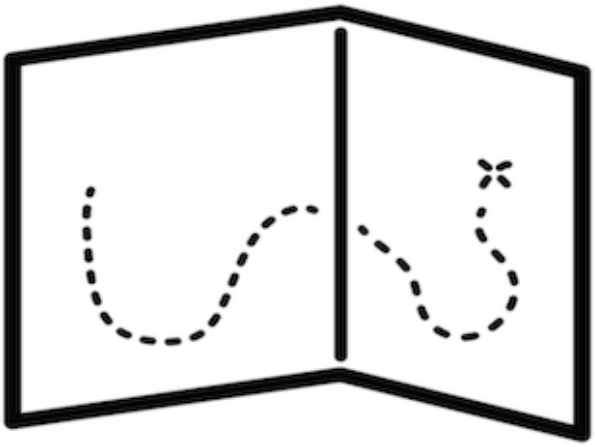
How to use Selenium, successfully



The Selenium Guidebook

by Dave Haeffner

Preface

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., 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, with a number of major programming languages, 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 support from Microsoft!).

Selenium can be run on your local computer, on a server (with Selenium Remote), on your own 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 plenty of gotchas to watch out for when you get into it. But don't worry, I'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 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), 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 somethings not there, add it to the bottom of the backlog. If it is there, make a note in the backlog item that this 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 the 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 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 development), then your progress will be slow and you'll likely end up asking for more developer 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 growing a test framework (a.k.a. a test harness).

As you are considering which language to go with, consider what open source frameworks already exist for the languages you are considering. Going with one will save you a lot of time and give you a host of functionality out of the box that you would otherwise have to build and maintain 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 (or looking to port your tests) 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 Java with JUnit.

Chapter 4

A Programming Primer

This section will prime you with just enough programming concepts (and how they pertain to testing) 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.

Installing Third-Party Libraries

There are numerous third-party libraries available in Java that can help you quickly add functionality to your test code. To handle installing them (and their dependencies) you should use some form of dependency management.

The examples in this book use Maven to manage third-party libraries.

Choosing An IDE (Integrated Development Environment)

Java is a vast and picky language. In order to write test code quickly and effectively (and to avoid absolute frustration), you'll want to use an IDE.

The two most prominent options available are Eclipse and IntelliJ IDEA.

The examples in this book were written using IntelliJ IDEA Community Edition (but either IDE will work just fine).

Installation

Here are some installation instructions to help you get started quickly.

- Linux
- OSX
- Windows

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 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 to be an effective test automator right now.

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

- Object Structures (Variables, Methods, and Classes)
- Access Modifiers (pubic, protected, private)
- Object Types (Strings, Integers, Booleans, etc.)
- Actions (Assertions and Conditionals)
- Annotations
- 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 case-sensitive
- must not be a keyword (or reserved word) in Java
- starts with a letter

If the variable name is one word, it should be all lowercase. If it's more than one word it should be CamelCase (e.g., `exampleVariable`).

You can store a value in a variable by using an equals sign (e.g., `=`). But you'll only be able to store a value of the type you specified when creating it.

```
String exampleVariable1 = "string value";
System.out.println(exampleVariable1.getClass());
// outputs: class java.lang.String

Integer exampleVariable2 = 42;
System.out.println(exampleVariable2.getClass());
// outputs: class java.lang.Integer
```

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

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

```
String pageTitle = driver.getTitle();
```

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. If you don't want to return anything, specify the return type as `void`.

Method naming follows similar conventions to variables. The main difference is that they tend to be a verb (since they denote some kind of an action to be performed). Also, the contents (e.g., the body) 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) {
  System.out.println(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:

- must match the name of the file they're stored in
- should be CamelCase for multiple words (e.g., class ExampleClass)
- should be descriptive

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 in your application (a.k.a. a page object). In the page object class you store the elements from the page you want to use (e.g., state) and the actions you can perform with those elements (e.g., behavior).

```
// code in page object class
public class Login {

    private WebDriver driver;
    By usernameLocator  = By.id("username");
    By passwordLocator  = By.id("password");
    By loginFormLocator = By.id("login");

    public void with(String username, String password) {
...
// code in test that uses the page object
Login login = new Login
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. Specifying nothing makes it visible only to classes within the same package. A package is a way to group related classes together under a simple name.

For members of a class (e.g., variables and methods) you can use `public`, `private`, `protected`, or nothing.

- `public` and nothing are the same as with classes
- `private` makes it so the member can only be accessed from within the class it was specified
- `protected` makes it so it can be accessed by other classes in the same package
- not specifying a modifier is the same as specifying `protected`

The best thing to do is to follow a "need-to-know" principle for your objects. 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 class it needs to be public for JUnit to use it
public class TestLogin {

    // Our Selenium object should only be accessed from within the same class
    private WebDriver driver;
```

Types

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 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 Boolean successMessagePresent() {
    return isDisplayed(successMessageLocator);
}
```

After specifying the return type when declaring the method, we use the `return` keyword in the method to return the final value.

Actions

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

Assertion

An assertion is a function that allows us to test assumptions about our application.

For instance, in our test we could be testing the login functionality of our application. After logging in, we could check to see if something specific is displayed on the page (e.g., a sign out button, a success notification, etc.). This display check would return a boolean, and we would use it to assert that it is what we expected.

```
// method that looks to see if a success message is displayed after logging in
    public Boolean successMessagePresent() {
        return isDisplayed(successMessageLocator);
    }

// assertion in our test to see if the value returned is the value expected
    assertTrue("success message not present", 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 if`, and `else`.

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

```
if (browser.equals("firefox")) {
        driver = new FirefoxDriver();
    } else if (browser.equals("chrome")) {
        System.setProperty("webdriver.chrome.driver",
                System.getProperty("user.dir") + "/vendor/chromedriver");
        driver = new ChromeDriver();
    }
```

NOTE: The commands **`System.setProperty`** and **`System.getProperty`** are used to set and retrieve runtime properties in Java. This is a handy function to know since it comes built into Java and it enables us to easily create and retrieve values when running our tests.

Annotations

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

The most common use of annotations in Selenium 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
    @Before
    public void setUp() {
    // this method will run before each test
...
    @Test
    public void descriptiveTestName() {
    // this method is a test
...
    @After
    public void tearDown() {
    // this method will run after each test
```

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:

- providing the class name
- using the `extends` keyword
- providing the name of the parent class

```
public class Parent {
  static String hairColor = "brown";
}

public class Child extends Parent {
  public static void main(String[] args) {
    System.out.println(hairColor);
  }
}

// running the Child class outputs "brown"
```

We'll see this in our tests when writing all of the common Selenium actions we intend to use into methods within a parent class (a.k.a. a base page object or facade layer). Inheriting this class will allow us to call these methods in our child classes (e.g., page objects). More on this in Chapter 9.

Additional Resources

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

- Learn Java Online
- Oracle Java Tutorials
- tutorialspoint
- Java Tutorial for Complete Beginners (video course on Udemy)
- Java In a Nutshell
- Java For Testers

Chapter 5
Anatomy Of A Good Acceptance Test

In order to write acceptance tests that perform well and are both maintainable and resilient, here 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.

Grouping Tests

As your test suite grows, you should have multiple test files, each containing a small grouping of tests broken out by functionality that they're exercising. This will go a long way towards organization and maintenance as your test suite grows -- as well as faster execution times (depending on your approach to parallelization).

Being Descriptive

Each test file should be named appropriately, and each test within it should have an informative name (even if it is a bit verbose). Also, each test (or grouping of tests) should be annotated with some helpful metadata to provide additional information and enable flexible test execution (more on flexible test execution in Chapter 15).

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 (as part of their pre-check-in testing) while also enabling you to intelligently wire your suite up to a Continuous Integration (CI) server for fast and dynamic feedback (more on CI servers in Chapter 16).

Test Runners

At the heart of every test suite is some kind of a test runner that does a lot of the heavy lifting (e.g., test group execution, easy global configuration for setup and teardown, reporting, etc.). Rather than reinvent the wheel, you can use one of the many that already exists. With it you can bolt on third party libraries to extend its functionality if there's something missing -- like parallelization.

Version Control

In order to effectively collaborate with other testers and developers, your test code must live in a version control system of some sort. 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 testing will be with Selenium) 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 (or find the login form and submit 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 copy (e.g., text, or 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 drill down into 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 generally not a hard thing for them to add helpful, semantic markup to make test automation easier. This is especially true when they know the use case you're trying to automate. The alternative can be a lengthy, painful process which will probably yield working test code -- but it will be brittle and hard to maintain test code.

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 the parent element (`form`) does. So instead

of clicking the submit button, we can find and submit the form.

Let's put these elements to use in our first test. First we'll need to create a package called `tests` in our `src/tests/java` directory. Then let's add a test file to the package called `TestLogin.java`. When we're done our directory structure should look like this.

```
pom.xml
src
    test
        java
            tests
                TestLogin.java
```

And here is the file populated with our Selenium commands and locators.

```java
//filename: tests/TestLogin.java

package tests;

import org.junit.Test;
import org.junit.Before;
import org.junit.After;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

public class TestLogin {

    private WebDriver driver;

    @Before
    public void setUp() {
        driver = new FirefoxDriver();
    }

    @Test
    public void succeeded() {
        driver.get("http://the-internet.herokuapp.com/login");
        driver.findElement(By.id("username")).sendKeys("tomsmith");
        driver.findElement(By.id("password")).sendKeys("SuperSecretPassword!");
        driver.findElement(By.id("login")).submit();
    }

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

After importing the requisite classes for JUnit and Selenium we create a class (e.g., `public class TestLogin` and declare a field variable to store and reference an instance of Selenium WebDriver (e.g., `private WebDriver driver;`).

We then add setup and teardown methods annotated with `@Before` and `@After` . In them we're creating an instance of Selenium (storing it in `driver`) and closing it (e.g., `driver.quit();`). Because of the `@Before` annotation, the `public void setUp()` method will load before the test and the `@After` annotation 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.

Our test is a method as well (`public void succeeded()`). JUnit knows this is a test because of the

`@Test` annotation. In this test we're visiting the login page by it's URL (with `driver.get();`), finding the input fields by their ID (with `driver.findElement(By.id())`), sending them text (with `.sendKeys();`), and submitting the form (with `.submit();`).

If we save this and run it (e.g., `mvn clean test` from the command-line), it will run and pass. But there's one thing missing -- an assertion. In order to find an element to make an assertion against, we need to see what the markup 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 assertions in this markup. There's the flash message class (most appealing), the logout button (appealing), or the copy from 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 spaces (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 to use it.

```
//filename: tests/TestLogin.java

package tests;

import static org.junit.Assert.*;
...
    @Test
    public void succeeded() {
        driver.get("http://the-internet.herokuapp.com/login");
        driver.findElement(By.id("username")).sendKeys("tomsmith");
        driver.findElement(By.id("password")).sendKeys("SuperSecretPassword!");
        driver.findElement(By.id("login")).submit();
        assertTrue("success message not present",
                driver.findElement(By.cssSelector(".flash.success")).isDisplayed());
    }
...
```

First, we had to import the JUnit assertion class. By importing it as `static` we're able to reference the assertion methods directly (without having to prepend `Assert.`). Next we add an assertion to the end of our test.

With `assertTrue` we are checking for a `true` (Boolean) response. If one is not received, a failure will be raised and the text we provided (e.g., `"success message not present"`) will be in displayed

in the failure output. With Selenium we are seeing if the success message is displayed (with `.isDisplayed()`). This Selenium command returns a Boolean. So if the element is visible in the browser, `true` will be returned, and our test will pass.

When we save this and run it (`mvn clean test` from the command-line) 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 assertion to force a failure and run it again. A simple fudging of the locator will suffice.

```
assertTrue("success message not present",
                driver.findElement(By.cssSelector(".flash.successasdf")).isDisplayed
());
```

If it fails, then we can feel confident that it's doing what we expect, and 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, give something like Firefinder a try.

Firefinder is an add-on to the popular web-development Firefox tool Firebug.

You first need to install Firebug, then Firefinder. Once you have it, verifying locators is a trivial task. And it works for both CSS and XPath locators.

NOTE: An alternative to FireFinder is FirePath. If you have problems with FireFinder, or if you just want to see which one works better for you, then give it a try.

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>
        ...
```

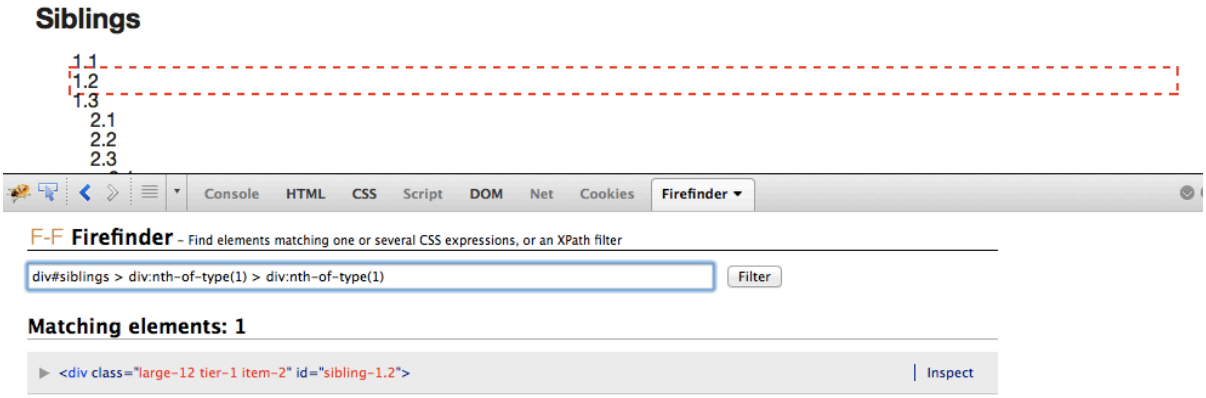
If we perform a `findElement` action using the following locator, it works.

```
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 strategy, 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 Firefinder we can actually see what our locators are doing. Here's what it shows us for the locators that "worked".

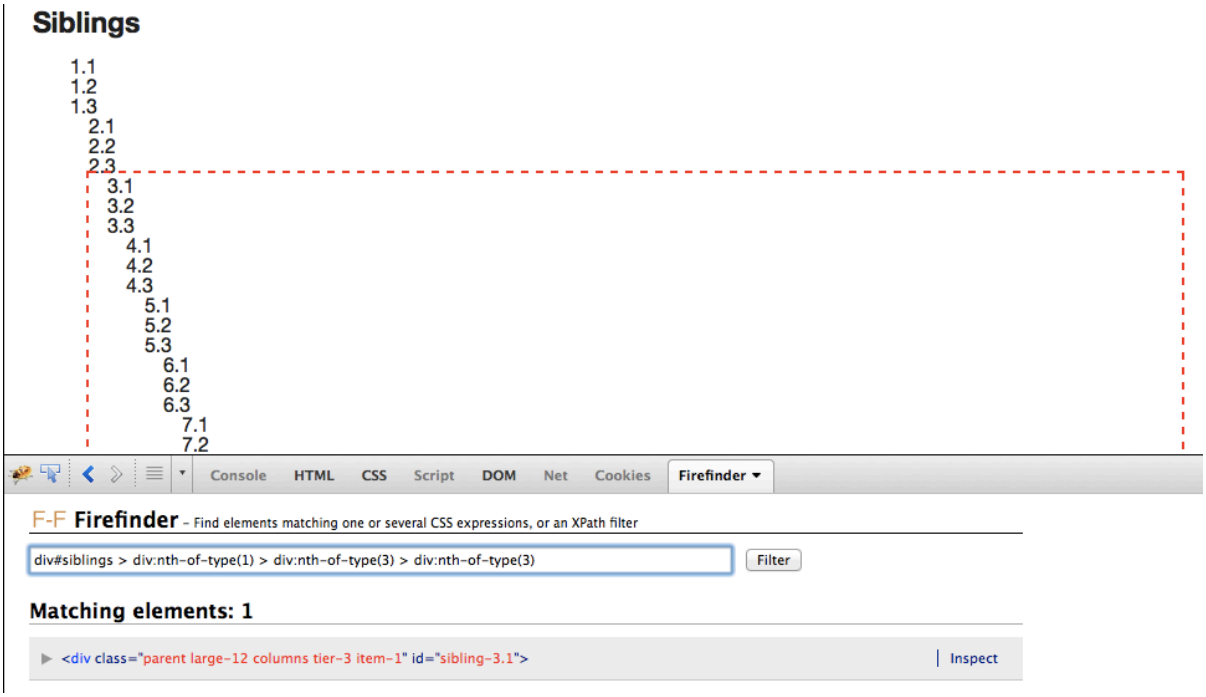


It looks like our locators are scoping to the wrong part of the first level (1.2). We need to reference the third part of each level (e.g., 1.3, 2.3, 3.3) in order to traverse deeper since the nested divs live under the third part of each level.

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)"));
```

And we can confirm that it works before changing any test code by looking in Firefinder.



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 package called `pageobjects` in our `src/tests/java` directory. Then let's add a file to the `pageobjects` package called `Login.java`. When we're done our directory structure should look like this.

```
.
| pom.xml
| src
|   | test
|       | java
|           | pageobjects
|           |   | Login.java
|           | tests
|               | TestLogin.java
```

And here's the code that goes with it.

```
// filename: pageobjects/Login.java

package pageobjects;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

public class Login {

    private WebDriver driver;
    By usernameLocator  = By.id("username");
    By passwordLocator  = By.id("password");
    By loginFormLocator = By.id("login");
    By successMessageLocator = By.cssSelector(".flash.success");

    public Login(WebDriver driver) {
        this.driver = driver;
        driver.get("http://the-internet.herokuapp.com/login");
    }

    public void with(String username, String password) {
        driver.findElement(usernameLocator).sendKeys(username);
        driver.findElement(passwordLocator).sendKeys(password);
        driver.findElement(loginFormLocator).submit();
    }

    public Boolean successMessagePresent() {
        return driver.findElement(successMessageLocator).isDisplayed();
    }

}
```

At the top of the file we specify the package where it lives and import the requisite classes from our libraries. We then declare the class (e.g., `public 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 Login(WebDriver driver)`) is the constructor. It will run whenever a new instance of the 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.get`).

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 strings parameters for the username and password we're able to make the functionality

here dynamic and reusable for additional tests.

The last method (e.g., `public Boolean 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/TestLogin.java

package tests;

import org.junit.Test;
import org.junit.Before;
import org.junit.After;
import static org.junit.Assert.*;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import pageobjects.Login;

public class TestLogin {

    private WebDriver driver;
    private Login login;

    @Before
    public void setUp() {
        driver = new FirefoxDriver();
        login = new Login(driver);
    }

    @Test
    public void succeeded() {
        login.with("tomsmith", "SuperSecretPassword!");
        assertTrue("success message not present",
                login.successMessagePresent());
    }

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

Since the page object lives in another package, we need to import it (e.g., `import pageobjects.Login;`).

Then it's a simple matter of specifying a field for it (e.g., `private Login 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. If you save this and run it (e.g., `mvn clean test` from the command-line), 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 (remember: controlled chaos) 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.

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

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

```
//filename: pageobjects/Login.java
...
    By successMessageLocator = By.cssSelector(".flash.success");
    By failureMessageLocator = By.cssSelector(".flash.error");
...
    public Boolean successMessagePresent() {
        return driver.findElement(successMessageLocator).isDisplayed();
    }

    public Boolean failureMessagePresent() {
        return driver.findElement(failureMessageLocator).isDisplayed();
    }
}
```

Now we're ready to add another test to check for a failure condition.

```
//filename: tests/TestLogin.java
...
@Test
    public void failed() {
        login.with("tomsmith", "bad password");
        assertTrue("failure message wasn't present after providing bogus credentials",
                login.failureMessagePresent());
    }
...
```

If we save these changes and run our tests (`mvn clean test`) 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.

```
@Test
    public void failed() {
        login.with("tomsmith", "bad password");
        assertFalse("success message was present after providing bogus credentials",
                login.successMessagePresent());
    }
```

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

```
org.openqa.selenium.NoSuchElementException: Unable to locate element: {"method":"css
selector","selector":".flash.error"}
Command duration or timeout: 123 milliseconds
For documentation on this error, please visit: http://seleniumhq.org/exceptions/
no_such_element.html
Build info: version: '2.43.1', revision: '5163bceef1bc36d43f3dc0b83c88998168a363a0',
time: '2014-09-10 09:43:55'
System info: host: 'asdf', ip: '192.168.1.112', os.name: 'Mac OS X', os.arch: 'x86_64',
os.version: '10.10.1', java.version: '1.8.0_25'
Driver info: org.openqa.selenium.firefox.FirefoxDriver
Capabilities [{applicationCacheEnabled=true, rotatable=false, handlesAlerts=true,
databaseEnabled=true, version=34.0.5, platform=MAC, nativeEvents=false, acceptSslCerts=
true, webStorageEnabled=true, locationContextEnabled=true, browserName=firefox,
takesScreenshot=true, javascriptEnabled=true, cssSelectorsEnabled=true}]
Session ID: b6648aef-5be5-e542-add1-265ed2a35a65
...
```

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 is more concise.

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.

As a rule, you want to keep assertions in your tests and out of your page objects. But this is the exception to the rule.

```
// filename: pagesobjects/Login.java
...
import static org.junit.Assert.assertTrue;

public class Login {
...
    public Login(WebDriver driver) {
        this.driver = driver;
        driver.get("http://the-internet.herokuapp.com/login");
        assertTrue("The login form is not present",
                driver.findElement(loginFormLocator).isDisplayed());
    }
...
```

After importing the assertion library we put it to use in our constructor (after the Selenium command that visits the login 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, the test will fail and provide an output message stating that the login form wasn't present.

Now when we save everything and run our tests (e.g., `mvn clean test` from the command-line), they will run just like before. But now we can rest assured that the tests will only proceed if the login form is present.

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/experience. The example demonstrated above is a simple approach. Here are some additional resources to consider as your testing practice grows:

- https://code.google.com/p/selenium/wiki/PageObjects
- https://code.google.com/p/selenium/wiki/PageFactory (a Page Object generator/helper built into Selenium)
- https://github.com/yandex-qatools/htmlelements (a simple Page Object framework by Yandex)

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 test suite 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)

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 leverage them 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 `Base.java` to the `pageobjects` package.

```
.
| pom.xml
| src
|   | test
|   |     | java
|   |         | pageobjects
|   |         |   | Base.java
|   |         |   | Login.java
|   |         | tests
|   |             | TestLogin.java
```

Next let's populate the file.

```
// filename: pageobjects/Base.java

package pageobjects;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;

public class Base {

    private WebDriver driver;

    public Base(WebDriver driver) {
        this.driver = driver;
    }

    public void visit(String url) {
        driver.get(url);
    }

    public WebElement find(By locator) {
        return driver.findElement(locator);
    }

    public void click(By locator) {
        find(locator).click();
    }

    public void type(String inputText, By locator) {
        find(locator).sendKeys(inputText);
    }

    public void submit(By locator) {
        find(locator).submit();
    }

    public Boolean isDisplayed(By locator) {
        return find(locator).isDisplayed();
    }

}
```

After declaring the class (e.g., `public class Base`) 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`, `submit`, 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, others renamed (for 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/Login.java

package pageobjects;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import static org.junit.Assert.assertTrue;

public class Login extends Base {

    By usernameLocator  = By.id("username");
    By passwordLocator  = By.id("password");
    By loginFormLocator = By.id("login");
    By successMessageLocator = By.cssSelector(".flash.success");
    By failureMessageLocator = By.cssSelector(".flash.error");

    public Login(WebDriver driver) {
        super(driver);
        visit("http://the-internet.herokuapp.com/login");
        assertTrue("The login form is not present",
                isDisplayed(loginFormLocator));
    }

    public void with(String username, String password) {
        type(username, usernameLocator);
        type(password, passwordLocator);
        submit(loginFormLocator);
    }

    public Boolean successMessagePresent() {
        return isDisplayed(successMessageLocator);
    }

    public Boolean failureMessagePresent() {
        return isDisplayed(failureMessageLocator);
    }
}
```

Two fundamental things have changed in our Login page object.

First, we've established inheritance between `Base` and `Login` with `public class Login extends Base`. This means that `Login` is now a child of `Base`. In order to make the methods in the parent class work, we call `super` (which loads the constructor of the parent class) and pass the `driver` object to it.

Second, we've swapped out all of the Selenium actions to use the methods made available from `Base` thanks to inheritance.

If we save everything and run our tests (e.g., `mvn clean test` from the command-line) the tests 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:

```
org.openqa.selenium.NoSuchElementException: Unable to locate element: {"method":"css
selector","selector":".flash.error"}
Command duration or timeout: 123 milliseconds
For documentation on this error, please visit: http://seleniumhq.org/exceptions/
no_such_element.html
Build info: version: '2.43.1', revision: '5163bceef1bc36d43f3dc0b83c88998168a363a0',
time: '2014-09-10 09:43:55'
System info: host: 'asdf', ip: '192.168.1.112', os.name: 'Mac OS X', os.arch: 'x86_64',
os.version: '10.10.1', java.version: '1.8.0_25'
Driver info: org.openqa.selenium.firefox.FirefoxDriver
Capabilities [{applicationCacheEnabled=true, rotatable=false, handlesAlerts=true,
databaseEnabled=true, version=34.0.5, platform=MAC, nativeEvents=false, acceptSslCerts=
true, webStorageEnabled=true, locationContextEnabled=true, browserName=firefox,
takesScreenshot=true, javascriptEnabled=true, cssSelectorsEnabled=true}]
Session ID: b6648aef-5be5-e542-add1-265ed2a35a65
...
```

The important thing to note is the exception Selenium offered up -- the part that comes just before `Unable to locate element` (e.g., `org.openqa.selenium.NoSuchElementException`).

Let's modify the `isDisplayed` method in our base page object to handle it.

```
// filename: pageobjects/Base.java
...
    public Boolean isDisplayed(By locator) {
        try {
            return find(locator).isDisplayed();
        } catch (org.openqa.selenium.NoSuchElementException exception) {
            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 `failed()` login test and alter it so it triggers a `NoSuchElementException` like before (to make sure things are working as we expect).

```
// filename: tests/TestLogin.java
...
    @Test
    public void failed() {
        login.with("tomsmith", "bad password");
        assertFalse("failure message wasn't present after providing bogus credentials",
                login.successMessagePresent());
    }
...
```

When we save our changes and run this test (`mvn clean test -Dtest=TestLogin#failed` from the command-line) 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. Gone are the days of waiting for the page to finish loading, or hard-coding sleeps, or doing a blanket wait time (a.k.a. an implicit wait). Nay. Now are the wonder years of waiting for an expected outcome to occur for a specified amount of time. If the outcome occurs before the amount of time specified, then the test will proceed. Otherwise, it will wait the full amount of time specified.

We accomplish this through the use of explicit waits.

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 which 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 `DynamicLoading.java` in the `pageobjects` package.

```
.
| pom.xml
| src
|   | test
|   |     | java
|   |         | pageobjects
|   |         |   | Base.java
|   |         |   | DynamicLoading.java
|   |         |   | Login.java
|   |         | tests
|   |             | TestLogin.java
```

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

```java
// filename: pageobjects/DynamicLoading.java

package pageobjects;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

public class DynamicLoading extends Base {

    By startButton = By.cssSelector("#start button");
    By finishText  = By.id("finish");

    public DynamicLoading(WebDriver driver) {
        super(driver);
    }

    public void loadExample(String exampleNumber) {
        visit("http://the-internet.herokuapp.com/dynamic_loading/" + exampleNumber);
        click(startButton);
    }

    public Boolean finishTextPresent() {
        return waitForIsDisplayed(finishText, 10);
    }

}
```

Since there are two examples to choose from we created the method `loadExample` which accepts a String of the example number we want to visit as an argument. 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, adding this new `waitForIsDisplayed` method to use them.

```
// filename: pageobjects/Base.java
...
    public Boolean waitForIsDisplayed(By locator, Integer... timeout) {
        try {
            waitFor(ExpectedConditions.visibilityOfElementLocated(locator),
                    (timeout.length > 0 ? timeout[0] : null));
        } catch (org.openqa.selenium.TimeoutException exception) {
            return false;
        }
        return true;
    }

    private void waitFor(ExpectedCondition<WebElement> condition, Integer timeout) {
        timeout = timeout != null ? timeout : 5;
        WebDriverWait wait = new WebDriverWait(driver, timeout);
        wait.until(condition);
    }

}
```

Selenium comes with a wait function which we wrap in a private method (e.g., `private void waitFor`) for reuse in this class.

This method accepts two arguments -- the condition we want to wait for (e.g., `ExpectedCondition<WebElement>`) and the amount of time we want Selenium to keep checking for (e.g., `Integer timeout`). If a `null` value is passed as an argument for the timeout, then the wait time is set to `5` seconds. This is handled by a ternary operator (e.g., `timeout = timeout != null ? timeout : 5;`).

The `waitForIsDisplayed` method has two parameters -- one for a locator, another for the timeout. Inside the method we call `waitFor` and send it an `ExpectedCondition` to check for the visibility of an element (e.g., `.visibilityOfElementLocated(locator)`). This is similar to our previous display check, but it uses a different Selenium API function that will work with the explicit waits function. You can see a full list of Selenium's `ExpectedConditions` here. Unfortunately, this function doesn't return a Boolean, so we provide one . If the condition is not met by Selenium in the amount of time provided, it will throw a timeout exception. When that happens, we catch it and return `false` . Otherwise, we return `true` .

It's worth noting that the second parameter is optional when calling `waitForIsDisplayed` (e.g., `Integer... timeout`). If a timeout value is specified, it will get passed to the `waitFor` method . `If nothing is specified,` null `will get passed instead. This gives us the freedom to call this method in our page objects without specifying a timeout (e.g.,` waitForIsDisplayed(locator) `) or with a timeout (e.g.,` waitForIsDisplayed(locator, 20)`).

More On Explicit Waits

In our page object when we're using `waitForIsDisplayed(finishText, 10)` we are telling Selenium to check if the finish text is visible on the page repeatedly. It will keep checking until either the element is displayed or reaches ten seconds -- whichever comes first.

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 static 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 implicit wait across local and remote browser drivers. Long story short, you'll see inconsistent and odd test behavior. 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 `TestDynamicLoading.java` in the `tests` package.

```
.
| pom.xml
| src
|   | test
|   |     | java
|   |         | pageobjects
|   |         |   | Base.java
|   |         |   | DynamicLoading.java
|   |         |   | Login.java
|   |         | tests
|   |             | TestDynamicLoading.java
|   |             | TestLogin.java
```

The contents of this test file are similar to `TestLogin` with regards to the imported classes and the `setUp` / `tearDown` methods.

```
// filename: tests/TestDynamicLoading.java

package tests;

import org.junit.Test;
import org.junit.Before;
import org.junit.After;
import static org.junit.Assert.*;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import pageobjects.DynamicLoading;

public class TestDynamicLoading {

    private WebDriver driver;
    private DynamicLoading dynamicLoading;

    @Before
    public void setUp() {
        driver = new FirefoxDriver();
        dynamicLoading = new DynamicLoading(driver);
    }

    @Test
    public void hiddenElementLoads() {
        dynamicLoading.loadExample("1");
        assertTrue("finish text didn't display after loading",
                dynamicLoading.finishTextPresent());
    }

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

In our test (e.g., `public void hiddenElementLoads()`) 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 (`mvn clean test -Dtest=TestDynamicLoading` from the command-line) it will run, wait for the loading bar to complete, and pass.

Part 3: Update Page Object And Add A New Test

Let's step through one 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 text 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 `TestDynamicLoading.java` called `elementAppears()` that will load this second example and perform the same check as we did for the previous test.

```
// filename: tests/TestDynamicLoading.java
...
    @Test
    public void hiddenElementLoads() {
        dynamicLoading.loadExample("1");
        assertTrue("finish text didn't display after loading",
                dynamicLoading.finishTextPresent());
    }

    @Test
    public void elementAppears() {
        dynamicLoading.loadExample("2");
        assertTrue("finish text didn't render after loading",
                dynamicLoading.finishTextPresent());
    }
...
```

When we run both tests (`mvn clean test -Dtests=TestDynamicLoading` from the command-line) we will see that the same approach will work for both cases.

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 against various browsers.

It's simple enough to write your tests locally against Firefox and assume you're all set. 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.

In my experience, Chrome execution is very fast, so you will 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). 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 and find the failures. Take each failed test, adjust your code as needed, and run it against the browsers you care about. Repeat until you make a pass all the way through each of the failed tests. Then run a batch of all your tests to see where they fall down. Repeat until everything's green.

Once you're on the other side of these issues, the amount of effort you need to put into it should diminish dramatically.

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 `Base.java` in the `tests` package.

```
.
| pom.xml
| src
|   | test
|       | java
|           | pageobjects
|           |   | Base.java
|           |   | DynamicLoading.java
|           |   | Login.java
|           | tests
|               | Base.java
|               | TestDynamicLoading.java
|               | TestLogin.java
```

And here are the contents of the file.

```java
// filename: tests/Base.java

package tests;

import org.junit.Rule;
import org.junit.rules.ExternalResource;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

public class Base {

    protected WebDriver driver;

    @Rule
    public ExternalResource resource = new ExternalResource() {

        @Override
        protected void before() throws Throwable {
            driver = new FirefoxDriver();
        }

        @Override
        protected void after() {
            driver.quit();
        }

    };

}
```

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

It's worth noting that we could have used methods with `@Before` and `@After` annotations just like before. But if we did that we'd be giving up the ability to use these annotations in our tests (which we'll want to use for page object instantiation).

To preserve this functionality we're using JUnit's `ExternalResource` Rule. This rule has `before` and `after` methods that execute prior to methods annotated with `@Before` and `@After`. For more info on JUnit's Rules, read this.

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

```java
// filename: tests/TestLogin.java
package tests;

import org.junit.Test;
import org.junit.Before;
import static org.junit.Assert.*;
import pageobjects.Login;

public class TestLogin extends Base {

    private Login login;

    @Before
    public void setUp() {
        login = new Login(driver);
    }

    @Test
    public void succeeded() {
        login.with("tomsmith", "SuperSecretPassword!");
        assertTrue("success message not present",
                login.successMessagePresent());
    }

    @Test
    public void failed() {
        login.with("tomsmith", "bad password");
        assertTrue("failure message wasn't present after providing bogus credentials",
                login.failureMessagePresent());
        assertFalse("success message was present after providing bogus credentials",
                login.successMessagePresent());
    }

}
```

```java
// filename: tests/TestDynamicLoading.java

package tests;

import org.junit.Test;
import org.junit.Before;
import static org.junit.Assert.*;
import pageobjects.DynamicLoading;

public class TestDynamicLoading extends Base {

    private DynamicLoading dynamicLoading;

    @Before
    public void setUp() {
        dynamicLoading = new DynamicLoading(driver);
    }

    @Test
    public void hiddenElementLoads() {
        dynamicLoading.loadExample("1");
        assertTrue("finish text didn't display after loading",
                dynamicLoading.finishTextPresent());
    }

    @Test
    public void elementAppears() {
        dynamicLoading.loadExample("2");
        assertTrue("finish text didn't render after loading",
                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.

First, let's create a file called `Config.java` in the `tests` package.

```
.
| pom.xml
| src
|   | test
|   |     | java
|   |         | pageobjects
|   |         |   | Base.java
|   |         |   | DynamicLoading.java
|   |         |   | Login.java
|   |         | tests
|   |             | Base.java
|   |             | Config.java
|   |             | TestDynamicLoading.java
|   |             | TestLogin.java
```

In it we'll create an interface and specify a field variable for the base URL.

```
// filename: tests/Config.java

package tests;

public interface Config {
    final String baseUrl = System.getProperty("baseUrl",
"http://the-internet.herokuapp.com");
}
```

In the interface we specify our `baseUrl` variable and have it fetch a runtime property of the same name. If there isn't one specified, a sensible default will be used (e.g., `"http://the-internet.herokuapp.com"`). Notice that the variable is marked as `final`. That's because we don't want configuration values to change after our tests start running. Marking them as `final` makes them immutable.

Now let's update our base page object to use `baseUrl` in the `visit` method.

```
// filename: pageobjects/Base.java
...
import tests.Config;

public class Base implements Config {
...
    public void visit(String url) {
       if (url.contains("http"))  {
           driver.get(url);
       } else {
           driver.get(baseUrl + url);
       }
    }
...
```

After importing the config file and implementing the interface (e.g., `public class Base implements Config`) we're able to reference `baseUrl` freely.

There could be a case where we'll want to visit a full URL in a test, so to be safe we've added a conditional check of the `url` parameter to see if a full URL was passed as an argument. If so, we'll visit it. If not, the `baseUrl` will be combined with value passed in `url` (e.g., a URL path) and used as the visiting URL.

Now all we need to do is update our page objects so they're no longer using a hard-coded URL.

```
// filename: pageobjects/Login.java
...
    public Login(WebDriver driver) {
        super(driver);
        visit("/login");
        assertTrue("The login form is not present",
                isDisplayed(loginFormLocator));
    }
...
```

```
// filename: pageobjects/DynamicLoading.java
...
    public void loadExample(String exampleNumber) {
        visit("/dynamic_loading/" + exampleNumber);
        click(startButton);
    }
...
```

Outro

Now when running our tests, we can specify a different base URL at run-time (e.g., `mvn clean test -DbaseUrl=http://localhost:4567`). If one isn't provided, the default we specified will be used. This will enable us to easily run our tests against different environments (e.g., local developmenet, a test environment, production, etc.).

We're also in a better position now. With our setup and teardown abstracted into a central location, 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 from Google and store the unzipped contents of it somewhere. The simplest thing to do is create a new folder for it (and other things like it) in our test code.

Let's create a `vendor` in the root of our project and place the ChromeDriver binary file there.

```
.
| pom.xml
| src
|   | test
|       | java
|           | pageobjects
|           |   | Base.java
|           |   | DynamicLoading.java
|           |   | Login.java
|           | tests
|               | Base.java
|               | Config.java
|               | TestDynamicLoading.java
|               | TestLogin.java
| vendor
| chromedriver
```

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 the `chromedriver` file to the path of our system, or we can pass in the path to the `chromedriver` file when configuring Selenium. For simplicity, 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 a `browser` variable to our `Config` interface.

```
// filename: tests/Config.java

package tests;

public interface Config {
    final String baseUrl = System.getProperty("baseUrl",
"http://the-internet.herokuapp.com");
    final String browser = System.getProperty("browser", "firefox");
}
```

Just like with the `baseUrl`, we'll fetch a runtime property of the same name. If one isn't provided, we'll use a sensible default (e.g., `"firefox"`).

Now we need to update the Selenium setup in our base test.

```
// filename: tests/Base.java
...
public class Base implements Config {
...
        @Override
        protected void before() throws Throwable {
            if (browser.equals("firefox")) {
                driver = new FirefoxDriver();
            } else if (browser.equals("chrome")) {
                System.setProperty("webdriver.chrome.driver",
                        System.getProperty("user.dir") + "/vendor/chromedriver");
                driver = new ChromeDriver();
            }
        }
```

After implementing `Config` (e.g., `public class Base implements Config`) we're able to access our new `browser` variable. So we put it to use with some conditional checks. If its set to `firefox` then we'll launch an instance of Firefox. If its set to `chrome` we tell Selenium where the `chromedriver` binary lives and then launch an instance of Chrome.

Now when we run our tests, we can specify Chrome as our browser at runtime (e.g., `mvn clean test -Dbrowser=chrome` from the command-line). And if we don't specify a browser, Firefox will be used.

It's worth noting that this will only be reasonably performant since it is launching and terminating the ChromeDriver binary executable before and after every test. There are alternative ways to set this up, but this is good enough to see where our tests fall down in Chrome (and it will not be the primary way we will run our tests a majority of the time -- more on that later in the book).

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
- FirefoxDriver
- InternetExplorer Driver
- OperaDriver
- SafariDriver

Chapter 13

Running Browsers In The Cloud

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

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 to scale and 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 `DesiredCapabilities`.

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 `DesiredCapabilities` 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 than we have been. Let's start by updating our `Config` interface to include the variables we'll need to specify.

```
// filename: tests/Config.java

package tests;

public interface Config {
    final String baseUrl        = System.getProperty("baseUrl",
"http://the-internet.herokuapp.com");
    final String browser        = System.getProperty("browser", "firefox");
    final String host           = System.getProperty("host", "localhost");
    final String browserVersion = System.getProperty("browserVersion", "33");
    final String platform       = System.getProperty("platform", "Windows XP");
    final String sauceUser      = "your-sauce-username";
    final String sauceKey       = "your-sauce-access-key";
}
```

In addition to the `baseUrl` and `browser` variables, we've added several other (e.g., `host`, `browserVersion`, `platform`, `sauceUser`, and `sauceKey`).

`host` enables us to specify whether our tests run locally or on Sauce Labs. If we don't specify a value at runtime, then the tests will execute locally.

With `browser`, `browserVeresion`, and `platform` 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. We've made so if no values are provided at run time, they will default to `firefox` version `33` running on `Windows XP`.

`sauceUser` is your Sauce username, and `sauceKey` is your Sauce Access Key (which can be found on your account dashboard). An alternative to hard-coding your credentials is to store them in environment variables and retrieve them.

```
final String sauceUser      = System.getenv("SAUCE_USERNAME");
    final String sauceKey       = System.getenv("SAUCE_ACCESS_KEY");
```

Do whichever you're more comfortable and familiar with.

Now we can update our base test class to work with Selenium Remote.

```
// filename: tests/Base.java
...
        @Override
        protected void before() throws Throwable {
            if (host.equals("saucelabs")) {
                DesiredCapabilities capabilities = new DesiredCapabilities();
                capabilities.setCapability("browserName", browser);
                capabilities.setCapability("version", browserVersion);
                capabilities.setCapability("platform", platform);
                String sauceUrl = String.format(
"http://%s:%s@ondemand.saucelabs.com:80/wd/hub",
                        sauceUser, sauceKey);
                driver = new RemoteWebDriver(new URL(sauceUrl), capabilities);
            } else if (host.equals("localhost")) {
                if (browser.equals("firefox")) {
                    driver = new FirefoxDriver();
                } else if (browser.equals("chrome")) {
                    System.setProperty("webdriver.chrome.driver",
                            System.getProperty("user.dir") + "/vendor/chromedriver");
                    driver = new ChromeDriver();
                }
            }
        }
...
```

In our `before` method we've added a new conditional flow (e.g., `if` / `else if`) to check the `host` variable.

We start by checking to see if it's set to `"saucelabs"`. If it is we create a `DesiredCapabilities` object, populate it (with `browser`, `browserVersion`, and `platform` values), and connect to Sauce Labs using Selenium Remote (passing in the `DesiredCapabilities` object). This will return a Selenium WebDriver object that we can use just like when running our tests locally.

If the `host` variable is set to `"localhost"` then our tests will run locally just like before.

If we save everything and run our tests in Sauce Labs (e.g., `mvn clean test -Dhost=saucelabs`) then on the account dashboard we'll see our tests running in Firefox 33 on Windows XP.

And if we wanted to run our tests on different browser and operating system combinations, here are what some of the commands would look like:

```
mvn clean test -Dhost=saucelabs -Dbrowser="internet explorer" -DbrowserVersion=8
mvn clean test -Dhost=saucelabs -Dbrowser="internet explorer" -DbrowserVersion=8 -
Dplatform="Windows 8.1"
mvn clean test -Dhost=saucelabs -Dbrowser=firefox -DbrowserVersion=26 -Dplatform=
"Windows 7"
mvn clean test -Dhost=saucelabs -Dbrowser=safari -DbrowserVersion=8 -Dplatform="OS X
10.10"
mvn clean test -Dhost=saucelabs -Dbrowser=chrome -DbrowserVersion=40 -Dplatform="OS X
10.8"
```

Notice the properties with quotations (e.g., `"internet explorer"` and `"OS X 10.10"`). When dealing with more than one word in a runtime property we need to make sure to surround them in double-quotes (or else our test code won't compile).

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`.

To grab the name from each test we'll use another one of JUnit's rules in the base test class -- a TestWatcher Rule.

```
// filename: tests/Base.java
...
    protected WebDriver driver;
    private String testName;
...
    @Rule
    public TestRule watcher = new TestWatcher() {
        protected void starting(Description description) {
            testName = description.getDisplayName();
        }
    };

}
```

After creating a variable to store the test name in, we add the `TestWatcher` rule. It has a method called `starting` that gives us access to the description of each test as its starting. So we grab the display name for the test and store it in the `testName` field variable.

Now we can add it to our `DesiredCapabilities`.

```
// filename: tests/Base.java
...
        @Override
        protected void before() throws Throwable {
            if (host.equals("saucelabs")) {
                DesiredCapabilities capabilities = new DesiredCapabilities();
                capabilities.setCapability("browserName", browser);
                capabilities.setCapability("version", browserVersion);
                capabilities.setCapability("platform", platform);
                capabilities.setCapability("name", testName);
...
```

Now when we run our tests in Sauce Labs (e.g., `mvn clean test -Dhost=saucelabs` from the command-line), 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 our test code, along with the help of the `saucerest` library, we can make this change easily enough.

We'll first need install the `saucerest` library by adding it to our `pom.xml` file.

```
// filename: pom.xml
...
        <dependency>
            <groupId>com.saucelabs</groupId>
            <artifactId>saucerest</artifactId>
            <version>1.0.17</version>
            <scope>test</scope>
        </dependency>

    </dependencies>

    <repositories>
        <repository>
            <id>saucelabs-repository</id>
            <url>https://repository-saucelabs.forge.cloudbees.com/release</url>
            <releases>
                <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>true</enabled>
            </snapshots>
        </repository>
    </repositories>

</project>
```

In addition to the dependency values we have to add the repository information of where to download the library from.

Next we need to update our base test class to use this library for each successful and failed test.

```
// filename: tests/Base.java

package tests;
...
import com.saucelabs.saucerest.SauceREST;

public class Base implements Config {

    protected WebDriver driver;
    private String testName;
    private String sessionId;
    private SauceREST sauceClient;
...
        @Override
        protected void before() throws Throwable {
            if (host.equals("saucelabs")) {
                DesiredCapabilities capabilities = new DesiredCapabilities();
                capabilities.setCapability("browserName", browser);
                capabilities.setCapability("version", browserVersion);
                capabilities.setCapability("platform", platform);
                capabilities.setCapability("name", testName);
                String sauceUrl = String.format(
"http://%s:%s@ondemand.saucelabs.com:80/wd/hub",
                        sauceUser, sauceKey);
                driver = new RemoteWebDriver(new URL(sauceUrl), capabilities);
                sessionId = ((RemoteWebDriver) driver).getSessionId().toString();
                sauceClient = new SauceREST(sauceUser, sauceKey);
...
```

After importing the `saucerest` library we create two new field variables. One to store the session ID of the test run (e.g., `private String sessionId`) and the other to store the `saucerest` client session (e.g., `private SauceREST sauceClient`).

Once a Sauce job is established we're able to get the seession ID from `RemoteWebDriver` and store it's string value in `sessionId`. We then create an instance of `SauceREST` (which connects to the Sauce API) and store the session in `sauceClient`.

Now to expand our `TestWatcher` rule to use these variables when a test fails or succeeds.

```
// filename: tests/Base.java
...
    @Rule
    public TestRule watcher = new TestWatcher() {

        protected void starting(Description description) {
            testName = description.getDisplayName();
        }

        @Override
        protected void failed(Throwable throwable, Description description) {
            if (host.equals("saucelabs")) {
                sauceClient.jobFailed(sessionId);
                System.out.println(String.format("https://saucelabs.com/tests/%s",
sessionId));
            }
        }

        @Override
        protected void succeeded(Description description) {
            if (host.equals("saucelabs")) {
                sauceClient.jobPassed(sessionId);
            }
        }

    };

}
```

In addition to the `starting` method offered by the `TestWatcher` rule there are `failed` and `succeeded` methods. Either one will execute after a test depending on its outcome.

With a conditional check in each we make sure the `sauceClient` commands only trigger only when a Sauce session has been established.

When a test is successful the `succeeded` method will fire, marking the Sauce job for the test as passed. When a test fails the `failed` method will trigger, and the Sauce job will get update as failed. Additionally, when there's a failure we'll want to know the URL for the job, so we concatenate the URL and output it to the console using the `System.out.println` command.

Now when we run our tests in Sauce (`mvn clean test -Dhost=saucelabs`) and navigate to the Sauce Labs Account page, 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 if there's a failure, we'll see the URL for the job in the failure output locally.

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 cert 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 that creates a secure tunnel between your machine and their 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 and store Sauce Connect in our `vendor` directory.

```
.
├── pom.xml
├── src
│   └── test
│       └── java
│           ├── pageobjects
│           │   ├── Base.java
│           │   ├── DynamicLoading.java
│           │   └── Login.java
│           └── tests
│               ├── Base.java
│               ├── Config.java
│               ├── TestDynamicLoading.java
│               └── TestLogin.java
├── vendor
│   ├── chromedriver
│   ├── sauce-connect
│   ├── bin
│   │   ├── sc
│   │   └── sc.dSYM
│   │       └── Contents
│   │           ├── Info.plist
│   │           └── Resources
│   │               └── DWARF
│   │                   └── sc
│   ├── include
│   │   └── sauceconnect.h
│   ├── lib
│   │   ├── libsauceconnect.a
│   │   └── libsauceconnect.la
│   └── license.html
```

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

```
> vendor/sauce-connect/bin/sc -u your-sauce-username -k your-sauce-access-key
Sauce Connect 4.3.6, build 1629 b134090
Warning, open file limit 2560 is too low.
Sauce Labs recommends setting it to at least 8000.
Starting up; pid 3050
Command line arguments: vendor/sauce-
your-sauce-username -k ****
Using no proxy for connecting to Sauce Labs REST
Resolving saucelabs.com to 162.222.73.28 took 48 ms
Started scproxy on port 65045.
Please wait for 'you may start your tests' to
Starting secure remote tunnel VM...
Secure remote tunnel VM provisioned.
Tunnel ID: 7e628ddafa7c4802b245c67d181c1b84
Secure remote tunnel VM is now: booting
Secure remote tunnel VM is now: running
Remote tunnel host is: maki76248.miso.saucelabs.com
Using no proxy for connecting to tunnel VM.
Resolving maki76248.miso.saucelabs.com to 162.222.
Starting Selenium listener...
Establishing secure TLS connection to tunnel...
Selenium listener started on port 4445.
Sauce Connect is up, you may start your tests.
Connection established.
```

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 could run `mvn clean test -Dhost=saucelabs -DbaseUrl=http://localhost:4567` from the command-line and it 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 its running.

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

```
Checking domain overlap for my domain sauce-connect.proxy, other tunnel domain sauce-
connect.proxy
Overlapping domain: sauce-connect.proxy, shutting down tunnel 7
e628ddafa7c4802b245c67d181c1b84.
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 the Maven Surefire Plugin it's extremely simple to setup.

Part 1: Setup

The installation and configuration for the Maven Surefire Plugin starts and ends in the `pom.xml` file. Here is what we need to add to it:

```
// filename: pom.xml
...
    <build>
        <pluginManagement>
            <plugins>
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-surefire-plugin</artifactId>
                    <version>2.18.1</version>
                    <configuration>
                        <parallel>classesAndMethods</parallel>
                        <threadCount>5</threadCount>
                    </configuration>
                </plugin>
            </plugins>
        </pluginManagement>
    </build>

</project>
```

Notice that there's no `<dependency>` section for installing this library. Everything is handled within the `<build><pluginManagement>` section. And aside from specifying a `groupId`, `artifactId`, and `version` we have a `configuration` section. In this section we've set the type of `parallel` execution we want (e.g., `classesAndMethods`) along with the number of concurrent threads we'd like to use (e.g., `5`).

When we save this file and run our tests (e.g., `mvn clean test`) we'll see multiple browsers launch at the same time -- completing a run through all of the tests at a much faster rate than before.

For a more thorough explanation of parallel execution with the Maven Surefire Plugin and the other options available, go here.

Part 2: Randomizing

When enabling parallel execution in your tests you may start to see odd, inconsistent behavior that is hard to track down.

This is often due to dependencies between tests that you didn't know were there. A great way to expose these kinds of issues and ensure your tests are ready for prime time is to execute them in a random order. This also has the added benefit of exercising the application you're testing in a random order (which could unearth previously unnoticed bugs).

Luckily, there is a library to enable this and setting it up is another simple change. After installing `random-jUnit` we just need to add an annotation to our base test class.

```
// filename: pom.xml
...
        <dependency>
            <groupId>net.sf.randomjunit</groupId>
            <artifactId>random-jUnit</artifactId>
            <version>1.0.2</version>
        </dependency>

    </dependencies>
...
```

This library requires no additional configuration in the `pom.xml`. It's just a standard installation. Now to modify the base test class.

```
// filename: tests/Base.java
...
import net.sf.randomjunit.RandomTestRunner;

@RunWith(RandomTestRunner.class)
public class Base implements Config {
...
```

After importing the library we use JUnit's `@RunWith` annotation to use the `RandomTestRunner.class` (which is made available by the random-jUnit library library). For more information on the JUnit's @RunWith annotation, go here.

Now when we run our tests (e.g., `mvn clean test` from the command-line) they will run in a random order.

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 JUnit's Categories we're able to easily achieve test grouping. We first need to specify the categories we want to use, and then place a test (or set of tests) into a category. We can then specify which group of tests to run by specifying a category at runtime.

Let's step how to set this up.

Part 1: Creating Categories

Each category we want will need to have its own interface. So let's create a new package called `groups` inside of the `tets` package. Inside `groups` we'll create three interface files -- `All.java`, `Deep.java`, and `Shallow.java`.

"Shallow" tests are equivalent to "smoke" (or "sanity") tests. These should pass before you can consider running other (e.g., "Deep") tests. Whereas "All" simply means all tests.

```
.
| pom.xml
| src
|   | test
|       | java
|           | pageobjects
|           |   | Base.java
|           |   | DynamicLoading.java
|           |   | Login.java
|           | tests
|               | Base.java
|               | Config.java
|               | TestDynamicLoading.java
|               | TestLogin.java
|               | groups
|                   | All.java
|                   | Deep.java
|                   | Shallow.java
```

Next, we'll populate the interface files.

```
// filename: tests/groups/All.java

package tests.groups;

public interface All { }
```

```
// filename: tests/groups/Deep.java

package tests.groups;

public interface Deep extends All { }
```

```
// filename: tests/groups/Shallow.java

package tests.groups;

public interface Shallow extends All { }
```

Since we can establish inheritance with interfaces (just like in classes), it's easy enough to connect one category with another. So for these categories, we've made it so `Deep` and `Shallow` are children of `All`. This way we can not only execute a subset of tests, but all tests if we wanted.

Part 2: Specifying Categories

Now that we have category interfaces created, we can add them to our tests. This can either be done on individual tests, or at the class level.

```
// filename: tests/TestLogin.java
...
import org.junit.experimental.categories.Category;
import tests.groups.Shallow;
...
    @Test
    @Category(Shallow.class)
    public void succeeded() {
...
    @Test
    @Category(Shallow.class)
    public void failed() {
...
```

```
// filename: tests/TestDynamicLoading.java
...
import org.junit.experimental.categories.Category;
import tests.groups.Deep;

@Category(Deep.class)
public class TestDynamicLoading extends Base {
...
```

Keep in mind that we can apply the same category to different tests across numerous files. That's just now how they were applied in this case.

Part 3: Running Categories

Before we can run our tests with these categories, we need to make it so we can specify a category as a runtime property. For that, we'll modify `pom.xml`.

```
// filename: pom.xml
...
    <properties>
        <groups>tests.groups.Shallow</groups>
    </properties>

    <build>
        <pluginManagement>
            <plugins>
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-surefire-plugin</artifactId>
                    <version>2.18.1</version>
                    <configuration>
                        <parallel>classesAndMethods</parallel>
                        <threadCount>5</threadCount>
                        <groups>${groups}</groups>
                    </configuration>
                </plugin>
            </plugins>
        </pluginManagement>
    </build>

</project>
```

Within the Maven Surefire Plugin we can specify which categories we want to run in the `<configuration><groups>` option of the `pom.xml`.

Under normal circumstances this would be a static value that would run every time. But we're able to achieve more flexibility at runtime by specifying a `<groups>` value under `<properties>`. This specifies a runtime property with a sensible default (e.g., if a value is provided at runtime it will be used, otherwise the value specified here in this file will be used).

To run a specific test category we need to provide the value for it at runtime by specifying the groups runtime property (e.g., `-Dgroups=`).

Here are the available execution commands given our current category interfaces:

```
mvn clean test -Dgroups=tests.groups.Shallow
mvn clean test -Dgroups=tests.groups.Deep
mvn clean test -Dgroups=tests.groups.All
```

You can also specify multiple tags by separating them with a commma (without spaces).

```
mvn clean test -Dgroups=tests.groups.Shallow,tests.groups.Deep
```

For more info on this functionality and other available options, check out the JUnit Categories documentation and the Maven Surefire Plugin documentation for JUnit.

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 Continous Integration server (a.k.a. CI) is responsible for merging code that is actively being developed into a central place (e.g., "trunk" 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.

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

- Bamboo
- Jenkins
- Solano Labs
- TravisCI

Let's pick one and step through an example.

A CI Example

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

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.

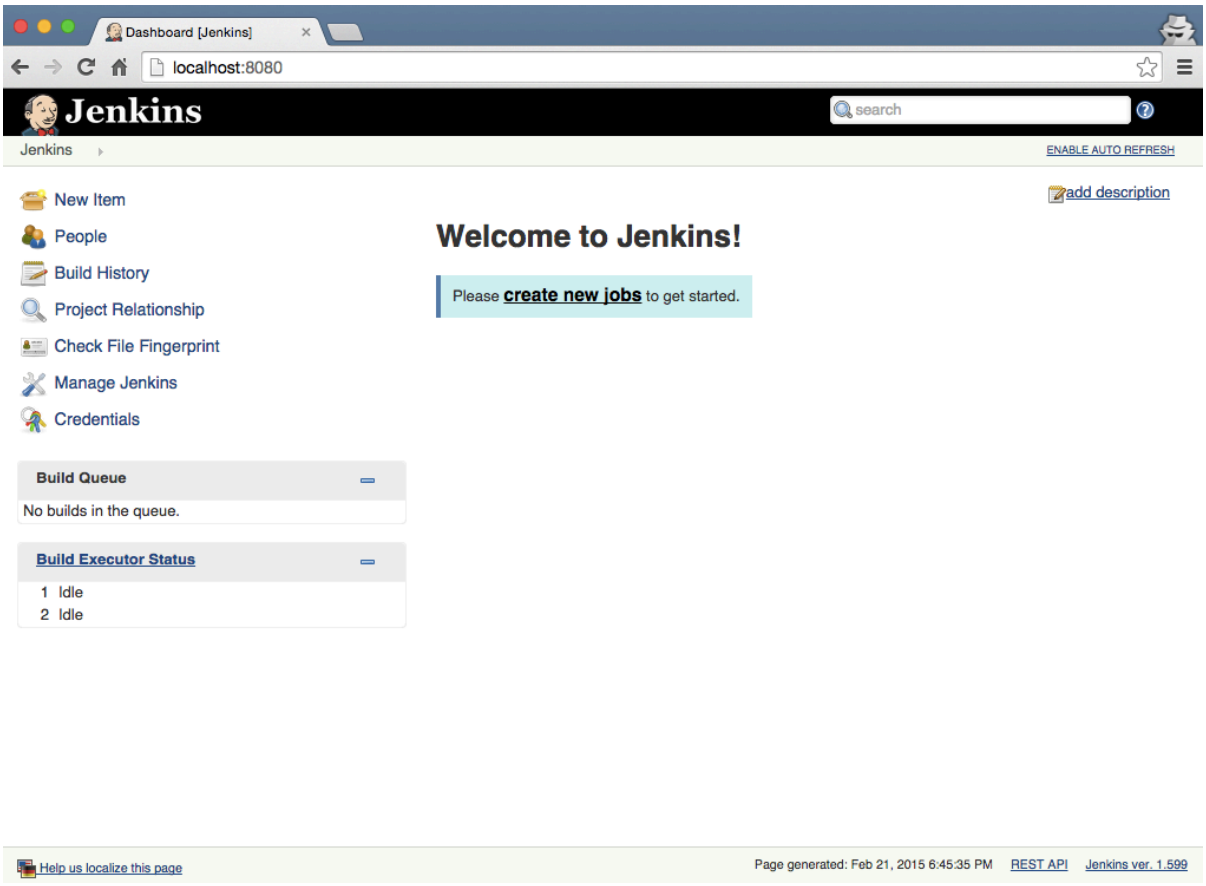
Part 1: Quick Setup

A simple way to get started is to grab the latest Jenkins war file. You can grab it from the Jenkins homepage, or from the direct download link on the homepage.

Once downloaded, launch it from the terminal.

```
> 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.



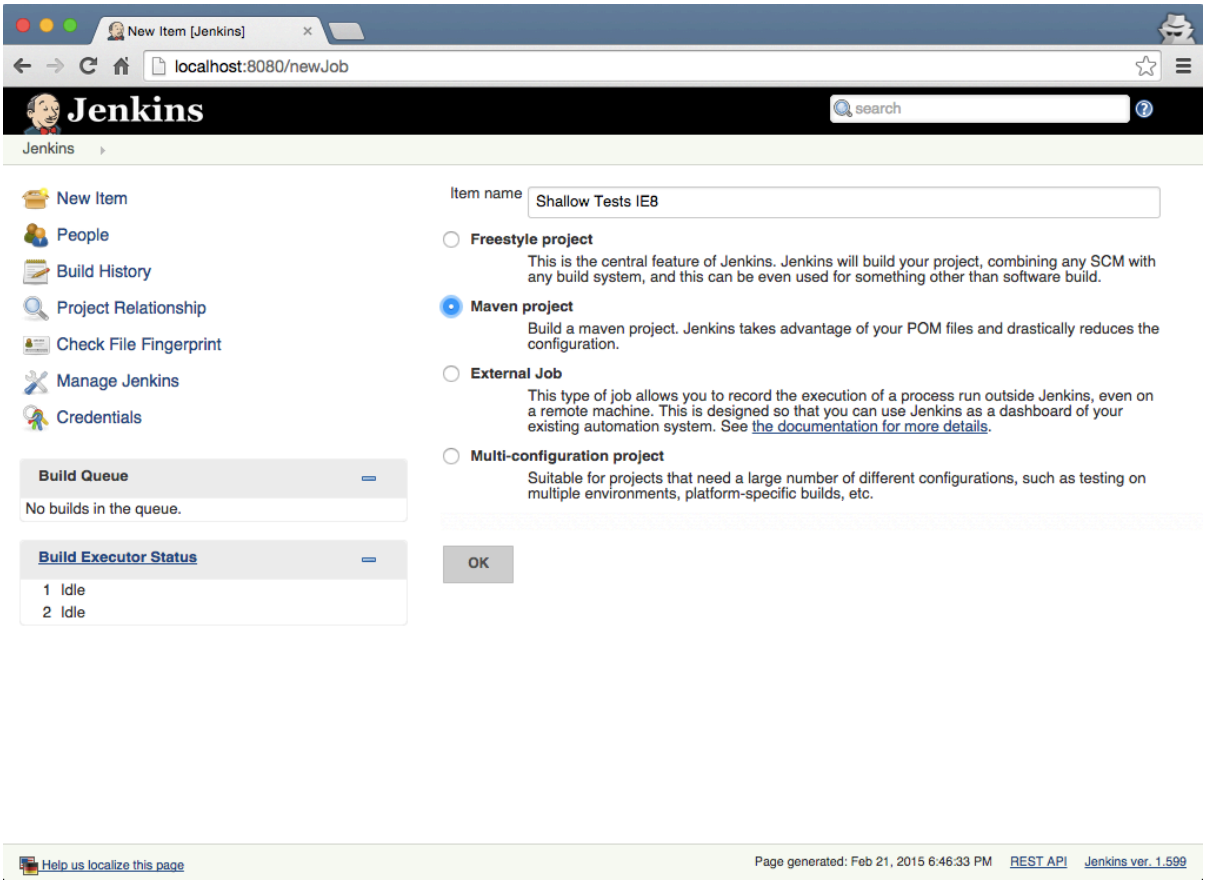
NOTE: Before moving to the next step, click **`ENABLE AUTO-REFRESH`** at the top right-hand side of the page. Otherwise you'll need to manually refresh the page (e.g., when running a job and waiting for results to appear).

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 an old version of Internet Explorer (e.g., IE8).

- Click `New Item` from the top-left of the Dashboard
- Give it a name (e.g., `Shallow Tests IE8`)

- Select the `Maven project` option
- Click `OK`



This will load a configuration screen for the Jenkins job.

![Jenkins Job Configuration[(screenshotjenkinsjob_config.png)

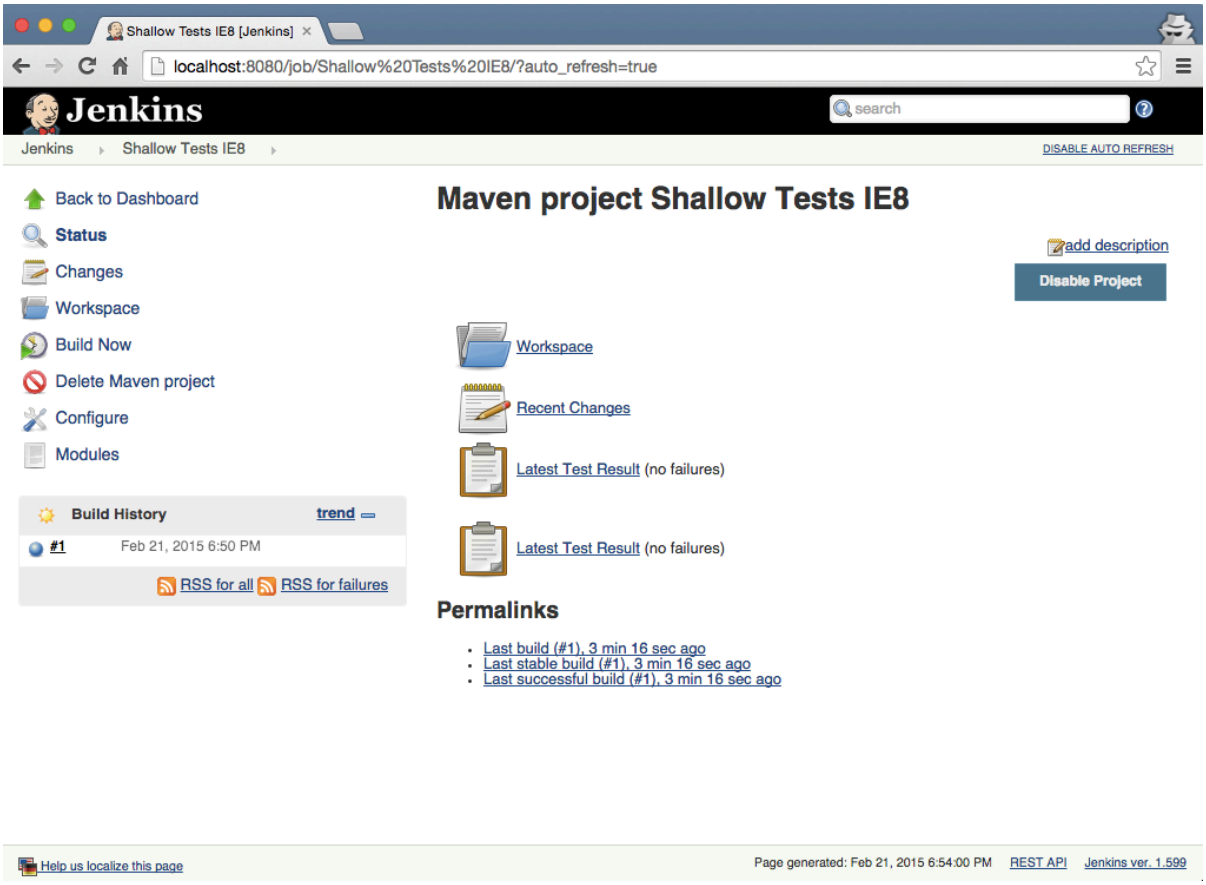
- Scroll down until you reach the `Build` section (near the bottom of the page)
- Specify the absolute path to your `pom.xml` in `Room POM`
- Specify the run commands you want to use in `Goals and options` (e.g., `clean test -Dhost=saucelabs -Dbrowser="internet explorer" -DbrowserVersion=8`)

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.

Now we're ready to save, run our tests, and view the job result.

- Click `Save`
- Click `Build Now` from the left-hand side of the screen

When the build completes, the result will be listed on the job's home screen. In this case, the job passed.



NOTE: If you had a different result, you can drill into a job to see what was happening behind the scenes. To do that click on the build you want from **`Build History`** and select **`Console Output`**. This output will be your best bet in tracking down an unexpected result.

A passing job means passing tests, which is great. But we'll also want to see what a failure looks like to make sure its helpful.

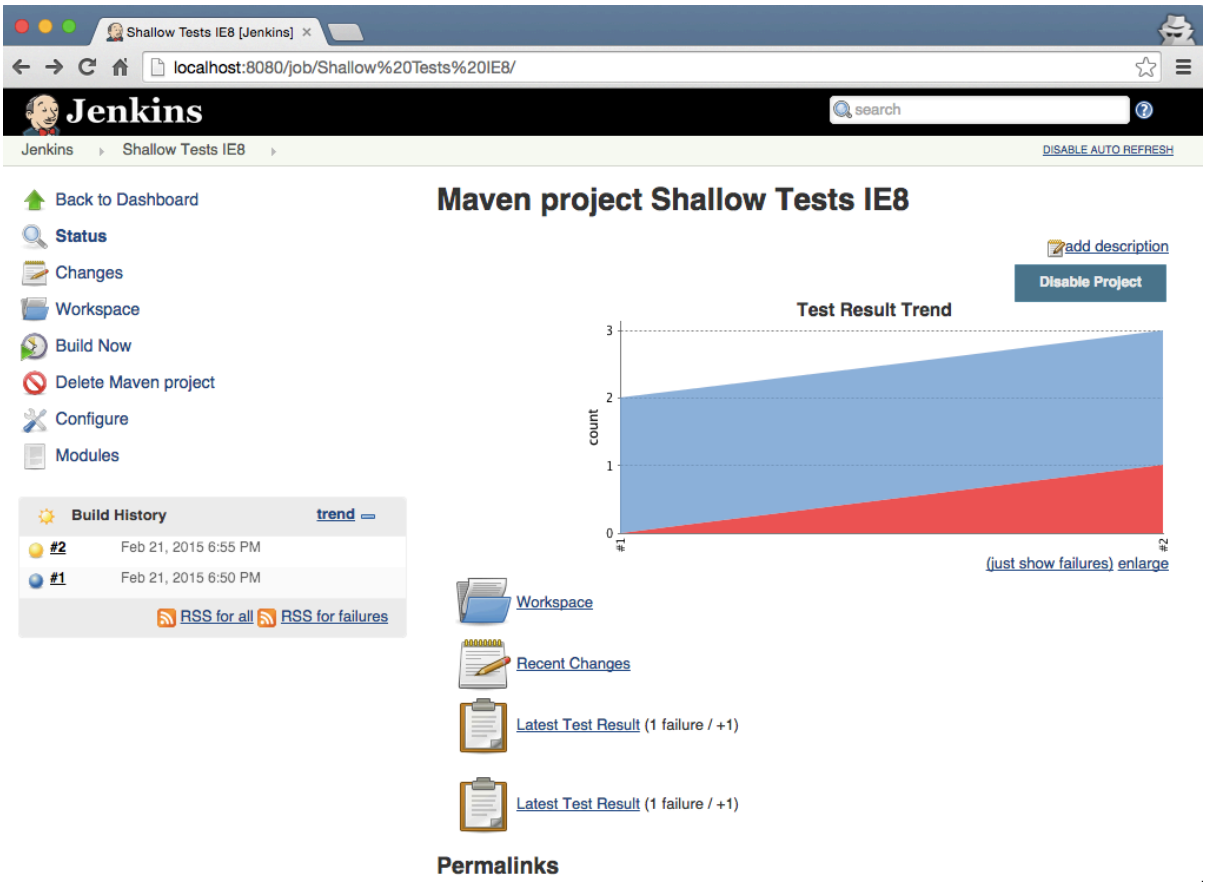
Part 3: Force A Failure

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

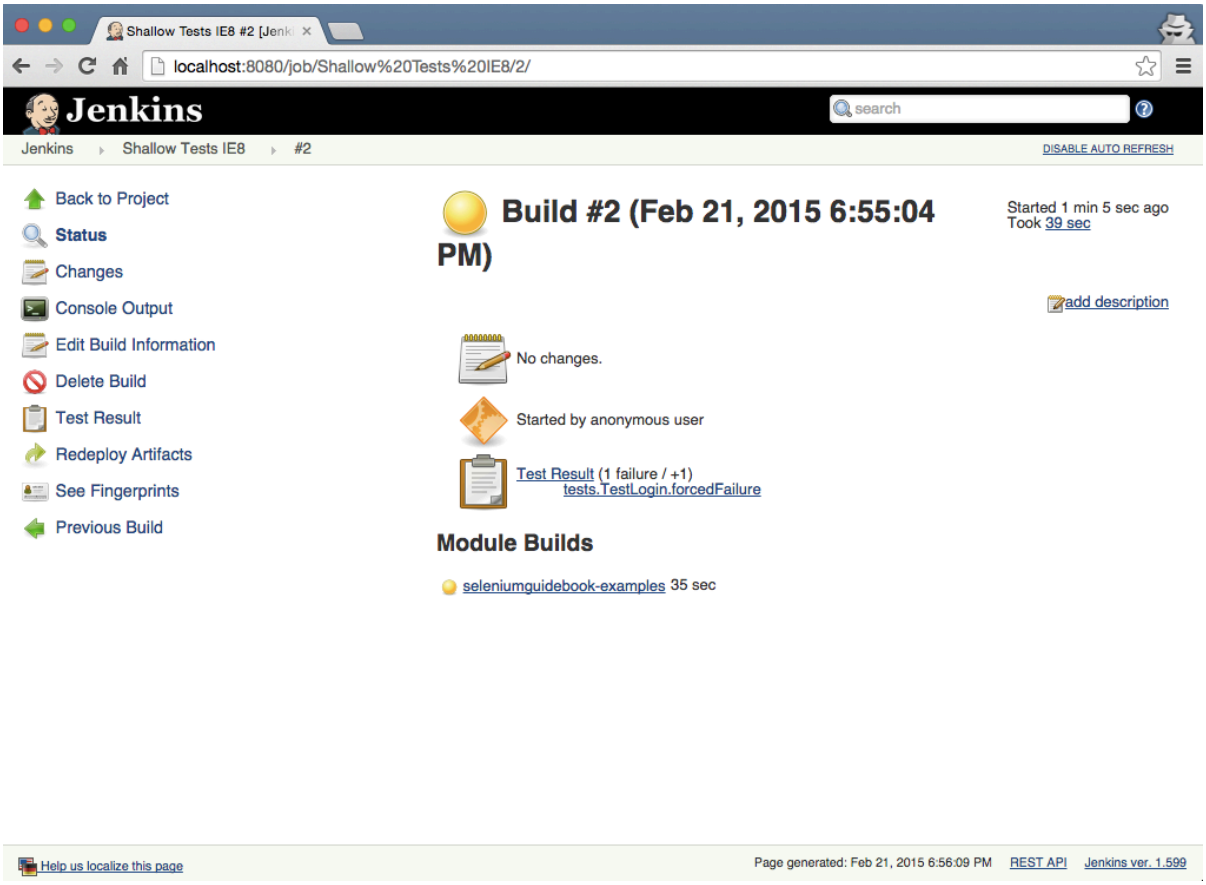
```
// filename: tests/TestLogin.java
...
    @Test
    @Category(Shallow.class)
    public void forcedFailure() {
        pageobjects.Login login = new pageobjects.Login(driver);
        login.with("tomsmithasdf", "SuperSecretPassword!");

        assertTrue("success message wasn't present after logging in",
                login.successMessagePresent());
    }

}
```

This test mimics our `failure` test by visiting the login page and providing invalid credentials. The difference is in the assertion. It will fail since a success message won't be present after attempting to login with bogus credentials.

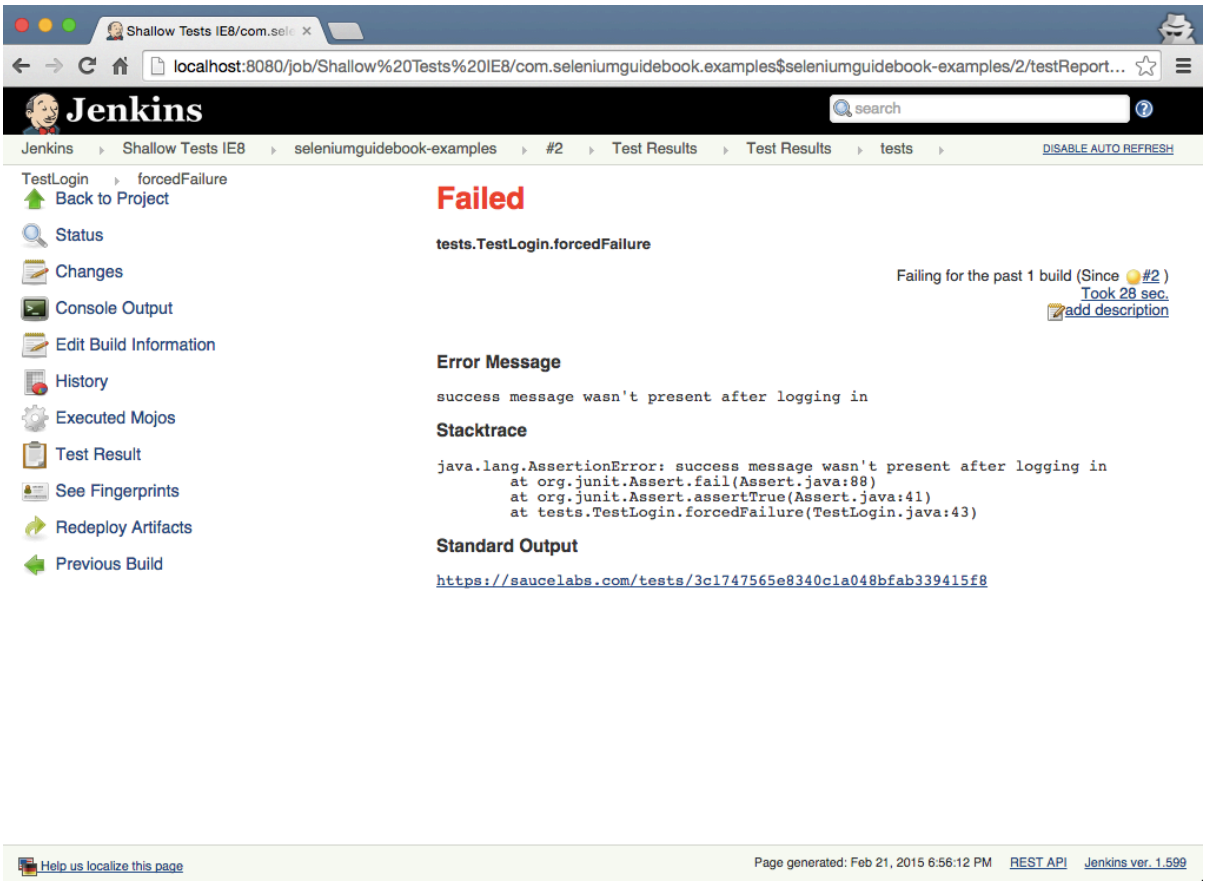
Now when we run our Jenkins job again, it will fail.



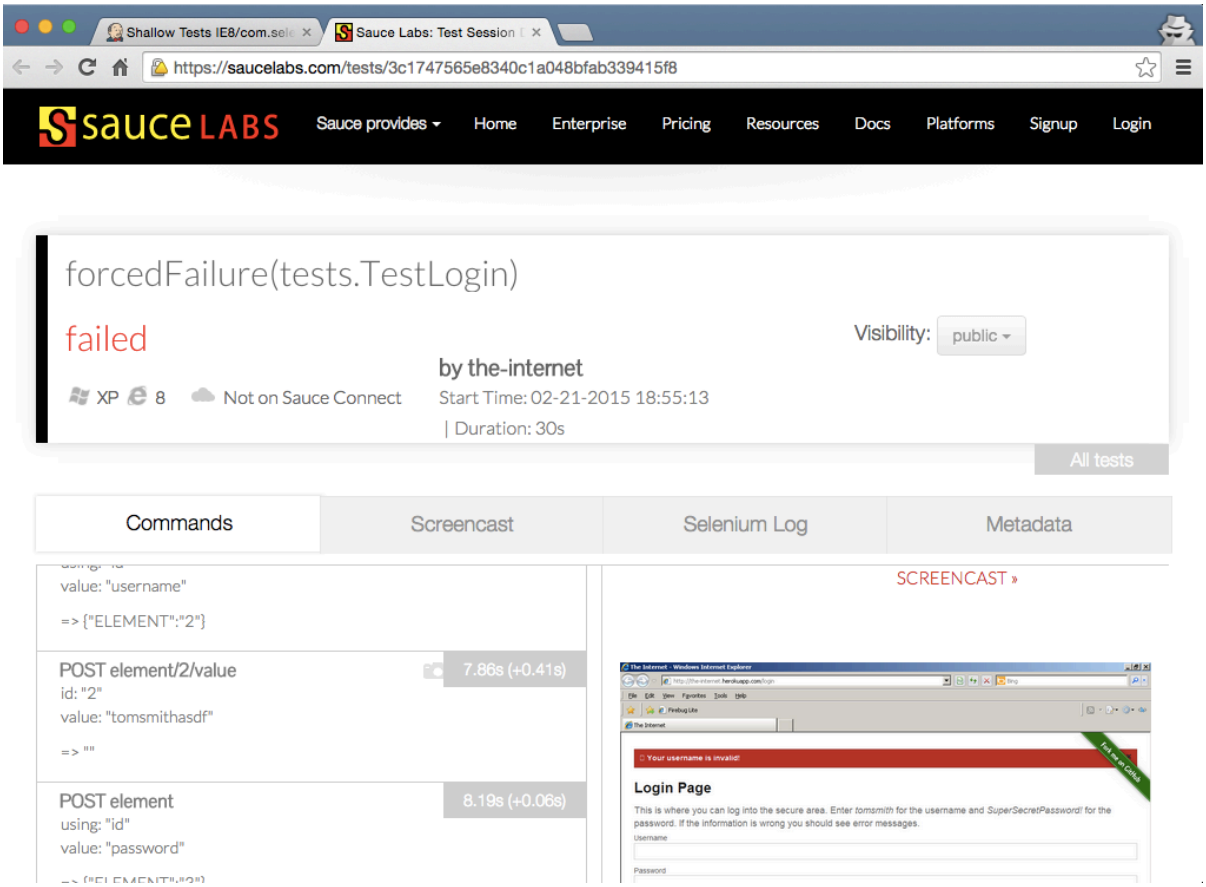
When we click on `Latest Test Result` we can see the test that failed (e.g., `tests.TestLogin.forcedFailure`).



And if we click on the failed test, we can see the failure message along with a 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.).

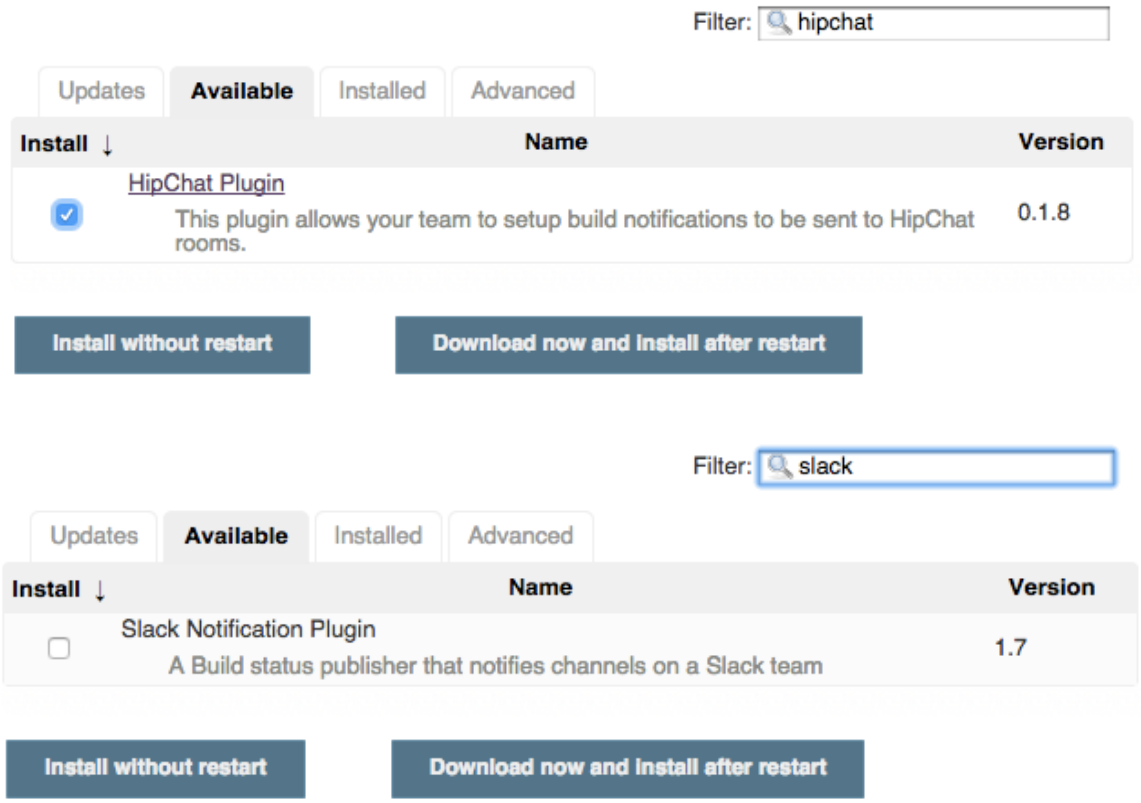


Notifications

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). And 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 HipChat or Slack, you would do a plugin search and find the following plugins:



After installing the plugin for your chat service, 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 code deployment action as a preliminary step before your "Shallow" and "Deep" jobs can be run. 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.