selenium-jupiter

A JUnit 5 extension for Selenium WebDriver

Boni García

Version 2.0.0

Table of Contents

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. Regarding the extension model of JUnit 5, it allows to incorporate extra capabilities for Jupiter tests. On the other hand, Selenium WebDriver is a testing framework which allows to control 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 WebDriver within Jupiter 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 in `@Test` methods as parameters. Concretely, *selenium-jupiter* allows to inject subtypes of the **WebDriver** interface (e.g. *ChromeDriver*, *FirefoxDriver*, and so on).

Using *selenium-jupiter* it's easy as pie. 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>2.0.0</version>
    <scope>test</scope>
</dependency>
```

NOTE *selenium-jupiter* 2.0.0 depends on ***selenium-java* 3.8.1**, ***webdrivermanager* 2.1.0**, **appium java-client 5.0.4**, and **docker-java 3.0.14**. Therefore, by using the *selenium-jupiter* dependency, those libraries (*selenium-java*, *webdrivermanager*, *appium java-client*, and *docker-java*) 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 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.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class ChromeAndFirefoxJupiterTest {

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

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

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

}
```

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. Although this browser is supported, Internet Explorer is deprecated (in favor of Edge) and its use is highly discouraged.
- `RemoteWebDriver`: Used to control remote browsers (*Selenium Grid*).
- `AppiumDriver`: Used to control mobile devices (Android, iOS).

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 known 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.

Motivation

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. One 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:

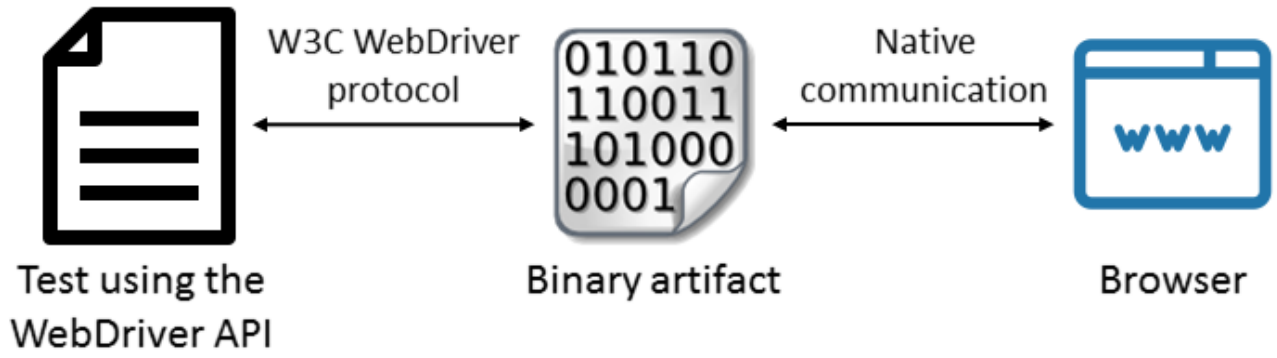

Figure 1. WebDriver general scenario

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:

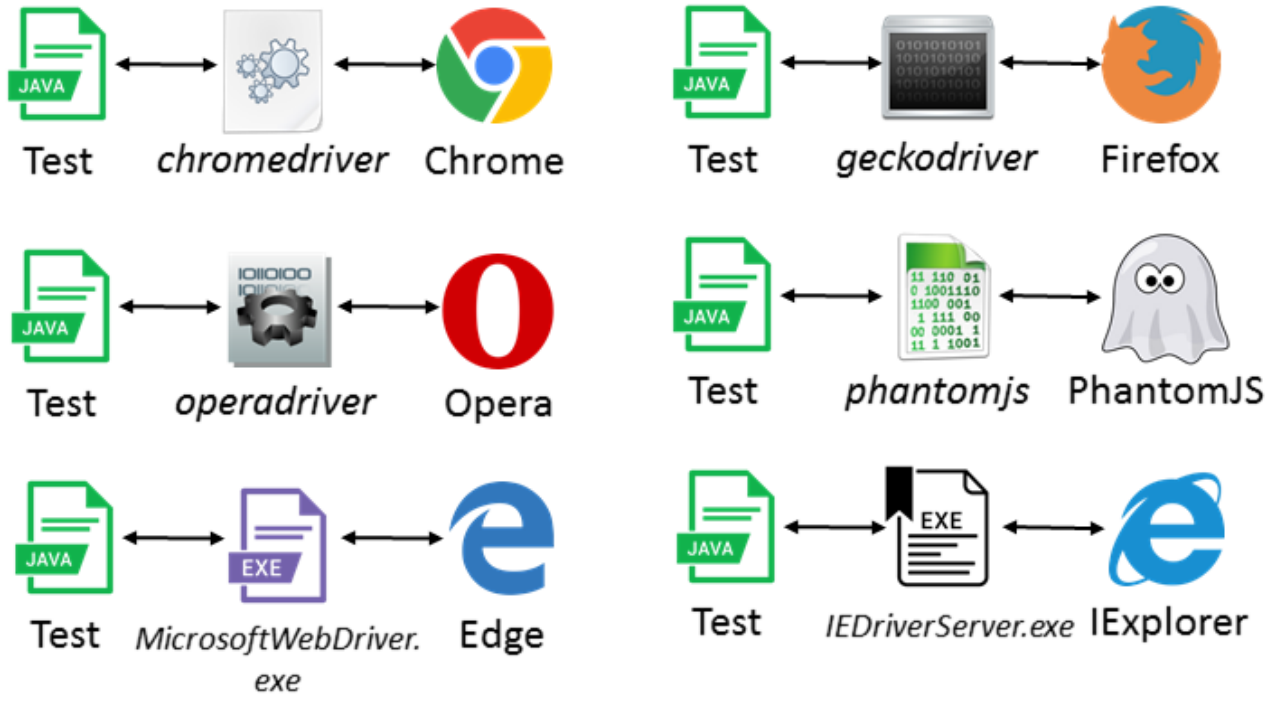

Figure 2. WebDriver scenario for Chrome, Firefox, Opera, PhantomJS, Edge, and Internet Explorer

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 and use it.

Local browsers

This section contains a comprehensive collection of basuc examples demonstrating the basic usage of *seleniun-jupiter* in JUnit 5 tests using different types of local browsers. All these examples are part of the test suite of *selenium-jupiter* and are executed on Travis CI.

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.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class ChromeJupiterTest {

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

    @Test
    public 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.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("A JUnit 5 extension for Selenium WebDriver"));
    }

    @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"));
    }

}
```

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.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class EdgeJupiterTest {

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

}
```

TIP

The required version of *MicrosoftWebDriver.exe* depends on the version on Edge to be used (more info here). By default, WebDriverManager downloads and uses the latest version of the binaries. Nevertheless, a concrete version can be fixed. Take a look to the advance examples section to find out how to setup the different options of WebDriverManager.

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.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("A JUnit 5 extension for Selenium WebDriver"));
    }

}
```

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.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class SafariJupiterTest {

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

}
```

PhamtomJS

PhamtomJS 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 PhamtomJS 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.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.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("A JUnit 5 extension for Selenium WebDriver"));
    }

}
```

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-java as Docker client and the Docker images for browsers provided by Selenoid.

As shown in the following section, the mode of operation is similar to local browser. We simply asks 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 `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 **ever green browsers**, and it is implementing by consuming the REST API of Docker Hub, asking for the list of Selenoid browsers. On the other hand, the second `@Test` (called `testChromeWithVersion`) a fixed version is set, in this case `62.0`.

```
import static io.github.bonigarcia.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.DockerBrowser;
import io.github.bonigarcia.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("A JUnit 5 extension for Selenium WebDriver"));
    }

    @Test
    public void testChromeWithVersion(
            @DockerBrowser(type = CHROME, version = "62.0") RemoteWebDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
                containsString("A JUnit 5 extension for Selenium WebDriver"));
    }

}
```

In this other example, wildcards will be 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 maintainance of the underlying infraestructure.

```
import static io.github.bonigarcia.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.DockerBrowser;
import io.github.bonigarcia.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class DockerChromeLatestJupiterTest {

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

    @Test
    public void testFormerChrome(
            @DockerBrowser(type = CHROME, version = "latest-1") RemoteWebDriver
driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
                containsString("A JUnit 5 extension for Selenium WebDriver"));
    }

}
```

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).

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.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.DockerBrowser;
import io.github.bonigarcia.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("A JUnit 5 extension for Selenium WebDriver"));
    }

    @Test
    public void testVersion(
            @DockerBrowser(type = FIREFOX, version = "56") RemoteWebDriver driver) {
        driver.get("https://bonigarcia.github.io/selenium-jupiter/");
        assertThat(driver.getTitle(),
                containsString("A JUnit 5 extension for Selenium WebDriver"));
    }

}
```

NOTE Notice that the version of the second test is simply `56`. The actual version of the image is `56.0`, 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.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.DockerBrowser;
import io.github.bonigarcia.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("A JUnit 5 extension for Selenium WebDriver"));
    }

}
```

Remote sessions (VNC)

selenium_jupiter allows to track the evolution of our browsers in Docker using Virtual Network Computing (VNC) sessions. By default, always a browser in Docker is started by *selenium_jupiter*, the VNC URL 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.

```
2018-01-16 00:57:04 [main] DEBUG i.g.b.handler.DockerDriverHandler - Session
a19b446bdffef1783edd737f48c7266d VNC URL:
http://192.168.99.100:61231/vnc.html?host=192.168.99.100&port=61222&path=vnc/94fce597b
678e94eeb86f148d9ea2d95&resize=scale&autoconnect=true&password=selenoid
```

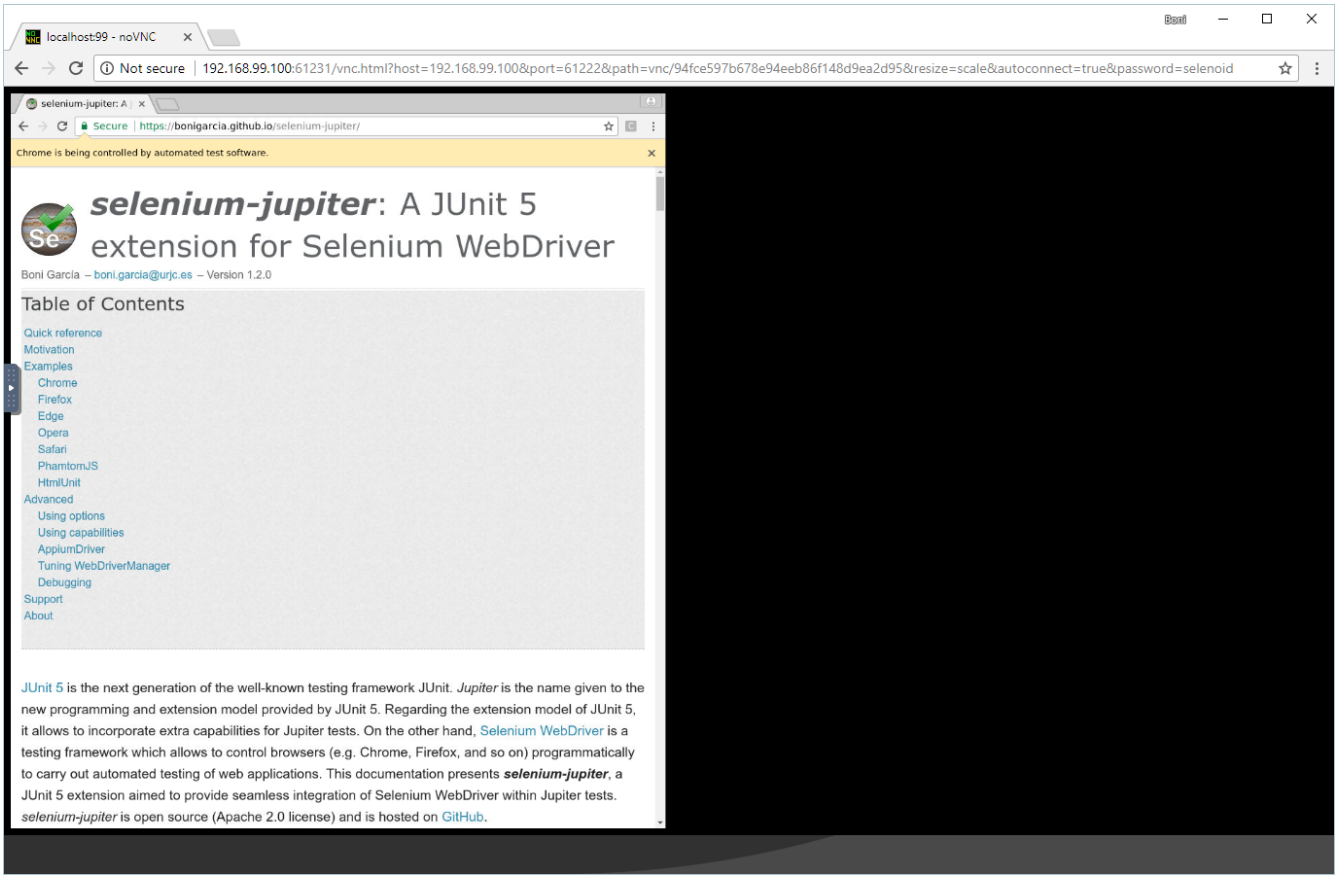

Figure 3. Example of VNC session on Chrome

NOTE	This capability can be deactivated using the configuration key `sel.jup.vnc` (more info on section Configuration).

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.

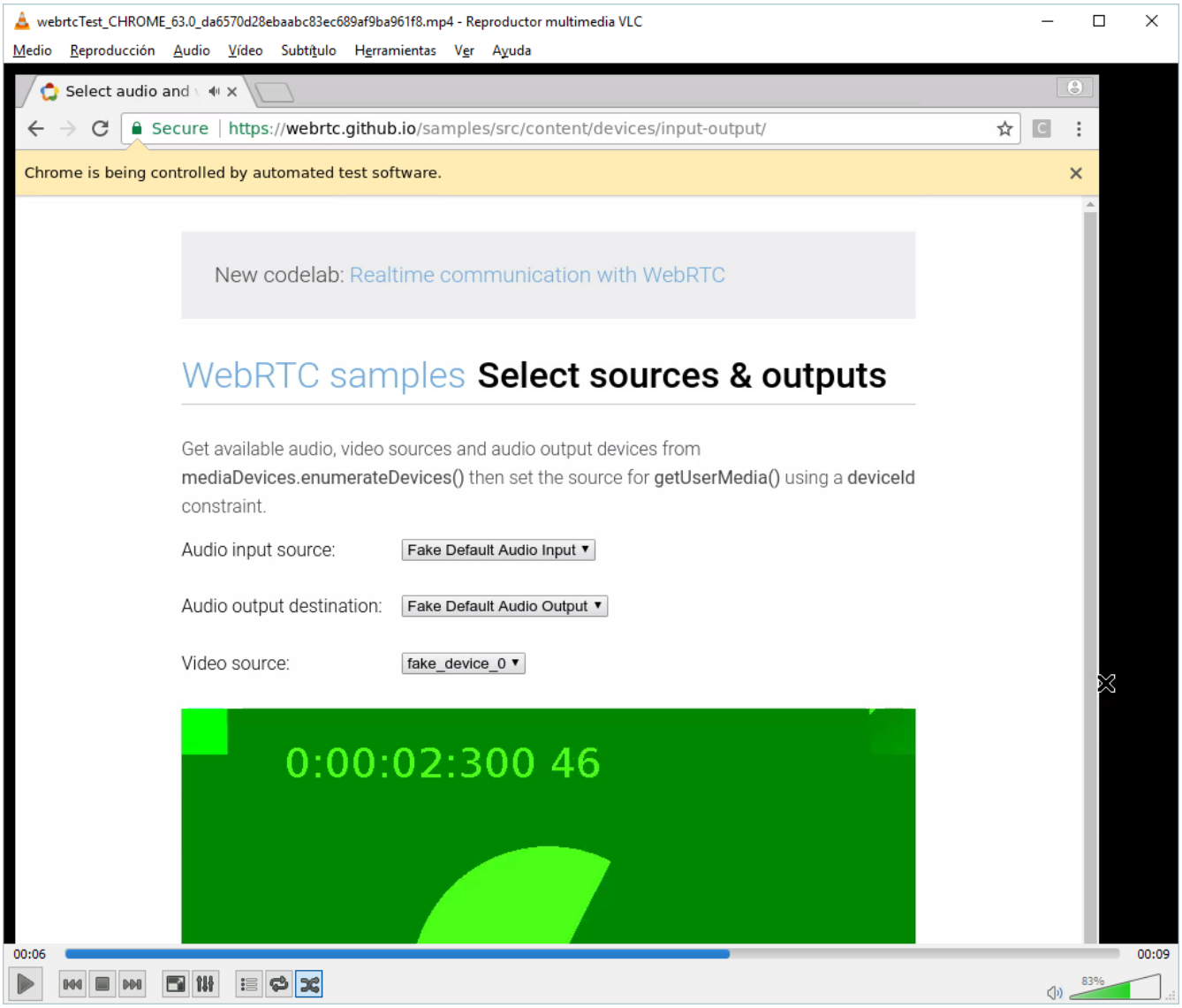

Figure 4. Example of recording played in VLC

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, instead of a type of `RemoteWebDriver`, the test will declare a `List<RemoteWebDriver>`. For example as follows:

```
import static io.github.bonigarcia.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.DockerBrowser;
import io.github.bonigarcia.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class PerformenceDockerChromeJupiterTest {

    static final int NUM_BROWSERS = 3;

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

    @Test
    public void testPerformance(
            @DockerBrowser(type = CHROME, 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(
                            "A JUnit 5 extension for Selenium WebDriver"));
                } 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. In this case, it might be interesting reduce the resource consumption by deactivating the VNC remote sessions setting the configuration key `sel.jup.vnc` to `false`.

Advanced features

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

import io.github.bonigarcia.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class TemplateTest {

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

}
```

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 `sel.jup.browser.template.json.file` do not start with the word `classpath:`, 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": "latest-2"
         }
      ]
   ]
}
```

When we execute the template test, in this case we will have three actual tests: the first using the *latest* version of Chrome, the second using the previous to stable version of Chrome (`latest-1`), and third using two versions older than the current stable (`latest-2`). For instance, if we run the test in Eclipse, we will get the following output:

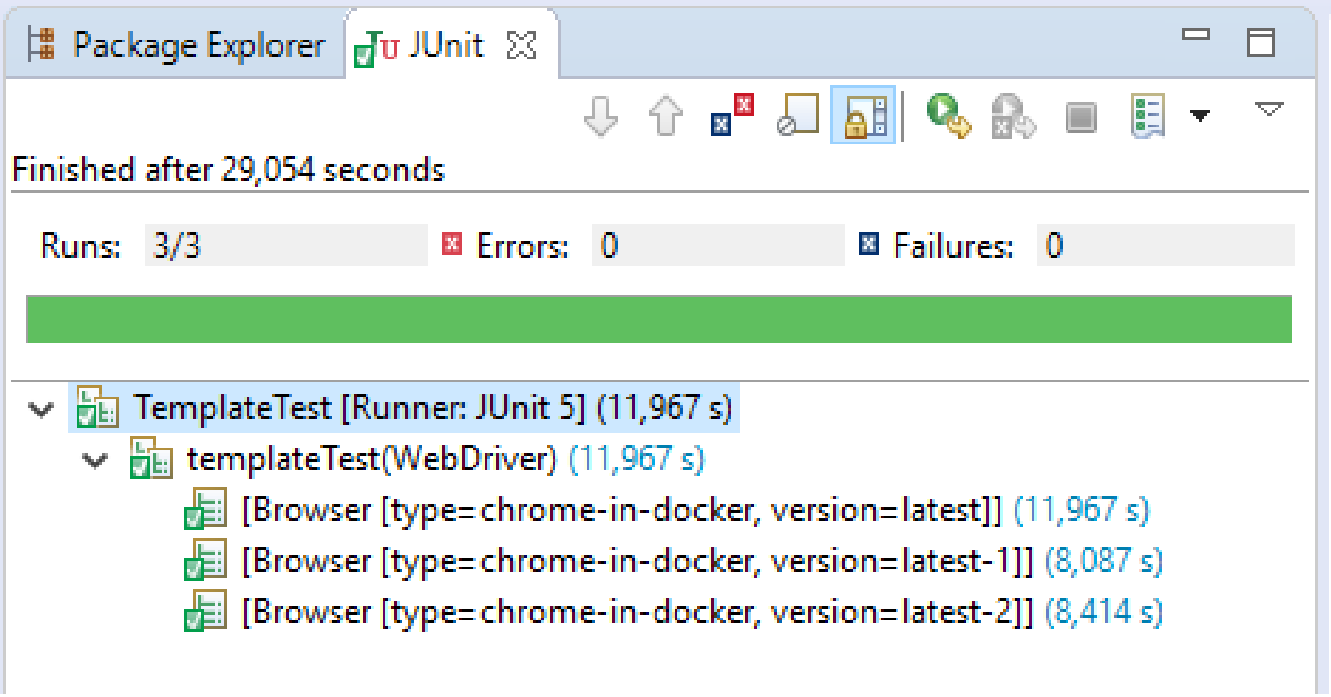

Figure 5. Example of test template execution (with one parameter) with Eclipse

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

- `type`: Type of browsers. The accepted values are:
  - `chrome`: For local Chrome browsers.
  - `firefox`: For local Firefox browsers.
  - `edge`: For local Edge 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.
- `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`, `57.0`, etc., depending of the browser).

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

import io.github.bonigarcia.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class TemplateTwoBrowsersTest {

    @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("A JUnit 5 extension for Selenium WebDriver"));
        assertThat(driver2.getTitle(),
                containsString("A JUnit 5 extension for Selenium WebDriver"));
    }

}
```

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 exection will be based on the headless browser PhantomJS (first parameter) and Opera in Docker (second paramter).

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

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

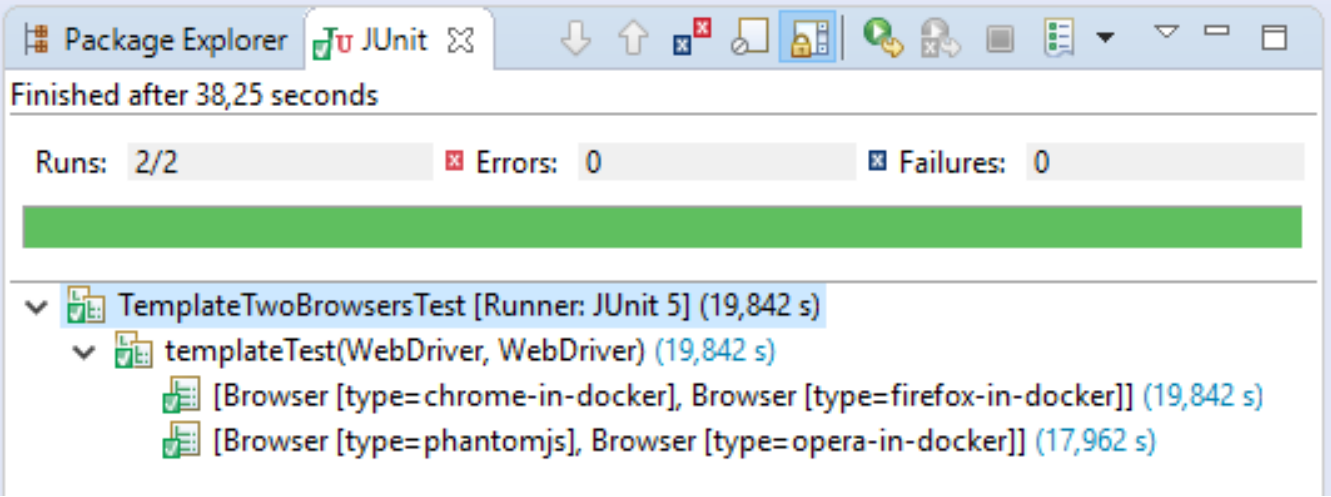

Figure 6. Example of test template execution (with two parameters) with Eclipse

Using options

So far, we have discovered how to use different local browsers (Chrome, Firefox, Edge, Opera, Safari, PhamtomJS, HtmlUnit) of Docker browsers (Chrome, Firefox, Opera) with its default options and supposing that the browser to be used is installed on the machine running the test. 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)?
- And what about remote browsers (Selenium Grid)? How is `RemoteWebDriver` supported by *selenium-jupiter*?

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:

- `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.
- `Options` (*field-level*): Annotation to configure *options* (e.g. `ChromeOptions` for Chrome, `FirefoOptions` for Firefox, `EdgeOptions` for Edge, `OperaOptions` for Opera, and `SafariOptions` for Safari).
- `DriverCapabilities` (*parameter-level* or *field-level*): Annotation to configure the desired *capabilities* (WebDriver's object `DesiredCapabilities`).
- `DriverUrl` (*parameter-level* or *field-level*): Annotation used to identify the URL value needed to instantiate a `RemoteWebDriver` object.

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. Keep reading to find out several examples about that.

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.Arguments;
import io.github.bonigarcia.Extensions;
import io.github.bonigarcia.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("A JUnit 5 extension for Selenium WebDriver"));
    }

    @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("A JUnit 5 extension for Selenium WebDriver"));
    }

}
```

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.Options;
import io.github.bonigarcia.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"));
    }

}
```

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 tets (i.e. tests using remote browsers). To that aim, an Selenium Hub (also known as *Selenium Server*) should be up an running, and its URL should 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 a 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.BeforeAll;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.grid.selenium.GridLauncherV3;
import org.openqa.selenium.Capabilities;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.remote.RemoteWebDriver;

import io.github.bonigarcia.DriverCapabilities;
import io.github.bonigarcia.DriverUrl;
import io.github.bonigarcia.SeleniumExtension;
import io.github.bonigarcia.wdm.ChromeDriverManager;
import io.github.bonigarcia.wdm.FirefoxDriverManager;

@ExtendWith(SeleniumExtension.class)
public class RemoteWebDriverJupiterTest {

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

    @DriverCapabilities
    Capabilities capabilities = firefox();

    @BeforeAll
    static void setup() throws Exception {
        // Start hub
        GridLauncherV3.main(new String[] { "-role", "hub", "-port", "4444" });

        // Register Chrome in hub
        ChromeDriverManager.getInstance().setup();
        GridLauncherV3.main(new String[] { "-role", "node", "-hub",
                "http://localhost:4444/grid/register", "-browser",
                "browserName=chrome", "-port", "5555" });

        // Register Firefox in hub
        FirefoxDriverManager.getInstance().setup();
        GridLauncherV3.main(new String[] { "-role", "node", "-hub",
                "http://localhost:4444/grid/register", "-browser",
                "browserName=firefox", "-port", "5556" });
    }

    @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("A JUnit 5 extension for Selenium WebDriver"));
    }

}
```

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 static org.openqa.selenium.chrome.ChromeOptions.CAPABILITY;
import static org.openqa.selenium.remote.DesiredCapabilities.chrome;

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.remote.DesiredCapabilities;

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

@ExtendWith(SeleniumExtension.class)
public class ChromeWithGlobalCapabilitiesJupiterTest {

    @DriverCapabilities
    DesiredCapabilities capabilities = chrome();
    {
        Map<String, String> mobileEmulation = new HashMap<String, String>();
        mobileEmulation.put("deviceName", "Nexus 5");
        Map<String, Object> chromeOptions = new HashMap<String, Object>();
        chromeOptions.put("mobileEmulation", mobileEmulation);
        capabilities.setCapability(CAPABILITY, chromeOptions);
    }

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

}
```

AppiumDriver

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.DriverCapabilities;
import io.github.bonigarcia.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 java.io.File;
import java.net.URISyntaxException;

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

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

@ExtendWith(SeleniumExtension.class)
public class AppiumApkJupiterTest {

    @DriverCapabilities
    DesiredCapabilities capabilities = new DesiredCapabilities();
    {
        try {
            File apk = new File(this.getClass()
                    .getResource("/selendroid-test-app.apk").toURI());
            capabilities.setCapability("app", apk.getAbsolutePath());
            capabilities.setCapability("deviceName", "Android");

        } catch (URISyntaxException e) {
            e.printStackTrace();
        }
    }

    @Test
    void testWithAndroid(AppiumDriver<MobileElement> driver)
            throws InterruptedException {
        WebElement button = driver.findElement(By.id("buttonStartWebview"));
        button.click();

        WebElement inputField = driver.findElement(By.id("name_input"));
        inputField.clear();
        inputField.sendKeys("Custom name");
    }

}
```

Tuning WebDriverManager

As introduced before, *selenium-jupiter* internally uses WebDriverManager to manage the required binary to control localc 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 static java.lang.System.setProperty;
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.ExtendWith;
import org.openqa.selenium.edge.EdgeDriver;

import io.github.bonigarcia.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class EdgeSettingVersionJupiterTest {

    @BeforeAll
    static void setup() {
        setProperty("wdm.edgeVersion", "3.14393");
    }

    @Test
    void webrtcTest(EdgeDriver driver) {
        driver.get("http://www.seleniumhq.org/");
        assertThat(driver.getTitle(),
                containsString("A JUnit 5 extension for Selenium WebDriver"));
    }

}
```

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 static java.lang.System.setProperty;
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.ExtendWith;
import org.openqa.selenium.chrome.ChromeDriver;

import io.github.bonigarcia.SeleniumExtension;

@ExtendWith(SeleniumExtension.class)
public class ForceCacheJupiterTest {

    @BeforeAll
    static void setup() {
        setProperty("wdm.forceCache", "true");
    }

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

}
```

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 valued 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 if 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.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
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)

1Zn//1lr384t5+RyCDmBhJBwSRACBAqhXFQsYlGLZeygrWVGbalTtVU72gv9fgfnK3Yq36mdqTqOLTL1R8fyq2
Wk3kCGSqEMlxKQiyQICZAAJySHnJyT5Nz2ZX3/ODGEk71DDuSCuN4vXrySnXXX2WWfvtc767Od51vMQxhg4HA6H
w+FwOOkTDofTfQkdiH5wOBwOh8PhcEzh2ovD4XA4HA5n8ODai8PhcDgcDmfw4NqLw+FwOBwOZ/Dg2ovD4XA4HA
5n8ODai8PhcDgcDmfw4NqLw+FwOBwOp5/5wx/+YPUnrr04HA6Hw+FwDofD4fQb3SWXq
2o
...
nr2SgWY2TMdyRZMNENknWYTIo75WxuUWNCij9k178gCKO3BHLtkeQjiMYEgXy66/8f5nl+x57/98mYAvFysJUm
ckrGBLMv/dC0SlJSUjrXpwn3Ba6MgAlILbL+//8/5fAruDfRSl08WQBr2oDm4uNtmZOPKRm7GSgptWXBFJPrAA
AAl0lEQVS4/biJEm74sxVmrvz///+U468MXDMRZc7zY5Aj1uDndWEG2gCWk+R5ARMgl5aZrga3N7XBT9vCbxrB
WPjx/3/96tsQz8LL4foDPzSVlH4crofHwrFXxyDZx1xHaWN7EjwAiT/f69Gf/8G1O5SUDCBnj2EW8huvfnTLaF
NSMsD043+89QUmwCz/yUhp9AFYT/DCDwBA0OZPpbBVqwAAAABJRU5ErkJggg==
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!
```

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 attache 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 an 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.

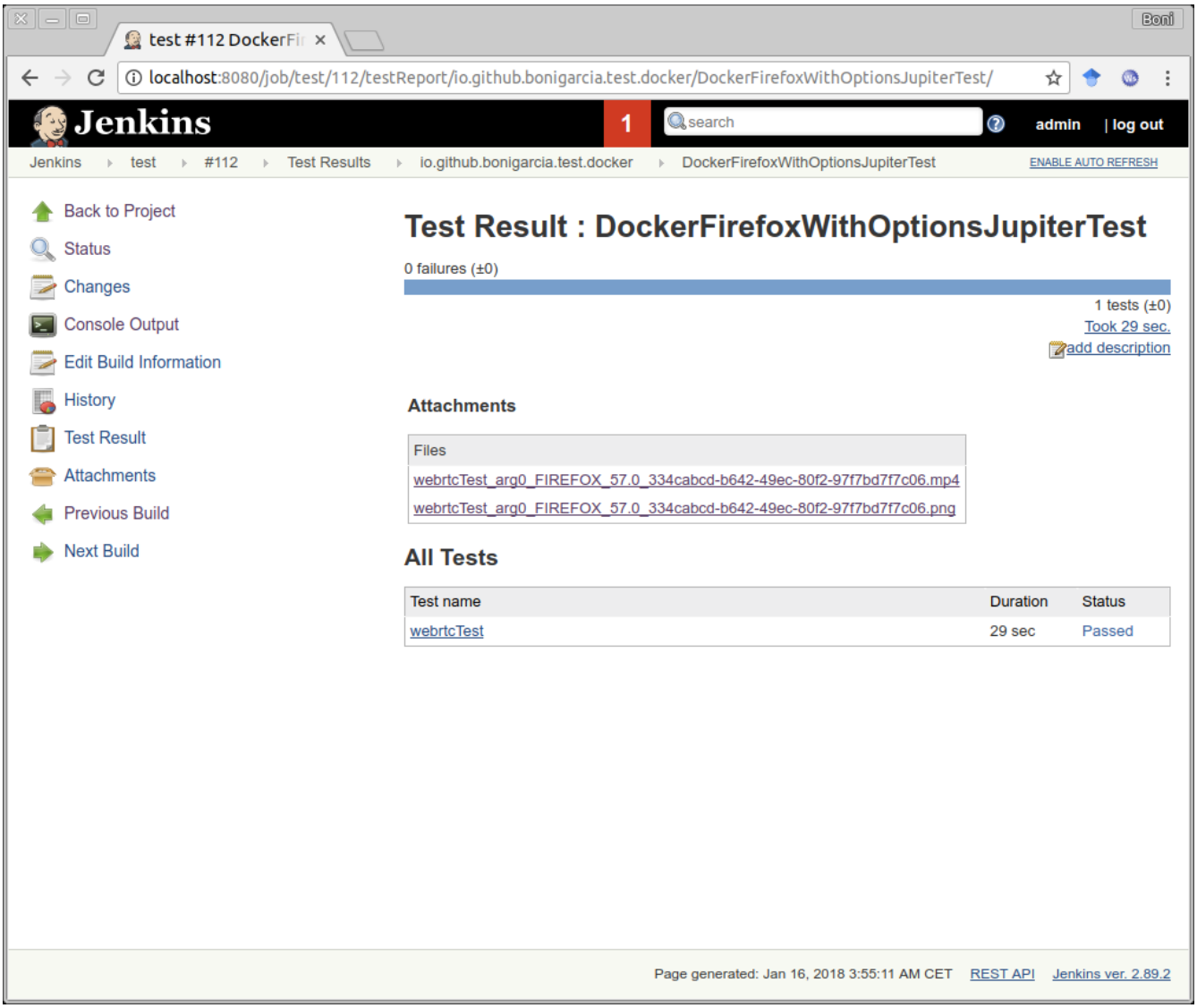

Figure 7. Example of test execution through Jenkins with attachements

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

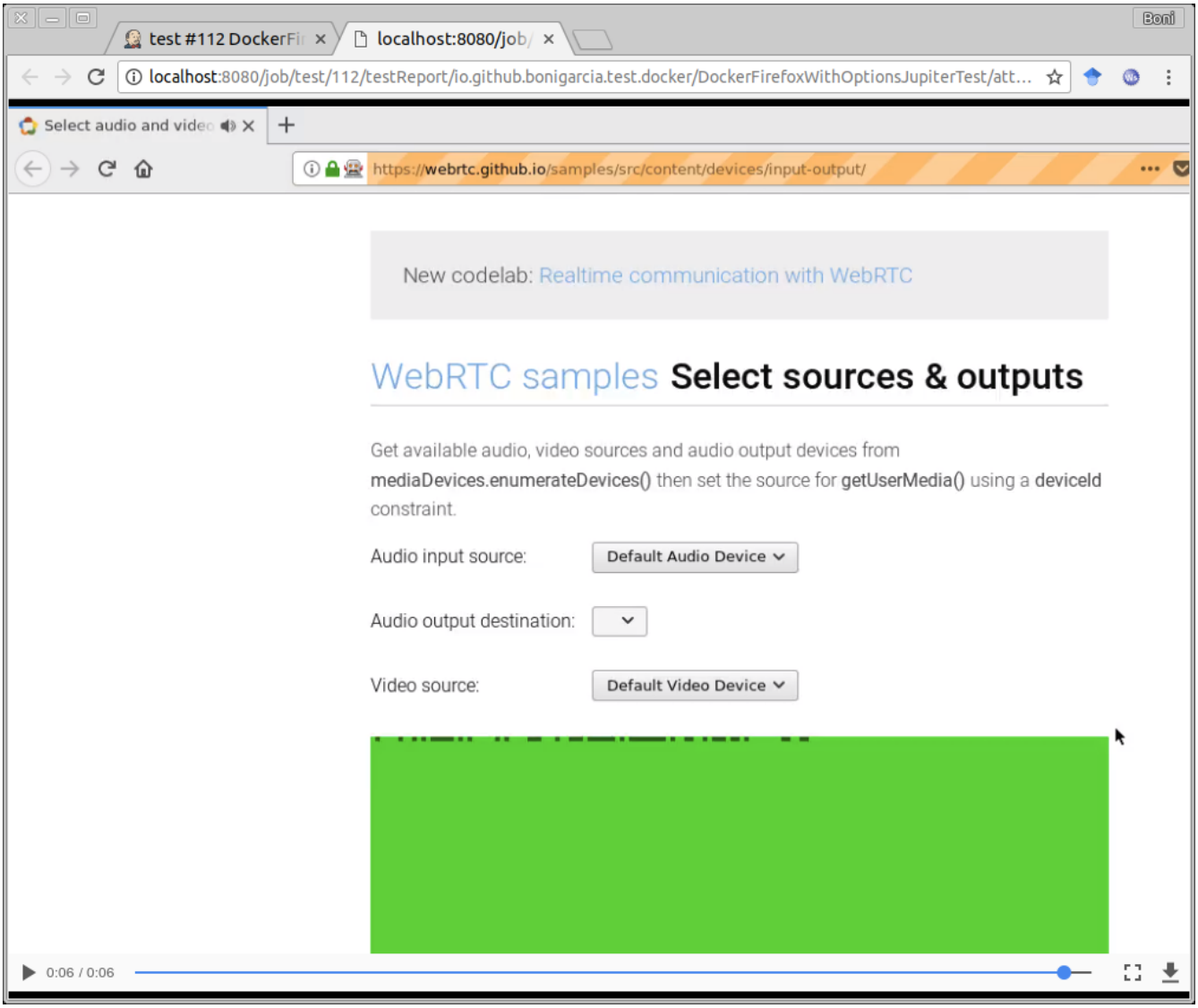

Figure 8. Example of test execution through Jenkins with attachements

Test template are also compatible with this feature. For instance, the following test test, when is executed in Jenkins using the the following configuration:

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

i. will result in the following attachements:

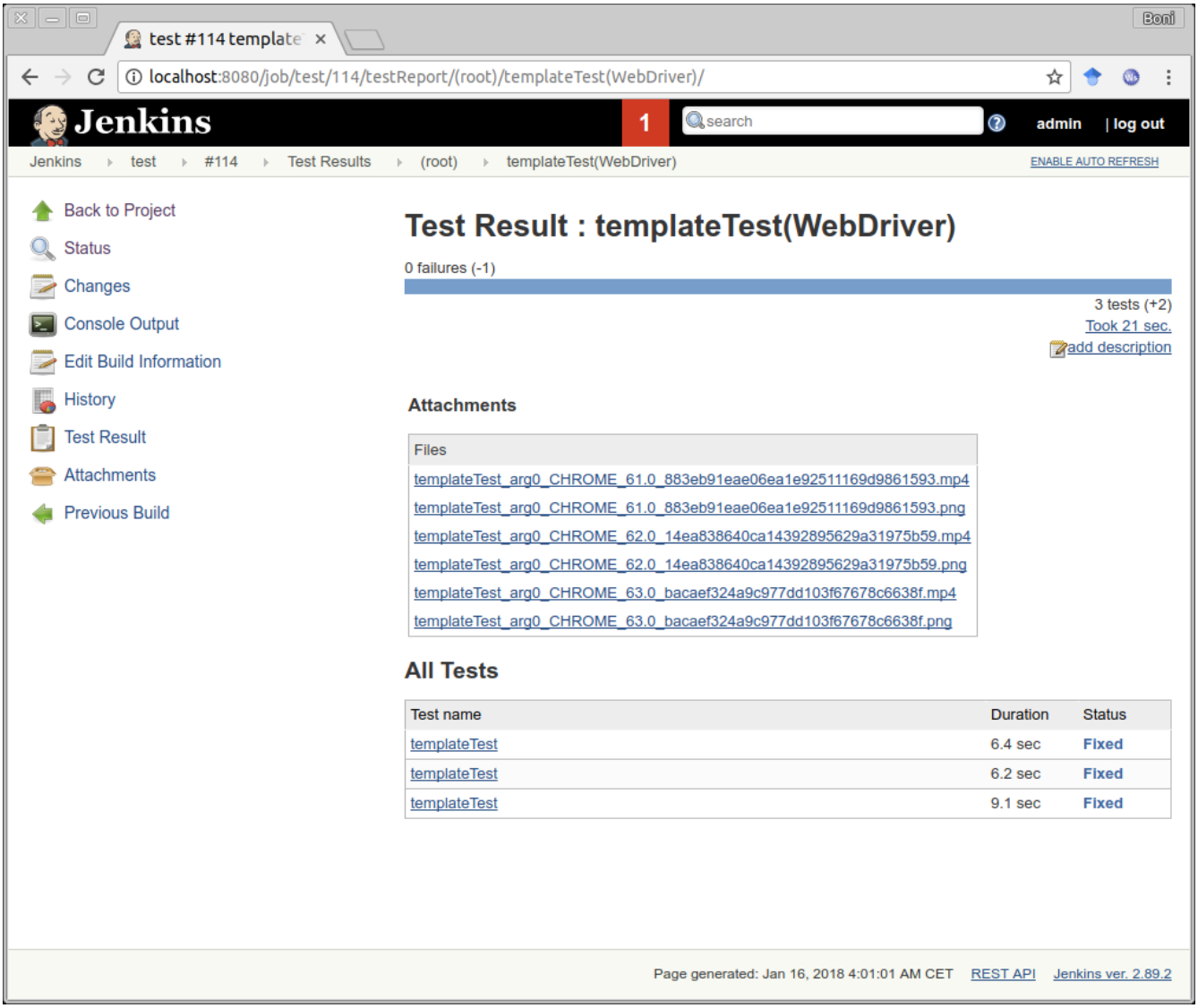

Figure 9. Example of template test execution through Jenkins with attachements

i. and we can see the recording, for instace:

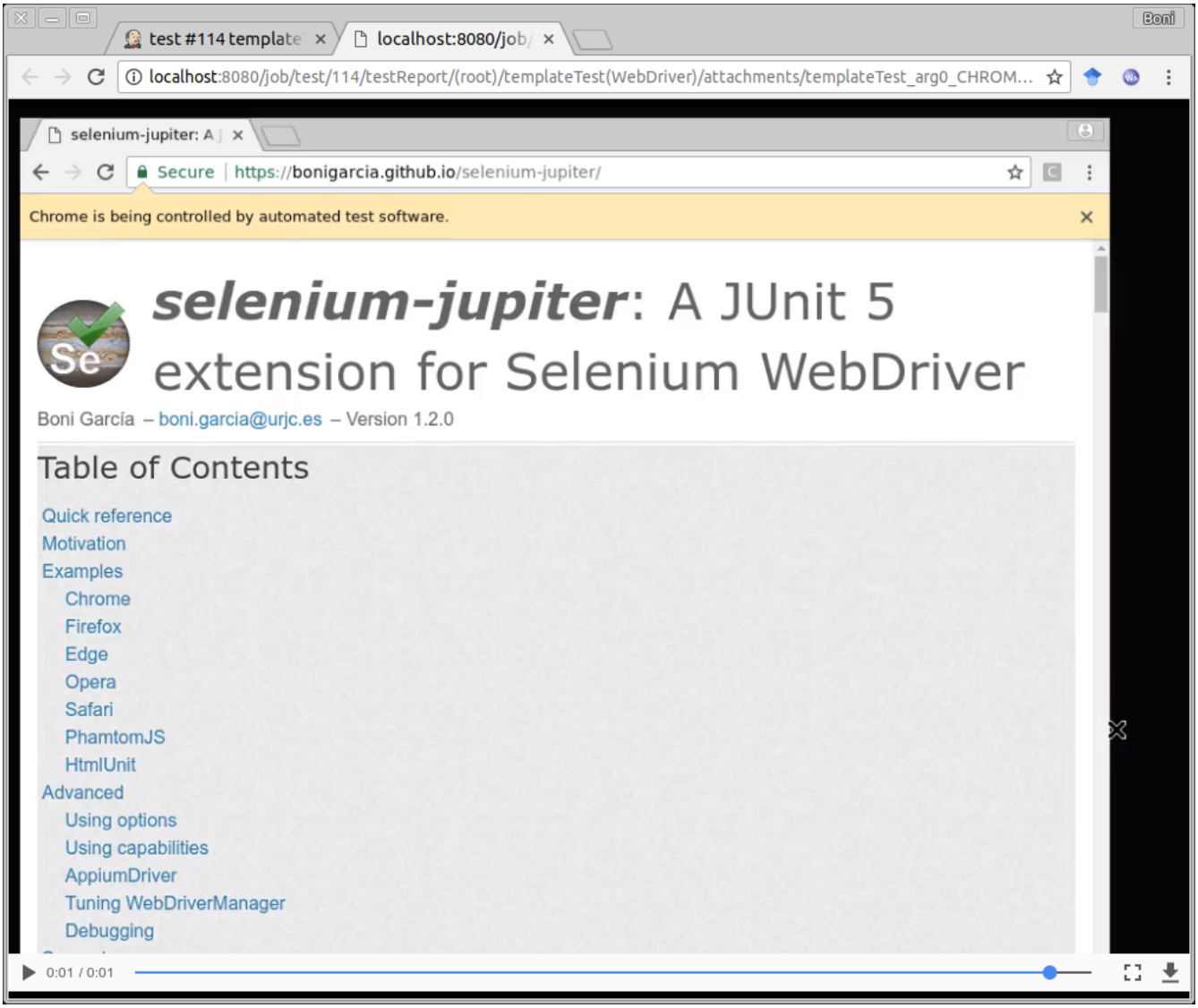

Figure 10. Example of template test execution through Jenkins with attachements

Configuration

Configuration parameters for *selenium_jupiter* are set in the selenium-jupiter.properties file. The following table summarizes all the configuration keys available.

Table 1. Configuration keys in selenium_jupiter

Configuration key	Description	Default value
`sel.jup.vnc`	Check VNC session for Docker browsers	`true`
`sel.jup.vnc.screen.resolution`	Screen resolution of VNC sessions (format *<width>x<height>x<colors-depth>*)	`1920x1080x24`
`sel.jup.vnc.create.redirect.html.page`	Redirect VNC URL to HTML page	`false`
`sel.jup.recording`	Record Docker browser session (in MP4 format)	`false`

Configuration key	Description	Default value
`sel.jup.recording.video.screen.size`	Video screen size for recordings (width and height)	`1024x768`
`sel.jup.recording.video.frame.rate`	Video frame rate for recordings	`12`
`sel.jup.recording.image`	Docker image for recordings	`selenoid/video-recorder:latest`
`sel.jup.output.folder`	Output folder for recordings, screenshots, and HTML redirect pages	`.`
`sel.jup.screenshot.at.the.end.of.tests`	Make screenshots at the end of the test	`whenfailure`
`sel.jup.screenshot.format`	Format for screenshots	`png`
`sel.jup.exception.when.no.driver`	Throw exception in case of exception or not	`true`
`sel.jup.browser.template.json.file`	Browsers scenario (JSON) path	`classpath:browsers.json`
`sel.jup.browser.list.from.docker.hub`	Update Docker images list from Docker Hub	`true`
`sel.jup.browser.session.timeout.duration`	Session timeout for Docker browsers (in Golang duration format)	`1m0s`
`sel.jup.selenoid.image`	Selenoid (Golang Selenium Hub) Docker iamage	`aerokube/selenoid:1.4.3`
`sel.jup.selenoid.port`	Selenoid port	`4444`
`sel.jup.selenoid.vnc.password`	VNC password for Selenoid sessions	`selenoid`
`sel.jup.novnc.image`	noVNC Docker image	`psharkey/novnc:3.3-t6`
`sel.jup.novnc.port`	noVNC Docker port	`8080`
`sel.jup.chrome.image.format`	Selenoid Docker images format for Chrome with VNC	`selenoid/vnc:chrome_%s`
`sel.jup.chrome.first.version`	First version of Docker Chrome (used when `sel.jup.browser.list.from.docker.hub =false`)	`48.0`
`sel.jup.chrome.latest.version`	Latest version of Docker Chrome (used when `sel.jup.browser.list.from.docker.hub =false`)	`63.0`
`sel.jup.firefox.image.format`	Selenoid Docker images format for Firefox with VNC	`selenoid/vnc:firefox_%s`
`sel.jup.firefox.first.version`	First version of Docker Firefox (used when `sel.jup.browser.list.from.docker.hub =false`)	`3.6`

Configuration key	Description	Default value
`sel.jup.firefox.latest.version`	Latest version of Docker Firefox (used when `sel.jup.browser.list.from.docker.hub =false`)	`57.0`
`sel.jup.opera.image.format`	Selenoid Docker images format for Opera with VNC	`selenoid/vnc:opera_%s`
`sel.jup.opera.first.version`	First version of Docker Opera (used when `sel.jup.browser.list.from.docker.hub =false`)	`33.0`
`sel.jup.opera.latest.version`	Latest version of Docker Opera (used when `sel.jup.browser.list.from.docker.hub =false`)	`50.0`
`sel.jup.docker.server.url`	URL to connect with the Docker Host	`` ``
`sel.jup.docker.wait.timeout.sec`	Timeout (in seconds) to wait for Docker container	`20`
`sel.jup.docker.poll.time.ms`	Poll time (in ms) for asking to Docker container if alive	`200`
`sel.jup.docker.default.socket`	Default Docker socket path	`/var/run/docker.sock`
`sel.jup.docker.default.host`	Default Docker host IP	`127.0.0.1`
`sel.jup.docker.hub.url`	Docker Hub URL	`https://hub.docker.com/`
`sel.jup.properties`	Location of the properties files (in the project classpath)	`/selenium-jupiter.properties`

These properties can be overwritten by Java system properties, for example:

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

i. or by command line, for example:

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

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`.

Support

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.

About