Getting Started with Selenium for Automated Website Testing

FREE Selenium eLearning Course from Sauce Labs

Sauce Labs offers a free eLearning course to help you get started with Selenium! You'll learn how to set up a Selenium environment on your local machine, walk through the seven parts of a Selenium script, see how to locate elements to test and perform actions on them, and then run a sample script on Sauce Labs. Check out our Training portal to enroll and get started.

Introducing Selenium

Selenium is designed to **automate web browser interaction**, so scripts can automatically perform the same interactions that any user can perform manually. Selenium can perform any sort of automated interaction, but was originally intended and is primarily used for automated web application testing.

This topic is intended to provide you with a quick overview of what Selenium does, and the basic components of a Selenium test script. For full documentation of Selenium with extensive examples in the most popular scripting languages, check out the documentation at SeleniumHQ.

Selenium Architecture

Selenium has a **client-server architecture**, and includes both client and server components

Selenium Client includes:

- The **WebDriver** API, which you use to develop test scripts to interact with page and application elements
- The `RemoteWebDriver` class, which communicates with a remote Selenium server

Selenium Server includes:

- A server component, to receive requests from Selenium Client 's `RemoteWebDriver` class
- The **WebDriver** API, to run tests against web browsers on a server machine
- **Selenium Grid**, implemented by Selenium Server in command-line options for grid features, including a central hub and nodes for various environments and desired browser capabilities

The Seven Basic Steps of Selenium Tests

There are seven basic steps in creating a Selenium test script, which apply to any test case and any application under test (AUT).

1. Create a WebDriver instance.
2. Navigate to a Web page.
3. Locate an HTML element on the Web page.
4. Perform an action on an HTML element.
5. Anticipate the browser response to the action.
6. Run tests and record test results using a test framework.
7. Conclude the test.

The Example Use Case and Web Application

The Login Use Case

Imagine that you want to test a very basic use case for any website or Web application, in which a user logs in and, upon successful authentication, receives a message. There are two basic processes in this use case, each of which we want to test:

- The **login** process, which involves the user entering a username and password in a form, and then clicking a **Submit** button.
- The **login response** process, in which the website displays the login response message**.**

The Foo Web Application

The Foo Web application, hosted at www.foo.com, implements the login process through a login form with HTML text inputs for username and password, and **Submit** and **Cancel** buttons. Depending on the success or failure of the login, Foo then displays a message. The aim of your Selenium test is to reproduce the action of a user who enters login information and clicks **Submit**, and then test whether the proper message is displayed.

HTML for the Login Web Page

This is the HTML code for the login form.

```
<html>
  <body>
      ...
    <form action="loginAction" id="loginForm">
      <label>User name: </label>
      <input type="text" name="username"><br>
      <label>Password: </label>
      <input type="text" name="password"><br>
      <button type="submit" id="loginButton">Log In</button>
      <button type="reset" id="reset">Clear</button>
    </form>
    ...
  </body>
</html>
```

And this is the HTML for the successful login response method.

```
<html>
  <body>
      ...
    <p class="message" id="loginResponse">Welcome to foo. You logged in
successfully.</p>
      ...
  </body>
</html>
```

Best Practices for Identifying Elements in HTML Code

When you write a Selenium test, you need to identify the elements that you want the test to interact with. In this code example, each of the elements you want to test is identified using either a name or id attribute, which follows HTML best coding practices.

- On the login web page, the username and password text input elements are identified uniquely by the values of their `name` attributes - `username` and `password`, respectively.
- The login form element is identified uniquely by the value of its `id` attribute, `login`.
- The login response message paragraph has a generic `message` class, but is identified uniquely by the value of its `id` attribute, `loginResponse`.

Creating an Instance of the `WebDriver` Interface

The `WebDriver` interface is the starting point for all uses of the Selenium WebDriver API. Instantiating the WebDriver interface is the first step in writing your Selenium test.

You create an instance of the `WebDriver` interface using a constructor for a specific web browser. The names of these constructors vary over web browsers, and invocations of constructors vary over programming languages.

Once you have created an instance of the `WebDriver` interface, you use this instance to invoke methods and to access other interfaces used in basic steps. You do so by assigning the instance to a variable when you create it, and by using that variable to invoke methods.

Example

This example instantiates the Firefox WebDriver, and assigns it a variable named `driver`.

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
WebDriver driver = new FirefoxDriver();
```

Local v. Remote WebDrivers

If you are running a Selenium test for a single type of browser on a local machine, you would use code similar to this example. However, if you are running your Selenium tests in the Sauce Labs browser cloud, you would want to instantiate the RemoteWebDriver, and you would set the browser/operating system combinations to use in your tests through Selenium's DesiredCapabilities, as shown from this example of a test written in Java. The scripts in Instant Selenium Tests include examples of how you would invoke RemoteWebDriver for various scripting languages.

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.openqa.selenium.remote.RemoteWebDriver;
import java.net.URL;

public class SampleSauceTest {

  public static final String USERNAME = "YOUR_USERNAME";
  public static final String ACCESS_KEY = "YOUR_ACCESS_KEY";
  public static final String URL = "http://" + USERNAME + ":" + ACCESS_KEY + "@ondemand.saucelabs.com:
80/wd/hub";
```

```
public static void main(String[] args) throws Exception {

  DesiredCapabilities caps = DesiredCapabilities.chrome();
  caps.setCapability("platform", "Windows XP");
  caps.setCapability("version", "43.0");

  WebDriver driver = new RemoteWebDriver(new URL(URL), caps);
```

Navigating to a Web Page

Once you've instantiated WebDriver, the next step is to navigate to the Web page you want to test. You do this by invoking the `get` method on the unique instance of the `WebDriver` interface, specifically on the `driver` variable. The `get` method takes the URL of the web page you want to test as an argument. It can be a string value, or an instance of a special type representing a URL or URI.

Example

This example invokes the `get` method on the `driver` variable to navigate to the web page at `www.foo.com`, passing a string argument value for its URL. You can find other examples in the SeleniumHQ documentation.

```
driver.get("http://www.foo.com");
```

Locating an HTML Element on a Web Page

In order to interact with a web page, you first locate HTML elements on the web page, then perform actions on those elements, such as entering text (for text input elements) or clicking (for button elements). The documentation at SeleniumHQ contains extensive information on the different methods for locating HTML, this topic summarizes the most common methods.

Locator Expressions

You use a locator expression to locate a unique HTML element or a specific collection of HTML elements. A locator expression is a key: value pair containing a **locator type** and a **locator value**.

The locator type indicates which aspects of any HTML element on a web page are evaluated and compared to the locator value in order to locate an HTML element. An aspect of an HTML element indicated by a locator type can include:

- A specific **attribute** such as `name` or `id`
- The **tag name** of the element, such as `form` or `button`
- For hyperlink elements or anchor tags, the visible **linked text**, such as `Foo` in `<a href="http://www.foo.com">Foo</a>`
- Any aspects given by a **CSS** selector, such as `...`
- Any aspects given by an **XPath** expression, such as `//form[@id="loginForm"]` or `//button[@type='submit']`.

Locator Methods on the By Class

The WebDriver API provides several **locator methods** to form locator expressions. Each locator method corresponds to a locator type, and forms a locator expression containing that type and a locator value passed as an argument when invoking the method.

In the WebDriver API for Java, locator methods are defined as `static` or class methods on the `By` class (whose name connotes that an HTML element is located *by* comparing an evaluated locator type to a locator value). For example, `By.name("password")` forms a locator expression whose locator type indicates the `name` attribute and whose locator value is the string `"password"`.

Finder Methods and the `WebElement` Interface

To use locator expressions formed by locator methods, the WebDriver API provides two **finder methods**, `findElement` (singular) and `findElements` (plural), both of which take a locator expression as an argument value.

Typically, a locator method is invoked on the `By` class in the argument position of a finder method to form a locator expression as argument value in a single line of code - e.g. `findElement(By.name("password"))`.

In the example below, the `findElement` and `findElements` finder methods are invoked on the unique instance of the `WebDriver` interface - e.g. on the variable `driver`, as in `driver.findElement(By.name("password"))`.

The finder methods search the DOM (Document Object Model) tree for the web page, evaluating locator types for HTML elements, and comparing their values to the locator value.

The return value of the `findElement` (singular) method is an instance of the `WebElement` interface, which represents an element of any HTML type and which is used to perform actions on the element. The `findElement` (singular) method returns a `WebElement` for the first HTML element in the DOM tree for which the evaluated locator type matches the locator value. The `findElements` (plural) method returns a list of elements - in the WebDriver API for Java, a `List<WebElement>` - for all HTML elements on the web page for which the evaluated locator type matches.

Example: Locate HTML text input elements for username and password

This example invokes the `findElement` method on the `driver` variable, using the `name` attribute to locate the `username` and `password` text input elements, and (optionally) the `id` attribute to locate the `form` element.

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebElement;
WebElement usernameElement = driver.findElement(By.name("username"));
WebElement passwordElement = driver.findElement(By.name("password"));
```

Optionally locate the HTML form element

```
WebElement formElement = driver.findElement(By.id("loginForm"));
```

Performing an Action on an HTML Element

Once you've identified the HTML elements you want your test to interact with, the next step is to interact with them. You perform an action on an HTML element by invoking an interaction method on an instance of the `WebElement` interface.

The `WebElement` interface declares basic interaction methods including:

- The `sendKeys` method, to enter text
- The `clear` method, to clear entered text
- The `submit` method, to submit a form

Example

This example first invokes the `sendKeys` method to enter text in the `username` and `password`elements, and then invokes the `submit` method to submit the `login` form.

Enter a user name and a password

```
usernameElement.sendKeys("Alan Smithee");
passwordElement.sendKeys("twilightZone");
```

Submit the form

> The `submit` method can be invoked either on any text input element on a form, or on the form element itself. This example shows both options.

```
passwordElement.submit();  // submit by text input element
or
formElement.submit();  // submit by form element
```

Anticipating Browser Response

When you click a **Submit** button, you know that you have to wait a second or two for the your action to reach the server, and for the server to respond, before you do anything else. If you're trying to test the response, and what happens afterwards, then you need to build that waiting time into your test. Otherwise, the test might fail because the elements that are expected for the next step haven't loaded into the browser you. The WebDriver API supports two basic techniques for anticipating browser response by waiting: **implicit waits** and **explicit waits**.

> **Do Not Mix Explicit and Implicit Waits**
>
> Do not mix implicit and explicit waits. Doing so can cause unpredictable wait times. For example setting an implicit wait of 10s and an explicit wait of 15 seconds, could cause a timeout to occur after 20 seconds.

Implicit Waits

Implicit waits set a definite, fixed elapsed time that applies to all `WebDriver` interactions. Using implicit waits is not a best practice because web browser response times are not definitely predictable and fixed elapsed times are not applicable to all interactions. Using explicit waits requires more technical sophistication, but is a Sauce Labs best practice.

Explicit Waits

Explicit waits wait until an **expected condition** occurs on the web page, or until a maximum wait time elapses. To use an explicit wait, you create an instance of the `WebDriverWait` class with a maximum wait time, and you invoke its `until` method with an expected condition.

The WebDriver API provides an `ExpectedConditions` class with methods for various standard types of expected condition. These methods return an instance of an `expected condition` class. You can pass an invocation of these standard expected-condition methods as argument values to `until` method. You can also pass - in ways that your programming language and its WebDriver API support - any function, code block or closure that returns a boolean value or an object reference to a found web element as an argument value to the `until` method. How this is done varies over programming languages, and is covered in depth in the Developing section of this documentation. The `until` method checks repeatedly, until the maximum wait time elapses, for a `true` boolean return value or a non-`null` object reference, as an indication that the expected condition has occurred.

Example

This example code illustrates how you could use either an explicit wait or an implicit wait to anticipate web browser response after submitting the login form.

Explicit Wait

```
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
WebDriverWait wait = new WebDriverWait(driver, 10); WebElement
messageElement = wait.until( ExpectedConditions.presenceOfElementLocated
(By.id("loginResponse")) );
```

Implicit Wait

```
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
```

Running Tests and Recording Test Results

Running tests and recording test results is the ultimate purpose of your test script: you run tests in an automated test script in order to **evaluate function and performance** in the AUT, without requiring human interaction.

Test Frameworks

To run test and to record test results, you use methods of a **test framework** for your programming language. There are many available test frameworks, including the frameworks in the so-called **XUnit**family, which includes:

- **JUnit** for Java
- **NUnit** for C#
- **unittest** or **pyunit** for Python
- **RUnit** for Ruby

For some programming languages, test frameworks other than those in the XUnit family are common - for example, the **RSpec** framework for Ruby. The Sauce Labs Sample Test Framework repos on GitHub contain over 60 examples of test frameworks set up to work with Sauce Labs.

Assertions

Most test frameworks implement the basic concept of an **assertion**, a method representing whether or not a logical condition holds after interaction with an AUT. Test frameworks generally declare methods whose names begin with the term `assert` and end with a term for a logical condition, e.g. `assertEquals` in JUnit. Generally, when the logical condition represented by an `assert` method does not hold, an exception for the condition is thrown. There are various approaches to using exceptions in most test frameworks. The SeleniumHQ documentation has more detailed information on using both assertions and verifications in your tests.

Recording Test Results

Recording of test results can be done in various ways, supported by the test framework or by a logging framework for the programming language, or by both together. Selenium also supports taking screenshots of web browser windows as a helpful additional type of recording. Because of the wide variations in recording technique, this beginning section omits recording, instead emphasizing a simple approach to applying a test using an `assert` method. The scripts in Instant Selenium Tests include examples of setting up reporting of test results to Sauce Labs, as do the framework scripts in the Sauce Labs Sample Test Frameworks github repos.

Example

The following example runs a test by asserting that the login response message is equal to an expected success message:

```
import junit.framework.Assert;
import junit.framework.TestCase;

WebElement messageElement     = driver.findElement(By.id("loginResponse"));
String message                 = messageElement.getText();
String successMsg             = "Welcome to foo. You logged in
successfully.";
assertEquals (message, successMsg);
```

Concluding a Test

The `quit` Method

You conclude a test by invoking the `quit` method on an instance of the `WebDriver` interface, e.g. on the `driver` variable.

The `quit` method concludes a test by disposing of resources, which allows later tests to run without resources and application state affected by earlier tests. The `quit` method:

- quits the web browser application, closing all web pages
- quits the WebDriver server, which interacts with the web browser
- releases `driver`, the variable referencing the unique instance of the `WebDriver` interface.

Example

The following example invokes the `quit` method on the `driver` variable:

```
driver.quit();
```

Example with All Steps

The following example includes code for all steps. The example also defines a Java test class `Example`, and its `main` method, so that the code can be run.

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

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

import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

import org.junit.Assert;

public class Example  {
  public static void main(String[] args) {

    // Create an instance of the driver
    WebDriver driver = new FirefoxDriver();

    // Navigate to a web page
    driver.get("http://www.foo.com");

    // Perform actions on HTML elements, entering text and submitting the
form
    WebElement usernameElement     = driver.findElement(By.name
("username"));
    WebElement passwordElement     = driver.findElement(By.name
("password"));
    WebElement formElement        = driver.findElement(By.id("loginForm"));

    usernameElement.sendKeys("Alan Smithee");
    passwordElement.sendKeys("twilightZone");

    //passwordElement.submit(); // submit by text input element
    formElement.submit();        // submit by form element


    // Anticipate web browser response, with an explicit wait
    WebDriverWait wait = new WebDriverWait(driver, 10);
    WebElement messageElement = wait.until(
           ExpectedConditions.presenceOfElementLocated(By.id
("loginResponse"))
            );

    // Run a test
    String message                = messageElement.getText();
    String successMsg            = "Welcome to foo. You logged in
successfully.";
    Assert.assertEquals (message, successMsg);

    // Conclude a test
```

```
    driver.quit();

  }
}
```

More Information

- http://www.seleniumhq.org
  The official Selenium website and documentation
- https://www.youtube.com/watch?v=qq1WCsAMZsk
  Automated testing guru Joe Colantonio demonstrates how to run a Selenium Test with Sauce Labs
- http://elementalselenium.com
  Weekly email tips on using Selenium for automated testing written by Dave Haeffner, compiled into a handy website