

Integration with Gradle



- [Overview](#)
- [How to use](#)
 - [Configuration](#)
 - [Examples](#)
 - [JUnit4](#)
 - [JUnit5](#)
 - [JUnit5 with additional information](#)
 - [TestNG](#)
 - [TestNG with additional information](#)
- [Tips](#)
- [References](#)

Overview

Gradle is an open-source build tool that provides flexibility, and performance through task parallelization. Gradle is known in Java projects, as an alternative to Maven, but also supports other languages. Find out more about its [features](#) on Gradle's site.

Integration with Xray is possible using Xray APIs, namely the [REST API for importing test results](#).

You can use this integration to upload test results to Xray, in one of the supported test reports/formats, so that you have visibility of your test results from your pipeline in Jira.

The examples detailed on this page are available in a [GitHub repo](#). They are not exhaustive; please consider them just as a reference and feel free to adapt them to your needs.

How to use

The following examples were made using Gradle 7.4.2 and JDK 11.

In order to invoke the REST API, we'll use the `curl` utility that is available on Unix, Linux, macOS, and other systems.

We don't need to install any specific Gradle plugin for this.

To import the results we use a custom Gradle task. We can then either invoke it by using `gradle` or `gradlew` (i.e., [Gradle Wrapper](#)).

Configuration

Our task will make use of several variables.

In Gradle, we can define some variables within the tasks themselves, and they will be instantiated during the configuration phase.

Another option is to use a `gradle.properties` file, such as the following example.

example of gradle.properties

```
# Jira server/DC specifics
jiraBaseUrl=https://jiraserver.local
jiraUsername=someuser
jiraPassword=somepass

reportFormat=junit
projectKey=CALC
version=v1.0
revision=123
testPlanKey=CALC-726
testExecKey=
testEnvironment=
```

Gradle will pick it and these variables will be available for the task(s) that we will use/implement.

Examples

JUnit4

In this [example](#), the tests are implemented using Java + JUnit 4.13.x.

We use a configuration file to define the REST API specifics (i.e., Jira server/DC URL along with a Jira username and password) and some parameters to identify, for example, the target project and version/release of the SUT.

In this case, we're using the standard JUnit endpoint (more info on the endpoint and the supported parameters is available in [Import Execution Results - REST](#)).

gradle.properties

```
# Jira server/DC specifics
jiraBaseUrl=https://jiraserver.local
jiraUsername=someuser
jiraPassword=somepass

reportFormat=junit
projectKey=CALC
version=v1.0
revision=123
testPlanKey=CALC-726
testExecKey=
testEnvironment=
```

We then generate a "standard" JUnit XML report and use a custom task to perform the REST API request that submits the results to Xray.

The following example shows a `build.gradle` file with a custom task named `importJUnitResultsToXrayDC` where we implement the logic to push the results to Xray.

The `test` task uses the JUnit4 runner by declaring `useJUnit()`.

build.gradle

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

jar {
    archiveBaseName = 'tutorial-gradle-junit4-basic'
    archiveVersion = '0.1.0'
```

```

}

sourceCompatibility = 1.8
targetCompatibility = 1.8

tasks.withType(JavaCompile) {
    options.compilerArgs += '-Xlint:deprecation'
}

test {
    useJUnit()
    reports {
        // destination = file('build/test-results/folder')
        junitXml.required = true
        html.required = false
    }
    ignoreFailures = true
}

dependencies {
    testImplementation 'junit:junit:4.13.2'
    testImplementation 'org.hamcrest:hamcrest-library:2.2'
}

configurations.all {
    resolutionStrategy.cacheChangingModulesFor 0, 'seconds'
    resolutionStrategy.cacheDynamicVersionsFor 0, 'seconds'
}

task importJUnitResultsToXrayDC(type: Exec) {
    description 'Imports Junit test results to Xray Test Management for Jira DC.'
    dependsOn 'test'

    // the following variables can be defined locally or in gradle.properties
    // - jiraBaseUrl, jiraUsername, jiraPassword
    // - reportFormat, projectKey version, revision, testPlanKey, testExecKey, testEnvironment

    def reportFile = "build/test-results/test/TEST-app.getxray.java.CalcTest.xml"

    def url = "${jiraBaseUrl}/rest/raven/2.0/import/execution/${reportFormat}?"
    if (projectKey?.trim()) {
        url += "&projectKey=${projectKey}"
    }
    if (version?.trim()) {
        url += "&fixVersion=${version}"
    }
    if (revision?.trim()) {
        url += "&revision=${revision}"
    }
    if (testPlanKey?.trim()) {
        url += "&testPlanKey=${testPlanKey}"
    }
    if (testExecKey?.trim()) {
        url += "&testExecKey=${testExecKey}"
    }
    if (testEnvironment?.trim()) {
        url += "&testEnvironments=${testEnvironment}"
    }

    commandLine 'curl', '--fail-with-body', '-H', 'Content-Type: multipart/form-data', '-u',
    "${jiraUsername}:${jiraPassword}", '-F', "file=@${reportFile}", url

    //store the output instead of printing to the console:
    standardOutput = new ByteArrayOutputStream()
    ignoreExitValue = false //true

    doLast {
        if (execResult.getExitValue() != 0) {
            println "ERROR: problem importing results to Xray"
        } else {
            println "Results imported to Xray!"
        }
    }
}

```

```

        }
        println standardOutput.toString()
    }
}

```

To run the tests and import them to Xray we can run Gradle as usual and add the name of the task we created earlier.

```
gradle clean compileJava test importJUnitResultsToXrayDC
```

In Xray, a Test Execution will be created accordingly.

The screenshot shows the Xray interface with a test execution details page and a test results table.

Test Execution Details:

- Type: Test Execution
- Status: IN PROGRESS (View Workflow)
- Priority: Trivial
- Affects Version/s: None
- Labels: None
- Test Plan: CALC-726
- Test Environments: None
- Revision: 123

Description:
Execution results imported from external source

Tests:

Rank	Key	Summary	Test Type	#Req	#Def	Assignee	Status
4	CALC-757	CanDivide	Generic	0	0	Xpand IT Admin	PASS
2	CALC-749	CanAddNumbers	Generic	0	0	Xpand IT Admin	FAIL
1	CALC-744	CanDoStuff	Generic	0	0	Xpand IT Admin	PASS
3	CALC-743	subtraction test	Generic	0	0	Xpand IT Admin	PASS
5	CALC-742	CanMultiply	Generic	0	0	Xpand IT Admin	PASS

Total Tests: 5

JUnit5

In this [example](#), the tests are implemented using Java + JUnit 5.8.x.

We use a configuration file to define the REST API specifics (i.e., Jira server/DC URL along with a Jira username and password) and some parameters to identify, for example, the target project and version/release of the SUT.

In this case, we're using the standard JUnit endpoint (more info on the endpoint and the supported parameters is available in [Import Execution Results - REST](#)).

gradle.properties

```
# Jira server/DC specifics
jiraBaseUrl=https://jiraserver.local
jiraUsername=someuser
jiraPassword=somepass

reportFormat=junit
projectKey=CALC
version=v1.0
revision=123
testPlanKey=CALC-726
testExecKey=
testEnvironment=
```

We then generate a "standard" JUnit XML report and use a custom task to perform the REST API request that submits the results to Xray.

The following example shows a `build.gradle` file with a custom task named `importJUnitResultsToXrayDC` where we implement the logic to push the results to Xray.

The test task uses the JUnit5 platform runner by declaring `useJUnitPlatform()`.

build.gradle

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

jar {
    archiveBaseName = 'tutorial-gradle-junit5-basic'
    archiveVersion = '0.1.0'
}

sourceCompatibility = 1.8
targetCompatibility = 1.8

test {
    useJUnitPlatform()
    reports {
        // destination = file('build/test-results/folder')
        junitXml.required = true
        html.required = false
    }
    ignoreFailures = true
}

dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.8.1'
    testImplementation 'org.hamcrest:hamcrest-library:2.2'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.8.1'
}

configurations.all {
    resolutionStrategy.cacheChangingModulesFor 0, 'seconds'
    resolutionStrategy.cacheDynamicVersionsFor 0, 'seconds'
}

task importJUnitResultsToXrayDC(type: Exec) {
    description 'Imports Junit test results to Xray Test Management for Jira DC.'
    dependsOn 'test'

    // the following variables can be defined locally or in gradle.properties
    // - jiraBaseUrl, jiraUsername, jiraPassword
    // - reportFormat, projectKey version, revision, testPlanKey, testExecKey, testEnvironment
```

```

def reportFile = "build/test-results/test/TEST-app.getxray.java.CalcTest.xml"

def url = "${jiraBaseUrl}/rest/raven/2.0/import/execution/${reportFormat}?"
if (projectKey?.trim()) {
    url += "&projectKey=${projectKey}"
}
if (version?.trim()) {
    url += "&fixVersion=${version}"
}
if (revision?.trim()) {
    url += "&revision=${revision}"
}
if (testPlanKey?.trim()) {
    url += "&testPlanKey=${testPlanKey}"
}
if (testExecKey?.trim()) {
    url += "&testExecKey=${testExecKey}"
}
if (testEnvironment?.trim()) {
    url += "&testEnvironments=${testEnvironment}"
}

commandLine 'curl', '--fail-with-body', '-H','Content-Type: multipart/form-data', '-u',
"${jiraUsername}:${jiraPassword}", '-F', "file=@${reportFile}", url

//store the output instead of printing to the console:
standardOutput = new ByteArrayOutputStream()
ignoreExitValue = false //true

doLast {
    if (execResult.getExitValue() != 0) {
        println "ERROR: problem importing results to Xray"
    } else {
        println "Results imported to Xray!"
    }
    println standardOutput.toString()
}
}

```

To run the tests and import them to Xray we can run gradle as usual and add the name of the task we created earlier.

```
gradle clean compileJava test importJUnitResultsToXrayDC
```

In Xray, a Test Execution will be created accordingly.

CALC / CALC-758
Execution results - TEST-app.getxray.java.CalcTest.xml - [1657214659860]

[Edit](#) [Comment](#) [Assign](#) [More](#) [Stop Progress](#) [Resolve Issue](#) [Close Issue](#) [Admin](#)

Details

Type:	<input checked="" type="checkbox"/> Test Execution	Status:	IN PROGRESS (View Workflow)
Priority:	<input type="radio"/> Trivial	Resolution:	Unresolved
Affects Version/s:	None	Fix Version/s:	v1.0
Labels:	None		
Test Plan:	CALC-726		
Test Environments:	None		
Revision:	123		

Description
Execution results imported from external source

Tests

[Add Tests](#) [...](#)

Overall Execution Status

4 PASS 1 FAIL

Total Tests: 5

[Filter\(s\)](#)

Rank	Key	Summary	Test Type	#Req	#Def	Assignee	Status	...
4	CALC-760	CanDivide()	Generic	0	0	Xpand IT Admin	PASS	...
2	CALC-759	CanAddNumbers()	Generic	0	0	Xpand IT Admin	FAIL	...
5	CALC-723	CanMultiply()	Generic	0	0	Xpand IT Admin	PASS	...
3	CALC-722	CanSubtract()	Generic	0	0	Xpand IT Admin	PASS	...
1	CALC-721	CanDoStuff()	Generic	0	0	Xpand IT Admin	PASS	...

Show 100 entries Columns ▾

Showing 1 to 5 of 5 entries First Previous [1](#) Next Last

JUnit5 with additional information

In this [example](#), the tests are implemented using Java + JUnit 5.8.x.

i In this case, we'll use the [enhanced capabilities that Xray provides for JUnit](#) allowing you to provide additional information during the execution of the tests, including screenshots and others; to do so, we'll use the [xray-junit-extensions](#) package.

We use a configuration file to define the REST API specifics (i.e., Jira server/DC URL along with a Jira username and password) and some parameters to identify, for example, the target project and version/release of the SUT.

In this case, we're using the standard JUnit endpoint (more info on the endpoint and the supported parameters is available in [Import Execution Results - REST](#)).

gradle.properties

```
# Jira server/DC specifics
jiraBaseUrl=https://jiraserver.local
jiraUsername=someuser
jiraPassword=somepass

reportFormat=junit
projectKey=CALC
version=v1.0
revision=123
testPlanKey=CALC-726
testExecKey=
testEnvironment=
```

We then generate a "standard" JUnit XML report and use a custom task to perform the REST API request that submits the results to Xray.

The following example shows a `build.gradle` file with a custom task named `importJUnitResultsToXrayDC` where we implement the logic to push the results to Xray.

The test task uses the JUnit5 platform runner by declaring `useJUnitPlatform()`.

build.gradle

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

jar {
    archiveBaseName = 'tutorial-gradle-junit5-enhanced'
    archiveVersion = '0.1.0'
}

sourceCompatibility = 1.8
targetCompatibility = 1.8

test {
    useJUnitPlatform()
    reports {
        // destination = file('build/test-results/folder')
        junitXml.required = true
        html.required = false
    }
    ignoreFailures = true
}

dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.8.1'
    testImplementation 'org.hamcrest:hamcrest-library:2.2'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.8.1'
    testImplementation 'app.getxray:xray-junit-extensions:0.6.1'
}

configurations.all {
    resolutionStrategy.cacheChangingModulesFor 0, 'seconds'
    resolutionStrategy.cacheDynamicVersionsFor 0, 'seconds'
}

task importJUnitResultsToXrayDC(type: Exec) {
    description 'Imports Junit test results to Xray Test Management for Jira DC.'
    dependsOn 'test'

    // the following variables can be defined locally or in gradle.properties
    // - jiraBaseUrl, jiraUsername, jiraPassword
    // - reportFormat, projectKey version, revision, testPlanKey, testExecKey, testEnvironment

    def reportFile = "reports/TEST-junit-jupiter.xml"

    def url = "${jiraBaseUrl}/rest/raven/2.0/import/execution/${reportFormat}?"
    if (projectKey?.trim()) {
        url += "&projectKey=${projectKey}"
    }
    if (version?.trim()) {
        url += "&fixVersion=${version}"
    }
    if (revision?.trim()) {
        url += "&revision=${revision}"
    }
    if (testPlanKey?.trim()) {
        url += "&testPlanKey=${testPlanKey}"
    }
    if (testExecKey?.trim()) {
        url += "&testExecKey=${testExecKey}"
    }
    if (testEnvironment?.trim()) {
```

```

        url += "&testEnvironments=${testEnvironment}"
    }

    commandLine 'curl', '--fail-with-body', '-H','Content-Type: multipart/form-data', '-u',
"${jiraUsername}:${jiraPassword}", '-F', "file=@${reportFile}", url

    //store the output instead of printing to the console:
    standardOutput = new ByteArrayOutputStream()
    ignoreExitValue = false //true

    doLast {
        if (execResult.getExitValue() != 0) {
            println "ERROR: problem importing results to Xray"
        } else {
            println "Results imported to Xray!"
        }
        println standardOutput.toString()
    }
}

```

To run the tests and import them to Xray we can run Gradle as usual and add the name of the task we created earlier.

```
gradle clean compileJava test importJUnitResultsToXrayDC
```

In Xray, a Test Execution will be created accordingly.

The screenshot shows the Xray interface with a test execution details page. At the top, there's a header with a profile icon, the project name 'CALC / CALC-761', and the title 'Execution results - TEST-junit-jupiter.xml - [1657214841741]'. Below the header are several buttons: Edit, Comment, Assign, More, Stop Progress, Resolve Issue, Close Issue, and Admin. The main content area has sections for 'Details' and 'Description'. Under 'Details', there are fields for Type (Test Execution), Priority (Trivial), Affects Version/s (None), Labels (None), Test Plan (CALC-726), Test Environments (None), Revision (123), Status (IN PROGRESS), Resolution (Unresolved), and Fix Version/s (v1.0). The 'Description' section contains the note 'Execution results imported from external source'. Below these, there's a 'Tests' section with a 'Add Tests' button and a table. The table has columns: Rank, Key, Summary, Test Type, #Req, #Def, Assignee, and Status. It lists five test cases: CALC-757 (CanDivide, Generic, 0 reqs, 0 defs, assigned to Xpand IT Admin, PASS), CALC-744 (CanDoStuff, Generic, 0 reqs, 0 defs, assigned to Xpand IT Admin, PASS), CALC-743 (subtraction test, Generic, 0 reqs, 0 defs, assigned to Xpand IT Admin, PASS), CALC-742 (CanMultiply, Generic, 0 reqs, 0 defs, assigned to Xpand IT Admin, PASS), and CALC-739 (CanAddNumbers(), Generic, 1 req, 0 defs, assigned to Xpand IT Admin, FAIL). The overall execution status bar at the bottom shows 4 PASS and 1 FAIL.

Rank	Key	Summary	Test Type	#Req	#Def	Assignee	Status
2	CALC-757	CanDivide	Generic	0	0	Xpand IT Admin	PASS
4	CALC-744	CanDoStuff	Generic	0	0	Xpand IT Admin	PASS
3	CALC-743	subtraction test	Generic	0	0	Xpand IT Admin	PASS
1	CALC-742	CanMultiply	Generic	0	0	Xpand IT Admin	PASS
5	CALC-739	CanAddNumbers()	Generic	1	0	Xpand IT Admin	FAIL

TestNG

In this [example](#), the tests are implemented using Java + TestNG 7.6.x.

We use a configuration file to define the REST API specifics (i.e., Jira server/DC URL along with a Jira username and password) and some parameters to identify, for example, the target project and version/release of the SUT.

In this case, we're using the standard TestNG endpoint (more info on the endpoint and the supported parameters is available in [Import Execution Results - REST](#)).

gradle.properties

```
# Jira server/DC specifics
jiraBaseUrl=https://jiraserver.local
jiraUsername=someuser
jiraPassword=somepass

reportFormat=testng
projectKey=CALC
version=v1.0
revision=123
testPlanKey=CALC-726
testExecKey=
testEnvironment=
```

We then generate a "standard" TestNG XML report and use a custom task to perform the REST API request that submits the results to Xray.

The following example shows a `build.gradle` file with a custom task named `importTestNGResultsToXrayDC` where we implement the logic to push the results to Xray.

The test task uses the TestNG runner by declaring `useTestNG()`.

build.gradle

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

jar {
    archiveBaseName = 'tutorial-gradle-testng-basic'
    archiveVersion = '0.1.0'
}

sourceCompatibility = 11
targetCompatibility = 11

test {
    useTestNG() {
        // report generation delegated to TestNG library:
        useDefaultListeners = true
    }
    reports {
        junitXml.required = false
        html.required = false
    }
    ignoreFailures = true
}

dependencies {
    testImplementation 'org.testng:testng:7.6.1'
}

configurations.all {
    resolutionStrategy.cacheChangingModulesFor 0, 'seconds'
    resolutionStrategy.cacheDynamicVersionsFor 0, 'seconds'
}

task importTestNGResultsToXrayDC(type: Exec) {
    description 'Imports TestNG test results to Xray Test Management for Jira DC.'
```

```

dependsOn 'test'

// the following variables can be defined locally or in gradle.properties
// - jiraBaseUrl, jiraUsername, jiraPassword
// - reportFormat, projectKey version, revision, testPlanKey, testExecKey, testEnvironment

def reportFile = "build/reports/tests/test/testng-results.xml"

def url = "${jiraBaseUrl}/rest/raven/2.0/import/execution/${reportFormat}?"
if (projectKey?.trim()) {
    url += "&projectKey=${projectKey}"
}
if (version?.trim()) {
    url += "&fixVersion=${version}"
}
if (revision?.trim()) {
    url += "&revision=${revision}"
}
if (testPlanKey?.trim()) {
    url += "&testPlanKey=${testPlanKey}"
}
if (testExecKey?.trim()) {
    url += "&testExecKey=${testExecKey}"
}
if (testEnvironment?.trim()) {
    url += "&testEnvironments=${testEnvironment}"
}

commandLine 'curl', '--fail-with-body', '-H','Content-Type: multipart/form-data', '-u',
"${jiraUsername}:${jiraPassword}", '-F', "file=@${reportFile}", url

//store the output instead of printing to the console:
standardOutput = new ByteArrayOutputStream()
ignoreExitValue = false //true

doLast {
    if (execResult.getExitValue() != 0) {
        println "ERROR: problem importing results to Xray"
    } else {
        println "Results imported to Xray!"
    }
    println standardOutput.toString()
}
}
}

```

To run the tests and import them to Xray we can run gradle as usual and add the name of the task we created earlier.

```
gradle clean compileJava test importTestNGResultsToXrayDC
```

In Xray, a Test Execution will be created accordingly.

CALC / CALC-762
Execution results - testng-results.xml - [1657214987754]

[Edit](#) [Comment](#) [Assign](#) [More](#) [Stop Progress](#) [Resolve Issue](#) [Close Issue](#) [Admin](#)

Details

Type:	<input checked="" type="checkbox"/> Test Execution	Status:	IN PROGRESS (View Workflow)
Priority:	<input type="radio"/> Trivial	Resolution:	Unresolved
Affects Version/s:	None	Fix Version/s:	v1.0
Labels:	None		
Test Plan:	CALC-726		
Test Environments:	None		
Revision:	123		

Description
Execution results imported from external source

Tests

[Add Tests](#) [...](#)

Overall Execution Status

4 PASS 1 FAIL

Total Tests: 5

[Filter\(s\)](#)

[Apply Rank](#)

Rank	Key	Summary	Test Type	#Req	#Def	Assignee	Status	...
2	CALC-757	CanDivide	Generic	0	0	Xpand IT Admin	PASS	...
1	CALC-749	CanAddNumbers	Generic	0	0	Xpand IT Admin	FAIL	...
3	CALC-744	CanDoStuff	Generic	0	0	Xpand IT Admin	PASS	...
5	CALC-743	subtraction test	Generic	0	0	Xpand IT Admin	PASS	...
4	CALC-742	CanMultiply	Generic	0	0	Xpand IT Admin	PASS	...

Showing 1 to 5 of 5 entries

First Previous [1](#) Next Last

TestNG with additional information

In this [example](#), the tests are implemented using Java + TestNG 7.6.x.

i In this case, we'll use the [enhanced capabilities that Xray provides for TestNG](#) allowing you to provide additional information during the execution of the tests, including screenshots and others; to do so, we'll use the [xray-testng-extensions](#) package.

We use a configuration file to define the REST API specifics (i.e., Jira server/DC URL along with a Jira username and password) and some parameters to identify, for example, the target project and version/release of the SUT.

In this case, we're using the standard TestNG endpoint (more info on the endpoint and the supported parameters is available in [Import Execution Results - REST](#)).

gradle.properties

```
# Jira server/DC specifics
jiraBaseUrl=https://jiraserver.local
jiraUsername=someuser
jiraPassword=somepass

reportFormat=testng
projectKey=CALC
version=v1.0
revision=123
testPlanKey=CALC-726
testExecKey=
testEnvironment=
```

We then generate a "standard" TestNG XML report and use a custom task to perform the REST API request that submits the results to Xray.

The following example shows a build.gradle file with a custom task named `importTestNGResultsToXrayDC` where we implement the logic to push the results to Xray.

In this case, we won't be using the built-in `test` task; instead, we implement a custom one named `testngTest` as we need to provide additional parameters to the `XMLReporter` class provided by TestNG that otherwise is not yet possible; this is what enables the feature of embedding additional attributes on the TestNG XML report that Xray can take advantage of.

build.gradle

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

jar {
    archiveBaseName = 'tutorial-gradle-testng-basic'
    archiveVersion = '0.1.0'
}

sourceCompatibility = 11
targetCompatibility = 11

test {
    useTestNG()
    reports {
        // destination = file('build/test-results/folder')
        junitXml.required = false
        html.required = false
    }
    ignoreFailures = true
}

repositories {
    mavenLocal()
    maven {
        url = uri('https://maven.xpand-it.com/artifactory/releases')
    }

    maven {
        url = uri('https://maven.pkg.github.com/bitcoder/*')
    }

    maven {
        url = uri('https://repo.maven.apache.org/maven2/')
    }
}

dependencies {
    testImplementation 'org.testng:testng:7.6.1'
    testImplementation 'com.xpandit.xray:xray-testng-extensions:1.1.0'
}

configurations.all {
    resolutionStrategy.cacheChangingModulesFor 0, 'seconds'
    resolutionStrategy.cacheDynamicVersionsFor 0, 'seconds'
}

task testngTest(type: JavaExec, dependsOn: [classes]) {
    group 'Verification'
    description 'Run TestNG tests'
    ignoreExitValue = true // to not throw exception if test fails
    mainClass = 'org.testng.TestNG'
    args('testng.xml', '-reporter',
        'org.testng.reporters.XMLReporter:generateTestResultAttributes=true,generateGroupsAttribute=true')
    classpath = sourceSets.test.runtimeClasspath
}

task importTestNGResultsToXrayDC(type: Exec) {
```

```

description 'Imports TestNG test results to Xray Test Management for Jira DC.'
dependsOn 'testngTest'

// the following variables can be defined locally or in gradle.properties
// - jiraBaseUrl, jiraUsername, jiraPassword
// - reportFormat, projectKey version, revision, testPlanKey, testExecKey, testEnvironment

def reportFile = "test-output/testng-results.xml"

def url = "${jiraBaseUrl}/rest/raven/2.0/import/execution/${reportFormat}?"
if (projectKey?.trim()) {
    url += "&projectKey=${projectKey}"
}
if (version?.trim()) {
    url += "&fixVersion=${version}"
}
if (revision?.trim()) {
    url += "&revision=${revision}"
}
if (testPlanKey?.trim()) {
    url += "&testPlanKey=${testPlanKey}"
}
if (testExecKey?.trim()) {
    url += "&testExecKey=${testExecKey}"
}
if (testEnvironment?.trim()) {
    url += "&testEnvironments=${testEnvironment}"
}

commandLine 'curl', '--fail-with-body', '-H', 'Content-Type: multipart/form-data', '-u',
"${jiraUsername}:${jiraPassword}", '-F', "file=@${reportFile}", url

//store the output instead of printing to the console:
standardOutput = new ByteArrayOutputStream()
ignoreExitValue = false

doLast {
    if (execResult.getExitValue() != 0) {
        println "ERROR: problem importing results to Xray"
    } else {
        println "Results imported to Xray!"
    }
    println standardOutput.toString()
}
}

```

To run the tests and import them to Xray we can run Gradle as usual and add the name of the task we created earlier.

```
gradle clean compileJava testngTest importTestNGResultsToXrayDC
```

In Xray, a Test Execution will be created accordingly.

CALC / CALC-763

Execution results - testng-results.xml - [1657215230204]

[Edit](#) [Comment](#) [Assign](#) [More](#) [Stop Progress](#) [Resolve Issue](#) [Close Issue](#) [Admin](#)

Details

Type:	<input checked="" type="checkbox"/> Test Execution	Status:	IN PROGRESS (View Workflow)
Priority:	<input type="radio"/> Trivial	Resolution:	Unresolved
Affects Version/s:	None	Fix Version/s:	v1.0
Labels:	None		
Test Plan:	CALC-726		
Test Environments:	None		
Revision:	123		

Description
Execution results imported from external source

Tests

[Add Tests](#) ...

Overall Execution Status

4 PASS 1 FAIL

Total Tests: 5

[Filter\(s\)](#)

Rank Key Summary Test Type #Req #Def Assignee Status

Rank	Key	Summary	Test Type	#Req	#Def	Assignee	Status
2	CALC-757	CanDivide	Generic	0	0	Xpand IT Admin	PASS
3	CALC-744	CanDoStuff	Generic	0	0	Xpand IT Admin	PASS
4	CALC-743	subtraction test	Generic	1	0	Xpand IT Admin	PASS
5	CALC-742	CanMultiply	Generic	1	0	Xpand IT Admin	PASS
1	CALC-739	CanAddNumbers()	Generic	1	0	Xpand IT Admin	FAIL

Showing 1 to 5 of 5 entries

First Previous [1](#) Next Last

Tips

- To debug existing tasks, run your Gradle command with "–stacktrace"
- On the custom tasks that execute a command using exec, set ignoreExitValue = false

References

- [Gradle documentation](#)
- [Gradle vs Maven](#)
- [Code for the examples showcasing the integration between Gradle and Xray](#)