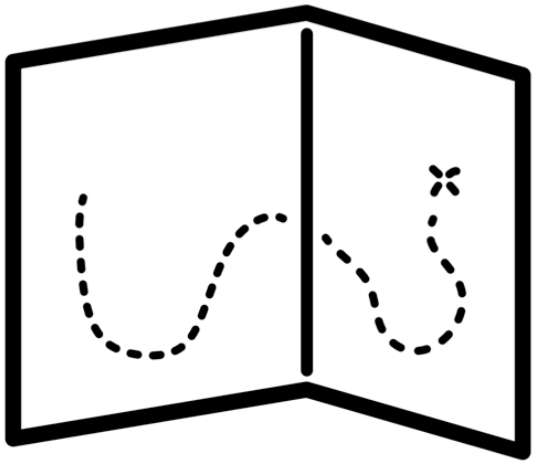
How to use Selenium, successfully



The Selenium Guidebook

C# Edition

by Dave Haeffner

Version 4.0.0

Preface

This book is not a full and comprehensive treatise that outlines every possible permutation of Selenium (the open-source software test automation tool for web applications). There are other books that already do this. My goal, instead, is to teach you the necessary pieces to use Selenium successfully for your circumstance.

What you have before you is a distilled and actionable guide culled from my consulting practice and full time positions held doing Quality Assurance over the past ten years.

My goal in writing this is to provide you with the materials I wish existed when I was starting out with automated acceptance testing. I hope it serves you well.

What This Book Will Cover

This book focuses on the latest stable version of Selenium 4 (a.k.a. Selenium WebDriver) and its use to test desktop browsers.

Record and Playback tools like Selenium IDE are a great option nowadays (no, really). But they will not be covered in this book. Instead, an approach of writing well factored tests, in code, is the focus of this book.

Who This Book Is For

This book is for anyone who wants to take automated acceptance testing seriously and isn't afraid to get their hands a little dirty.

That is to say, this book is for anyone who wants to use computers for what they're good at, and free you up (and potentially the people on your team) to do what they are inherently good at (which does not include repetitive, mundane testing tasks). And don't worry if you're new to programming. I'll cover the essentials so you'll have a good place to start from.

About The Examples In This Book

The examples in this book are written in C#, but the strategies and patterns used are applicable regardless of your technical stack.

The tests in this book are written to exercise functionality from an open-source project I created and maintain called the-internet -- available here on GitHub and viewable here on Heroku.

The test examples are written to run against NUnit with NuGet managing the third-party dependencies.

All of the code examples from the book are available in an accompanying zip file. It contains folders for each chapter where code was written or altered. Chapters with multiple parts will have multiple sub-folders (e.g., code examples referenced Chapter 9, Part 2 can be found in `09/02/` in the zip file).

How To Read This Book

Chapters 1 through 5 focus on the things you need to consider when it comes to test strategy, programming language selection, and good test design. Chapter 6 is where we first start to code. From there, the examples build upon each other through chapter 17.

Chapter 18 paints a picture of the Selenium landscape so you're better able to find information on your own.

Feedback

If you find an error in the book (e.g., grammar issue, code issue, etc.) or have questions/feedback -- please feel free to e-mail me at hello@seleniumguidebook.com.

If you submit something and I end up using it in a future version of the book I'll give you a shout-out in the Acknowledgements.

Table of Contents

Chapter 1
Selenium In A Nutshell

Selenium is a software robot sent from the future to help us test web applications. But keep in mind that it's not one of those fancy shape-shifting robots than can run really fast. It's more like one of those really strong robots that's not very fast and is best suited for accomplishing a certain objective.

That is to say -- Selenium is really good at a specific set of things. If you know what those are and stick to them then you can easily write reliable, scalable, and maintainable tests that you and your team can trust.

What Selenium Is and Is Not Good At

Selenium is built to automate browsers, specifically human interaction with them. Things like navigating to pages, clicking on elements, typing text into input fields, etc.

It's less ideal for checking lower-level functionality like HTTP status codes or HTTP headers. While you can use Selenium this way, it requires additional setup of a third-party tool (e.g., a proxy server like BrowserMob Proxy), and it is a slippery slope since there are numerous edge cases to consider at this level.

Selenium Highlights

Selenium works on every major browser, in every major programming language, and on every major operating system. Each language binding and browser are actively being developed to stay current. Yes, even Internet Explorer (thanks to Jim Evans!). And WebDriver (the thing which drivers Selenium) has become a W3C specification.

Selenium can be run on your local computer, on a remote server, on a set of servers (with Selenium Grid), or on a third-party cloud provider (like Sauce Labs). As your test suite grows, your test runs will take longer to complete. To speed them up you will want to run them in parallel, which is where the benefit of having your own servers or using a cloud provider comes in -- that, and the ability to have numerous browser and operating system combinations to run your tests on.

One of the guiding philosophies of Selenium is to be able to write your tests once and run them across multiple browsers. While this is a rosy proposition, it's not entirely accurate. There are some gotchas to watch out for when you get into it. But don't worry, We'll step through these in detail throughout the book.

Chapter 2
Defining A Test Strategy

A great way to increase your chances of success with automated web testing is to first map out a testing strategy. The best way to do it is to answer these four questions:

1. How does your business make money?
2. What features in your application are being used?
3. What browsers are your users using?
4. What things have broken in the application before?

NOTE: For organizations that don't deal directly in dollars and cents (e.g., non-profits, federal agencies, etc.) you should focus on how the application generates value for the end user and the organization.

After answering these questions you will have an understanding of the functionality and browsers that matter for the application you're testing. This will help you focus your efforts on the things that matter most.

This strategy works best for applications with existing functionality and does not speak directly to testing new functionality that's being actively developed. That's not to say that the two couldn't co-exist. It largely depends on your available resources and pace of development. But in order to reach high quality at speed, you first have to go slow.

What To Do With The Answers

After answering these questions you should end up with a prioritized punch list (a.k.a., backlog) of test automation tasks that you can work off of and track progress against.

Question 1: Money/Value

Every company's application makes money (or generates value) through core functionality that is defined by a series of increasingly-valuable user interactions -- a.k.a. a "funnel". Your answers to this question will help you determine what your funnel is.

These items will be your highest priority for automation. Start an automation backlog to keep track of them.

Question 2: Usage Data

Odds are your application offers a robust set of functionality well beyond your funnel. Your answers to this question will help highlight what it is. And if you're basing these answers on usage data (e.g., Google Analytics, etc.), then it will be broken down from highly used to lightly used.

Tack these items onto your automation backlog (below the items from question #1) based on their frequency of use.

Question 3: Browsers

Now that you know what functionality is business critical and widely adopted by your users, you need to determine what browsers to focus your automated web testing efforts on. Your usage data will tell you this as well. It will help you determine which browsers you can reasonably avoid testing in (e.g., based on non-existent or low usage numbers).

Note the top 2 (or 3 depending on your numbers), but focus on the top 1 for now. This is the browser you will start using for automation.

Question 4: Risky Bits

To round out the strategy it is also best to think about what things have broken in the application before. To answer this question it's best to check your defect/bug tracker (if you have one) and to ask your team. What you end up with may read like a laundry list of browser specific issues or functionality that has been flaky or forgotten about in the past. But it's all useful information.

Be sure to check this list against your automation backlog. If something's not there, add it to the bottom of the backlog. If it is there, make a note in the backlog item that it has been an issue in the past.

If the issue has happened numerous times and has the potential to occur again, move the item up in the backlog. And if issues keep cropping up that are related to a specific browser, compare this browser to your short list of browsers from question #3. If it's a browser that's not in your list but it's still important (e.g., a small pocket of influential users), track it on the backlog, but put it at the bottom.

Now You Are Ready

Having answered these questions, you should now have a prioritized backlog of critical business functionality, a short list of browsers to focus on, and an understanding of the risky parts of your application to watch out for. With it, you're on the right track -- focusing on things that matter for your business and its users.

Chapter 3
Picking A Language

In order to work well with Selenium you need to choose a programming language to write your automated acceptance tests in. Conventional wisdom will tell you to choose the same language that the application is written in. That way if you get stuck you can ask the developers on your team for help. But if you're not proficient in this language (or new to programming), then your progress will be slow and you'll likely end up asking for more of a developer's help than they have time for -- hindering your automation efforts and setting you up for failure.

A great way to determine which language to go with is to answer one simple question: Who will own the automated tests?

The answer to this, and the discussion that unfolds from it, will help you more effectively choose a programming language.

What To Do With The Answer

If you're a tester (or a team of testers) and you will be building and maintaining the test automation, then it's best to consider what languages you (and your team) already have experience with, or are interested in working with. Since your Selenium tests don't need to be written in the same language as the application you are testing, have a discussion with your team to see what interests them and go from there.

If you're a developer who is working on the application and just looking to add automated acceptance testing to the mix, then it makes sense to continue with the same language.

Some Additional Food For Thought

As your suite of tests grows you will find the need to add functionality to make things easier to use, run faster, offer better reporting, etc. When this happens you will effectively be doing two jobs; no longer just writing automated tests but also building and maintaining a framework (a.k.a. a test harness).

As you're considering which language to go with consider what open source frameworks already exist for the languages you are considering. Going with one can save you a lot of time and give you a host of functionality out of the box that you would otherwise have to create yourself -- and they're FREE.

You can find a list of open source Selenium WebDriver frameworks and their respective languages here.

Outro

Choosing a programming language for automated testing is not a decision that should be taken lightly. If you're just starting out then considering and discussing these things will help position you for long term success.

With all that being said, the rest of this book will show you how to build your own test harness (in addition to writing well factored tests) in C# with NUnit.

Chapter 4
A Programming Primer

This section will prime you with just enough programming concepts (and how they pertain to Selenium) so you have some working knowledge and a vocabulary. This will help you more effectively comprehend what you will see throughout this book (and in your work afterwards).

Don't get too hung up on the details though. If something doesn't make sense it should once we dig into the full examples in the following chapters. Consider this more of a glossary than a code school.

Installation

Visual Studio Code is a phenomenal Integrated Development Environment (IDE) for the Microsoft Development Ecosystem. It has everything you need.

Once installed, you'll want to install a couple of extension from the built-in marketplace in the IDE: `C#` and `.NET Core Test Explorer`.

Installing Third-Party Packages

All of the packages (a.k.a. libraries) used in this book are installed with NuGet. NuGet is a package manager for the Microsoft Development Ecosystem where developers can publish packages they've created and consumers (like us) can download and install them.

Getting set up is easy:

1. Install the .NET SDK for your operating system -- nuget comes bundled with it
2. Search NuGet for the package you want to install (like Selenium)
3. Open the page for the package you want to install
4. Copy the install command from the page
5. Paste it into the command line and run it

The primary packages we will be using throughout the book are:

- `Selenium.WebDriver`
- `Selenium.Support`
- `NUnit`
- `NUnit3TestAdapter`

To save time, you can create a new project using an NUnit template with the following command:

```
dotnet new nunit -n NUnit-Tests --framework netcoreapp 2.0
```

Programming Concepts In A Nutshell

Programming can be a deep and intimidating rabbit hole if you're new to it. But don't worry. When it comes to automated browser testing there is only a small subset of programming that we really need to know in order to get started. Granted, the more you know, the better off you'll be. But you don't need to know a whole lot in order to be effective right now.

Of all the programming concepts out there, here are the ones you'll need to pay attention right now:

- Object Structures (Variables, Methods, and Classes)
- Access Modifiers (public, protected, private, etc.)
- Types of Objects (Strings, Integers, Booleans, etc.)
- Actions (Assertions and Conditionals)
- Attributes
- Inheritance

Let's step through each and how they pertain to testing with Selenium.

Object Structures

Variables

Variables are objects where you can store and retrieve values. They are created and referenced by a name that:

- is not case sensitive
- is not a keyword or reserved word in C#
- starts with a letter

Since variable names are not case sensitive there are a lot of varied opinions out about how to write them (e.g., `camelCase`, `PascalCase`, `_underBarCase`). You can read various threads on the topic here, here, and here. There's really no one specific code style. It's really just a matter of opinion. If you're working on a team that has their own preference, go with it. If not, then start with something simple that works for you.

You can store a value in a variable by using an equals sign (e.g., `=`). You can either specify the type of the object or let C# figure out what the type is through type inference.

```
string ExampleVariable1 = "string value";
Console.WriteLine(ExampleVariable1);
// outputs: string value

var ExampleVariable2 = 42;
Console.WriteLine(ExampleVariable2);
// outputs: 42
```

NOTE: In the code snippet above we're using `Console.WriteLine();` to output a message. This is a common command that is useful for generating output to the console (a.k.a. terminal window).

In Selenium, a common example of variable usage is storing an element (or a value from an element) in the beginning of a test to be referenced later on. For example, getting a page's title.

```
string PageTitle = Driver.Title();
```

NOTE: `Driver` is the variable we will use to interact with Selenium throughout the book. More on that later.

Methods

Throughout our tests we'll want to group common actions together for easy reuse. We do this by placing them into methods. We define a method within a class (more on those next) by specifying a modifier (which we'll cover in `Access Modifiers`), a return type, and a name.

A return type is used to specify what type of an object you want to return after the method is executed (more on Object Types in a bit). If you don't want to return anything, specify the return type as `void`.

Method names typically start with a capital letter and are PascalCase (e.g., each word is capitalized). The biggest difference between method and variable names is that method names tend to be a verb (since they denote some kind of an action to be performed). Also, the contents of the method are wrapped in opening and closing brackets (e.g., `{}`).

```
public void SayHello() {
  // your code goes here
}
```

Additionally, you can make a method accept an argument when calling it. This is done with a parameter.

```
public void SayHello(string message) {
  Console.WriteLine(message);
}
```

We'll see methods put to use in numerous places in our test code. First and foremost each of our tests will use them when setting up and tearing down instances of Selenium.

```
public void SetUp() {
    Driver = new FirefoxDriver();
}

public void TearDown() {
    Driver.Quit();
}
```

Classes

Classes are a useful way to store the state and behavior of something complex for reuse. They are where variables and methods live. And they're defined with the word `class` followed by the name you wish to give it. Class names:

- should start with a capital letter
- should be PascalCase for multiple words (e.g., class ExampleClass)
- should be descriptive (e.g., a noun or noun phrase)

To use a class you first have to define it. You then create an instance of it (a.k.a. instantiation). Once you have a class instance you can access the methods within it to trigger an action.

The most common example of this in Selenium is when you want to represent a page of the application you're testing (a.k.a. a page object). In the page object class you store the elements from the page you want to use (a.k.a. state) in variables and the actions you can perform with those elements (a.k.a. behavior) in methods.

```
// 1. Code in a page object class
namespace PageObjects
{
    class LoginPage
    {
        IWebDriver Driver;
        By UsernameInput = By.Id("username");
        By PasswordInput = By.Id("password");
        By SubmitButton = By.CssSelector("button");
        By SuccessMessage = By.CssSelector(".flash.success");

        public void With(string username, string password)
// ...

// 2. Code in a test that uses the page object class
LoginPage Login = new LoginPage
Login.With("username", "password");
```

Access Modifiers

When specifying an object (e.g., a variable, method, or class) you can apply a modifier. This modifier denotes what else can access the object. This is also known as "scope".

For classes you can apply `public` or nothing. `public` makes the class visible to all other classes (including those outside of the package or assembly). Specifying nothing sets the class scope to `internal` which makes the class available to just other classes in the same package.

For members of a class (e.g., variables and methods) you can use `public`, `protected`, `private`, and `internal`. There's also `protected internal`, but you're not likely to use it.

- `public` and `internal` behave just like they do with classes
- `private` makes it so the member can only be accessed from within the class it was specified
- `protected` makes it so the member can only be accessed from within the class it was specified and from a derivative class (more on inheritance later)

The best thing to do is to follow a "need-to-know" principle for your class members. Start with a `private` scope and only elevate it when appropriate (e.g., from `private` to `protected`, from `protected` to `public`, etc.).

In our Selenium tests, we'll end up with various modifiers for our objects.

```
// When creating a test method it needs to be public for NUnit to use it
[Test]
    public void ValidAccount()
    {
```

Types of Objects

Objects can be of various types, and when declaring a method we need to specify what type it will return. If it returns nothing, we specify `void`. But if it returns something (e.g., a boolean) then we need to specify that.

The two most common types of objects we'll see initially in our tests are strings and booleans. Strings are a series of alpha-numeric characters stored in double-quotes. Booleans are a `true` or `false` value.

A common example of specifying a return type in our test code is when we use Selenium to see if something is displayed on a page.

```
public bool SuccessMessagePresent()
    {
        return Driver.FindElement(SuccessMessage).Displayed;
    }
```

After specifying the return type when declaring the method (e.g., `public bool`) we use the `return` keyword in the method body to return the final value.

Actions

A benefit of booleans is that we can use them to perform an assertion.

Assertions

An assertion is a function that allows us to test assumptions about our application and notify us if there are any unexpected differences.

A common example of this is testing the login functionality of an application. After logging in we could check to see if a specific element is displayed on the page (e.g., a sign out button, a success notification, etc.). We could use this element to perform a display check which would return a boolean value. We would use this value to assert that it is what we expect (e.g., `true`). If it is `true`, then the test passes. If it's not `true`, then the test fails.

```
// A method that looks to see if a success message is displayed after logging in
    public bool SuccessMessagePresent() {
        return Driver.FindElement(SuccessMessage).Displayed;
    }

// An assertion in our test to see if the value returned is the value expected
    Assert.That(Login.SuccessMessagePresent());
```

Conditionals

In addition to assertions, we can also leverage booleans in conditionals. Conditionals (a.k.a. control flow statements) are a way to break up the flow of code so that only certain chunks of it are executed based on predefined criteria. The most common control flow statements we'll use are `if`, `else`, `switch` / `case`.

The most common use of this will be in how we configure Selenium to run different browsers.

```
switch (BrowserName.ToLower())
    {
        case "firefox":
            Driver = new FirefoxDriver();
            break;
        case "chrome":
            Driver = new ChromeDriver(VendorDirectory);
            break;
    }
```

Attributes

Attributes are a form of metadata. They are used by various libraries to enable additional functionality.

The most common use of attributes in our tests is when specifying different types of methods (e.g., a setup method, a teardown method, a test method, etc.) to be run at different times in our test execution.

```
// Methods in a test file

[SetUp]
    // this method will run before each test
    public void SetUp()
    {
        Driver = new FirefoxDriver();
    }

[TearDown]
    // this method will run after each test
    public void TearDown()
    {
        Driver.Quit();
    }

[Test]
    // This method is a test
    public void ValidAccount()
    {
```

Inheritance

Classes have the ability to connect to one-another through parent/child inheritance. By having a single parent class we can store common actions in methods that can be readily available to all child classes.

Inheritance is done when declaring a child class by:

- using the `:` symbol
- providing the name of the parent class

```
class Parent {
      static string HairColor = "brown";
    }

    Child : Parent {
      public void Child() {
        Console.WriteLine(HairColor);
      }
    }

// Creating an instance of the Child class outputs "brown"
```

We'll see this a couple of times in our test code when we start writing easier to maintain code

(e.g., grouping common Selenium actions into methods within a parent class for our page objects, or storing central setup and teardown functionality into a base test). More on this in Chapters 9 and 11.

Additional Resources

Here are some additional resources that can help you continue your C# learning journey.

- Getting Started with C# on MSDN
- C# Tutorial on tutorialspoint
- C# in a Nutshell: The Definitive Reference
- Learn C# Programming Udemy course
- Lynda.com C# Training and Tutorials
- Pluralsight C# Course

Chapter 5
Anatomy Of A Good Acceptance Test

In order to write automated web tests that are easy to maintain, perform well, and are ultimately resilient there are some simple guidelines to follow:

- Write atomic and autonomous tests
- Group like tests together in small batches
- Be descriptive
- Use a Test Runner
- Store tests in a Version Control System

Atomic & Autonomous Tests

Each test needs to be concise (e.g., testing a single feature rather than multiple features) and be capable of being run independently (e.g., sets up its own data rather than relying on a previous test to do it). Doing this may require a mental shift, discipline, and more up front effort. But it will make a dramatic impact on the quality, effectiveness, and maintainability of your tests. Especially when you get into parallel test execution.

Grouping Tests

As your test suite grows you will have numerous test files. Each one containing a grouping of tests that exercise similar functionality. These test files should be grouped together in a simple directory structure as the groupings become obvious. If you want to create a test run of disparate tests, this is something that is easy to handle when using a Test Runner (covered briefly below, and in-depth in Chapter 16).

Being Descriptive

A test file should have a high level name that denotes what the tests within it are exercising. Each test should have an informative name (even if it is a bit verbose). Also, each test (or grouping of tests) should include some helpful metadata (e.g., tags, or categories) which can provide additional information about the test as well as enable flexible test execution (more on that in Chapter 16). This way all or part of your test suite can be run, and the results will be informative thanks to helpful naming.

This enables developers to run a subset of tests to exercise functionality they just modified while also enabling you to intelligently wire your test suite up to a Continuous Integration (CI) server for fast and dynamic feedback (more on CI servers in Chapter 17).

Test Runners

At the heart of every test harness is some kind of a test runner that does a lot of the heavy lifting (e.g., test execution, centralized configuration, test output, etc.). Rather than reinvent the wheel you can use one of the many test runners that exist today. With it you can bolt on third party libraries to extend its functionality if there's something missing.

Version Control

In order to effectively collaborate with other testers and developers on your team, your test code must live in a version control system. Look to see what your development team uses and add your code to it. Otherwise, set up one of the following:

- Git
- Mercurial
- Subversion

Keep in mind that your test code can live in a separate repository from the code of the application you're testing. Combining them may be advantageous but if all you're doing is writing and running tests against web endpoints (which is a majority of what your Selenium tests will be doing) then leaving your test code in a separate repository is a fine way to go.

Chapter 6
Writing Your First Test

Fundamentally, Selenium works with two pieces of information -- the element on a page you want to use and what you want to do with it. This one-two punch will be repeated over and over until you achieve the outcome you want in your application -- at which point you will perform an assertion to confirm that the result is what you intended.

Let's take logging in to a website as an example. With Selenium you would:

1. Visit the login page of a site
2. Find the login form's username field and input the username
3. Find the login form's password field and input the password
4. Find the submit button and click it

Selenium is able to find and interact with elements on a page by way of various locator strategies. The list includes (sorted alphabetically):

- Class
- CSS Selector
- ID
- Link Text
- Name
- Partial Link Text
- Tag Name
- XPath

While each serves a purpose, you only need to know a few to start writing effective tests.

How To Find Locators

The simplest way to find locators is to inspect the elements on a page. The best way to do this is from within your web browser. Fortunately, popular browsers come pre-loaded with development tools that make this simple to accomplish.

When viewing the page, right-click on the element you want to interact with and click Inspect Element. This will bring up a small window with all of the markup for the page but zoomed into your highlighted selection. From here you can see if there are unique or descriptive attributes you can work with.

How To Find Quality Elements

You want to find an element that is unique, descriptive, and unlikely to change.

Ripe candidates for this are `id` and `class` attributes. Whereas text (e.g., the text of a link) is less ideal since it is more apt to change. This may not hold true for when you make assertions, but it's a good goal to strive for.

If the elements you are attempting to work with don't have unique `id` or `class` attributes directly on them, look at the element that houses them (a.k.a. the parent element). Oftentimes the parent element has a unique element that you can use to start with and walk down to the child element you want to use.

When you can't find any unique elements have a conversation with your development team letting them know what you are trying to accomplish. It's typically a trivial thing for them to add helpful semantic markup to a page to make it more testable. This is especially true when they know the use case you're trying to automate. The alternative can be a lengthy and painful process which will probably yield working test code but it will be brittle and hard to maintain.

Once you've identified the target elements for your test, you need to craft a locator using one Selenium's strategies.

An Example

Part 1: Find The Elements And Write The Test

Here's the markup for a standard login form (pulled from [the login example on the-internet]).

```
<form name="login" id="login" action="/authenticate" method="post">
   <div class="row">
    <div class="large-6 small-12 columns">
      <label for="username">Username</label>
      <input type="text" name="username" id="username">
    </div>
  </div>
  <div class="row">
    <div class="large-6 small-12 columns">
      <label for="password">Password</label>
      <input type="password" name="password" id="password">
    </div>
  </div>
    <button class="radius" type="submit"><i class="icon-2x icon-signin"> Login
</i></button>
</form>
```

Note the unique elements on the form. The username input field has a unique `id`, as does the password input field. The submit button doesn't, but it's the only button on the page so we can easily find it and click it.

Let's put these elements to use in our first test. First we'll need to create a new folder called `Tests` in the root of our project directory. In it we'll create a new test file `LoginTest.cs`. We'll also need to create a `Vendor` directory for third-party files and download `geckodriver` into it. Grab the latest release for your operating system from here and unpack its contents into the `Vendor` directory.

When we're done our directory structure should look like this.

```
Tests
   LoginTest.cs
Vendor
    geckodriver
```

NOTE: Other items in the directory have been omitted for brevity (and will continue to be omitted for the remainder of the book). See the accompanying zip file for full access to the files and directories from this and future chapters.

Here is the code we will add to the test file for our Selenium commands, locators, etc.

```
//filename: Tests/LoginTest.cs
using System;
using NUnit.Framework;
using OpenQA.Selenium;
using OpenQA.Selenium.Firefox;

namespace Tests
{
[TestFixture]
    class LoginTest : BaseTest
    {
        private static string VendorDirectory = System.IO.Directory.GetParent(
            System.AppContext.BaseDirectory).
            Parent.Parent.Parent.FullName
            + @"/vendor";
        protected IWebDriver Driver;

[SetUp]
        public new void SetUp()
        {
            var Service = FirefoxDriverService.CreateDefaultService(VendorDirectory);
            Driver = new FirefoxDriver(Service);
        }

[Test]
        public void ValidAccount()
        {
            Driver.FindElement(By.Id("username")).SendKeys("tomsmith");
            Driver.FindElement(By.Id("password")).SendKeys("SuperSecretPassword!");
            Driver.FindElement(By.CssSelector("button")).Click();
        }
    }
}
```

After including the requisite classes for NUnit and Selenium we declare a class (e.g., `public class LoginTest` and add an attribute to it that denotes that this is a test class (e.g., `[TestFixtures]`). We then declare two field variables to store and reference the vendor directory (e.g., `private static string VendorDirectory`) and an instance of Selenium WebDriver (e.g., `IWebDriver Driver;`).

Next we add setup and teardown methods with the attributes `[SetUp]` and `[TearDown]`. In them we're creating an instance of Selenium (storing it in `Driver`) and closing it (e.g., `Driver.Quit();`). Because of the `[SetUp]` attribute, the `public void SetUp()` method will load before our test and the `[TearDown]` attribute will make the `public void TearDown()` method load after the test. This abstraction enables us to write our test with a focus on the behavior we want to exercise in the browser, rather than clutter it up with setup and teardown details.

In order for the instantiation of Selenium to work with Firefox, we need to specify the path to the `geckodriver` file we downloaded into the `Vendor` directory. We did this when declaring the `VendorDirectory` variable -- by finding the full path to the directory by looking up the base directory of this project, traversing up to its top-level, tacking on `/Vendor` to it, and storing its result. Now we put it to use by creating a `FirefoxDriverService` object (which accepts the `VendorDirectory` variable) and pass it into our call to create a new instance of `FirefoxDriver`.

Our test is a method as well (`public void ValidAccount()`). NUnit knows this is a test because of the `[Test]` attribute. In this test we're visiting the login page by its URL (with `Driver.Navigate().GoToUrl()`), finding the input fields by their ID (with `Driver.FindElement(By.Id())`), inputting text into them (with `.SendKeys();`), and submitting the form by clicking the submit button (e.g., `By.CssSelector("button")).Click();`).

If we save this and run it (by clicking `Test`, `Run`, `All Tests` or `CTRL` + `R`, `A`), it will run and pass. But there's one thing missing -- an assertion. In order to find an element to write an assertion against we need to see what the markup of the page is after submitting the login form.

Part 2: Figure Out What To Assert

Here is the markup that renders on the page after logging in.

```
<div class="row">
  <div id="flash-messages" class="large-12 columns">
    <div data-alert="" id="flash" class="flash success">
      You logged into a secure area!
      <a href="#" class="close">x</a>
    </div>
  </div>
</div>

<div id="content" class="large-12 columns">
  <div class="example">
    <h2><i class="icon-lock"></i> Secure Area</h2>
    <h4 class="subheader">Welcome to the Secure Area. When you are done click logout
below.</h4>
    <a class="button secondary radius" href="/logout"><i class="icon-2x icon-signout">
Logout</i></a>
  </div>
</div>
```

There are a couple of elements we can use for our assertion in this markup. There's the flash message class (most appealing), the logout button (appealing), or the copy from either the `h2` or the flash message (least appealing).

Since the flash message class name is descriptive, denotes a successful login, and is less likely to

change than the copy, let's go with that.

```
class="flash success"
```

When we try to access an element like this (e.g., with a multi-worded class) we will need to use a CSS selector or an XPath.

NOTE: Both CSS selectors and XPath work well, but the examples throughout this book will focus on how to use CSS selectors.

A Quick Primer on CSS Selectors

In web design CSS (Cascading Style Sheets) are used to apply styles to the markup (HTML) on a page. CSS is able to do this by declaring which bits of the markup it wants to alter through the use of selectors. Selenium operates in a similar manner but instead of changing the style of elements, it interacts with them by clicking, getting values, typing, sending keys, etc.

CSS selectors are a pretty straightforward and handy way to write locators, especially for hard to reach elements.

For right now, here's what you need to know. In CSS, class names start with a dot (`.`). For classes with multiple words, put a dot in front of each word, and remove the space between them (e.g., `.flash.success` for `class='flash success'`).

For a good resource on CSS Selectors I encourage you to check out Sauce Labs' write up on them.

Part 3: Write The Assertion And Verify It

Now that we have our locator, let's add an assertion that uses it.

```
//filename: tests/LoginTest.cs
// ...
[Test]
        public void ValidAccount()
        {
            Driver.Navigate().GoToUrl("http://the-internet.herokuapp.com/login");
            Driver.FindElement(By.Id("username")).SendKeys("tomsmith");
            Driver.FindElement(By.Id("password")).SendKeys("SuperSecretPassword!");
            Driver.FindElement(By.CssSelector("button")).Click();
            Assert.That(Driver.FindElement(By.CssSelector(".flash.success")).Displayed
);
        }
    }
}
```

With `Assert.That` we are checking for a `true` Boolean response. If one is not received the test will fail. With Selenium we are seeing if the success message element is displayed on the page (with `.Displayed`). This Selenium command returns a Boolean. So if the element is rendered on the page and is visible (e.g., not hidden or covered up by an overlay), `true` will be returned, and our test will pass.

But we're not done yet. Unfortunately if we run this it would fail due to a documented bug in `geckodriver`. Clicking the submit button triggers a new page to load and under normal circumstances Selenium would automatically wait for this new page to initially load before looking for the `.flash.success` element. As of right now this automatic wait is not observed which will cause Selenium to inadvertently look for an element that hasn't appeared yet. To account for this, we'll need to add a brief delay into our test.

```
//filename: tests/LoginTest.cs
// ...
[Test]
        public void ValidAccount()
        {
            Driver.Navigate().GoToUrl("http://the-internet.herokuapp.com/login");
            Driver.FindElement(By.Id("username")).SendKeys("tomsmith");
            Driver.FindElement(By.Id("password")).SendKeys("SuperSecretPassword!");
            Driver.FindElement(By.CssSelector("button")).Click();
            System.Threading.Thread.Sleep(1000);
            Assert.That(Driver.FindElement(By.CssSelector(".flash.success")).Displayed
);
        }
    }
}
```

NOTE: Adding a hard-coded sleep (even for just 1 second -- a.k.a. `1000` milliseconds) is NOT a recommended practice. There are MUCH better ways to make your test wait. They'll be covered in Chapter 10 where we'll also revisit this code and improve upon it to remove this hard-coded sleep.

When we save this and run it it will run and pass just like before, but now there is an assertion which will catch a failure if something is amiss.

Just To Make Sure

Just to make certain that this test is doing what we think it should, let's change the locator in the assertion to attempt to force a failure and run it again. A simple fudging of the locator will suffice.

```
Assert.That(Driver.FindElement(By.CssSelector(".flash.successasdf")).Displayed);
```

If it fails then we can feel reasonably confident that the test is doing what we expect and we can

change the assertion back to normal before committing our code.

This trick will save you more trouble that you know. Practice it often.

Chapter 7

Verifying Your Locators

If you're fortunate enough to be working with unique IDs and Classes, then you're usually all set. But when you have to handle more complex actions like traversing a page, or you need to run down odd test behavior, it can be a real challenge to verify that you have the right locators to accomplish what you want.

Instead of the painful and tedious process of trying out various locators in your tests until you get what you're looking for, try verifying them in the browser instead.

A Solution

Built into every major browser is the ability to verify locators from the JavaScript Console.

Simply open the developer tools in your browser and navigate to the JavaScript Console (e.g., right-click on an element, select `Inspect Element`, and click into the `Console` tab). From here it's a simple matter of specifying the CSS selector you want to look up by the `$$('')` command (e.g., `$$('#username')`) and hovering your mouse over what is returned in the console. The element that was found will be highlighted in the viewport.

An Example

Let's try to identify the locators necessary to traverse a few levels into a large set of nested divs.

```
<!-- a snippet from http://the-internet.herokuapp.com/large -->

<div id='siblings'>
  <div id='sibling-1.1'>1.1
  <div id='sibling-1.2'>1.2</div>
  <div id='sibling-1.3'>1.3</div>
    <div id='sibling-2.1'>2.1
    <div id='sibling-2.2'>2.2</div>
    <div id='sibling-2.2'>2.3</div>
      <div id='sibling-3.1'>3.1
      <div id='sibling-3.2'>3.2</div>
      <div id='sibling-3.2'>3.3</div>
        <div id='sibling-3.1'>4.1
        <div id='sibling-3.2'>4.2</div>
        <div id='sibling-3.2'>4.3</div>
        <!-- ... -->
```

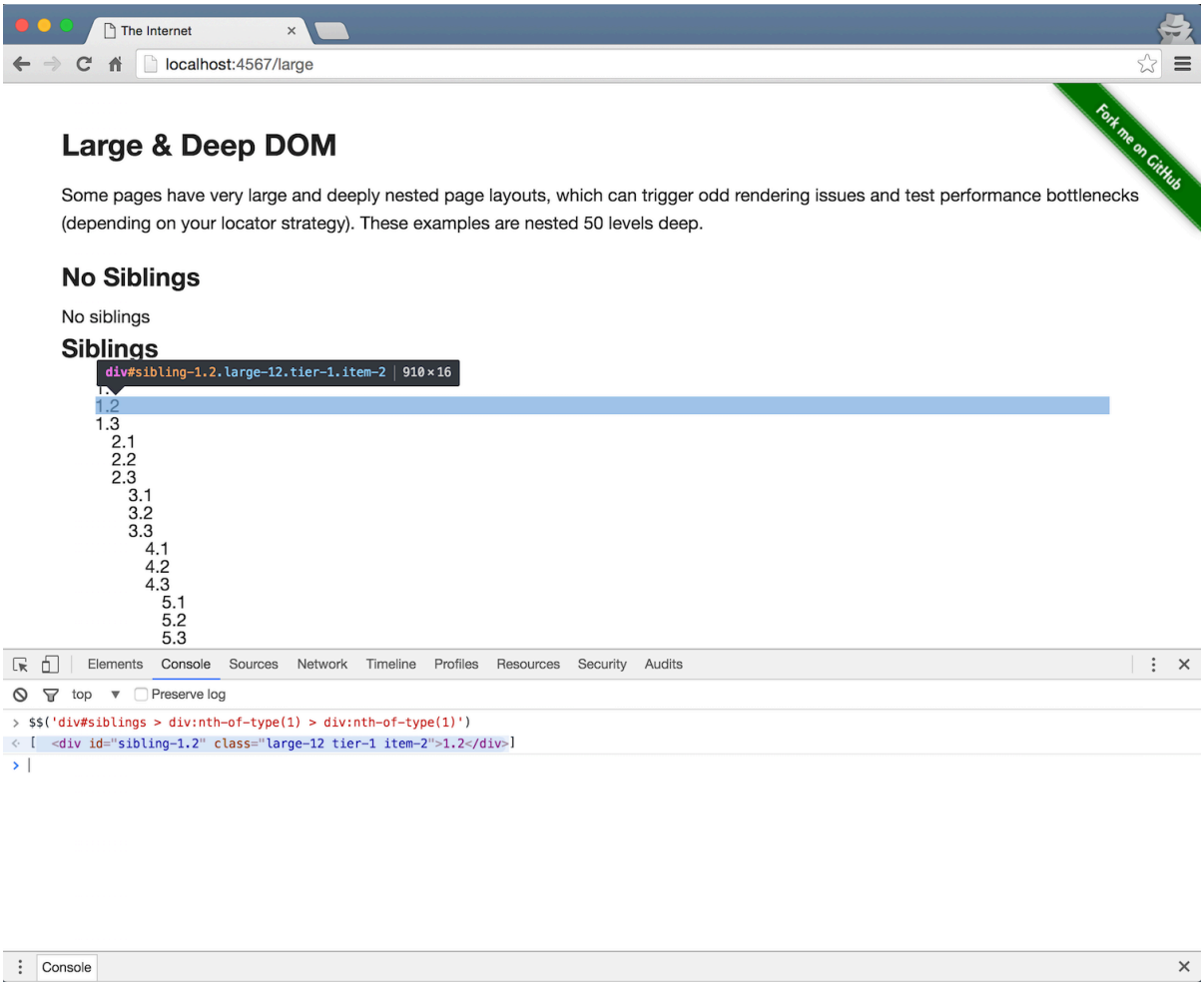
If we perform a `findElement` action using the following locator, it works.

```
Driver.FindElement(By.CssSelector, "div#siblings > div:nth-of-type(1) >
div:nth-of-type(1)");
```

But if we try to go one level deeper with the same approach, it won't work.

```
Driver.FindElement(By.CssSelector, "div#siblings > div:nth-of-type(1) >
div:nth-of-type(1) > div:nth-of-type(1)");
```

Fortunately with our in-browser approach to verifying our locators, we can quickly discern where the issue is. Here's what it shows us for the locators that "worked".

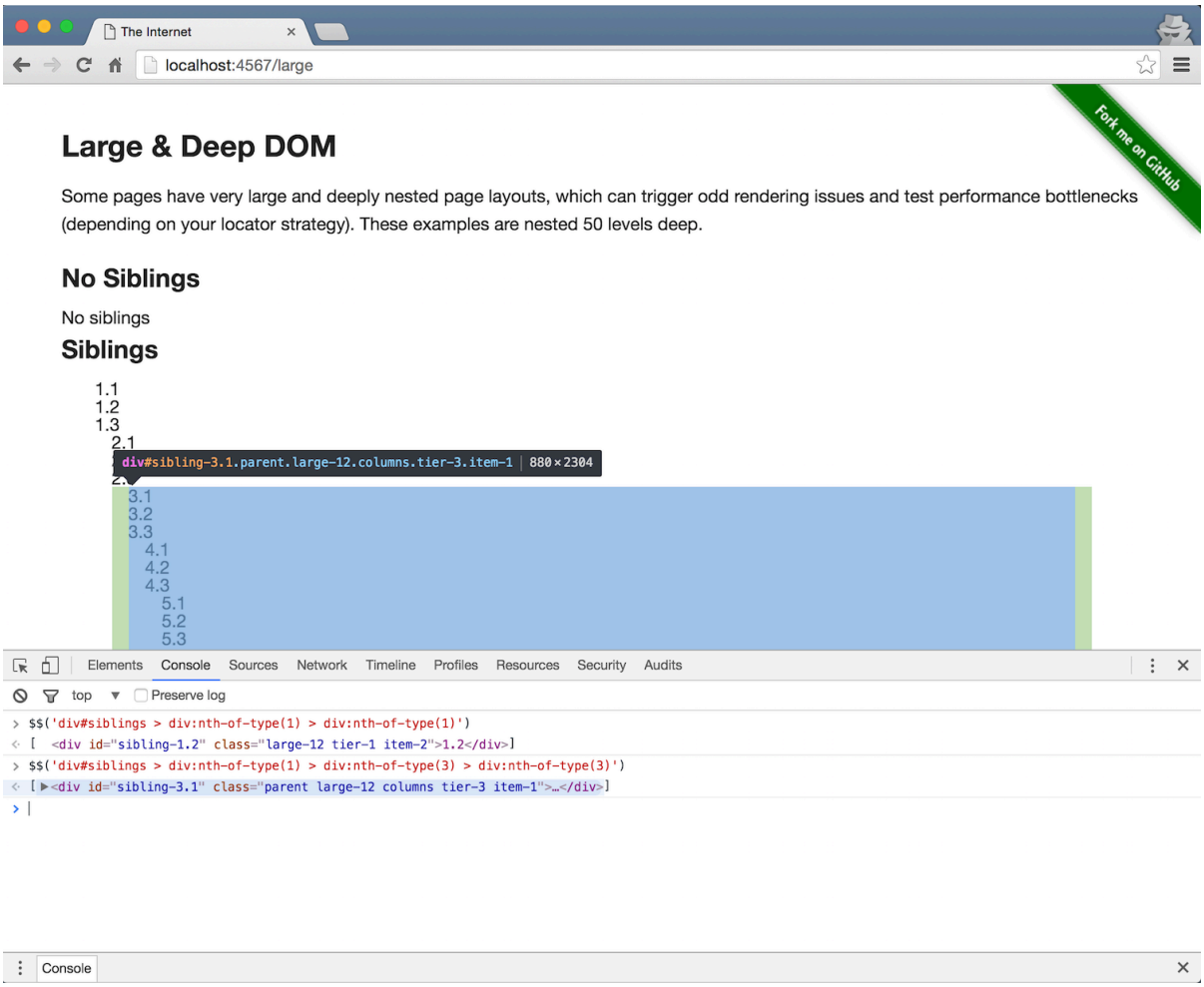


It looks like our locators are scoping to the wrong part of the first level (1.2). But we need to reference the third part of each level (e.g., 1.3, 2.3, 3.3) in order to traverse deeper since the nested divs live under the third part of each level.

So if we try this locator instead, it should work.

```
Driver.FindElement(By.CssSelector, "div#siblings > div:nth-of-type(1) >
div:nth-of-type(3) > div:nth-of-type(3)");
```

We can confirm that it works before changing any test code by looking in the JavaScript Console first.



This should help save you time and frustration when running down tricky locators in your tests. It definitely has for me.

Chapter 8
Writing Re-usable Test Code

One of the biggest challenges with Selenium tests is that they can be brittle and challenging to maintain over time. This is largely due to the fact that things in the application you're testing change -- causing your tests to break.

But the reality of a software project is that change is a constant. So we need to account for this reality somehow in our test code in order to be successful.

Enter Page Objects.

A Page Objects Primer

Rather than write your test code directly against your app, you can model the behavior of your application into simple objects and write your tests against them instead. That way when your app changes and your tests break, you only have to update your test code in one place to fix it.

With this approach we not only get the benefit of controlled chaos, we also get reusable functionality across our suite of tests and more readable tests.

An Example

Part 1: Create A Page Object And Update Test

Let's take our login example from earlier, create a page object for it, and update our test accordingly.

First we'll need to create a new folder called `PageObjects` in the root of our project (just like we did for `Tests`). Then let's add a file to the `PageObjects` directory called `LoginPage.cs`. When we're done our directory structure should look like this.

```
PageObjects
   LoginPage.cs
Tests
   LoginTest.cs
Vendor
    geckodriver.exe
```

And here's the code that goes with it.

```
// filename: PageObjects/LoginPage.cs
using OpenQA.Selenium;

namespace PageObjects
{
    class LoginPage
    {
        IWebDriver Driver;
        By UsernameInput = By.Id("username");
        By PasswordInput = By.Id("password");
        By SubmitButton = By.CssSelector("button");
        By SuccessMessage = By.CssSelector(".flash.success");

        public LoginPage(IWebDriver driver)
        {
            Driver = driver;
            Driver.Navigate().GoToUrl("http://the-internet.herokuapp.com/login");
        }

        public void With(string username, string password)
        {
            Driver.FindElement(UsernameInput).SendKeys(username);
            Driver.FindElement(PasswordInput).SendKeys(password);
            Driver.FindElement(SubmitButton).Click();
        }

        public bool SuccessMessagePresent()
        {
            return Driver.FindElement(SuccessMessage).Displayed;
        }

    }
}
```

At the top of the file we include Selenium and specify the namespace (which is the same as the folder name of where the file lives). We then declare the class (e.g., `class Login`), specify our field variables (for the Selenium instance and the page's locators), and add three methods.

The first method (e.g., `public LoginPage(IWebDriver driver)`) is the constructor. It will run whenever a new instance of this class is created. In order for this class to work we need access to the Selenium driver object, so we accept it as a parameter here and store it in the `Driver` field (so other methods can access it). Then the login page is visited (with `Driver.Navigate().GoToUrl`).

The second method (e.g., `public void With(string username, string password)`) is the core functionality of the login page. It's responsible for filling in the login form and submitting it. By accepting string parameters for the username and password we're able to make the functionality

here reusable for additional tests.

The last method (e.g., `public bool SuccessMessagePresent()`) is the display check from earlier that was used in our assertion. It will return a Boolean result just like before.

Now let's update our test to use this page object.

```
// filename: Tests/LoginTest.cs
using System;
using NUnit.Framework;
using OpenQA.Selenium;
using OpenQA.Selenium.Firefox;
using PageObjects;

namespace Tests
{
[TestFixture]
    class LoginTest
    {
        private static string VendorDirectory = System.IO.Directory.GetParent(
            System.AppContext.BaseDirectory).
            Parent.Parent.Parent.FullName
            + @"/vendor";
        IWebDriver Driver;
        LoginPage Login;

[SetUp]
        public void SetUp()
        {
            var Service = FirefoxDriverService.CreateDefaultService(VendorDirectory);
            Driver = new FirefoxDriver(Service);
            Login = new LoginPage(Driver);
        }

[TearDown]
        public void TearDown()
        {
            Driver.Quit();
        }

[Test]
        public void ValidAccount()
        {
            Login.With("tomsmith", "SuperSecretPassword!");
            Assert.That(Login.SuccessMessagePresent);
        }
    }
}
```

Since the page objects are in another namespace we need to include it (e.g., `using PageObjects;`).

Then it's a simple matter of specifying a field variable for the page object (e.g., `LoginPage Login`),

creating an instance of it in our `SetUp()` method (passing the `Driver` object to it as an argument), and updating the test with the new actions.

Now the test is more concise and readable. And when you save everything and run it, it will run and pass just like before.

Part 2: Write Another Test

Creating a page object may feel like more work than what we started with initially. But it's well worth the effort since we're in a much sturdier position and able easily write follow-on tests (since the specifics of the page are abstracted away for simple reuse).

Let's add another test for a failed login to demonstrate.

First, let's take a look at the markup that gets rendered when we provide invalid credentials:

```
<div id="flash-messages" class="large-12 columns">
  <div data-alert="" id="flash" class="flash error">
    Your username is invalid!
    <a href="#" class="close">x</a>
  </div>
</div>
```

Here is the element we'll want to use in our assertion.

```
class="flash error"
```

Let's add a locator for this element to our page object along with a new method to perform a display check against it.

```
//filename: PageObjects/LoginPage.cs
// ...
        By FailureMessage = By.CssSelector(".flash.error");
// ...
        public bool FailureMessagePresent()
        {
            return Driver.FindElement(FailureMessage).Displayed;
        }
    }
}
```

Now we're ready to add a test for failed login to our `Tests/LoginTest.cs` file.

```
//filename: Tests/LoginTest.cs
// ...

[Test]
        public void BadPasswordProvided()
        {
            Login.With("tomsmith", "bad password");
            Assert.That(Login.FailureMessagePresent);
        }
    }
}
```

If we save these changes and run our tests we will see two browser windows open (one after the other) testing for successful and failure login scenarios.

Why Asserting False Won't Work (yet)

You may be wondering why we didn't just check to see if the success message wasn't present in our assertion.

```
Assert.That(Login.SuccessMessagePresent, Is.False);
```

There are two problems with this approach. First, our test will fail. This is because Selenium errors when it looks for an element that's not present on the page -- which looks like this:

```
Result StackTrace:
Result Message: OpenQA.Selenium.NoSuchElementException : Unable to locate element: {
"method":"css selector","selector":".flash.success"}
```

But don't worry, we'll address this in the next chapter.

Second, the absence of a success message doesn't necessarily indicate a failed login. The assertion we ended up with originally is more effective.

Part 3: Confirm We're In The Right Place

Before we can call our page object finished, there's one more addition we should make. We'll want to add an assertion to make sure that Selenium is in the right place before proceeding. This will help add some resiliency to our test.

```
// filename: PagesObjects/LoginPage.cs
// ...
    class LoginPage
    {
        IWebDriver Driver;
        By LoginForm = By.Id("login");
// ...
        public LoginPage(IWebDriver driver)
        {
            Driver = driver;
            Driver.Navigate().GoToUrl("http://the-internet.herokuapp.com/login");
            if (!Driver.FindElement(LoginForm).Displayed)
            {
                throw new System.Exception("Page not ready");
            }
        }
// ...
```

After adding a locator for the center-piece of this page (the login form) we put it to use in our constructor after the Selenium command that visits the page. With it we're checking to see that the login form is displayed. If it is the tests using this page object will proceed. If not an error will be throw, failing the test, with a message stating that the page wasn't ready.

Now when we save everything and run our tests they will run just like before. But now we can feel confident that the tests will only proceed if login page is in a ready state.

Outro

With Page Objects you'll be able to easily maintain and extend your tests. But how you write your Page Objects may vary depending on your preference and experience. The example demonstrated above is a simple approach. It's worth taking a look at the Selenium project wiki page for Page Objects here (even if its examples are only written in Java). There's also Martin Fowler's seminal blog post on the topic as well (link).

Chapter 9
Writing Really Re-usable Test Code

In the previous chapter we stepped through creating a simple page object to capture the behavior of the page we were interacting with. While this was a good start, it leaves some room for improvement.

As our test suite grows and we add more page objects we will start to see common behavior that we will want to use over and over again throughout our suite. If we leave this unchecked we will end up with duplicative code which will slowly make our page objects harder to maintain.

Right now we are using Selenium actions directly in our page object. While on the face of it this may seem fine, it has some long term impacts, like:

- slower page object creation due to the lack of a simple Domain Specific Language (DSL)
- test maintenance issues when the Selenium API changes (e.g., major changes between Selenium RC and Selenium WebDriver)
- the inability to swap out the driver for your tests (e.g., mobile, REST, etc.)

With a Base Page Object (a.k.a. a facade layer) we can easily side step these concerns by abstracting all of our common actions into a central class and leveraging it in our page objects.

An Example

Let's step through an example with our login page object.

Part 1: Create The Base Page Object

First let's create the base page object by adding a file called `BasePage.cs` to the `pageobjects` package.

```
PageObjects
   BasePage.cs
   LoginPage.cs
Tests
   LoginTest.cs
Vendor
    geckodriver.exe
```

Next let's populate the file.

```
// filename: PageObjects/BasePage.cs
using OpenQA.Selenium;

namespace PageObjects
{
    class BasePage
    {
        IWebDriver Driver;

        protected BasePage(IWebDriver driver)
        {
            this.Driver = driver;
        }

        protected void Visit(string url)
        {
            Driver.Navigate().GoToUrl(url);
        }

        protected IWebElement Find(By locator)
        {
            return Driver.FindElement(locator);
        }

        protected void Click(By locator)
        {
            Find(locator).Click();
        }

        protected void Type(By locator, string inputText)
        {
            Find(locator).SendKeys(inputText);
        }

        protected bool IsDisplayed(By locator)
        {
            return Find(locator).Displayed;
        }
    }
}
```

After declaring the class (e.g., `class BasePage`) we receive and store an instance of Selenium just like in our Login page object. But what's different here is the methods that come after the constructor (e.g., `Visit`, `Find`, `Click`, `Type`, and `IsDisplayed`). Each one stores a specific behavior we've used in our tests. Some of the names are the same as you've seen in Selenium, others renamed (for improved readability).

Now that we have all of our Selenium actions in one place, let's update our login page object to leverage this facade.

```
// filename: PageObjects/LoginPage.cs
using OpenQA.Selenium;
using NUnit.Framework;

namespace PageObjects
{
    class LoginPage : BasePage
    {
        By LoginForm = By.Id("login");
        By UsernameInput = By.Id("username");
        By PasswordInput = By.Id("password");
        By SubmitButton = By.CssSelector("button");
        By SuccessMessage = By.CssSelector(".flash.success");
        By FailureMessage = By.CssSelector(".flash.error");

        public LoginPage(IWebDriver driver)
        {
            Visit("http://the-internet.herokuapp.com/login");
            Assert.That(IsDisplayed(LoginForm));
        }

        public void With(string username, string password)
        {
            Type(UsernameInput, username);
            Type(PasswordInput, password);
            Click(SubmitButton);
        }

        public bool SuccessMessagePresent()
        {
            return IsDisplayed(SuccessMessage);
        }

        public bool FailureMessagePresent()
        {
            return IsDisplayed(FailureMessage);
        }
    }
}
```

Two fundamental things have changed in our Login page object.

First, we've established inheritance between `BasePage` and `LoginPage` with `class LoginPage :`

`BasePage`. This means that `LoginPage` is now a child of `BasePage` and can access its methods. Second, we've swapped out all of the Selenium actions to use the methods made available from `BasePage` thanks to inheritance.

If we save everything and run our tests they will run and pass just like before. But now, our page objects are more readable, simpler to write, and easier to maintain and extend.

Part 2: Add Some Error Handling

Remember in the previous chapter when we ran into an error with Selenium when we looked for an element that wasn't on the page? Let's address that now.

To recap -- here's the error message we saw:

```
Result StackTrace:
Result Message: OpenQA.Selenium.NoSuchElementException : Unable to locate element: {
"method":"css selector","selector":".flash.success"}
```

The important thing to note is the exception Selenium offered up -- the part that comes before `Unable to locate element` (e.g., `OpenQA.Selenium.NoSuchElementException`). Let's modify the `IsDisplayed` method in our base page object to handle it.

```
// filename: PageObjects/BasePage.cs
// ...
        protected bool IsDisplayed(By locator)
        {
            try {
                return Find(locator).Displayed;
            } catch(OpenQA.Selenium.NoSuchElementException) {
                return false;
            }
        }
    }
}
```

By wrapping our Selenium action (e.g., `return Find(locator).isDisplayed();`) in a `try` / `catch` we're able to catch the exception and return `false` instead. This will enable us to see if an element is on the page. If it's not, we'll receive a `false` Boolean rather than an exception.

With this new handling in place, let's revisit our `InvalidAccount()` login test and alter it so it checks to see if the success message is not present (which would normally trigger a `NoSuchElementException`) to make sure things work as we expect.

```
// filename: Tests/LoginTest.cs
// ...
[Test]
        public void BadPasswordProvided()
        {
            Login.With("tomsmith", "bad password");
            Assert.That(Login.SuccessMessagePresent, Is.False);
        }
    }
}
```

When we save our changes and run this test it will run and pass without throwing an exception.

Chapter 10

Writing Resilient Test Code

Ideally you should be able to write your tests once and run them across all supported browsers. While this is a rosy proposition, there is some work to make this a reliable success. And sometimes there may be a hack or two involved. But the lengths you must go really depends on the browsers you care about and the functionality you're dealing with in your application.

By using high quality locators we're already in good shape, but there are still some issues to deal with. Most notably... timing. This is especially true when working with dynamic, JavaScript heavy pages (which is more the rule than the exception in a majority of web applications you'll deal with).

But there is a simple approach that makes up the bedrock of reliable and resilient Selenium tests -- and that's how you wait and interact with elements. The best way to accomplish this is through the use of explicit waits.

An Explicit Waits Primer

Explicit waits are applied to individual test actions. Each time you want to use one you specify an amount of time (in seconds) and the Selenium action you want to accomplish.

Selenium will repeatedly try this action until either it can be accomplished, or until the amount of time specified has been reached. If the latter occurs, a timeout exception will be thrown.

An Example

Let's step through an example that demonstrates this against a dynamic page on the-internet. The functionality is pretty simple -- there is a button. When you click it a loading bar appears for 5 seconds. After that it disappears and is replaced with the text `Hello World!`.

Part 1: Create A New Page Object And Update The Base Page Object

Here's the markup from the page.

```
<div class="example">
  <h3>Dynamically Loaded Page Elements</h3>
  <h4>Example 1: Element on page that is hidden</h4>

  <br>

  <div id="start">
    <button>Start</button>
  </div>

  <div id="finish" style="display:none">
    <h4>Hello World!</h4>
  </div>

</div>
```

At a glance it's simple enough to tell that there are unique `id` attributes that we can use to find and click on the start button and verify the finish text.

When writing automation for new functionality like this, you may find it easier to write the test first (to get it working how you'd like) and then create a page object for it (pulling out the behavior and locators from your test). There's no right or wrong answer here. Do what feels intuitive to you. But for this example, we'll create the page object first, and then write the test.

Let's create a new page object file called `DynamicLoadingPage.cs` in the `PageObjects` directory.

```
PageObjects
   BasePage.cs
   DynamicLoadingPage.cs
   LoginPage.cs
Tests
   LoginTest.cs
Vendor
    geckodriver.exe
```

In this file we'll establish inheritance to the base page object and specify the locators and behavior we'll want to use.

```
// filename: PageObjects/DynamicLoadingPage.cs
using OpenQA.Selenium;

namespace PageObjects
{
    class DynamicLoadingPage : BasePage
    {
        By StartButton = By.CssSelector("#start > button");
        By FinishText = By.Id("finish");

        public DynamicLoadingPage(IWebDriver driver) { }

        public void LoadExample(int exampleNumber)
        {
            Visit("http://the-internet.herokuapp.com/dynamic_loading/" + exampleNumber
);
            Click(StartButton);
        }

        public bool FinishTextPresent()
        {
            return IsDisplayed(FinishText, 10);
        }
    }
}
```

Since there are two dynamic loading examples to choose from on the-internet we created the method `LoadExample` which accepts a number as an argument so we can specify which of the examples we want to visit and start.

And similar to our Login page object, we have a display check for the finish text (e.g., `FinishTextPresent()`). This check is slightly different though. Aside from the different name, it has a second argument (an integer value of `10`). This second argument is how we'll specify how long we'd like Selenium to wait for an element to be displayed before giving up.

Let's update our base page object to enable explicit waits by adding an overloaded method for `IsDisplayed` (e.g., a method with the same name but different parameters).

```
// filename: PageObjects/BasePage.cs
// ...
        protected bool IsDisplayed(By locator)
        {
            try {
                return Find(locator).Displayed;
            } catch(OpenQA.Selenium.NoSuchElementException) {
                return false;
            }
        }

        protected bool IsDisplayed(By locator, int maxWaitTime)
        {
            try {
                WebDriverWait wait = new WebDriverWait(Driver, System.TimeSpan.
FromSeconds(maxWaitTime));
                wait.Until(driver => driver.FindElement(locator).Displayed);
                return true;
            } catch(OpenQA.Selenium.WebDriverTimeoutException) {
                return false;
            }
        }
    }
}
```

Selenium comes with a wait function which we wrap in a new `IsDisplayed` method.

The `waitForIsDisplayed` method has two parameters -- one for a locator (e.g., `By locator`) and another for the timeout (e.g., `maxWaitTime`). Inside the method we create an instance of `WebDriverWait`, specify the timeout as seconds, pass in `maxWaitTime`, specify the condition we want to wait for, and pass in locator we're interested in. The "expected condition" we want to wait for is doing the same thing as our previous display lookup, but with the explicit waits function.

This function doesn't return a Boolean so we have to provide one (e.g., `return true;`). If the condition is not met by Selenium in the amount of time provided it will throw a timeout exception. So we catch it and return `false` instead when that happens.

More On Explicit Waits

It's important to set a reasonably sized default timeout for the explicit wait method. But you want to be careful not to make it too high. Otherwise you can run into similar timing issues you get from an implicit wait. But set it too low and your tests will be brittle, forcing you to run down trivial and transient issues.

The major benefit of explicit waits is that if the behavior on the page takes longer than we expect

(e.g., due to slow load times, or a feature change), we can simply adjust this one wait time to fix the test -- rather than increase a blanket wait time (which impacts every test). And since the wait is dynamic (e.g., constantly polling), it won't take the full amount of time to complete (like a hard-coded `Thread.Sleep();` would).

If you're thinking about mixing explicit waits with an implicit wait -- don't. If you use both together you're going to run into issues later on due to inconsistent implementations of the implicit wait functionality across local and remote browser drivers. Long story short, you'll end up with tests that could fail randomly and when they do they will be hard to debug. You can read more about the specifics here.

Part 2: Write A Test To Use The New Page Object

Now that we have our new page object and an updated base page, it's time to write our test to use it.

Let's create a new file called `DynamicLoadingTest.cs` in the `Tests` directory.

```
PageObjects
   BasePage.cs
   DynamicLoadingPage.cs
   LoginPage.cs
Tests
   DynamicLoadingTest.cs
   LoginTest.cs
Vendor
    geckodriver.exe
```

The contents of this test file are similar to `LoginTest` with regards to the imported classes and the `SetUp` / `TearDown` methods.

```
// filename: Tests/DynamicLoadingTest.cs
using NUnit.Framework;
using OpenQA.Selenium;
using OpenQA.Selenium.Firefox;
using PageObjects;

namespace Tests
{
[TestFixture]
    class DynamicLoadingTest
    {
        private static string VendorDirectory = System.IO.Directory.GetParent(
            System.AppContext.BaseDirectory).
            Parent.Parent.Parent.FullName
            + @"/vendor";
        IWebDriver Driver;
        DynamicLoadingPage DynamicLoading;

[SetUp]
        public void SetUp()
        {
            var Service = FirefoxDriverService.CreateDefaultService(VendorDirectory);
            Driver = new FirefoxDriver(Service);
            DynamicLoading = new DynamicLoadingPage(Driver);
        }

[TearDown]
        public void TearDown()
        {
            Driver.Quit();
        }

[Test]
        public void ElementHidden()
        {
            DynamicLoading.LoadExample(1);
            Assert.That(DynamicLoading.FinishTextPresent);
        }
    }
}
```

In our test (e.g., `public void ElementHidden()`) we are visiting the first dynamic loading example and clicking the start button (which is accomplished in `DynamicLoading.LoadExample(1)`). We're then asserting that the finish text gets rendered.

When we save this and run it it will:

- Launch a browser
- Visit the page
- Click the start button
- Wait for the loading bar to complete
- Find the finish text
- Assert that it is displayed.

Part 3: Update Page Object And Add A New Test

Let's step through one more example to see if our explicit wait approach holds up.

The second dynamic loading example is laid out similarly to the last one. The only difference is that it renders the final text after the progress bar completes (whereas the previous example had the element on the page but it was hidden).

Here's the markup for it.

```
<div class="example">
  <h3>Dynamically Loaded Page Elements</h3>
  <h4>Example 2: Element rendered after the fact</h4>

  <br>

  <div id="start">
    <button>Start</button>
  </div>

  <br>
</div>
```

In order to find the selector for the finish text element we need to inspect the page after the loading bar sequence finishes. Here's what it looks like.

```
<div id="finish" style=""><h4>Hello World!</h4></div>
```

Let's add a second test to `DynamicLoadingTest.cs` called `ElementRendered()` that will load this second example and perform the same check as we did for the previous test.

```
// filename: Tests/DynamicLoadingTest.cs
// ...
[Test]
        public void ElementRendered()
        {
            DynamicLoading.LoadExample(2);
            Assert.That(DynamicLoading.FinishTextPresent);
        }
    }
}
```

When we run both tests we will see that the same approach will work in both cases of how the page is constructed.

Browser Timing

Using explicit waits gets you pretty far. But there are a few things you'll want to think about when it comes to writing your tests to work on various browsers.

It's simple enough to write your tests locally against Firefox and assume you're all set. But once you start to run things against other browsers, you may be in for a rude awakening. The first thing you're likely to run into is the speed of execution. A lot of your tests will start to fail when you point them at either Chrome or Internet Explorer, and likely for different reasons.

Chrome execution can sometimes be faster than Firefox, so you could see some odd timeout failures. This is an indicator that you need to add explicit waits to parts of your page objects that don't already have them. And the inverse is true when running things against older version of Internet Explorer (e.g., IE 8, 9, etc.). This is an indicator that your explicit wait times are not long enough since the browser is taking longer to respond -- so your tests timeout.

The best approach to solve this is an iterative one. Run your tests in a target browser and see which ones fail. Take each failed test, adjust your code as needed, and re-run it against the target browser until they all pass. Repeat for each browser you care about until everything is green.

Closing Thoughts

By explicitly waiting to complete an action, our tests are in a much more resilient position because Selenium will keep trying for a reasonable amount of time rather than trying just once. And each action can be tuned to meet the needs of each circumstance. Couple that with the dynamic nature of explicit waits, and you have something that will work in a multitude of circumstances -- helping you endure even the toughest of browsers to automate.

This is one of the most important concepts in testing with Selenium. Use explicits waits often.

Chapter 11

Prepping For Use

Now that we have some tests and page objects, we'll want to start thinking about how to structure our test code to be more flexible. That way it can scale to meet our needs.

Part 1: Global Setup & Teardown

We'll start by pulling the Selenium setup and teardown out of our tests and into a central location.

Similar to our base page object, we'll want to create a base test. So let's create a new file called `BaseTest.cs` in the `Tests` directory.

```
PageObjects
   BasePage.cs
   DynamicLoadingPage.cs
   LoginPage.cs
Tests
   BaseTest.cs
   DynamicLoadingTest.cs
   LoginTest.cs
Vendor
    geckodriver.exe
```

And here are the contents of the file.

```
// filename: Tests/BaseTest.cs
using OpenQA.Selenium;
using NUnit.Framework;
using OpenQA.Selenium.Firefox;

namespace Tests
{
[TestFixture]
    class BaseTest
    {
        private static string VendorDirectory = System.IO.Directory.GetParent(
            System.AppContext.BaseDirectory).
            Parent.Parent.Parent.FullName
            + @"/vendor";
        protected IWebDriver Driver;

[SetUp]
        protected void SetUp()
        {
            var Service = FirefoxDriverService.CreateDefaultService(VendorDirectory);
            Driver = new FirefoxDriver(Service);
        }

[TearDown]
        protected void TearDown()
        {
            Driver.Quit();
        }
    }
}
```

After importing a few necessary classes we specify the `BaseTest` class and wire up some methods that will take care of setting up and tearing down Selenium before and after each test.

Now let's update our tests to establish inheritance with this base test class, remove the Selenium setup/teardown actions, and remove the unnecessary `using` statements. When we're done our test files should look like this:

```
// filename: Tests/LoginTest.cs
using NUnit.Framework;
using PageObjects;

namespace Tests
{
[TestFixture]
    class LoginTest : BaseTest
    {
        LoginPage Login;

[SetUp]
        public new void SetUp()
        {
            Login = new LoginPage(Driver);
        }

[Test]
        public void ValidAccount()
        {
            Login.With("tomsmith", "SuperSecretPassword!");
            Assert.That(Login.SuccessMessagePresent);
        }

[Test]
        public void BadPasswordProvided()
        {
            Login.With("tomsmith", "bad password");
            Assert.That(Login.FailureMessagePresent);
        }
    }
}
```

```
// filename: Tests/DynamicLoadingTest.cs
using NUnit.Framework;
using PageObjects;

namespace Tests
{
[TestFixture]
    class DynamicLoadingTest : BaseTest
    {
        DynamicLoadingPage DynamicLoading;

[SetUp]
        public new void SetUp()
        {
            DynamicLoading = new DynamicLoadingPage(Driver);
        }

[Test]
        public void ElementHidden()
        {
            DynamicLoading.LoadExample(1);
            Assert.That(DynamicLoading.FinishTextPresent);
        }

[Test]
        public void ElementRendered()
        {
            DynamicLoading.LoadExample(2);
            Assert.That(DynamicLoading.FinishTextPresent);
        }
    }
}
```

Part 2: Base URL

It's a given that we'll need to run our tests against different environments (e.g., local, test, staging, production, etc.). So let's make it so we can specify a different base URL for our tests at runtime.

We can do this by leveraging an environment variable and updating our `BaseTest` to receive it or set a sensible default if one isn't provided.

```
// filename: Tests/BaseTest.cs
// ...
    class BaseTest
    {
        private static string VendorDirectory = System.IO.Directory.GetParent(
            System.AppContext.BaseDirectory).
            Parent.Parent.Parent.FullName
            + @"/vendor";
        protected IWebDriver Driver;
        public static string BaseUrl;

[SetUp]
        protected void SetUp()
        {
            BaseUrl     = System.Environment.GetEnvironmentVariable("BASE_URL") ??
"http://the-internet.herokuapp.com";
// ...
```

By using `System.Environment.GetEnvironmentVariable` we are able to access environment variables by their name, which we do for `BASE_URL`, storing it in a public variable of `BaseUrl`. If no value is provided, we default to the publicly available end-point of `the-internet`.

Now to use the `BaseUrl` in our page objects.

```
// filename: PageObjects/BasePage.cs
// ...
        protected void Visit(string url)
        {
            if (url.StartsWith("http"))
            {
                Driver.Navigate().GoToUrl(url);
            } else
            {
                Driver.Navigate().GoToUrl(Tests.BaseTest.BaseUrl + url);
            }
        }
// ...
```

In `Visit` there could be a case where we'll want to navigate to a full URL so to be safe we've added a conditional check of the `url` parameter to see if a full URL was passed in. If so, we'll visit it. If not, the `BaseUrl` will be combined with the URL path that was passed in to `url` to create a full URL.

Now all we need to do is update our page objects so they're no longer using hard-coded URLs when calling `Visit`.

```
// filename: PageObjects/LoginPage.cs
// ...
        public LoginPage(IWebDriver driver)
        {
            Visit("/login");
            Assert.That(IsDisplayed(LoginForm));
        }
// ...
```

```
// filename: PageObjects/DynamicLoadingPage.cs
// ...
        public void LoadExample(int exampleNumber)
        {
            Visit("/dynamic_loading/" + exampleNumber);
            Click(StartButton);
        }
// ...
```

Outro

Now when running our tests, we can specify a different base URL by specifying a `BASE_URL` either on our machine or when running our tests from the command-line (e.g., `BASE_URL=http://your-new-url dotnet test`). We're also in a better position now with our setup and teardown abstracted into a central location. Now we can easily extend our test framework to run our tests on other browsers.

Chapter 12

Running A Different Browser Locally

It's straightforward to get your tests running locally against Firefox (that's what we've been doing up until now). But when you want to run them against a different browser like Chrome, Safari, or Internet Explorer you quickly run into configuration overhead that can seem overly complex and lacking in good documentation or examples.

A Brief Primer On Browser Drivers

With the introduction of WebDriver (circa Selenium 2) a lot of benefits were realized (e.g., more effective and faster browser execution, no more single host origin issues, etc). But with it came some architectural and configuration differences that may not be widely known. Namely -- browser drivers.

WebDriver works with each of the major browsers through a browser driver which is (ideally but not always) maintained by the browser manufacturer. It is an executable file (consider it a thin layer or a shim) that acts as a bridge between Selenium and the browser.

Let's step through an example using ChromeDriver.

An Example

Before starting, we'll need to download the latest ChromeDriver binary executable for our operating system from here (pick the highest numbered directory) an unzip its contents into the `Vendor` directory.

```
PageObjects
   BasePage.cs
   DynamicLoadingPage.cs
   LoginPage.cs
Tests
   BaseTest.cs
   DynamicLoadingTest.cs
   LoginTest.cs
Vendor
    chromedriver
    geckodriver
```

In order for Selenium to use this binary we have to make sure it knows where it is. There are two ways to do that. We can add `chromedriver` to the path of our system, or use the same approach we did for Firefox (specifying the path to the vendor directory through a service object). Let's go

with the latter option.

We'll also want to make sure our test suite can run either Firefox or Chrome. To do that, we'll need to make a couple of changes.

First, let's add support for a `BROWSER_NAME` environment variable.

```
// filename: Tests/BaseTest.cs
// ...
    class BaseTest
    {
        // ...
        private static string BrowserName;

[SetUp]
        protected void SetUp()
        {
            BaseUrl       = System.Environment.GetEnvironmentVariable("BASE_URL") ??
"http://the-internet.herokuapp.com";
            BrowserName   = System.Environment.GetEnvironmentVariable("BROWSER_NAME")
?? "firefox";
            // ...
```

After creating field variables for both the browser name and the vendor directory we update the `LoadConfigValues` private method to populate them. Fetching the `BrowserName` is straightforward. We're doing the same thing that we did for `ApplicationBaseUrl` and just providing the new key. For the vendor directory we're looking up the full path to the root of the test directory (by way of looking up the assembly location for the base test and its parent directory) and appending `\Vendor` with a string literal (e.g., `@`). This gives us the full path to the `Vendor` directory.

Now to modify the browser setup.

```
// filename: Tests/BaseTest.cs
using OpenQA.Selenium.Chrome;
// ...
[SetUp]
        protected void SetUp()
        {
            // ...
            switch (BrowserName.ToLower())
            {
                case "firefox":
                {
                    var Service = FirefoxDriverService.CreateDefaultService(
VendorDirectory);
                    Driver = new FirefoxDriver(Service);
                    break;
                }
                case "chrome":
                {
                    var Service = ChromeDriverService.CreateDefaultService(
VendorDirectory);
                    Driver = new ChromeDriver(Service);
                    break;
                }
            }
        }
// ...
```

In `SetUp` we're performing a conditional check against the `BrowserName` field variable (after we set it to lower-case letters for consistency). When the value is set to `"firefox"` we run the tests just like before. When set to `"chrome"` we create a Chrome service object, give it the path to the vendor directory, and create a new instance of ChromeDriver.

Now we can run our tests on a different browser by specifying its name at run-time (e.g., `BROWSER_NAME="chrome" dotnet test`).

Additional Browsers

A similar approach can be applied to other browser drivers, with the only real limitation being the operating system you're running. But remember -- no two browser drivers are alike. Be sure to check out the documentation for the browser you care about to find out the specific requirements:

- ChromeDriver
- EdgeDriver
- geckodriver (Firefox)

- InternetExplorerDriver
- SafariDriver

Chapter 13

Running Browsers In The Cloud

If you've ever needed to test features in an older browser like Internet Explorer 8 or 9 then odds are you ran a virtual machine (VM) on your computer with a "legit" version of Windows.

Handy, but what happens when you need to check things on multiple versions of IE? Now you're looking at multiple VMs. And what about when you need cover other browser and Operating System (OS) combinations? Now you're looking at provisioning, running, and maintaining your own set of machines and standing up something like Selenium Grid to coordinate tests across them.

Rather than take on the overhead of a test infrastructure you can easily outsource this to a third-party cloud provider like Sauce Labs.

A Selenium Remote, Selenium Grid, And Sauce Labs Primer

At the heart of Selenium at scale is the use of Selenium Grid and Selenium Remote.

Selenium Grid lets you distribute test execution across several machines and you connect to it with Selenium Remote. You tell the Grid which browser and OS you want your test to run on through the use of Selenium Remote's `Capabilities`.

Under the hood this is how Sauce Labs works. They are ultimately running Selenium Grid behind the scenes, and they receive and execute tests through Selenium Remote and the `Capabilities` you set.

Let's dig in with an example.

An Example

Part 1: Initial Setup

NOTE: You'll need an account to use Sauce Labs. Their free trial offers enough to get you started. And if you're signing up because you want to test an open source project, then be sure to check out their Open Sauce account.

With Sauce Labs we need to provide specifics about what we want in our test environment, our credentials, and configure Selenium a little bit differently. Let's start by updating our `BaseTest.cs` file to include the details we'll need to specify.

In addition to the `BaseUrl` and `BrowserName` key/values, we'll add some more (e.g., `Host`, `BrowserVersion`, `PlatformName`, `sauceUsername`, and `sauceAccessKey`).

`Host` enables us to specify whether our tests run locally or on Sauce Labs.

With `BrowserName`, `BrowserVeresion`, and `PlatformName` we can specify which browser and operating system combination we want our tests to run on. You can see a full list of Sauce's available platform options here. They also have a handy configuration generator (which will tell you what values to plug into your test) here. `sauceUsername` and `sauceAccessKey` enable us to connect and run our tests.

Now we can update our base test class to work with Selenium Remote. Let's start by pulling in the new configuration values.

```
// filename: Tests/BaseTest.cs
// ...
    class BaseTest
    {
        private static string VendorDirectory = System.IO.Directory.GetParent(
            System.AppContext.BaseDirectory).
            Parent.Parent.Parent.FullName
            + @"/vendor";
        protected IWebDriver Driver;
        public static string BaseUrl;
        private static string Host;
        private static string BrowserName;
        private static string BrowserVersion;
        private static string PlatformName;

[SetUp]
        protected void SetUp()
        {
            BaseUrl         = System.Environment.GetEnvironmentVariable("BASE_URL") ??
"http://the-internet.herokuapp.com";
            Host            = System.Environment.GetEnvironmentVariable("HOST") ??
"saucelabs";
            BrowserName     = System.Environment.GetEnvironmentVariable("BROWSER_NAME")
?? "ie";
            BrowserVersion  = System.Environment.GetEnvironmentVariable(
"BROWSER_VERSION") ?? "10.0";
            PlatformName    = System.Environment.GetEnvironmentVariable("PLATFORM_NAME"
) ?? "Windows 8";
            var sauceUsername = System.Environment.GetEnvironmentVariable(
"SAUCE_USERNAME");
            var sauceAccessKey = System.Environment.GetEnvironmentVariable(
"SAUCE_ACCESS_KEY");
            var url = new Uri($
"http://{sauceUsername}:{sauceAccessKey}@ondemand.saucelabs.com:80/wd/hub");
// ...
```

Now to update our browser setup.

```
// filename: Tests/BaseTest.cs
// ...
using OpenQA.Selenium.IE;
using OpenQA.Selenium.Edge;
using OpenQA.Selenium.Remote;
// ...
[SetUp]
        protected void SetUp()
```

```
            // ...
            switch (Host.ToLower())
            {
                case "localhost":
                    switch (BrowserName.ToLower())
                    {
                        case "firefox":
                        {
                            var Service = FirefoxDriverService.CreateDefaultService(
VendorDirectory);
                            Driver = new FirefoxDriver(Service);
                            break;
                        }
                        case "chrome":
                        {
                            var Service = ChromeDriverService.CreateDefaultService(
VendorDirectory);
                            Driver = new ChromeDriver(Service);
                            break;
                        }
                    }
                    break;
                case "saucelabs":
                    switch (BrowserName.ToLower())
                    {
                        case "chrome":
                        {
                            ChromeOptions options = new ChromeOptions();
                            options.PlatformName = PlatformName;
                            options.BrowserVersion = BrowserVersion;
                            Driver = new RemoteWebDriver(url, options.ToCapabilities
());
                            break;
                        }
                        case "edge":
                        {
                            EdgeOptions options = new EdgeOptions();
                            options.PlatformName = PlatformName;
                            options.BrowserVersion = BrowserVersion;
                            Driver = new RemoteWebDriver(url, options.ToCapabilities
());
                            break;
                        }
                        case "firefox":
                        {
                            ChromeOptions options = new ChromeOptions();
                            options.PlatformName = PlatformName;
```

```
                            options.BrowserVersion = BrowserVersion;
                            Driver = new RemoteWebDriver(url, options.ToCapabilities
());
                            break;
                        }
                        case "ie":
                        {
                            InternetExplorerOptions options = new
InternetExplorerOptions();
                            options.PlatformName = PlatformName;
                            options.BrowserVersion = BrowserVersion;
                            Driver = new RemoteWebDriver(url, options.ToCapabilities
());
                            break;
                        }
                    }
                    break;
            }
        }
// ...
```

In our `SetUp` method we've amended our conditional flow to check the `Host` variable first. We start by checking to see if it's set to `"localhost"` or `"saucelabs"`. If it's set to `"localhost"` we carry on just like before (checking the `BrowserName` value to determine which browser to launch locally).

If it's set to `"saucelabs"` we create an options object for the given browser, populate it with the `BrowserVersion` and `PlatformName` values and account URL (e.g., `"username"` and `"accessKey"` used in a basic auth URL). We then connect to Sauce Labs using Selenium Remote and pass in the options object. This will return a Selenium WebDriver instance that we can use just like when running our tests locally, except the browser is living on a machine in Sauce Labs' cloud.

If we save everything and run our tests they will execute in Sauce Labs and on the account dashboard we'll see our tests running in Internet Explorer 10 on Windows 8. And to run the tests on different combinations, you can specify them through environment variables on the command-line (e.g., `BROWSER_NAME="safari" BROWSER_VERSION="12.0" PLATFORM_NAME="macOS 10.14" dotnet test`).

Part 2: Test Name

It's great that our tests are running on Sauce Labs. But we're not done yet because the test name in each Sauce job is getting set to `unnamed job`. This makes it extremely challenging to know what test was run in the job. To remedy this we'll need to pass in the test name in `DesiredCapabilities`.

```
// filename: Tests/BaseTest.cs
// ...
[TearDown]
        protected void TearDown()
        {
            if (Host.Equals("saucelabs"))
            {
                var testName = TestContext.CurrentContext.Test.Name;
                ((IJavaScriptExecutor)Driver).ExecuteScript("sauce:job-name=" +
testName);
            }
            Driver.Quit();
        }
    }
}
```

Getting the test name from NUnit is a simple matter of calling `TestContext.CurrentContext.Test.Name`. This gives us the name of the test that is currently running, and we pass it to Sauce Labs through Selenium's `ExecuteScript` command. This runs JavaScript in the context of the application as it's running.

Now when we run our tests in Sauce Labs [the account dashboard] will show the tests running with a correct name.

Part 3: Test Status

There's still one more thing we'll need to handle, and that's setting the status of the Sauce Labs job after it completes.

Right now regardless of the outcome of a test, the job in Sauce Labs will register as `Finished`. Ideally we want to know if the job was a `Pass` or a `Fail`. That way we can tell at a glance if a test failed or not. And with a couple of tweaks to the `TearDown` method in our base test we can make this happen easily enough.

```
// filename: Tests/BaseTest.cs
// ...
[TearDown]
        protected void TearDown()
        {
            if (Host.Equals("saucelabs"))
            {
                var testName = TestContext.CurrentContext.Test.Name;
                bool testPassed = TestContext.CurrentContext.Result.Outcome.Status.
ToString() == "Passed";
                ((IJavaScriptExecutor)Driver).ExecuteScript("sauce:job-name=" +
testName);
                ((IJavaScriptExecutor)Driver).ExecuteScript("sauce:job-result=" + (
testPassed ? "passed" : "failed"));
                if(!testPassed)
                {
                    TestContext.WriteLine($"See a job at
https://saucelabs.com/tests/{((RemoteWebDriver)Driver).SessionId}");
                }
            }
            Driver.Quit();
        }
    }
}
```

We first check to see if our tests are running against Sauce Labs. If so we check what the test result was and store the boolean result in a local variable. We then use Selenium's JavaScript Executor (just like with the test name) to pass the test result onto Sauce Labs. After that we output the URL for the Sauce Labs job to the console.

Now when we run our tests in Sauce Labs and navigate to the Sauce Labs Account dashboard, we will see our tests running like before. But now there will be a proper test status when they finish (e.g., `Pass` or `Fail`) and we'll see the URL for the job in the console output as well. This enables us to easily jump to a specific job in Sauce Labs if we want to.

Part 4: Sauce Connect

There are various ways that companies make their pre-production application available for testing. Some use an obscure public URL and protect it with some form of authentication (e.g., Basic Auth, or certificate based authentication). Others keep it behind their firewall. For those that stay behind a firewall, Sauce Labs has you covered.

They have a program called Sauce Connect Proxy that creates a secure tunnel between your machine and their private cloud. With it you can run tests in Sauce Labs and test applications that are only available on your private network.

To use Sauce Connect you need to download and run it. There's a copy for each operating system -- get yours here and run it from the command-line. In the context of our existing test code let's download Sauce Connect, unzip its contents, and store it in our `vendor` directory.

```
PageObjects
   BasePage.cs
   DynamicLoadingPage.cs
   LoginPage.cs
Tests
   BaseTest.cs
   DynamicLoadingTest.cs
   LoginTest.cs
Vendor
    chromedriver
    geckodriver
    sc
```

Now we just need to launch the application while specifying our Sauce account credentials.

```
vendor/sc -u $SAUCE_USERNAME -k $SAUCE_ACCESS_KEY
// ...
Starting Selenium listener...
Establishing secure TLS connection to tunnel...
Selenium listener started on port 4445.
Sauce Connect is up, you may start your tests.
```

Now that the tunnel is established, we could run our tests against a local instance of our application (e.g., the-internet). Assuming the application was set up and running on our local machine, we run our tests against it by specifying a different base URL at runtime (e.g., `BASE_URL=http://localhost:4567 dotnet test`) and they would work.

To see the status of the tunnel, we can view it on the tunnel page of the account dashboard. To shut the tunnel down, we can do it manually from this page. Or we can issue a `Ctrl+C` command to the terminal window where it's running.

When the tunnel is closing, here's what you'll see.

```
Got signal 2
Cleaning up.
Removing tunnel 21ff9664b06c4edaa4bd573cdc1fbac1.
All jobs using tunnel have finished.
Waiting for the connection to terminate...
Connection closed (8).
Goodbye.
```

Chapter 14
Speeding Up Your Test Runs

We've made huge strides by leveraging page objects, a base page object, explicit waits, and connecting our tests to Sauce Labs. But we're not done yet. Our tests still take a good deal of time to run since they're executing in series (e.g., one after another). As our suite grows this slowness will grow with it.

With parallelization we can easily remedy this pain before it becomes acute by executing multiple tests at the same time. And with what's built into NUnit it's extremely simple to do.

Configuration

For each test class we need to add an attribute to it that denotes that it should be run in parallel.

```
// filename: Tests/LoginTest.cs
// ...
namespace Tests
{
[TestFixture]
[Parallelizable]
    class LoginTest : BaseTest
    {
// ...
```

```
// filename: Tests/DynamicLoadingTest.cs
// ...
namespace Tests
{
[TestFixture]
[Parallelizable]
    class DynamicLoadingTest : BaseTest
    {
// ...
```

Adding the `[Parallelizable]` attribute before the class declaration for each test is all that's needed. When we save everything and run our tests from within Visual Studio we'll see multiple browsers launch at the same time -- completing a run through all of the tests at a faster rate than before.

One Drawback

Just because we have 4 tests does not mean we will have 4 parallel threads running at the same time. In NUnit parallel execution is only supported at the fixture (a.k.a. test class) level.

So instead of 4 threads, we'll get 2 (one for `LoginTest.cs` and another for `DynamicLoadingTest.cs`). A single thread will execute all of the test methods within a fixture one-at-a-time (rather than all at once).

For more details on parallel execution in NUnit check out the documentation here.

Chapter 15

Flexible Test Execution

In order to get the most out of our tests we'll want a way to break them up into relevant, targeted chunks. Running tests in smaller groupings like this (along with parallel execution) will help keep test run times to a minimum and help enhance the amount of feedback you get in a timely fashion.

With NUnit's Category Attribute we're able to easily achieve test grouping. It's a simple matter of adding the Category Attribute to either a test method or test class along with a helpful name.

Let's step how to set this up.

Specifying Categories

```
// filename: Tests/LoginTest.cs
// ...
[Test]
[Category("Shallow")]
        public void ValidAccount()
        {
            Login.With("tomsmith", "SuperSecretPassword!");
            Assert.That(Login.SuccessMessagePresent);
        }

[Test]
[Category("Deep")]
        public void BadPasswordProvided()
        {
            Login.With("tomsmith", "bad password");
            Assert.That(Login.FailureMessagePresent);
        }
// ...
```

```
// filename: Tests/DynamicLoadingTest.cs
// ...
[TestFixture]
[Parallelizable]
[Category("Deep")]
    class DynamicLoadingTest : BaseTest
// ...
```

In both of our test files we've applied two different categories, `"Shallow"` and `"Deep"`. "Shallow" tests are roughly equivalent to "smoke" or "sanity" tests. These should pass before you can consider running other tests which aren't as mission critical and may take longer to run (e.g., "Deep").

Categories are powerful since they can be applied across different test files, enabling you to create a dynamic grouping of tests at runtime.

Running Categories

With `dotnet test` we can specify which category to launch at runtime.

To run a specific test category we need to append an additional argument when running the tests, like so:

```
dotnet test --filter TestCategory=category name
```

Here are the available execution commands given for our current categories:

```
dotnet test --filter TestCategory=Shallow
dotnet test --filter TestCategory=Deep
```

For more info on this functionality and other available options, check out the NUnit Category documentation and the dotnet Test Selection Documentation.

Chapter 16

Automating Your Test Runs

You'll probably get a lot of mileage out of your test suite in its current form if you just run things from your computer, look at the results, and tell people when there are issues. But that only helps you solve part of the problem.

The real goal in test automation is to find issues reliably, quickly, and automatically. We've built things to be reliable and quick. Now we need to make them run on their own, and ideally, in sync with the development workflow you are a part of.

To do that we need to use a Continuous Integration server.

A Continuous Integration Server Primer

A Continuous Integration server (a.k.a. CI) is responsible for merging code that is actively being developed into a central place (e.g., "trunk", "head", or "master") frequently (e.g., several times a day, or on every code commit) to find issues early so they can be addressed quickly — all for the sake of releasing working software in a timely fashion.

With CI we can automate our test runs so they can happen as part of the development workflow. The lion's share of tests that are typically run on a CI Server are unit (and potentially integration) tests. But we can very easily add in our Selenium tests too.

There are numerous CI Servers available for use today, most notably:

- Bamboo
- CircleCI
- Jenkins
- TravisCI

Let's pick one and step through an example.

A CI Example

Jenkins is a fully functional, widely adopted, free and open-source CI server. Its a great candidate for us to try.

Lets start by setting it up on the same machine as our test code. Keep in mind that this isn't the "proper" way to go about this — its merely beneficial for this example. To do it right, the Jenkins server (e.g., master node) would live on a machine of its own.

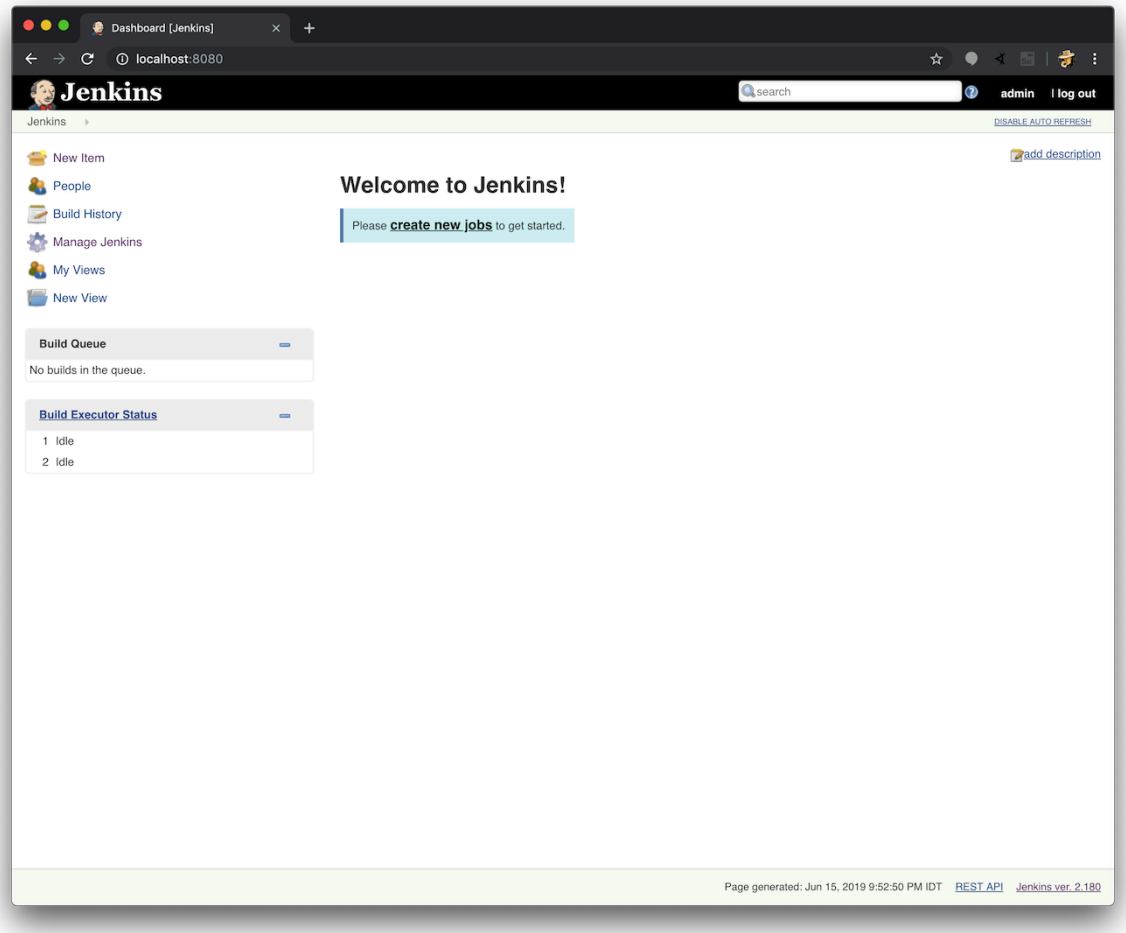
Part 1: Quick Setup

A simple way to get started is to grab the latest Jenkins war file. You can grab it from the Jenkins download page.

Once downloaded, launch it from the command-line and follow the setup steps provided.

```
> java -jar jenkins.war
// ...
hudson.WebAppMain$3 run
INFO: Jenkins is fully up and running
```

You will now be able to use Jenkins by visiting http://localhost:8080/ in your browser.

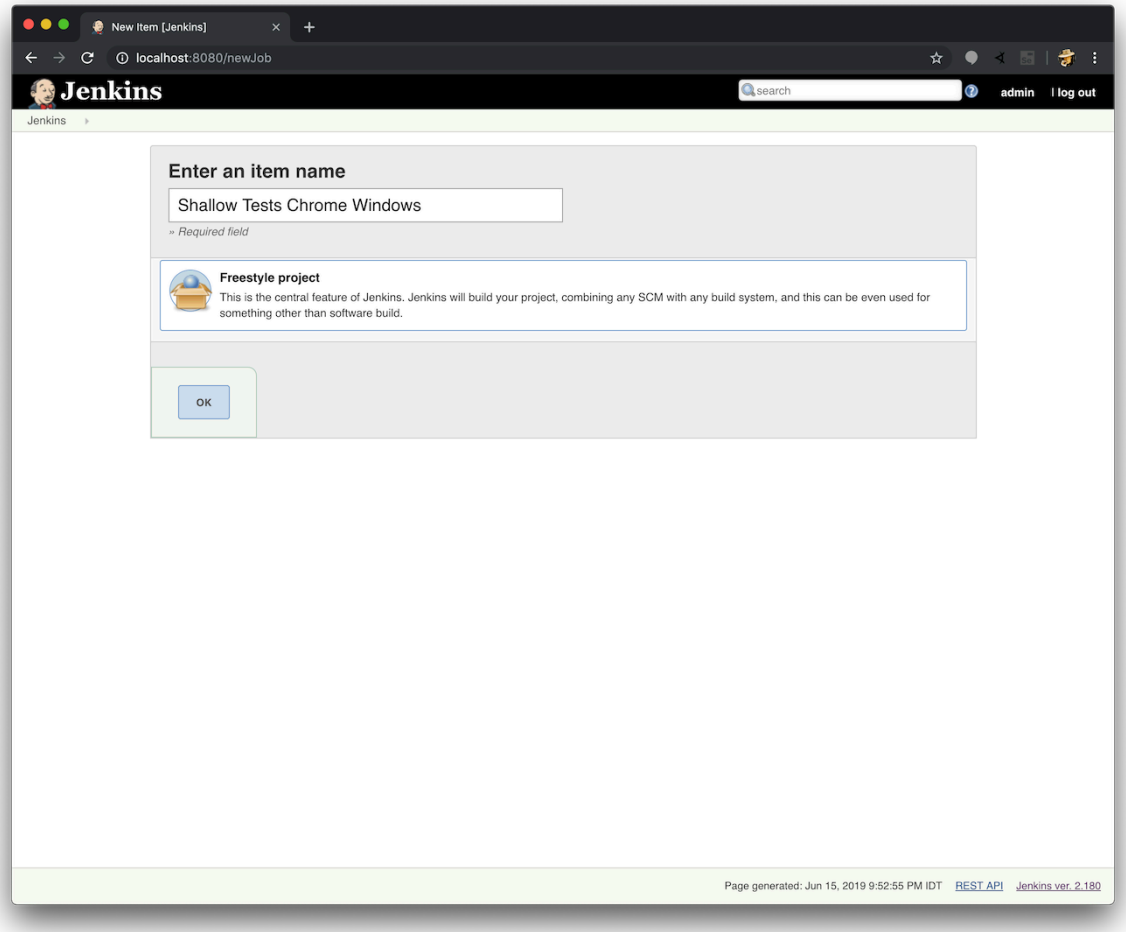


NOTE: Before moving to the next step, click **`ENABLE AUTO-REFRESH`** at the top right-hand side of the page. Otherwise you'll need to manually refresh the page (e.g., when running a job and waiting for results to appear).

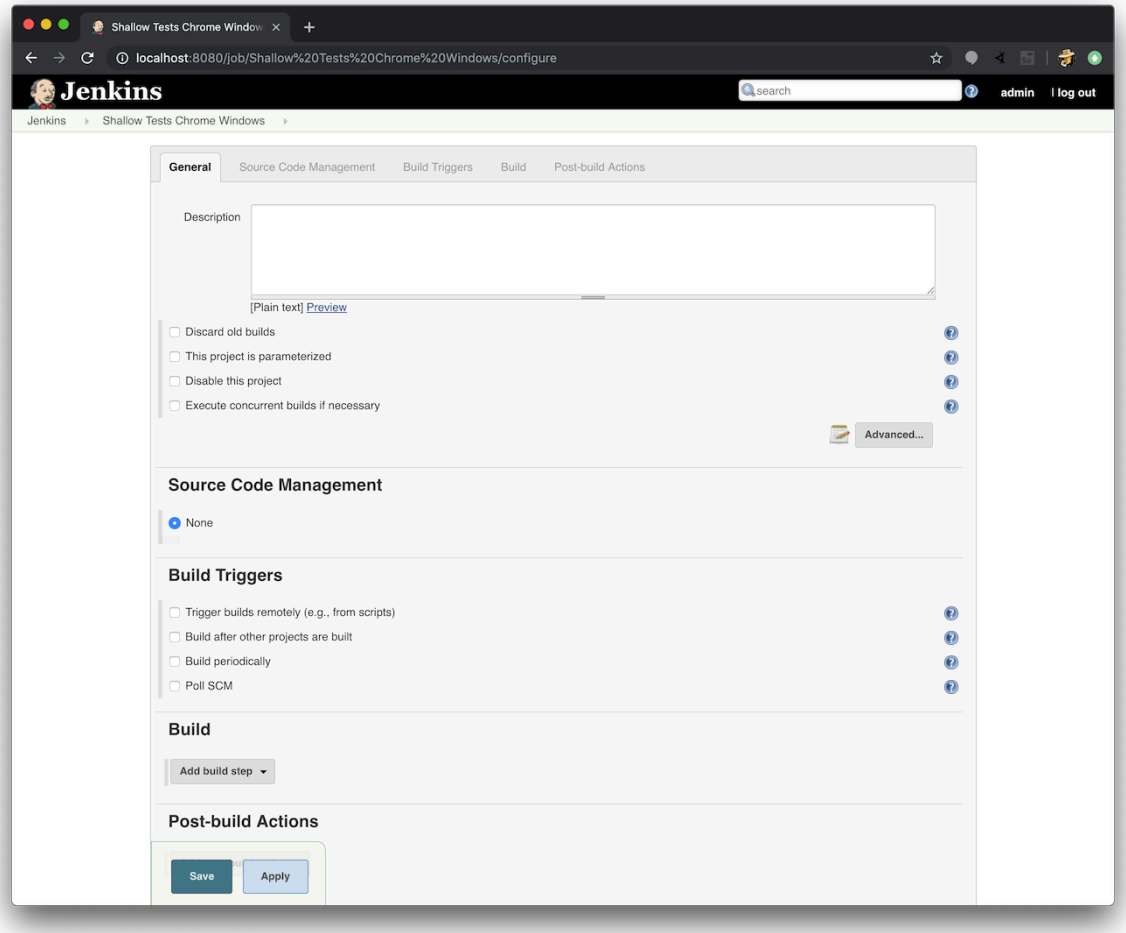
Part 2: Job Creation And Configuration

Now that Jenkins is loaded in the browser, let's create a Job and configure it to run our `shallow` tests against Chrome on Windows 10.

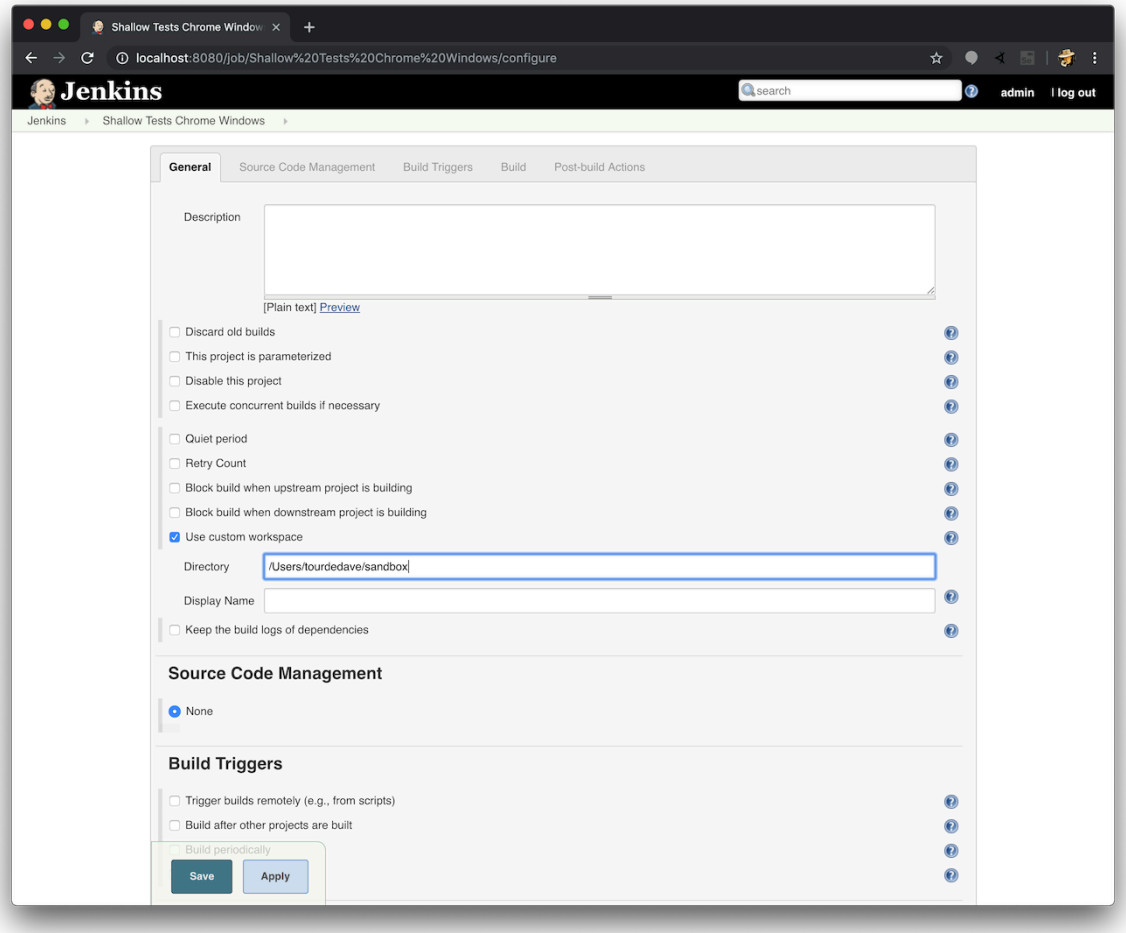
- Click `New Item` from the top-left of the Dashboard
- Give it a name (e.g., `Shallow Tests Chrome 50 Windows 10`)
- Select the `Freestyle project` option
- Click `OK`



This will load a configuration screen for the Jenkins job.

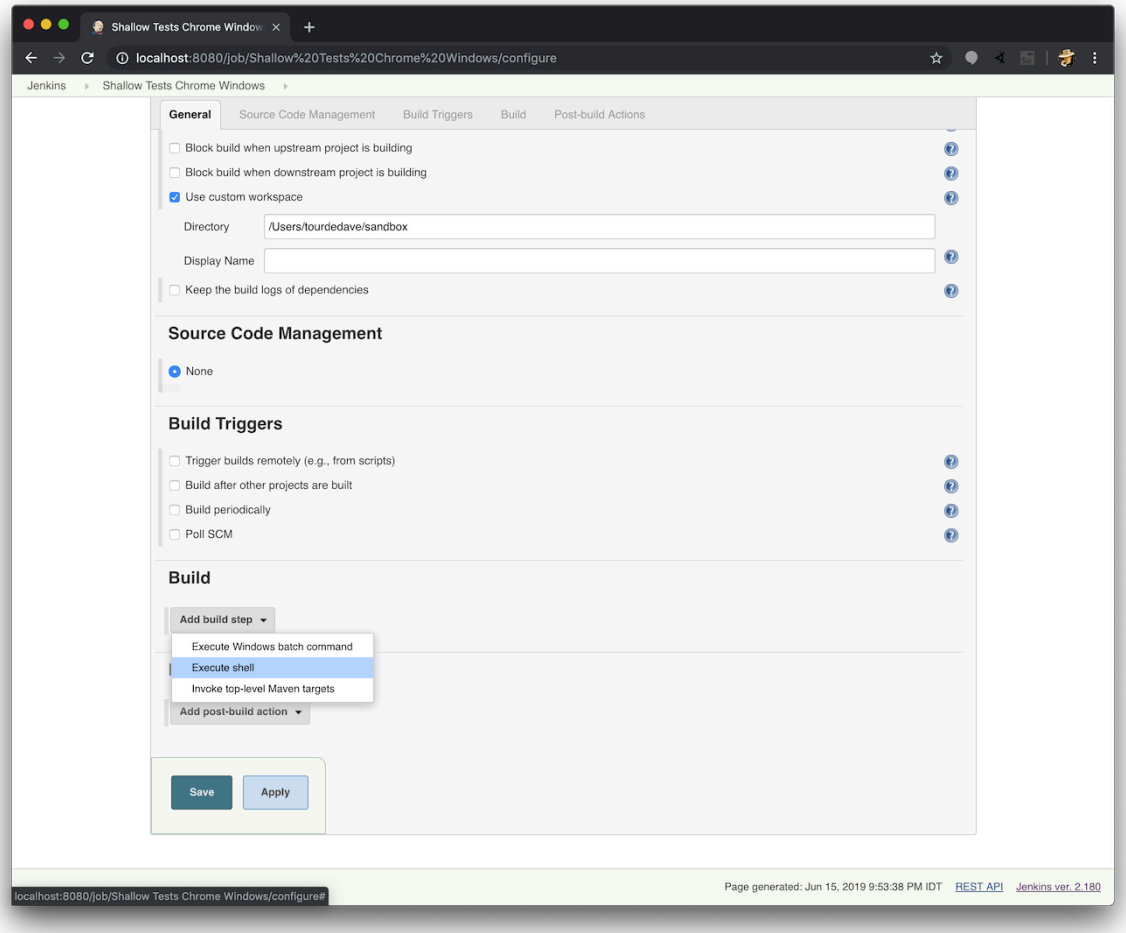


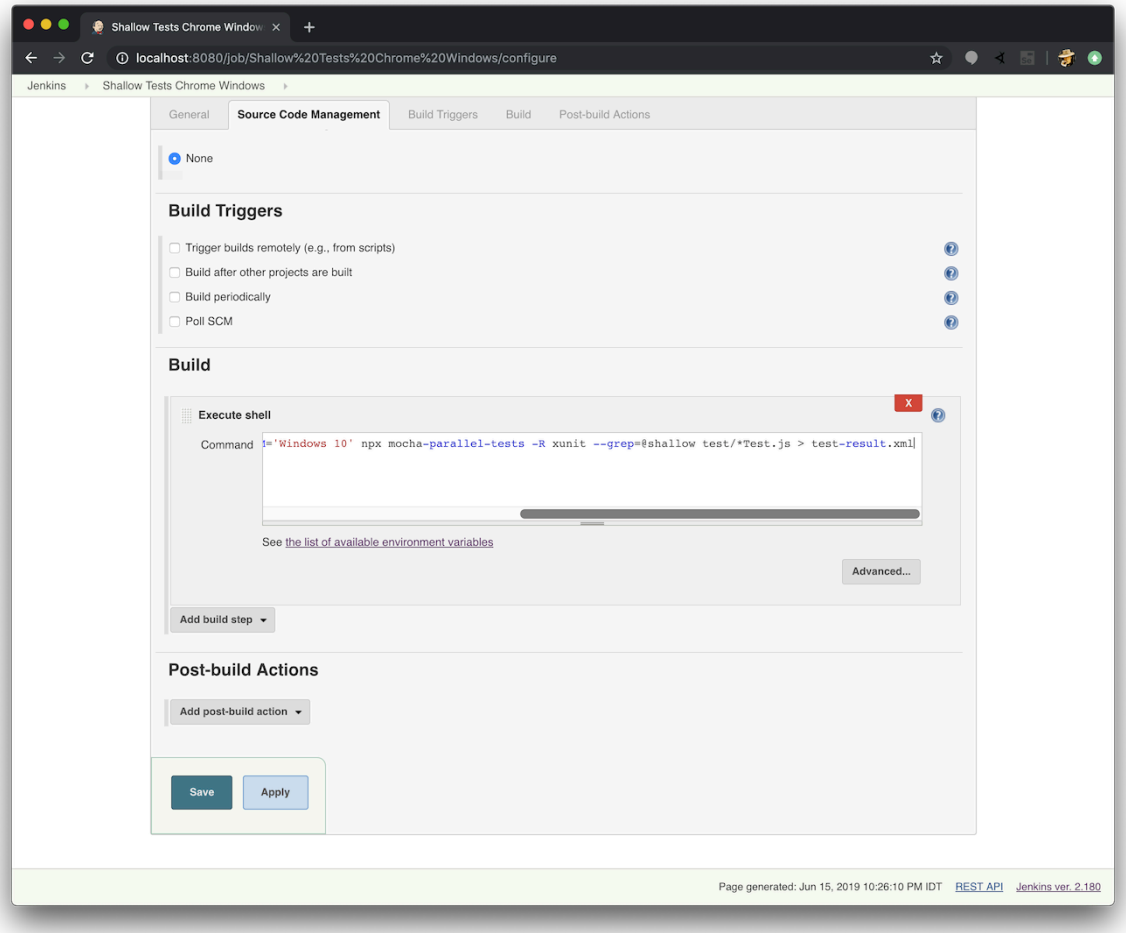
- In the `Advanced Project Options` section select the `Advanced` button
- Choose the checkbox for `Use custom workspace`
- Provide the full path to your test code
- Leave the `Display Name` field blank



NOTE: Ideally, your test code would live in a version control system and you would configure your job (under **`Source Code Management`**) to pull it in and run it. To use this approach you may need to install a plugin to handle it. For more info on plugins in Jenkins, go here.

- Scroll down to the `Build` section and select `Add build step`
- Select `Execute shell`
- Specify the commands needed to launch the tests


Shallow Tests Chrome Windows
localhost:8080/job/Shallow%20Tests%20Chrome%20Windows/configure
Jenkins
Shallow Tests Chrome Windows
General
Source Code Management
Build Triggers
Build
Post-build Actions
Block build when upstream project is building
Block build when downstream project is building
Use custom workspace
Directory
/Users/tourdedave/sandbox
Display Name
Keep the build logs of dependencies
Source Code Management
None
Build Triggers
Trigger builds remotely (e.g., from scripts)
Build after other projects are built
Build periodically
Poll SCM
Build
Add build step
Execute Windows batch command
Execute shell
Invoke top-level Maven targets
Add post-build action
Save
Apply
Page generated: Jun 15, 2019 9:53:38 PM IDT
REST API
Jenkins ver. 2.180
localhost:8080/job/Shallow Tests Chrome Windows/configure#




```
dotnet test --filter TestCategory=Shallow --test-adapter-path:.
--logger:"nunit;LogFilePath=test-result.xml"
```

The `--test-adapter-path` and `--logger` arguments specify the reporter we'd like to use for the test results, and where we'd like to output them -- in an XML report that Jenkins expects.

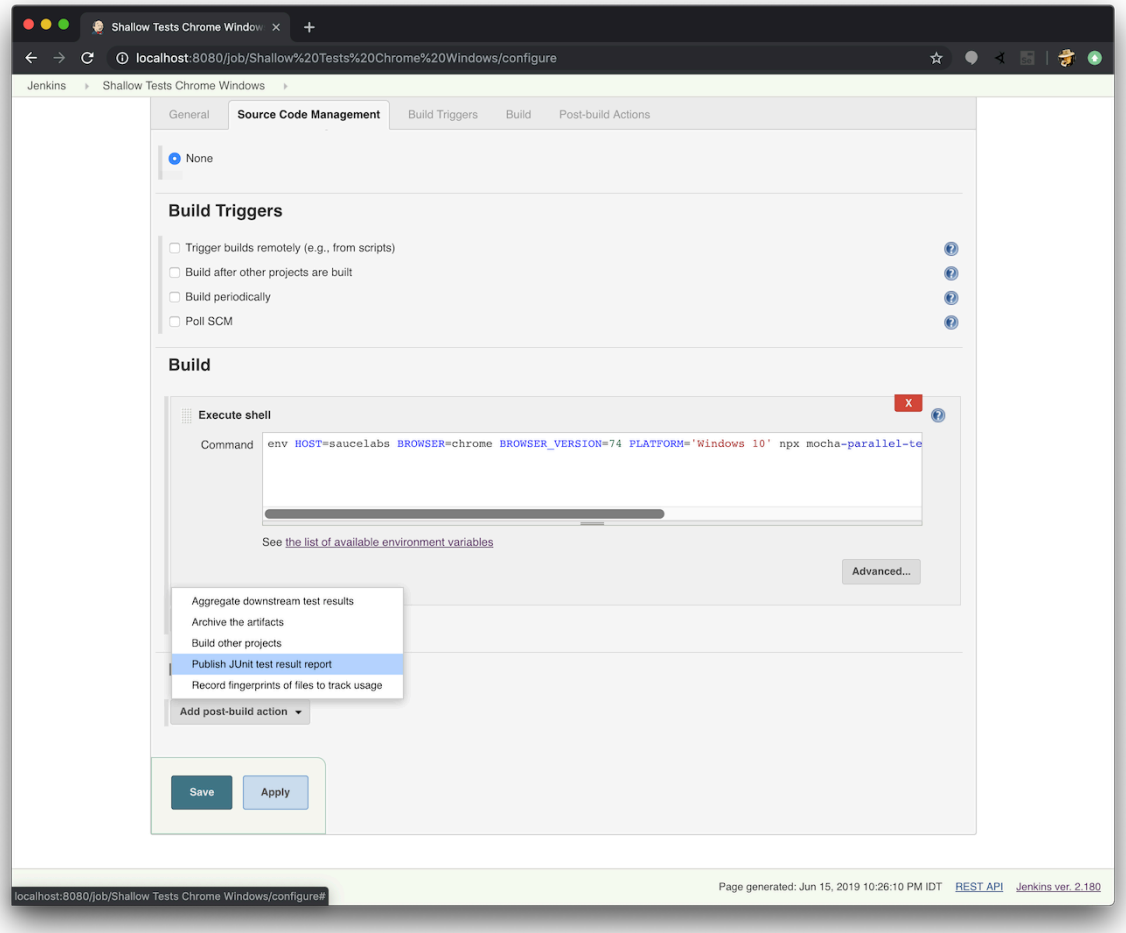
Before we run the command we'll need to install the NUnit XML log adapter from Nuget.

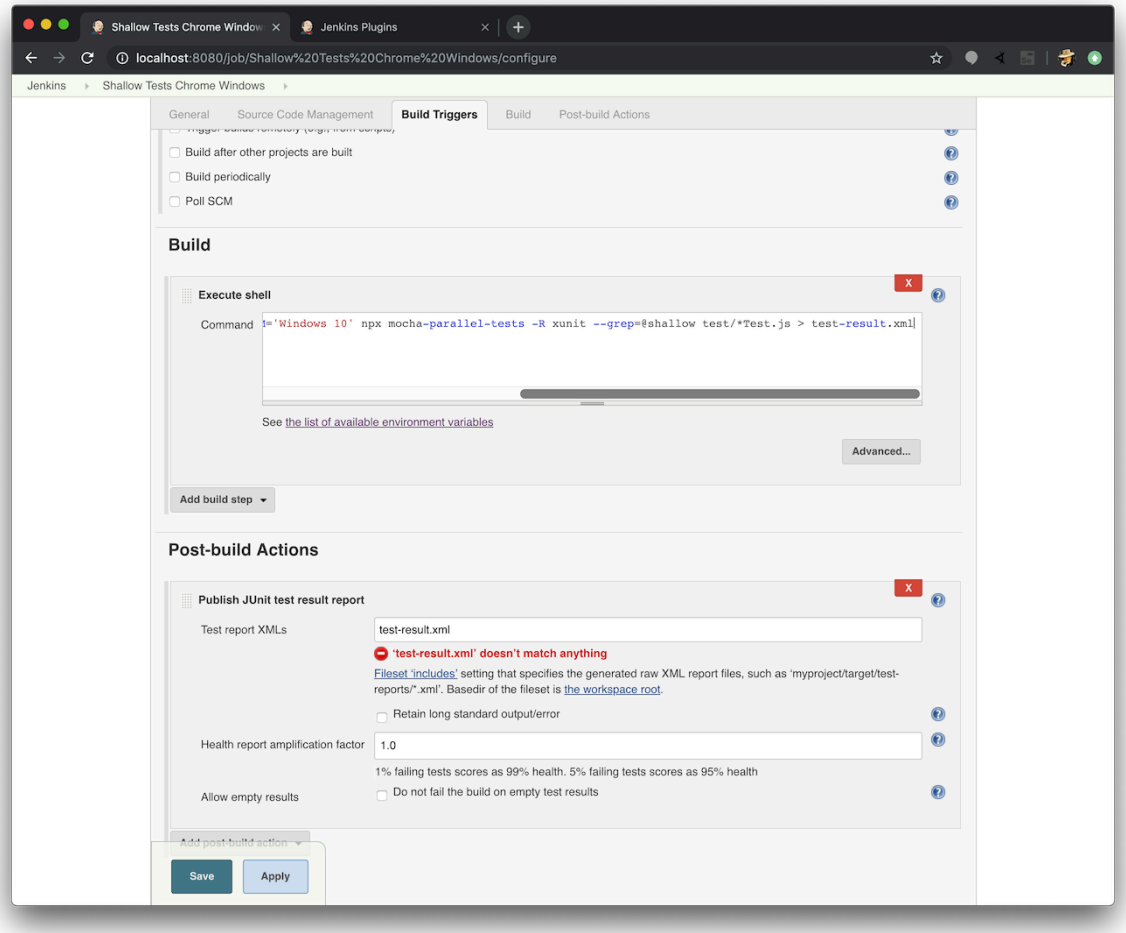
```
dotnet add package NunitXml.TestLogger
```

Now let's hop back over to Jenkins and configure the job to consume the test results.

- Under `Post-build Actions` select `Add post build action`
- Select `Publish JUnit test result report`
- Add the name of the result file specified in the command -- `test-result.xml`
- Click `Save`

NOTE: If this post build action isn't available to you, you will need to install the JUnit Jenkins plugin.


Shallow Tests Chrome Window
localhost:8080/job/Shallow%20Tests%20Chrome%20Windows/configure
Jenkins
Shallow Tests Chrome Windows
General
Source Code Management
Build Triggers
Build
Post-build Actions
None
Build Triggers
Trigger builds remotely (e.g., from scripts)
Build after other projects are built
Build periodically
Poll SCM
Build
Execute shell
Command
env HOST=saucelabs BROWSER=chrome BROWSER_VERSION=74 PLATFORM='Windows 10' npx mocha-parallel-te
See the list of available environment variables
Advanced...
Aggregate downstream test results
Archive the artifacts
Build other projects
Publish JUnit test result report
Record fingerprints of files to track usage
Add post-build action
Save
Apply
Page generated: Jun 15, 2019 10:26:10 PM IDT
REST API
Jenkins ver. 2.180
localhost:8080/job/Shallow Tests Chrome Windows/configure#



Now our tests are ready to be run, but before we do, let's go ahead and add a failing test so we can demonstrate the test report.

Part 3: Force A Failure

Let's add a new test method to `LoginTest.cs` that will fail every time we run it.

```
// filename: Test/LoginTest.cs
// ...
[Test]
[Category("Shallow")]
        public void ForcedFailure()
        {
            Login.With("tomsmith", "bad password");
            Assert.That(Login.SuccessMessagePresent);
        }
    }
}
```

One more thing we'll want to do is update how we're outputting the Sauce Labs job URL when there's a test failure. Right now we're outputting it to the console, but with the XML report

generation this information may be hard to find in our Jenkins job. So let's make sure it shows up in the stack trace, and ultimately, the final test result report.

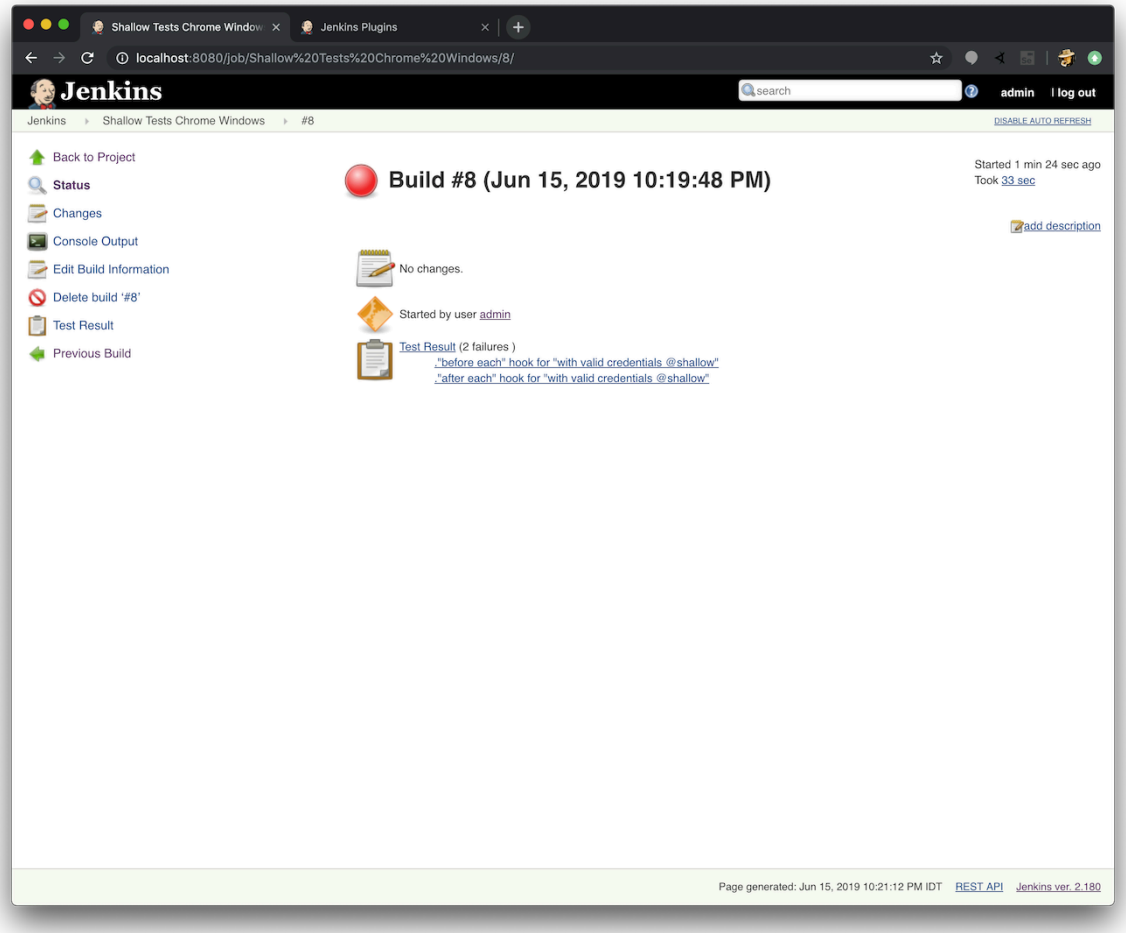
```
// filename: Test/BaseTest.cs
// ...
[TearDown]
        protected void TearDown()
        {
            try
            {
                if (Host.Equals("saucelabs"))
                {
                  var testName = TestContext.CurrentContext.Test.Name;
                  bool testPassed = TestContext.CurrentContext.Result.Outcome.Status.
ToString() == "Passed";
                    ((IJavaScriptExecutor)Driver).ExecuteScript("sauce:job-name=" +
testName);
                    ((IJavaScriptExecutor)Driver).ExecuteScript("sauce:job-result=" + (
testPassed ? "passed" : "failed"));
                    if(!testPassed)
                    {
                        throw new System.Exception("https://saucelabs.com/tests/" + ((
RemoteWebDriver)Driver).SessionId);
                    }
                }
            }
            finally
            {
                Driver.Quit();
            }
        }
    }
}
```

In the `TearDown` method of our BaseTest we make it so we throw an exception with the Sauce Labs job URL when there's a test failure and ensure that `Driver.Quit()` is called regardless. This change will make the job URL show up in a relevant spot in the XML test report.

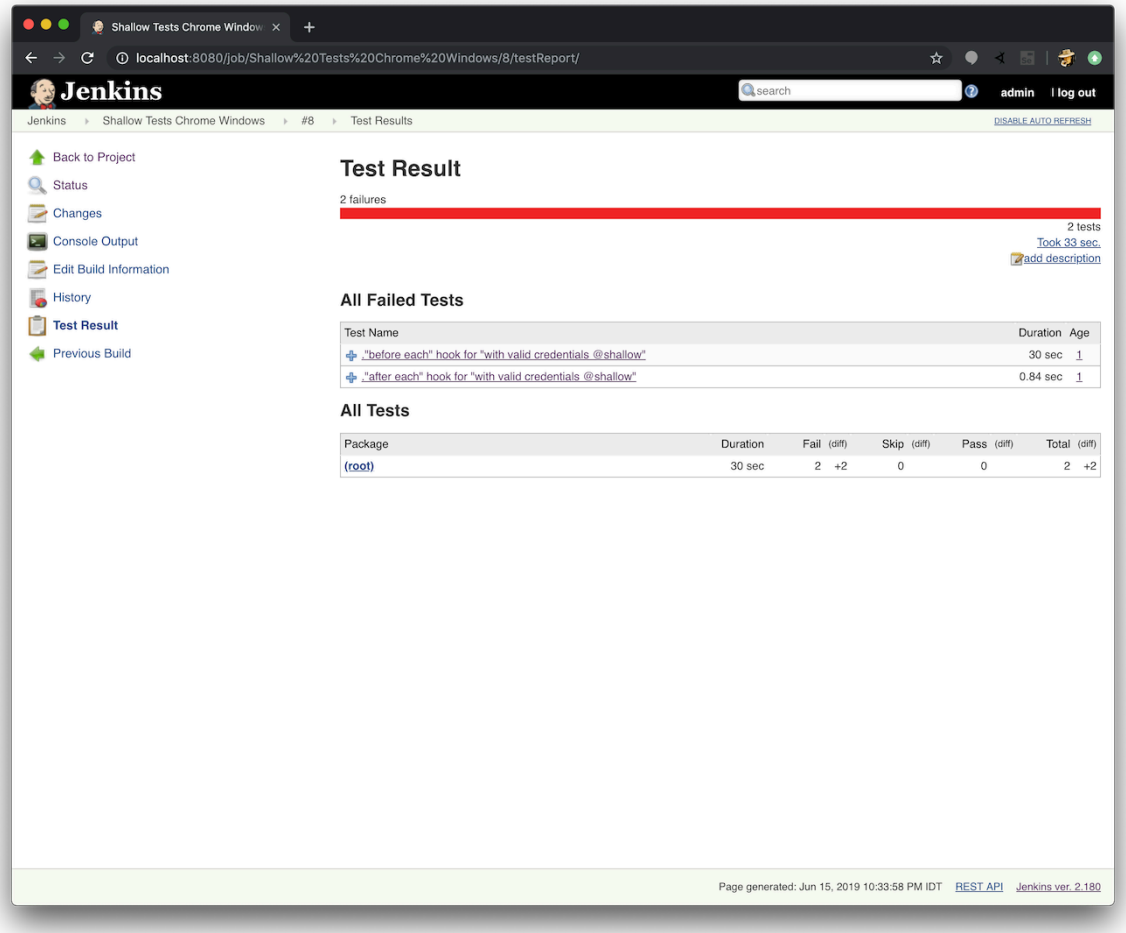
Now let's run our Jenkins job by clicking `Build Now` from the left-hand side of the screen.

NOTE: You can peer behind the scenes of a job while it's running (and after it completes) by clicking on the build you want from **`Build History`** and selecting **`Console Output`**. This output will be your best bet in tracking down an unexpected result.

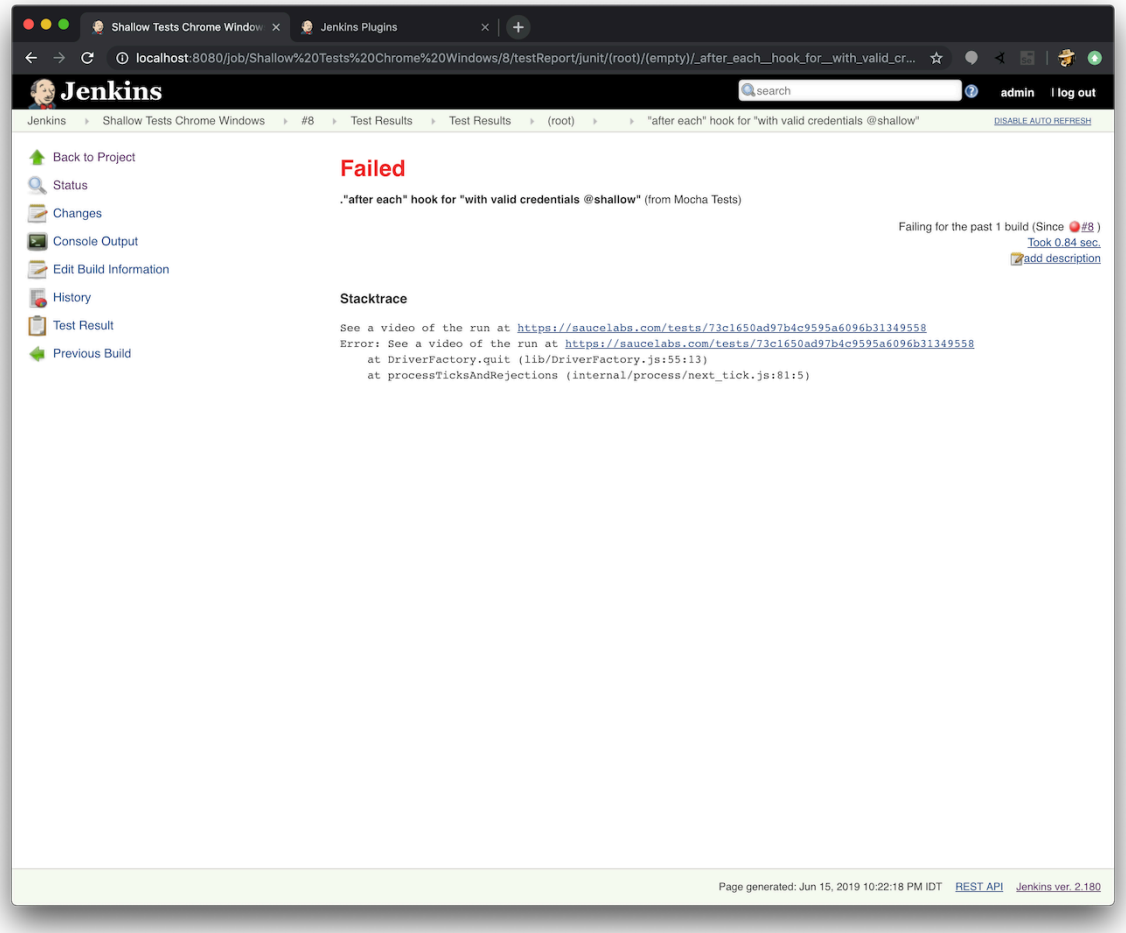
When the test completes, it will be marked as failed.



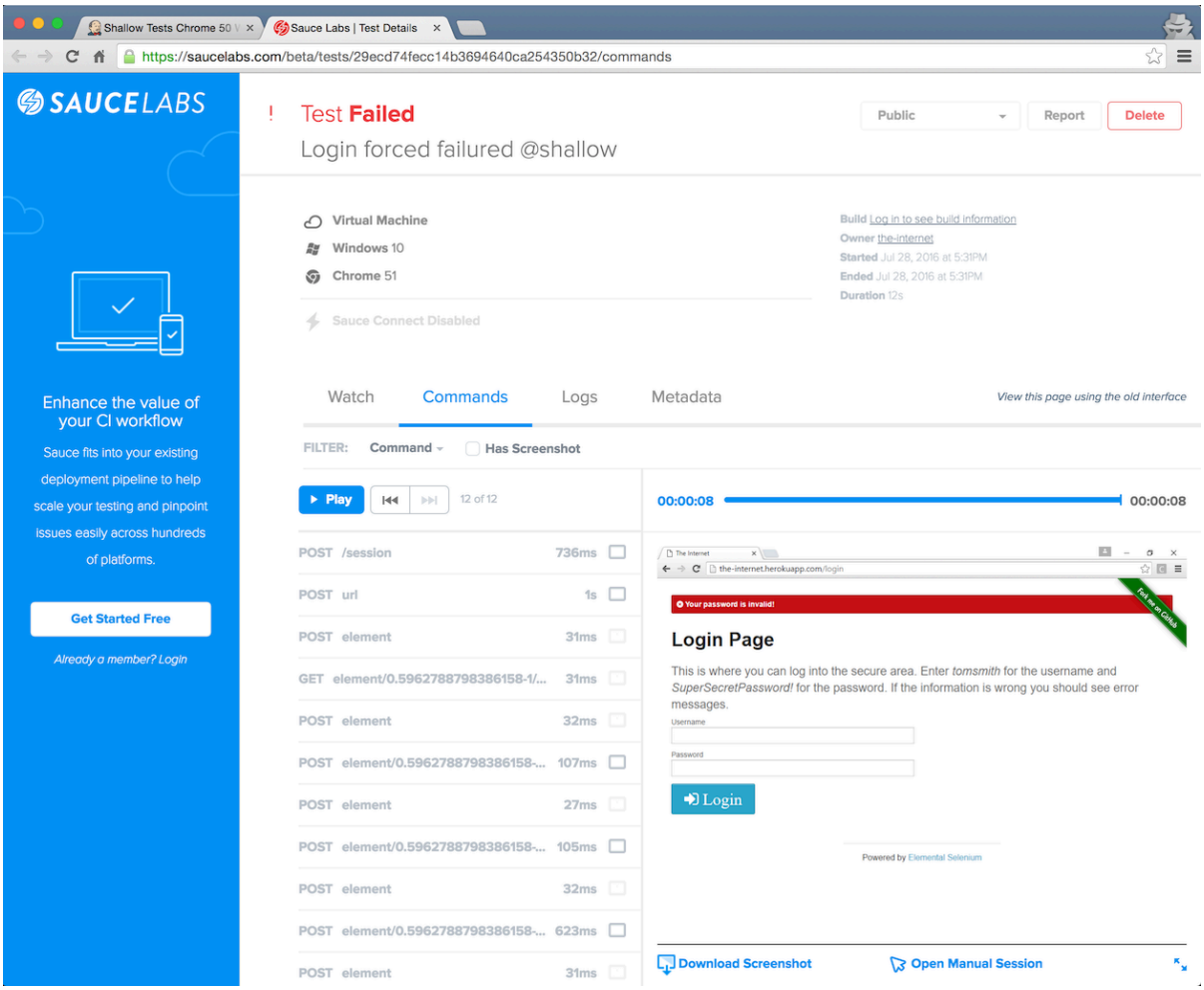
When we click on `Latest Test Result` we can see the test that failed (e.g., `Login.forced failure @shallow`). The other failure listed (e.g., `."after each" hook for "forced failure @shallow"`) is from the teardown of our test. It contains the Sauce Labs job URL.



If we click on the failed test we can see the stack trace from the test failure. If we click on the failure from the teardown we can see the URL to the job in Sauce Labs.



When we follow the URL to the Sauce Labs job we're able to see what happened during the test run (e.g., we can replay a video of the test, see what Selenium commands were issued, etc.).

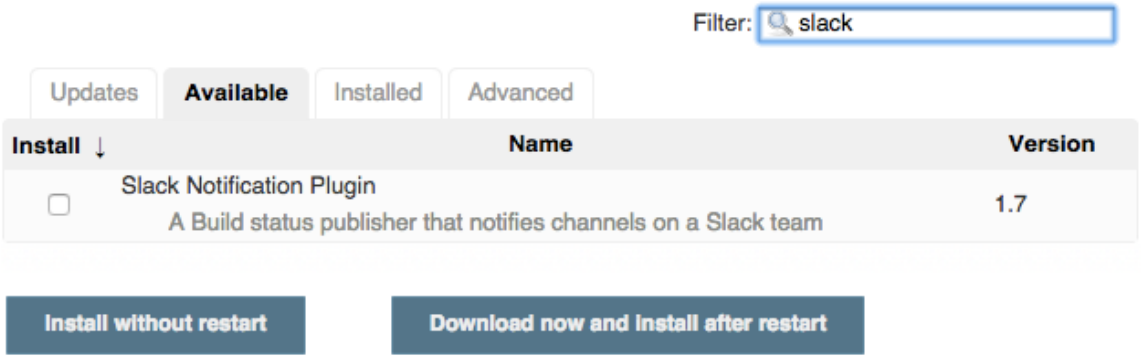


Notifications

In order to maximize your CI effectiveness, you'll want to send out notifications to alert your team members when there's a failure.

There are numerous ways to go about this (e.g., e-mail, chat, text, co-located visual cues, etc). Thankfully there are numerous, freely available plugins that can help facilitate whichever method you want. You can find out more about Jenkins' plugins here.

For instance, if you wanted to use chat notifications and you use a service like Slack, you would do a plugin search:



After installing the plugin, you will need to provide the necessary information to configure it (e.g., an authorization token, the channel/chat room where you want notifications to go, what kinds of notifications you want sent, etc.) and then add it as a `Post-build Action` to your job (or jobs).

After installing and configuring a plugin, when your CI job runs and fails, a notification will be sent to the chat room you configured.

Ideal Workflow

In the last chapter we covered test grouping with categories and applied some preliminary ones to our tests (e.g., "Shallow" and "Deep"). These categories are perfect for setting up an initial acceptance test automation workflow.

To start the workflow we'll want to identify a triggering event. Something like a CI job for unit or integration tests that the developers on your team use. Whenever that runs and passes, we can trigger our "Shallow" test job to run (e.g., our smoke or sanity tests). If the job passes then we can trigger a job for "Deep" tests to run. Assuming that passes, we can consider the code ready to be promoted to the next phase of release (e.g., manual testing, push to a staging, etc.) and send out a relevant notification to the team.

NOTE: You may need to incorporate a deployment action as a preliminary step before your "Shallow" and "Deep" jobs can be run (to make sure your tests have an environment available to be run against). Consult a developer on your team for help if that's the case.

Outro

By using a CI Server you're able to put your tests to work by using computers for what they're good at -- automation. This frees you up to focus on more important things. But keep in mind that there are numerous ways to configure your CI server. Be sure to tune it to what works best for you and your team. It's well worth the effort.

Chapter 17
Finding Information On Your Own

There is information all around us when it comes to Selenium. But it it can be challenging to sift through it, or know where to look.

Here is a list breaking down a majority of the Selenium resources available, and what they're useful for.

Documentation & Tips

- Selenium HQ

This is the official Selenium project documentation site. It's a bit dated, but there is loads of helpful information here. You just have to get the hang of how the navigate site to find what you need.

- The Selenium Wiki

This is where all the good stuff is -- mainly, documentation about the various language bindings and browser drivers. If you're not already familiar with it, take a look.

- Elemental Selenium Archives

Every tip I've written is freely available on the tips archive page. There are over 70 different Selenium problems and solutions covered. They're in Ruby, but the code has been open-sourced with a fair number of them being ported into other programming languages. You can find the code for them here.

Blogs

- The official Selenium blog

This is where news of the Selenium project gets announced, and there's also the occasional round-up of what's going on in the tech space (as it relates to testing). Definitely worth a look.

- A list of "all" Selenium WebDriver blogs

At some point, someone rounded up a large list of blogs from Selenium practitioners and committers. It's a pretty good list.

Other Books

- Selenium Testing Tools Cookbook

This book outlines some great ways to leverage Selenium. It's clear that Gundecha has a very pragmatic approach that will yield great results.

- Selenium Design Patterns and Best Practices

Dima Kovalenko's book covers useful tactics and strategies for successful test automation with Selenium. I was a technical reviewer for the book and think it's a tremendous resource. The book covers Ruby, but he has ported the examples to Java. You can find them here.

Meetups

- All Selenium Meetups listed on Meetup.com

A listing of all in-person Selenium Meetups are available on Meetup.com. If you're near a major city, odds are there's one waiting for you.

- How to start your own Selenium Meetup

If there's not a Selenium Meetup near you, start one! Sauce Labs has a great write up on how to do it.

Conferences

- Selenium Conf

This is the official conference of the Selenium project where practitioners and committers gather and share their latest knowledge and experiences with testing. There are two conferences a year, with the location changing every time (e.g., it's been in San Francisco, London, Boston, Bangalore, Portland, Austin, Berlin, Chicago, and Tokyo).

- Selenium Camp

This is an annual Selenium conference in Eastern Europe (in Kiev, Ukraine) organized by the folks at XP Injection. It's a terrific conference. If you can make the trip, I highly recommend it.

- List of other testing conferences

A helpful website that lists all of the testing conferences out there.

Videos

- Selenium Conference Talks

All of the talks from The Selenium Conference are recorded and made freely available online. This is a tremendous resource.

- Selenium Meetup Talks

Some of the Selenium Meetups make it a point to record their talks and publish them afterwards. Here are some of them. They are a great way to see what other people are doing and pick up some new tips.

Mailing Lists

- Selenium Developers List

This is where developers discuss changes to the Selenium project, both technically and administratively.

- Selenium Users Google Group
- Selenium LinkedIn Users Group

The signal to noise ratio in these groups can be challenging at times. But you can occasionally find some answers to your questions.

Forums

- Stack Overflow
- Quora
- Reddit

These are the usual forums where you can go looking for answers to questions you're facing (in addition to the mailing lists above).

Issues

- Selenium Issue Tracker

If you're running into a specific and repeatable issue that just doesn't make sense, you may have found a bug in Selenium. You'll want to check the Selenium Issue Tracker to see if it has already been reported. If not, then create a new issue -- assuming you're able to provide a short and self-contained example that reproduces the problem.

This is known as SSCCE (a Short, Self Contained, Correct (Compilable), Example). For a

tongue-in-cheek take on the topic, see this post.

Chatting With the Selenium Community

The Selenium Chat Channel is arguably the best way to connect with the Selenium community and get questions answered. This is where committers and practitioners hang out day-in and day-out.

You can connect either through Slack or IRC. Details on how to connect are available here.

Once connected, feel free to say hello and introduce yourself. But more importantly, ask your question. If it looks like no one is chatting, ask it anyway. Someone will see it and eventually respond. They always do. In order to get your answer, you'll probably need to hang around for a bit. But the benefit of being a fly on the wall is that you gain insight into other problems people face, possible solutions, and the current state of the Selenium project and its various pieces.

Chapter 18
Now You Are Ready

The journey for doing Selenium successfully can be long and arduous. But by adhering to the principals in this book, you will avoid a majority of the pitfalls around you. You're also in a better position now -- armed with all of the information necessary to continue your Selenium journey.

You are ready. Keep going, and best of luck!

If you have any questions, feedback, or want help -- get in touch!

Cheers,
Dave H