

# WebDriverManager

Boni García

Version 5.0.0-SNAPSHOT

# Table of Contents

1. Motivation .....	1
2. Setup .....	3
3. Features .....	4
3.1. Driver Management .....	4
3.2. Browser Finder .....	9
3.3. WebDriver Builder .....	11
3.4. Browsers in Docker .....	12
4. Other Usages .....	20
4.1. WDM CLI .....	20
4.2. WDM Server .....	22
4.3. WDM Agent .....	23
4.4. Selenium-Jupiter .....	24
5. Examples .....	26
6. Advanced Configuration .....	27
7. Known Issues .....	34
8. Support .....	36
9. Contributing .....	37
10. Further Documentation .....	38
11. About .....	39

# 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 carry out 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).

[WebDriverManager](#) is an open-source Java library that carries out this [driver management process in a fully automated manner](#). In addition, as of version 5, WebDriverManager provides other relevant features, such as the ability to [discover if a browser is installed in the system](#), [building WebDriver objects](#) (such as [ChromeDriver](#), [FirefoxDriver](#), etc.), and the capability to run [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>5.0.0</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:5.0.0")
}
```

# 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 that allow creating *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 in this case) 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() {
        if (driver != null) {
            driver.quit();
        }
    }

    @Test
    void test() {
        // Exercise
        driver.get("https://bonigarcia.org/webdrivermanager/");
        String title = driver.getTitle();

        // Verify
        assertThat(title).contains("WebDriverManager");
    }
}

```

WebDriverManager provides a set *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](#)).

### 3.1.1. Resolution Algorithm

WebDriverManager executes a *resolution algorithm* when calling to `setup()` in a given manager. You can find all the internal details of this algorithm in the paper [Automated driver management for Selenium WebDriver](#), published in the Springer Journal of Empirical Software Engineering. 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 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 local resources (i.e., without using the shell nor requesting external



servers).

### 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` (e.g., "Chrome", "Firefox", "Edge", etc.).
- `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 instantiates the proper manager using this parameter.

```

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() {
        if (driver != null) {
            driver.quit();
        }
    }

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

        // Driver instantiation
        driver = webDriverClass.getDeclaredConstructor().newInstance();

        // Exercise
        driver.get("https://bonigarcia.org/webdrivermanager/");
        String title = driver.getTitle();

        // Verify
        assertThat(title).contains("WebDriverManager");
    }
}

```

### 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 out 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 needs 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) if it is 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() {
        if (driver != null) {
            driver.quit();
        }
    }

    @Test
    void test() {
        // Exercise
        driver.get("https://bonigarcia.org/webdrivermanager/");
        String title = driver.getTitle();

        // Verify
        assertThat(title).contains("WebDriverManager");
    }
}

```



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. To complement this feature, WebDriverManager provides the method **quit()** to close the created **WebDriver** instances gracefully.

```
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;

    WebDriverManager wdm = WebDriverManager.chromedriver();

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

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

    @Test
    void test() {
        // Exercise
        driver.get("https://bonigarcia.org/webdrivermanager/");
        String title = driver.getTitle();

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

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

- 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). See [example](#). This method is equivalent to the configuration key `wdm.remoteAddress` (see [configuration](#) section).
- Integer parameter in the method `create()`. This option is used to create a list of `WebDriver` objects instead of a single instance. See [example](#).

## 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.org/webdrivermanager/");
        assertThat(driver.getTitle()).contains("WebDriverManager");
    }

}

```



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](#). Therefore, the available browsers to be executed as Docker containers in `WebDriverManager` are: Chrome (like the previous example), Firefox (see [example](#)), Edge (see [example](#)), Opera (see [example](#)), and Safari (see [example](#)).

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 that a hardware server or virtual machine with

nested virtualization support is required 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 further effort.

Nevertheless, we can force to use 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 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 DockerChromeBetaTest {

    WebDriver driver;

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

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

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

    @Test
    void test() {
        driver.get("https://bonigarcia.org/webdrivermanager/");
        assertThat(driver.getTitle()).contains("WebDriverManager");
    }
}

```

### 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 of a browser 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 java.net.HttpURLConnection.HTTP_OK;
import static org.assertj.core.api.Assertions.assertThat;

import java.net.HttpURLConnection;
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.org/webdrivermanager/");
        assertThat(driver.getTitle()).contains("WebDriverManager");

        // Verify URL for remote session
        URL dockerSessionUrl = wdm.getDockerNoVncUrl();
        HttpURLConnection huc = (HttpURLConnection) dockerSessionUrl
            .openConnection();
        assertThat(huc.getResponseCode()).isEqualTo(HTTP_OK);

        // Active wait for manual inspection
        Thread.sleep(Duration.ofSeconds(60).toMillis());
    }
}
```

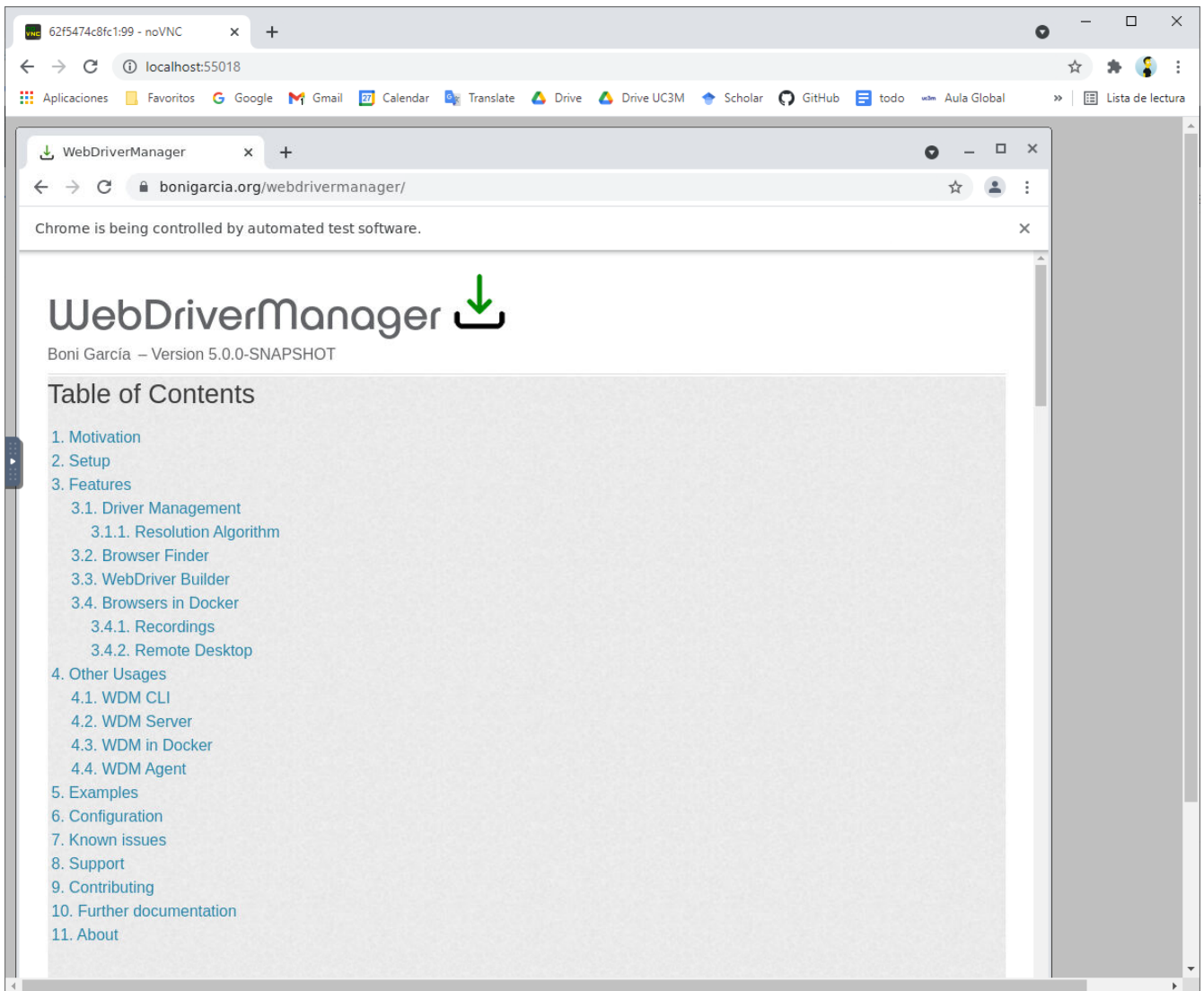


Figure 2. Example of noVNC session using Chrome in Docker

### 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, in the same folder of the project root. The following example shows a simple using this feature.

```

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.org/webdrivermanager/");
        assertThat(driver.getTitle()).contains("WebDriverManager");

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

        driver.findElement(By.partialLinkText("Browsers in Docker")).click();

        // Active wait 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, customizing the recording output (folder and filename), or accessing to Docker service and client to make custom operations with Docker containers (e.g., run commands in the browser container). See the [advanced configuration](#) section for specific details about it.

# 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. WDM 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 released on [GitHub](#) with every new version of WebDriverManager. You can download the latest version 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-5.0.0-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:5.0.0
```

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=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-5.0.0-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:5.0.0
```

## 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-5.0.0-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:5.0.0
```



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-5.0.0-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:5.0.0
```

## 4.2. WDM 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 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.org/webdrivermanager/";
    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. WDM 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-5.0.0-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 (called [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 ChromeJupiterTest {

    @Test
    void test(ChromeDriver driver) {
        driver.get("https://bonigarcia.org/selenium-jupiter/");
        assertThat(driver.getTitle()).contains("Selenium-Jupiter");
    }
}

```

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](#).

# Chapter 5. Examples

The following repositories contain different examples using WebDriverManager:

- [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-Jupiter Examples](#): Examples using JUnit 5 and [Selenium-Jupiter](#).

# 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 the following tables, each manager provides a method called `config()` that allows configuring all the possible parameters of WebDriverManager.

The second alternative to tune WebDriverManager is using *Java system properties*. In this method, each WebDriverManager API method has its equivalence of one 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 are set as Java system properties and can be passed when executing the test, 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 appear from converting each configuration key name (e.g., `wdm.cachePath`) to uppercase, replacing the symbol `.` by `_` (e.g., `WDM_CACHEPATH`). This alternative can be helpful to global setup parameters at the operating system level and for specifying parameters 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:5.0.0
```



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 <a href="#">resolution algorithm</a> )	Custom driver version
browserVersion(String)	wdm.chromeVersion, wdm.operaVersion, wdm.edgeVersion, wdm.firefoxVersion, wdm.chromiumVersion	"" (automatic browser version detection using the <a href="#">commands database</a> )	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(OperatingSystem)	wdm.os	"" (automatic OS discovery)	Force a given operating system ( <a href="#">WIN</a> , <a href="#">LINUX</a> , or <a href="#">MAC</a> )
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
<code>driverRepositoryUrl(URL)</code>	<code>wdm.chromeDriverUrl</code> , <code>wdm.operaDriverUrl</code> , <code>wdm.edgeDriverUrl</code> , <code>wdm.geckoDriverUrl</code> , <code>wdm.iExplorerDriverUrl</code>	Driver specific URLs (available in <a href="#">webdrivermanager.properties</a> )	Change the repository URL in which the drivers are hosted
<code>useMirror()</code>	<code>wdm.useMirror=true</code>	<code>false</code> (not using mirrors)	Enable the use of an existing <a href="#">mirror</a> (available for chromedriver, geckodriver, and operadriver)
<code>proxy(String)</code>	<code>wdm.proxy</code>	<code>""</code> (no proxy)	Use an HTTP proxy for the network connection (using the notation <code>proxy:port</code> or <code>username:password@proxy:port</code> ). It can be also configured using the environment variable <code>HTTPS_PROXY</code>
<code>proxyUser(String)</code>	<code>wdm.proxyUser</code>	<code>""</code> (no proxy user)	Username for the HTTP proxy. It can be also configured using the environment variable <code>HTTPS_PROXY_USER</code>
<code>proxyPass(String)</code>	<code>wdm.proxyPass</code>	<code>""</code> (no proxy password)	Password for HTTP proxy. It can be also configured using the environment variable <code>HTTPS_PROXY_PASS</code>
<code>gitHubToken(String)</code>	<code>wdm.gitHubToken</code>	<code>""</code> (no GitHub token)	Personal access token for authenticated GitHub requests (see <a href="#">known issues</a> ). It can be also configured using the environment variable <code>GITHUB_TOKEN</code>
<code>ignoreVersions(String...)</code>	<code>wdm.ignoreVersions</code>	<code>""</code> (no ignored versions)	Ignore specific driver version(s)
<code>timeout(int)</code>	<code>wdm.timeout</code>	<code>30</code>	Timeout (in seconds) to connect and download drivers from online repositories

API method	Configuration key	Default value	Description
<code>properties(String)</code>	<code>wdm.properties</code>	<code>webdrivermanager.properties</code>	Properties file (in the project classpath) for configuration values
<code>avoidExport()</code>	<code>wdm.avoidExport=true</code>	<code>false</code> (export driver paths as Java properties (e.g. <code>webdriver.chrome.driver</code> ))	Avoid step 4 in the <a href="#">resolution algorithm</a> (for instance, in the <a href="#">CLI</a> mode)
<code>exportParameter(String)</code>	<code>wdm.chromeDriverExport</code> , <code>wdm.geckoDriverExport</code> , <code>wdm.edgeDriverExport</code> , <code>wdm.iExplorerDriverExport</code> , <code>wdm.operaDriverExport</code>	Java property name used to export the driver path (available in <a href="#">webdrivermanager.properties</a> )	Set custom property name
<code>avoidOutputTree()</code>	<code>wdm.avoidOutputTree=true</code>	<code>false</code> (create output tree in the driver cache, e.g., <code>chromedriver/linux64/2.37</code> )	Avoid output tree (for instance, in the <a href="#">CLI</a> mode)
<code>avoidFallback()</code>	<code>wdm.avoidFallback=true</code>	<code>false</code> (use a retries mechanism if any problem happens during the resolution algorithm)	Avoid the fallback mechanism
<code>avoidBrowserDetection()</code>	<code>wdm.avoidBrowserDetection=true</code>	<code>false</code> (browser version is detected, and the corresponding driver version is discovered)	Force to use the latest version available for a given driver
<code>avoidReadReleaseFromRepository()</code>	<code>wdm.avoidReadReleaseFromRepository=true</code>	<code>false</code> (discover driver release using info from the repository, e.g., <a href="#">chromedriver-latest</a> or <a href="#">msedgedriver-latest</a> )	Avoid using the repository info and use the <a href="#">versions database</a> instead
<code>avoidTmpFolder()</code>	<code>wdm.avoidTmpFolder=true</code>	<code>false</code> (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
<code>browserVersionDetectionCommand(String)</code>	<code>wdm.browserVersionDetectionCommand</code>	<code>""</code> (automatic discovery using the <a href="#">commands database</a> )	Custom browser version detection command (see example <a href="#">here</a> )
<code>browserVersionDetectionRegex(String)</code>	<code>wdm.browserVersionDetectionRegex</code>	<code>[^\\d\\^\\.]</code>	Regular expression used to extract the browser version from the shell
<code>useLocalVersionsPropertiesFirst()</code>	<code>wdm.versionsPropertiesOnlineFirst=true</code>	<code>false</code> (the online <a href="#">versions database</a> is used in the <a href="#">resolution algorithm</a> )	Use local copy of the versions database
<code>useLocalCommandsPropertiesFirst()</code>	<code>wdm.commandsPropertiesOnlineFirst=true</code>	<code>false</code> (the online <a href="#">commands database</a> is used in the <a href="#">resolution algorithm</a> )	Use local copy of the commands database
<code>versionsPropertiesUrl(URL)</code>	<code>wdm.versionsPropertiesUrl</code>	Raw version of the online <a href="#">versions database</a>	Change versions database URL
<code>commandsPropertiesUrl(URL)</code>	<code>wdm.commandsPropertiesUrl</code>	Raw version of the online <a href="#">cmands database</a>	Change commands database URL
<code>clearDriverCache()</code>	<code>wdm.clearDriverCache=true</code>	<code>false</code> (not cleaning driver cache)	Clean driver cache
<code>clearResolutionCache()</code>	<code>wdm.clearResolutionCache=true</code>	<code>false</code> (not cleaning resolution cache)	Clean resolution cache
<code>ttl(int)</code>	<code>wdm.ttl</code>	<code>86400</code> (i.e., 1 day)	TTL in seconds in which the resolved driver versions are valid in the resolution cache.
<code>ttlBrowsers(int)</code>	<code>wdm.ttlForBrowsers</code>	<code>3600</code> (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
<code>dockerDaemonUrl(String)</code>	<code>wdm.dockerDaemonUrl</code>	<code>""</code>	URL of remote <a href="#">Docker daemon</a>

API method	Configuration key	Default value	Description
<code>dockerTimezone(String)</code>	<code>wdm.dockerTimezone</code>	Etc/UTC	<a href="#">Timezone</a> in browser container
<code>dockerNetwork(String)</code>	<code>wdm.dockerNetwork</code>	bridge	Docker network name
<code>dockerLang(String)</code>	<code>wdm.dockerLang</code>	EN	<a href="#">Language</a> in Docker container
<code>dockerShmSize(String)</code>	<code>wdm.dockerShmSize</code>	256m	Docker <a href="#">shared memory</a> in bytes. Unit is optional and can be <b>b</b> (bytes), <b>k</b> (kilobytes), <b>m</b> (megabytes), or <b>g</b> (gigabytes)
<code>dockerTmpfsSize(String)</code>	<code>wdm.dockerTmpfsSize</code>	128m	Docker in-memory filesystem ( <a href="#">tmpfs</a> ). Units follows the same approach than the shared memory
<code>dockerTmpfsMount(String)</code>	<code>wdm.dockerTmpfsMount</code>	/tmp	Mount point for in-memory filesystem
<code>dockerStopTimeoutSec(Integer)</code>	<code>wdm.dockerStopTimeoutSec</code>	5	Max time to kill a container (in seconds) after stopping it
<code>enableVnc()</code>	<code>wdm.dockerEnableVnc=true</code>	false	Enable <a href="#">desktop remote session</a> for browsers in Docker
<code>enableRecording()</code>	<code>wdm.dockerEnableRecording=true</code>	false	Enable the <a href="#">recordings</a> of the browser session in Docker
<code>dockerScreenResolution(String)</code>	<code>wdm.dockerScreenResolution</code>	1280x1080x24	Screen resolution of the browser desktop session in format <code>&lt;width&gt;x&lt;height&gt;x&lt;colors-depth&gt;</code>
<code>dockerRecordingFrameRate(int)</code>	<code>wdm.dockerRecordingFrameRate</code>	12	Frame rate for recordings
<code>dockerRecordingOutput(String)</code> <code>dockerRecordingOutput(Path)</code>	<code>wdm.dockerRecordingOutput</code>	.	Path for the recording output
<code>dockerRecordingPrefix(String)</code>	<code>wdm.dockerRecordingPrefix</code>	Browser name	Prefix to be appended to the session id (in the recording filename)

API method	Configuration key	Default value	Description
<code>dockerCustomImage(String)</code>	<code>wdm.dockerCustomImage</code>	<code>""</code>	Custom image to be used as browser in Docker
<code>dockerVolume(String)</code> <code>dockerVolumes(String[])</code>	<code>wdm.dockerVolumes</code>	<code>""</code>	Comma-separated volumes (or string array) using the format " <code>\local\path:\container\path</code> "

Table 3. Configuration capabilities for WebDriverManager Server

API method	Configuration key	Default value	Description
<code>serverPort(int)</code>	<code>wdm.serverPort</code>	<code>4444</code>	Port of WebDriverManager Server
<code>serverPath(String)</code>	<code>wdm.serverPath</code>	<code>/</code>	Path of WebDriverManager Server
<code>serverTimeoutSec(int)</code>	<code>wdm.serverTimeoutSec</code>	<code>60</code>	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 to some extent by using the argument `--disable-gpu` in Chrome and Edge. This argument already passed to these containers to avoid the issue. Nevertheless, some situations are still impossible to fix (e.g., when using <https://hub.docker.com/r/bonigarcia/webdrivermanager> [WebDriverManager in Docker in the CLI or Server mode]).

# Chapter 8. Support

There are several ways to try to get community support related WebDriverManager. First, questions about WebDriverManager can be discussed in [StackOverflow](#), using the tag *webdrivermanager\_java*. In addition, comments, suggestions, and bug-reporting should be made using the [GitHub issues](#).

# Chapter 9. Contributing

If you think WebDriverManager can be enhanced, consider contributing to the project through a [pull request](#).

In addition, both [WebDriverManager](#) and [Selenium-Jupiter](#) are in [OpenCollective](#), an online funding platform for open and transparent communities. You can contribute as a [donation](#) (i.e., an individual custom contribution), as a [backer](#) (i.e., an individual recurring contribution), or as [sponsor](#) (i.e., a recurring contribution by a company).

# Chapter 10. Further Documentation

There are other resources related to WebDriverManager and automated testing you can find helpful. For instance, 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.

Or the following book:

- García, Boni. [Mastering Software Testing with JUnit 5](#). Packt Publishing Ltd, 2017.



# Chapter 11. About

WebDriverManager (Copyright © 2017-2021) is an open-source project created and maintained by [Boni García \(@boni\\_gg\)](#), licensed under the terms of [Apache 2.0 License](#). This documentation (also available in [PDF](#) and [EPUB](#)) is released under the terms of [CC BY-NC-SA 2.0](#).