The First Things You Need To Know

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

But before we dig in, there are a few things you'll want to know before you write your first test.

Define a Test Strategy

A great way to increase your chances of automated web testing success is to focus your efforts by mapping out a testing strategy. The best way to do that is to answer four questions:

1. How does your business make money (or generate value for the end-user)?
2. How do your users use your application?
3. What browsers are your users using?
4. What things have broken in the application before?

After answering these, you will have a good understanding of the functionality and browsers that matter most for the application you're testing. This will help you narrow down your initial efforts to the most important things.

From the answers you should be able to build a prioritized list (or backlog) of critical business functionality, a short list of the browsers to focus on, and include the risky parts of your application to watch out for. This prioritized list will help you make sure you're on the right track (e.g., focusing on things that matter for the business and its users).

Pick a Programming 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 as what 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?**

Also, as you are considering which language to go with, consider what open source frameworks already exist for the languages you're eyeing. Going with one can save you a lot of time and give you a host of functionality out of the box that you would otherwise have to build and maintain yourself -- and it's FREE.

You can see a list of available open source Selenium frameworks here.

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.

For this course we'll be using the Java programming language. If you need help installing Java, then check out one of the following links:

- Windows
- Mac
- Linux

Choose an Editor

In order to be productive when writing Java code, you will want to use an integrated development environment (IDE). Here are some of the more popular ones:

- Eclipse
- IntelliJ

Next, we'll be diving into how to decompose a web app and writing our first test.

Writing Your First Selenium Test

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

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

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

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

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

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

How To Find Locators

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

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). Often-times the parent element has a unique element that you can use to start with and drill down to the child or descendent 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 a trivial thing for them to add helpful, semantic markup to make the application more testable. This is especially true when they know the use case you're trying to automate. The alternative is for you to muddle through it in 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. Let's step through an example of this as we write our first test.

An Example

Part 1: Find The Elements And Write The Test

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

```
<form name="login" id="login" action="/authenticate" method="post">
   <div class="row">
    <div class="large-6 small-12 columns">
      <label for="username">Username</label>
```

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

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

Let's put these elements to use in our first test. First we'll need to create a 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.

```
//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.cssSelector("button")).click();
    }

    @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 by clicking the submit button (with`By.cssSelector("button")).click();`).

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 course 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.cssSelector("button")).click();
        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.

In the next lesson, we'll learn about writing maintainable test code.

How To Write Maintainable Tests

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 (as well as more readable tests).

Let's step through an example.

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 submitButton     = By.cssSelector("button");
    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(submitButton).click();
    }

    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.

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:

- Page Objects documentation from the Selenium project
- Page Factory (a Page Object generator/helper built into Selenium)
- HTML Elements (a simple Page Object framework by Yandex)

Now that you understand how to write maintainable tests with page objects, you're ready to dive into the next lesson: writing resilient tests.
