Selenium-Jupiter

Boni García

Version 4.0.0-SNAPSHOT, 13-08-2021

Table of Contents

Chapter 1. Motivation

Selenium WebDriver is a library that allows controlling web browsers programmatically. It provides a cross-browser API that can be used to drive web browsers (e.g., Chrome, Edge, or Firefox, among others) using different programming languages (e.g., Java, JavaScript, Python, C#, or Ruby). The primary use of Selenium WebDriver is implementing automated tests for web applications. For this reason, it is a common practice to embed the calls to the Selenium WebDriver API in tests implemented using a unit testing framework like JUnit.

JUnit 5 is the next generation of the popular testing framework JUnit. It provides a brand-new programming and extension model called *Jupiter*. The comprehensive nature of Jupiter is very convenient to develop Selenium WebDriver tests on the top of JUnit 5.

What is Selenium-Jupiter?

Selenium-Jupiter is an open-source Java library that implements a JUnit 5 extension for developing Selenium WebDriver tests. Selenium-Jupiter uses several features of the Jupiter extension (such as parameters resolution, test templates, or conditional test execution). Thanks to this, the resulting Selenium-Jupiter tests follow a minimalist approach (i.e., the required boilerplate code for WebDriver is reduced) while providing a wide range of advanced features for end-to-end testing.

Chapter 2. Setup

Selenium-Jupiter should be used as a Java dependency. We typically use a *build tool* (such as Maven or Gradle) to resolve the Selenium-Jupiter dependency. In Maven, it can be done as follows (notice that it is declared using the `test` scope since it is typically used in tests classes):

```
<dependency>
    <groupId>io.github.bonigarcia</groupId>
    <artifactId>selenium-jupiter</artifactId>
    <version>4.0.0-SNAPSHOT</version>
    <scope>test</scope>
</dependency>
```

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

```
dependencies {
    testImplementation("io.github.bonigarcia:selenium-jupiter:4.0.0-SNAPSHOT")
}
```

As of version 4, Selenium-Jupiter does not include Selenium WebDriver as a transitive dependency.

Chapter 3. Features

Selenium-Jupiter provides an extension class for Jupiter tests called `SeleniumJupiter` (available in the package `io.github.bonigarcia.seljup`). This extension class should be used as a parameter in the annotation `@ExtendWith` (or `@RegisterExtension`) of a Jupiter test.

3.1. Local browsers

At first glance, Selenium-Jupiter seamlessly uses local browsers (e.g., Chrome, Firefox, Edge, etc.). To that aim, Selenium-Jupiter uses the *parameter resolution* mechanism provided by Jupiter. This way, we simply need to declare a parameter of the `WebDriver` hierarchy (e.g., `ChromeDriver` to control Chrome, `FirefoxDriver` to control Firefox, etc.) as a test parameter. Internally, Selenium-Jupiter uses WebDriverManager to manage the required driver (e.g., chromedriver for Chrome) before starting the test. Then, it instantiates the `WebDriver` object and injects it into the test. Finally, it terminates the browser gracefully. All in all, the required boilerplate of a test using Selenium WebDriver and Selenium-Jupiter is reduced to the minimum. The following test shows a basic example of this feature:

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

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

import io.github.bonigarcia.seljup.SeleniumJupiter;

@ExtendWith(SeleniumJupiter.class)
class ChromeTest {

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

}
```

The possible `WebDriver` types we can specify as test parameters are `FirefoxDriver` (see example), `EdgeDriver` (see example), `SafariDriver` (see example), `OperaDriver` (see example), `InternetExplorerDriver` (see example), and `ChromiumDriver` (see example).

Most of the features provided by Selenium-Jupiter can be used in conjunction with Selenium WebDriver versions 3 and 4. Nevertheless, some of them (e.g., the class `ChromiumDriver` to control Chromium browsers) are only available in Selenium WebDriver 4.

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

3.2. Conditional tests

Selenium-Jupiter provides the class-level annotation `@EnabledIfBrowserAvailable` to conditionally skip tests. Thanks to this annotation, a test is skipped when the browser specified as a parameter is not available in the system. This capability allows executing the same test suite in different machines (e.g., a local Mac OS with Safari and a Linux CI server) independently of the browser availability. The following example shows a test that is skipped if Safari is not in the system.

```
import static io.github.bonigarcia.seljup.Browser.SAFARI;
import static org.assertj.core.api.Assertions.assertThat;

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

import io.github.bonigarcia.seljup.EnabledIfBrowserAvailable;
import io.github.bonigarcia.seljup.SeleniumJupiter;

@EnabledIfBrowserAvailable(SAFARI)
@ExtendWith(SeleniumJupiter.class)
class SafariTest {

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

}
```

3.3. Remote browsers

Selenium-Jupiter also allows controlling remote browsers. These browsers are typically provided by a Selenium Grid infrastructure or a cloud provider (such as Sauce Labs, BrowserStack, CrossBrowserTesting, etc.). To this aim, Selenium-Jupiter provides two annotations:

- `@DriverUrl`: Annotation used to identify the URL of the Selenium Server.
- `@DriverCapabilities`: Annotation used to configure the desired capabilities, such as the browser type, version, platform, etc.

Both annotations can be defined at parameter or field-level. The following example shows a test

using these annotations at the parameter-level. This example uses Selenium Grid in standalone mode to start a Selenium Server listening to port 4444 and hosting a Firefox browser. Notice that the type of parameter injected in the test is the generic `WebDriver` interface (it could also be `RemoteWebDriver`).

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

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

import io.github.bonigarcia.seljup.DriverCapabilities;
import io.github.bonigarcia.seljup.DriverUrl;
import io.github.bonigarcia.seljup.SeleniumJupiter;
import io.github.bonigarcia.wdm.WebDriverManager;

@ExtendWith(SeleniumJupiter.class)
class RemoteFirefoxTest {

    @BeforeAll
    static void setup() {
        // Resolve driver
        WebDriverManager.firefoxdriver().setup();

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

    @Test
    void test(@DriverUrl("http://localhost:4444/") @DriverCapabilities({
            "browserName=firefox" }) WebDriver driver) {
        driver.get("https://bonigarcia.dev/selenium-webdriver-java/");
        assertThat(driver.getTitle()).contains("Selenium WebDriver");
    }

}
```

You can find an example of an equivalent remote test but defining the remote URL and capabilities at field-level here. A similar example, but using Sauce Labs is this one.

3.4. Docker browsers

Selenium-Jupiter allows using browsers in Docker containers in an effortless manner. To that aim, we need to associate the `@DockerBrowser` annotation to parameters of the type `WebDriver` or `RemoteWebDriver` in Jupiter tests. Simply with that, Selenium-Jupiter discovers the latest version of the browser in Docker Hub, pulls this image (if necessary), and starts the browser in the Docker container. The following example shows how to use this feature with a Chrome browser in Docker:

```
import static io.github.bonigarcia.seljup.BrowserType.CHROME;
import static org.assertj.core.api.Assertions.assertThat;

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

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

@ExtendWith(SeleniumJupiter.class)
class DockerChromeTest {

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

}
```

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

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

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

3.4.1. Browser Versions

A significant aspect of the Docker containers for browsers is that the latest version on Docker Hub is used when the browser version is not specified (like the examples explained before). This way, these *dockerized* browsers are auto-maintained, in the sense that these tests use the latest version available without any additional effort.

Nevertheless, we can use a browser version in Docker using the attribute `version()` of the `@DockerBrowser` annotation. This attribute accepts a `String` parameter specifying the version. This version can be fixed (e.g., `91.0`), and it also accepts the following wildcards:

- `"latest"` : To specify the latest version explicitly (default option).

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

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

```
import static io.github.bonigarcia.seljup.BrowserType.CHROME;
import static org.assertj.core.api.Assertions.assertThat;

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

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

@ExtendWith(SeleniumJupiter.class)
class DockerChromeBetaTest {

    @Test
    void test(
            @DockerBrowser(type = CHROME, version = "beta") WebDriver driver) {
        driver.get("https://bonigarcia.dev/selenium-webdriver-java/");
        assertThat(driver.getTitle()).contains("Selenium WebDriver");
    }

}
```

3.4.2. Remote Desktop

A possible inconvenience of using browsers in Docker is that we cannot see what is happening inside the container by default. Selenium-Jupiter allows connecting to the remote desktop session by simply setting to `true` the attribute `vnc` of the `@DockerBrowser` annotation to improve this situation. When using this option, two different technologies are used internally:

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

The following example shows a test that enables this feature. We can see the noVNC URL in the `INFO` trace logs. We can use this URL to inspect and interact with the browser during the test execution

(as shown in the following picture).

```
import static io.github.bonigarcia.seljup.BrowserType.CHROME;
import static org.assertj.core.api.Assertions.assertThat;

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

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

@ExtendWith(SeleniumJupiter.class)
class DockerChromeVncTest {

    @Test
    void test(
            @DockerBrowser(type = CHROME, vnc = true, lang = "ES", timezone =
"Europe/Madrid") WebDriver driver)
            throws Exception {
        driver.get("https://bonigarcia.dev/selenium-webdriver-java/");
        assertThat(driver.getTitle()).contains("Selenium WebDriver");

        // Uncomment this line to have time for manual inspection
        // Thread.sleep(30000);
    }

}
```

Notice that the previous example also uses the attributes `lang` and `timezone` to change the language and timezone of the browser container.

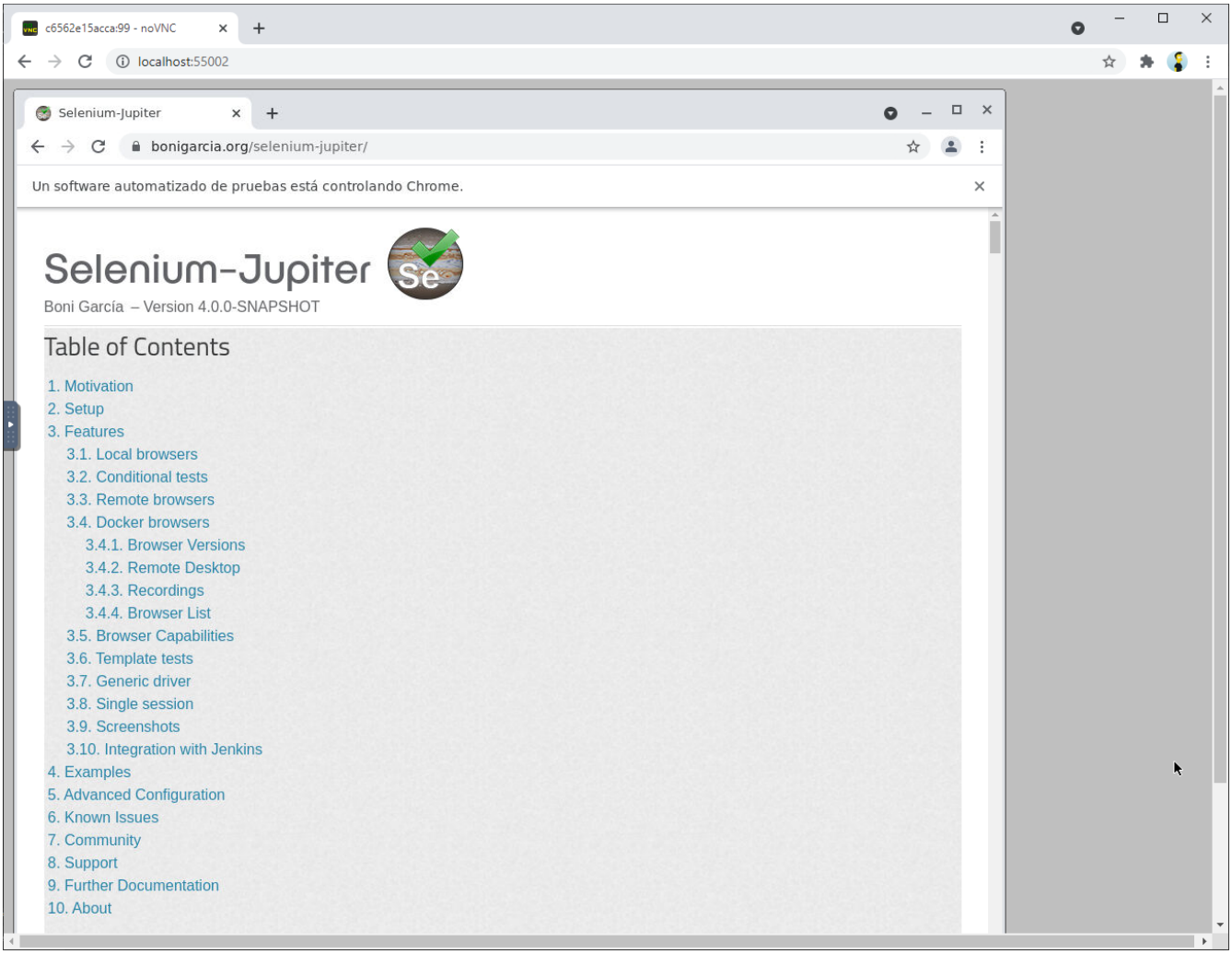

Figure 1. Example of noVNC session using Chrome in Docker

3.4.3. Recordings

The following related feature is the recording of the remote session of a dockerized browser. To enable it, we need to set to `true` the attribute `recording` of `@DockerBrowser`. Internally, Selenium-Jupiter starts another Docker container using FFmpeg to record the browser session. At the end of the test, we can find the recording in MP4, by default, in the same folder of the project root. The following example shows a simple using this feature.

```
import static io.github.bonigarcia.seljup.BrowserType.CHROME;
import static java.lang.invoke.MethodHandles.lookup;
import static org.assertj.core.api.Assertions.assertThat;
import static org.slf4j.LoggerFactory.getLogger;

import java.io.File;
import java.time.Duration;

import org.junit.jupiter.api.AfterEach;
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.SeleniumJupiter;

@ExtendWith(SeleniumJupiter.class)
class DockerChromeRecordingTest {

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

    File recordingFile;

    @AfterEach
    void teardown() {
        if (recordingFile != null) {
            assertThat(recordingFile).exists();
            recordingFile.delete();
        }
    }

    @Test
    void recordingTest(
            @DockerBrowser(type = CHROME, recording = true) RemoteWebDriver driver)
            throws InterruptedException {
        driver.get("https://bonigarcia.dev/selenium-webdriver-java/");
        assertThat(driver.getTitle()).contains("Selenium WebDriver");

        Thread.sleep(Duration.ofMillis(5).toSeconds());

        recordingFile = new File(
                "recordingTest_arg0_chrome_" + driver.getSessionId() + ".mp4");
    }

}
```

Notice that the previous example deletes the recording at the end of the test. If you want to keep the recording, omit the line `recordingFile.delete();`. The recording filename comprises the test name, parameter number, browser name, and session id (using the underscore symbol to separate these parts).

3.4.4. Browser List

Another important new feature of browsers in Docker is the possibility of asking for *many* of them by the same test. This feature can be used to implement performance or load tests seamlessly. To use this feature, we need into account two aspects. First, you need to declare the attribute `size` of the annotation `@DockerBrowser`. This numeric value sets the number of browsers demanded by the test. Second, the test should declare a `List<RemoteWebDriver>` (or `List<WebDriver>`) parameter. 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.assertj.core.api.Assertions.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.SeleniumJupiter;

@ExtendWith(SeleniumJupiter.class)
class DockerListTest {

    static final int NUM_BROWSERS = 2;

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

    @Test
    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 {}", driver.getSessionId());
                    driver.get("https://bonigarcia.dev/selenium-webdriver-java/");
                    assertThat(driver.getTitle()).contains("Selenium WebDriver");
                } finally {
                    latch.countDown();
                }
            });
        });

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

}
```

3.5. Browser Capabilities

Selenium-Jupiter provides different annotations to specify browser-specific capabilities (e.g., options, arguments, etc.) for the `WebDriver` objects to be created. These annotations are the following:

Table 1. Selenium-Jupiter annotations for specifying capabilities

Anotation	Level	Description
`@Options`	Field-level	To configure browsre options (e.g. `ChromeOptions` for Chrome, `FirefoxOptions` for Firefox, `EdgeOptions` for Edge, `OperaOptions` for Opera, and `SafariOptions` for Safari)
`@Arguments`	Parameter-level	To specify browser arguments (e.g., `--headless`, `--incognito`, etc.)
`@Preferences`	Parameter-level	To specify Firefox arguments (e.g., `media.navigator.streams.fake=true`, etc.)
`@Binary`	Parameter-level	To set the location of the browser binary
`@Extensions`	Parameter-level	To add browser extensions (*plugins*)

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 set the argument `--headless`, used for headless Chrome (i.e., without GUI). In the second test (`webrtcTest`), we use 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 add 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.

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

import java.time.Duration;

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.SeleniumJupiter;

@ExtendWith(SeleniumJupiter.class)
class ChromeOptionsTest {

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

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

        Thread.sleep(Duration.ofSeconds(5).toMillis());
    }

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

}
```

You can find plenty of examples using the annotations presented in Selenium-Jupiter annotations for specifying capabilities in the Selenium-Jupiter tests.

You can use the capabilities annotations also for browsers in Docker (see example).

3.6. Template tests

Selenium-Jupiter uses a standard feature of JUnit 5 called test templates. Test templates are a special kind of test in which the same test logic is executed several times according to some custom data. In Selenium-Jupiter, the data to feed a test template is known as *browser scenario*.

The following example shows a test using this feature. Notice that instead of declaring the method with the usual `@Test` annotation, we use the JUnit 5's `@TestTemplate`. Then, the parameter type of the test template method is `WebDriver`. This type is the generic interface of Selenium WebDriver, and the concise type (i.e. `ChromeDriver`, `FirefoxDriver`, etc.) is determined by Selenium-Jupiter in runtime.

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

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

import io.github.bonigarcia.seljup.SeleniumJupiter;

@ExtendWith(SeleniumJupiter.class)
class TemplateTest {

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

}
```

The next piece we need to execute this test template is the browser scenario. By default, a browser scenario is defined using a custom JSON notation (see below for details about this notation) called `browsers.json` and located in the project classpath (see advanced configuration section for other possibilities). For example:

```
{
   "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": "dev"
         }
      ]
   ]
}
```

When we run the template test using this browser scenario, we will execute four tests using Chrome in Docker: the first using the stable (*latest*) version, the second using the previous to the stable version(`latest-1`), the third using the beta version (`beta`), and the last one using the development version (`dev`). If we run the test in Eclipse, for example, we will get the following output:

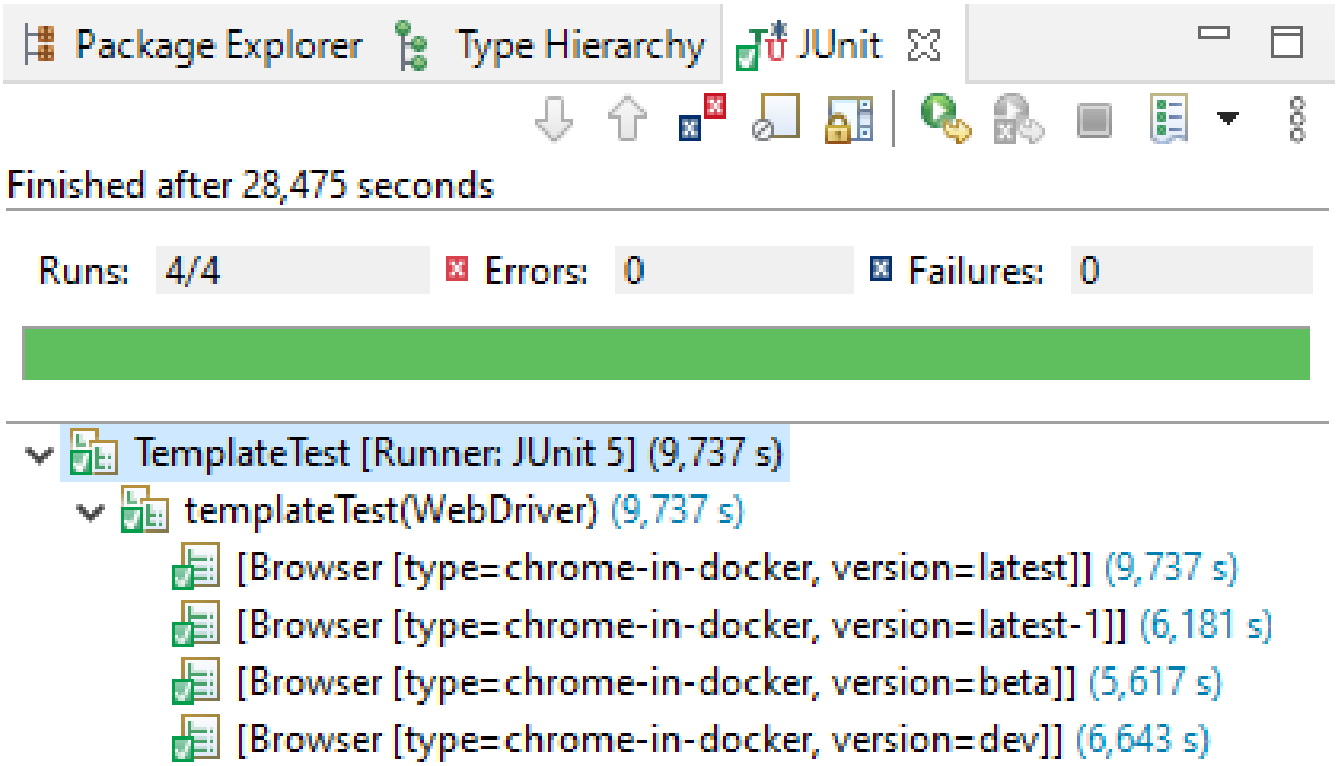

Figure 2. Example of test template execution in Eclipse

The fields accepted in the JSON notation are the following:

- `type`: Type of browsers. It can be local (`chrome` , `firefox` , `edge` , `opera`, `safari` , `chromium` , or `iexplorer`) or in Docker (`chrome-in-docker`, `firefox-in-docker`, `edge-in-docker`, `opera-in-docker`, `safari-in-docker`, or `chrome-mobile`).
- `version`: Optional value for the version. Concrete versions (e.g., `63.0`, `58.0`, etc.) and wildcards (`latest`, `latest-N`, `beta`, and `dev`) are accepted.
- `arguments`: Arguments passed to the browser (e.g., `--headless`, `--incognito`, etc.)
- `preferences`: Arguments for Firefox (e.g., `media.navigator.streams.fake=true`).
- `capabilities`: Custom key-value browser capabilities. See example.
- `remoteUrl`: Selenium Server URL (for remote tests).

You can find different examples in the Selenium-Jupiter tests. For instance: a test registering browser in the scenario programmatically (instead of JSON), a test to use a custom JSON browser scenario (and capabilities), or another test using two browsers in the same test.

3.7. Generic driver

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

See the advanced configuration section for further information about the configuration capabilities of Selenium-Jupiter.

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

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

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

import io.github.bonigarcia.seljup.SeleniumJupiter;

@ExtendWith(SeleniumJupiter.class)
class GenericTest {

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

}
```

3.8. 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. 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, `testStep1()` is executed first. 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.assertj.core.api.Assertions.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.By;
```

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.slf4j.Logger;

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

@ExtendWith(SeleniumJupiter.class)
@TestMethodOrder(OrderAnnotation.class)
@SingleSession
class OrderedTest {

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

    WebDriver driver;

    OrderedTest(ChromeDriver driver) {
        this.driver = driver;
    }

    @Test
    @Order(1)
    void testStep1() {
        log.debug("Step 1: {}", driver);
        driver.get("https://bonigarcia.dev/selenium-webdriver-java/");
        assertThat(driver.getTitle()).contains("Selenium WebDriver");

    }

    @Test
    @Order(2)
    void testStep2() {
        log.debug("Step 2: {}", driver);
        WebElement form = driver.findElement(By.partialLinkText("form"));
        assertTrue(form.isDisplayed());
        form.click();
    }

}
```

3.9. Screenshots

Selenium-Jupiter allows 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 test shows an example. Notice that this feature can be enabled using the Java configuration (method `config()` of the Selenium-Jupiter instance. This instance can be obtained using the JUnit 5 annotation `@RegisterExtension` (see advanced configuration section for more information about configuration). The default image name is composed of the test name plus the `WebDriver` *sessionId*.

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

import java.io.File;

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

import io.github.bonigarcia.seljup.SeleniumJupiter;

class ScreenshotPngTest {

    @RegisterExtension
    static SeleniumJupiter seleniumJupiter = new SeleniumJupiter();

    static File imageFile;

    @BeforeAll
    static void setup() {
        seleniumJupiter.getConfig().enableScreenshot();
        seleniumJupiter.getConfig().takeScreenshotAsPng(); // Default option
    }

    @AfterAll
    static void teardown() {
        assertThat(imageFile).exists();
        imageFile.delete();
    }

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

        imageFile = new File(
                "screenshotTest_arg0_chrome_" + driver.getSessionId() + ".png");
    }

}
```

Notice that the previous example deletes the image at the end of the test. If you want to keep the picture, omit the line `imageFile.delete();`. The screenshot filename follows the same convention as the recordings.

3.10. Integration with Jenkins

Selenium-Jupiter seamlessly integrates with Jenkins through the Jenkins attachment plugin. This integration allows to attach output files of Selenium-Jupiter (typically PNG screenshots and MP4 recordings of Docker browsers) and keep these files associated with job execution in Jenkins. This is done using the method `useSurefireOutputFolder()` (or its equivalence configuration key `sel.jup.output.folder`) in Selenium-Jupiter (see advanced configuration). For instance:

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

import java.time.Duration;

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.SeleniumJupiter;

class JenkinsTest {

    @RegisterExtension
    static SeleniumJupiter seleniumJupiter = new SeleniumJupiter();

    @BeforeAll
    static void setup() {
        seleniumJupiter.getConfig().enableScreenshot();
        seleniumJupiter.getConfig().enableRecording();
        seleniumJupiter.getConfig().useSurefireOutputFolder();
    }

    @Test
    void jenkinsTest(@DockerBrowser(type = CHROME) WebDriver driver)
            throws InterruptedException {
        driver.get("https://bonigarcia.dev/selenium-webdriver-java/");
        assertThat(driver.getTitle()).contains("Selenium WebDriver");

        Thread.sleep(Duration.ofSeconds(5).toMillis());
    }

}
```

If we run this test in Jenkins in which is already installed and configured the attachment plugin, at the end of the execution, a recording in MP4 and a PNG screenshot are attached to the Jenkins job, as shown in the following picture:

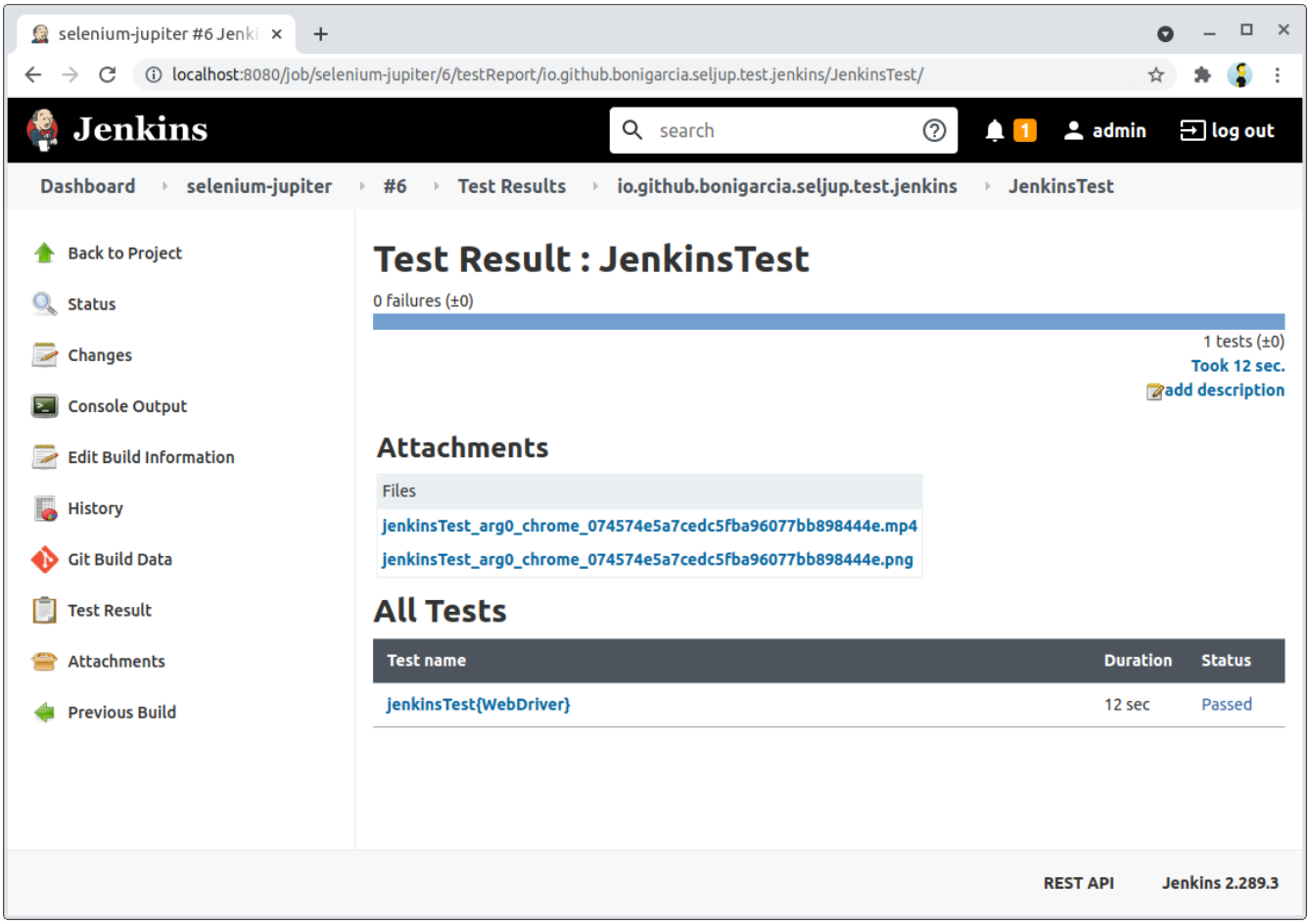

Figure 3. Example of test execution through Jenkins with attachments

Chapter 4. Examples

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

- Selenium-Jupiter Examples: Different examples with JUnit 5, Selenium WebDriver, and Selenium-Jupiter.
- Selenium-Jupiter WebRTC: Tests aimed to assess WebRTC video conference applications with Selenium-Jupiter.
- Selenium WebDriver with Java: A comprehensive collection of Selenium WebDriver 4 examples using Java as language binding.

Chapter 5. Advanced Configuration

Selenium-Jupiter provides different ways of configuration. First, by using its *Java configurator*. This capability is accessible by getting an instance of Selenium-Jupiter using the JUnit 5 annotation `@RegisterExtension` (see example).

The second alternative to tune Selenium-Jupiter is using *Java system properties*. This way, each Java configurator method has its equivalence of a unique *configuration key*. For instance, the method `setOutputFolder()` (used to specify the output folder for screenshots and recordings) is equivalent to the configuration key `sel.jup.output.folder`. These types of configuration keys can be passed when executing the tests, for example, using Maven:

```
mvn test -Dsel.jup.output.folder=/custom/path/to/output/folder
```

The third way to configure Selenium-Jupiter is using *environmental variables*. The names for these variables are made from converting each configuration key name (e.g., `sel.jup.output.folder`) to uppercase and replacing the symbol `.` with `_` (e.g., `SEL_JUP_OUTPUT_FOLDER`). These variables can be helpful to global setup parameters at the operating system level.

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

The following table describes all the possible Java methods in the Selenium-Jupiter configurator and its equivalent configuration keys.

Table 2. Configuration capabilities for Selenium-Jupiter

Java API	Configuration key	Default value	Description
`setOutputFolder(String)` `useSurefireOutputFolder()`	`sel.jup.output.folder`	`.`	Output folder for recordings and screenshots. This key accepts the special value `surefire-reports` for the integration with Jenkins
`setScreenshot(boolean)` `enableScreenshot()`	`sel.jup.screenshot`	`false`	Enable screenshots at the end of the test
`setScreenshotWhenFailure(boolean)` `enableScreenshotWhenFailure()`	`sel.jup.screenshot.when.failure`	`false`	Enable screenshots at the end of the test only if the test fails

Java API	Configuration key	Default value	Description

`setScreenshotFormat(String)`
`takeScreenshotAsPng()`
`takeScreenshotAsBase64()`
`takeScreenshotAsBase64AndPng()`
	`sel.jup.screenshot.format`	`png`	Configure screenshot format (accepted values: `png`, `base`, and `base64andpng`)

`setRecording(boolean)` `enableRecording()`	`sel.jup.recording`	`false`	Enable recording of Docker browsers
`setRecordingWhenFailure(true)` `enableRecordingWhenFailure()`	`sel.jup.recording.when.failure`	`false`	Enable recording of Docker browsers only if test fails
`setVnc(boolean)` `enableVnc()`	`sel.jup.vnc`	`false`	Enable the remote access in Docker browsers
`setBrowserTemplateJsonFile(String)`	`sel.jup.browser.template.json.file`	`classpath:browsers.json`	Location of the JSON browser scenario. If this value do not start with the word `classpath:`, the JSON file willbe searched using the specified path
`setBrowserTemplateJsonContent(String)`	`sel.jup.browser.template.json.content`	`""`	Content of JSON browsers scenario
`setProperties(String)`	`sel.jup.properties`	`selenium-jupiter.properties`	Properties file (in the project classpath) for default configuration values
`setSeleniumServerUrl(String)`	`sel.jup.selenium.server.url`	`""`	Remote URL used for remote (or template) tests

Tuning WebDriverManager

Selenium-Jupiter relies on WebDriverManager for driver resolution (e.g., chromedriver, geckodriver, etc.) and `WebDriver` instantiation (including browsers in Docker). For this reason, the configuration key `wdm.defaultBrowser` is inherited from WebDriverManager to set up the default browser in the generic driver. If you need to customize WebDriverManager, the Selenium-Jupiter Java configurator provides the method `setManager(WebDriverManager)` to specify a custom WebDriverManager instance (see example).

Chapter 6. Known Issues

HTTP response code 403

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

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

To avoid this problem, WebDriverManager can make authenticated requests using a personal access token. Take a look at the WebDriverManager documentation to discover how to set up this token.

Testing localhost

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

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

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

Chrome 92+ in Docker

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

Chapter 7. Community

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

Chapter 8. Support

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

Chapter 9. Further Documentation

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

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

Or the following book:

- García, Boni. Mastering Software Testing with JUnit 5. *Packt Publishing Ltd*, 2017.

Chapter 10. About