COURSEWARE



# Opening A Web Browser With Selenium

## Contents



## Overview

Selenium is used for automating actions on a Web page. The first step in doing this is opening a Web page in a browser.

## Adding the Maven dependency

Once appropriate WebDriver implementations have been downloaded, the Selenium Maven depenendency must be added to the project.

Create a Maven project and in the `pom.xml` add the following JUnit and Selenium dependencies:

▶ Maven Dependencies

> The JUnit and Selenium dependencies can be found on MVNRepository

## Setting up a test file

To run Selenium as a test, the JUnit 4 dependency is brought in. This is important is `WebDriver` implementations in Java are closeable resources, i.e. they need closing when we are done with them. JUnit 4 enables us to initialise and close objects before and after each test using the `@Before` and `@After` annotations on methods.

```
public class TestLayoutExample {

    private WebDriver driver;

    @Before
    public void setup() {
        System.setProperty("webdriver.chrome.driver",
"src/test/resources/chromedriver.exe");
        driver = new ChromeDriver();
    }

    @Test
    public void someTest() {
        // some test stuff
    }

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

The above example demonstrates a class with a `WebDriver` as a field, this field is initialised with a `WebDriver` implementation in the `@Before` method called `setup()`. The `WebDriver` is then closed after each test in the `tearDown()` method annotated with `@After`.

## Setting the driver implementation

As there are multiple driver implementations, it is important to select the correct driver to use for a test suite. Generally, the driver is set using the `System.setProperty(String property, String value)` static method. For example:

```
// Chrome
System.setProperty("webdriver.chrome.driver",
"src/test/resources/chromedriver.exe");
WebDriver chromeDriver = new ChromeDriver();

// Firefox
System.setProperty("webdriver.gecko.driver",
"src/test/resources/geckodriver.exe");
WebDriver firefoxDriver = new FirefoxDriver();

// Edge
System.setProperty("webdriver.edge.driver",
"src/test/resources/edgedriver.exe");
WebDriver edgeDriver = new EdgeDriver();
```

> If the driver is added to the path, you can directly create it without specifying its path as a property, i.e. just do `WebDriver driver = new ChromeDriver()` once it has been added to the path

An alternative method of setting the driver property is by passing a Maven property when running the test phase, the property will specify the driver and its associated path like `System.setProperty()`:

```
mvn test -Dwebdriver.chrome.driver=src/test/resources/chromedriver.exe
```

Navigate to Selenium's page on installing browser drivers to find out more.

## Creating a WebDriver factory method

Currently, the setup only allows for one specific kind of `WebDriver` implementation to be used even though we are technically programming to the `WebDriver` interface rather than a specific implementation like `ChromeDriver`. An important part of testing websites is *cross-browser testing*, this verifies whether a service is cross-browser compatible or not. To handle this, a `WebDriverFactory` class can be created that retrieves a specific `WebDriver` based on a passed in Maven property:

```
public class WebDriverFactory {

    public static WebDriver getDriver() throws Exception {
        // get the value of a property called "browser", or default to "chrome"
if unavailable
        String webDriver = System.getProperty("browser", "chrome");

        switch (webDriver.toUpperCase()) {
            case "CHROME":
                System.setProperty("webdriver.chrome.driver",
"src/test/resources/chromedriver.exe");
                return new ChromeDriver();
            case "FIREFOX":
                System.setProperty("webdriver.gecko.driver",
"src/test/resources/geckodriver.exe");
                return new FirefoxDriver();
            default:
                throw new Exception("[Fatal] No driver available: No browser
property supplied and could not default to ChromeDriver")
        }
    }
}
```

The above method, `getDriver()`, will look for a property called `browser` when it is invoked. This is passed in via the Maven command like so:

```
mvn test -Dbrowser=chrome
```

The method will then return an appropriate driver based on the input or throw an exception. This allows for the `setup()` methods of the test classes to be simplified to the following:

```
@Before
public void setup() {
    driver = WebDriverFactory.getDriver();
}
```

> Our test file now does not have a tight coupling between itself and a specific instance of a `WebDriver`, the test file is *loosely coupled* with a `WebDriver` instance instead by programming to its interface.

## WebDriver capabilities

In Selenium, each `WebDriver` implementation is capable of taking a `Capabilities` object as input to its constructor. This allows for experimental features of a browser to be implemented in a test, as well as offering some generic control over the drivers configuration. Implementations take the format `BrowserOptions`, i.e. `ChromeOptions` and `FirefoxOptions` for example.

```
ChromeOptions options = new ChromeOptions();
options.addArgument("disable-popup-blocking");
options.addArgument("incognito");
// options.addArgument("headless");
WebDriver driver = new ChromeDriver(options);
```

The above example blocks popups and sets the browser to start in incognito mode, the `ChromeOptions` instance is then used to initialise the new `ChromeDriver`.

## Tutorial

A simple test class is shown below:

(Place the `chromedriver.exe` inside the `src/test/resources` folder.)

▶ GooglePageTest

Here the domain `"google"` is verified to be going to the correct web page by verifying the title.

Using this strategy domains and sub-domains can be checked.

## Exercises

1. Change the URL to use the Bing search engine instead; verify the correct Web page is loaded by asserting the Web page title.
2. Using the `WebDriverFactory` class, add the ability to create an `EdgeDriver` to the `getDriver()` factory method
3. Create an overloaded `getDriver()` factory method that takes a `Capabilities` object and initialises `WebDriver` implementations using that object
4. Can you think of a way to have the `getDriver()` method call the `getDriver(Capabilities capabilities)` method to reduce code duplication? Try to implement this to help keep your code **DRY**.