Integration with Gradle

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'
```

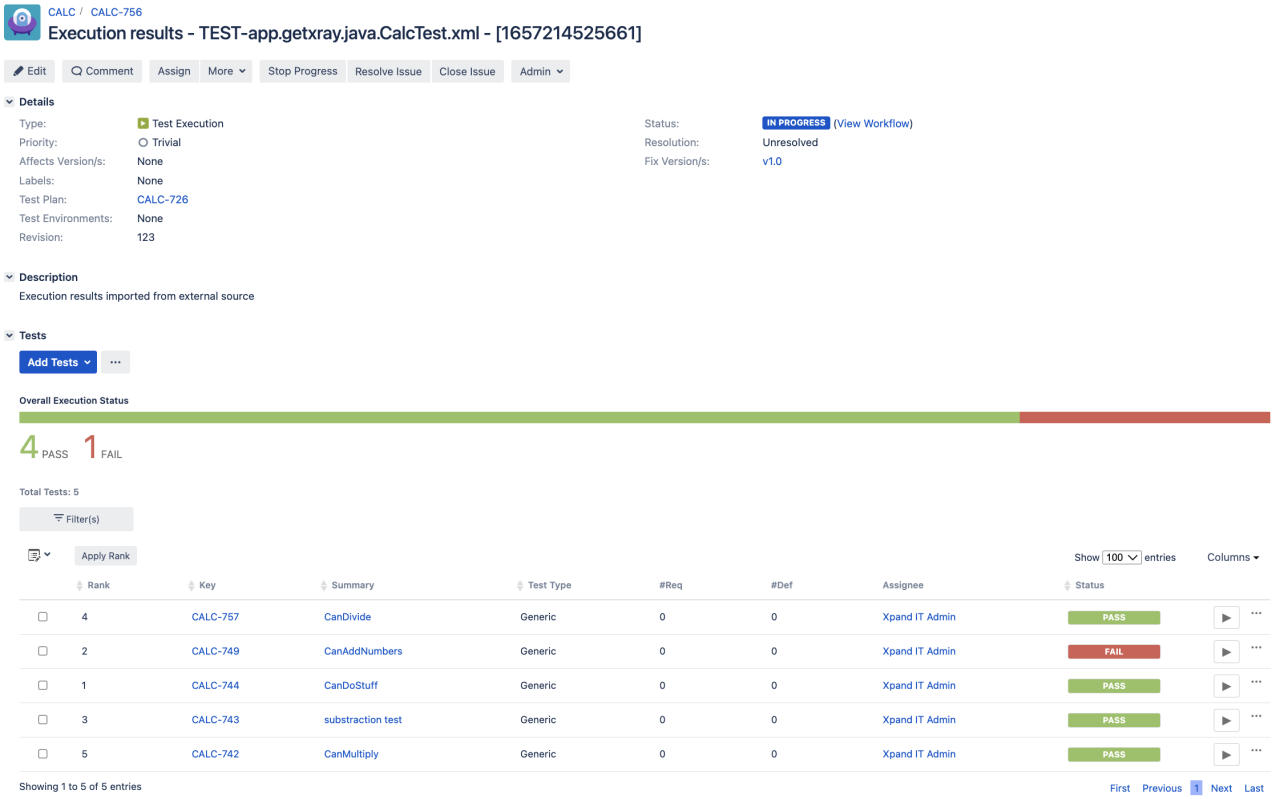
```
}

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.



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

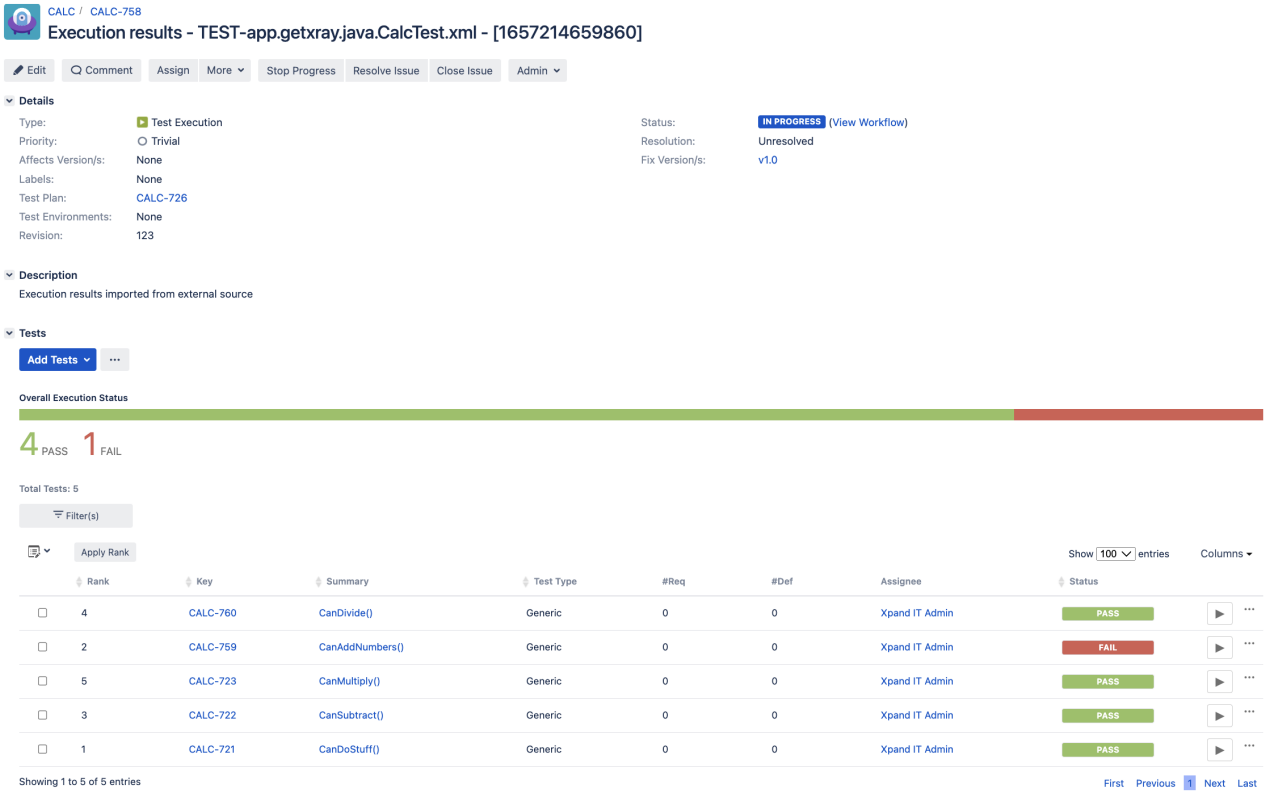
```
apply plugin: 'java'

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.



JUnit5 with additional information

In this example, the tests are implemented using Java + JUnit 5.8.x.

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

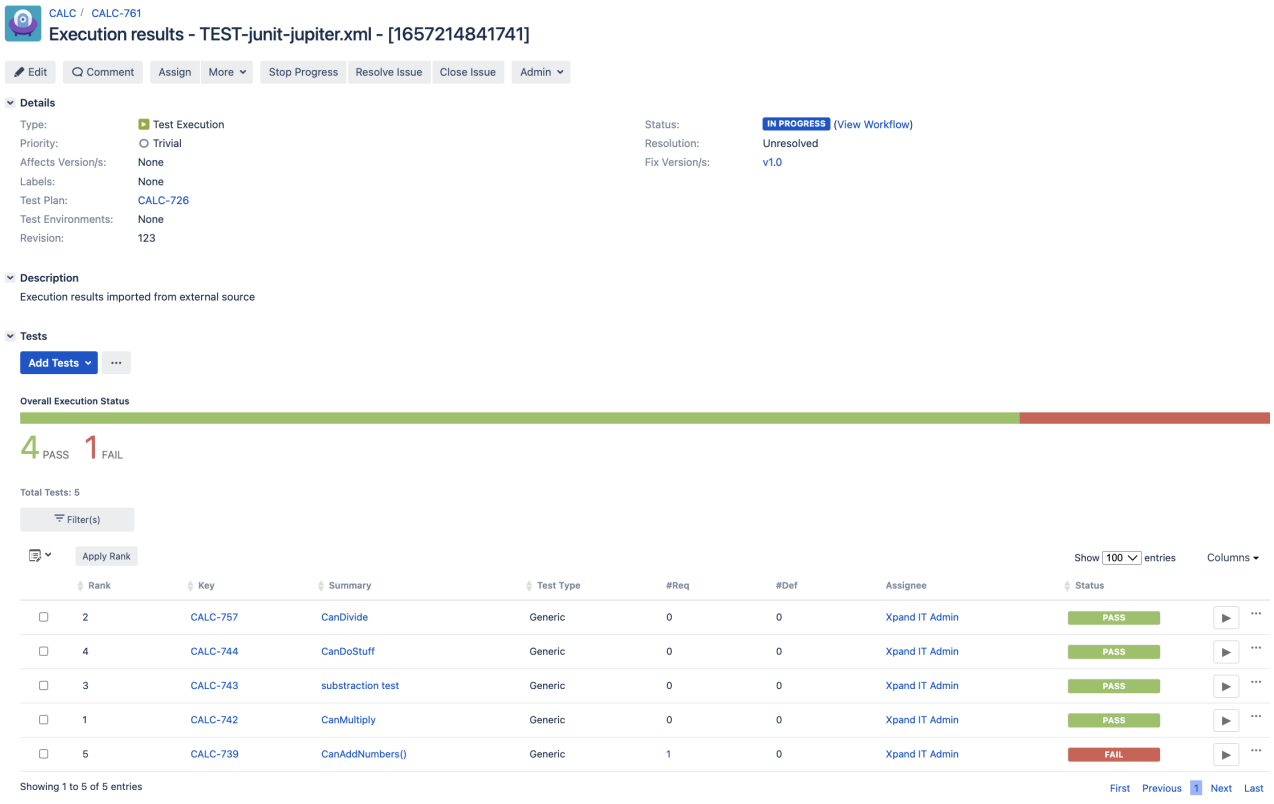
```
apply plugin: 'java'

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.



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

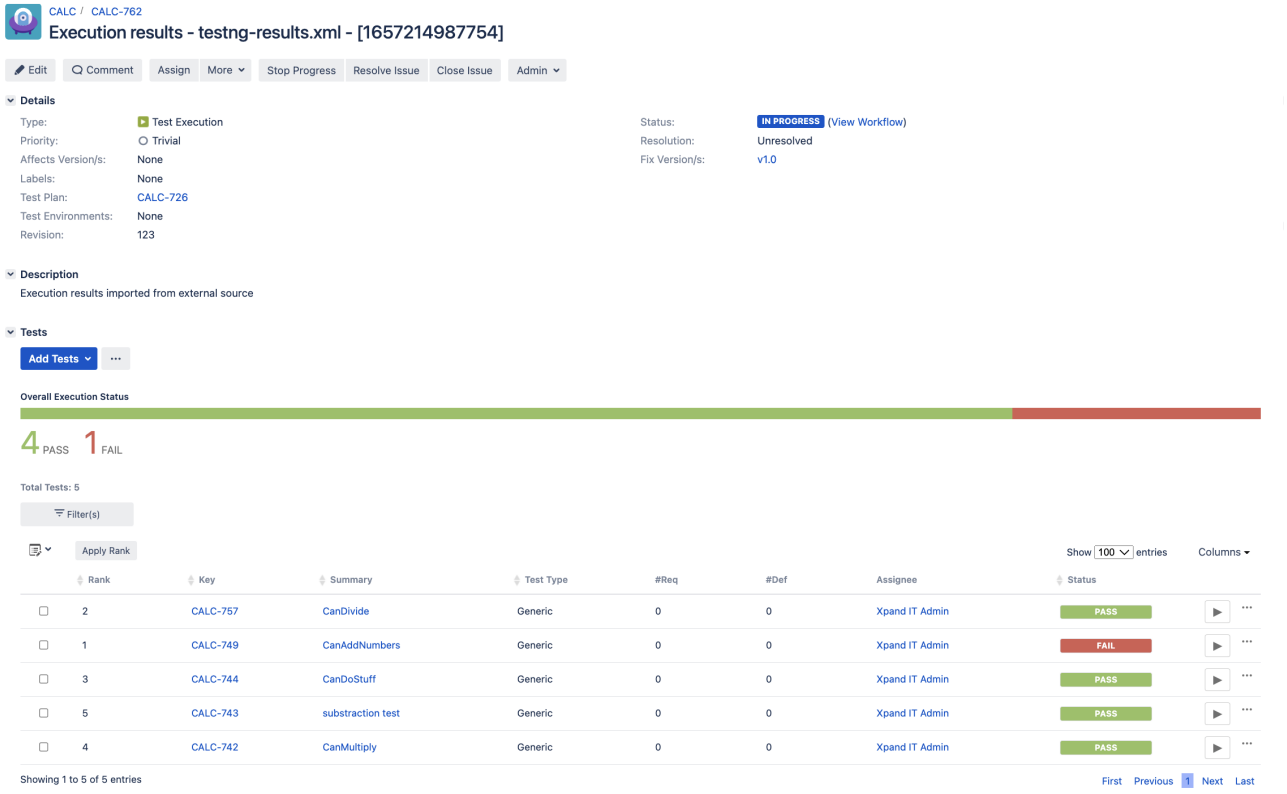
```
apply plugin: 'java'

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.



TestNG with additional information

In this example, the tests are implemented using Java + TestNG 7.6.x.

> 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

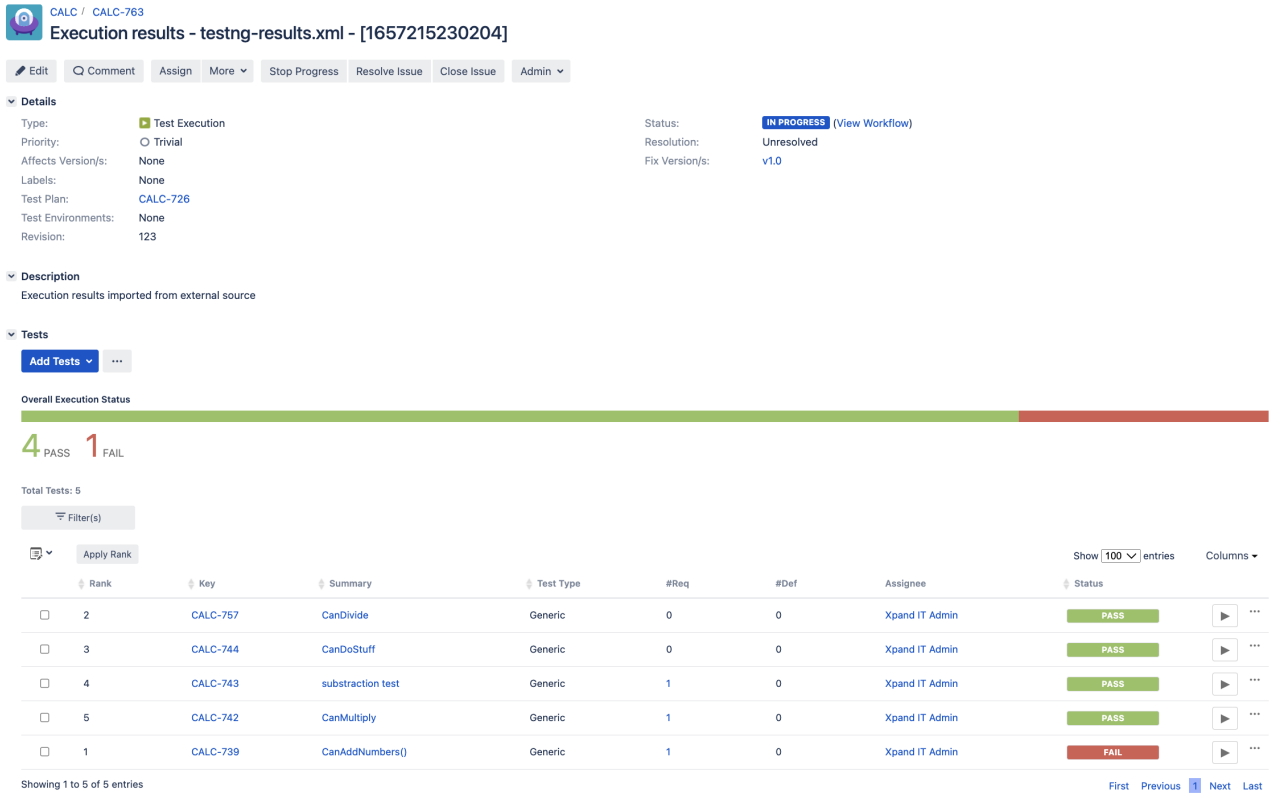
```
apply plugin: 'java'

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.



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