

Selenium-Jupiter

JUnit 5 extension for Selenium

Boni García

Version 3.3.2

Table of Contents

Quick reference	1
Local browsers	3
Chrome	6
Firefox	7
Opera	8
Edge	9
Internet Explorer	10
Safari	10
PhantomJS	11
HtmlUnit	12
Selenide	12
Docker browsers	15
Chrome	16
Firefox	18
Opera	19
Edge	19
Internet Explorer	20
Android	21
Remote sessions (VNC)	23
Recordings	25
Performance tests	26
Selenide	28
Remote browsers	30
Using capabilities	30
Mobile testing with Appium	32
Using SauceLabs	34
Using Selenide	36
Advanced features	37
Using options	37
Template tests	40
Generic driver	46
Integration with Jenkins	47
Using Genymotion	51
Single session	53
Selenium-Jupiter CLI	55
Selenium-Jupiter Server	57
Configuration	58
Tuning WebDriverManager	64

Screenshots	66
Known issues	67
Testing localhost	67
Using old versions of Selenium	68
Recordings in Windows or Mac	69
About	70

JUnit 5 is the next generation of the well-known testing framework JUnit. *Jupiter* is the name given to the new programming and extension model provided by JUnit 5. The extension model of JUnit 5, it allows to incorporate extra capabilities for JUnit 5 tests. On the other hand, **Selenium** is a testing framework which allows to control local (*Selenium WebDriver*) or remote (*Selenium Grid*) browsers (e.g. Chrome, Firefox, and so on) programmatically to carry out automated testing of web applications. This documentation presents **Selenium-Jupiter**, a JUnit 5 extension aimed to provide seamless integration of Selenium and Appium for JUnit 5 tests. *Selenium-Jupiter* is open source (Apache 2.0 license) and is hosted on [GitHub](#).

Quick reference

Selenium-Jupiter has been built using the [dependency injection](#) capability provided by the extension model of JUnit 5. Thank to this feature, different types objects can be injected in JUnit 5 as methods or constructor parameters in `@Test` classes. Concretely, *Selenium-Jupiter* allows to inject subtypes of the **WebDriver** interface (e.g. *ChromeDriver*, *FirefoxDriver*, and so on).

Using *Selenium-Jupiter* is very easy. First, you need to import the dependency in your project (typically as *test* dependency). In Maven, it is done as follows:

```
<dependency>
  <groupId>io.github.bonigarcia</groupId>
  <artifactId>selenium-jupiter</artifactId>
  <version>3.3.2</version>
  <scope>test</scope>
</dependency>
```

The dependency (typically *test*) to be declared in a Gradle project is as follows:

```
dependencies {
    testCompile("io.github.bonigarcia:selenium-jupiter:3.3.2")
}
```

NOTE

Selenium-Jupiter 3.3.2 depends on **selenium-java 3.141.59**, **webdrivermanager 3.7.1**, **appium java-client 7.0.0**, **spotify docker-client 8.16.0**, **htmlunit 2.36.0**, and **selenide 5.3.1**. Therefore, by using the *Selenium-Jupiter* dependency, these libraries will be added as transitive dependencies to your project.

Then, you need to declare *Selenium-Jupiter* extension in your JUnit 5 test, simply annotating your test with `@ExtendWith(SeleniumExtension.class)`. Finally, you need to include one or more parameters in your `@Test` methods (or constructor) whose types implements the **WebDriver** interface (e.g. **ChromeDriver** to use Chrome, **FirefoxDriver** for Firefox, and so for). That's it. *Selenium-Jupiter*

control the lifecycle of the **WebDriver** object internally, and you just need to use the **WebDriver** object in your test to drive the browser(s) you want. For example:

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

import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class ChromeAndFirefoxJupiterTest {

    @Test
    public void testWithOneChrome(ChromeDriver chromeDriver) {
        // Use Chrome in this test
    }

    @Test
    public void testWithFirefox(FirefoxDriver firefoxDriver) {
        // Use Firefox in this test
    }

    @Test
    public void testWithChromeAndFirefox(ChromeDriver chromeDriver,
        FirefoxDriver firefoxDriver) {
        // Use Chrome and Firefox in this test
    }

}
```

NOTE

As of JUnit 5.1, extensions can be registered programmatically via **@RegisterExtension** or automatically via Java's **ServiceLoader**. Both mechanisms can be used with *Selenium-Jupiter*.

The **WebDriver** subtypes supported by *Selenium-Jupiter* are the following:

- **ChromeDriver**: Used to control Google Chrome browser.
- **FirefoxDriver**: Used to control Firefox browser.
- **EdgeDriver**: Used to control Microsoft Edge browser.
- **OperaDriver**: Used to control Opera browser.
- **SafariDriver**: Used to control Apple Safari browser (only possible in OSX El Capitan or greater).
- **HtmlUnitDriver**: Used to control HtmlUnit (headless browser).
- **PhantomJSDriver**: Used to control PhantomJS (headless browser).
- **InternetExplorerDriver**: Used to control Microsoft Internet Explorer.

- **RemoteWebDriver**: Used to control remote browsers (*Selenium Grid*).
- **AppiumDriver**: Used to control mobile devices (Android, iOS).
- **SelenideDriver**: Used to control web browsers with Selenide (Chrome by default).

WARNING

The browser to be used must be installed in the machine running the test beforehand (except in the case of **RemoteWebDriver**, in which the requirement is to know a Selenium Server URL). In the case of mobile devices (**AppiumDriver**), the emulator should be up and running in local or available in a Appium Server identified by an URL.

You can also inject **WebDriver** objects declaring them as constructor parameters, instead of as test method parameters. For example as follows:

```
import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.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.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class ChromeInConstructorJupiterTest {

    ChromeDriver driver;

    public ChromeInConstructorJupiterTest(ChromeDriver driver) {
        this.driver = driver;
    }

    @Test
    public void testGlobalChrome() {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}
```

This documentation contains a comprehensive collection of basic examples demonstrating the basic usage of *Selenium-Jupiter* in JUnit 5 tests using different types of browsers. All these examples are part of the test suite of *Selenium-Jupiter* and are executed on [Travis CI](#).

Local browsers

Selenium WebDriver allows to control different types of browsers (such as Chrome, Firefox, Edge,

and so on) programmatically using different programming languages. This is very useful to implement automated tests for web applications. Nevertheless, in order to use WebDriver, we need to pay a prize. For security reasons, the automated manipulation of a browser can only be done using native features of the browser. In practical terms, it means that a binary file must be placed in between the test using the WebDriver API and the actual browser. On the one hand, the communication between the WebDriver object and that binary is done using the ([W3C WebDriver specification](#), formerly called *JSON Wire Protocol*). It consists basically on a REST service using JSON for requests and responses. On the other hand, the communication between the binary and the browser is done using native capabilities of the browser. Therefore, the general schema of Selenium WebDriver can be illustrated as follows:

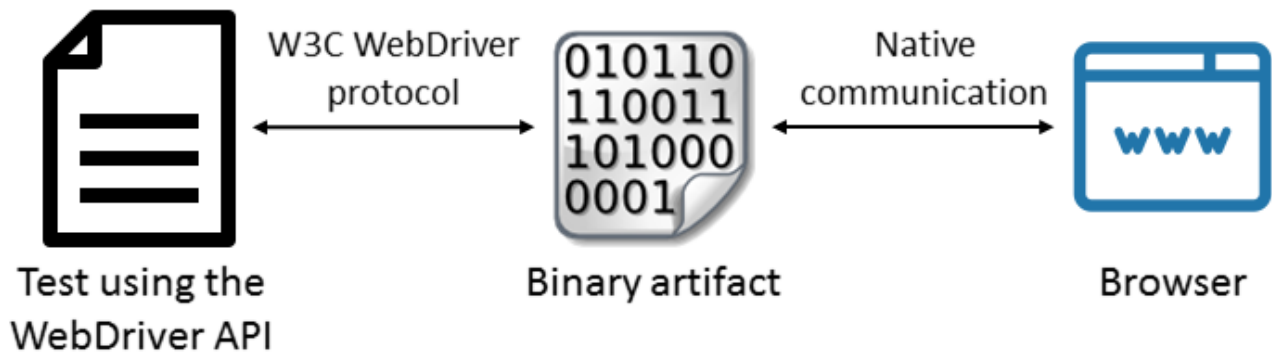


Figure 1. WebDriver general scenario

From a tester point of view, the need of this binary component is a pain in the neck, since it should be downloaded manually for the proper platform running the test (i.e. Windows, Linux, Mac). Moreover, the binary version should be constantly updated. The majority of browsers evolve quite fast, and the corresponding binary file required by WebDriver needs to be also updated. The following picture shows a fine-grained diagram of the different flavor of WebDriver binaries and browsers:

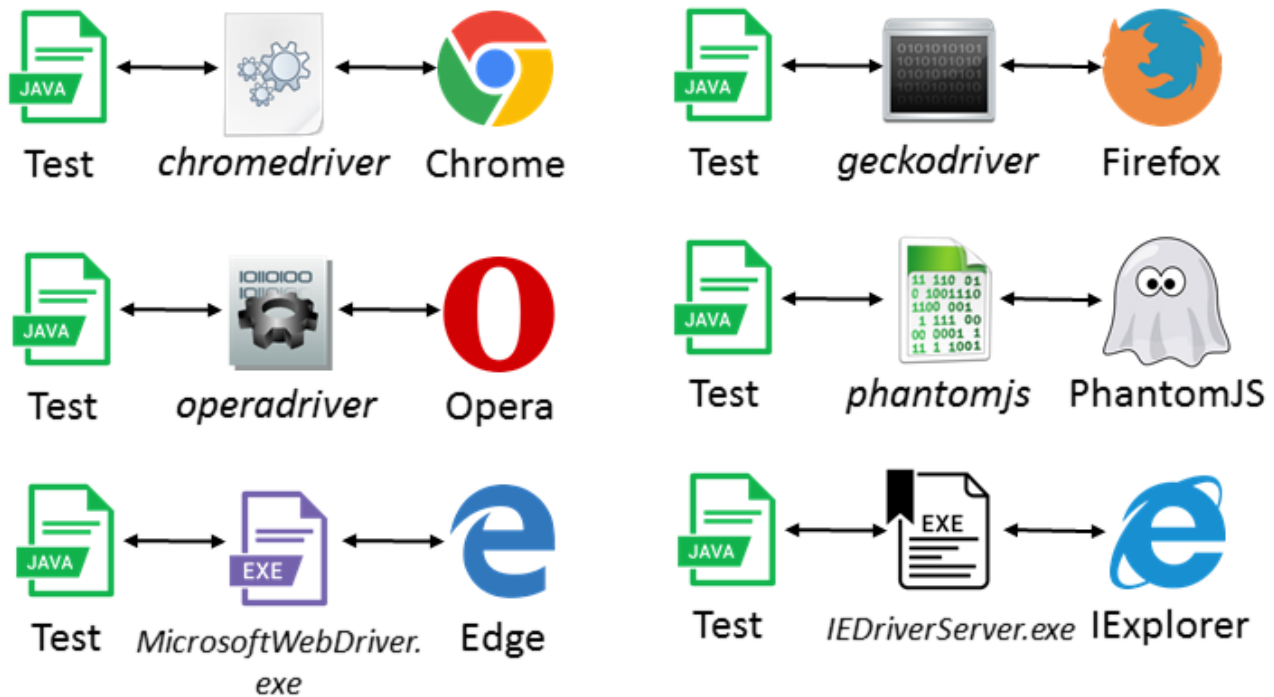


Figure 2. WebDriver scenario for Chrome, Firefox, Opera, PhantomJS, Edge, and Internet Explorer

Concerning Java, in order to locate these drivers, the absolute path of the binary controlling the browser should be exported in a given environment variable before creating a WebDriver instance, as follows:

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

In order to simplify the life of Java WebDriver users, in March 2015 the utility [WebDriverManager](#) was first released. WebDriverManager is a library which automates all this process (download the proper binary and export the proper variable) for Java in runtime. The WebDriverManager API is quite simple, providing a singleton object for each of the above mentioned browsers:

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

The solution implemented by WebDriverManager is today supported by similar tools for other languages, such as [webdriver-manager](#) for **Node.js** or [WebDriverManager.Net](#) for **.NET**.

On September 2017, a new major version of the well-know testing JUnit framework was released.

This leads to ***Selenium-Jupiter***, which can be seen as the natural evolution of *WebDriverManager* for **JUnit 5** tests. Internally, *Selenium-Jupiter* is built using two foundations:

1. It uses *WebDriverManager* to manage the binaries requires by *WebDriver*.
2. It uses the *dependency injection* feature of the extension model of JUnit 5 to inject *WebDriver* objects within `@Test` methods.

All in all, using Selenium *WebDriver* to control browsers using Java was never that easy. Using JUnit 5 and *Selenium-Jupiter*, you simply need to declare the flavor of browser you want to use in your test method (or constructor) and use it.

Chrome

The following example contains a simple usage of Chrome in JUnit 5. The complete source code of this test is hosted on [GitHub](#). Notice that this class contains two tests (methods annotated with `@Test`). The first one (`testWithOneChrome`) declares just one `ChromeDriver` parameter, and therefore this test controls a single Chrome browser. On the other hand, the second `@Test` (`testWithTwoChromes`) declares two different `ChromeDriver` parameters, and so, it controls two Chrome browsers.

```

import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.CoreMatchers.equalTo;
import static org.hamcrest.CoreMatchers.startsWith;
import static org.hamcrest.MatcherAssert.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.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
class ChromeJupiterTest {

    @Test
    void testWithOneChrome(ChromeDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }

    @Test
    void testWithTwoChromes(ChromeDriver driver1, ChromeDriver driver2) {
        driver1.get("http://www.seleniumhq.org/");
        driver2.get("http://junit.org/junit5/");
        assertThat(driver1.getTitle(), startsWith("Selenium"));
        assertThat(driver2.getTitle(), equalTo("JUnit 5"));
    }
}

```

Firefox

The following `test` uses Firefox as browser(s). To that aim, `@Test` methods simply need to include `FirefoxDriver` parameters.

```

import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.CoreMatchers.equalTo;
import static org.hamcrest.CoreMatchers.startsWith;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.firefox.FirefoxDriver;

import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class FirefoxJupiterTest {

    @Test
    public void testWithOneFirefox(FirefoxDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }

    @Test
    public void testWithTwoFirefoxs(FirefoxDriver driver1,
        FirefoxDriver driver2) {
        driver1.get("http://www.seleniumhq.org/");
        driver2.get("http://junit.org/junit5/");
        assertThat(driver1.getTitle(), startsWith("Selenium"));
        assertThat(driver2.getTitle(), equalTo("JUnit 5"));
    }
}

```

Opera

Are you one of the few using Opera? No problem, you can still make automated [tests](#) with JUnit 5, WebDriver, and *Selenium-Jupiter*, as follows:

```

import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.opera.OperaDriver;

import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class OperaJupiterTest {

    @Test
    public void test(OperaDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}

```

Edge

The following [example](#) uses one Edge browser. This test should be executed on a Windows machine with Edge.

```

import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.edge.EdgeDriver;

import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class EdgeJupiterTest {

    @Test
    void edgeTest(EdgeDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}

```

Internet Explorer

The following [example](#) uses one Internet Explorer browser. This test should be executed on a Windows machine with Internet Explorer.

```
import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.ie.InternetExplorerDriver;

import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class IExplorerJupiterTest {

    @Test
    void iExplorerTest(InternetExplorerDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}
```

Safari

You can also use [Safari](#) in conjunction with *Selenium-Jupiter*. Take into account that *SafariDriver* requires Safari 10 running on OSX El Capitan or greater.

```

import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.safari.SafariDriver;

import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class SafariJupiterTest {

    @Test
    public void test(SafariDriver driver) {
        driver.get("http://www.seleniumhq.org/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}

```

PhantomJS

PhantomJS is a headless browser (i.e. a browser without GUI), and it can be convenient for different types of tests. The following [example](#) demonstrates how to use PhantomJS with *Selenium-Jupiter*.

```

import static org.hamcrest.CoreMatchers.notNullValue;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.phantomjs.PhantomJSDriver;

import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class PhantomjsJupiterTest {

    @Test
    public void test(PhantomJSDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getPageSource(), notNullValue());
    }
}

```

HtmlUnit

HtmlUnit is another headless browser that can be used easily in a Jupiter test, for [example](#) like this:

```
import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.htmlunit.HtmlUnitDriver;

import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class HtmlUnitJupiterTest {

    @Test
    public void test(HtmlUnitDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}
```

Selenide

As of version 3.3.0 *Selenium-Jupiter* provides seamless integration with **Selenide**, which is a high-level testing framework which defines a fluent API on the top of Selenium WebDriver. For more information, take a look to [Selenide website](#).

In order to use a local browser with Selenide in *Selenium-Jupiter* we follow the same philosophy than in other browsers. Using the JUnit 5 dependency injection mechanism, we simply need to declare **SelenideDriver** parameters in our test (or constructor) parameters. *Selenium-Jupiter* will instantiate properly the objects, making them available for the test(s). Then, we simply need to invoke the Selenide API in our test body to assess some web under test.

Take a look to the following example. By default, a **SelenideDriver** object use Chrome driver. Therefore, the following test use the local Chrome browser (installed in the machine running the test) with Selenide:

```

import static com.codeborne.selenide.Condition.visible;
import static org.openqa.selenium.By.linkText;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;

import com.codeborne.selenide.SelenideDriver;
import com.codeborne.selenide.SelenideElement;

import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class SelenideDefaultJupiterTest {

    @Test
    public void testSelenide(SelenideDriver driver) {
        driver.open("https://bonigarcia.github.io/selenium-jupiter/");
        SelenideElement about = driver.$(linkText("About"));
        about.shouldBe(visible);
        about.click();
    }
}

```

Besides, *Selenium-Jupiter* provides the annotation `@SelenideConfiguration` to configure `SelenideDriver` objects. This annotation can be used at parameter-level (i.e. to configure a single object) or field-level (i.e. to configure all the objects within a test class). The following test shows a couple of example of parameter-level configuration. The first test use a Firefox browser in headless mode. On the other hand, the second test uses two different browsers in the same test: Chrome and Firefox.


```

import static com.codeborne.selenide.Browsers.CHROME;
import static com.codeborne.selenide.Browsers.FIREFOX;
import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;

import com.codeborne.selenide.SelenideDriver;

import io.github.bonigarcia.seljup.SelenideConfiguration;
import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class SelenideParameterConfigJupiterTest {

    @Test
    public void testHeadlessFirefoxSelenide(
        @SelenideConfiguration(browser = FIREFOX, headless = true) SelenideDriver
        driver) {
        exercise(driver);
    }

    @Test
    public void testChromeAndFirefoxSelenide(
        @SelenideConfiguration(browser = CHROME) SelenideDriver chrome,
        @SelenideConfiguration(browser = FIREFOX) SelenideDriver firefox) {
        exercise(chrome, firefox);
    }

    private void exercise(SelenideDriver... drivers) {
        for (SelenideDriver driver : drivers) {
            driver.open("https://bonigarcia.github.io/selenium-jupiter/");
            assertThat(driver.title(),
                containsString("JUnit 5 extension for Selenium"));
        }
    }
}

```

The annotation `@SelenideConfiguration` can also be used to configure globally Selenide in a test class by annotating fields of the type `SelenideConfig`, for example as follows:

```

import static com.codeborne.selenide.Browsers.FIREFOX;
import static com.codeborne.selenide.Condition.visible;
import static org.openqa.selenium.By.linkText;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;

import com.codeborne.selenide.SelenideConfig;
import com.codeborne.selenide.SelenideDriver;
import com.codeborne.selenide.SelenideElement;

import io.github.bonigarcia.seljup.SelenideConfiguration;
import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class SelenideGlobalConfigJupiterTest {

    @SelenideConfiguration
    SelenideConfig selenideConfig = new SelenideConfig().browser(FIREFOX)
        .startMaximized(true);

    @Test
    public void testSelenideConfig(SelenideDriver driver) {
        driver.open("https://bonigarcia.github.io/selenium-jupiter/");
        SelenideElement about = driver.$(linkText("About"));
        about.shouldBe(visible);
        about.click();
    }
}

```

NOTE

Chrome options (arguments and preferences) and Firefox profiles can be configured in Selenide using JVM properties using the command line as `-Dchromeoptions.args=,..., -Dchromeoptions.prefs=,..., and -Dfirefoxprofile.=,.`

Docker browsers

As of version 2.0.0, *Selenium-Jupiter* allows to ask for browsers in [Docker](#) containers. The only requirement to use this feature is to install [Docker Engine](#) in the machine running the tests. Internally, *Selenium-Jupiter* uses a [docker-client](#) and different Docker images for browsers, namely:

- Stable versions of Docker browser, provided by [Selenoid](#).
- Beta and nightly versions of Docker browser, provided by [ElasticTest](#).
- Browsers in Android devices, provided by [Budi Utomo](#).

As shown in the following section, the mode of operation is similar to local browser. We simply ask for browsers in Docker simply declaring parameters in our `@Test` methods, and *Selenium-Jupiter*

will make magic for us: it downloads the proper Docker image for the browser, start it, and instantiate the object of type `WebDriver` or `RemoteWebDriver` to control the browser from our test. The annotation `@DockerBrowser` need to be declared in the parameter to mark the `WebDriver` object as a browser in Docker.

Chrome

The following example contains a simple test example using Chrome browsers in Docker. Check out the code [here](#). As you can see, the first `@Test` method (called `testChrome`) declares a parameter of type `RemoteWebDriver`. This parameter is annotated with `@DockerBrowser`. This annotation requires to set the browser type, in this case `CHROME`. If no version is specified, then the latest version of the browser will be used. This feature is known as **evergreen Docker browsers**, and it is implementing by consuming the REST API of [Docker Hub](#), asking for the list of [Selenium](#) browsers. On the other hand, the second `@Test` (called `testChromeWithVersion`) a fixed version is set, in this case `67.0`.

```
import static io.github.bonigarcia.seljup.BrowserType.CHROME;
import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.remote.RemoteWebDriver;

import io.github.bonigarcia.seljup.DockerBrowser;
import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class DockerChromeJupiterTest {

    @Test
    public void testChrome(
        @DockerBrowser(type = CHROME) RemoteWebDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }

    @Test
    public void testChromeWithVersion(
        @DockerBrowser(type = CHROME, version = "76.0") RemoteWebDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}
```

In this other example, wildcards are used to set the browser version. In the first `@Test` (method

`testLatestChrome`), we use the literal `latest` to mark the use of the latest version (in fact the use of `latest` is exactly the same that not declaring the `version` attribute). The second `@Test` (method `testFormerChrome`) sets the version as `latest-1`. This should be read as *latest version minus one*, in other words, the previous version to the stable version at the time of the test execution. Notice that the concrete versions for both test will evolve in time, since new versions are released constantly. All in all, you have the certainty of using the latest versions of the browser without any kind of extra configuration nor maintenance of the underlying infrastructure.

Moreover, in third (`testBetaChrome`) and fourth (`testUnstableChrome`) test **beta** and **unstable** (i.e. development) versions are used. This feature is available both for [Chrome](#) and [Firefox](#), thanks to the Docker images provided by the [ElasTest project](#).

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.WebDriver;

import io.github.bonigarcia.seljup.DockerBrowser;
import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class DockerChromeLatestJupiterTest {

    @Test
    public void testLatestChrome(
        @DockerBrowser(type = CHROME, version = "latest") WebDriver driver) {
        // Use stable version of Chrome in this test
    }

    @Test
    public void testFormerChrome(
        @DockerBrowser(type = CHROME, version = "latest-1") WebDriver driver) {
        // Use previous to stable version of Chrome in this test
    }

    @Test
    public void testBetaChrome(
        @DockerBrowser(type = CHROME, version = "beta") WebDriver driver) {
        // Use beta version of Chrome in this test
    }

    @Test
    public void testUnstableChrome(
        @DockerBrowser(type = CHROME, version = "unstable") WebDriver driver) {
        // Use development version of Chrome in this test
    }
}
```

NOTE

The label *latest-** is supported, where *** is a number for a former version to the current stable. For instance, *latest-2* means the two previous version to the stable (for instance, if at the time of running a test the latest version is 63.0, *latest-2* will mean version 61.0).

NOTE

As of version 3.2.2, *Selenium-Jupiter* allows to define an array of volumes to be mounted in the Docker container using syntax "`\local\path:\container\path`". The character `~` can be used to specify the home directory in the local host. For example: `@DockerBrowser(type = CHROME, volumes = {"~/home/selenium" }) WebDriver driver.`

Firefox

The use of Firefox is equivalent. With respect to the previous example, it simply change the type of browser. Versioning works exactly the same.

```
import static io.github.bonigarcia.seljup.BrowserType.FIREFOX;
import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.remote.RemoteWebDriver;

import io.github.bonigarcia.seljup.DockerBrowser;
import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class DockerFirefoxJupiterTest {

    @Test
    public void testLatest(
        @DockerBrowser(type = FIREFOX) RemoteWebDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }

    @Test
    public void testVersion(
        @DockerBrowser(type = FIREFOX, version = "64") RemoteWebDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}
```

NOTE

Notice that the version of the second test is simply **64**. The actual version of the image is **64.0**, but *Selenium-Jupiter* supposes that version is **.0** if not specified.

Opera

Again, the use of Opera browsers in Docker is the same, simply changing the browser type to **OPERA**.

```
import static io.github.bonigarcia.seljup.BrowserType.OPERA;
import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.remote.RemoteWebDriver;

import io.github.bonigarcia.seljup.DockerBrowser;
import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class DockerOperaJupiterTest {

    @Test
    public void testOpera(@DockerBrowser(type = OPERA) RemoteWebDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}
```

Edge

Selenium-Jupiter can be used to control a Microsoft Edge browser in a Docker container. Due to licensing issues, the required Docker image is not available in Docker Hub, and it must be built in local, following the [tutorial provided by Aerokube](#). If that tutorial is followed step by step, a Docker image named **windows/edge:18** will be available in the local machine. Then, *Selenium-Jupiter* can be used to control the Edge browser as follows:

```

import static io.github.bonigarcia.seljup.BrowserType.EDGE;
import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.RegisterExtension;
import org.openqa.selenium.WebDriver;

import io.github.bonigarcia.seljup.DockerBrowser;
import io.github.bonigarcia.seljup.SeleniumExtension;

public class DockerEdgeJupiterTest {

    @RegisterExtension
    static SeleniumExtension seleniumExtension = new SeleniumExtension();

    @BeforeAll
    static void setup() {
        seleniumExtension.getConfig().setVnc(true);
        seleniumExtension.getConfig().setRecording(true);
    }

    @Test
    public void testEdge(@DockerBrowser(type = EDGE) WebDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}

```

Internet Explorer

The same way than Edge, *Selenium-Jupiter* can also be used to control a Internet Explorer browser in a Docker container. This time, the [tutorial provided by Aerokube](#) need to be used to build the `windows/ie:11` image. Then, *Selenium-Jupiter* uses that image, for example as follows:

```

import static io.github.bonigarcia.seljup.BrowserType.IEXPLORER;
import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.RegisterExtension;
import org.openqa.selenium.WebDriver;

import io.github.bonigarcia.seljup.DockerBrowser;
import io.github.bonigarcia.seljup.SeleniumExtension;

public class DockerIExplorerJupiterTest {

    @RegisterExtension
    static SeleniumExtension seleniumExtension = new SeleniumExtension();

    @BeforeAll
    static void setup() {
        seleniumExtension.getConfig().setVnc(true);
        seleniumExtension.getConfig().setRecording(true);
    }

    @Test
    public void testIExplorer(
        @DockerBrowser(type = IEXPLORER) WebDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}

```

Android

The use of browser devices in *Selenium-Jupiter* is straightforward. Simply change the browser type to **ANDROID** as follows.


```

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.RegisterExtension;
import org.openqa.selenium.remote.RemoteWebDriver;

import io.github.bonigarcia.seljup.DockerBrowser;
import io.github.bonigarcia.seljup.SeleniumExtension;

public class AndroidJupiterTest {

    @RegisterExtension
    static SeleniumExtension seleniumExtension = new SeleniumExtension();

    @BeforeAll
    static void setup() {
        seleniumExtension.getConfig().setVnc(true);
        seleniumExtension.getConfig().setRecording(true);
    }

    @Test
    public void testAndroid(
        @DockerBrowser(type = ANDROID, version = "8.1", deviceName = "Nexus S")
        RemoteWebDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}

```

The supported Android devices are summarized in the following table:

Table 1. Android devices in Selenium-Jupiter

Android version	API level	Browser name
5.0.1	21	browser
5.1.1	22	browser
6.0	23	chrome
7.0	24	chrome
7.1.1	25	chrome
8.0	26	chrome
8.1	27	chrome
9.0	28	chrome

Moreover, the type of devices are the following:

Table 2. Android devices types in Selenium-Jupiter

Type	Device name
Phone	Samsung Galaxy S6
Phone	Nexus 4
Phone	Nexus 5
Phone	Nexus One
Phone	Nexus S
Tablet	Nexus 7

Android (9.0) with Chrome in **Samsung Galaxy S6** is the default setup for Android devices. These values can be changed using the [configuration](#) capabilities or by means of the the following parameters in the `@DockerBrowser` annotation

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.remote.RemoteWebDriver;

import io.github.bonigarcia.seljup.DockerBrowser;
import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class AndroidCustomJupiterTest {

    @Test
    public void testAndroid(@DockerBrowser(type = ANDROID, version = "9.0",
        deviceName = "Nexus S") RemoteWebDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}
```

Remote sessions (VNC)

Selenium-Jupiter allows to track the evolution of our browsers in Docker using Virtual Network Computing (VNC) sessions. This feature is disabled by default, but it can be activated using the configuration key `sel.jup.vnc` to `true` (more info on section [Configuration](#)).

When it is activated, the VNC URL of a browser in Docker is printed in the test log, concretely using the *DEBUG* level (see example below). Simply copying and pasting that URL in a real browser we can take a look to the browser while the test is being executed. We can even interact with the Docker browser.

```
2018-03-31 17:07:03 [main] INFO i.g.b.handler.DockerDriverHandler - VNC URL (copy and
paste in a browser navigation bar to interact with remote session)
2018-03-31 17:07:03 [main] INFO i.g.b.handler.DockerDriverHandler -
http://192.168.99.100:32769/vnc.html?host=192.168.99.100&port=32768&path=vnc/aa39e2562
bf0f58bfbad0924d22ca958&resize=scale&autoconnect=true&password=selenoid
```

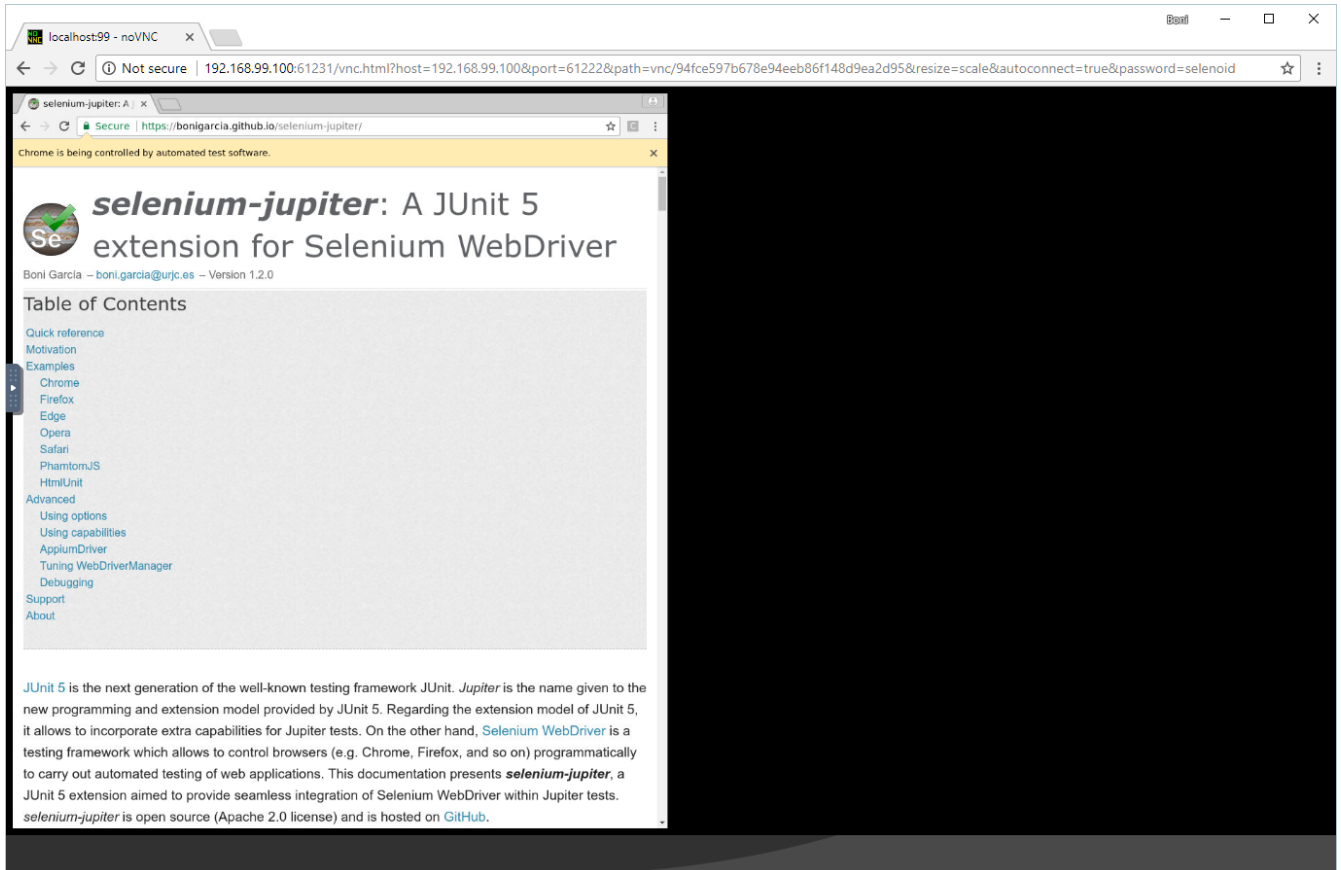


Figure 3. Example of VNC session of Chrome (desktop)

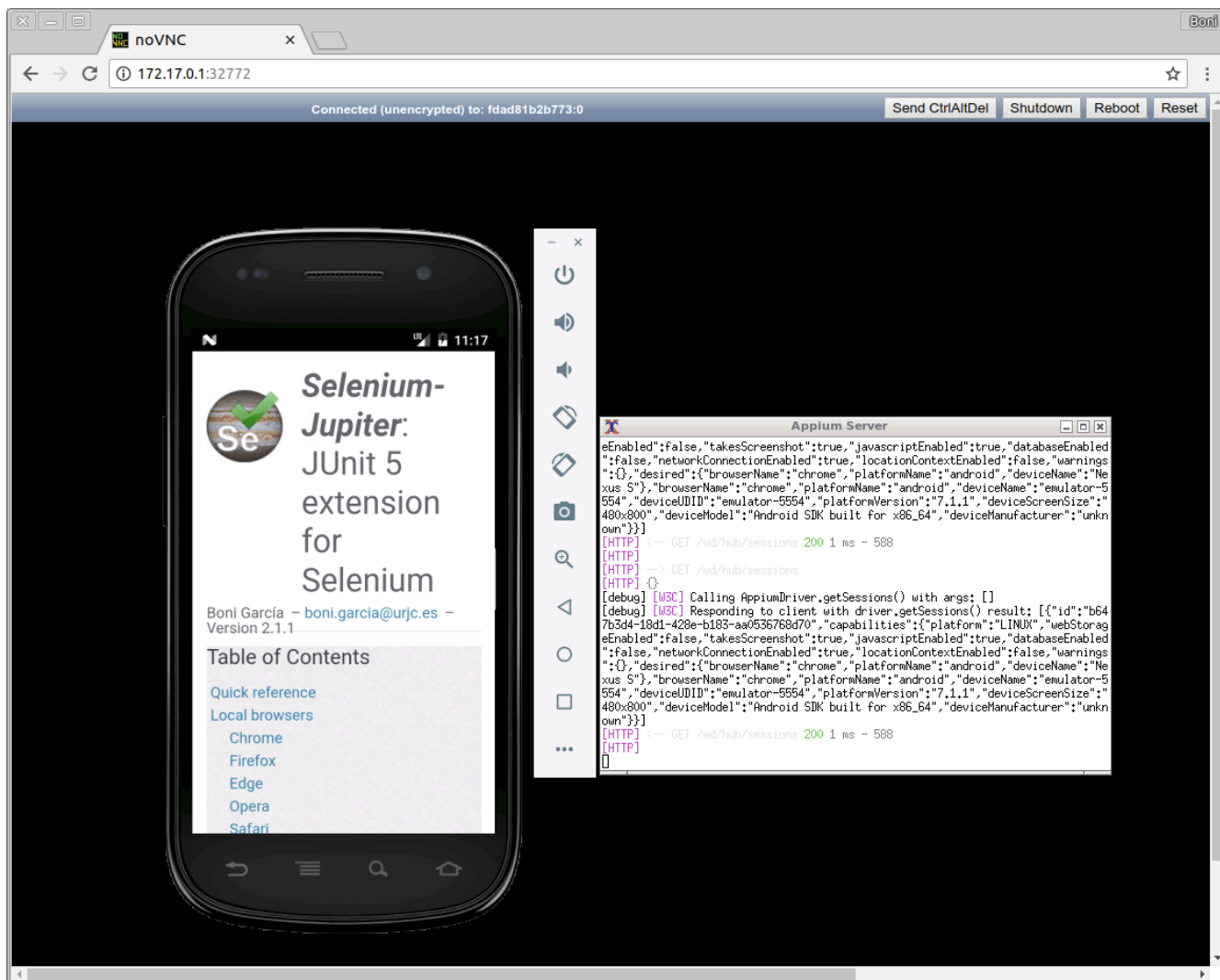


Figure 4. Example of VNC session of Chrome (Android)

NOTE

In addition to log the VNC URL, as of *Selenium-Jupiter* 2.1.0, the value of this URL is exported as Java property as `vnc.session.url` (i.e. `System.setProperty("vnc.session.url", vncUrl);`).

Recordings

Selenium-Jupiter allows to record the sessions of browsers in Docker. This capability is not activated by default, but it activated simply setting the configuration key `sel.jup.recording` to `true` (see section [Configuration](#) for further details about configuration).

This way, a recording in MP4 format will be stored at the end of the test which uses one or several browsers in Docker. The output folder in which the recording is stored is configured by means of the configuration key `sel.jup.output.folder`, whose default value is `.` (i.e. the current folder in which the test is executed). The following picture shows an example of recording.

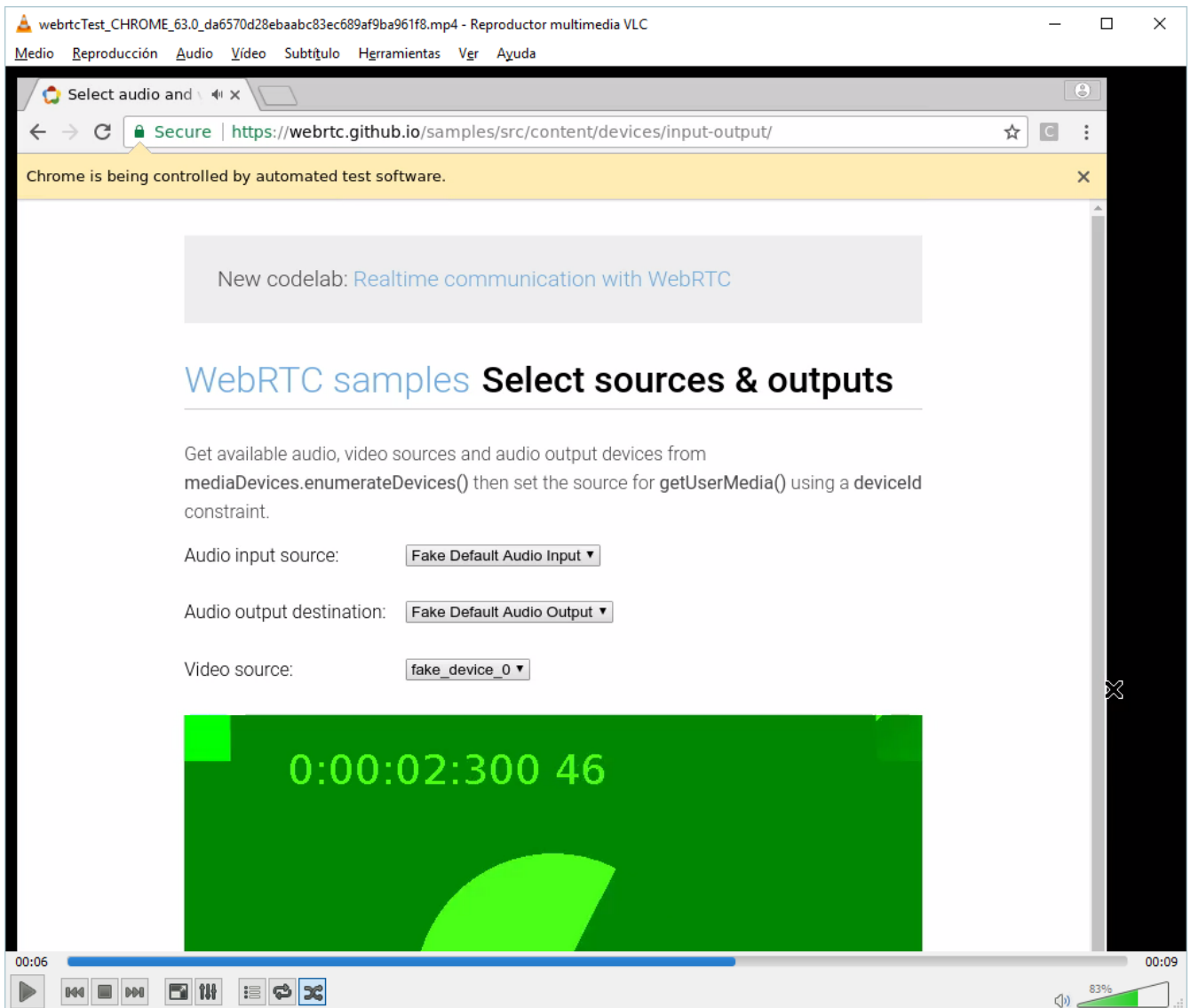


Figure 5. Example of recording played in VLC

Performance tests

Another important new feature of browsers in Docker is the possibility of asking for *many of them* by the same test. This can be used to implement performance tests in a seamless way. To use this feature, we need into account two aspects. First of all, the attribute `size` of the annotation `@DockerBrowser` should be declared. This numeric value sets the number of browsers demanded by the test. Second, the test declares a `List<RemoteWebDriver>` (or `List<WebDriver>`). For example as follows:

```
import static io.github.bonigarcia.seljup.BrowserType.CHROME;
import static java.lang.invoke.MethodHandles.lookup;
import static java.util.concurrent.Executors.newFixedThreadPool;
import static java.util.concurrent.TimeUnit.SECONDS;
import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.slf4j.LoggerFactory.getLogger;

import java.util.List;
```

```

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.remote.RemoteWebDriver;
import org.slf4j.Logger;

import io.github.bonigarcia.seljup.DockerBrowser;
import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class PerformanceDockerChromeJupiterTest {

    static final int NUM_BROWSERS = 3;

    final Logger log = getLogger(lookup().lookupClass());

    @Test
    public void testPerformance(
        @DockerBrowser(type = CHROME, version = "76.0", size = NUM_BROWSERS) List
        <RemoteWebDriver> driverList)
        throws InterruptedException {

        ExecutorService executorService = newFixedThreadPool(NUM_BROWSERS);
        CountDownLatch latch = new CountDownLatch(NUM_BROWSERS);

        driverList.forEach((driver) -> {
            executorService.submit(() -> {
                try {
                    log.info("Session id {}",
                        ((RemoteWebDriver) driver).getSessionId());
                    driver.get(
                        "https://bonigarcia.github.io/selenium-jupiter/");
                    assertThat(driver.getTitle(), containsString(
                        "JUnit 5 extension for Selenium"));
                } finally {
                    latch.countDown();
                }
            });
        });

        latch.await(50, SECONDS);
        executorService.shutdown();
    }
}

```

This example requires a list of 3 Chrome browsers in Docker. Then, it executed in parallel a given logic. Notice that if the number of browsers is high, the CPU and memory consumption of the test

running the machine will increase accordingly.

Selenide

Browsers in Docker containers can be also controlled using Selenide very easily in *Selenium-Jupiter*. As usual, we need to declare `SelenideDriver` parameters in our test (or constructor), but this time annotated also with `@DockerBrowser`. For example, the following class implements two tests. The first one use the latest version of Chrome and the second test uses the beta version of Firefox. Both browsers are executed in Docker containers, and their versions (latest and beta) are figured out in test runtime in a transparent way for developers. Then, in the tests logic, we control the browsers in Docker containers using the fluent Selenide API.

```

import static com.codeborne.selenide.Condition.visible;
import static io.github.bonigarcia.seljup.BrowserType.CHROME;
import static io.github.bonigarcia.seljup.BrowserType.FIREFOX;
import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.openqa.selenium.By.linkText;

import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.RegisterExtension;

import com.codeborne.selenide.SelenideDriver;
import com.codeborne.selenide.SelenideElement;

import io.github.bonigarcia.seljup.DockerBrowser;
import io.github.bonigarcia.seljup.SeleniumExtension;

public class SelenideDockerJupiterTest {

    @RegisterExtension
    static SeleniumExtension seleniumExtension = new SeleniumExtension();

    @BeforeAll
    static void setup() {
        seleniumExtension.getConfig().setVnc(true);
    }

    @Test
    public void testDockerSelenideChrome(
        @DockerBrowser(type = CHROME) SelenideDriver driver) {
        driver.open("https://bonigarcia.github.io/selenium-jupiter/");
        SelenideElement about = driver.$(linkText("About"));
        about.shouldBe(visible);
        about.click();
    }

    @Test
    public void testDockerSelenideFirefox(
        @DockerBrowser(type = FIREFOX, version = "beta") SelenideDriver driver) {
        driver.open("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.title(),
            containsString("JUnit 5 extension for Selenium"));
    }
}

```

As usual, we can use the remote access with VNC or the recording capabilities provided by *Selenium-Jupiter* when using Docker. In the example before, we enable the remote access by VNC.

Remote browsers

Selenium-Jupiter also supports remote browsers, using a [Selenium Grid](#) server. To that aim, a couple of custom annotations can be used:

- **DriverUrl** (*parameter-level* or *field-level*): Annotation used to identify the URL value needed to instantiate a **RemoteWebDriver** object. This URL can be setup using the configuration key `sel.jup.selenium.server.url`. This value can be used to use browsers from cloud providers, such as [Saucelabs](#) or [BrowserStack](#).
- **DriverCapabilities** (*parameter-level* or *field-level*): Annotation to configure the desired *capabilities* (WebDriver's object **DesiredCapabilities**).

Using capabilities

The annotation **@DriverCapabilities** is used to specify WebDriver capabilities (i.e. type browser, version, platform, etc.). These capabilities are typically used for Selenium Grid tests (i.e. tests using remote browsers). To that aim, a Selenium Hub (also known as *Selenium Server*) should be up and running, and its URL should be known. This URL will be specified using the *Selenium-Jupiter* annotation **@DriverUrl**.

The following example provides a complete [example](#) about this. As you can see, in the test setup (**@BeforeAll**) a Selenium Grid is implemented, first starting a Hub (a.k.a. *Selenium Server*), and then a couple of nodes (Chrome and Firefox) are registered in the Hub. Therefore, remote test using **RemoteWebDriver** can be executed, simply pointing to the Hub (whose URL in this case is <http://localhost:4444/wd/hub> in this example) and selecting the browser to be used using the **Capabilities**.

```

import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.openqa.selenium.remote.DesiredCapabilities.firefox;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.Capabilities;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.remote.RemoteWebDriver;

import io.github.bonigarcia.seljup.DriverCapabilities;
import io.github.bonigarcia.seljup.DriverUrl;
import io.github.bonigarcia.seljup.SeleniumExtension;
@ExtendWith(SeleniumExtension.class)
public class RemoteWebDriverJupiterTest {

    @DriverUrl
    String url = "http://localhost:4444/wd/hub";

    @DriverCapabilities
    Capabilities capabilities = firefox();

    @Test
    void testWithRemoteChrome(@DriverUrl("http://localhost:4444/wd/hub")
        @DriverCapabilities("browserName=chrome") RemoteWebDriver driver) {
        exercise(driver);
    }

    @Test
    void testWithRemoteFirefox(RemoteWebDriver driver) {
        exercise(driver);
    }

    void exercise(WebDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}

```

The following class contains an example which uses Chrome as browser and capabilities defined using `@DriverCapabilities`. Concretely, this example uses the mobile emulation feature provided out of the box by Chrome (i.e. render the web page using small screen resolutions to emulate smartphones).

```

import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;

import java.util.HashMap;
import java.util.Map;

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

import io.github.bonigarcia.seljup.Options;
import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class ChromeWithGlobalCapabilitiesJupiterTest {

    @Options
    ChromeOptions options = new ChromeOptions();
    {
        Map<String, String> mobileEmulation = new HashMap<>();
        mobileEmulation.put("deviceName", "Nexus 5");
        options.setExperimentalOption("mobileEmulation", mobileEmulation);
    }

    @Test
    void chromeTest(ChromeDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}

```

Mobile testing with Appium

The annotation `@DriverCapabilities` can be also used to specify the desired capabilities to create an instances of `AppiumDriver` to drive mobile devices (Android or iOS). If not `@DriverUrl` is specified, *Selenium-Jupiter* will start automatically an instance of Appium Server (by default in port 4723) in the localhost after each test execution (this server is shutdown before each test). For example:

```

import static org.junit.jupiter.api.Assertions.assertTrue;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.By;
import org.openqa.selenium.WebElement;

import io.appium.java_client.AppiumDriver;
import io.github.bonigarcia.seljup.DriverCapabilities;
import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class AppiumChromeJupiterTest {

    @Test
    void testWithAndroid(
        @DriverCapabilities({ "browserName=chrome",
                             "deviceName=Android" }) AppiumDriver<WebElement> driver)
        throws InterruptedException {

        String context = driver.getContext();
        driver.context("NATIVE_APP");
        driver.findElement(By.id("com.android.chrome:id/terms_accept")).click();
        driver.findElement(By.id("com.android.chrome:id/negative_button"))
            .click();
        driver.context(context);

        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertTrue(driver.getTitle().contains("JUnit 5 extension"));
    }
}

```

We can also specify a custom Appium Server URL changing the value of `@DriverUrl`, at field-level or parameter-level:

```

import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.remote.DesiredCapabilities;

import io.appium.java_client.AppiumDriver;
import io.github.bonigarcia.seljup.DriverCapabilities;
import io.github.bonigarcia.seljup.DriverUrl;
import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class AppiumWithGlobalOptionsChromeJupiterTest {

    @DriverUrl
    String url = "http://localhost:4723/wd/hub";

    @DriverCapabilities
    DesiredCapabilities capabilities = new DesiredCapabilities();
    {
        capabilities.setCapability("browserName", "chrome");
        capabilities.setCapability("deviceName", "Samsung Galaxy S6");
    }

    @Test
    void testWithAndroid(AppiumDriver<WebElement> driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}

```

Using SauceLabs

Selenium-Jupiter can be used to control remote browsers in the [SauceLabs](#) cloud.

In the following example, first the `@DriverUrl` is used to specify the SauceLabs URL. Then the annotation `@DriverCapabilities` is used to configure the SauceLabs credentials (*username* and *accessKey*) together with the desired capabilities. In the example we use Chrome 59.0 in Windows 10:

```

import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.openqa.selenium.remote.RemoteWebDriver;

import io.github.bonigarcia.seljup.DriverCapabilities;
import io.github.bonigarcia.seljup.DriverUrl;
import io.github.bonigarcia.seljup.SeleniumExtension;

@Disabled
@ExtendWith(SeleniumExtension.class)
public class SauceLabsJupiterTest {

    @DriverUrl
    String url = "https://ondemand.eu-central-1.saucelabs.com/wd/hub";

    @DriverCapabilities
    DesiredCapabilities capabilities = new DesiredCapabilities();
    {
        capabilities.setCapability("username", "<my-saucelabs-user>");
        capabilities.setCapability("accessKey", "<my-saucelabs-key>");
        capabilities.setCapability("browserName", "Chrome");
        capabilities.setCapability("platform", "Windows 10");
        capabilities.setCapability("version", "59.0");
        capabilities.setCapability("name", "selenium-jupiter-and-saucelabs");
    }

    @Test
    void testWithSauceLabs(RemoteWebDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}

```

As usual, the execution of the test can be seen in the SauceLabs dashboard:

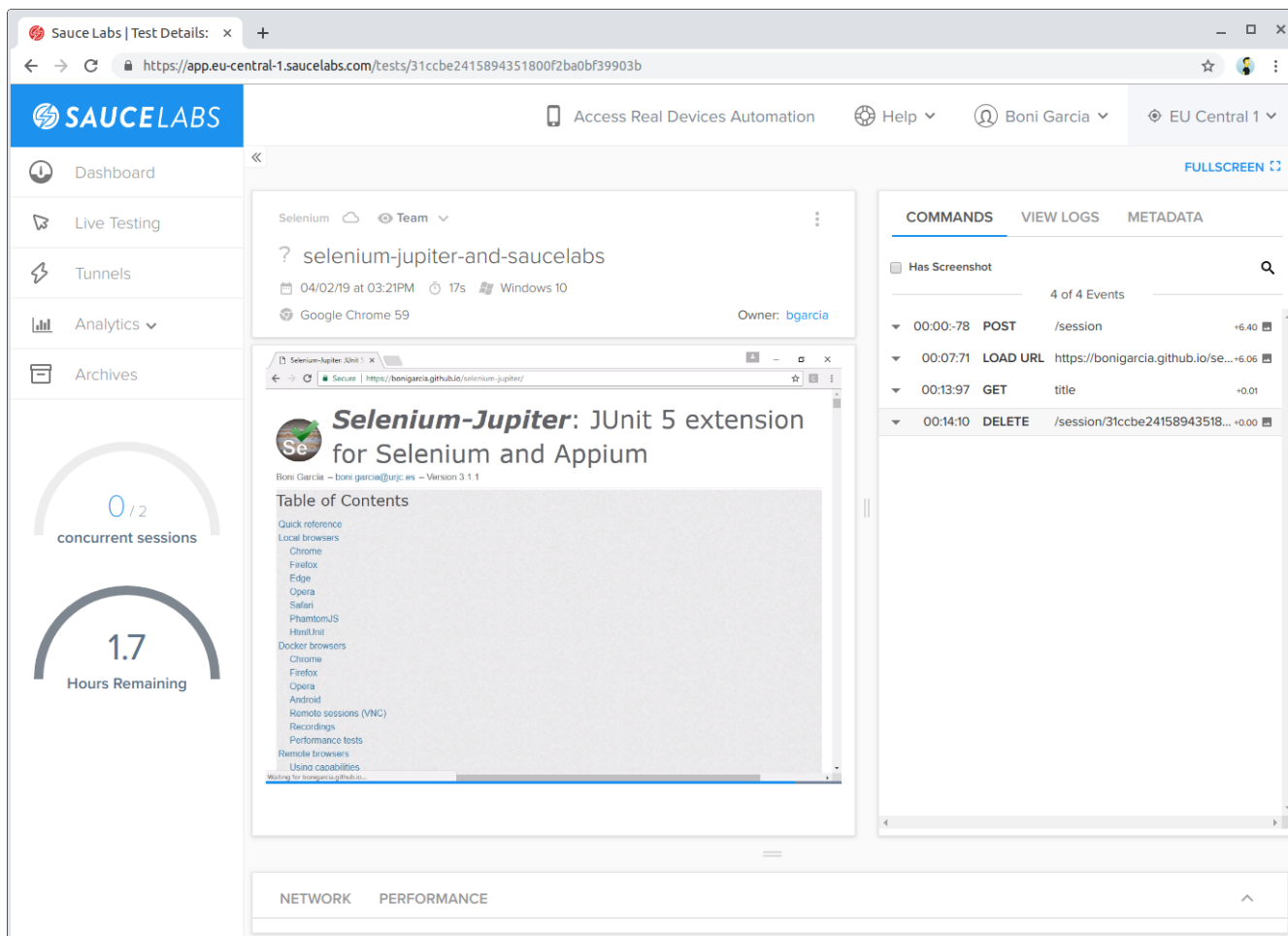


Figure 6. SauceLabs dashboard

Using Selenide

Selenium_Jupiter allows to control remote browsers also with Selenide. To do that, we simply need to declare `SelenideDriver` parameters in conjunction with the annotations `@DriverUrl` (to specify the Selenium Server URL) and `@DriverCapabilities` (to specify the desired capabilities). These annotations can be used both at parameter-level (applied to a single object) and field-level (applied globally in a test class).

The following test shows an example of both usages. On the other hand, the first test uses a `SelenideDriver` object which is going to connect to the Selenium Server given in the field `url` and with desired capabilities given in `capabilities` (i.e. Firefox). On the other hand, the second declares another `SelenideDriver` parameter configured to use the same Selenium Server in the localhost, but requesting a Chrome browser for this test.

```

import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.remote.DesiredCapabilities;

import com.codeborne.selenide.SelenideDriver;

import io.github.bonigarcia.seljup.DriverCapabilities;
import io.github.bonigarcia.seljup.DriverUrl;
import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class SelenideRemoteJupiterTest {

    @DriverUrl
    String url = "http://localhost:4444/wd/hub";

    @DriverCapabilities
    DesiredCapabilities capabilities = DesiredCapabilities.firefox();

    @Test
    public void testRemoteSelenideGlobalConfig(SelenideDriver driver) {
        exercise(driver);
    }

    @Test
    public void testRemoteSelenideParameterConfig(
        @DriverUrl("http://localhost:4444/wd/hub")
        @DriverCapabilities("browserName=chrome") SelenideDriver driver) {
        exercise(driver);
    }

    private void exercise(SelenideDriver driver) {
        driver.open("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.title(),
            containsString("JUnit 5 extension for Selenium"));
    }
}

```

Advanced features

Using options

So far, we have discovered how to use different local browsers (Chrome, Firefox, Edge, Opera, Safari, PhantomJS, HtmlUnit), Docker browsers (Chrome, Firefox, Opera), or even remote browsers

with Selenium Grid. In any case, the default options are used. Nevertheless, if you have used intensively Selenium WebDriver, different questions might come to your mind:

- What if I need to specify options (e.g. `ChromeOptions`, `FirefoxOptions`, etc) to my WebDriver object?
- What if need to specify desired capabilities (e.g. browser type, version, platform)?

In order to support the advance features of Selenium WebDriver, *Selenium-Jupiter* provides several annotations aimed to allow a fine-grained control of the WebDriver object instantiation. These annotations are:

- **Options** (*field-level*): Annotation to configure *options* (e.g. `ChromeOptions` for Chrome, `FirefoxOptions` for Firefox, `EdgeOptions` for Edge, `OperaOptions` for Opera, and `SafariOptions` for Safari).
- **Arguments** (*parameter-level*) : Used to add arguments to the options.
- **Preferences** (*parameter-level*) : Used to set preferences to the options.
- **Binary** (*parameter-level*) : Used to set the location of the browser binary.
- **Extensions** (*parameter-level*) : User to add extensions to the browser.

The annotations marked as *parameter-level* are applied to a single WebDriver parameter. The annotations marked as *field-level* are applied globally in a test class.

The following [example](#) shows how to specify options for Chrome. In the first test (called `headlessTest`), we are setting the argument `--headless`, used in Chrome to work as a headless browser. In the second test (`webrtcTest`), we are using two different arguments: `--use-fake-device-for-media-stream` and `--use-fake-ui-for-media-stream`, used to fake user media (i.e. camera and microphone) in `WebRTC` applications. In the third test (`extensionTest`), we are adding an extension to Chrome using the `@Extensions` annotation. The value of this field is an extension file that will be searched: i) using value as its relative/absolute path; ii) using value as a file name in the project classpath.

```

import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.CoreMatchers.equalTo;
import static org.hamcrest.MatcherAssert.assertThat;

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

import io.github.bonigarcia.seljup.Arguments;
import io.github.bonigarcia.seljup.Extensions;
import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class ChromeWithOptionsJupiterTest {

    @Test
    void headlessTest(@Arguments("--headless") ChromeDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }

    @Test
    void webrtcTest(@Arguments({ "--use-fake-device-for-media-stream",
        "--use-fake-ui-for-media-stream" }) ChromeDriver driver) {
        driver.get(
            "https://webrtc.github.io/samples/src/content/devices/input-output/");
        assertThat(driver.findElement(By.id("video")).getTagName(),
            equalTo("video"));
    }

    @Test
    void extensionTest(@Extensions("hello_world.crx") ChromeDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}

```

As introduced before, this annotation `@Options` can be used also at *field-level*, as shown in this other [example](#). This test is setting to `true` the Firefox preferences `media.navigator.streams.fake` and `media.navigator.permission.disabled`, used also for WebRTC.

```

import static org.hamcrest.CoreMatchers.equalTo;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.By;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.firefox.FirefoxOptions;

import io.github.bonigarcia.seljup.Options;
import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class FirefoxWithGlobalOptionsJupiterTest {

    @Options
    FirefoxOptions firefoxOptions = new FirefoxOptions();
    {
        // Flag to use fake media for WebRTC user media
        firefoxOptions.addPreference("media.navigator.streams.fake", true);

        // Flag to avoid granting access to user media
        firefoxOptions.addPreference("media.navigator.permission.disabled",
            true);
    }

    @Test
    public void webrtcTest(FirefoxDriver driver) {
        driver.get(
            "https://webrtc.github.io/samples/src/content/devices/input-output/");
        assertThat(driver.findElement(By.id("video")).getTagName(),
            equalTo("video"));
    }
}

```

Template tests

Selenium-Jupiter takes advantage on the standard feature of JUnit 5 called **test templates**. Test templates can be seen as an special kind of parameterized tests, in which the test is executed several times according to the data provided by some extension. In our case, the extension is *Selenium-Jupiter* itself, and the test template is configured using a custom file in JSON called **browsers scenario**.

Let's see some examples. Consider the following test. A couple of things are new in this test. First of all, instead of declaring the method with the usual **@Test** annotation, we are using the JUnit 5's annotation **@TestTemplate**. With this we are saying to JUnit that this method is not a regular test case but a template. Second, the parameter type of the method **templateTest** is **WebDriver**. This is the

generic interface of Selenium WebDriver, and the concise type (i.e. `ChromeDriver`, `FirefoxDriver`, `RemoteWebDriver`, etc.) will be determined by *Selenium-Jupiter* in runtime.

```
import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.TestTemplate;
import org.junit.jupiter.api.extension.RegisterExtension;
import org.openqa.selenium.WebDriver;

import io.github.bonigarcia.seljup.SeleniumExtension;

public class TemplateTest {

    @RegisterExtension
    static SeleniumExtension seleniumExtension = new SeleniumExtension();

    @TestTemplate
    void templateTest(WebDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}
```

The last piece we need in this test template is what we call *browser scenario*. As introduced before, this scenario is defined in a JSON file following a simple notation.

The path of the JSON browser scenario is established in the configuration key called `sel.jup.browser.template.json.file`. By default, this key has the value `classpath:browsers.json`. This means that the JSON scenario is defined in a file called `browsers.json` located in the classpath (see section [Configuration](#) for further details about configuration).

NOTE	If the configuration key <code>sel.jup.browser.template.json.file</code> do not start with the word <code>classpath:</code> , the file will be searched using relative of absolute paths.
-------------	---

Now imagine that the content of the file `browsers.json` is as follows:

```
{
  "browsers": [
    [
      {
        "type": "chrome-in-docker",
        "version": "latest"
      }
    ],
    [
      {
        "type": "chrome-in-docker",
        "version": "latest-1"
      }
    ],
    [
      {
        "type": "chrome-in-docker",
        "version": "beta"
      }
    ],
    [
      {
        "type": "chrome-in-docker",
        "version": "unstable"
      }
    ]
  ]
}
```

When we execute the template test, in this case we will have four actual tests: the first using the *latest* version of Chrome, the second using the previous to stable version of Chrome (*latest-1*), the third using the beta version of Chrome (*beta*), and another test using the development version of Chrome (*unstable*). For instance, if we run the test in Eclipse, we will get the following output:

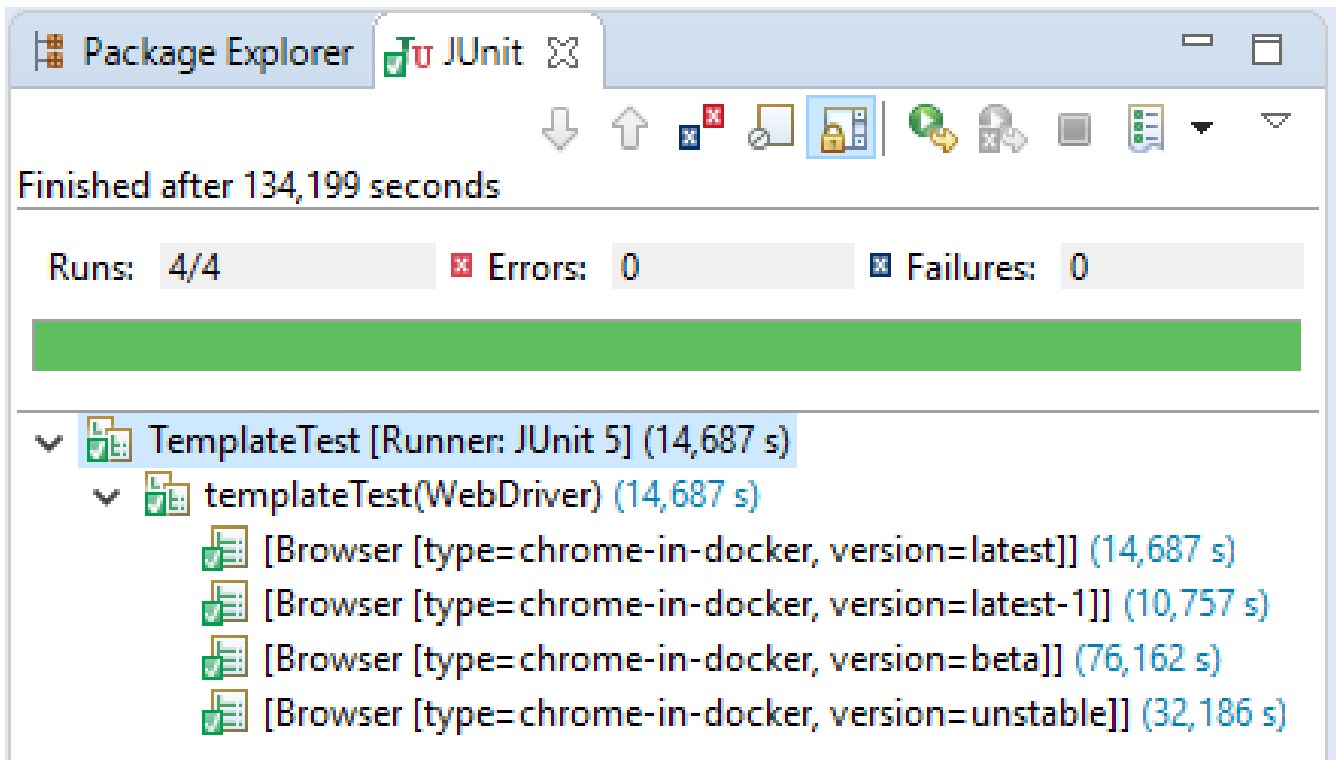


Figure 7. Example of test template execution in Eclipse

Generally speaking, a browser within the JSON scenario is defined using the following parameters:

- **type**: Type of browsers. The accepted values are:
 - **chrome**: For local Chrome browsers.
 - **firefox**: For local Firefox browsers.
 - **edge**: For local Edge browsers.
 - **ieexplorer**: For local Internet Explorer browsers.
 - **opera**: For local Opera browsers.
 - **safari**: For local Safari browsers.
 - **appium**: For local mobile emulated devices.
 - **phantomjs**: For local PhtanomJS headless browsers.
 - **chrome-in-docker**: For Chrome browsers in Docker.
 - **firefox-in-docker**: For Firefox browsers in Docker.
 - **opera-in-docker**: For Opera browsers in Docker.
 - **android**: For web browsers in Android devices in Docker containers.
- **version**: Optional value for the version. Wildcard for latest versions (**latest**, **latest-1**, etc) are accepted. Concrete versions are also valid (e.g. **63.0**, **58.0**, etc., depending of the browser). **Beta** and **unstable** (i.e. development) versions for Chrome and Firefox are also supported (using the labels **beta** and **unstable** labels respectively).
- **deviceName**: Also for **android** type, the device type can be specified (Samsung Galaxy S6, Nexus 4, Nexus 5, etc.).

Finally, more than one parameters can be defined in the test template. For instance, consider the

following test in which a couple of `WebDriver` parameters are declared in the test template.

```
import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.TestTemplate;
import org.junit.jupiter.api.extension.RegisterExtension;
import org.openqa.selenium.WebDriver;

import io.github.bonigarcia.seljup.SeleniumExtension;

public class TemplateTwoBrowsersTest {

    @RegisterExtension
    static SeleniumExtension seleniumExtension = new SeleniumExtension();

    @TestTemplate
    void templateTest(WebDriver driver1, WebDriver driver2) {
        driver1.get("https://bonigarcia.github.io/selenium-jupiter/");
        driver2.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver1.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
        assertThat(driver2.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}
```

The JSON scenario should be defined accordingly. Each browser array in this case (for each test template execution) should declare two browsers. For instance, using the following JSON scenario, the first execution will be based on Chrome in Docker (first parameter) and Firefox in Docker (second parameter); and the second execution will be based on a local Chrome (first parameter) and the headless browser PhantomJS (second parameter).

```

{
  "browsers": [
    [
      {
        "type": "chrome-in-docker"
      },
      {
        "type": "firefox-in-docker"
      }
    ],
    [
      {
        "type": "chrome"
      },
      {
        "type": "phantomjs"
      }
    ]
  ]
}

```

If we execute this test using in GUI, the JUnit tab shows two tests executed with the values defined in the JSON scenario.

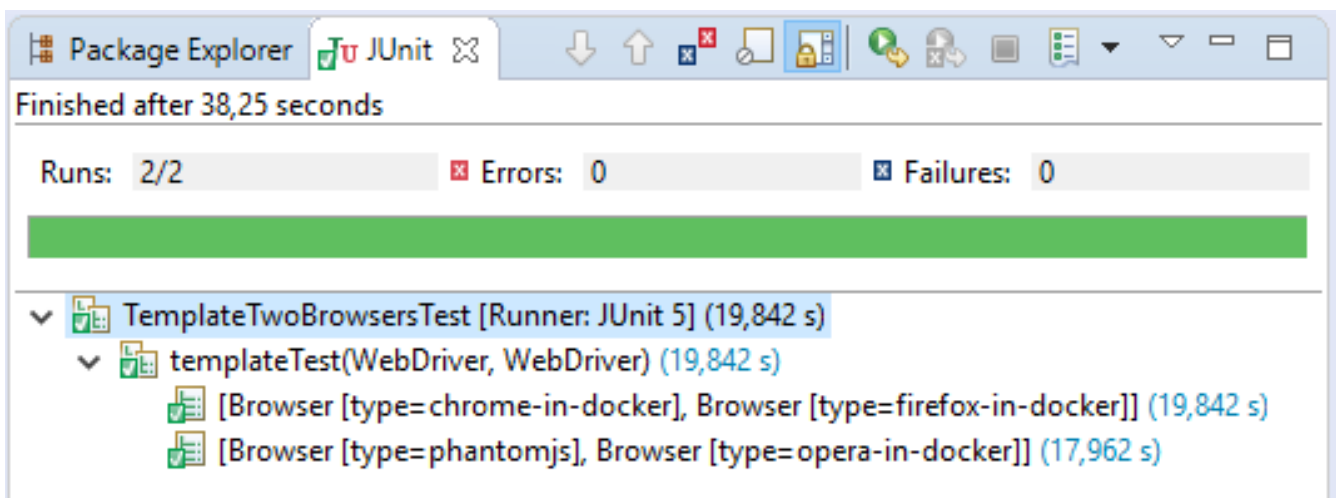


Figure 8. Example of test template execution (with two parameters) in Eclipse

As of version 2.2.0, *Selenium-Jupiter* allows to configure the browser scenario programmatically using the JUnit 5 `@RegisterExtension` annotation. To that aim, the method `addBrowsers` of the `SeleniumExtension` instance is used to add different browser(s) to the scenario. In the following example the test is executed twice, one using Chrome and the second using Firefox.


```

import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.TestTemplate;
import org.junit.jupiter.api.extension.RegisterExtension;
import org.openqa.selenium.WebDriver;

import io.github.bonigarcia.seljup.BrowserBuilder;
import io.github.bonigarcia.seljup.BrowsersTemplate.Browser;
import io.github.bonigarcia.seljup.SeleniumExtension;

public class TemplateRegisterTest {

    @RegisterExtension
    static SeleniumExtension seleniumExtension = new SeleniumExtension();

    @BeforeAll
    static void setup() {
        Browser chrome = BrowserBuilder.chrome().build();
        Browser firefox = BrowserBuilder.firefox().build();
        seleniumExtension.addBrowsers(chrome);
        seleniumExtension.addBrowsers(firefox);
    }

    @TestTemplate
    void templateTest(WebDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}

```

Generic driver

As of version 2.1.0, *Selenium-Jupiter* allows to use a configurable WebDriver object. This generic driver is declared as usual (i.e. as test method or constructor parameter) using the type `RemoteWebDriver` or `WebDriver`. The concrete type of browser to be used is established using the configuration key `sel.jup.default.browser`. The default value for this key is `chrome-in-docker`. All the values used in the template test defined in the previous section (i.e. `chrome`, `firefox`, `edge`, `chrome-in-docker`, `firefox-in-docker`, `android`, etc.) can be used also to define the type of browser in this mode.

For instance, the following test, if no additional configuration is done, will use Chrome in Docker as browser:

```

import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;

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

import io.github.bonigarcia.seljup.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class GenericTest {

    @Test
    void genericTest(WebDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}

```

If the resolution of this browser finishes with exception (for instance, when executing the test in a host without Docker), a list of browser fallback will be used. This list is managed using the configuration key `sel.jup.default.browser.fallback`. By default, this key has the value `chrome,firefox,safari,edge,phantomjs`, meaning that the first fallback browser is a local Chrome, then local Firefox, then local Safari, then local Edge, and finally PhantomJS (headless browser).

The version of the generic browser (in case of Docker browsers) is managed with the key `sel.jup.default.version` (`latest` by default). The versions of the fallback browsers can be also managed, this time using the configuration key `sel.jup.default.browser.fallback.version`.

Integration with Jenkins

Selenium-Jupiter provides seamless integration with Jenkins through one of its plugins: the [Jenkins attachment plugin](#). The idea is to provide the ability to attach output files (typically PNG screenshots and MP4 recordings of Docker browsers), and keep these files attached to the job execution. This is done in *Selenium-Jupiter* setting the configuration key `sel.jup.output.folder` to a special value: `surefire-reports`.

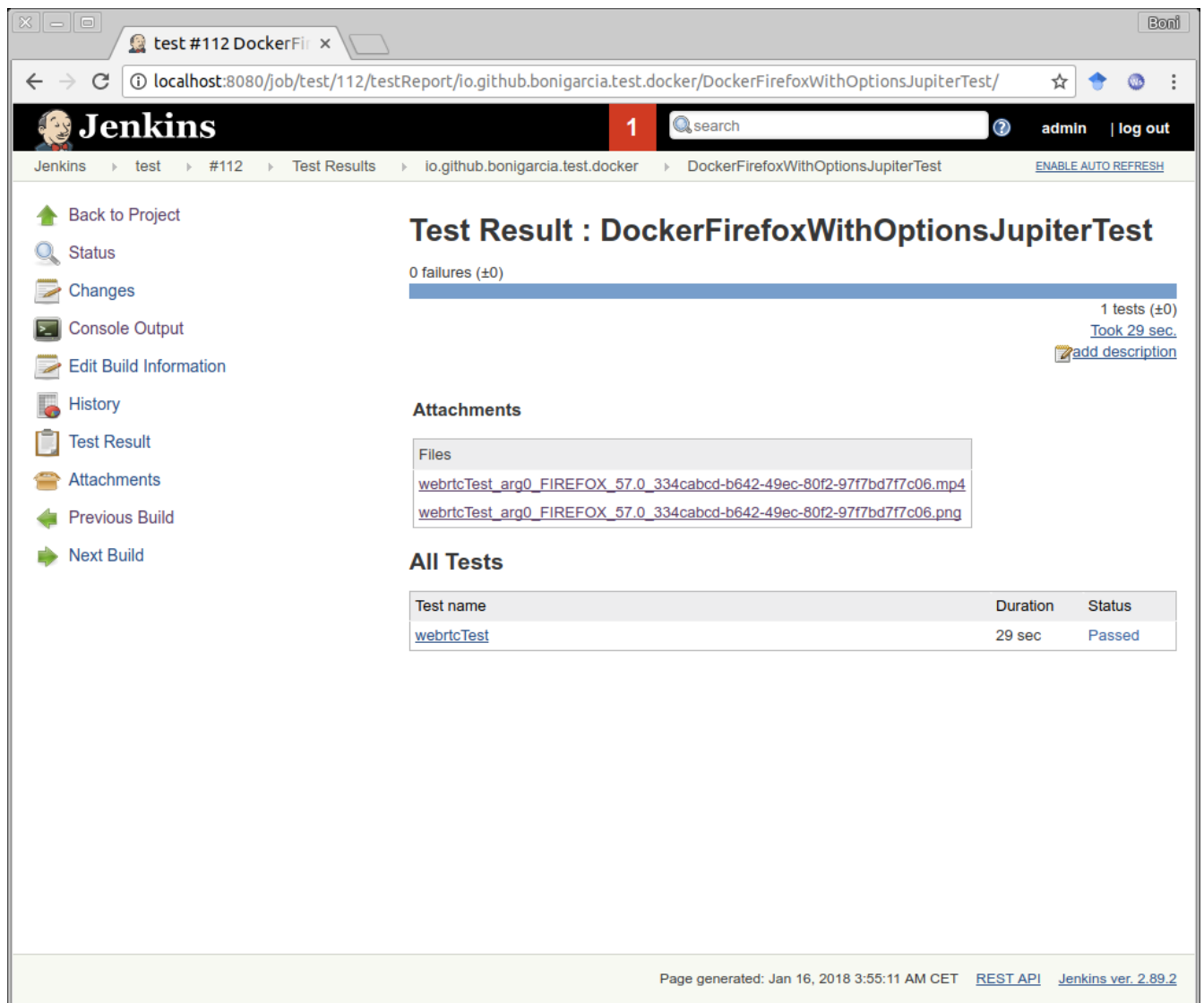
When this configuration key is configured with that value, *Selenium-Jupiter* will store the generated files in the proper folder, in a way that the Jenkins attachment plugin is able to find those files and export them in the Jenkins GUI. For instance, consider the following [test](#), when is executed in Jenkins (with the attachment plugin) and the following configuration:

```

mvn clean test -Dtest=DockerFirefoxWithOptionsJupiterTest -Dsel.jup.recording=true
-Dsel.jup.output.folder=surefire-reports -Dsel.jup.screenshot.at.the.end.of.tests=true

```

In this case, at the the execution of this test, two recordings in MP4 and two screenshots in PNG will be attached to the job as follows.



The screenshot shows the Jenkins web interface. The browser address bar indicates the URL: `localhost:8080/job/test/112/testReport/io.github.bonigarcia.test.docker/DockerFirefoxWithOptionsJupiterTest/`. The Jenkins logo and navigation menu are on the left. The main content area displays the test results for `DockerFirefoxWithOptionsJupiterTest`. It shows 0 failures and 1 test passed, taking 29 seconds. Two attachments are listed: `webrtcTest_arg0_FIREFOX_57.0_334cabcd-b642-49ec-80f2-97f7bd7f7c06.mp4` and `webrtcTest_arg0_FIREFOX_57.0_334cabcd-b642-49ec-80f2-97f7bd7f7c06.png`. Below the attachments, a table lists the test results.

Test name	Duration	Status
webrtcTest	29 sec	Passed

Page generated: Jan 16, 2018 3:55:11 AM CET [REST API](#) [Jenkins ver. 2.89.2](#)

Figure 9. Example of test execution through Jenkins with attachments

We can watch the recording simply clicking in the attached MP4 files.

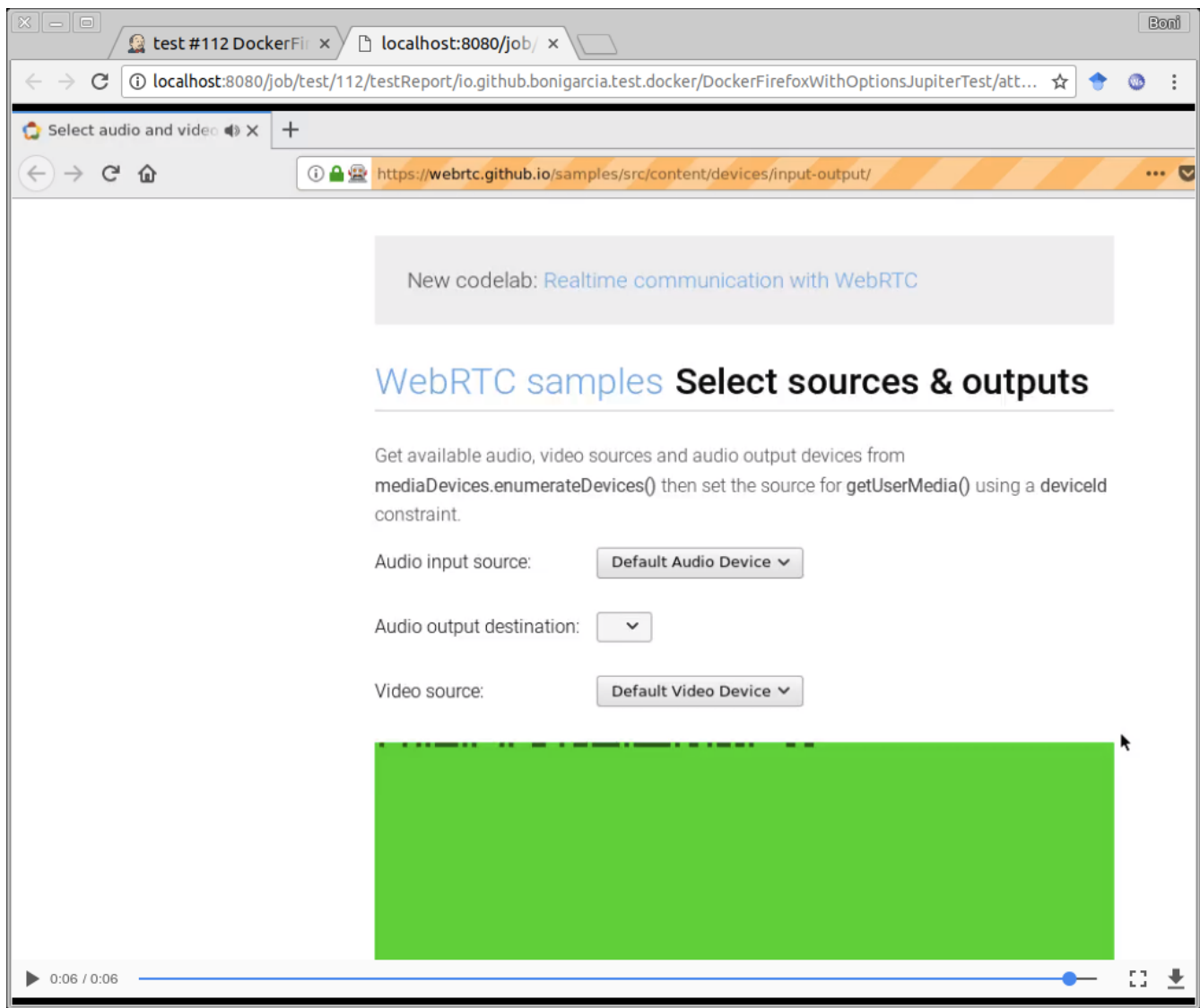


Figure 10. Example of test execution through Jenkins with attachments

Test template are also compatible with this feature. For instance, consider the following test [test](#). When is executed in Jenkins using the configuration below, the following attachments will be available on Jenkins:

```
mvn clean test -Dtest=TemplateTest -Dsel.jup.recording=true  
-Dsel.jup.output.folder=surefire-reports -Dsel.jup.screenshot.at.the.end.of.tests=true
```

The screenshot shows the Jenkins web interface for a test result. The browser address bar shows the URL: `localhost:8080/job/test/114/testReport/(root)/templateTest(WebDriver)/`. The Jenkins logo and navigation menu are at the top. The main heading is "Test Result : templateTest(WebDriver)". Below this, it says "0 failures (-1)" and "3 tests (+2) Took 21 sec." with a link to "add description".

Under the "Attachments" section, there is a list of files for each test run:

- templateTest_arg0_CHROME_61.0_883eb91eae06ea1e92511169d9861593.mp4
- templateTest_arg0_CHROME_61.0_883eb91eae06ea1e92511169d9861593.png
- templateTest_arg0_CHROME_62.0_14ea838640ca14392895629a31975b59.mp4
- templateTest_arg0_CHROME_62.0_14ea838640ca14392895629a31975b59.png
- templateTest_arg0_CHROME_63.0_bacaef324a9c977dd103f67678c6638f.mp4
- templateTest_arg0_CHROME_63.0_bacaef324a9c977dd103f67678c6638f.png

Below the attachments, the "All Tests" section contains a table:

Test name	Duration	Status
templateTest	6.4 sec	Fixed
templateTest	6.2 sec	Fixed
templateTest	9.1 sec	Fixed

At the bottom, it says "Page generated: Jan 16, 2018 4:01:01 AM CET" with links to "REST API" and "Jenkins ver. 2.89.2".

Figure 11. Example of template test execution through Jenkins with attachments

And we will be able to watch the recording:

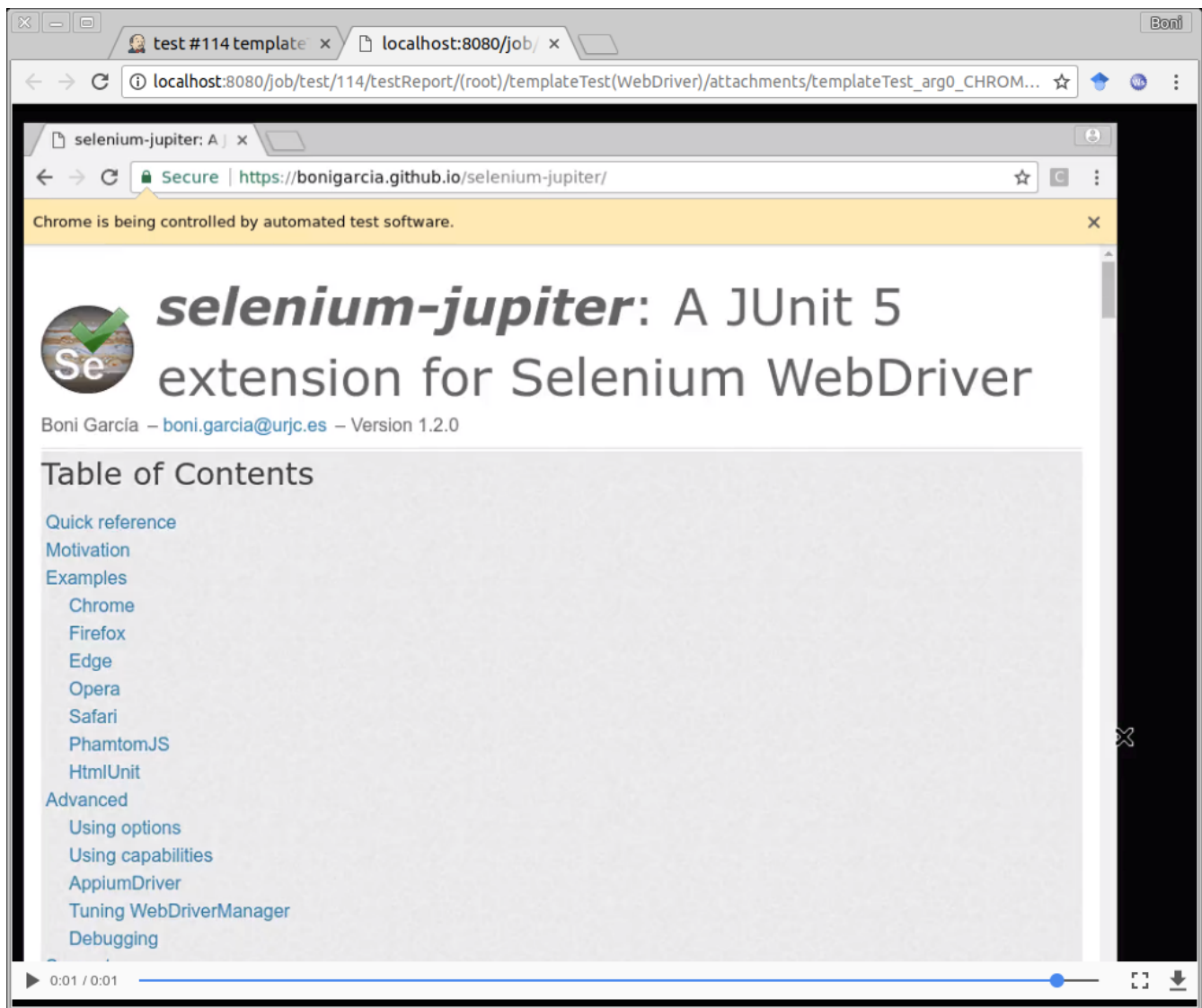


Figure 12. Example of template test execution through Jenkins with attachments

Using Genymotion

The use of [Genymotion](#) for using Android devices in the cloud is possible as of Selenium-Jupiter 3.1.0. This feature is used using a Software as a Service (SaaS) approach. In order to use this feature, we need to select `GENYMOTION_SAAS` in the `cloud` parameter of `@DockerBrowser` in a test. First, we need a valid Genymotion account. This account will be configured using the methods `setAndroidGenymotionUser()`, `setAndroidGenymotionPassword()` and `setAndroidGenymotionLicense()` for user, password, and license respectively. Moreover, we need to specify the following values:

- Genymotion template name (method `setAndroidGenymotionTemplate()`).
- Android version (method `setAndroidGenymotionAndroidVersion()`).
- Android API level (method `setAndroidGenymotionAndroidApi()`).
- Android screen size (method `setAndroidGenymotionScreenSize()`).

All this values will be internally managed using a label specified with `setAndroidGenymotionDeviceName()`. This label will be used later in the actual test in the `deviceName` value. The following test shows an example:

```

import static io.github.bonigarcia.seljup.BrowserType.ANDROID;
import static io.github.bonigarcia.seljup.CloudType.GENYMOTION_SAAS;
import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.RegisterExtension;
import org.openqa.selenium.remote.RemoteWebDriver;

import io.github.bonigarcia.seljup.DockerBrowser;
import io.github.bonigarcia.seljup.SeleniumExtension;
import io.github.bonigarcia.seljup.config.Config;

public class AndroidGenymotionJupiterTest {

    @RegisterExtension
    static SeleniumExtension seleniumExtension = new SeleniumExtension();

    @BeforeAll
    static void setup() {
        Config config = seleniumExtension.getConfig();
        config.setAndroidGenymotionDeviceName("SamsungS7V6");
        config.setAndroidGenymotionTemplate("Samsung Galaxy S7");
        config.setAndroidGenymotionAndroidVersion("6.0.0");
        config.setAndroidGenymotionAndroidApi("23");
        config.setAndroidGenymotionScreenSize("1440x2560");

        // The following values need to be set (it can be overridden using Java
        // properties or environment variables)
        config.setAndroidGenymotionUser("my-genymotion-user");
        config.setAndroidGenymotionPassword("my-genymotion-pass");
        config.setAndroidGenymotionLicense("my-genymotion-license");
    }

    @Test
    public void testAndroidInGenymotionSaas(
        @DockerBrowser(type = ANDROID, cloud = GENYMOTION_SAAS, deviceName =
        "SamsungS7V6", browserName = "browser")
        RemoteWebDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}

```

We can see the execution of the test in the [Genymotion dashboard](#) while the test is being executed. For the example before, we see the following Android device:

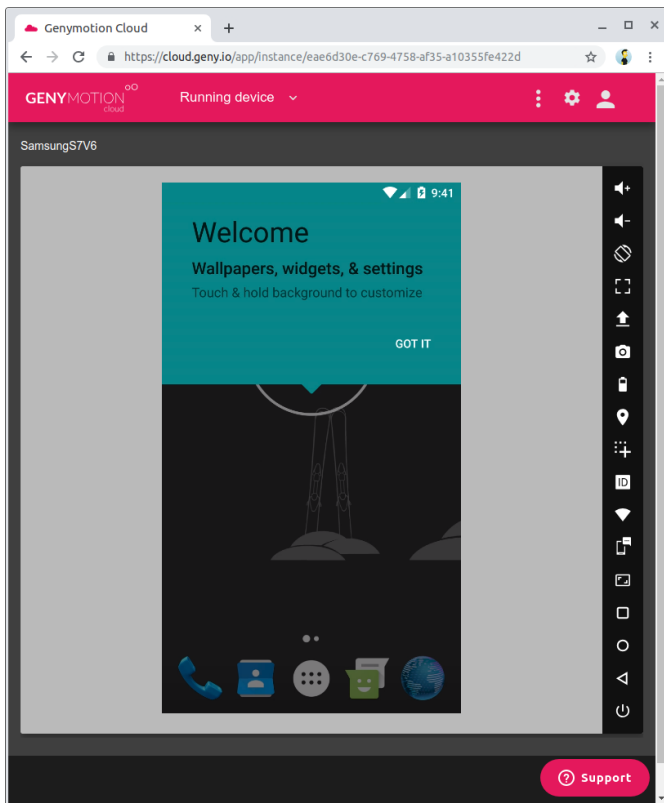


Figure 13. Example of test execution in Genymotion

Notice that *Selenium-Jupiter* downloads the *chromedriver* binary required by internal Appium server contained in the internal Docker container which is used to connect with Genymotion SaaS. By default, the version of the *chromedriver* binary is related with the installed browser. The following table shows a summary of the *chromedriver* versions managed by *Selenium-Jupiter* depending on the Android API level. If other version of *chromedriver* is required, it can be specified using the method `setAndroidGenymotionChromedriver()`.

Table 3. Versions of *chromedriver* handled by *Selenium-Jupiter* for Genymotion

Android API level	<i>chromedriver</i> version
21	2.21
22	2.13
23	2.18
24	2.23
25	2.28
26	2.31
27	2.33
28	2.40

Single session

By default, the instances provided by *Selenium-Jupiter* (e.g. `ChromeDriver`, `FirefoxDriver`, etc) are created **before each** test, and are disposed **after each** test. As of *Selenium-Jupiter* 3.3.0, this default behavior can be changed using the class-level annotation `@SingleSession`. The instances provided in this case will be created **before all** tests, and are disposed **after all** tests.

The following test shows an example of this feature. As you can see, this test uses the ordering capability provided as of JUnit 5.4. The browser (Chrome in this case) is available at the beginning for the tests. According to the given order, first `testStep1()` is executed. Then, the session (i.e. the same browser in the same state) is used by the second test, `testStep2()`. At the end of all tests, the browser is closed.

```
import static java.lang.invoke.MethodHandles.lookup;
import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.slf4j.LoggerFactory.getLogger;

import org.junit.jupiter.api.MethodOrderer.OrderAnnotation;
import org.junit.jupiter.api.Order;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestMethodOrder;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.remote.RemoteWebDriver;
import org.slf4j.Logger;

import io.github.bonigarcia.seljup.SeleniumExtension;
import io.github.bonigarcia.seljup.SingleSession;

@ExtendWith(SeleniumExtension.class)
@TestMethodOrder(OrderAnnotation.class)
@SingleSession
public class OrderedJupiterTest {

    final Logger log = getLogger(lookup().lookupClass());

    RemoteWebDriver driver;

    public OrderedJupiterTest(ChromeDriver driver) {
        this.driver = driver;
    }

    @Test
    @Order(1)
    public void testStep1() {
        log.debug("Step 1: {}", driver);
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }

    @Test
    @Order(2)
    public void testStep2() {
```

```
        log.debug("Step 2: {}", driver);
        WebElement about = driver.findElementByLinkText("About");
        assertTrue(about.isDisplayed());
        about.click();
    }
}
```

Selenium-Jupiter CLI

As of version 2.1.0, *Selenium-Jupiter* can be used interactively from the shell as a regular command line interface (CLI) tool. In this mode, *Selenium-Jupiter* allows to get the VNC session of Docker browser. There are two ways of using this feature:

- Directly from the source code, using Maven. The command to be used is `mvn exec:java -Dexec.args="browserName <version>"`. For instance:

```

$ mvn exec:java -Dexec.args="chrome beta"
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building selenium-jupiter 3.3.2
[INFO] -----
[INFO] --- exec-maven-plugin:1.6.0:java (default-cli) @ selenium-jupiter ---
[INFO] Using Selenium-Jupiter to execute chrome beta in Docker
[INFO] Using CHROME version beta
[INFO] Starting Docker container aerokube/selenoid:1.8.4
[DEBUG] Creating WebDriver for CHROME at http://172.17.0.1:32911/wd/hub
Jan 24, 2019 1:09:04 AM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: OSS
[INFO] Starting Docker container psharkey/novnc:3.3-t6
[INFO] Session id ee7200775c466c8526c77e7eb2495557
[INFO] VNC URL (copy and paste in a browser navigation bar to interact with remote
session)
[INFO]
http://172.17.0.1:32912/vnc.html?host=172.17.0.1&port=32911&path=vnc/ee7200775c466c852
6c77e7eb2495557&resize=scale&autoconnect=true&password=selenoid
[INFO] Press ENTER to exit

[INFO] Stopping Docker container aerokube/selenoid:1.8.4
[INFO] Stopping Docker container psharkey/novnc:3.3-t6
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 01:10 min
[INFO] Finished at: 2018-03-31T17:51:15+02:00
[INFO] Final Memory: 27M/390M
[INFO] -----

```

- Using *Selenium-Jupiter* as a [fat-jar](#). This jar can be created using the command `mvn compile assembly:single` from the source code, and then `java -jar selenium-jupiter.jar browserName <version>`. For instance:

```
$ java -jar selenium-jupiter-3.3.2-fat.jar firefox
[INFO] Using Selenium-Jupiter to execute firefox (latest) in Docker
[INFO] Using FIREFOX version 66.0 (latest)
[INFO] Pulling Docker image aerokube/selenoid:1.8.4
[INFO] Starting Docker container aerokube/selenoid:1.8.4
[DEBUG] Creating WebDriver for FIREFOX at http://172.17.0.1:32909/wd/hub
Jan 24, 2019 1:08:15 AM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: W3C
[INFO] Starting Docker container psharkey/novnc:3.3-t6
[INFO] Session id 2356ceb4-53f6-49d5-bae0-3072faa47ad2
[INFO] VNC URL (copy and paste in a browser navigation bar to interact with remote session)
[INFO] http://172.17.0.1:32910/vnc.html?host=172.17.0.1&port=32909&path=vnc/2356ceb4-53f6-49d5-bae0-3072faa47ad2&resize=scale&autoconnect=true&password=selenoid
[INFO] Press ENTER to exit

[INFO] Stopping Docker container aerokube/selenoid:1.8.4
[INFO] Stopping Docker container psharkey/novnc:3.3-t6
```

NOTE

As of version 2.2.0, the parameter `browserName` can be used to select an `android` device. In this case, an addition parameter can be specified: `deviceName` for the device type (Samsung Galaxy S6, Nexus 4, Nexus 5, etc.).

Selenium-Jupiter Server

As of version 3.0.0, Selenium-Jupiter can be used as a server. To start this mode, the shell is used. Once again, two options are allowed:

- Directly from the source code and Maven. The command to be used is `mvn exec:java -Dexec.args="server <port>"`. If the second argument is not specified, the default port will be used (4042):

```
$ mvn exec:java -Dexec.args="server"
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building Selenium-Jupiter 3.3.2
[INFO] -----
[INFO]
[INFO] --- exec-maven-plugin:1.6.0:java (default-cli) @ selenium-jupiter ---
[INFO] Selenium-Jupiter server listening on http://localhost:4042/wd/hub
```

- Using Selenium-Jupiter as a `fat-jar`. For instance:

```
> java -jar webdrivermanager-3.3.2-fat.jar server
[INFO] Selenium-Jupiter server listening on http://localhost:4042/wd/hub
```

When the Selenium-Jupiter server is up and running, it acts as a regular Selenium Server for Docker browsers (Chrome, Firefox, Opera, Android), and its URL can be used in tests using regular Selenium's `RemoteWebDriver` objects.

Configuration

Default configuration parameters for *Selenium-Jupiter* are set in the `selenium-jupiter.properties` file. The following table summarizes all the configuration keys available.

Table 4. Configuration keys in Selenium-Jupiter

Configuration key	Description	Default value
<code>sel.jup.vnc</code>	Check VNC session for Docker browsers	<code>false</code>
<code>sel.jup.vnc.screen.resolution</code>	Screen resolution of VNC sessions (format <code><width>x<height>x<colors-depth></code>)	<code>1920x1080x24</code>
<code>sel.jup.vnc.create.redirect.html.page</code>	Redirect VNC URL to HTML page	<code>false</code>
<code>sel.jup.vnc.export</code>	Java property name in which the VNC URL will be exported	<code>vnc.session.url</code>
<code>sel.jup.recording</code>	Record Docker browser session (in MP4 format)	<code>false</code>
<code>sel.jup.recording.video.screen.size</code>	Video screen size for recordings (width and height)	<code>1024x768</code>
<code>sel.jup.recording.video.frame.rate</code>	Video frame rate for recordings	<code>12</code>
<code>sel.jup.recording.image</code>	Docker image for recordings	<code>selenoid/video-recorder:latest</code>
<code>sel.jup.output.folder</code>	Output folder for recordings, screenshots, and HTML redirect pages	<code>.</code>
<code>sel.jup.screenshot.at.the.end.of.tests</code>	Make screenshots at the end of the test	<code>false</code>
<code>sel.jup.screenshot.format</code>	Format for screenshots	<code>base64</code>
<code>sel.jup.exception.when.no.driver</code>	Throw exception in case of exception or not	<code>true</code>
<code>sel.jup.browser.template.json.file</code>	Browsers scenario (JSON) path	<code>classpath:browsers.json</code>
<code>sel.jup.browser.template.json.content</code>	Browsers scenario (JSON) content	
<code>sel.jup.default.browser</code>	Browser for generic driver	<code>chrome-in-docker</code>
<code>sel.jup.default.version</code>	Version for generic driver	<code>latest</code>
<code>sel.jup.default.browser.fallback</code>	Fallback browser list for generic driver	<code>chrome,firefox,safari,edge,phantomjs</code>

Configuration key	Description	Default value
<code>sel.jup.default.browser.fallback.version</code>	Fallback version list for generic driver	<code>latest,latest,latest,latest,latest</code>
<code>sel.jup.remote.webdriver.wait.timeout.sec</code>	Time in seconds to wait the creating of RemoteWebDriver objects	<code>20</code>
<code>sel.jup.remote.webdriver.poll.time.sec</code>	Poll time in seconds to wait while creating RemoteWebDriver objects	<code>2</code>
<code>sel.jup.wdm.use.preferences</code>	As of Selenium-Jupiter, the latest version of Docker browsers are stored persistently as Java preferences. Its use can be deactivated using this flag	<code>true</code>
<code>sel.jup.ttl.sec</code>	The resolved Docker browsers versions has attached a time-to-live in seconds, configurable using this key	<code>86400</code>
<code>sel.jup.browser.list.from.docker.hub</code>	Update Docker images list from Docker Hub	<code>true</code>
<code>sel.jup.browser.session.timeout.duration</code>	Session timeout for Docker browsers (in Golang duration format)	<code>1m0s</code>
<code>sel.jup.selenium.image</code>	Selenium (Golang Selenium Hub) Docker image	<code>aerokube/selenium:1.9.3</code>
<code>sel.jup.selenium.port</code>	Selenium port	<code>4444</code>
<code>sel.jup.selenium.vnc.password</code>	VNC password for Selenium sessions	<code>selenium</code>
<code>sel.jup.novnc.image</code>	noVNC Docker image	<code>psharkey/novnc:3.3-t6</code>
<code>sel.jup.novnc.port</code>	noVNC Docker port	<code>8080</code>
<code>sel.jup.chrome.image.format</code>	Selenium Docker images format for Chrome with VNC	<code>selenium/vnc:chrome_%s</code>
<code>sel.jup.chrome.first.version</code>	First version of Docker Chrome (used when <code>sel.jup.browser.list.from.docker.hub =false</code>)	<code>48.0</code>
<code>sel.jup.chrome.latest.version</code>	Latest version of Docker Chrome (used when <code>sel.jup.browser.list.from.docker.hub =false</code>)	<code>78.0</code>
<code>sel.jup.chrome.path</code>	Path for Hub when using Chrome in Docker as browser	<code>/</code>
<code>sel.jup.chrome.beta.image</code>	Selenium Docker image format for Chrome beta	<code>elastestbrowsers/chrome:beta</code>

Configuration key	Description	Default value
<code>sel.jup.chrome.beta.path</code>	Path for Hub when using Chrome beta in Docker as browser	<code>/wd/hub</code>
<code>sel.jup.chrome.unstable.image</code>	Selenium Docker image format for Chrome unstable	<code>elastestbrowsers/chrome:unstable</code>
<code>sel.jup.chrome.unstable.path</code>	Path for Hub when using Chrome unstable in Docker as browser	<code>/wd/hub</code>
<code>sel.jup.firefox.image.format</code>	Selenium Docker images format for Firefox with VNC	<code>selenium/vnc:firefox_%s</code>
<code>sel.jup.firefox.first.version</code>	First version of Docker Firefox (used when <code>sel.jup.browser.list.from.docker.hub = false</code>)	<code>3.6</code>
<code>sel.jup.firefox.latest.version</code>	Latest version of Docker Firefox (used when <code>sel.jup.browser.list.from.docker.hub = false</code>)	<code>70.0</code>
<code>sel.jup.firefox.path</code>	Path for Hub when using Firefox in Docker as browser	<code>/wd/hub</code>
<code>sel.jup.firefox.beta.image</code>	Selenium Docker image format for Firefox beta	<code>elastestbrowsers/firefox:beta</code>
<code>sel.jup.firefox.beta.path</code>	Path for Hub when using Firefox beta in Docker as browser	<code>/wd/hub</code>
<code>sel.jup.firefox.unstable.image</code>	Selenium Docker image format for Firefox unstable	<code>elastestbrowsers/firefox:nightly</code>
<code>sel.jup.firefox.unstable.path</code>	Path for Hub when using Firefox beta in Docker as unstable	<code>/wd/hub</code>
<code>sel.jup.opera.image.format</code>	Selenium Docker images format for Opera with VNC	<code>selenium/vnc:opera_%s</code>
<code>sel.jup.opera.first.version</code>	First version of Docker Opera (used when <code>sel.jup.browser.list.from.docker.hub = false</code>)	<code>33.0</code>
<code>sel.jup.opera.latest.version</code>	Latest version of Docker Opera (used when <code>sel.jup.browser.list.from.docker.hub = false</code>)	<code>64.0</code>
<code>sel.jup.opera.path</code>	Path for Hub when using Opera in Docker as browser	<code>/</code>
<code>sel.jup.edge.image</code>	Docker image format for Edge	<code>windows/edge:%s</code>
<code>sel.jup.edge.latest.version</code>	Latest version of Docker Edge	<code>18</code>

Configuration key	Description	Default value
<code>sel.jup.edge.path</code>	Path for Hub when using Opera in Docker as browser	<code>/</code>
<code>sel.jup.iexplorer.image</code>	Docker image format for Edge	<code>windows/ie:%s</code>
<code>sel.jup.iexplorer.latest.version</code>	Latest version of Docker Edge	<code>11</code>
<code>sel.jup.iexplorer.path</code>	Path for Hub when using Opera in Docker as browser	<code>/</code>
<code>sel.jup.android.default.version</code>	Default version of Android devices in Docker	<code>9.0</code>
<code>sel.jup.android.image.5.0.1</code>	Docker image for version 5.0.1 of Android devices	<code>butomo1989/docker-android-x86-5.0.1:1.5-p6</code>
<code>sel.jup.android.image.5.1.1</code>	Docker image for version 5.1.1 of Android devices	<code>butomo1989/docker-android-x86-5.1.1:1.5-p6</code>
<code>sel.jup.android.image.6.0</code>	Docker image for version 6.0 of Android devices	<code>butomo1989/docker-android-x86-6.0:1.5-p6</code>
<code>sel.jup.android.image.7.0</code>	Docker image for version 7.0 of Android devices	<code>butomo1989/docker-android-x86-7.0:1.5-p6</code>
<code>sel.jup.android.image.7.1.1</code>	Docker image for version 7.1.1 of Android devices	<code>butomo1989/docker-android-x86-7.1.1:1.5-p6</code>
<code>sel.jup.android.image.8.0</code>	Docker image for version 8.0 of Android devices	<code>butomo1989/docker-android-x86-8.0:1.5-p6</code>
<code>sel.jup.android.image.8.1</code>	Docker image for version 8.1 of Android devices	<code>butomo1989/docker-android-x86-8.1:1.5-p6</code>
<code>sel.jup.android.image.9.0</code>	Docker image for version 9.0 of Android devices	<code>butomo1989/docker-android-x86-9.0:1.5-p6</code>
<code>sel.jup.android.image.genymotion</code>	Docker image for Genymotion usage	<code>budtmo/docker-android-genymotion:1.7-p0</code>
<code>sel.jup.android.genymotion.user</code>	Genymotion SaaS user	
<code>sel.jup.android.genymotion.password</code>	Genymotion SaaS password	
<code>sel.jup.android.genymotion.license</code>	Genymotion SaaS license	
<code>sel.jup.android.genymotion.template</code>	Genymotion SaaS template	
<code>sel.jup.android.genymotion.device.name</code>	Genymotion SaaS device	
<code>sel.jup.android.genymotion.android.version</code>	Genymotion SaaS Android version	
<code>sel.jup.android.genymotion.android.api</code>	Genymotion SaaS Android API level	
<code>sel.jup.android.genymotion.screen.size</code>	Genymotion SaaS Android screen size	

Configuration key	Description	Default value
<code>sel.jup.android.genymotion.chromedriver</code>	Genymotion SaaS chromedriver version	
<code>sel.jup.android.novnc.port</code>	Internal port of noVNC in Docker containers for Android devices	6080
<code>sel.jup.android.appium.port</code>	Internal port of Appium server in Docker containers for Android devices	4723
<code>sel.jup.android.device.name</code>	Default device name for Android in Docker	Samsung Galaxy S6
<code>sel.jup.android.browser.name</code>	Default browser name for Android in Docker	chrome
<code>sel.jup.android.device.timeout.sec</code>	Timeout (in seconds) to wait Android devices to be available	200
<code>sel.jup.android.device.startup.timeout.sec</code>	Amount of time that should pass between start of container and the first attempt to connect to Appium in container	75
<code>sel.jup.android.appium.ping.period.sec</code>	Amount of time that should pass after failed attempt to initialize Appium before the next attempt	10
<code>sel.jup.android.logging</code>	If true Android and Appium log files will be written into <code>sel.jup.android.logs.folder</code> directory	false
<code>sel.jup.android.logs.folder</code>	Folder with Android logs relative to <code>sel.jup.output.folder</code> . It contains files such as <i>appium.log</i> (Appium server log) or <i>docker-android.stdout.log</i> (Android emulator log)	androidLogs
<code>sel.jup.android.appium.loglevel</code>	Log level for Appium console and logfile (<code>console[:file]</code>), either <code>debug</code> , <code>info</code> , <code>warn</code> , or <code>error</code>	debug
<code>sel.jup.android.appium.logfile</code>	Log file for Appium output	
<code>sel.jup.android.screen.width</code>	Android device screen width (in pixels)	1900
<code>sel.jup.android.screen.height</code>	Android device screen height (in pixels)	900
<code>sel.jup.android.screen.depth</code>	Android device screen color depth	24+32

Configuration key	Description	Default value
<code>sel.jup.docker.server.url</code>	URL to connect with the Docker Host	
<code>sel.jup.docker.wait.timeout.sec</code>	Timeout (in seconds) to wait for Docker container	20
<code>sel.jup.docker.poll.time.ms</code>	Poll time (in ms) for asking to Docker container if alive	200
<code>sel.jup.docker.default.socket</code>	Default Docker socket path	<code>/var/run/docker.sock</code>
<code>sel.jup.docker.hub.url</code>	Docker Hub URL	https://hub.docker.com/
<code>sel.jup.docker.stop.timeout.sec</code>	Timeout in seconds to stop Docker containers at the end of tests	5
<code>sel.jup.docker.api.version</code>	Docker API version	1.35
<code>sel.jup.docker.host</code>	Host to use when connecting to exposed docker port instead of dynamic lookup	
<code>sel.jup.docker.network</code>	Docker network	bridge
<code>sel.jup.docker.timezone</code>	Timezone for browsers in Docker containers	Europe/Madrid
<code>sel.jup.docker.lang</code>	Language for Docker containers	en
<code>sel.jup.docker.startup.timeout.duration</code>	Docker startup timeout in duration format	3m
<code>sel.jup.server.port</code>	Selenium-Jupiter Server port	4042
<code>sel.jup.server.path</code>	Selenium-Jupiter Server path	<code>/wd/hub</code>
<code>sel.jup.server.timeout.sec</code>	Selenium-Jupiter Server timeout (in seconds)	180
<code>sel.jup.properties</code>	Location of the properties files (in the project classpath)	<code>/selenium-jupiter.properties</code>
<code>sel.jup.selenium.server.url</code>	Selenium Server URL, to be used instead of <code>@DriverUrl</code> or for browsers in Docker	

These properties can be overwritten in different ways. As of version 2.1.0 of *Selenium-Jupiter*, the configuration manager can be used:

```
SeleniumExtension seleniumExtension = new SeleniumExtension();

seleniumExtension.getConfig().setVnc(true);
seleniumExtension.getConfig().setRecording(true);
seleniumExtension.getConfig().useSurefireOutputFolder();
seleniumExtension.getConfig().setBrowserListFromDockerHub(false);
seleniumExtension.getConfig().chromedriver().setForceCache(true);
seleniumExtension.getConfig().chromedriver().setOverride(true);
```

We can also use Java system properties, for example:

```
System.setProperty("sel.jup.recording", "true");
```

i. or by command line, for example:

```
-Dsel.jup.recording=true
```

Moreover, the value of these properties can be overridden by means of environmental variables. The name of these variables result from putting the name in uppercase and replacing the symbol . by _. For example, the property sel.jup.recording can be overridden by the environment variable `SEL_JUP_RECORDING`.

Tuning WebDriverManager

As introduced before, *Selenium-Jupiter* internally uses [WebDriverManager](#) to manage the required binary to control local browsers. This tool can be configured in several ways, for example to force using a given version of the binary (by default it tries to use the latest version), or force to use the cache (instead of connecting to the online repository to download the binary artifact). For further information about this configuration capabilities, please take a look to the [WebDriverManager documentation](#).

In this section we are going to present a couple of simple examples tuning somehow WebDriverManger. The following example shows how to force a version number for a binary, concretely for Edge:

```

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.RegisterExtension;
import org.openqa.selenium.edge.EdgeDriver;

import io.github.bonigarcia.seljup.SeleniumExtension;

public class EdgeSettingVersionJupiterTest {

    @RegisterExtension
    static SeleniumExtension seleniumExtension = new SeleniumExtension();

    @BeforeAll
    static void setup() {
        seleniumExtension.getConfig().edgedriver().version("3.14393");
    }

    @Test
    void webRTCtest(EdgeDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}

```

This other example shows how to force cache (i.e. binaries previously downloaded by WebDriverManager) to avoid the connection with online repository to check the latest version:

```

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

import io.github.bonigarcia.seljup.SeleniumExtension;

public class ForceCacheJupiterTest {

    @RegisterExtension
    static SeleniumExtension seleniumExtension = new SeleniumExtension();

    @BeforeAll
    static void setup() {
        seleniumExtension.getConfig().chromedriver().config()
            .setForceCache(true);
    }

    @Test
    public void test(ChromeDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
            containsString("JUnit 5 extension for Selenium"));
    }
}

```

Screenshots

Selenium-Jupiter provides several built-in features for making **screenshots** for each of the browser sessions at the end of the test. These screenshots, can be encoded as **Base64** or stored as **PNG** images. The following configuration keys are used to control the way and format in which screenshots are made:

- **sel.jup.screenshot.at.the.end.of.tests**: This key indicates whether or not screenshots will be made at the end of every browser session. The accepted values for this configuration key are:
 - **true** : Screenshots are always taken at the end of tests.
 - **false** : Screenshots are not taken at the end of tests.
 - **whenfailure** : Screenshots are only taken if the test fails.
- **sel.jup.screenshot.format**: Format for the screenshot. The accepted values for this key are two:
 - **base64** : Base64 screenshots are logged using the **debug** level of ([Simple Logging Facade for Java \(SLF4J\)](#)). You can copy&paste the resulting Base 64 string in the URL bar of any browser and watch the screenshot.
 - **png** : Screenshots are stored as PNG images. The output folder for these images is configured using the configuration key **sel.jup.output.folder** (the default value of this property is **.**, i.e. the local folder).

Take into account that a big **base64** string will be added to your logs if this option is configured. This feature can be especially useful for build server in the cloud (such as Travis CI), in which we don't have access to the server file system but can track easily the test output log.

TESTS

Running io.github.bonigarcia.test.basic.ChromeJupiterTest

...

2017-12-13 02:41:53 [main] DEBUG i.g.bonigarcia.SeleniumExtension - Screenshot (in Base64) at the end of session 5712cce700bb76d8f5f5d65a00e2c7bc (copy&paste this string as URL in browser to watch it)

data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAYkAAANaCAIAAAACvpRSAAAgAE1EQVR4nOy9e3xV1Zn//1lr384t5+RyCDmBhJBwSRACBAqhXFQsY1GLZeygrWVGba1TtVU72gv9fgfnK3Yq36mdqTq0LTL1R8fyq2Wk3kCGSqEMLxKQiyQICZAAJySHnJyT5Nz2ZX3/ODGEk71DDuSCuN4vXrySnXX2WWfvtc7670d51vMQxhg4HA6Hw+Fw0OkTDofTfQkdiH5wOBw0h8PhcEzh2ovD4XA4HA5n80Dai8PhcDgcDmfW4NqLw+FwOBwOZ/Dg2ovD4XA4HA5n80Dai8PhcDgcDmfW4NqLw+FwOBwOp5/5wx/+YPUUnrr04HA6Hw+Fw+pOk8LKsX1x7cTgcDofD4fQb3SWXqfzi2o

...

nr2SgWY2TMdyRZMNENknWYTIo75WxuUWNCij9k178gCKO3BHLtkeQjiMYEgXy66/8f5nl+x57/98mYAvFysJUmckrGBLMv/dC0SLJSUjrXpwn3Ba6MgAlILbL+/8/5fAruDfRS108WQBr2oDm4uNtmZOPKRm7GSgptWXBfJPrAAAL01EQVS4/biJEm74sxVmrVz///+U468MXDMRZc7zY5Aj1uDndWEG2gCWk+R5ARMg15aZrga3N7XBT9vCbXrBWPjx/3/96tsQz8LL4foDPzSV1H4crofHwrFXxyDZx1xHaWN7EjwAiT/f69Gf/8G105SUDCBnj2EW8huvfnTLafNSMsD043+89QUmwCz/yUhp9AFYT/DCDwBA00ZPpbBVqWAAAABJRu5ErkJggg==

Tests run: 2, Failures: 1, Errors: 0, Skipped: 1, Time elapsed: 7.219 sec <<< FAILURE!

- in io.github.bonigarcia.test.basic.ChromeJupiterTest

testWithOneChrome(ChromeDriver) Time elapsed: 6.594 sec <<< FAILURE!

Known issues

Testing localhost

A daily use case in web development is testing an web application deployed in a local server (<http://localhost:port/path>). In the case of using Docker browsers to test this web application, we need to be aware that *localhost* inside a Docker container is not the local host anymore, but the container. To handle this issue, different approaches can be taken.

- If your host (i.e. the local machine running the tests and hosting the web application under test) is Linux, Docker creates a bridge named **docker0** by default. Both the Docker host and the Docker containers have an IP address on that bridge. We can find out the equivalent IP address to localhost using the following command (in this example, the address **172.17.0.1** will be used to replace localhost in our tests):

```
$ 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
```

- If your host is Mac and using *Docker for Mac*, we can use the special DNS name `docker.for.mac.host.internal` which will resolve to the internal IP address used by the host.
- If your host is Windows and using *Docker for Windows*, we can use the special DNS name `docker.for.win.host.internal`.

Using old versions of Selenium

Selenium-Jupiter requires **Selenium 3**. In fact, *selenium-java* 3.x is incorporated as transitive dependency when using *Selenium-Jupiter*. Nevertheless, it might occur that an old version of Selenium (e.g. 2.x) is used in a project. In that case, *Selenium-Jupiter* will typically fail as follows:

```
org.junit.jupiter.api.extension.ParameterResolutionException: Failed to resolve
parameter [...] in executable [...]
    at
org.junit.jupiter.engine.execution.ExecutableInvoker.resolveParameter(ExecutableInvoke
r.java:221)
    ...
Caused by: java.lang.NoClassDefFoundError: org/openqa/selenium/MutableCapabilities
    at java.lang.Class.getDeclaredConstructors0(Native Method)
    at java.lang.Class.privateGetDeclaredConstructors(Unknown Source)
    at java.lang.Class.getConstructor0(Unknown Source)
    at java.lang.Class.getDeclaredConstructor(Unknown Source)
    at
io.github.bonigarcia.SeleniumExtension.getDriverHandler(SeleniumExtension.java:228)
    at
io.github.bonigarcia.SeleniumExtension.resolveParameter(SeleniumExtension.java:175)
    at
org.junit.jupiter.engine.execution.ExecutableInvoker.resolveParameter(ExecutableInvoke
r.java:207)
```

The solution is forcing the latest version of *selenium-java* to 3.x.

For example, this issue is likely to happen in Spring-Boot 1.x projects. When using a Spring-Boot 1.x parent, Selenium 2 artifacts versions are established by default. To solve it, we need to force at least the following Selenium artifacts:

```

<properties>
  <selenium.version>3.11.0</selenium.version>
</properties>

<dependencies>
  <dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>${selenium.version}</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-api</artifactId>
    <version>${selenium.version}</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-chrome-driver</artifactId>
    <version>${selenium.version}</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-firefox-driver</artifactId>
    <version>${selenium.version}</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-remote-driver</artifactId>
    <version>${selenium.version}</version>
    <scope>test</scope>
  </dependency>
</dependencies>

```

Recordings in Windows or Mac

Docker for Windows/Mac only mounts by default the home of your user (e.g. `C:\Users\your-user` in Windows). If *Selenium-Jupiter* is executed from a different parent folder of that home, it won't be possible to write the MP4 recording (by default it uses the current folder, i.e. `.`). This configuration can be changed using the default mappings of in Docker for Windows/Mac settings.

As an alternative, the value of the configuration key `sel.jup.output.folder` of *Selenium-Jupiter* can be used to some path inside the user folder (e.g. `C:\User\your-user\whatever`). The MP4 recording should end in there.

About

Selenium-Jupiter (Copyright © 2017-2019) is a project created by [Boni Garcia \(@boni_gg\)](#) licensed under [Apache 2.0 License](#). This documentation is released under the terms of [CC BY 3.0](#) (also available in [PDF](#)). There are several ways to get in touch with *Selenium-Jupiter*:

- Questions about *Selenium-Jupiter* are supposed to be discussed in [StackOverflow](#), using the tag *selenium-jupiter*.
- Comments, suggestions and bug-reporting should be done using the [GitHub issues](#).
- If you think *Selenium-Jupiter* can be enhanced, consider contribute to the project by means of a [pull request](#).