Appium

How to run your Java Appium script in Perfecto

On this page:

This section provides instructions on how to run Appium Driver tests with Java in Perfecto. It assumes that you:

- Are familiar with Appium
- 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: `LocalAppium.java` and `PerfectoAppium.java`

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

Prerequisites

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

- Java Development Kit 1.8
- 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)
- Android Studio
- Local Appium server, running at http://127.0.0.1, port 4723
  For more information, see this introduction to Appium.
- The Samsung Calculator app from the Google Play on your Perfecto Samsung Galaxy device

1 | Get started

The starting point is `LocalAppium.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 connects to an emulator, opens the calculator app on an Android device, and performs an addition validation (1+1=2).

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. Run the script to make sure it executes without any issues.

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, Smart Reporting information, and test data. We also want to make sure that the script exits gracefully.

The updated script is called `PerfectoAppium.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 search for the following line in `PerfectoAppium.java` and replace `<<security token>>` with your Perfecto security token.

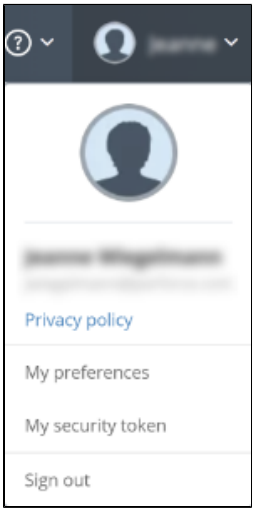
```
String securityToken = "<<security token>>";
```

Alternatively, you can pass the token as a Maven property, as follows:

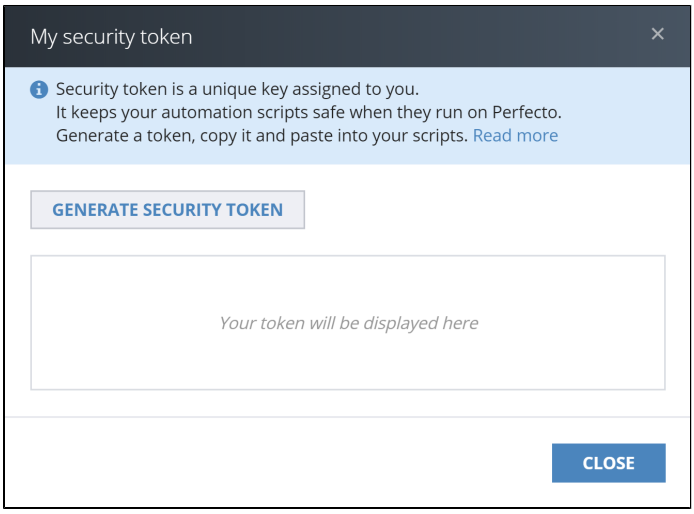
```
-DsecurityToken=<<SECURITY TOKEN>>
```

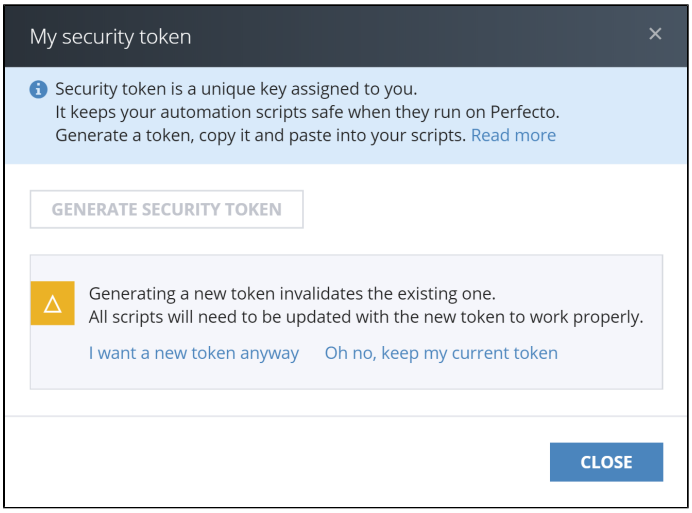
To generate a security token:

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**.

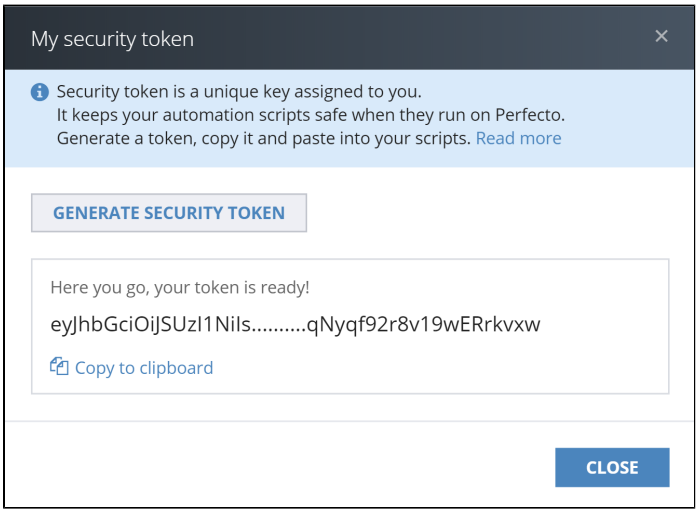


2. In the **My security token** form, click **Generate Security Token**.





3. Click **Copy to clipboard**. Then paste it into any scripts that you want to run with Perfecto.



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 included the these capabilities, as shown in the following code snippet:

- `model`: The device model
- `openDeviceTimeout`: The timeout, in minutes, to wait for a specific device in case it is not available at the start of the script (use with caution)
- `appPackage`: The Java package of the Android app you want to run

For more information on capabilities, see:

- Define capabilities
- Supported Appium capabilities
- Use capabilities to select a device

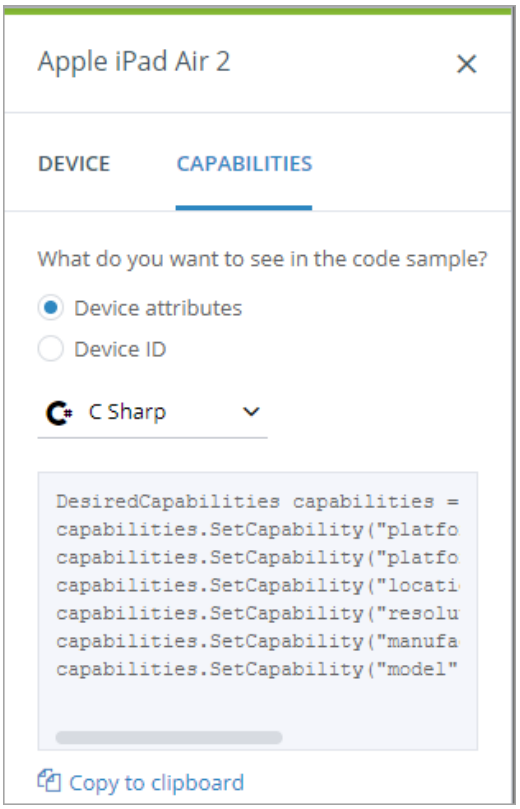
```
DesiredCapabilities capabilities = new DesiredCapabilities(browserName, "", Platform.ANY);
...
capabilities.setCapability("model", "Galaxy.*");
capabilities.setCapability("openDeviceTimeout", 2);
capabilities.setCapability("appPackage", "com.sec.android.app.popupcalculator");
```

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:
    a. In the list or tile view, click a device.
    b. In the details pane on the right, click the **Capabilities** tab.

c. (Mobile device only) Depending on what you want to see in your code sample, select **Device attributes** or **Device ID**.



d. From the drop-down list, select the programming language to use.
e. The code snippet is updated automatically.
f. Click Copy to clipboard.
g. 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. Search for the following line in `PerfectoAppium.java` and replace `<<cloud name>>` with the name of your Perfecto cloud (for example `demo`).

```
String cloudName = "<<cloud name>>";
```

The general structure of the URL string is as follows:

```
"https://" + Utils.fetchCloudName(cloudName) + ".perfectomobile.com/nexperience/perfectomobile/wd/hub"
```

Here is the respective line from our sample script:

```
WebDriver driver = new RemoteWebDriver(new URL("https://" + Utils.fetchCloudName(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);
```

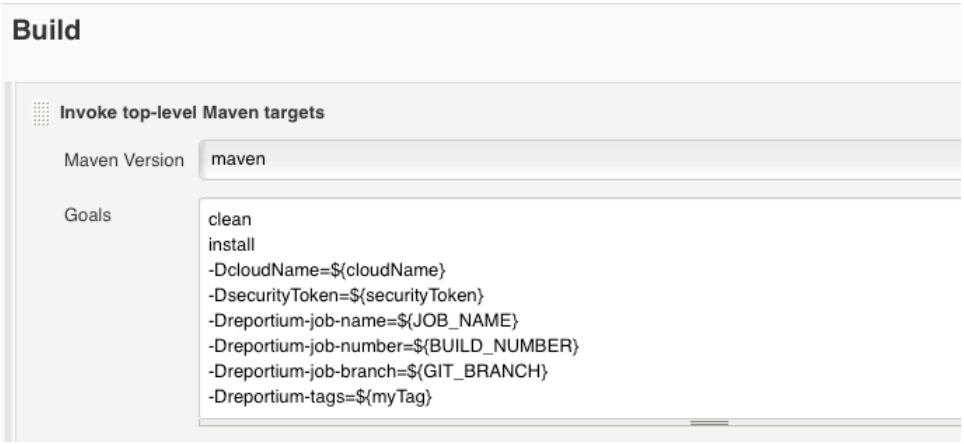
In addition to supplying the driver link, the context supports optional settings, such as adding:

- **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.



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 block

```
try {
        reportiumClient.testStart("My Calculator Test", new TestContext("tag2", "tag3"));
        reportiumClient.stepStart("Perform addition");
        driver.findElement(By.xpath("//*[contains(@resource-id,'1')]")).click();
        driver.findElement(By.xpath("//*[contains(@resource-id,'add')]")).click();
        driver.findElement(By.xpath("//*[contains(@resource-id,'1')]")).click();
        driver.findElement(By.xpath("//*[contains(@resource-id,'equal')]")).click();
        reportiumClient.stepEnd();
        reportiumClient.stepStart("Verify Total");
        WebElement results=driver.findElement(By.xpath("//*[contains(@class,'EditText')]"));
        if (!results.getText().equals("2"))
                throw new RuntimeException("Actual calculated number is : \"+results.
getText() + \". It did not match with expected value: 2");
        reportiumClient.stepEnd();
                //STOP TEST
        TestResult testResult = TestResultFactory.createSuccess();
        reportiumClient.testStop(testResult);
}
```

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 *stepStart()* meth od, providing the label of the step, and the end of each logical step with the *stepEnd()* method (for the report).

G | Close Smart Reporting and driver

To close Smart Reporting and the driver, 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.

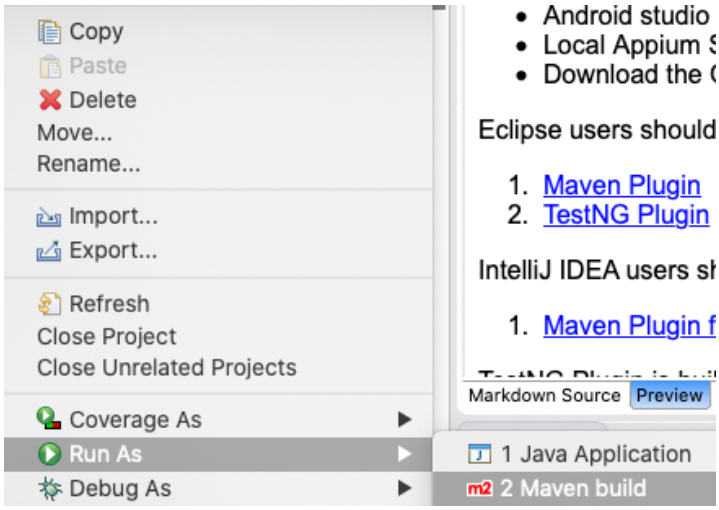
Last, make sure to close and quit the RemoteWebDriver and retrieve the Smart Reporting URL for the generated test report.

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

H | Execute the test

Now that your script is ready to run, perform the following steps to execute the test.

1. Right-click the `pom.xml` file and select **Run As > Maven build**.

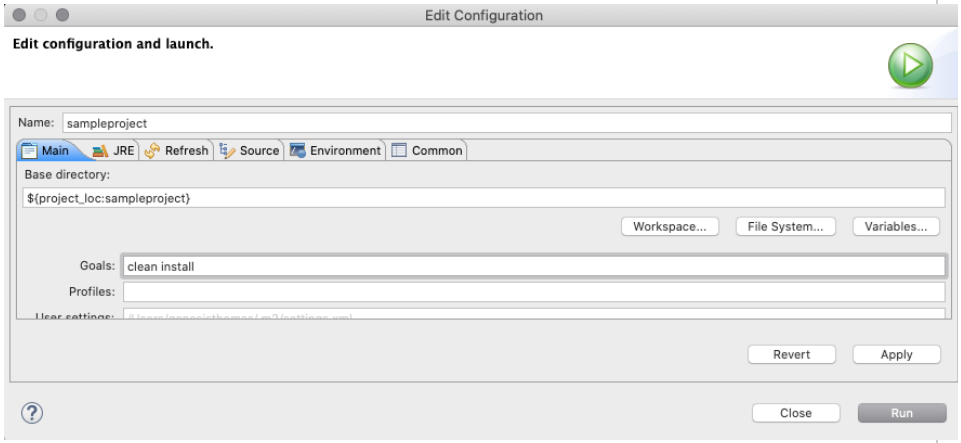


2. In the **Edit Configuration** form, on the **Main** tab:

    a. In the **Goals** field, enter the following Maven goals:

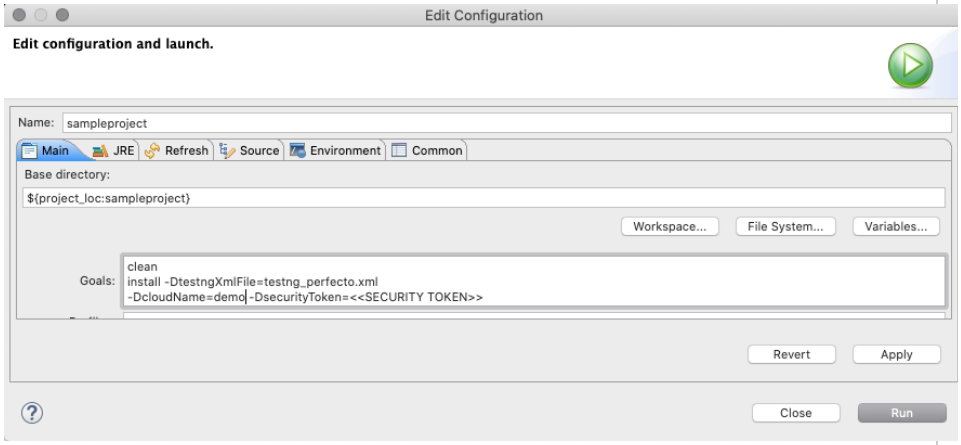
    - If the credentials are hard coded:

```
clean install
```



    - If the credentials are passed as parameters:

```
clean
install
-DcloudName=${cloudName}
-DsecurityToken=${securityToken}
-DtestngXmlFile=testng_perfecto.xml
```



b. Click **Run**.

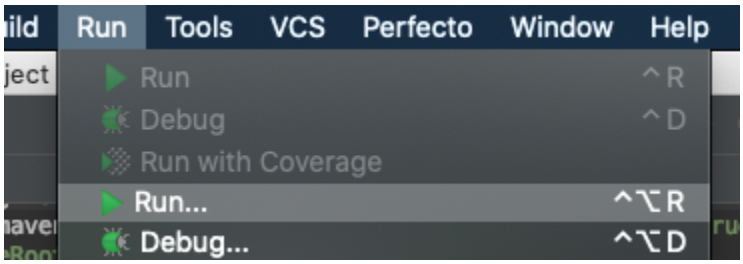
The following image shows sample run results.

```
Web & Mobile App Testing | Continuous Testing | Perfecto
https://ps.app.perfectomobile.com/reporting/library?externalId[0]=[redacted]@perfectomobile.com_RemoteWebDri
https://ps.app.perfectomobile.com/reporting/library?externalId[0]=[redacted]@perfectomobile.com_RemoteWebDri
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 60.334 sec - in TestSuite

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ sampleproject ---
[INFO] Building jar: /Users/[redacted]/eclipse-workspace/sampleproject/target/sampleproject-0.0.1-SNAPS
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ sampleproject ---
[INFO] Installing /Users/[redacted]/eclipse-workspace/sampleproject/target/sampleproject-0.0.1-SNAPSHOT
[INFO] Installing /Users/[redacted]/eclipse-workspace/sampleproject/pom.xml to /Users/[redacted]/.m2
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 01:02 min
[INFO] Finished at: 2020-02-17T11:38:52+05:30
[INFO] ------------------------------------------------------------------------
```

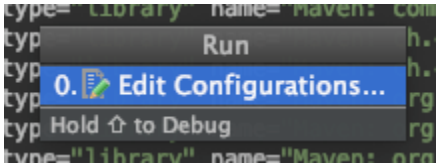
The first time you run this test, you need to edit the run configuration.

To run this test for the first time:

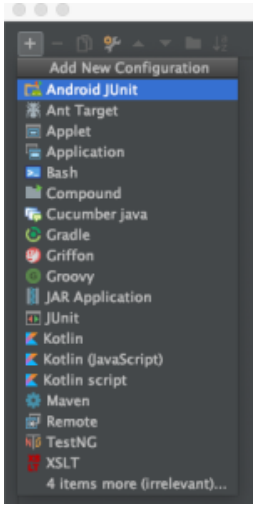
1. From the **Run** menu, select **Run**.



2. Select **Edit Configurations**.

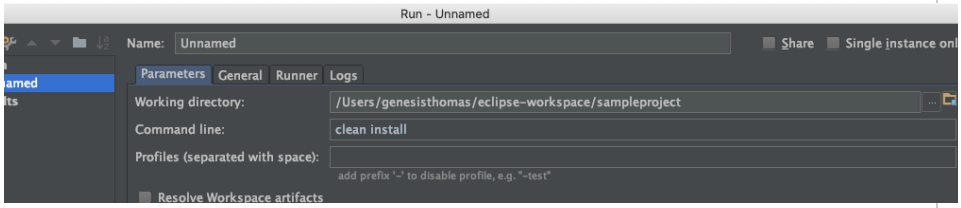


3. Select the plus sign and select **Maven**.



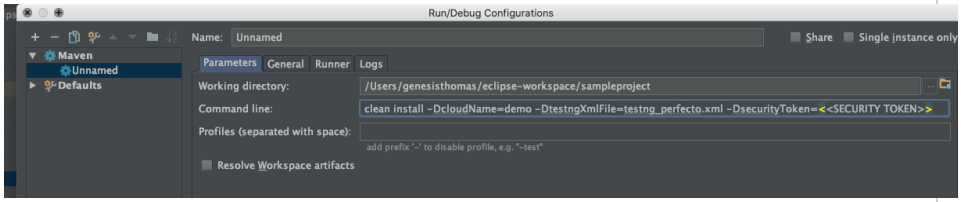
4. In the **Run** form, do the following:
    a. In the **Name** field, enter a descriptive name for the run.
    b. On the **Parameters** tab, in the **Command line** field, enter the following command, depending on whether credentials are hard-coded or passed as parameters.
        - If credentials are hard-coded:

```
clean
install
```



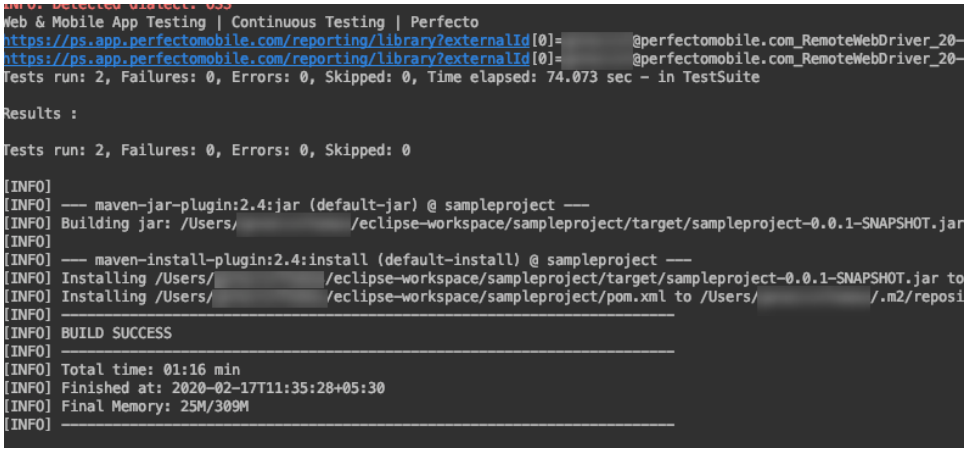
        - If credentials are passed as parameters:

```
clean
install
-DcloudName=${cloudName}
-DsecurityToken=${securityToken}
-DtestngXmlFile=testng_perfecto.xml
```



5. Click **Run**.

    The following image shows sample run results.



To perform subsequent runs:

- From the **Run** menu, select **Run > *<Name>***.

> **Note:** You can pass the TestNG file name as a Maven property to execute specific TestNG files, as follows:
>
> ```
> -DtestngXmlFile=<<testng file name>>.xml
> ```
>
> `testng_perfecto.xml` is configured by default, and it is not mandatory to pass it as a Maven property. However, you can override it by passing your preferred TestNG name as a Maven property, such as `-DtestingXmlFile=testng.xml`, to execute all 4 tests in parallel.

Also in this section:

- Install and start an application
- Supported Appium capabilities
- Appium/XCUITest scripts on iOS
- New architecture for Appium web and hybrid testing on iOS
- New architecture for Appium testing on Android
- UIAutomation Selector support in Appium