Mastering Selenium WebDriver

Increase the performance, capability, and reliability of your automated checks by mastering Selenium WebDriver

Mark Collin
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 2 'Producing the Right Feedback When Failing'
- A synopsis of the book’s content
- More information on Mastering Selenium WebDriver
Mark Collin is a professional software tester who has been working in the software testing industry since 2001. He started his career in the financial sector before moving in consultancy. As a consultant, he has had the privilege of working on numerous projects in many different sectors for various large and well-known companies. This has allowed him to gain an eclectic range of skills and proficiencies, which include test automation, security and penetration testing, and performance testing.

He loves technology and is always exploring new and exciting technology stacks to see how they can enhance the capability and reliability of automated checks. He is fully aware that though automation is good, at the moment, nothing can replace the human mind when it comes to having a well-rounded test solution.

He is a great believer in open source technology and spends a lot of time contributing towards open source projects. He is the creator and maintainer of the driver-binary-downloader-maven-plugin, which allows Maven to download ChromeDriver, OperaDriver, the IE driver, and PhantomJS to your machine as a part of a standard Maven build. He is also a core contributor to the jmeter-maven-plugin, a tool that allows you to run JMeter tests through Maven. Mark has also contributed code to the core Selenium code base.
Preface

This book is going to focus on some of the more advanced aspects of Selenium. It will help you develop a greater understanding of Selenium as a test tool and give you a series of strategies to help you create reliable and extensible test frameworks.

In the world of automation, there is rarely only one correct way of doing things. This book will provide you with a series of minimalistic implementations that are flexible enough to be customized to your specific needs.

This book is not going to teach you how to write bloated test frameworks that hide Selenium behind an impenetrable veil of obscurity. Instead, it will show you how to complement Selenium with useful additions that fit seamlessly into the rich and well-crafted API that Selenium already offers you.

What this book covers

Chapter 1, Creating a Fast Feedback Loop, shows you how to build a small but powerful Selenium framework that enables you to get started quickly. We will focus on building something that enables people to run Selenium tests quickly and easily without having to download and configure libraries and external binaries themselves.

Chapter 2, Producing the Right Feedback When Failing, shows you how to get the framework, which was built in the first chapter, running on a continuous integration server. We will then extend the framework so that it can connect to Sauce Labs (or any other Selenium Grid) and take screenshots when tests fail. We will finish off by looking at stack traces and how we can use the information in them to find out why our tests fail.

Chapter 3, Exceptions Are Actually Oracles, examines common exceptions that are thrown by Selenium when running tests. We will work through the exceptions in detail and explore ways to make sure that our tests don't throw them when they shouldn't.
Preface

Chapter 4, *The Waiting Game*, explores the most common cause behind test failures in automation. It will explain in detail how waits work in Selenium and how you should use them to ensure that you have stable and reliable tests.

Chapter 5, *Working with Effective Page Objects*, shows you how to use page objects in Selenium. It focuses on proper separation of concerns and also demonstrates how to use the Page Factory classes in the Selenium support package. It finishes off by demonstrating how to build fluent page objects.

Chapter 6, *Utilizing the Advanced User Interactions API*, shows how you can automate challenging scenarios such as hover menus and drag-and-drop controls. It will also highlight some of the problems that you may come across when using the Advanced User Interactions API.

Chapter 7, *JavaScript Execution with Selenium*, introduces the JavaScript executor and shows how you can use it to work around complex automation problems. We will also look at how we can execute asynchronous scripts that use a callback to notify Selenium that they have completed execution.

Chapter 8, *Keeping It Real*, shows you what cannot be done with Selenium. We will then go through a series of scenarios that demonstrate how to extend Selenium to work with external libraries and applications so that we can use the right tool for the job.

Chapter 9, *Hooking Docker into Selenium*, introduces Docker. We will have a look at how we can spin up a Selenium Grid using Docker and start the Docker containers as a part of the build process.

Chapter 10, *Selenium – the Future*, talks about how Selenium is changing as it becomes a W3C specification. You will also find out how you can help shape the future of Selenium by contributing to the project in multiple ways.
In this chapter, we are going to have a look at how we can make life easier for you when tests start failing. We will:

- Discuss where our tests should live and examine why
- Have a look at test reliability
- Have a look at ways we can force our tests to be run regularly
- Talk about continuous integration and continuous delivery
- Extend the project we started in the previous chapter so that it can run against a Selenium Grid
- Have a look at ways to diagnose problems with our tests

**Location, location, location**

Many companies still have discrete test and development teams. This is obviously not an ideal situation as the test team is usually not completely aware of what the development team is building. This also provides us with additional challenges if the test team is tasked with writing automated functional tests using the web frontend.

The usual problem is that the test team is behind the development team; how far behind depends upon how frequent development releases are. The thing is, it doesn’t really matter how far behind the development team you are. If you are behind them you will always be playing catch-up. When you are playing catch-up, you are constantly updating your scripts to make them work with a new software release.
Some people may call fixing their scripts to work with new functionality refactoring; they are wrong! Refactoring is rewriting your code to make it cleaner and more efficient. The actual code, or in our case test script, functionality does not change. If you are changing the way your code works, you are not refactoring.

While constantly updating your scripts is not necessarily a bad thing, having your tests break every time there is a new release of code is a bad thing. If your tests continually stop working for no good reason, people are going to stop trusting them. When they see a failing build they will assume that it's another problem with the tests, and not an issue with the website you are testing.

So we need to find a way to stop our tests from failing all of the time for no good reason. Let's start off with something easy that shouldn't be too controversial; let's make sure that the test code always lives in the same code repository as the application code.

How does this help?

Well, if the test code lives in the same repository as the application code it is accessible to all the developers. In the previous chapter, we had a look at how we could make it really easy for developers to just check out our tests and run them. If we make sure that our test code is in the same code base as the application code, we have also ensured that any developers who are working on the application will automatically have a copy of our tests. This means that all you have to do now is give your developers a command to run and then they can run the tests themselves against their local copy of code and see if any break.

Another advantage of having your test code in the same repository as the application code is that developers have full access to it. They can see how things work, and they can change the tests as they change the functionality of the application. The ideal scenario is that every change made to the system by developers also results in a change to the tests to keep them in sync. This way, the tests don't start failing for no real reason when the next application release happens and your tests become something more than an automated regression check; they become living documentation that describes how the application works.
Tests are living documentation

So what do I mean by living documentation? As the application is built, automated tests are continually being written to ensure that specific criteria are met. These tests come in many different shapes and sizes, ranging from unit tests to integration tests and leading up to end-to-end functional tests and beyond. All of these tests describe, in some way, how the application works. You have to admit that this sounds just like documentation.

This documentation may not be perfect, but that doesn't stop it from being documentation. Think of an application that has some unit tests, and maybe one badly written end-to-end test. I would equate that with the sort of documentation that you would get with a cheap electrical product, from somewhere like China. It comes with a manual that will undoubtedly include a small, badly written English section which doesn't really tell you very much. It will also have lots of documentation in a language that you probably don't understand, in this case Chinese, which is very useful for somebody who speaks that language. This doesn't mean that the product is bad; it's just hard to work out what to do with it. Most of the time, you can work out what to do without the manual. If it's really complex you will probably go and find somebody who either speaks Chinese, or knows how the product works, and get them to explain it to you.

When I talk about tests as documentation, I usually think of different test phases as being documentation for different people. Let's take the unit tests; these are highly technical in nature and explain how tiny bits of the system work in extreme detail. If you compared this to the manual of an electrical product, these would probably be the tech specs in the appendix that provide lots of in-depth information that most consumers don't care about. Integration tests would probably be the part of the manual that explains how to connect your electrical appliance to other electrical appliances. This is very useful if you are going to connect to another electrical appliance, but you probably don't care about it that much if you aren't. Finally the functional end-to-end tests are the bit of the documentation that actually tells you how to use the appliance. This is the part of the manual that will be read the most by the average user (they probably don't care about the technical nitty-gritty).
I think one of the most important things that you can do when writing automated tests is make sure that they are good documentation. This means: make sure that you describe how all the parts of the application you are testing work (or, to put it another way, have a high level of test coverage). The hardest part though is making the tests understandable for people who are not technical. This is where domain-specific languages (DSLs) come in where you can hide the inner workings of the tests behind human-readable language. Good tests are like good documentation; if they are really good they will use plain English and describe things so well that the person reading them will not need to go anywhere else to ask for help.

So why is it living documentation, rather than just normal documentation? Well, it's living because every time the application you are testing changes, the automated tests change as well. They evolve with the product and continue to explain how it works in its current state. If our build is successful, our documentation describes how the system currently works.

Do not think of automated tests as regression tests that are there to detect changes in behavior. Think of them as living documentation that describes how the product works. If somebody comes and asks you how something works, you should ideally be able to open a test that can answer their question. If you can't, you probably have some missing documentation.

So where does regression testing come into this? Well it doesn't. We don't need a regression-testing phase. Our test documentation tells us how the product works. When the functionality of our product changes, the tests are updated to tell us how the new functionality works. Our existing documentation for the old functionality doesn't change unless the functionality changes.

Our test documentation covers regression and new functionality.

**Reliability**

When it comes to automation, reliability of tests is the key. If your tests are not reliable they will not be trusted, which can have far-reaching consequences. I'm sure you have all worked in environments where test reliability has been hard for one of many reasons; let's have a look at a couple of scenarios.
The test automation team that works in isolation

One of the more common reasons that tests are not reliable is having a dedicated test automation team who work in isolation from the team that develops the application. This should really be avoided if possible as the test automation team is always playing catch-up. The development team rolls out new features that the test automation teams have to automate, but they are never sure what is coming next. They usually find out that existing features have changed when their tests break and as well as fixing them they need to work out what the new functionality is and whether it is behaving as expected.

Something that normally happens in situations like these is that the test manager realizes that they don't have enough time to do everything and they look for ways to reduce the workload. Do you fix the existing failing tests and not automate new tests, instead getting some manual regression scripts put together to cover the gap? Do you continue automating the new functionality and accept that some of your old tests will fail? If you do, how do you deal with failing tests? Maybe put aside some time to fix failing tests and accept that your tests will never be green?

This is where you usually start to hear suggestions that it is time to lower the bar for the automated tests. "It should be fine as long as 95 percent of the automated tests pass; we know we have high coverage and those failing 5 percent are probably due to changes to the system that we haven't yet had the time to deal with." Everybody is happy at first; they continue to automate things and make sure that 95 percent of tests are always passing. Soon though the pass mark starts to dip below 95 percent. A couple of weeks later a pragmatic decision is taken to lower the bar to 90 percent, then 85 percent, then 80 percent. Before you know it tests are failing all over the place and you have no idea which failures are legitimate problems with the application, which ones are expected failures, and which ones are intermittent failures.

When tests go red nobody really pays attention any more, they just talk about that magic 80-percent line: It's a high number – we must have a decent product if that many tests are still passing, right? If things dip below that line we massage a few failing tests and make them pass, usually the low hanging fruit because we don't have time to spend trying to tackle the really thorny issues.
I hate to break it to you, but if you are in this situation your automation experiment has failed and nobody trusts your tests. Instead of looking at that 80 percent number you need to look at the other side of the coin; 20 percent of the functionality of your site is not working as expected and you don't know why! You need to stop the developers from writing any new code and work out how to fix the massive mess that you are currently in. How did you get here? You didn't think test reliability mattered and that mistake came back to bite you.

**Oh that test always flickers, don't worry about it**

This scenario is one that can occur in both isolated automation teams and integrated teams where everybody works together. You have probably seen automated tests that are not totally reliable; you know that one flickering test that occasionally fails for no obvious reason. Somebody once had a look at it and said that there was no reason for it to fail so it got ignored; and now, whenever it fails again, somebody says, "Oh, it's that flickering test again, don't worry about it. It will be green again soon".

A flickering test is one that intermittently fails for no obvious reason and then passes when you run it again. There are various phrases used to describe tests like this; you may have heard of them described as flaky tests, random failures, unstable tests, or some other name unique to your company.

The thing is that we now have a problem; tests do not flicker for no reason. This test is desperately trying to tell you something and you are ignoring it. What is it trying to tell you? Well you can't be sure until you have found out why it is flickering; it could be one of many things. Among the many possibilities a few are:

- The test is not actually checking what you think it is checking
- The test may be badly written
- There may be an intermittent fault in the application that is under test (for example, there may be a race condition nobody has identified yet)
- Maybe you have some problems with a date/time implementation (it's something that is notoriously hard to get right and the cause of many bugs in many systems)
- Network problems—is there a proxy getting in the way?

The point is that while your test is flickering we don't know what the problem is, but don't fool yourself; there is a problem. It's a problem that will at some point come back and bite you if you don't fix it.
Let's imagine for a moment that the software you are testing is something that buys and sells shares and you are pushing new releases out daily because your company has to stay ahead of the game. You have a test that has been flickering for as long as you can remember. Somebody once had a look at it, said they couldn't find any problems with the code, and that the test was just unreliable; this has been accepted and now everybody just does a quick manual check if it goes red. A new cut of code goes in and that test that keeps flickering goes red again. You are used to that test flickering and everything seems to work normally when you perform a quick manual test, so you ignore it. The release goes ahead, but there is a problem; suddenly your trading software starts selling when it should be buying, and buying when it should be selling. It isn't picked up instantly because the software has been through testing and must be good so no problems are expected. An hour later all hell has broken loose, the software has sold all the wrong stock and bought a load of rubbish. In the space of an hour the company has lost half its value and there is nothing that can be done to rectify the situation. There is an investigation and it's found that the flickering test wasn't actually flickering this time; it failed for a good reason, one that wasn't instantly obvious when performing a quick manual check. All eyes turn to you; it was you who validated the code that should never have been released and they need somebody to blame; if only that stupid test hadn't been flickering for as long as you can remember...

The preceding scenario is an extreme, but hopefully you get the point: flickering tests are dangerous and something that should not be tolerated.

We ideally want to be in a state where every test failure means that there is an undocumented change to the system. What do we do about undocumented changes? Well, that depends. If we didn't mean to make the change, we revert it. If we did mean to make the change, we update the documentation (our automated tests) to support it.

**Baking in reliability**

How can we try to enforce reliability and make sure that these changes are picked up early?

We could ask our developers to run the tests before every push, but sometimes people forget. Maybe they didn't forget, but it's a small change and it doesn't seem worth going through a full test run for something so minor. (Have you ever heard somebody say, "It's only a CSS change"?) Making sure that the tests are run and passed before every push to the centralized source code repository takes discipline.

What do we do if our team lacks discipline? What if we still keep getting failures that should have been easily caught, even after we have asked people to run the tests before they push the code to the central repository? If nothing else works we could have a discussion with the developers about enforcing this rule.
Producing the Right Feedback When Failing

This is actually surprisingly easy; most source code management (SCM) systems support hooks. These are actions that are automatically triggered when you use a specific SCM function. Let's have a look at how we can implement hooks in some of the most widely used SCM systems.

**Git**

First of all, we need to go to the SCM root folder (the place where we originally cloned our project). Git creates a hidden folder called .git that holds all the information about your project that Git needs to do its job. We are going to go into this folder, and then into the hooks sub folder:

```bash
cd .git/hooks
```

Git has a series of predefined hook names. Whenever you perform a Git command, Git will have a look in the hooks folder to see if there are any files that match any predefined hook names that would be triggered as a result of the command. If there are matches Git will run them. We want to make sure that our project can be built, and all of our tests are run, before we push any code to Git. To make this happen we are going to add a file called pre-push. When that file is added we are going to populate it with the following content:

```bash
#!/usr/bin/env bash
mvn clean install
```

This hook will now be triggered every time we use the `git push` command.

One thing to note about Git hooks is that they are individual for every user; they are not controlled by the repository you push to, or pull from. If you want them automatically installed for developers who use your code base, you need to think outside the box. You could for example write a script that copies them into the .git/hooks folder as part of your build.

We could have added a pre-commit hook, but we don't really care if the code doesn't work on the developer's local machine (they may be half way through a big change and committing code to make sure they don't lose anything). What we do care about is that the code works when it is pushed to the central source code repository.

If you are a Windows user, you may be looking at the preceding script and thinking that it looks very much like something that you would put on a *nix system. Don't worry, Git for Windows installs Git bash, which it will use to interpret this script, so it will work on Windows as well.
SVN

SVN (subversion) hooks are a little more complicated than Git hooks; they will depend upon how your system is configured to a degree. The hooks are stored in your SVN repository in a sub folder called hooks. As with Git, they need to have specific names (a full list of which is available in the SVN manual). For our purposes we are only interested in the pre-commit hook, so let's start off with a *nix-based environment. First of all we need to create a file called pre-commit, and then we will populate it with:

```bash
#!/usr/bin/env bash
mvn clean install
```

As you can see it looks identical to the Git hook script; however there may be problems. SVN hooks are run against an empty environment, so if you are using an environment variable to make mvn a recognized command, things may not work. If there is a symlink in /usr/bin or /usr/local/bin/, you should be fine; if not, you will probably need to specify the absolute file path location to the mvn command.

Now we need to also make this hook work for people using Windows. It will be very similar, but this time the file needs to be called pre-commit.bat; this is because SVN looks for different files in different operating systems.

`mvn clean install`

Again it's pretty similar; we just don't need to have a Bash shebang. Windows suffers from the same empty environment problems so again you will probably have to supply an absolute file path to your mvn install command. Let's hope that everybody developing in Windows has installed Maven to the same place.

It is worth bearing in mind that hooks like this are not infallible; if you have some local changes on your machine that are not committed the tests may pass, but that code will not be pushed to the central code repository, resulting in a build failure if anybody else tries to run it. As with all things, this is not a silver bullet, but it can certainly help.

We have now made sure that our tests run before the code is pushed to our central code repository so we should have caught the vast majority of errors; however, things are still not perfect. It's possible that one of the developers made a code change that they forgot to commit. In this case the tests will run on their local machine and pass, but an important file that makes this change work will be missing from the source control. This is one of the causes of works on my machine problems.
It's also possible that all files have been committed and the tests pass, but the environment that is on the developers' machines is nothing like the production environment where the code will be deployed. This is probably the main cause of works on my machine problems.

What do we do to mitigate these risks and ensure that we quickly find out when things do go wrong despite everybody doing their best to ensure everything works?

**Continuous integration is the key**

Continuous integration is a way to try and mitigate the issues that we come across by only building and testing code on our development machines. Our continuous integration server will monitor our source code repository and then every time it detects a change it will trigger a series of actions. The first action will be to build the code, running any tests that it can as it builds the code (usually unit tests), and then creating a deployable artifact. This artifact will then usually be deployed to a server that is a replica of the live environment. Once this code has been deployed to a server, the rest of our tests will be run against that server to ensure that everything is working as expected. If things do not work as expected, the build fails and the development team is notified so that they can fix the problems. It's important to note that we only build the artifact once; if we rebuild it multiple times, we will be testing artifacts that are potentially different at every step (maybe it was built with a different version of Java, maybe it had different properties applied to it, and so on).

With continuous integration we are looking for a workflow like this:

![Basic CI Workflow](image)

Most continuous integration systems also have big visible dashboards to let people know the status of the build at all times; if your screen ever goes red, people should stop what they are doing and fix the problem as soon as possible.
Let’s have a look at how easily we can get our tests running on a continuous integration server. This is not going to be a fully featured continuous integration setup, just enough for you to run the tests we have built so far. It should be enough to familiarize you with the technologies, though.

The first thing we are going to do is configure a Maven profile. This will enable us to isolate our Selenium tests from the rest of the build if desired, so that we can turn them in a separate UI block of tests on our continuous integration server. This is a very simple change to our POM; we are simply going to wrap our `<build>` and `<dependencies>` blocks with a profile block. It will look like this:

```xml
<profiles>
  <profile>
    <id>selenium</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    <build>
      <plugins>
        <plugin>
          ...
        </plugin>
      </plugins>
    </build>
    <dependencies>
      <dependency>
        ...
      </dependency>
    </dependencies>
  </profile>
</profiles>
```

As you can see, we have created a profile called selenium. If we want to run this in isolation we can now use the following command:

```bash
mvn clean install -Pselenium
```

You will also notice that we have added `<activeByDefault>true</activeByDefault>`; this will ensure that this profile is active if no profiles are specified on the command line, so you will find that the following command still works:

```bash
mvn clean install
```

This is to ensure that our tests are still run as part of a normal build, and the SCM hooks that we set up previously still do their job.
Producing the Right Feedback When Failing

We are going to look at two popular continuous integration servers. TeamCity is a personal favorite of mine and Jenkins is one that is prolific; you have probably seen a Jenkins install at some point in your career.

**TeamCity**

TeamCity ([https://www.jetbrains.com/teamcity/](https://www.jetbrains.com/teamcity/)) is an enterprise-level continuous integration server. It supports a lot of technologies out-of-the-box and is very reliable and capable. One of my favorite features is the ability to spin up AWS ([Amazon Web Services](http://aws.amazon.com)) cloud build agents. You will need to create the build agent AMI ([Amazon Machines Image](https://aws.amazon.com)) but, once you have done this, your TeamCity server can start up, however, many build agents are required and then shut them down again when the build has finished.

A basic TeamCity install should be pretty simple; you just need an application server that can host WAR files. One of the most commonly used application servers is Apache Tomcat, and the install is pretty simple. If you have a working Tomcat install, then all you need to do is drop the WAR into the `webapps` directory. Tomcat will do the rest for you.

Let's have a look at how we can get our build, up-and-running in TeamCity. When you first get into a new install of TeamCity you should see the following screen:
Now, follow these steps:

1. Let's start off by clicking on the **Create New Project** button.

![Creating a new project](image1)

2. We then need to provide a name for our project and we can add a description to let people know what the project does. Bear in mind that this is not an actual build we are creating yet, it is something that will hold all of our builds for this project. Selenium Tests is probably not a great name for a project, but that's all we have at the moment. Click on **Create** and you will see your project created.

3. We then need to scroll down to the **Build Configurations** section:

![Build Configurations](image2)

Projects need build configurations
4. When you get there, click on the **Create Build Configuration** button.

![Creating a build configuration](image)

Creating a build configuration

This is where we are going to create our build. I've simply called it **WebDriver** because it is going to run **WebDriver** tests, I'm sure you can come up with a better name for your build configuration.

5. When you are happy with the name for your configuration, click on the **Create** button.

![Configuring source control](image)

Configuring source control
6. Now we will be asked to configure our source control system so that TeamCity can monitor your source control for changes. I've selected Git and put in some valid values as a guideline; you will obviously need to put in values that relate to your source control system. Once you have created your source control link, it will ask you about the build you want to create.

![Build Configuration Settings](image)

**Configurations need build steps**

7. This is where we get into the meat of our build; click on **Add build step** to get started.

![Adding a Maven build step](image)
8. First of all we need to select the type of build; in our case, we have a Maven project so select **Maven**.

![Configuring your Maven build step](image)

9. Finally, we just need to put in the details of our Maven build; you will need to click on the **Show Advanced** options link to display the items in orange. Scroll down and click on **Save** and your TeamCity build is all ready to go. We now just need to make sure that it will trigger every time that you check code into your source code repository. Click on **Triggers**.

![Triggers](image)

10. This is where you set up a list of actions that will result in a build being performed. Click on **Add new trigger** and select **VCS Trigger**.

---

[56]
Adding a SCM trigger

11. If you click on **Save** now, a trigger will be set up and will trigger a build every time you push code to your central source code repository.

**Jenkins**

Jenkins (http://jenkins-ci.org) is a firm favorite in the **continuous integration** (CI) world and is the basis for some cloud services (for example: CloudBees—https://www.cloudbees.com). It is very widely used and no section on continuous integration would be complete without mentioning it.

A basic Jenkins install should be pretty simple; you just need an application server that can host WAR files. One of the most commonly used application servers is Apache Tomcat, and the install is pretty simple. If you have a working Tomcat install, then all you need to do is drop the WAR into the `webapps` directory. Tomcat will do the rest for you.

Let's have a look at how we can set up a build in Jenkins that will enable us to run our tests.
Producing the Right Feedback When Failing

Now, follow these steps:

1. The first thing to do is create a new project:

   ![Creating a project](image)

   Creating a project

2. Put in the name of your build and then select the **Build a maven project** option.

   Jenkins can do clever things if you have a Maven project. One annoyance is that it will disable `maven-failsafe-plugin` and `maven-surefire-plugin` build failures and let the Maven portion of your build complete. It then checks to see if there were any failures and marks the build as unstable if there were. This means a failed build may well show up as yellow instead of red. To get around this you can always select a freestyle project and add a Maven build step.
Next click on **OK** and you will be taken to a screen that looks like this:

![Jenkins configuration screen](image)

The project configuration

3. You then need to enter two bits of information—first of all the information about your code repository:

   ![Source Code Management](image)

   Selecting your SCM

   If you want to use Git with Jenkins, you will need to download the Git plugin. Jenkins does not support Git out-of-the-box.
4. Then we need to set up our Maven job:

![Maven build configuration](image)

Configuring our Maven build step

5. Using Maven actually makes it very easy, that's all there is to it. You should now be able to run your Jenkins build and it will download all the dependencies and run everything for you.

So far we have looked at how we can set up a very simple continuous integration service; however this is only the tip of the iceberg. We have used continuous integration to give us a fast feedback loop so that we are notified of, and can react to, problems quickly. What if we could extend this to tell us not just whether there are any problems, but whether something is ready to be deployed into production instead? This is the goal of continuous delivery.

![Continuous Delivery workflow](image)
So what's next after continuous delivery — how about continuous deployment? This is where we get to the point in our pipeline where we are confident that, as soon as every continuous delivery phase has been marked as passed, the code will automatically be deployed to live. Just imagine a new feature being completed and within a matter of hours we have performed enough testing on that functionality to be confident about releasing it, so that the feature is immediately available to your customers.

We haven't got quite that far yet. We now have a basic setup that we can use to run our tests on a CI; however, we still have some pretty big gaps. The basic CI setup that you have so far is running on one operating system and cannot run our tests against all browser/operating system combinations. We can deal with this issue by setting up various build agents that connect to our CI server and run different versions of operating systems/browsers. This does however take time to configure and can be quite fiddly. You could also extend the capabilities of your CI server by setting up a Selenium Grid that your CI server can connect to and run various Selenium test jobs. Again this can be very powerful, but it also does have setup costs. This is where third-party services such as Sauce Labs (https://saucelabs.com) can be used. Most third-party grid services have free tiers, which can be very useful when you are getting started and working out what works for you. Remember that getting set up with one third-party service does not lock you into it. One Selenium Grid is pretty much the same as another, so even though you start off using a third-party server, there is nothing to stop you building up your own grid, or configuring your own build agents and moving away from the third-party service in the future.
Extending our capabilities by using a Selenium Grid

Since we already have a working Maven implementation, let's enhance it so that it can connect to a Selenium Grid. These enhancements will enable you to connect to any Selenium Grid, but we are going to specifically look at connecting to a third-party service provided by Sauce Labs since it offers a free tier. Let's have a look at the modifications we need to make to our TestNG code.

We will start off with the modifications to our POM; first of all we are going to add some properties that we can configure on the command line:

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  <!-- Dependency versions -->
  <selenium.version>2.45.0</selenium.version>
  <!-- Configurable variables -->
  <threads>1</threads>
  <browser>firefox</browser>
  <overwrite.binaries>false</overwrite.binaries>
  <remote>false</remote>
  <seleniumGridURL/>
  <platform/>
  <browserVersion/>
</properties>
```

I've left the `seleniumGridURL` element blank because I don't know your Selenium Grid URL, but you can give this a value if you want. The same applies to `platform` and `browserVersion`. Next we need to make sure these properties are read in as system properties so we need to modify our `maven-failsafe-plugin` configuration:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>2.17</version>
  <configuration>
    <parallel>methods</parallel>
    <threadCount>${threads}</threadCount>
    <systemProperties>
      <browser>${browser}</browser>
      <remoteDriver>${remote}</remoteDriver>
      <gridURL>${seleniumGridURL}</gridURL>
      <desiredPlatform>${platform}</desiredPlatform>
    </systemProperties>
  </configuration>
</plugin>
```
<desiredBrowserVersion>${browserVersion}</desiredBrowserVersion>
<!--Set properties passed in by the driver binary downloader-->
<phantomjs.binary.path>${phantomjs.binary.path}</phantomjs.binary.path>
<webdriver.chrome.driver>${webdriver.chrome.driver}</webdriver.chrome.driver>
<webdriver.ie.driver>${webdriver.ie.driver}</webdriver.ie.driver>
<webdriver.opera.driver>${webdriver.opera.driver}</webdriver.opera.driver>
</systemProperties>
<includes>
<include>**/*WD.java</include>
</includes>
</configuration>
<executions>
<execution>
<goals>
<goal>integration-test</goal>
<goal>verify</goal>
</goals>
</execution>
</executions>
</plugin>

This will again make our properties available to our test code. Next we need to make some modifications to our WebDriverThread class. First of all we are going to add a new class variable called useRemoteWebDriver:

```java
private final DriverType defaultDriverType = FIREFOX;
private final String browser = System.getProperty("browser").toUpperCase();
private final String operatingSystem = System.getProperty("os.name").toUpperCase();
private final String systemArchitecture = System.getProperty("os.arch");
private final boolean useRemoteWebDriver = Boolean.getBoolean("remoteDriver");
```

This variable is going to read in the system property that we set in our POM and work out whether we want to use a RemoteWebDriver instance or not. Then we need to update our instantiateWebDriver method:

```java
private void instantiateWebDriver(DesiredCapabilities desiredCapabilities) throws MalformedURLException {
```
Producing the Right Feedback When Failing

System.out.println(" ");
System.out.println("Current Operating System: " + operatingSystem);
System.out.println("Current Architecture: " + systemArchitecture);
System.out.println("Current Browser Selection: " + selectedDriverType);
System.out.println(" ");

if (useRemoteWebDriver) {
    URL seleniumGridURL = new URL(System.getProperty("gridURL"));
    String desiredBrowserVersion = System.getProperty("desiredBrowserVersion");
    String desiredPlatform = System.getProperty("desiredPlatform");

    if (null != desiredPlatform && !desiredPlatform.isEmpty()) {
        desiredCapabilities.setPlatform(Platform.valueOf(desiredPlatform.toUpperCase()));
    }

    if (null != desiredBrowserVersion && !desiredBrowserVersion.isEmpty()) {
        desiredCapabilities.setVersion(desiredBrowserVersion);
    }

    webdriver = new RemoteWebDriver(seleniumGridURL, desiredCapabilities);
} else {
    webdriver = selectedDriverType.getWebDriverObject(desiredCapabilities);
}

This is where all of the hard work is done. We are using our useRemoteWebDriver object to work out whether we want to instantiate a normal WebDriver object, or a RemoteWebDriver object. If we want to instantiate a RemoteWebDriver object we start off by reading in the system properties we set in our POM. The most important bit of information is seleniumGridURL. If we don't have this, we don't know where to go to connect to the grid. We are reading in the system property and trying to generate a URL from it. If the URL is not valid an InvalidURLException will be thrown; this is fine because we won't be able to connect to a grid anyway at this point so we may as well end our test run there and then.
The other two bits of information are optional. If we supply a desiredPlatform and desiredBrowserVersion, the Selenium Grid will use an agent matching these criteria. If we don't supply this information the, Selenium Grid will just grab any free agent and run our test on it. Looking at this code, it's not instantly obvious what browser we are requesting; don't worry, it's covered. Each DesiredCapabilities object will set a browser type by default. So if we create DesiredCapabilities.firefox(), we will be asking the Selenium Grid to run our test against Firefox. This is one of the reasons we originally kept getDesiredCapabilities() separate from instantiateWebDriver().

We are now all done. The easiest way to test this is to set up a free account with a Selenium Grid provider such as Sauce Labs (https://saucelabs.com) and run your tests against them. To do that, put the following into your command line (obviously you'll need to supply your own Sauce Labs username and access key for this to work):

```
mvn clean install -Dremote=true -DseleniumGridURL=http://{username}:{accessKey}@ondemand.saucelabs.com:80/wd/hub -Dplatform=xp -Dbrowser=firefox -DbrowserVersion=33
```

You should now have a working CI system and the ability to run your tests remotely.

It's great to be able to connect to a third-party grid and see all your tests running without having to do the hard setup work; however this does give us some new challenges. When you are running your tests remotely, it's a lot harder to work out what the problem is when things go wrong, especially if they appear to work locally. We now need to find a way to make it easier to diagnose problems with our tests when we run them remotely.

A picture paints a thousand words

Even if you have made your tests totally reliable, they will fail occasionally. When this happens it is often very hard to describe what the problem is with words alone. If one of your tests fails, wouldn't it be easier to explain what went wrong if you had a picture of what was happening in the browser when the error happened? I know that, when one of my Selenium tests fails, the first thing I want to know is what was on the screen at the time of failure. If I know what was on the screen at the time of failure I will be able to diagnose the vast majority of issues without having to hunt through a stack trace for a specific line number and then looking at the associated code to try and work out what went wrong. Wouldn't it be nice if we got a screenshot showing what was on the screen every time a test failed? Let's take the project that we built in Chapter 1, Creating a Fast Feedback Loop, and extend it a bit to take a screenshot every time there is a test failure. Let's have a look at how we can implement this in TestNG.
Producing the Right Feedback When Failing

First of all we are going to create a package called listeners.

Then we are going to implement a custom listener for TestNG that will detect a test failure and then capture a screenshot for us.

```java
package com.masteringselenium.listeners;
import org.openqa.selenium.OutputType;
import org.openqa.selenium.TakesScreenshot;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.remote.Augmenter;
import org.testng.ITestResult;
import org.testng.TestListenerAdapter;
import java.io.File;
import java.io.IOException;
import static com.masteringselenium.DriverFactory.getDriver;

public class ScreenshotListener extends TestListenerAdapter {
    private boolean createFile(File screenshot) throws IOException {
        boolean fileCreated = false;
        if (screenshot.exists()) {
            fileCreated = true;
        } else {
            File parentDirectory = new
                File(screenshot.getParent());
            File screenshotDirectory = new
                File(parentDirectory.getAbsolutePath() + "/screenshots/" +
                    System.currentTimeMillis() + ".png";
            screenshotDirectory.mkdir();
            try {
                File targetFile = new
                    File(screenshotDirectory, "/" +
                    screenshot.getName());
                File sourceFile = screenshot;
                FileOutputStream fileOutputStream = new
                    FileOutputStream(targetFile);
                sourceFile.transferTo(fileOutputStream);
                fileOutputStream.close();
                fileCreated = true;
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return fileCreated;
    }

    public void onTestFailure(ITestResult tr) {
        try {
            WebDriver driver = getDriver();
            TakesScreenshot screenshot = (TakesScreenshot) driver;
            File screenshotFile = screenshot.getScreenshotAs(OutputType.FILE);
            File screenshotDirectory = new
                File(screenshotFile.getParent());
            File screenshotName = new
                File(screenshotDirectory, "/" +
                    screenshotFile.getName());
            screenshotDirectory.mkdir();
            File targetFile = new
                File(screenshotDirectory, "/" +
                    screenshotName.getName());
            File sourceFile = screenshotFile;
            FileOutputStream fileOutputStream = new
                FileOutputStream(targetFile);
            sourceFile.transferTo(fileOutputStream);
            fileOutputStream.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```
if (parentDirectory.exists() ||
    parentDirectory.mkdirs()) {
    fileCreated = screenshot.createNewFile();
}

return fileCreated;

private void writeScreenshotToFile(WebDriver driver, File screenshot) throws IOException {
    FileOutputStream screenshotStream = new
    FileOutputStream(screenshot);
    screenshotStream.write(((TakesScreenshot)
    driver).getScreenshotAs(OutputType.BYTES));
    screenshotStream.close();
}

@Override
public void onTestFailure(ITestResult failingTest) {
    try {
        WebDriver driver = getDriver();
        String screenshotDirectory =
            System.getProperty("screenshotDirectory");
        String screenshotAbsolutePath = screenshotDirectory +
            File.separator + System.currentTimeMillis() + "_" +
            failingTest.getName() + ".png";
        File screenshot = new File(screenshotAbsolutePath);
        if (createFile(screenshot)) {
            try {
                writeScreenshotToFile(driver, screenshot);
            } catch (ClassCastException weNeedToAugmentOurDriverObject) {
                writeScreenshotToFile(new
                    Augmenter().augment(driver), screenshot);
            }
            System.out.println("Written screenshot to " +
                screenshotAbsolutePath);
        } else {
            System.err.println("Unable to create " +
                screenshotAbsolutePath);
        }
    } catch (Exception ex) {
        System.err.println("Unable to capture screenshot...");
        ex.printStackTrace();
    }
}
First of all we have the rather imaginatively named `createFile` method that will try to create a file. Next we have the equally imaginatively named `writeScreenShotToFile` method that will try and write the screenshot to a file. Notice that we aren't catching any exceptions in these methods, because we will do that in the listener.

TestNG can get itself in a twist if exceptions are thrown in listeners. It will generally trap them so that your test run doesn't stop, but it doesn't fail the test when it does this. If your tests are passing but you have failures and stack traces, check to see if it's the listener at fault.

Finally we have the actual listener. The first thing that you will notice is that it has a `try-catch` wrapping the whole method. While we do want a screenshot to show us what has gone wrong, we probably don't want to kill our test run if we are unable to capture it or write a screenshot to disk for some reason. To make sure that we don't disrupt the test run we catch the error, and log it out to the console for future reference. We then carry on with what we were doing before.

You cannot cast all driver implementations in Selenium into a `TakesScreenshot` object. As a result we capture the `ClassCastException` for driver implementations that cannot be cast into a `TakesScreenshot` object and augment them instead. We don't just augment everything because a driver object that doesn't need to be augmented will throw an error if you try. It is usually `RemoteWebDriver` instances that need to be augmented. Apart from augmenting the driver object when required, the main job of this function is to generate a filename for the screenshot. We want to make sure that the filename is unique so that we don't accidentally overwrite any screenshots. To do this we use the current timestamp, and the name of the current test. We could use a randomly generated `GUID (Globally Unique Identifier)` but timestamps make it easier to track what happened at what time. Finally we want to log the absolute path to the screenshot out to console. This will make it easy to find any screenshots that have been created.

As you may have noticed in the preceding code, we are using a system property to get the directory that we save our screenshots in; we need to set this system property in our POM. We need to modify the `maven-failsafe-plugin` section so that it looks like this:

```xml
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>2.17</version>
  <configuration>
    <parallel>methods</parallel>
    <threadCount>${threads}</threadCount>
    <systemProperties>
```
<browser>${browser}</browser>
<screenshotDirectory>${project.build.directory}/screenshots</screenshotDirectory>
<remoteDriver>${remote}</remoteDriver>
<gridURL>${seleniumGridURL}</gridURL>
<desiredPlatform>${platform}</desiredPlatform>
<desiredBrowserVersion>${browserVersion}</desiredBrowserVersion>
<!--Set properties passed in by the driver binary downloader-->
<phantomjs.binary.path>${phantomjs.binary.path}</phantomjs.binary.path>
<webdriver.chrome.driver>${webdriver.chrome.driver}</webdriver.chrome.driver>
<webdriver.ie.driver>${webdriver.ie.driver}</webdriver.ie.driver>
<webdriver.opera.driver>${webdriver.opera.driver}</webdriver.opera.driver>
</systemProperties>
<includes>
  <include>**/*WD.java</include>
</includes>
</configuration>
<executions>
  <execution>
    <goals>
      <goal>integration-test</goal>
      <goal>verify</goal>
    </goals>
  </execution>
</executions>
</plugin>

We are only going to add a system property variable; we aren't going to make this a value that you can override on the command line. We have however used a Maven variable to specify the screenshot directory location. Maven has a series of predefined variables that you can use; ${project.build.directory} will provide you with the location of your target directory. Whenever Maven builds your project it will compile all of the files into a temporary directory called target, it will then run all of your tests and store the results in this directory. This directory is basically a little sandbox for Maven to play in while it's doing its stuff. By default this will be created in the folder that holds your POM file.

When performing Maven builds it is generally good practice to use the clean command:

mvn clean install
Producing the Right Feedback When Failing

The `clean` command deletes the target directory to make sure that when you build your project you don't have anything left over from the previous build that may cause problems. Generally speaking, when we run tests we are only going to be interested in the result of the current test run (any previous results should have been archived for future reference), so we are going to make sure that our screenshots are saved to this directory. To keep things clean we are generating a screenshots' subdirectory that we will store our screenshots in.

Now that our screenshot listener is ready, we just have to tell our tests to use it. This is surprisingly simple; all of our tests extend our `DriverFactory`, so we just add a `@Listeners` annotation to it.

```java
import com.masteringselenium.listeners.ScreenshotListener;
import org.testng.annotations.Listeners;

@Listeners(ScreenshotListener.class)
public class DriverFactory
```

From this point onwards if any of our tests fail a screenshot will automatically be taken.

Why don't you give it a go? Try changing your test to make it fail so that screenshots are generated. Try putting some Windows or OS dialogs in front of your browser while the tests are running and taking screenshots. Does this affect what you see on the screen?

Screenshots are a very useful aid when it comes to diagnosing problems with your tests, but sometimes things go wrong on a page that looks completely normal. How do we go about diagnosing these sorts of problems?

**Don't be afraid of the big bad stack trace**

It's surprising how many people are intimidated by stack traces. A reaction that I regularly see when a stack trace appears on screen is panic!

"Oh my God, something has gone wrong! There are hundreds of lines of text talking about code I don't recognize and I can't take it all in; what do I do?"
The first thing to do is to relax; stack traces have a lot of information but they are actually really friendly and helpful things. Let’s modify our project to produce a stack trace and work through it. We are going to make a small change to the `getDriver()` method in `DriverFactory` to force it to always return a null, as follows:

```java
public static WebDriver getDriver() {
    return null;
}
```

This is going to make sure that we never return a driver object, something that we would expect to cause errors. Let’s run our tests again, but make sure that Maven displays a stack trace by using the `-e` switch:

```
mvn clean install -e
```

This time you should see a couple of stack traces output to the terminal; the first one should look like this:

```
class BasicTestsMD {  
    void googleCheeseExample() throws Exception {  
        String googleCheeseExample = driver.navigate().to("http://www.google.com/" + "cheese").getTitle();  
        System.out.println("googleCheeseExample = " + googleCheeseExample);  
    }  
    void setup() {  
        BasicTestsMD driver = new BasicTestsMD();  
        driver.googleCheeseExample();  
    }  
}
```

A stack trace shown when the build fails
Producing the Right Feedback When Failing

It's not too big so let's have a look at it in more detail. The first line tells you the root cause of our problem: we have got a NullPointerException. You have probably seen these before. Our code is complaining because it was expecting to have some sort of object at some point and we didn't give it one. Next we have a series of lines of text that tell us where in the application the problem occurred.

We have quite a few lines of code that are referred to in this stack trace, most of them unfamiliar as we didn't write them. Let's start at the bottom and work our way up. We first of all have the line of code that was running when our test failed; this is Thread. java line 745. This thread is using a run method (on ThreadPoolExecutor. java line 617) that is using a runWorker method (on ThreadPoolExecutor. java line 1142), and this carries on up the stack trace. What we are seeing is a hierarchy of code with all the various methods that are being used. We are also being told which line of code in that method caused a problem.

We are specifically interested in the lines that relate to the code that we have written—in this case, the second and third lines of the stack trace. You can see that it is giving us two very useful bits of information; it's telling us where in our code the problem has occurred and what sort of problem it is. If we have a look at our code, we can see what it was trying to do when the failure occurred so that we can try and work out what the problem is. Let's start with the second line; first of all it tells us which method is causing the problem. In this case it is com.masteringseleium.DriverFactory. clearCookies. It then tells us which line of this method is causing us a problem—in this case DriverFactory. java line 35. This is where our clearCookies() method tries to get a WebDriver instance from our WebDriverThread class, and then uses it to try and clear all the cookies.

Now, if you remember, we modified getDriver() to return a null instead of a valid driver object. This matches up with the first bit of information in our stack trace (the NullPointerException). Obviously we cannot call .manage(). deleteAllCookies() on a null, hence the null pointer error.

So why didn't it fail in WebDriverThread; after all, that's where the problem is? Passing a null around is quite valid. It's trying to do something with the null that causes the problem. This is why it also didn't fail on line 30 of DriverFactory. The getDriver() method just passes on what was returned from WebDriverThread, it doesn't actually try to do anything with it. The first time that we tried to do anything with the null is when it failed, which was at line 35 of the DriverFactory class.
When it is explained it can seem quite obvious, but it takes a while to get used to reading stack traces. The important thing to remember with stack traces is to read them in full. Don't be scared of them, or skim through them and guess at the problem. Stack traces provide a lot of useful information to help you diagnose problems. They may not take you directly to the problematic bit of code but they give you a great place to start.

Try causing some more errors in your code and then run your tests again. See if you can work your way back to the problem you put in your code by reading the stack trace.

**Summary**

After reading through this chapter you should:

- Think of automated tests as living documentation rather than automatic regression
- Have a good understanding of why reliability matters
- Have an understanding of Continuous Integration, Continuous Delivery, and Continuous Deployment
- Be able to configure your tests to run in a Maven profile
- Be able to set up a test build in a Continuous Integration Server
- Be able to connect to a Selenium Grid (both local and third party)
- Be able to take screenshots of test failures
- Be able to read a stack trace and work out what the causes of your test failures are.

In the next chapter, we are going to have a look at exceptions generated by Selenium. We will work through various exceptions that you may see and what they mean.
Where to buy this book
You can buy Mastering Selenium WebDriver from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.