WebDriverManager

Boni García

Version 5.1.0, 17-02-2022

Table of Contents

Chapter 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.

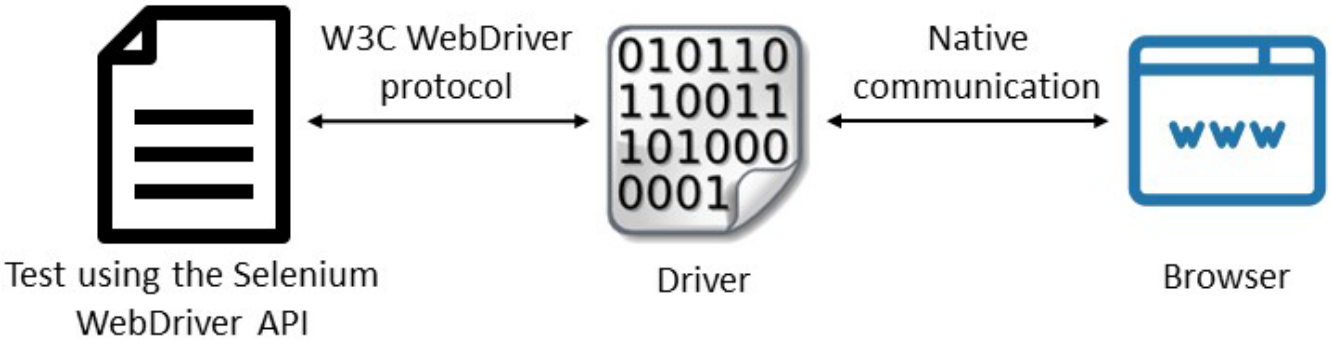

Figure 1. Selenium WebDriver Architecture

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).

What is WebDriverManager?

WebDriverManager is an open-source Java library that carries out the management (i.e., download, setup, and maintenance) of the drivers required by Selenium WebDriver (e.g., chromedriver, geckodriver, msedgedriver, etc.) in a fully automated manner. In addition, as of version 5, WebDriverManager provides other relevant features, such as the capability to discover browsers installed in the local system, building WebDriver objects (such as `ChromeDriver`, `FirefoxDriver`, `EdgeDriver`, etc.), and running browsers in Docker containers seamlessly.

Chapter 2. Setup

WebDriverManager is primarily used as a Java dependency (although other usages are also possible). 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.bonigarcia</groupId>
    <artifactId>webdrivermanager</artifactId>
    <version>5.1.0</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.bonigarcia:webdrivermanager:5.1.0")
}
```

Chapter 3. Features

WebDriverManager provides a fluent API available using the class `WebDriverManager` (package `io.github.bonigarcia.wdm`). This class provides a group of static methods to create *managers*, i.e., objects devoted to providing automated driver management and other features.

3.1. Driver Management

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

```
import static org.assertj.core.api.Assertions.assertThat;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

import io.github.bonigarcia.wdm.WebDriverManager;

class ChromeTest {

    WebDriver driver;

    @BeforeAll
    static void setupClass() {
        WebDriverManager.chromedriver().setup();
    }

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

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

    @Test
    void test() {
        // Exercise
        driver.get("https://bonigarcia.dev/selenium-webdriver-java/");
        String title = driver.getTitle();

        // Verify
        assertThat(title).contains("Selenium WebDriver");
    }

}
```

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

```
WebDriverManager.chromedriver().setup();
WebDriverManager.firefoxdriver().setup();
WebDriverManager.edgedriver().setup();
WebDriverManager.operadriver().setup();
WebDriverManager.chromiumdriver().setup()
WebDriverManager.iedriver().setup();
```

As of version 5, WebDriverManager also provides a manager for Safari (called `safaridriver()`). The case of the Safari browser is particular since this browser does not require to manage its driver to work with Selenium WebDriver (in other words, the Safari driver is built-in within the browser). Nevertheless, WebDriverManager provides this manager to be used in the WebDriver builder (especially with Docker).

Although not mandatory, it is highly recommended to use a logger library to trace your application and tests. In the case of WebDriverManager, you will see the relevant steps of the driver management following its traces. See for example the following tutorial to use SLF4J and Logback. Also, you can see an example of a WebDriverManager test using logging here (this example uses this configuration file).

3.1.1. Resolution Algorithm

WebDriverManager executes a *resolution algorithm* when calling to `setup()` in a given manager. You can find all its internal details in the paper Automated driver management for Selenium WebDriver, published in the Springer Journal of Empirical Software Engineering in 2021. The most relevant parts of this algorithm are the following:

1. WebDriverManager tries to find the browser version. To this aim, WebDriverManager 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 `~/.cache/selenium` 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).

3.1.2. Generic Manager

WebDriverManager provides a *generic manager*, i.e., a manager that can be parameterized to act as a specific manager (for Chrome, Firefox, etc.). You can create a manager using this feature using the method `getInstance()` of the WebDriverManager API. The method can be invoked using the following options:

- `getInstance(Class<? extends WebDriver> webDriverClass)`: Where `webDriverClass` is a class of the Selenium WebDriver standard hierarchy, such as `ChromeDriver.class`, `FirefoxDriver.class`, etc.
- `getInstance(DriverManagerType driverManagerType)`: Where `driverManagerType` is an enumeration provided by WebDriverManager to identify the available managers.
- `getInstance(String browserName)`: Where `browserName` is the usual browser name as `String` (i.e., `"Chrome"`, `"Firefox"`, `"Edge"`, `"Opera"`, `"Chromium"`, `"Safari"`, or `"IExplorer"`).
- `getInstance()`: If no parameter is specified, the configuration key `wdm.defaultBrowser` is used to select the manager (Chrome by default). See the advanced configuration section for further information about the configuration capabilities of WebDriverManager.

The following example shows a JUnit 5 parameterized test in which the test is repeated twice (using the classes `ChromeDriver.class` and `FirefoxDriver.class` as test parameters). As you can see, WebDriverManager uses this parameter to instantiate the proper manager and create the `WebDriver` instance (see WebDriver Builder section for more information about this feature).

```
import static org.assertj.core.api.Assertions.assertThat;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

import io.github.bonigarcia.wdm.WebDriverManager;

class GenericTest {

    WebDriver driver;

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

    @ParameterizedTest
    @ValueSource(classes = { ChromeDriver.class, FirefoxDriver.class })
    void test(Class<? extends WebDriver> webDriverClass) {
        // Driver management and WebDriver instantiation
        driver = WebDriverManager.getInstance(webDriverClass).create();

        // Exercise
        driver.get("https://bonigarcia.dev/selenium-webdriver-java/");
        String title = driver.getTitle();

        // Verify
        assertThat(title).contains("Selenium WebDriver");
    }

}
```

3.1.3. Advanced Settings

The driver management provided by WebDriverManager can be tuned in many different ways. For example, the WebDriverManager API allows configuring the driver (or browser) version to be resolved, the location of the cache path, the operating system, the architecture, the proxy settings (for the network connection), the TTL, or cleaning the driver and resolution cache, among many other attributes. See the advanced configuration settings for specific details about it. In addition, there are several WebDriverManager API specific to driver management, namely:

- `getDownloadedDriverPath()`: Used to find out the path of the resolved driver in the current instance of WebDriverManager.
- `getDownloadedDriverVersion()`: Use to find out the version of the driver resolved in the current

instance of WebDriverManager.

- `getDriverVersions()`: Used to find the list of available driver versions in a given manager.
- `getDriverManagerType()`: Used to get the driver manager type (and `enum`) of a given manager.

Each manager was a singleton object in older WebDriverManager releases (e.g., 4.x), while in version 5, a new manager instance is created each time. Therefore, the usage of `getDownloadedDriverPath()` and `getDownloadedDriverVersion()` can be different in WebDriverManager 5 (i.e., these methods need to be invoked using a WebDriverManager instance previously created).

3.2. Browser Finder

As of version 5, WebDriverManager allows detecting if a given browser is installed or not in the local system. To this aim, each manager provides the method `getBrowserPath()`. This method returns an `Optional<Path>`, which is empty if a given browser is not installed in the system or the browser path (within the optional object) when detected.

The following example shows an example using this feature. In this test, the optional browser path is used to disable conditionally (i.e., skip) the test using an AssertJ assumption (although other built-in assumptions available in JUnit 5 or other unit testing frameworks are also possible). All in all, this test should be executed in a Mac OS system (which should have Safari out of the box), but it should be skipped in any other operating system.

```
import static org.assertj.core.api.Assertions.assertThat;
import static org.assertj.core.api.Assumptions.assumeThat;

import java.nio.file.Path;
import java.util.Optional;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.safari.SafariDriver;

import io.github.bonigarcia.wdm.WebDriverManager;

class SafariTest {

    WebDriver driver;

    @BeforeAll
    static void setupClass() {
        Optional<Path> browserPath = WebDriverManager.safaridriver()
                .getBrowserPath();
        assumeThat(browserPath).isPresent();
    }

    @BeforeEach
    void setupTest() {
        driver = new SafariDriver();
    }

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

    @Test
    void test() {
        // Exercise
        driver.get("https://bonigarcia.dev/selenium-webdriver-java/");
        String title = driver.getTitle();

        // Verify
        assertThat(title).contains("Selenium WebDriver");
    }
}
```

Internally, WebDriverManager uses the content of the commands database to detect the possible browser paths in different operating systems.

3.3. WebDriver Builder

As of version 5, WebDriverManager allows instantiating `WebDriver` objects (e.g. `ChromeDriver`, `FirefoxDriver`, etc.) using the WebDriverManager API. This feature is available using the method `create()` of each manager. The following example shows a test using this feature. Notice that the WebDriverManager call to the `setup()` method is not required when using this feature since the driver management is done internally by WebDriverManager. WebDriverManager provides the method `quit()` to close the created `WebDriver` instances gracefully to complement this feature.

```
import static org.assertj.core.api.Assertions.assertThat;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.openqa.selenium.WebDriver;

import io.github.bonigarcia.wdm.WebDriverManager;

class ChromeCreateTest {

    WebDriver driver;

    @BeforeEach
    void setupTest() {
        driver = WebDriverManager.chromedriver().create();
    }

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

    @Test
    void test() {
        driver.get("https://bonigarcia.dev/selenium-webdriver-java/");
        assertThat(driver.getTitle()).contains("Selenium WebDriver");
    }

}
```

When using this feature, WebDriverManager stores internally a reference to the `WebDriver` objects created. In addition, a shutdown hook watches these objects are correctly released before shutting down the JVM. You can play with this feature removing the `teardown` method of the example before.

The WebDriverManager API provides different methods to enhance the creation of `WebDriver` objects, such as:

- Integer parameter in the method `create()`. This option is used to create a list of `WebDriver` objects instead of a single instance. See example.
- Method `capabilities()`: To specify WebDriver `Capabilities` (see example).
- Method `remoteAddress()`: To specify the remote URL in the `RemoteWebDriver` instance, typically when using a Selenium Server or a cloud provider (such as Sauce Labs, LambdaTest, etc.). This method is equivalent to the configuration key `wdm.remoteAddress` (see configuration section). The following example shows a test using this method. Notice that this test starts Selenium Grid in standalone mode before the test. To that, WebDriverManager is also used (since the browser controlled by Selenium Grid also requires a driver):

```
import static org.assertj.core.api.Assertions.assertThat;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.grid.Main;

import io.github.bonigarcia.wdm.WebDriverManager;

class ChromeRemoteTest {

    WebDriver driver;

    @BeforeAll
    static void setupClass() {
        // Resolve driver for Selenium Grid
        WebDriverManager.chromedriver().setup();

        // Start Selenium Grid in standalone mode
        Main.main(new String[] { "standalone", "--port", "4444" });
    }

    @BeforeEach
    void setupTest() {
        driver = WebDriverManager.chromedriver()
                .remoteAddress("http://localhost:4444/wd/hub").create();
    }

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

    @Test
    void test() {
        driver.get("https://bonigarcia.dev/selenium-webdriver-java/");
        assertThat(driver.getTitle()).contains("Selenium WebDriver");
    }

}
```

3.4. Browsers in Docker

Another relevant new feature available in WebDriverManager 5 is the ability to create browsers in Docker containers out of the box. The requirement to use this feature is to have installed a Docker Engine in the machine running the tests. To use it, we need to invoke the method `browserInDocker()` in conjunction with `create()` of a given manager. This way, WebDriverManager pulls the image

from Docker Hub, starts the container, and instantiates the `WebDriver` object to use it. The following test shows a simple example using Chrome in Docker:

```
import static org.assertj.core.api.Assertions.assertThat;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.openqa.selenium.WebDriver;

import io.github.bonigarcia.wdm.WebDriverManager;

class DockerChromeTest {

    WebDriver driver;

    WebDriverManager wdm = WebDriverManager.chromedriver().browserInDocker();

    @BeforeEach
    void setupTest() {
        driver = wdm.create();
    }

    @AfterEach
    void teardown() {
        wdm.quit();
    }

    @Test
    void test() {
        driver.get("https://bonigarcia.dev/selenium-webdriver-java/");
        assertThat(driver.getTitle()).contains("Selenium WebDriver");
    }

}
```

When using browsers in Docker containers, the call to the method `quit()` of the WebDriverManager API allows the disposal of the browser container(s).

The used Docker images by WebDriverManager have been created and maintained by Aerokube (you can check the available versions on the browser images page). Therefore, the available browsers to be executed as Docker containers in WebDriverManager are Chrome (like the previous example), Firefox, Edge, Opera, and Safari.

The case of Safari is particular since a real Safari browser can only be executed under a Mac OS machine. This way, the *Safari* Docker containers use the WebKit engine. This engine is the same used in browser containers, and therefore, from a functional point of view, both browsers (a real Safari and this Docker image) should behave in the same way.

In addition to these browsers, there is one more alternative: Chrome Mobile (i.e., Chrome on an Android device). To use this browser, you need to invoke the method `browserInDockerAndroid()` of a Chrome manager, just like in this example.

Notice you will need hardware virtualization (hypervisor) or a virtual machine with nested virtualization support to run Chrome Mobile images.

3.4.1. Browser Versions

A significant aspect of the browser containers presented so far is that WebDriverManager connects to Docker Hub to discover the latest available release when the browser version is not specified (like the examples explained before). This way, the *dockerized* browsers of tests handled by WebDriverManager are auto-maintained, in the sense that these tests use the latest version available without any additional effort.

Nevertheless, we can force a given browser version in Docker using the method `browserVersion()` of the WebDriverManager API. This method accepts a `String` parameter specifying the version. This version can be fixed (e.g., `91.0`), and it also accepts the following wildcards:

- `"latest"` : To specify the latest version explicitly (default option).
- `"latest-N"` : Where `N` is an integer value to be subtracted from the current stable version. For example, if we specify `latest-1` (i.e., *latest version minus one*), the previous version to the stable release will be used (see an example here).
- `"beta"`: To use the beta version. This version is only available for Chrome and Firefox, thanks to the Docker images maintained by Twilio (a fork of the Aerokube images for the beta and development versions of Chrome and Firefox).
- `"dev"`: To use the development version (again, for Chrome and Firefox).

The following example shows a test using Chrome beta in Docker (see a similar example using Firefox dev here).

```
import static io.github.bonigarcia.wdm.WebDriverManager.isDockerAvailable;

import static org.assertj.core.api.Assertions.assertThat;
import static org.assertj.core.api.Assumptions.assumeThat;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.openqa.selenium.WebDriver;

import io.github.bonigarcia.wdm.WebDriverManager;

class DockerChromeBetaTest {

    WebDriver driver;

    WebDriverManager wdm = WebDriverManager.chromedriver().browserInDocker()
            .browserVersion("beta");

    @BeforeEach
    void setupTest() {
        assumeThat(isDockerAvailable()).isTrue();
        driver = wdm.create();
    }

    @AfterEach
    void teardown() {
        wdm.quit();
    }

    @Test
    void test() {
        driver.get("https://bonigarcia.dev/selenium-webdriver-java/");
        assertThat(driver.getTitle()).contains("Selenium WebDriver");
    }

}
```

3.4.2. Remote Desktop

A possible inconvenience of using browsers in Docker is that we cannot see what is happening inside the container by default. To improve this situation, WebDriverManager allows connecting to the remote desktop session simply invoking the method `enableVnc()` of a dockerized browser. When using this option, two different technologies are used internally:

- Virtual Network Computing (VNC), a graphical desktop sharing system. In WebDriverManager, a VNC server is started in the browser container.
- noVNC, a open-source web-based VNC client. In WebDriverManager, a custom noVNC Docker image is used to connect through noVNC.

The following example shows a test that enables this feature. WebDriverManager writes the noVNC URL in the `INFO` trace logs. In addition, as shown in this example, this URL can be found by invoking the method `getDockerNoVncUrl()`. We can use this URL to inspect and interact with the browser during the test execution (as shown in the following picture).

```
import static org.assertj.core.api.Assertions.assertThat;

import java.net.URL;
import java.time.Duration;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.openqa.selenium.WebDriver;

import io.github.bonigarcia.wdm.WebDriverManager;

class DockerChromeVncTest {

    WebDriver driver;

    WebDriverManager wdm = WebDriverManager.chromedriver().browserInDocker()
            .enableVnc();

    @BeforeEach
    void setupTest() {
        driver = wdm.create();
    }

    @AfterEach
    void teardown() {
        wdm.quit();
    }

    @Test
    void test() throws Exception {
        driver.get("https://bonigarcia.dev/selenium-webdriver-java/");
        assertThat(driver.getTitle()).contains("Selenium WebDriver");

        // Verify URL for remote session
        URL noVncUrl = wdm.getDockerNoVncUrl();
        assertThat(noVncUrl).isNotNull();

        // Pause for manual inspection
        Thread.sleep(Duration.ofSeconds(60).toMillis());
    }

}
```

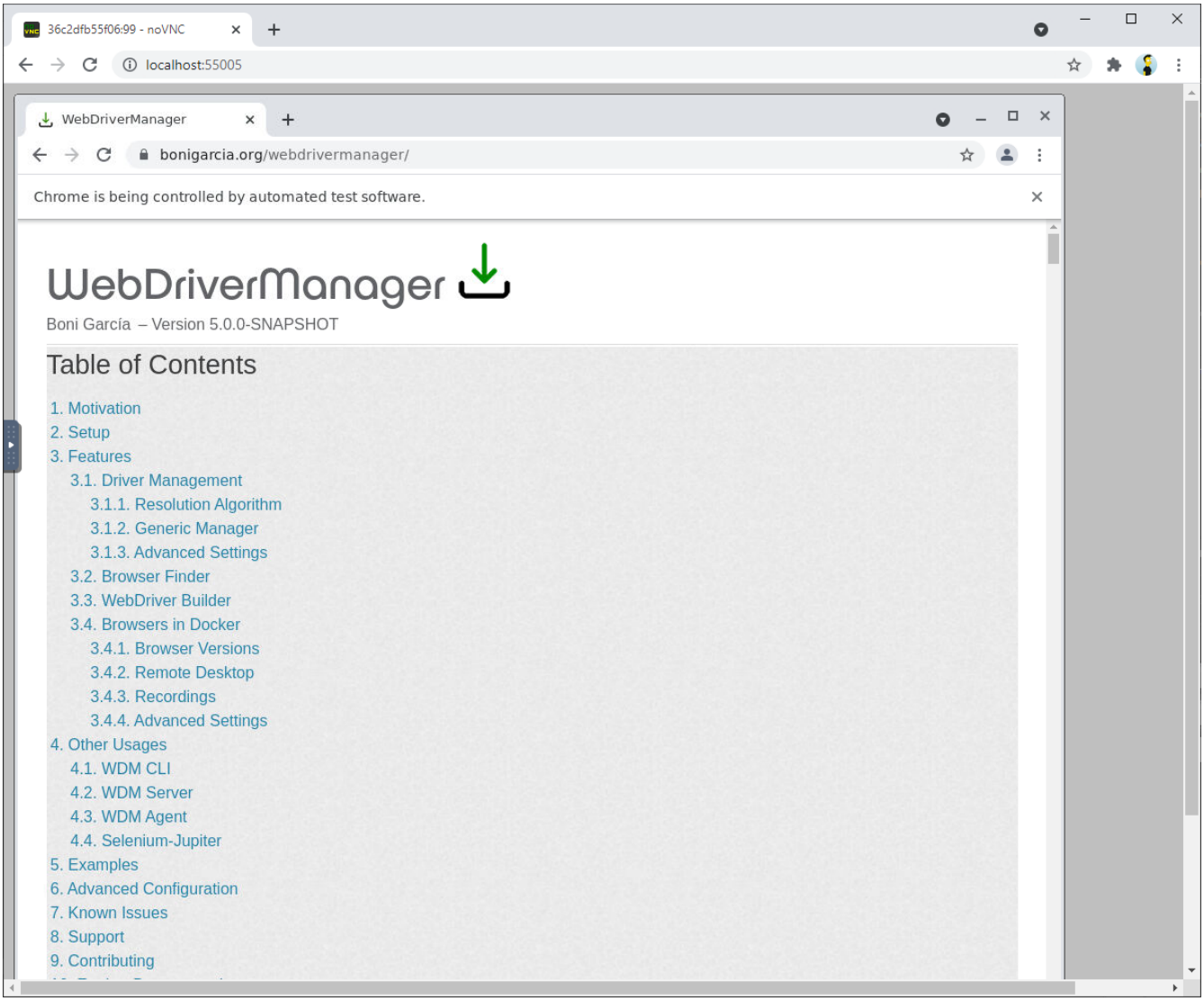

Figure 2. Example of noVNC session using Chrome in Docker

3.4.3. Recordings

The following related feature is the recording of the remote session of a dockerized browser. To enable it, we need to invoke the method `enableRecording()` in WebDriverManager. Internally, WebDriverManager starts another Docker container using FFmpeg to record the browser session. At the end of the test, we can find the recording in MP4 (by default, with a filename composed by the browser name followed by the symbol `and the system timestamp, plus another` and the session id) located in the project root folder (you can change this behavior using configuration capabilities. The following test shows an example of this feature.

```java
import static org.assertj.core.api.Assertions.assertThat;

import java.nio.file.Path;
import java.time.Duration;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

import io.github.bonigarcia.wdm.WebDriverManager;

class DockerChromeRecordingTest {

    WebDriver driver;

    WebDriverManager wdm = WebDriverManager.chromedriver().browserInDocker()
            .enableRecording();

    @BeforeEach
    void setupTest() {
        driver = wdm.create();
    }

    @AfterEach
    void teardown() {
        wdm.quit();
    }

    @Test
    void test() throws Exception {
        driver.get("https://bonigarcia.dev/selenium-webdriver-java/");
        assertThat(driver.getTitle()).contains("Selenium WebDriver");

        // Pause to see the navigation in the recording
        Thread.sleep(Duration.ofSeconds(2).toMillis());

        driver.findElement(By.partialLinkText("form")).click();

        // Pause to generate a longer recording
        Thread.sleep(Duration.ofSeconds(2).toMillis());

        // Verify recoding file
        Path recordingPath = wdm.getDockerRecordingPath();
        assertThat(recordingPath).exists();
    }

}
```

3.4.4. Advanced Settings

The dockerized browser provided by WebDriverManager can be configured in different ways. For example, the WebDriverManager API allows using volumes, customizing the language and timezone inside the browser container, using custom images, configuring remote Docker daemon, customizing shared memory and in-memory filesystem (*tmpfs*), changing the screen resolution and video frame rate, or customizing the recording output (folder and filename). See the advanced configuration section for specific details about it.

In addition, the WebDriverManager API provides several methods to get the most of the dockerized browsers, namely:

- `getDockerRecordingPath()`: Get path of the session recording.
- `getDockerNoVncUrl()`: Ger URL of the remote desktop noVNC session.
- `getDockerSeleniumServerUrl()`: Get URL of the underlying Selenium Server (inside the container) that allows controlling the remote (dockerized) browser.
- `getDockerService()`: It allows access to the Docker service and client (based on docker-java) to make custom operations with Docker containers (e.g., run commands in the browser container, see example here).
- `getDockerBrowserContainerId()`: Get browser container id (required for advance operation using the Docker client)
- `getWebDriver()`: Get the previously created `WebDriver` object (the same as the returned by the method `create()`.
- `getWebDriverList`: Get the previously created `WebDriver` objects (if any).

A manager instance can be used to create more than one `WebDriver` object. For this reason, the methods `getDockerRecordingPath()`, `getDockerNoVncUrl()`, `getDockerSeleniumServerUrl()`, `getDockerBrowserContainerId()`, and `quit()` are overloaded, allowing to specify an `WebDriver` instance. When no parameter is specified in these methods, WebDriverManager returns the first `WebDriver` object (this is a usual case, i.e., a manager creates a single instance of `WebDriver`). You can see an example of a manager used to create more than one browser here.

Chapter 4. Other Usages

In addition to as a regular Java dependency, WebDriverManager can be used in other ambits. This section summarizes these usages.

4.1. WebDriverManager CLI

WebDriverManager can be used interactively from the Command Line Interface (CLI), i.e., the shell. There are three different ways to use WebDriverManager as a CLI tool:

1. Using the WebDriverManager *fat-JAR* (i.e., WebDriverManager with all its dependencies in a single executable JAR file). This JAR file is generated from the source using the Maven command `mvn compile assembly:single`, and it is released on GitHub with every new version of WebDriverManager. You can download the latest of this fat-JAR from here. Once you get this file, you need to use the following command in the shell (where `<args>` are the accepted arguments, explained below):

```
java -jar webdrivermanager-5.1.0-fat.jar <args>
```

2. Using the source code. WebDriverManager is hosted on GitHub. We can use Maven to manage its Java source code. For example, to run the CLI mode using Maven and the source code, we need to invoke the following Maven command in the shell from the project root:

```
mvn exec:java -Dexec.args="<args>"
```

3. Using the WebDriverManager Docker container. Each new release of WebDriverManager is pushed to Docker Hub as a container based on OpenJDK plus the WebDriverManager fat-JAR. The default command to run the WebDriverManager Docker container is described below.

```
docker run --rm -e ARGS="<args>" bonigarcia/webdrivermanager:5.1.0
```

WebDriverManager CLI can be used for three different purposes. We can see these options launching the CLI with empty or invalid arguments (`<args>` in the commands before). In this case, the output of WebDriverManager CLI is the following:

```
[ERROR] The valid arguments for WebDriverManager CLI are:
[ERROR] 1. For resolving drivers locally:
[ERROR]         resolveDriverFor browserName <driverVersion>
[ERROR] (where browserName is: chrome|edge|firefox|opera|chromium|iexplorer)
[ERROR]
[ERROR] 2. For running a browser in a Docker (and use it trough noVNC):
[ERROR]         runInDocker browserName <browserVersion>
[ERROR] (where browserName is: chrome|edge|firefox|opera|safari|chrome-mobile)
[ERROR]
[ERROR] 3. For starting WebDriverManager Server:
[ERROR]         server <port>
[ERROR] (where the default port is 4444)
```

Option 1: Driver Resolver

WebDriverManager CLI can be used to resolve drivers (e.g., chromedriver, geckodriver) applying the usual resolution algorithm from the shell. This feature can be interesting if we want to download drivers outside a Java program. To use this option, we need to invoke WebDriverManager CLI using the following arguments (supposing we need to resolve chromedriver):

- Fat-JAR:

```
java -jar webdrivermanager-5.1.0-fat.jar resolveDriverFor chrome
```

- Source code:

```
mvn exec:java -Dexec.args="resolveDriverFor chrome"
```

- Docker container:

```
docker run --rm -v ${PWD}:/wdm -e ARGS="resolveDriverFor chrome"
bonigarcia/webdrivermanager:5.1.0
```

Option 2: Browsers in Docker

WebDriverManager CLI can be used to execute browsers in Docker containers and interact with them using noVNC. This feature can be interesting for *exploratory testing* for web applications using different types and versions of web browsers. To use this option, the arguments we need to use are the following (supposing we want to use Chrome):

- Fat-JAR:

```
java -jar webdrivermanager-5.1.0-fat.jar runInDocker chrome
```

- Source code:

```
mvn exec:java -Dexec.args="runInDocker chrome"
```

- Docker container:

```
docker run --rm -it -v /var/run/docker.sock:/var/run/docker.sock -e ARGS="runInDocker
chrome 91" bonigarcia/webdrivermanager:5.1.0
```

There is an open issue with Chrome 92+ and Docker at the time of this writing. For this reason, the previous Docker container command uses Chrome 91. For further information, see known issues.

Option 3: Server

Finally, WebDriverManager CLI allows starting the WebDriverManager Server, as follows (see next section for more details about it):

- Fat-JAR:

```
java -jar webdrivermanager-5.1.0-fat.jar server
```

- Source code:

```
mvn exec:java -Dexec.args="server"
```

- Docker container:

```
docker run --rm -p 4444:4444 -v /var/run/docker.sock:/var/run/docker.sock
bonigarcia/webdrivermanager:5.1.0
```

4.2. WebDriverManager Server

The WebDriverManager Server is based on HTTP and offers two types of services. First, it can be used to resolve drivers (chromedriver, geckodriver, etc.). The WebDriverManager Server exposes a simple REST-like API to this aim. WebDriverManager Server sends the resolved driver as an HTTP attachment in the response. The endpoints provided by this API are the following (supposing that WebDriverManager is running the localhost in its default port, i.e., 4444):

- http://localhost:4444/chromedriver: To resolve chromedriver.
- http://localhost:4444/firefoxdriver: To resolve geckodriver.
- http://localhost:4444/edgedriver: To resolve msedgedriver.

- http://localhost:4444/operadriver: To resolve geckodriver.
- http://localhost:4444/iedriver: To resolve geckodriver.

Second, the WebDriverManager Server acts as a regular Selenium Server (i.e., a *hub* in the classical Selenium Grid architecture). This feature can be used to create remote `WebDriver` instances using the WebDriverManager Server (even for different language bindings than Java). The following example shows a Node.js test using Selenium WebDriver and WebDriverManager Server (notice that by default, the WebDriverManager Server URL does not require any path, i.e., http://localhost:4444/):

```
var webdriver = require("selenium-webdriver");

async function wdmServerTest() {
    var wdmServerUrl = "http://localhost:4444/";
    var capabilities = {
        browserName : "chrome",
        version: "91.0"
    };

    try {
        var driver = await new webdriver.Builder().usingServer(wdmServerUrl)
                .withCapabilities(capabilities).build();

        var sutUrl = "https://bonigarcia.dev/selenium-webdriver-java/";
        await driver.get(sutUrl);

        await driver.getTitle().then(function(title) {
            console.log("The title of " + sutUrl + " is '" + title + "'")
        });

    } catch (err) {
        console.error("Something went wrong!\n", err.stack);

    } finally {
        if (driver) {
            driver.quit();
        }
    }
}

wdmServerTest();
```

4.3. WebDriverManager Agent

WebDriverManager can also be used as *Java Agent*. In this case, and using the JVM instrumentation API, WebDriverManager intercepts calls to applications running on the JVM and modifies their bytecode. In particular, the WebDriverManager Agent uses this technique to check the objects being created in the JVM. Before Selenium WebDriver objects are instantiated (`ChromeDriver`,

`FirefoxDriver`, etc.), the required manager is used to resolve its driver (chromedriver, geckodriver, etc.). Thanks to this approach, we can get rid of the WebDriverManager call (e.g. `WebDriverManager.chromedriver.setup();`) from our test, for example:

```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

class ChromeAgentTest {

    WebDriver driver;

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

    @AfterEach
    void teardown() {
        if (driver != null) {
            driver.quit();
        }
    }

    @Test
    void test() {
        // Test logic
    }

}
```

To configure the WebDriverManager Agent, we need to specify the path of the WebDriverManager fat-JAR using the JVM flag `-javaagent:/path/to/webdrivermanager-5.1.0-fat.jar`. Alternatively, it can be done using Maven (see a complete project example here).

4.4. Selenium-Jupiter

WebDriverManager is the heart of the project Selenium-Jupiter, an open-source JUnit 5 extension for Selenium WebDriver. Selenium-Jupiter uses the programming and extension model provided by JUnit 5 (named Jupiter) together with WebDriverManager to create tests with reduced boilerplate code. For instance, thanks to the Jupiter feature for parameter resolution, we can declare a type of the `WebDriver` hierarchy (e.g., `ChromeDriver`) as a test parameter. Internally, Selenium-Jupiter resolves its driver and creates the instance before tests, and then, the browser is gracefully closed at the end of the test. For example:

```
import static org.assertj.core.api.Assertions.assertThat;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.chrome.ChromeDriver;

import io.github.bonigarcia.seljup.SeleniumJupiter;

@ExtendWith(SeleniumJupiter.class)
class ChromeTest {

    @Test
    void test(ChromeDriver driver) {
        driver.get("https://bonigarcia.dev/selenium-webdriver-java/");
        assertThat(driver.getTitle()).contains("Selenium WebDriver");
    }

}
```

Selenium-Jupiter provides more different features, such as seamless integration with Docker, test templates (for cross-browser testing), conditional test execution (depending on the availability of browsers), conditional screenshots and recordings (when tests fail), and more. For further details, read the Selenium-Jupiter documentation.

4.5. Selenium Grid

Selenium Grid is an infrastructure that allows serving remote browsers to be used in Selenium WebDriver tests. The nodes in which the browsers are executed in Selenium Grid also require managing the required drivers (chromedriver, geckodriver, etc.). To manage these drivers, as usual, we can use WebDriverManager.

For example, this test shows how to start a Selenium Grid in standalone mode, resolving the driver with WebDriverManager and registering a Chrome browser in the Selenium Server. In addition, if we start Selenium Grid from the shell, we can complement it with WebDriverManager CLI to resolve the required drivers, as follows:

```
boni@ubuntu:~$ java -jar webdrivermanager-5.1.0-fat.jar resolveDriverFor chrome
[INFO] Using WebDriverManager to resolve chrome
[DEBUG] Detecting chrome version using online commands.properties
[DEBUG] Running command on the shell: [google-chrome, --version]
[DEBUG] Result: Google Chrome 92.0.4515.107
[DEBUG] Latest version of chromedriver according to
https://chromedriver.storage.googleapis.com/LATEST_RELEASE_92 is 92.0.4515.107
[INFO] Using chromedriver 92.0.4515.107 (resolved driver for Chrome 92)
[INFO] Reading https://chromedriver.storage.googleapis.com/ to seek chromedriver
[DEBUG] Driver to be downloaded chromedriver 92.0.4515.107
[INFO] Downloading
https://chromedriver.storage.googleapis.com/92.0.4515.107/chromedriver_linux64.zip
[INFO] Extracting driver from compressed file chromedriver_linux64.zip
[INFO] Driver location: /home/boni/chromedriver

boni@ubuntu:~$ java -jar selenium-server-4.0.0-beta-4.jar standalone
20:19:03.815 INFO [LoggingOptions.configureLogEncoding] - Using the system default
encoding
20:19:03.818 INFO [OpenTelemetryTracer.createTracer] - Using OpenTelemetry for tracing
20:19:04.272 INFO [NodeOptions.getSessionFactories] - Detected 8 available processors
20:19:04.284 INFO [NodeOptions.discoverDrivers] - Discovered 1 driver(s)
20:19:04.296 INFO [NodeOptions.report] - Adding Chrome for {"browserName":
"chrome","platformName": "LINUX"} 8 times
20:19:04.310 INFO [Node.<init>] - Binding additional locator mechanisms: name, id
20:19:04.322 INFO [LocalDistributor.add] - Added node 4651dc4d-3e07-43e7-b8d5-
7368c95ea76d at http://172.17.0.1:4444.
20:19:04.324 INFO [GridModel.setAvailability] - Switching node 4651dc4d-3e07-43e7-
b8d5-7368c95ea76d (uri: http://172.17.0.1:4444) from DOWN to UP
20:19:04.438 INFO [Standalone.execute] - Started Selenium Standalone 4.0.0-beta-4
(revision 29f46d02dd): http://172.17.0.1:4444
```

4.6. Appium

Appium is a test automation framework for mobile applications. In the same way that Selenium WebDriver, Appium also speaks the W3C WebDriver protocol to drive web browsers on mobile devices. For this reason, the required driver must be present to control mobile browsers with Appium.

Again, WebDriverManager can help in this task. The following snippet shows a Java method that creates a `WebDriver` object using the driver path for Chrome browser in an Android device:

```
    public WebDriver createChromeAndroidDriver(String browserVersion,
            String deviceName, URL appiumServerUrl) {
        // Resolve driver and get its path
        WebDriverManager wdm = WebDriverManager.chromedriver()
                .browserVersion(browserVersion);
        wdm.setup();
        String chromedriverPath = wdm.getDownloadedDriverPath();

        // Create WebDriver instance using the driver path
        DesiredCapabilities capabilities = new DesiredCapabilities();
        capabilities.setCapability("browserName", "chrome");
        capabilities.setCapability("version", browserVersion);
        capabilities.setCapability("deviceName", deviceName);
        capabilities.setCapability("platformName", "android");
        capabilities.setCapability("chromedriverExecutable", chromedriverPath);

        return new AndroidDriver<WebElement>(appiumServerUrl, capabilities);
    }
```

Chapter 5. Examples

All the examples presented in this documentation are available in the WebDriverManager tests. Moreover, different public repositories contain test examples using WebDriverManager, such as:

- WebDriverManager Basic: A simple project using JUnit 5, Selenium WebDriver, and WebDriverManager.
- WebDriverManager Examples: Different examples with JUnit 5, Selenium WebDriver, and WebDriverManager.
- WebDriverManager Spring-Boot: Another simple project using Spring-Boot, JUnit 5, Selenium WebDriver, and WebDriverManager.
- WebDriverManager Agent Example: Maven project using WebDriverManager as Agent.
- Selenium WebDriver with Java: A comprehensive collection of Selenium WebDriver 4 examples using Java as language binding.

Chapter 6. Advanced Configuration

WebDriverManager 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):

```
WebDriverManager.chromedriver().driverVersion("81.0.4044.138").setup();
WebDriverManager.firefoxdriver().browserVersion("75").setup();
WebDriverManager.operadriver().proxy("server:port").setup();
WebDriverManager.edgedriver().mac().setup();
```

In addition to the methods presented in this section, each manager provides a `config()` method that allows configuring all the possible parameters of WebDriverManager. In addition, WebDriverManager provides the metho `reset()` to restore to the default values the parameters of a given manager.

The second alternative to tune WebDriverManager is using *Java system properties*. In this method, each WebDriverManager API method has its equivalence of a unique *configuration key*. For instance, the API method `cachePath()` (used to specify the driver cache folder) is equivalent to the configuration key `wdm.cachePath`. These types of configuration keys can be passed when executing the tests, for example, using Maven:

```
mvn test -Dwdm.cachePath=/custom/path/to/driver/cache
```

The third way to configure WebDriverManager is using *environmental variables*. The names for these variables are made from converting each configuration key name (e.g., `wdm.cachePath`) to uppercase and replacing the symbol `.` with `_` (e.g., `WDM_CACHEPATH`). These variables can be helpful to global setup parameters at the operating system level. Also, it allows specifying a custom setup when using WebDriverManager as a Docker container. For example:

```
docker run --rm -v ${PWD}:/wdm -e ARGS="resolveDriverFor chrome" -e
WDM_CHROMEVERSION=84 bonigarcia/webdrivermanager:5.1.0
```

The preference order of these configuration alternatives is (in case of overlapping) is: 1) Environmental Variables. 2) Java system properties. 3) Java API.

The remainder of this section describes all the possible Java methods in the WebDriverManager API and its equivalent configuration keys in three groups of capabilities: driver management, browsers in Docker, and WebDriverManager Server.

Table 1. Configuration capabilities for driver management

API method	Configuration key	Default value	Description
`cachePath(String)`	`wdm.cachePath`	`~/.cache/selenium`	Folder to store drivers locally
`resolutionCachePath(String)`	`wdm.resolutionCachePath`	`~/.cache/selenium`	Folder to store the resolution cache
`driverVersion(String)`	`wdm.chromeDriverVersion`, `wdm.operaDriverVersion`, `wdm.iExplorerDriverVersion`, `wdm.edgeDriverVersion`, `wdm.geckoDriverVersion`, `wdm.chromiumDriverVersion`	`""` (automatic driver version discovery through the resolution algorithm)	Custom driver version
`browserVersion(String)`	`wdm.chromeVersion`, `wdm.operaVersion`, `wdm.edgeVersion`, `wdm.firefoxVersion`, `wdm.chromiumVersion`	`""` (automatic browser version detection using the commands database)	Custom browser version (major)
`forceDownload()`	`wdm.forceDownload=true`	`false` (drivers in cache are reused if available)	Force downloading driver (even if it is already in the cache)
`useBetaVersions()`	`wdm.useBetaVersions=true`	`false` (driver versions are skipped)	Allow the use beta versions (if possible)
`architecture(Architecture)`	`wdm.architecture`	`""` (automatic architecture discovery)	Force a given architecture for a driver
`arch32()`	`wdm.architecture=X32`	`""` (automatic architecture discovery)	Use 32-bit driver version
`arch64()`	`wdm.architecture=X64`	`""` (automatic architecture discovery)	Use 64-bit driver version
`arm64()`	`wdm.architecture=ARM64`	`""` (automatic architecture discovery)	Use ARM (Aarch) 64-bit driver version
`operatingSystem(Operating System)`	`wdm.os`	`""` (automatic OS discovery)	Force a given operating system (`WIN`, `LINUX`, or `MAC`
`win()`	`wdm.os=WIN`	`""` (automatic OS discovery)	Force Windows
`linux()`	`wdm.os=LINUX`	`""` (automatic OS discovery)	Force Linux
`mac()`	`wdm.os=MAC`	`""` (automatic OS discovery)	Force Mac OS

API method	Configuration key	Default value	Description
`driverRepositoryUrl(URL)`	`wdm.chromeDriverUrl`, `wdm.operaDriverUrl`, `wdm.edgeDriverUrl`, `wdm.geckoDriverUrl`, `wdm.iExplorerDriverUrl`	Driver specific URLs (available in webdrivermanager.properties)	Change the repository URL in which the drivers are hosted
`useMirror()`	`wdm.useMirror=true`	`false` (not using mirrors)	Enable the use of a driver repository mirror (available for chromedriver, geckodriver, and operadriver)
`proxy(String)`	`wdm.proxy`	`""` (no proxy)	Use an HTTP proxy for the network connection (using the notation `proxy:port` or `username:password@proxy:port`). It can be also configured using the environment variable `HTTPS_PROXY`
`proxyUser(String)`	`wdm.proxyUser`	`""` (no proxy user)	Username for the HTTP proxy. It can be also configured using the environment variable `HTTPS_PROXY_USER`
`proxyPass(String)`	`wdm.proxyPass`	`""` (no proxy password)	Password for HTTP proxy. It can be also configured using the environment variable `HTTPS_PROXY_PASS`
`gitHubToken(String)`	`wdm.gitHubToken`	`""` (no GitHub token)	Personal access token for authenticated GitHub requests (see known issues). It can be also configured using the environment variable `GITHUB_TOKEN`
`ignoreVersions(String ...)`	`wdm.ignoreVersions`	`""` (no ignored versions)	Ignore specific driver version(s)
`timeout(int)`	`wdm.timeout`	`30`	Timeout (in seconds) to connect and download drivers from online repositories

API method	Configuration key	Default value	Description
`properties(String)`	`wdm.properties`	`webdrivermanager.properties`	Properties file (in the project classpath) for default configuration values
`avoidExport()`	`wdm.avoidExport=true`	`false` (export driver paths as Java properties (e.g. `webdriver.chrome.driver`)	Avoid step 4 in the resolution algorithm (for instance, in the CLI mode)
`exportParameter(String)`	`wdm.chromeDriverExport`, `wdm.geckoDriverExport`, `wdm.edgeDriverExport`, `wdm.iExplorerDriverExport`, `wdm.operaDriverExport`	Java property name used to export the driver path (available in webdrivermanager.properties)	Set custom property name
`avoidOutputTree()`	`wdm.avoidOutputTree=true`	`false` (create output tree in the driver cache, e.g., `chromedriver/linux64/2.37`	Avoid output tree (for instance, in the CLI mode)
`avoidFallback()`	`wdm.avoidFallback=true`	`false` (use a retries mechanism if any problem happens during the resolution algorithm)	Avoid the fallback mechanism
`avoidBrowserDetection()`	`wdm.avoidBrowserDetection =true`	`false` (browser version is detected, and the corresponding driver version s discovered)	Force to use the latest version available for a given driver
`avoidReadReleaseFromRepository()`	`wdm.avoidReadReleaseFrom Repository=true`	`false` (discover driver relase using info from the repository, e.g., chromedriver-latest or msedgedriver-latest)	Avoid using the repository info and use the versions database instead
`avoidTmpFolder()`	`wdm.avoidTmpFolder=true`	`false` (Each driver release (typically compressed) is copied in a temporal folder in the local machine, and then the driver is extracted and copied to the driver cache	Avoid using a temporal folder to download drivers (and handle driver release directly on driver cache)

API method	Configuration key	Default value	Description
`browserVersionDetection Command(String)`	`wdm.browserVersion DetectionCommand`	`""` (automatic discovery using the commands database)	Custom browser version detection command (see example here)
`browserVersionDetection Regex(String)`	`wdm.browserVersion DetectionRegex`	`[^\\d\^\\.]`	Regular expression used to extract the browser version from the shell
`useLocalVersionsProper ties First()`	`wdm.versionsProperties OnlineFirst=true`	`false` (the online versions database is used in the resolution algorithm)	Use local copy of the versions database
`useLocalCommandsProper ties First()`	`wdm.commandsProperties OnlineFirst=true`	`false` (the online commands database is used in the resolution algorithm)	Use local copy of the commands database
`versionsPropertiesUrl( URL)`	`wdm.versionsProperties Url`	Raw version of the online versions database	Change versions database URL
`commandsPropertiesUrl( URL)`	`wdm.commandsProperties Url`	Raw version of the online commands database	Change commands database URL
`clearDriverCache()`	`wdm.clearDriverCache=t rue`	`false` (not cleaning driver cache)	Clean driver cache
`clearResolutionCache()`	`wdm.clearResolutionCac he =true`	`false` (not cleaning resolution cache)	Clean resolution cache
`ttl(int)`	`wdm.ttl`	`86400` (i.e., 1 day)	TTL in seconds in which the resolved driver versions are valid in the resolution cache.
`ttlBrowsers(int)`	`wdm.ttlForBrowsers`	`3600` (i.e., 1 hour)	TTL value in seconds in which the browser versions are valid in the resolution cache (also used for dockerized browsers).

Table 2. Configuration capabilities for browsers in Docker

API method	Configuration key	Default value	Description
`dockerDaemonUrl(String )`	`wdm.dockerDaemonUrl`	`""`	URL of remote Docker daemon

API method	Configuration key	Default value	Description
`dockerTimezone(String)`	`wdm.dockerTimezone`	`Etc/UTC`	Timezone of the browser container
`dockerNetwork(String)`	`wdm.dockerNetwork`	`bridge`	Docker network name
`dockerLang(String)`	`wdm.dockerLang`	`EN`	Language of the browser container
`dockerShmSize(String)`	`wdm.dockerShmSize`	`256m`	Docker shared memory in bytes. Unit is optional and can be `b` (bytes), `k` (kilobytes), `m` (megabytes), or `g` (gigabytes)
`dockerTmpfsSize(String)`	`wdm.dockerTmpfsSize`	`128m`	Docker in-memory filesystem (tmpfs). Units follows the same approach than the shared memory
`dockerTmpfsMount(String)`	`wdm.dockerTmpfsMount`	`/tmp`	Mount point for in-memory filesystem
`dockerStopTimeoutSec(Integer)`	`wdm.dockerStopTimeoutSec`	`5`	Max time to kill a container (in seconds) after stopping it
`enableVnc()`	`wdm.dockerEnableVnc=true`	`false`	Enable desktop remote session for browsers in Docker
`viewOnly()`	`wdm.dockerViewOnly=true`	`false`	Run remote desktop session (noVNC) in view-only mode
`enableRecording()`	`wdm.dockerEnableRecording=true`	`false`	Enable the recordings of the browser session in Docker
`dockerScreenResolution(String)`	`wdm.dockerScreenResolution`	`1280x1080x24`	Screen resolution of the browser desktop session in format `<width>x<height>x<colors-depth>`
`dockerRecordingFrameRate(int)`	`wdm.dockerRecordingFrameRate`	`12`	Frame rate for recordings
`dockerRecordingOutput(String)` `dockerRecordingOutput(Path)`	`wdm.dockerRecordingOutput`	`.`	Path for the recording output. This value can be a folder or complete path (if it ends with `.mp4`)

API method	Configuration key	Default value	Description
`dockerRecordingPrefix(String)`	`wdm.dockerRecordingPrefix`	Browser name	Prefix to be appended to default filname (i.e., broser name plus _ plus session id)
`dockerCustomImage(String)`	`wdm.dockerCustomImage`	`""`	Custom image to be used as browser in Docker
`dockerVolumes(String[])`	`wdm.dockerVolumes`	`""`	Docker volumes (single or array) using the format `"\local\path:\container\path"`
`dockerExtraHosts(String[])`	`wdm.dockerExtraHosts`	`""`	Docker Extra Hosts (single or array) using the format `"hostname:IP"` `("host1:192.168.48.82, host2:192.168.48.16")`
`dockerPrivateEndpoint(String)`	`wdm.dockerPrivateEndpoint`	`""`	Used to prefix pull images when you have a private registry with authentication i.e docker-hub-remote.myprivate.com will be prefixed to pull as docker-hub-remote.myprivate.com/selenoid/vnc and so on for any images used (video recorder, novnc etc..), docker login docker-hub-remote.myprivate.com is still required in order to get the auth credentials stored in the .docker/config.json, for MacOS users make sure to configure your engine to store the credentials in the config.json instead of keychain storage.

Table 3. Configuration capabilities for WebDriverManager Server

API method	Configuration key	Default value	Description
`serverPort(int)`	`wdm.serverPort`	`4444`	Port of WebDriverManager Server
`serverPath(String)`	`wdm.serverPath`	`/`	Path of WebDriverManager Server
`serverTimeoutSec(int)`	`wdm.serverTimeoutSec`	`60`	Timeout (in seconds) for WebDriverManager server

Chapter 7. Known Issues

HTTP response code 403

Some of the drivers (e.g., geckodriver or operadriver) are hosted on GitHub. When external clients (like WebDriverManager) makes many consecutive requests to GitHub, and due to its traffic rate limit, it eventually responds with an HTTP 403 error (*forbidden*), as follows:

```
io.github.bonigarcia.wdm.config.WebDriverManagerException: Error HTTP 403 executing
https://api.github.com/repos/mozilla/geckodriver/releases
    at io.github.bonigarcia.wdm.online.HttpClient.execute(HttpClient.java:172)
    at
io.github.bonigarcia.wdm.WebDriverManager.openGitHubConnection(WebDriverManager.java:1
266)
    at
io.github.bonigarcia.wdm.WebDriverManager.getDriversFromGitHub(WebDriverManager.java:1
280)
    at
io.github.bonigarcia.wdm.managers.FirefoxDriverManager.getDriverUrls(FirefoxDriverMana
ger.java:95)
    at
io.github.bonigarcia.wdm.WebDriverManager.createUrlHandler(WebDriverManager.java:1111)
    at io.github.bonigarcia.wdm.WebDriverManager.download(WebDriverManager.java:959)
    at io.github.bonigarcia.wdm.WebDriverManager.manage(WebDriverManager.java:877)
    at io.github.bonigarcia.wdm.WebDriverManager.fallback(WebDriverManager.java:1106)
    at
io.github.bonigarcia.wdm.WebDriverManager.handleException(WebDriverManager.java:1083)
    at io.github.bonigarcia.wdm.WebDriverManager.manage(WebDriverManager.java:883)
    at io.github.bonigarcia.wdm.WebDriverManager.setup(WebDriverManager.java:328)
```

To avoid this problem, WebDriverManager can make authenticated requests using a personal access token. See the advanced configuration section to discover how to set up this token in WebDriverManager.

Testing localhost

A typical case in web development is testing a web application deployed in the local host. In this case, and when using browsers in Docker containers, we need to know that the address `localhost` inside a Docker container is not the host's address but the container address. To solve this problem, we can take different approaches. In Linux, we can use the gateway address for inter-container communication. This address is usually `172.17.0.1`, and can be discovered as follows:

```
$ ip addr show docker0
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN
group default
    link/ether 02:42:b4:83:10:c8 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
       valid_lft forever preferred_lft forever
```

When the host is Mac OS or Windows, we can use the DNS name `host.docker.internal`, which will be resolved to the internal IP address used by the host.

Chrome 92+ in Docker

There is an open issue with Chrome 92+ and Docker at the time of this writing. The problem can be solved by using the argument `--disable-gpu` in Chrome and Edge. This argument is used by WebDriverManager 5 to avoid the issue. Nevertheless, some situations are still impossible to fix (e.g., when using WebDriverManager in Docker as CLI or Server).

Chapter 8. Community

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

Chapter 9. Support

WebDriverManager 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).

Chapter 10. Further Documentation

There are other resources related to Selenium-Jupiter and automated testing you can find helpful. For instance, the following books:

- García, Boni. Hands-On Selenium WebDriver with Java. *O'Reilly Media, Inc.*, Currently on early relase. Final release available on 2022.
- García, Boni. Mastering Software Testing with JUnit 5. *Packt Publishing Ltd*, 2017.

Or the following journal papers:

- García, Boni, et al. "Automated driver management for Selenium WebDriver." *Empirical Software Engineering* 26.5 (2021): 1-51.
- García, Boni, et al. "A survey of the Selenium ecosystem." *Electronics* 9.7 (2020): 1067.
- García, Boni, et al. "Assessment of QoE for video and audio in WebRTC applications using full-reference models." *Electronics* 9.3 (2020): 462.
- García, Boni, et al. "Practical evaluation of VMAF perceptual video quality for WebRTC applications." *Electronics* 8.8 (2019): 854.
- García, Boni, et al. "WebRTC testing: challenges and practical solutions." *IEEE Communications Standards Magazine* 1.2 (2017): 36-42.
- García, Boni, and Juan Carlos Dueñas. "Web browsing automation for applications quality control." *Journal of web engineering* (2015): 474-502.

Chapter 11. About