first user story I created in the feature file, written using Cucumber. There were many other user stories I created for this piece of work which all have to be signed off and agreed by the client. These user stories directly correlate to the business requirements and the functionality needed in the final product. By following these user stories, myself and my team can rest assured that the work we are doing is directly needed by the client. This specific user story tests that the web app home page is rendered correctly and then tests that the ‘Create Thing’ button is working correctly.

**Building the Test Class**

The first thing I did was to create a new branch. This enables me to make my edits to the code without interfering with the master branch.

In Figure 2.1, constants and field variables are initialised and declared at the start of the class.

Figure 2.1 – Class variables

Selenium detects methods annotated with @Before and executes those before each scenario.

Figure 2.2 – @Before & @After functions

As you can see in Figure 2.2, the field variables are instantiated in this step. Each of the classes ending ‘Page’ is a custom class I wrote that makes page features testable. I structured the tests aligning each class to a page, which makes the tests easy to read, maintain and debug.

Writing my tests in an Object Orientated (OO) way like above made it easy for me to manage what methods I could enact on each page object, but also a very easy way for me to track my progress through the test at any given point.

Selenium detects methods annotated with @After and executes those after each scenario

The method tearDown() first clears down everything that may have been created within each test that is associated with the THING\_PREFIX of ‘TEST-‘ including any lines in a database (This can be seen in the code at the end of this file where I show the whole class). It then closes the Webdriver, ready to start fresh on a new test.

Cucumber scenarios are automatically translated into methods when the script is executed.

Figure 2.3 – Cucumber @Given scenario

The method names map to the scenario task names. The scenario body calls the appropriate method from the page classes I defined above.

The method annotated with @Given in Figure 2.3 initiates the sequence. I initially want to just navigate to the home page. The Selenium driver objects’ get() method loads the URL passed into it, which in this case is the homepage.

In the Figure 2.4, the test simulates a user clicking the ‘create’ button.



Figure 2.4 – Button click

This is what I mentioned earlier about readability. Anyone can read this and immediately know what the method is used for.

If we dive a bit deeper into the clickCreateButton() method below, you can see exactly what needs to be done, initially to find the element on the page and then to click on it.

Figure 2.5 – Click function

The code above is taken from the ThingsListPage class, which houses all the methods I use to navigate and interact with the thingsListPage object.

Selenium detects objects annotated with @FindBy and assigns them to the page element that I specify, in this case, the ‘createThingButton’.

Now I can take advantage of the in-built method ‘click()’ that the WebElement uses.

As you can see, I have taken four lines of code which could make my test look particularly messy (especially if there are multiple elements to find a interact with) and refactored it down to just one line of code in my test.

**Running Tests & Debugging**

Next, I want to see if my test has passed. In this example I just want to know that I have navigated to the ‘New Thing’ page. The only problem here was that I did not know how to do this. I solved this by beginning to do some research around the problem. I quickly found that there are multiple ways to do this, but from my research using Google and StackOverflow, I found that the most simple and common way was to find an element on the new page that was unique to that page and check that it has loaded. I chose the main title/header of the page.

Figure 3.1 – Test scenario

Looking at Figure 3.1, the line ‘Assert.assertTrue(newThingPage.findPageHeader())’ is the bit of code which is checking if my test has passed. It runs the findPageHeader() method on the newThingPage object. Similar to our personalised clickCreateButton() method we used in the @When part of the test, the findPageHeader() method works behind the scenes to find the element on the page and then check that it is displayed.

Ultimately, if the element is displayed, then I will be on the correct page. If it is not displayed, then something has gone wrong.

I ran the test in Figure 3.1, but it failed. The error message explained that the WebDriver could not find the web element I was looking for. At first glance I thought that this problem could be down to the fact that the newThingPage object had not been instantiated properly, therefore being ‘null’ and having no elements or attributes associated with it.

There were two things I did here to try and get some visibility on the problem.

The first was to add a logger to the entry of the method, as show in Figure 3.2. A logger is and object which prints out information to the terminal. It is good practice to add logging into applications, as it is an easy way to give visibility of what is going on inside the application and is very useful when debugging code. This also increases the readability, reusability and maintainability of the code.

Figure 3.2 – Test scenario – adding logger

The second approach I took was to use the debug function within the IDE itself. To debug a test, I have to tell IntelliJ where I would like to pause the program, a place where I can stop and look around to investigate the test failure. I initially added my breakpoint on the exact line that was throwing the error.

I then ran the test in debug mode, and it paused where I asked it to. The first thing I saw when looking at the terminal was my logger statement, which confirmed to me that the newThingPage items was working perfectly fine.

From here I could then take a deeper dive the newThingPage object. The debug also proved to me that all my components had instantiated properly and were perfect for use.

I concluded from this that all my components were instantiating correctly but somehow still failing. Next I ran the test a few times and tried to watch closely as the Webdriver navigated the Chrome web browser.

With some discussion with my team I realised that the Webdriver was running the code quicker than the Chrome webpage could operate. This can be a common problem when choosing to run the tests in the browser rather than headless (just on the command line) as some machines are faster than others, and my current machine couldn’t quite keep up with the processing power needed to run the tests in the browser efficiently. Ultimately, my test would search for the header on the new page, but the new page would not have been rendered in the browser. So even though my test would eventually end up on the ‘New Thing’ page, my test had already run and failed.

This problem could have been overcome by simply running the tests on the command line. However due to the fact that we have client demo days every two weeks, where we demo the work we have been doing, along with any major improvements, it was crucial that we could show the client and the stakeholders how the code and the tests react in a real-world setting (the browser) as that is how they will be using it.

Doing some research, I came across the Awaitility DSL which can be added through a dependency, the same as Selenium is added. This framework allows me to create a wait within the test until a situation has been asserted.

Reading the code in Figure 3.3, within my @Then method, I am saying that I want Awaitility to use the await() method and wait for up to 5 seconds until it can assert that the ‘New Thing’ page header is there.

Figure 3.3 – Test scenario one - adding Awaitility library

This test would usually need just a couple of tenths of a second to get onto the new page, and as soon as it found the element, the test passed and it moved onto the next one. Even though I stated it could wait up to 5 seconds, it never did.

The importance of this piece of work, and many more to follow that are just like it, prove to us that what we are building is functional and meets the clients’ needs with regards to page navigation, whilst also giving us as a team a great foundation within this test suite to create more advanced tests at a quicker speed.

**Peer review and Merging the Branch**

After finishing the code for this ticket, I believed that the acceptance criteria had been met. All that remained was for me to get it reviewed and merged back into master. I pushed my changes to the branch I made earlier by entering the following commands to my terminal:

$ git add .

$ git commit -S -m "Acceptance criteria 1 created and test passing"

$ git push --set-upstream origin initial-branch

A screenshot of a cell phone

Description automatically generatedFrom there I was then able to go to the Gitlab UI and create a merge request. I then asked a senior member of my team to review my code and merge if they were happy with what I had produced.

Figure 4.1 – Asking for a merge request

In Figure 4.1 you can see a small exchange between myself and one of the senior developers on our team. I am letting them know that I have created a merge request for the work I have done on these tests. They then review my code and I can go back and make any amendments, which is a great way for me to adhere to standards within the company, but also learn how to look out for any spelling errors or small bugs that may impact my code in the future. The comments that my colleague pointed out were a variation of things such as discrepancies in my camelCase and that fact that I could refactor part of my tear down method by extracting some of the code into a different method, improving the readability of the class overall.

**Conclusion**

I really enjoyed this piece of work and it gave me a really good platform to learn new things, such as Selenium and front end testing/development, but also helped me reinforce what I knew about react from my time at Makers, as I had not done much front end development since then.

I have to say that even though Java is not technically an OO programming language, it was useful for me to structure my tests and classes in an OO way so that I could manage how they were passed around and what functions were able to be called on them. This also helped keep things simple for my colleagues, when they came to add tests or edit the ones we already have. The clean code and the OO design meant that they were able to easily follow the tests, resulting in time and effort saved.

My main challenge would definitely have been finding the root of the problem to which the Awaitility library solved. This problem arose from the fact that my failing tests were running too fast for the webpage to be rendered. Selenium then failed when trying to find elements on the webpage to then verify they were there. I realised this through debugging in my IDE and following the error messages. When my test is saying it cannot find a page header that I know is there, I can then deduce that it can either not find the element, through spelling the classname incorrectly, or the test was failing to fast for Selenium to catch up. As I double checked all classnames meticulously I knew it had to be the latter. I spoke to members of my team about the consequences of adding a wait into tests, and the feedback I got was that it was generally fine but to try and use them sparingly, as tests should not need much outside help to pass. From playing around with my tests and following error messages, I realised that I only needed to add the wait in at some parts where my tests were changing web page, this gave the test those extra few milliseconds to catch up and render the webpage.

This was also one of the first times on my project that I was able to build out classes from scratch. It gave me a great opportunity to take what I already knew about Java best practices but also comply with the coding standards we set at Capgemini. For example, we like to follow a pattern of instantiating all the objects we need at the top of the class, then these objects can be assigned, when I need them. Personally, I think this makes the code look a lot cleaner and can help the IDE to detect whether an object is not being used, thus highlighting the code for possibly deletion, further cleaning my code and making it more efficient.

**Appendix**

1. **The Full Test Class**