

# Selcukes

Ramesh Babu Prudhvi

Version 1.6.0, 2022-04-24T01:47:48Z

# Table of Contents

1. Selcukes Core .....	1
1.1. Overview .....	1
1.2. Features .....	1
1.3. First Steps .....	2
2. WebDriver Binaries .....	4
2.1. Motivation .....	4
2.2. Setup .....	5
2.3. Driver Management .....	5
2.4. Advanced Configuration .....	7
3. Selcukes DataBind .....	8
3.1. Setup .....	8
3.2. Data Parser .....	8
4. Community .....	9
5. Support .....	10

# Chapter 1. Selcukes Core

Selcukes Core can help your team to build high-quality and highly scalable automated tests.

## 1.1. Overview

### Open Source

As an open source company, we're passionately engaged in numerous projects, initiatives and standards where we represent the needs and requirements of our many customers and partners.

### Unified Framework API

All features that we provide- use the same syntax. Once you learn how to write tests for the web, you can start immediately writing for mobile, desktop, or REST.

### Test Everything

Web, including responsive, iOS, Android, Desktop, and REST services.

### Integrations

Seamlessly integrate the framework with your existing tools and processes. Execute tests in the clouds, distributed and publish test results in reporting solutions.

## 1.2. Features

1. Easy add new logic to your tests without causing regression issues. Control the whole execution workflow - change browsers or reuse them. Retry your failing tests to make sure that there is a real problem.
2. Automate UI actions or user scenarios on real devices or emulators with Selcukes features from test creation to execution.
3. Cross-platform screenshot and video recording on test failure


What can be tested?	Selcukes	Benefits
<b>Web automation cross browser</b> - Chrome, Firefox, Edge, IE, Opera, Chrome Headless, Firefox Headless.		The standard for web automation. Open source. Large community. All major cloud platforms support it. Browser vendors support.
<b>Mobile Automation</b> - iOS, Android native, hybrid and mobile web apps.		The standard for mobile automation. Open source. Large community. Compatible with major cloud providers. Can run in Selenium Grid.
<b>Desktop Automation</b> - WPF, WinForms and Universal applications.	WinAppDriver	Officially supported by Microsoft. Can run in Selenium Grid.
<b>API Automation</b> - REST web services.	RestAssured	The standard for API automation in Java. Open source. Large community.
	Java	Open source. Run tests on Windows, Linux and MacOS.
	TestNG, JUnit	Rich set of assert libraries. Open source. Large community.
	Support all major cloud providers - CrossBrowserTesting, SauceLabs, Browserstack (mobile,web)	
	Easily integrated with all CI systems	

Figure 1. Technologies

## 1.3. First Steps

What you need to start

### 1.3.1. Java 11

Test Automation is a development activity, so you will need some familiarity with Java. Selcukes uses Java 11, so make sure you have a [JDK 11 or later](#) installed.

### 1.3.2. An Integrated Development Environment

You will need a modern IDE to work with Java. We recommend IntelliJ (you can download the Community Edition, which is free, [from here](#). But Eclipse will work fine as well.

### 1.3.3. A Build Tool

You will also need a build tool, either Gradle or Maven, to run your tests and generate your reports. Make sure you have either [Gradle 3.x or higher](#) or [Maven 3.3.x or higher](#) installed.

### 1.3.4. Getting Started

The quickest way to create a new project is to take one of the starter projects on Github. You can find the starter project for TestNG at <https://github.com/selcukes/selcukes-java-skeleton>.

You can clone this repository:

```
git clone https://github.com/selcukes/selcukes-java-skeleton.git
cd selcukes-java-skeleton
```

Or simply [download a zip file from here](#).

If you are using Maven, you can also create a new project using one of the Serenity [Maven Archetypes](#).

### 1.3.5. So far so good?

The starter project comes with a demo test that you can run. From the command line, run either

```
$ mvn clean verify
```

or

```
$ gradle clean test
```

# Chapter 2. WebDriver Binaries

**WebDriver Binaries** is an open-source Java library that automatically downloads and configures the binary drivers (e.g., chromedriver, geckodriver, msedgedriver, etc.) required by Selenium WebDriver.

## 2.1. Motivation

**Selenium WebDriver** is a library that allows controlling web browsers programmatically. It provides a cross-browser API that can be used to drive web browsers (e.g., Chrome, Edge, or Firefox, among others) using different programming languages (e.g., Java, JavaScript, Python, C#, or Ruby). The primary use of Selenium WebDriver is implementing automated tests for web applications.

Selenium WebDriver carries out the automation using the native support of each browser. For this reason, we need to place a binary file called *driver* between the test using the Selenium WebDriver API and the browser to be controlled. Examples of drivers for major web browsers nowadays are **chromedriver** (for Chrome), **geckodriver** (for Firefox), or **msedgedriver** (for Edge). As you can see in the following picture, the communication between the WebDriver API and the driver binary is done using a standard protocol called **W3C WebDriver** (formerly the so-called *JSON Wire Protocol*). Then, the communication between the driver and the browser is done using the native capabilities of each browser.

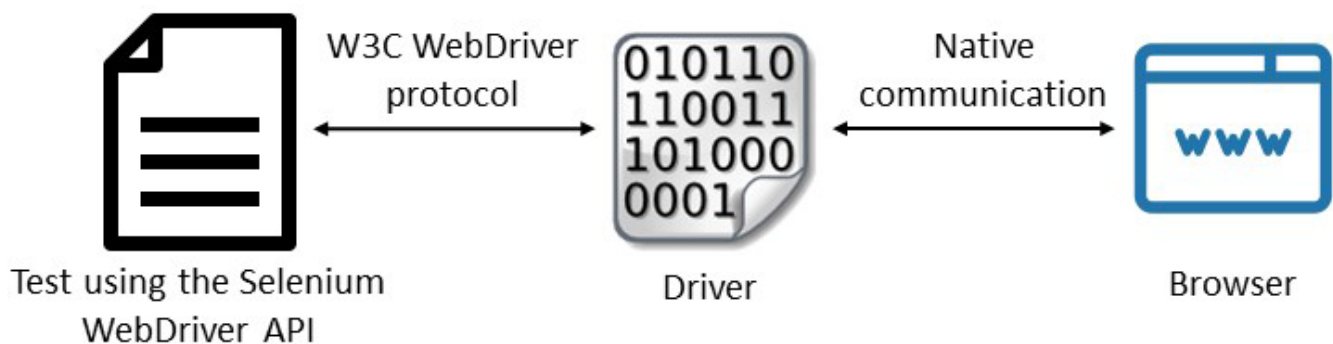


Figure 2. Selenium WebDriver Architecture

From a practical point of view, we need to make a *driver management process* to use Selenium WebDriver. This process consists on:

1. **Download.** Drivers are platform-specific binary files. To download the proper driver, we have to identify the driver type we need (e.g., chromedriver if we want to use Chrome), the operating system (typically, Windows, Linux, or Mac OS), the architecture (typically, 32 or 64 bits), and very important, the driver version. Concerning the version, each driver release is usually compatible with a given browser version(s). For this reason, we need to discover the correct driver version for a specific browser release (typically reading the driver documentation or release notes).
2. **Setup.** Once we have downloaded the driver to our computer, we need to provide a way to locate this driver from our Selenium WebDriver tests. In Java, this setup can be done in two different ways. First, we can add the driver location to our **PATH** environmental variable. Second, we can use *Java system properties* to export the driver path. Each driver path should be

identified using a given system property, as follows:

```
System.setProperty("webdriver.chrome.driver", "/path/to/chromedriver");
System.setProperty("webdriver.gecko.driver", "/path/to/geckodriver");
System.setProperty("webdriver.edge.driver", "/path/to/msedgedriver");
System.setProperty("webdriver.opera.driver", "/path/to/operadriver");
System.setProperty("webdriver.ie.driver", "C:/path/to/IEDriverServer.exe");
```

3. Maintenance. Last but not least, we need to warranty the compatibility between driver and browser in time. This step is relevant since modern browsers automatically upgrade themselves (i.e., they are *evergreen* browsers), and for this reason, the compatibility driver-browser is not warranted in the long run. For instance, when a WebDriver test using Chrome faces a driver incompatibility, it reports the following error message: *"this version of chromedriver only supports chrome version N."* As you can see in [StackOverflow](#), this is a recurrent problem for manually managed drivers (chromedriver in this case).

## 2.2. Setup

WebDriver Binaries is primarily used as a Java dependency . We typically use a *build tool* (such as [Maven](#) or [Gradle](#)) to resolve the WebDriverManager dependency. In Maven, it can be done as follows (notice that it is declared using the `test` scope, since it is typically used in tests classes):

```
<dependency>
  <groupId>io.github.selcukes</groupId>
  <artifactId>webdriver-binaries</artifactId>
  <version>1.6.0</version>
  <scope>test</scope>
</dependency>
```

In the case of a Gradle project, we can declare WebDriverManager as follows (again, for tests):

```
dependencies {
    testImplementation("io.github.selcukes:webdriver-binaries:1.6.0")
}
```

## 2.3. Driver Management

The primary use of WebDriver Binaries is the automation of driver management. For using this feature, you need to select a given driver in the WebDriver Binaries API (e.g., `chromeDriver()` for Chrome) and invoke the method `setup()`. The following example shows a test case using [JUnit 5](#), Selenium WebDriver, WebDriver Binaries. In this test, we invoke WebDriver Binaries in the setup method for all tests (`@BeforeAll`). This way, the required binary (chromeDriver) will be available for all the WebDriver tests using Chrome in this class.

```

public class BrowserTest {
    WebDriver driver;

    @BeforeClass
    static void setupClass() {
        WebDriverBinary.chromeDriver().setup();
    }

    @BeforeTest
    void setupTest() {
        driver = new ChromeDriver();
    }

    @AfterTest
    void teardown() {
        driver.quit();
    }

    @Test
    void test() {
        driver.get("https://google.com");
        String title = driver.getTitle();
        Assert.assertEquals(title, "Google");
    }
}

```

WebDriver Binaries provides a set of *binaries* for Chrome, Firefox, Edge, Opera, Chromium, and Internet Explorer. The basic use of these binary is the following:

```

WebDriverBinary.chromeDriver().setup();
WebDriverBinary.firefoxDriver().setup();
WebDriverBinary.ieDriver().setup();
WebDriverBinary.edgeDriver().setup();
WebDriverBinary.edgeDriver().setup();
WebDriverBinary.operaDriver().setup();

```

### 2.3.1. Resolution Algorithm

WebDriver Binaries executes a *resolution algorithm* when calling to `setup()` in a given manager. The most relevant parts of this algorithm are the following:

1. WebDriverBinary tries to find the browser version. To this aim, WebDriverBinary uses internally a knowledge database called commands database. This database is a collection of shell commands used to discover the version of a given browser in the different operating systems (e.g., `google-chrome --version` for Chrome in Linux).
2. Using the browser version, it tries to find the proper driver version. This process is different for each browser. In Chrome and Edge, their respective drivers (chromedriver and msedgedriver)



maintainers also publish resources to identify the suitable driver version for a given major browser release. For instance, to find out the version of chromedriver required for Chrome 89, we need to read the following [file](#). Unfortunately, this information is not available in other browsers (e.g., Firefox and Opera) or older versions of Chrome and Firefox. For this reason, WebDriverManager uses another knowledge database called [versions database](#). This database maps the browser releases with the known compatible driver versions.

3. Once the driver version is discovered, WebDriverManager downloads this driver to a local cache (located at `%temp%webdriver` by default). These drivers are reused in subsequent calls.
4. Finally, WebDriverManager exports the driver path using Java system properties (e.g., `webdriver.chrome.driver` in the case of the Chrome manager).

This process automated the first two stages of the driver management previously introduced, i.e., download and setup. To support the third stage (i.e., maintenance), WebDriverManager implements *resolution cache*. This cache (called by default `resolution.properties` and stored in the root of the driver cache) is a file that stores the relationship between the resolved driver and browser versions. This relationship is valid during a given *time-to-live* (TTL). The default value for this TTL is 1 hour for browsers and 1 day for drivers. In other words, the discovered browser version is valid for 1 hour, and the driver version is considered correct for 1 day. This mechanism improves the performance dramatically since the second (and following) calls to the resolution algorithm for the same browser are resolved using only local resources (i.e., without using the shell nor requesting external services).

## 2.4. Advanced Configuration

WebDriver Binaries provides different ways of configuration. First, by using its *Java API*. To that aim, each manager (e.g., `chromeDriver()`, `firefoxDriver()`, etc., allows to concatenate different methods of this API to specify custom options or preferences. For example (the explanation of these methods and the other possibilities are explained in the tables at the end of this section):

```
WebDriverBinary.firefoxDriver().version("v0.26.0").setup();
WebDriverBinary.chromeDriver().targetPath("temp").setup();
WebDriverBinary.firefoxDriver().arch32().setup();
```

# Chapter 3. Selcukes DataBind

Selcukes DataBind helps to parse Json and Yml files

## 3.1. Setup

Selcukes DataBind used as a Java dependency in Maven, it can be done as follows

```
<dependency>
  <groupId>io.github.selcukes</groupId>
  <artifactId>selcukes-databind</artifactId>
  <version>1.6.0</version>
</dependency>
```

In the case of a Gradle project, we can declare Selcukes DataBind as follows:

```
dependencies {
    implementation("io.github.selcukes:selcukes-databind:1.6.0")
}
```

## 3.2. Data Parser

# Chapter 4. Community

There are two ways to try to get community support related to Selcukes. First, questions about it can be discussed in [StackOverflow](#), using the tag *selcukes\_java*. In addition, comments, suggestions, and bug-reporting should be made using the [GitHub issues](#). Finally, if you think Selcukes can be enhanced, consider contributing to the project through a [pull request](#).

# Chapter 5. Support

[Selcukes](#) is part of [OpenCollective](#), an online funding platform for open and transparent communities. You can support the project by contributing as a backer (i.e., a personal [donation](#) or [recurring contribution](#)) or as a [sponsor](#) (i.e., a recurring contribution by a company).