Getting Starting With Selenium

TABLE OF CONTENTS

PREFACE

In this guide, we'll explore the key concepts and practical techniques you need to know to leverage the power of Selenium effectively. We'll cover everything from setting up your development environment to writing your first Selenium script and handling advanced scenarios. Throughout the journey, we'll focus on using Selenium WebDriver, the primary component of Selenium that provides a convenient API for browser automation.

INTRODUCTION

Selenium 2.0 revolutionized web application testing by providing a more robust, efficient, and user-friendly framework. Its adoption by the testing community grew rapidly, and it became the de facto standard for web UI automation. Selenium 2.0 laid the foundation for future versions of Selenium, including Selenium 3.0 and beyond, which continue to evolve and improve the testing landscape.

WHAT IS SELENIUM 2.0?

Selenium 2.0 is a major release of the Selenium framework that introduced significant improvements and enhancements over its predecessor, Selenium 1.0 (also known as Selenium RC or Remote Control). It was designed to overcome the limitations and challenges faced by Selenium 1.0 and to provide a more robust and efficient tool for web application testing.

One of the key features of Selenium 2.0 is the introduction of WebDriver, a powerful API that allows direct interaction with web browsers. WebDriver provides a more modern and intuitive approach to browser automation compared to the previous Selenium RC API. It offers a more natural and seamless way to control the browser, mimicking user actions such as clicking buttons, filling forms, navigating pages, and verifying page content.

With WebDriver, developers can write tests in multiple programming languages, including Java, C#, Python, Ruby, and more. This language binding flexibility makes Selenium 2.0 accessible to a broader range of developers and enables teams to use their preferred programming language for test automation.

In addition to WebDriver, Selenium 2.0 introduced a more modular architecture. The framework was split into separate components, each responsible for a specific aspect of web testing. This modular structure allows for better maintainability, extensibility, and scalability of the Selenium framework.

Another notable improvement in Selenium 2.0 is the support for multiple browsers through browser-specific driver implementations. Selenium 2.0 provides dedicated drivers for popular web browsers such as Firefox, Chrome, Internet Explorer, Safari, and Opera. These drivers facilitate seamless communication between Selenium and the respective browsers, enabling accurate and reliable browser automation.

Selenium 2.0 also introduced various features and capabilities to enhance the testing experience. It improved the handling of AJAX-based applications, added support for HTML5 elements and events, and enhanced the handling of pop-up dialogs and alerts. It also introduced advanced features like screenshots, timeouts, browser profiles, and more, empowering testers to tackle complex testing scenarios.

Furthermore, Selenium 2.0 supports integration with various testing frameworks and tools, allowing testers to leverage their existing test infrastructure. It can seamlessly integrate with tools like TestNG, JUnit, and Maven, enabling the execution of Selenium tests as part of a comprehensive test suite or continuous integration (CI) pipeline.

ARCHITECTURE

Selenium 2.0 follows a client-server architecture, where the client interacts with the Selenium server to control the web browser. The Selenium WebDriver API is used to communicate with different browser drivers, which in turn control the web browsers. This architecture allows Selenium to support multiple browsers and operating systems.

INSTALLATION

To get started with Selenium 2.0, you need to set up the necessary dependencies and drivers based on your programming language and browser preference. Here are the installation steps for various languages:

JAVA (MAVEN)

If you're using Maven as your build tool, you can simply add the Selenium Java bindings as a dependency in your project's `pom.xml` file. Here's an example:

```
<dependency>

<groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>2.0.0</version>
</dependency>
```

RUBY

To use Selenium 2.0 with Ruby, you need to install the Selenium Ruby gem. Open your command prompt or terminal and run the following command:

```
gem install selenium-webdriver
```

PYTHON

For Python, you can install Selenium 2.0 using pip, the Python package manager. Run the following command in your command prompt or terminal:

```
pip install selenium
```

C#

To use Selenium 2.0 with C#, you need to add the Selenium C# bindings to your project. You can download the Selenium C# bindings from the official Selenium website or include them as a NuGet package in your Visual Studio project.

DRIVER IMPLEMENTATIONS

Selenium 2.0 supports various browser drivers that enable communication between Selenium and web browsers. Each driver implementation provides a way to launch and control a specific web browser using the WebDriver API. Let's explore some commonly used driver implementations:

FIREFOX

To use Firefox with Selenium 2.0, you

need to download the Firefox driver executable and set the path to the driver in your code. Here's an example in Java:

```
System.setProperty("webdriver.gecko.driver", "path/to/geckodriver.exe");
WebDriver driver = new
FirefoxDriver();
```

INTERNET EXPLORER

To use Internet Explorer with Selenium 2.0, you need to download the Internet Explorer driver executable and set the path to the driver in your code. Here's an example in Java:

```
System.setProperty("webdriver.ie.driver", "path/to/IEDriverServer.exe");
WebDriver driver = new
InternetExplorerDriver();
```

CHROME

To use Chrome with Selenium 2.0, you need to download the Chrome driver executable and set the path to the driver in your code. Here's an example in Java:

```
System.setProperty("webdriver.chrome.driver",
"path/to/chromedriver.exe");
WebDriver driver = new
ChromeDriver();
```

HTMLUNIT

HtmlUnit is a headless browser implementation that does not require any additional drivers. It is useful for testing web applications without the need for a visible browser window. Here's an example in

Java:

```
WebDriver driver = new
HtmlUnitDriver();
```

SELENIUM RC EMULATION

Selenium 2.0 introduced Selenium RC emulation to provide backward compatibility with Selenium RC. Selenium RC was the predecessor of Selenium WebDriver and had its own set of APIs for automating web browsers. However, Selenium RC had certain limitations and was not as efficient and reliable as WebDriver.

To bridge the gap between Selenium RC and WebDriver, Selenium 2.0 introduced the concept of RC emulation. With RC emulation, you can migrate your existing Selenium RC tests to Selenium 2.0 without making significant changes to your code. This ensures that your legacy tests continue to work while taking advantage of the enhanced capabilities and performance of WebDriver.

Selenium RC emulation works by emulating the Selenium RC API using WebDriver. It translates the Selenium RC commands into their corresponding WebDriver commands, allowing you to execute your Selenium RC tests using WebDriver-based browsers.

To use Selenium RC emulation, you need to create a `RemoteWebDriver` instance and specify the URL of the Selenium server and the desired browser capabilities. Here's an example:

```
WebDriver driver = new
RemoteWebDriver(new
URL("https://localhost:4444/wd/hub")
, DesiredCapabilities.firefox());
```

In the example above, we create a `RemoteWebDriver` instance by providing the URL of the Selenium server (typically running on `localhost` at port `4444`) and specifying the desired browser capabilities (in this case, we're using Firefox).

Once the `RemoteWebDriver` instance is created, you can use it to execute your Selenium RC tests using the familiar Selenium RC commands. The WebDriver commands will be translated into their equivalent WebDriver commands behind the scenes.

For example, in Selenium RC, you might have used the `open` command to navigate to a URL:

```
selenium.open("https://www.example.c
om");
```

In Selenium 2.0 with RC emulation, you can achieve the same result using WebDriver's `get` method:

```
driver.get("https://www.example.com"
);
```

Similarly, other Selenium RC commands like `type`, `click`, `select`, etc., can be replaced with their WebDriver equivalents.

The RC emulation feature in Selenium 2.0 allows you to seamlessly transition from Selenium RC to WebDriver, ensuring your existing tests continue to function while leveraging the improved performance and stability of WebDriver. However, it's recommended to gradually update your tests to use WebDriver's native API for better maintainability and readability.

Note: Selenium RC emulation is provided for backward compatibility and is not actively developed or supported. It's recommended to migrate your tests to use WebDriver's native API to take full advantage of the latest features and improvements.

PAGE INTERACTION MODEL

The Page Interaction Model in Selenium 2.0 refers to the approach used to interact with web pages and perform actions on elements within those pages. The primary interface for interacting with web elements is the WebDriver API, which provides a rich set of methods for finding, manipulating, and verifying elements on a web page.

To interact with elements on a web page using the Page Interaction Model, you typically follow these steps:

Locating Elements

You use various locator strategies provided by WebDriver, such as `By.id()`, `By.name()`, `By.xpath()`, etc., to locate elements on the web page. These locators help you identify specific elements based on their attributes, text content, or hierarchical relationships.

Finding Elements

Once you have defined the locator strategy, you can use WebDriver's `findElement()` or `findElements()` methods to locate one or multiple elements on the page, respectively. The `findElement()` method returns the first matching element, while `findElements()` returns a list of all matching elements.

Performing Actions

After finding the desired elements, you can perform various actions on them, such as clicking on buttons, filling input fields, selecting options from dropdowns, submitting forms, etc. WebDriver provides methods like `click()`, `sendKeys()`, `submit()`, `selectByVisibleText()`, and many more to perform these actions.

Here's an example of interacting with elements using the Page Interaction Model:

```
WebElement searchBox =
driver.findElement(By.id("search-
box"));
searchBox.sendKeys("Selenium");
searchBox.submit();
```

In the above example, we first locate the search box element using its ID attribute. Then, we use the `sendKeys()` method to enter the text "Selenium" into the search box. Finally, we submit the form using the `submit()` method.

The Page Interaction Model provides a flexible and intuitive way to simulate user actions and interact with web elements during automation testing. It allows you to replicate user behavior, test user flows, and validate the expected behavior of web applications.

By combining the Page Interaction Model with other features of Selenium, such as synchronization, waits, and assertions, you can build robust and reliable tests that ensure the functionality and usability of your web applications.

Remember to always consider proper error handling and synchronization mechanisms, such as explicit waits, to ensure that your automation scripts interact with the elements at the right time and in the expected state.

Using the Page Interaction Model effectively can help you create comprehensive test cases and uncover potential issues in your web applications, providing you with confidence in the quality and performance of your software.

PAGE OBJECT PATTERN

The Page Object pattern is a widely adopted design pattern in Selenium test automation. It helps in structuring and organizing the test code by separating the test logic from the page-specific details. This pattern promotes code reusability, maintainability, and readability, making test scripts more robust and easier to maintain.

In the Page Object pattern, each web page or component of a web application is represented by a separate class called a "Page Object." This class encapsulates the elements on the page and the actions that can be performed on those elements. The main goal is to create a clear separation between the test code and the underlying implementation details of the web application.

Modularity

With the Page Object pattern, each page or component of the web application has its own dedicated class. This modularity allows for better organization and management of the test code. It also enables easy identification and modification of elements and actions specific to a particular page, without impacting other parts of the test code.

Reusability

Page Objects promote code reusability. Since each page or component has its own class, the same Page Object can be reused across multiple test cases. This eliminates code duplication and improves maintainability. Changes to the page's structure or functionality can be made in a single place, and the modifications will automatically propagate to all the tests using that Page Object.

Readability

By encapsulating the elements and actions within a Page Object, the test code becomes more readable and understandable. Test cases written using Page Objects read like natural language and serve as documentation of the application’s behavior. This makes it easier for team members, including developers and testers, to collaborate and understand the purpose and flow of the tests.

Synchronization

Page Objects provide a centralized location to handle synchronization issues. Implicit waits or explicit waits can be encapsulated within the Page Object methods, ensuring that the test cases wait for the expected state of the page before proceeding. This helps in dealing with dynamic elements, AJAX requests, and other timing-related issues.

Maintenance

As web applications evolve, their UI elements may change. With the Page Object pattern, when the structure or functionality of a page changes, the required updates can be made in a single place - the corresponding Page Object class. This reduces the maintenance effort, as modifications are localized to the affected Page Object, rather than scattered across multiple test cases.

```
public class LoginPage {
    private WebDriver driver;
    private By usernameInput =
By.id("username");
    private By passwordInput =
By.id("password");
    private By loginButton =
By.id("login-button");

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

    public void login(String
username, String password) {

driver.findElement(usernameInput).se
ndKeys(username);

driver.findElement(passwordInput).se
ndKeys(password);

driver.findElement(loginButton).clic
k();
    }
}

// Usage:
LoginPage loginPage = new
LoginPage(driver);
loginPage.login("myusername",
"mypassword");
```

XPATH SUPPORT

XPath is a powerful query language that allows you to navigate and select elements in an XML or HTML document. In the context of Selenium 2.0, XPath is commonly used to locate elements on a web page based on their attributes, text content, or hierarchical relationships.

XPath expressions are written using a combination of element names, attributes, and operators to create a path to the desired element. Selenium provides the `By.xpath()` method to locate elements using XPath expressions.

Here’s a closer look at some commonly used XPath expressions with Selenium 2.0:

ABSOLUTE XPATH

Absolute XPath expressions specify the complete path to an element from the root of the document. They start with a forward slash ("/") and traverse through the document’s elements. Here’s an example:

```
/html/body/div[1]/form/input[2]
```

RELATIVE XPATH

Relative XPath expressions are more flexible and efficient. They specify the path to an element relative to another element, making them less dependent on the document's structure. Here's an example:

```
//form/input[@name='username']
```

In the above example, the XPath expression selects the `input` element within a `form` element that has the attribute `name` with the value 'username'.

XPATH AXES

XPath axes allow you to navigate through different parts of the document, such as parent, child, sibling, and ancestor elements. Here are some common XPath axes:

Xpath Axis	Description
ancestor	Selects all ancestor elements of the current element
parent	Selects the parent element of the current element
child	Selects all direct child elements of the current element
following-sibling	Selects all sibling elements that come after the current element
preceding-sibling	Selects all sibling elements that come before the current element

XPath axes are useful when you want to select elements based on their relationship with other elements in the document.

XPATH FUNCTIONS AND PREDICATES

XPath functions and predicates provide additional capabilities to refine element selection. They allow you to apply conditions, comparisons, and calculations within XPath expressions. Here are some examples:

Function	Description
text()	Selects the text content of an element.
contains()	Checks if an element's attribute or text content contains a specific value.
starts-with()	Checks if an element's attribute or text content starts with a specific value.
position()	Returns the position of an element in a list of elements.
last()	Selects the last element in a list of elements.

XPath functions and predicates enable you to create more specific and dynamic element selections based on various criteria.

USING XPATH IN SELENIUM 2.0

To locate elements using XPath in Selenium 2.0, you can utilize the `By.xpath()` method. Here's an example of using XPath to locate an input field with a specific attribute:

```
WebElement element =
driver.findElement(By.xpath("//input
[@id='username']"));
```

In the above example, the XPath expression selects an `input` element with the attribute `id` equal to 'username'.

XPath support in Selenium 2.0 is powerful and flexible, allowing you to locate elements accurately based on different criteria. It is especially useful when working with complex web page structures or when other locating strategies, such as ID or class, are not available or reliable.

Remember to carefully construct XPath expressions to target specific elements without relying too heavily on the document's structure, as it may change over time. Regularly test and update your

XPath expressions as needed to ensure their accuracy and maintainability in your automated tests.

REMOTE WEBDRIVER

RemoteWebDriver is a class provided by Selenium 2.0 that allows you to execute your tests on remote machines. This is particularly useful when you want to run your tests on different operating systems, browsers, or devices without the need to set up the drivers locally.

By using RemoteWebDriver, you can establish a connection to a remote WebDriver server and delegate the test execution to that server. The server can be running on a different machine or even in a cloud-based infrastructure.

To use RemoteWebDriver, you need to specify the URL of the remote WebDriver server and the desired capabilities, which define the browser and platform configurations for your test. The server can handle requests from multiple clients simultaneously, making it suitable for parallel test execution.

Here’s an example of using RemoteWebDriver to connect to a remote WebDriver server:

```
WebDriver driver = new
RemoteWebDriver(new
URL("https://remote-
machine:4444/wd/hub"),
DesiredCapabilities.chrome());
```

In the example above, we create a new instance of RemoteWebDriver by providing the URL of the remote WebDriver server and the desired capabilities for the Chrome browser. The URL points to the server’s endpoint (e.g., https://remote-machine:4444/wd/hub), where the server is listening for incoming WebDriver requests.

The desired capabilities specify the browser, platform, and other configurations for the remote session. In this case, we set the desired capabilities to use Chrome.

By leveraging RemoteWebDriver, you can perform cross-browser testing by executing the same test suite on different browsers running on remote machines. This allows you to ensure the compatibility and functionality of your web application across various environments.

It’s worth noting that the remote WebDriver server should be set up and running before you can establish a connection using RemoteWebDriver. The Selenium documentation provides detailed instructions on how to set up and configure the remote server for different browsers and platforms.

RemoteWebDriver provides a flexible and scalable solution for distributed test execution, making it a valuable tool for teams working on large-scale automation projects or running tests in parallel across multiple environments.

With RemoteWebDriver, you can easily expand your test coverage by running tests on different browsers, platforms, and devices without the need to maintain a local infrastructure for each configuration. This enables efficient and comprehensive test execution while saving time and resources.

SELENIUM GRID

RemoteWebDriver is built on top of the WebDriver protocol, which defines a standardized way for different WebDriver implementations to communicate with browsers. It acts as a bridge between your test code and the WebDriver server running on a remote machine.

The remote WebDriver server can be set up in various ways, depending on your requirements. One popular choice is to use Selenium Grid, which allows you to set up a hub that manages multiple WebDriver nodes running on different machines. This setup enables parallel test execution across multiple browsers and platforms.

Here are the key steps involved in using RemoteWebDriver with Selenium Grid:

Start the Selenium Grid Hub: The hub acts as a central point for distributing test requests to the appropriate WebDriver nodes. You can start the hub by running the following command:

```
java -jar selenium-server-
```

```
standalone.jar -role hub
```

Register WebDriver Nodes: Each WebDriver node represents a specific browser and platform combination. You can start one or more WebDriver nodes by running the following command:

```
java -jar selenium-server-
standalone.jar -role node -hub
https://hub-url:4444/grid/register
```

Establish a Connection with RemoteWebDriver: In your test code, you need to create an instance of RemoteWebDriver and provide the URL of the Selenium Grid hub. Here's an example:

```
DesiredCapabilities capabilities =
DesiredCapabilities.chrome();
WebDriver driver = new
RemoteWebDriver(new
URL("https://hub-url:4444/wd/hub"),
capabilities);
```

In the example above, we create a new instance of RemoteWebDriver by specifying the URL of the Selenium Grid hub and the desired capabilities for the Chrome browser.

Execute Tests: Once the RemoteWebDriver is instantiated, you can write your test code as usual. The WebDriver commands you use will be sent to the appropriate WebDriver node based on the desired capabilities you provided.

Selenium Grid handles the distribution of test execution across available nodes, allowing you to run tests in parallel and maximize resource utilization. It automatically routes test requests to nodes that match the desired capabilities specified in the RemoteWebDriver setup.

Using RemoteWebDriver with Selenium Grid provides several benefits:

- Scalability: Selenium Grid allows you to scale your test execution by adding more WebDriver nodes. This ensures efficient utilization of resources and faster test execution.
- Cross-Browser Testing: With RemoteWebDriver and Selenium Grid, you can easily run your tests on different browsers and platforms simultaneously, enabling comprehensive cross-browser testing.
- Parallel Execution: Selenium Grid enables parallel test execution across multiple nodes, allowing you to execute multiple tests concurrently and reduce overall test execution time.
- Centralized Management: Selenium Grid provides a centralized hub to manage and distribute test execution. This simplifies the setup and maintenance of your test infrastructure.

It's important to note that when using RemoteWebDriver with Selenium Grid, you need to ensure that the appropriate WebDriver binaries and drivers are installed and configured on the remote machines. The WebDriver binaries must match the desired browser and version specified in the desired capabilities.

By leveraging RemoteWebDriver and Selenium Grid, you can create a robust and scalable test infrastructure that supports distributed test execution across various browsers, platforms, and devices. This enables you to achieve comprehensive test coverage and efficient test execution in a parallel and distributed testing environment.

MOBILE DEVICE SUPPORT

Selenium 2.0 provides a way to automate mobile web applications by integrating with the Appium framework. Appium is an open-source tool that extends the capabilities of Selenium WebDriver to support mobile application testing. With Appium, you can write tests in your preferred programming language and use the WebDriver API to interact with mobile apps.

SETTING UP APPIUM

To get started with mobile device support in Selenium 2.0, you need to set up the Appium server and configure the desired capabilities for the target device or emulator. Here are the general steps to set up Appium:

- Install Node.js: Appium requires Node.js to run. You can download and install Node.js from the official Node.js website (https://nodejs.org).

- Install Appium: Open your command prompt or terminal and run the following command to install Appium globally:

```
npm install -g appium
```

<ol start="3"> * Install Appium Dependencies: Appium relies on various dependencies, such as the Android SDK for Android devices and Xcode for iOS devices. Make sure to install these dependencies based on the platform you want to test. * Start Appium Server: Once the dependencies are installed, start the Appium server by running the following command in your command prompt or terminal:

```
appium
```

This will start the Appium server and allow it to listen for incoming WebDriver requests.

CONFIGURING DESIRED CAPABILITIES

Before running your mobile tests with Selenium 2.0, you need to configure the desired capabilities to specify the target device or emulator, platform, and other relevant settings. Here's an example of configuring desired capabilities for Android:

```
DesiredCapabilities capabilities =
new DesiredCapabilities();
capabilities.setCapability("deviceNa
me", "Android Device");
capabilities.setCapability("platform
Name", "Android");
capabilities.setCapability("browserN
ame", "Chrome");
```

In this example, we set the `deviceName` capability to specify the target Android device or emulator. The `platformName` capability indicates the target mobile platform (Android or iOS), and the `browserName` capability specifies the browser to use for testing (in this case, Chrome).

RUNNING MOBILE TESTS

Once the Appium server is up and running, and you have configured the desired capabilities, you can run your mobile tests using Selenium 2.0. Here's an example of initializing the WebDriver with Appium:

```
WebDriver driver = new
RemoteWebDriver(new
URL("https://localhost:4723/wd/hub")
, capabilities);
```

In this example, we create an instance of the `RemoteWebDriver` class and provide the URL of the Appium server along with the desired capabilities.

From here, you can use the WebDriver API to interact with the mobile web application, just like you would with desktop web testing. You can find elements, perform actions, and verify the expected behavior of your mobile app.

APPIUM FEATURES

Appium provides additional features and capabilities to enhance your mobile testing experience. Some notable features include:

- **Cross-platform support**: Appium allows you to write tests in a single language (such as Java, Python, or JavaScript) and run them on both Android and iOS devices, making it convenient for cross-platform testing.
- **Support for native apps**: In addition to mobile web applications, Appium also supports automating native mobile apps. You can use the same WebDriver API to interact with native app elements and perform actions.
- **Built-in gestures and interactions**: Appium provides built-in support for common mobile gestures, such as swiping, tapping, and scrolling. These gestures can be easily incorporated into your test scripts to simulate user interactions.
- **Integration with cloud-based testing services**: Appium can be integrated with cloud-based testing services, allowing you to run your mobile tests on a wide range of real devices hosted in the cloud. This enables you to cover a broader spectrum of devices and

configurations without the need for physical devices.

LIMITATIONS AND CONSIDERATIONS

While mobile device support in Selenium 2.0 and Appium is powerful, there are a few limitations and considerations to keep in mind:

- **Device-specific behavior**: Mobile devices can exhibit different behaviors, screen sizes, and capabilities. It's crucial to consider device-specific variations and test on a diverse range of devices to ensure optimal app performance.
- **Performance and stability**: Emulators and simulators may not fully replicate the behavior and performance of real devices. It's recommended to perform testing on real devices whenever possible to obtain more accurate results.
- **Appium server and dependencies**: Appium relies on the Appium server and various dependencies, such as the Android SDK or Xcode. Keeping these components up to date and ensuring compatibility with your testing environment is important.
- **Appium version compatibility**: Appium is under active development, and new versions may introduce changes or deprecate certain features. Ensure compatibility between your Appium version, Selenium version, and other related libraries.

By considering these factors and leveraging the capabilities of Appium, you can effectively automate the testing of mobile web applications and ensure their functionality across different mobile platforms.