Selenium

How to run your Java Selenium script in Perfecto

On this page:

This section provides instructions on how to run Selenium WebDriver tests with Java in Perfecto. It assumes that you are:

- Familiar with Selenium
- Have existing tests to work with
- Are a novice user of Perfecto

To run your tests in Perfecto, you need to modify your existing scripts to include:

- What driver you want to use
- Where your Perfecto instance is located
- Who you are
- What devices you want to work on

We provide two versions of the same script to help you understand how you can modify your existing scripts to run them in Perfecto: `LocalSelenium.java` and `PerfectoSelenium.java`

For information on running the final script, see the the README.md file included with the sample project.

Prerequisites

Before you get started, make sure you have installed the following:

- Java
- Selenium WebDriver for Chrome
- An IDE of your choice, such as Eclipse or IntelliJ IDEA
- If you work with Eclipse, TestNG for Eclipse and the Maven plugin
- If you work with IntelliJ IDEA, the Maven plugin
- Maven (download and install)

1 | Get started

The starting point is `LocalSelenium.java`, a short Java script with Maven dependencies. The `pom.xml` file is institutional here because it holds all configurations and dependencies. In its initial state, the file is very simple.

Note: We have simplified the script intentionally. It only serves the purpose of showing you how to connect to Perfecto.

The script accesses the Perfecto website and verifies the title.

To get started:

1. Access the sample project in GitHub and copy the clone URL: https://github.com/PerfectoMobileSA/PerfectoSampleProject
2. Open your IDE and check out the project from GitHub.
3. Download the OS specific chromedriver into the `libs` folder of the project and update the `webdriver.chrome.driver` as applicable.
4. Run the `LocalSelenium.java` project as TestNG Test.

2 | Configure the script for Perfecto

In this step, we update the `pom.xml` file with the required Perfecto dependencies and modify the script from Step 1 to add in security information, the Perfecto cloud name, driver details, Smart Reporting information, and test data. We also want to make sure that the script exits gracefully.

The updated script is called `PerfectoSelenium.java`. The following procedure walks you through the configuration.

Expand a step to view its content.

A | Copy the dependencies

Copy the dependencies in the `pom.xml` file and paste them into the `pom.xml` file of your own project. This step is essential because the imports in our final script only work when the dependencies specified in this file are available.

B | Supply the security token

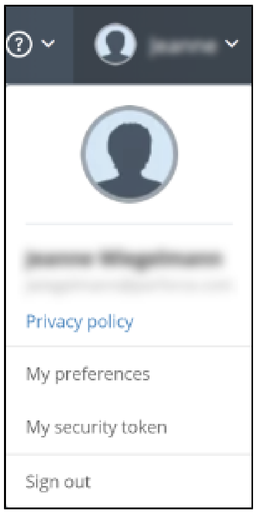
Generate your security token through the Perfecto UI. Then define your security token as a system variable called `securityToken`. The script uses this variable to retrieve the security token:

```
String securityToken = System.getProperty("securityToken");
...
capabilities.setCapability("securityToken", securityToken);
```

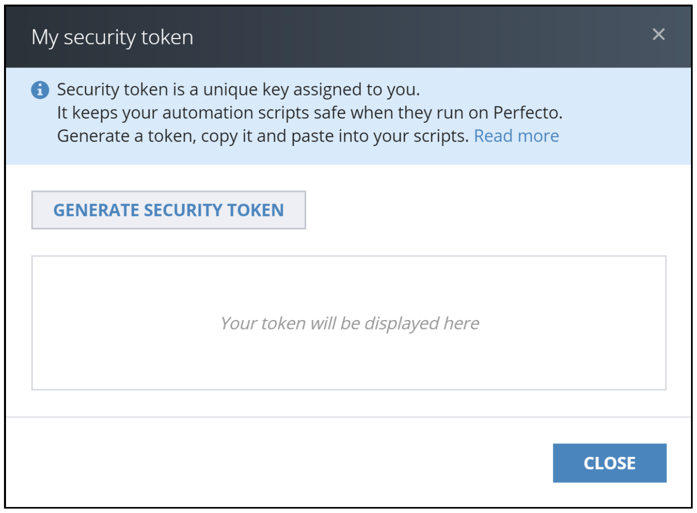
This is the recommended way of including the security token, but you can also pass your security token from Maven or any CI tool, such as Jenkins, by passing the `-DsecurityToken=<<token>> system property` while running the install goal of Maven. For more information on Maven Command Line Options, see https://books.sonatype.com/mvnref-book/reference/running-sect-options.html.

To generate a security token:

1. In the Perfecto UI at <*YourCloud*>.app.perfectomobile.com (where *YourCloud* is your actual cloud name, such as *mobilecloud*), click your user name and select **My security token**.



2. In the **My security token** form, click **Generate Security Token**.

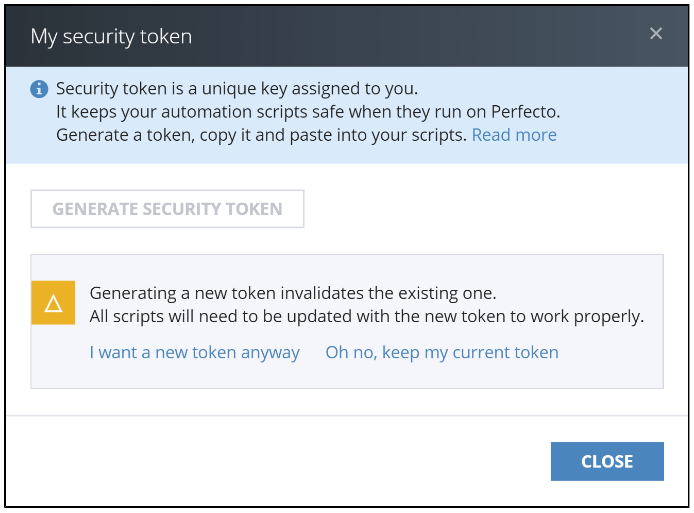

My security token

Security token is a unique key assigned to you.
It keeps your automation scripts safe when they run on Perfecto.
Generate a token, copy it and paste into your scripts. Read more

GENERATE SECURITY TOKEN

Your token will be displayed here

CLOSE


My security token

Security token is a unique key assigned to you.
It keeps your automation scripts safe when they run on Perfecto.
Generate a token, copy it and paste into your scripts. Read more

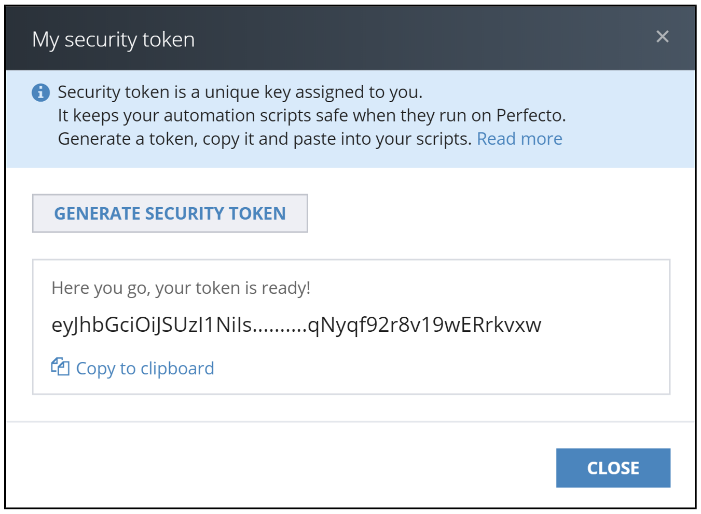
GENERATE SECURITY TOKEN

Generating a new token invalidates the existing one.
All scripts will need to be updated with the new token to work properly.

I want a new token anyway   Oh no, keep my current token

CLOSE

3. Click **Copy to clipboard**. Then paste it into any scripts that you want to run with Perfecto. See Use a security token in automation scripts below.


My security token

Security token is a unique key assigned to you.
It keeps your automation scripts safe when they run on Perfecto.
Generate a token, copy it and paste into your scripts. Read more

GENERATE SECURITY TOKEN

Here you go, your token is ready!
eyJhbGciOiJSUzI1NiIs..........qNyqf92r8v19wERrkvxw
Copy to clipboard

CLOSE

4. Click **Close**.

C | Select a device

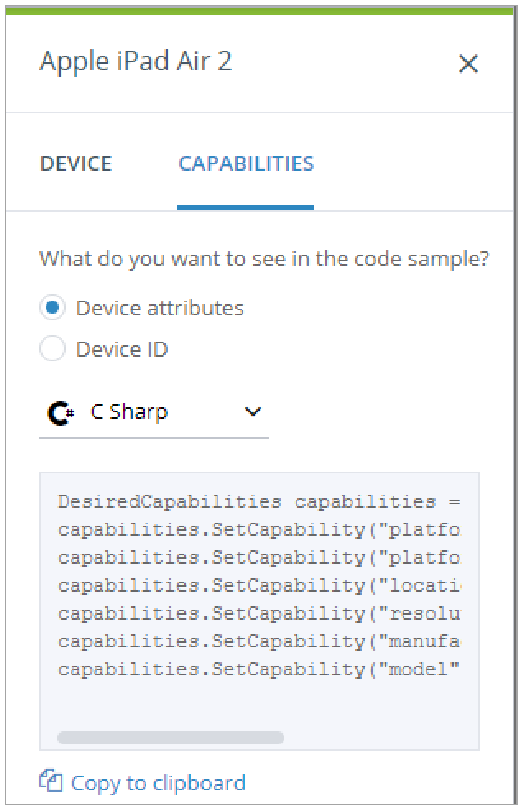
Use capabilities to select a device from the Perfecto lab. You can be as generic or specific as needed. In our script, we have only included the `platformName` capability (which specifies the device operating system), as shown in the following code snippet. For more information, see Define capabilities and Use capabilities to select a device.

```
DesiredCapabilities capabilities = new DesiredCapabilities
(browserName, "", Platform.ANY);
...
capabilities.setCapability("platformName", "Android");
```

You can use the **Manual Testing** view in the Perfecto UI to generate a code snippet with device-specific capabilities that you can then copy-paste into your script.

To generate capabilities (for mobile or web devices) through the Perfecto UI:

1. On the Perfecto landing page, under **Manual Testing**, click **Open Device**.
2. In the **Manual Testing** view, do the following to generate capabilities:
    1. In the list or tile view, click a device.
    2. In the details pane on the right, click the **Capabilities** tab.
    3. (Mobile device only) Depending on what you want to see in your code sample, select **Device attributes** or **Device ID**.



Apple iPad Air 2

What do you want to see in the code sample?

Device attributes [✓]

Device ID

C Sharp

```
DesiredCapabilities capabilities =
capabilities.SetCapability("platfo
capabilities.SetCapability("platfo
capabilities.SetCapability("locati
capabilities.SetCapability("resolu
capabilities.SetCapability("manufa
capabilities.SetCapability("model"
```

    4. From the drop-down list, select the programming language to use.
    5. The code snippet is updated automatically.

6. Click Copy to clipboard.
7. Paste into your existing script.

D | Provide the URL to connect to the Perfecto cloud

As part of creating an instance of the RemoteWebDriver, you need to supply the URL of your Perfecto cloud. Define the name of your Perfecto cloud as a system variable by passing your cloud name as a `-DcloudName=<<`*`cloud name`*`>>` from the CI/Maven system property while running the install goal of Maven, or simply hard-code your cloud name in the script.

The general structure of the URL string is as follows:

`"https://" + cloudName + "/perfectomobile.com/nexperience/perfectomobile/wd/hub"`

Here are the respective lines from our sample script:

```
String cloudName = System.getProperty("cloudName");
...
RemoteWebDriver driver = new RemoteWebDriver(new URL
("https://" + cloudName + ".perfectomobile.com/nexperience
/perfectomobile/wd/hub"), capabilities);
```

E | Create an instance of the reporting client

To get the most out of running your test in Perfecto, you need to create an instance of the Smart Reporting client (***ReportiumClient***). This will allow you to later retrieve and analyze the test report. The reporting client is responsible for gathering basic information about the test and transmitting it to the Smart Reporting system.

In our script, we show how to use the ***ReportiumClientFactory*** class' *createPerfectoReportiumClient()* method. Use the ***PerfectoExecutionContext*** class to supply the link to the factory class creating the client instance. Use the *withWebDriver()* method to supply the link of the driver instance.

Use the *build()* method to create the context object's instance and supply this to the *createPerfectoReportiumClient()* method when creating the ***ReportiumClient*** instance.

```
PerfectoExecutionContext perfectoExecutionContext;
if(System.getProperty("reportium-job-name") != null) {
                perfectoExecutionContext = new
PerfectoExecutionContext.PerfectoExecutionContextBuilder()
                                .withProject(new Project
("My Project", "1.0"))
                                .withJob(new Job(System.
getProperty("reportium-job-name") , Integer.parseInt
(System.getProperty("reportium-job-number"))))
                                .withContextTags("tag1")
                                .withWebDriver(driver)
                                .build();
} else {
                perfectoExecutionContext = new
PerfectoExecutionContext.PerfectoExecutionContextBuilder()
                                .withProject(new Project
("My Project", "1.0"))
                                .withContextTags("tag1")
                                .withWebDriver(driver)
                                .build();
}
ReportiumClient reportiumClient = new
ReportiumClientFactory().createPerfectoReportiumClient
(perfectoExecutionContext);
```

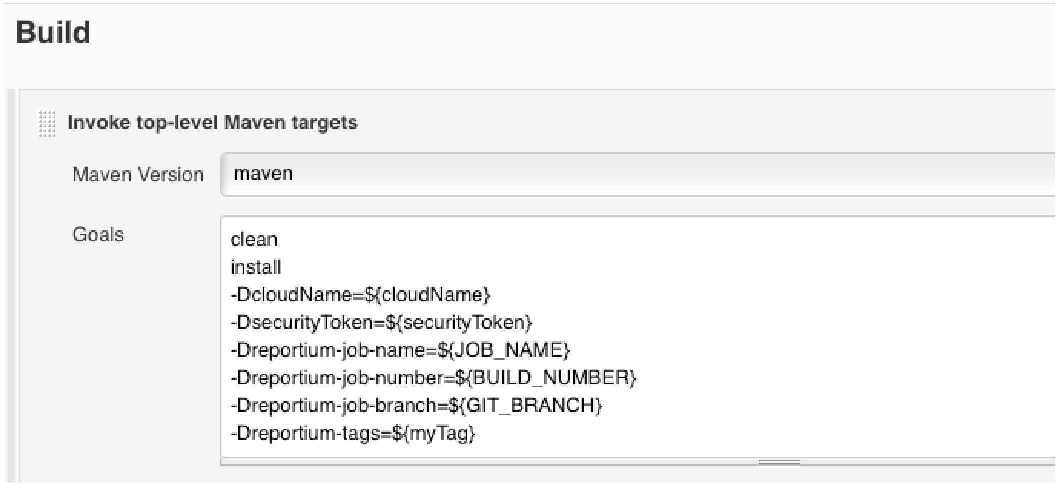
In addition to supplying the driver link, the context supports optional settings, such as adding:

- **Reporting tags:** Tags are used as a freestyle text for filtering the reports in the Reporting app. Use the *withContextTags()* method as shown in the following code snippet.
- **CI job information:** Job information is used to add your test runs to the CI Dashboard view. Use the *withJob()* method of the ***PerfectoExecutionContext*** instance, supplying the job name and job number, when creating the **ReportiumClient** instance.

  Our sample script uses system variables to fetch the `reportium-job-name` and the `reportium-job-number` values . Both system variables are sent from CI tools like Jenkins as a Maven `-D` system property. For example:

  `-Dreportium-job-name=${JOB_NAME} -Dreportium-job-number=${BUILD_NUMBER}`

  The following figure illustrates how the `reportium-job-name` and the `reportium-job-number` system variables get their value from Jenkins. These variables are passed as -D parameters for install goals of Maven.



Maven Version: maven

Goals:
```
clean
install
-DcloudName=${cloudName}
-DsecurityToken=${securityToken}
-Dreportium-job-name=${JOB_NAME}
-Dreportium-job-number=${BUILD_NUMBER}
-Dreportium-job-branch=${GIT_BRANCH}
-Dreportium-tags=${myTag}
```

Our script supports both local and CI-based executions (for more information, see Add reporting to Jenkins > Supply Maven or Ivy parameters).

F | Insert your test information

Replace the existing test code in the `try` bock with your own test information.

```
try {

     reportiumClient.testStart("Perfecto mobile web test",
new TestContext("tag2", "tag3"));
                        reportiumClient.stepStart("browser
navigate to perfecto");
                                driver.get("https://www.
perfecto.io");
                        reportiumClient.stepEnd();

                        reportiumClient.stepStart("Verify
title");
                                String aTitle = driver.
getTitle();
                                System.out.println(aTitle);
                                //compare the actual title
with the expected title
                                if (!aTitle.equals("Web &
Mobile App Testing | Continuous Testing | Perfecto"))
                                        throw new
RuntimeException("Title is mismatched");
                        reportiumClient.stepEnd();


            ...
        }
```

In our example, the test is separated into logical groupings of actions as logical steps. Each step is labeled and appears in the Execution Report together with the component actions. The beginning of each logical step is indicated with the *ste pStart()* method, providing the label of the step, and the end of each logical step with the *stepEnd()* method (for the report).

G | Stop the test

To end the test, supply an indication of the final outcome of the test by generating a ***TestResult*** instance.

```
//STOP TEST
    TestResult testResult = TestResultFactory.
createSuccess();
    reportiumClient.testStop(testResult);

} catch (Exception e) {
                        TestResult testResult =
TestResultFactory.createFailure(e);
                        reportiumClient.testStop
(testResult);
                        e.printStackTrace();
}
```

In our example, the **createSuccess** method notifies the reporting server that the test resulted in a successful status. The **createFailure** method notifies the reporting server that the test resulted in an unsuccessful status and supports adding a notification message that is displayed in the test report. Our script also provides a failure reason, but this is optional. To learn more about failure reasons in reporting, see Work with failure reasons.

Last, make sure to close and quit the RemoteWebDriver and retrieve the Smart Reporting URL for the generated test report.

```
finally {
                        driver.close();
                        driver.quit();
                        // Retrieve the URL to the
DigitalZoom Report
                        String reportURL = reportiumClient.
getReportUrl();
                        System.out.println(reportURL);
        }
```

Also in this section:

- Use the Selenium Wait functions
- Visual analysis with RemoteWebDriver