Analyzing with SonarQube Scanner for Gradle

By SonarSource – GNU LGPL 3 – Issue Tracker – Sources

SonarQube Scanner for Gradle 2.7.1 – Compatible with SonarQube 6.7+ (LTS)

Table of Contents

Features

The SonarQube Scanner for Gradle provides an easy way to start SonarQube analysis of a Gradle project.

The ability to execute the SonarQube analysis via a regular Gradle task makes it available anywhere Gradle is available (developer build, CI server, etc.), without the need to manually download, setup, and maintain a SonarQube Runner installation. The Gradle build already has much of the information needed for SonarQube to successfully analyze a project. By preconfiguring the analysis based on that information, the need for manual configuration is reduced significantly.

Compatibility

The SonarQube Scanner for Gradle version 2.x is compatible with Gradle versions 1.12+ and SonarQube versions 5.6+.

Bytecode created by `javac` compilation is required for Java analysis, including Android projects.

Users of SonarQube Scanner for Gradle 1.x?

Module key generation strategy was changed in 2.0 to ensure unicity (see **SONARGRADL-12** - Getting issue details... STATUS).

Upgrading from 1.x to 2.x without care will make all previous issues reappear as "new". You can avoid that by following this workflow:

- Recommended: Backup your SQ database in case something goes wrong
- Use module key update feature to change key of all modules (except root module) following this pattern: <root module key>:<module path> Reminder: <root module key> is unchanged and should be equal to [<group>:]<name>
- Update SonarQube plugin in your build.gradle to 2.x
- Run an analysis
- Verify issue history was preserved

Prerequisites

- SonarQube is already installed
- At least the minimal version of Java supported by your SonarQube server is in use
- The language plugins for each of the languages you wish to analyze are installed
- You have read Analyzing Code Source.

Configure the Scanner

Installation is automatic, but certain global properties should still be configured. A good place to configure global properties is *~/.gradle/gradle.properties*. Be aware that we are using System properties so all properties should be prefixed by **`systemProp.`**

gradle.properties

```
systemProp.sonar.host.url=http://localhost:9000

#----- Token generated from an account with 'publish analysis' permission
systemProp.sonar.login=<token>
```

Analysis

1 - Activate the scanner in your build

For Gradle 2.1+:

build.gradle

```
plugins {
  id "org.sonarqube" version "2.7.1"
}
```

More details on **https://plugins.gradle.org/plugin/org.sonarqube**

Assuming a local SonarQube server with out-of-the-box settings is up and running, no further mandatory configuration is required.

2 - Run analysis

Execute **`gradle sonarqube`** and wait until the build has completed, then open the web page indicated at the bottom of the console output. You should now be able to browse the analysis results.

Configure analysis properties

The SonarQube Scanner for Gradle leverages information contained in Gradle's object model to provide smart defaults for many of the standard SonarQube properties. The defaults are summarized in the tables below.

Gradle defaults for standard SonarQube properties

Property	Gradle default
`sonar.projectKey`	[`${project.group}:`]`${project.name}` for root module `<root module key>:<module path>` for submodules
`sonar.projectName`	`${project.name}`
`sonar.projectDescription`	`${project.description}`
`sonar.projectVersion`	`${project.version}`
`sonar.projectBaseDir`	`${project.projectDir}`
`sonar.working.directory`	`${project.buildDir}/sonar`

Notice that additional defaults are provided for projects that have the `java-base` or `java` plugin applied:

Additional defaults for `Java` projects

Property	Gradle default
`sonar.sourceEncoding`	`${project.compileJava.options.encoding}`
`sonar.java.source`	`${project.sourceCompatibility}`

sonar.java.target	${project.targetCompatibility}
sonar.sources	${sourceSets.main.allSource.srcDirs} (filtered to only include existing directories)
sonar.tests	${sourceSets.test.allSource.srcDirs} (filtered to only include existing directories)
sonar.java.binaries	${sourceSets.main.output.classesDir}
sonar.java.libraries	${sourceSets.main.compileClasspath} (filtering to only include files; rt.jar and jfxrt.jar added if necessary)
sonar.java.test.binaries	${sourceSets.test.output.classeDir}
sonar.java.test.libraries	${sourceSets.test.compileClasspath} (filtering to only include files; rt.jar and jfxrt.jar added if necessary)
sonar.junit.reportPaths	${test.testResultsDir} (if the directory exists)

Additional defaults for Groovy projects

Same settings than for Java projects plus:

Property	Gradle default
sonar.groovy.binaries	${sourceSets.main.output.classesDir}

Additional defaults when JaCoCo plugin is applied

Property	Gradle default
sonar.jacoco.reportPaths	${jacoco.destinationFile}
sonar.groovy.jacoco.reportPath	${jacoco.destinationFile}

Additional defaults for Android projects (com.android.application, com.android.library or com.android.test)

By default the first variant of type "debug" will be used to configure the analysis. You can override the name of the variant to be used using the parameter 'androidVariant':

build.gradle

```
sonarqube {
    androidVariant 'fullDebug'
}
```

Property	Gradle default
sonar.sources (for non test variants)	${variant.sourcesets.map} (ManifestFile/CDirectories/AidlDirectories/AssetsDirectories/CppDirectories /JavaDirectories/RenderscriptDirectories/ResDirectories/ResourcesDirectories)
sonar.tests (for test variants)	${variant.sourcesets.map} (ManifestFile/CDirectories/AidlDirectories/AssetsDirectories/CppDirectories /JavaDirectories/RenderscriptDirectories/ResDirectories/ResourcesDirectories)
sonar.java[.test].binaries	${variant.destinationDir}
sonar.java[.test].libraries	${variant.javaCompile.classpath} + ${bootclasspath}
sonar.java.source	${variant.javaCompile.sourceCompatibility}

`sonar.java.target`	`${variant.javaCompile.targetCompatibility}`

Passing manual properties / overriding defaults

The SonarQube Scanner for Gradle adds a `SonarQubeExtension` extension to project and its subprojects, which allows you to configure/override the analysis properties.

build.gradle

```
sonarqube {
    properties {
        property "sonar.exclusions", "**/*Generated.java"
    }
}
```

Alternatively, SonarQube properties can be set from the command line. See "Configuring properties from the command line" for more information.

Analyzing Multi-Project Builds

To analyze a project hierarchy, apply the SonarQube plugin to the root project of the hierarchy. Typically (but not necessarily) this will be the root project of the Gradle build. Information pertaining to the analysis as a whole has to be configured in the `sonarqube` block of this project. Any properties set on the command line also apply to this project.

Global configuration settings

build.gradle

```
sonarqube {
    properties {
        property "sonar.sourceEncoding", "UTF-8"
    }
}
```

Shared configuration settings

Configuration shared between subprojects can be configured in a `subprojects` block.

build.gradle

```
subprojects {
    sonarqube {
        properties {
            property "sonar.sources", "src"
        }
    }
}
```

Individual configuration settings

Project-specific information is configured in the `sonarqube` block of the corresponding project.

build.gradle

```
project(":project1") {
    sonarqube {
        properties {
            property "sonar.branch", "Foo"
        }
    }}
```

Skipping analysis of a project

To skip SonarQube analysis for a particular subproject, set `sonarqube.skipProject` to `true`.

build.gradle

```
project(":project2") {
    sonarqube {
        skipProject = true
    }
}
```

Analyzing Custom Source Sets

By default, the SonarQube Scanner for Gradle passes on the project's `main` source set as production sources, and the project's `test` source set as test sources. This works regardless of the project's source directory layout. Additional source sets can be added as needed.

Analyzing custom source sets

build.gradle

```
sonarqube {
    properties {
        properties["sonar.sources"] += sourceSets.custom.allSource.srcDirs
        properties["sonar.tests"] += sourceSets.integTest.allSource.srcDirs
    }
}
```

More on configuring SonarQube properties

Let's take a closer look at the `sonarqube.properties {}` block. As we have already seen in the examples, the `property()` method allows you to set new properties or override existing ones. Furthermore, all properties that have been configured up to this point, including all properties preconfigured by Gradle, are available via the `properties` accessor.

Entries in the `properties` map can be read and written with the usual Groovy syntax. To facilitate their manipulation, values still have their “idiomatic” type (`File`, `List`, etc.). After the sonarProperties block has been evaluated, values are converted to Strings as follows: Collection values are (recursively) converted to comma-separated Strings, and all other values are converted by calling their `toString()` method.

Because the `sonarProperties` block is evaluated lazily, properties of Gradle's object model can be safely referenced from within the block, without having to fear that they have not yet been set.

Setting properties from the Command Line

SonarQube properties can also be set from the command line, by setting a system property named exactly like the SonarQube property in question. This can be useful when dealing with sensitive information (e.g. credentials), environment information, or for ad-hoc configuration.

```
gradle sonarqube -Dsonar.host.url=http://sonar.mycompany.com -Dsonar.verbose=true
```

While certainly useful at times, we do recommend to keep the bulk of the configuration in a (versioned) build script, readily available to everyone.

A SonarQube property value set via a system property overrides any value set in a build script (for the same property). When analyzing a project hierarchy, values set via system properties apply to the root project of the analyzed hierarchy. Each system property starting with ""`sonar.`" will be taken into account.

Task dependencies

Before executing the `sonarqube` task, all tasks producing output to be included in the SonarQube analysis need to be executed. Typically, these are compile tasks, test tasks, and code coverage tasks. To meet these needs, the plugins adds a task dependency from `sonarqube` on `test` if the `java` plugin is applied. Further task dependencies can be added as needed. For example:

build.gradle

```
project.tasks["sonarqube"].dependsOn "anotherTask"
```

Example

A simple working example is available at this URL so you can check everything is correctly configured in your env: https://github.com/SonarSource/sonar-scanning-examples/tree/master/sonarqube-scanner-gradle