

SonarQube Integration in einem Android Projekt

Um eine bessere Code Qualität an unseren Softwareprodukten zu gewährleisten haben wir uns im Unternehmen entschlossen eine statische Code Analyse einzuführen.

Ein besonderer Augenmerk liegt hierbei auf der Code Coverage und das Einhalten von zuvor festgelegten Programmier-Richtlinien.

In diesem Artikel geht es um das Erstellen+Anzeigen von Testreports auf einem SonarQube Server mittels Jenkins, um das Herunterladen+Anzeigen dieser in Android Studio und um das Anzeigen von lokalen, neuen SonarQube Issues. Desweiteren wird noch kurz auf die SonarQube-Benutzer-Oberfläche eingegangen.

SonarQube ist ein Tool für die statische Code Analyse. Hierfür werden zuvor erstellte Test Reports von SonarQube eingelesen und nach bestimmten Richtlinien und Regeln ausgewertet.

Vorbereitung des lokalen Projekts

Zuerst muss das Android-Projekt so konfiguriert werden, dass es Jacoco Test Reports erstellt. Dafür wird eine jacoco.gradle Datei erstellt:

```

apply plugin: 'jacoco'

ext {
    coverageSourceDirs = 'src/test/java'
}

jacoco {
    toolVersion = "0.8.3"
    reportsDir = file("${buildDir}/reports")
}

task jacocoTestReport(type: JacocoReport, dependsOn: "testDebugUnitTest") {
    group = "Reporting"
    description = "Generate Jacoco coverage reports for Debug build"

    reports {
        xml.enabled = true
        html.enabled = true
    }

    def excludes = ['**/R.class',
        '**/R$.class',
        '**/build/generated/**',
        '**/*$ViewBinder.*',
        '**/*$InjectAdapter.*',
        '**/*$Injector.*',
        '**/BuildConfig.*',
        '**/Manifest.*',
        '**/*Test.*',
        '**/CiMattersApplication.*',
        'android/**/*.*']

    def debugTree = fileTree(
        dir: "${buildDir}/intermediates/javac/debug/compileDebugJavaWithJavac/classes",
        excludes: excludes)
    def kotlinDebugTree = fileTree(
        dir: "${buildDir}/tmp/kotlin-classes/debug",
        excludes: excludes)
    def mainSrc = "${project.projectDir}/src/main/java"

    classDirectories = files([debugTree], [kotlinDebugTree])
    executionData = files("${buildDir}/jacoco/testDebugUnitTest.exec")
    sourceDirectories = files([mainSrc])
}

```

In dieser Datei wird der Gradle-Task zur Erstellung des Test-Reports angelegt und die entsprechenden Pfade für die Source- und Class Dateien für die Java- und Kotlin Klassen angegeben.

In der build.gradle Datei des Moduls muss dann noch die jacoco.gradle Datei hinzugefügt werden.

```

apply from: '../jacoco.gradle'

android {

```

Zum Schluss müssen noch einige Konfigurations-Einstellungen für Sonar in den gradle.properties angegeben werden.

```

sonar.projectVersion=3.2.2.0.1
#
sonar.projectKey=gsp
sonar.projectName=Douglas Beauty Tab (Android)
sonar.modules=gsp
sonar.sources=src/main/java/de/douglas/gsp
sonar.scm.provider=git
sonar.java.binaries=build/intermediates/javac/debug/compileDebugJavaWithJavac/classes/de/douglas/gsp
sonar.coverage.jacoco.xmlReportPaths=build/reports/jacocoTestReport/jacocoTestReport.xml

```

Nun können mit den folgenden Gradle-Tasks die Test-Reports erstellt werden.

1. Build

- baut das Projekt und generiert die benötigten Class-Dateien

2. **testDebugUnitTest**

- führt die Tests aus und erstellt einen jacoco Test-Report
(modul/build/jacoco/testDebugUnitTest.exec)

3. **jacocoTestReport**

- erstellt die Test-Reports im HTML Format
(modul/build/reports/jacocoTestReport/)

Es

kann auch nur der jacocoTestReport-Task ausgeführt werden, da dieser eine Abhängigkeit zu den anderen Tasks beinhaltet. Bei einem cleanen Projekt benötigt dies aber mehr Zeit.

Nach dem Ausführen der Gradle-Tasks liegen die Reports im Modul-Ordner unter build/reports/jacoco/TestReports im HTML-Format vor und können im Browser angezeigt werden.

gsp > de.douglas.gsp.base.error > Errors

Errors

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods
getErrorMessage(Integer, String, ErrorContext)		97%		95%	1 11	0 7	0 1
errorCode(Throwable)		95%		75%	1 3	1 5	0 1
static (...)		100%		n/a	0 1	0 39	0 1
getErrorMessage(Throwable, ErrorContext)		100%		100%	0 3	0 4	0 1
getErrorMessage(Response, ErrorContext)		100%		n/a	0 1	0 1	0 1
getErrorMessage\$default(Integer, String, ErrorContext, int, Object)		100%		n/a	0 1	0 1	0 1
getErrorMessage\$default(Throwable, ErrorContext, int, Object)		100%		n/a	0 1	0 1	0 1
getErrorMessage\$default(Response, ErrorContext, int, Object)		100%		n/a	0 1	0 1	0 1
Total	3 of 378	99%	2 of 28	92%	2 22	1 59	0 8

Vorbereitung des Jenkins-Servers

Um sich die Reports im Sonar an zu schauen, kann man diese über den Jenkins-Server zu Sonar übertragen.

Im Jenkins muss zuerst das Sonar-Plugin installiert werden, bei dem die URL des Servers eingetragen wird und der Sonar Runner ausgewählt werden.

Die Einstellung für das Sonar-Plugin findet man unter:
Manage Jenkins -> Configure System -> SonarQube

SonarQube servers

Environment variables ☐ Enable injection of SonarQube server configuration as build environment variables
If checked, job administrators will be able to inject a SonarQube server configuration as environment variables in the build.

SonarQube installations

Name

Server URL
Default is http://localhost:9000

Server authentication token
SonarQube authentication token. Mandatory when anonymous access is disabled.

[Advanced...](#)

[Delete SonarQube](#)

[Add SonarQube](#)

List of SonarQube installations

Die Einstellung für den Sonar-Runner befindet sich unter:
 Manage Jenkins -> Global Tool Configure -> SonarQube Scanner

SonarQube Scanner

SonarQube Scanner installations

☐ SonarQube Scanner

Name

☒ Install automatically [?](#)

☐ Install from Maven Central

Version

[Delete Installer](#)

[Add Installer](#)

[Delete SonarQube Scanner](#)

[Add SonarQube Scanner](#)

List of SonarQube Scanner installations on this system

In dem Jenkins-Job müssen dann noch die folgenden Gradle-Tasks eingetragen werden:

Invoke Gradle script

☐ Invoke Gradle

☒ Use Gradle Wrapper

Make gradlew executable ☐

Wrapper location

Tasks

```
gsp:clean
gsp:test
gsp:assembleDebug
jacocoTestReport
```

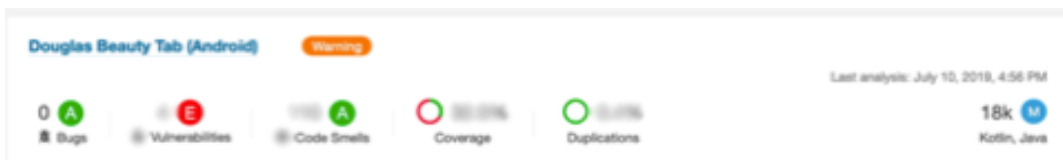
Anschließend wird der SonarQube Scanner konfiguriert. Hier muss die Datei mit den Properties für Sonar angegeben werden.

Nun wird jedes Mal, wenn der Jenkins-Job für das Projekt ausgeführt wird, auch der SonarQube Scanner ausgeführt und veröffentlicht seinen Report auf dem SonarQube-Server.

Benutzeroberfläche des Sonar-Servers

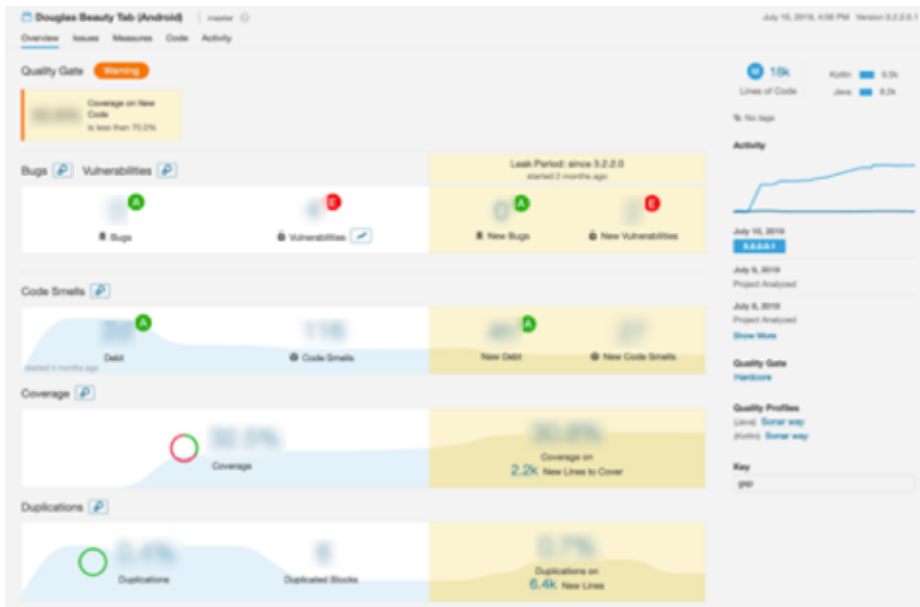
Damit Sonar die Testreports empfangen kann und diese korrekt angezeigt werden, müssen hier noch das SonarKotlin-, das SonarJava- und das Git-Plugin installiert werden.

Auf der Startseite von Sonar werden alle Projekte aufgelistet und eine Übersicht der Analyse angezeigt.



Klickt man auf ein Projekt, werden detailliertere und zusätzlich Informationen angezeigt.

Man kann die Analyse in 2 Kategorien einteilen. Das sind zum Einen Messungen (Code Coverage, Duplication) die SonarQube anhand des Codes durchführt und zum Anderen Issues (Bugs, Vulnerabilities, Code Smells), die durch Verletzung von zuvor festgelegten Code Richtlinien auftreten. Eine zusätzliche Richtlinie kann man für ein sogenanntes Quality Gate festlegen. Dies kann zum Beispiel eine Mindestanforderung von der Code Coverage des neu entwickelten Codes sein.



Über die einzelnen Analyse-Kategorien gelangt man zu einer detaillierten Ansicht der einzelnen Issues oder Messungen.

Display Mode

Issues Effort

Type

- Bug 0
- Vulnerability 2**
- Code Smell 24

Severity

gsp / src/.../modules/customer/enrollment/BeautyCardEnrollPresenter.java

Make enrollPresenterState a static final constant or non-public and provide accessors if needed. 25 days ago L87 cwe

gsp / src/.../gsp/monitoring/usecases/HandleRequestError.kt

Make sure using this hardcoded IP address is safe here. 4 months ago L32 cert

2 of 2 shown

Issues (Bugs, Vulnerabilities, Code Smells)

Coverage

Overview

On new code

Coverage	30.8%
Lines to Cover	2,230
Uncovered Lines	1,575
Line Coverage	29.4%
Conditions to Cover	843
Uncovered Conditions	551
Condition Coverage	34.6%

Coverage 32.5%

Leak Period: since 3.2.2.0

	Coverage	Uncovered Lines	Uncovered Conditions
src/main/java/de/douglas/gsp/modules/customer/enrollment/usecases/AbortEnrollmentUseCase.kt	0.0%	34	6
src/main/java/de/douglas/gsp/base/Interop/Action.java	0.0%	3	-
src/main/java/de/douglas/gsp/base/Interop/Action1.java	0.0%	3	-
src/main/java/de/douglas/gsp/dagger/ActivityBuilderModule.kt	0.0%	18	-
src/main/java/de/douglas/gsp/utills/ActivityHelper.java	0.0%	33	10
src/main/java/de/douglas/gsp/dagger/ActivityModule.kt	0.0%	3	-
src/main/java/de/douglas/gsp/base/activities/ActivityResult.kt	0.0%	4	-

Messungen (Code Coverage, Duplications)

SonarQube Community Plugin

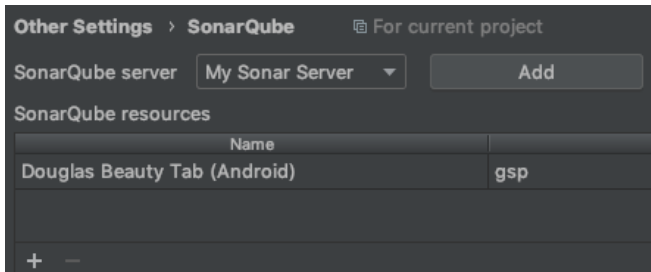
Für

Android-Studio gibt es das SonarQube Community Plugin, welches 2 Funktionen mit sich bringt:

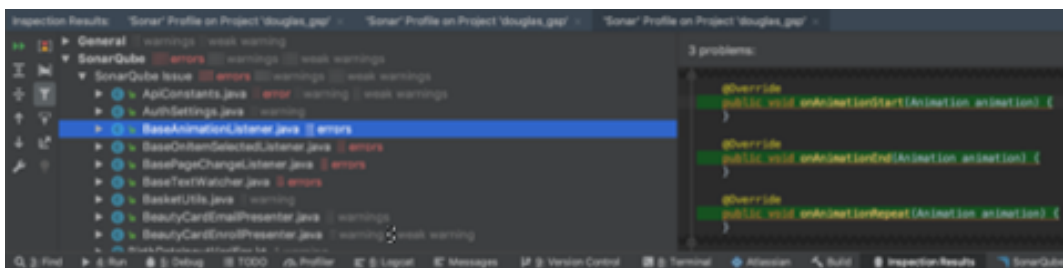
1. Anzeigen von vorhandenen Issues vom Sonar-Server
2. Anzeigen von neuen Issues mittels einer lokalen Projekt-Analyse

Vorhandene Issues vom Sonar-Server

Um sich die vorhandenen Issues vom Server herunterzuladen, muss das Plugin zuerst konfiguriert werden. Hierzu ruft man dieses in den Einstellungen von Android-Studio auf und trägt den Sonar-Server ein. Anschließend wählt man das Projekt in dem Abschnitt Ressourcen aus.

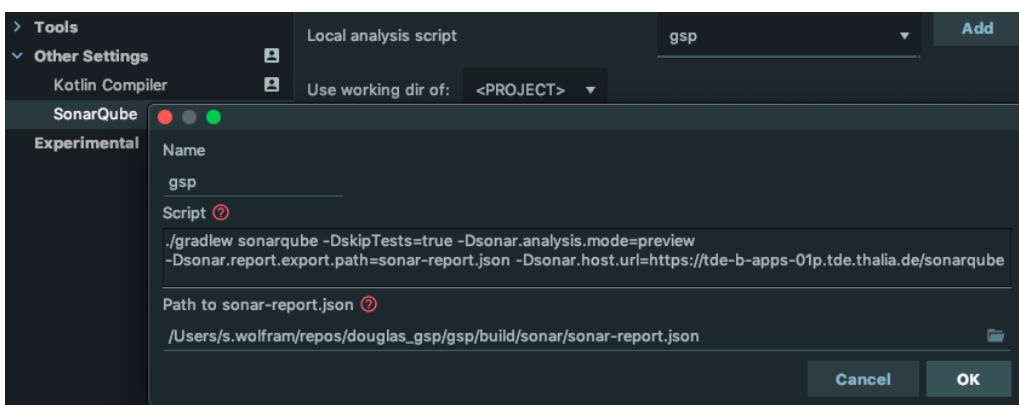


Unter dem Reiter Analyse -> Inspect Code muss in dem Inspection Profile SonarQube ausgewählt werden. Nun werden bei der Code Inspection die Sonar-Issues vom Server heruntergeladen und in der IDE angezeigt.



Lokale Projekt-Analyse mit dem Sonar-Gradle-Plugin

Um das lokale Projekt nach Sonar-Issues zu analysieren muss zunächst das SonarQube-Script konfiguriert werden. Hier muss der Sonar-Gradle-Task und der Pfad zum sonar-report eingestellt werden. Der Sonar-Report ist vor dem ersten Ausführen noch nicht vorhanden, da er erst durch den Gradle-Task erstellt wird.



Als nächster Schritt muss lokal SonarQube konfiguriert werden. Hierzu wird

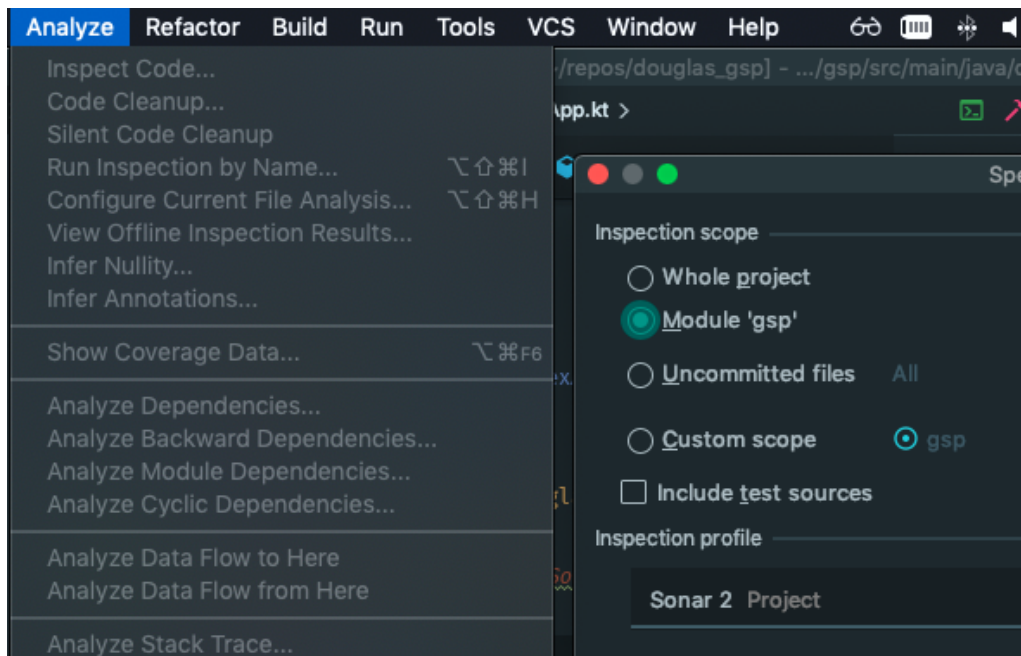
zuerst in der build.gradle Datei des Projekts sonarQube als Dependencie hinzugefügt.

```
dependencies {  
    classpath "org.sonarsource.scanner.gradle:sonarqube-gradle-plugin:2.7.1"
```

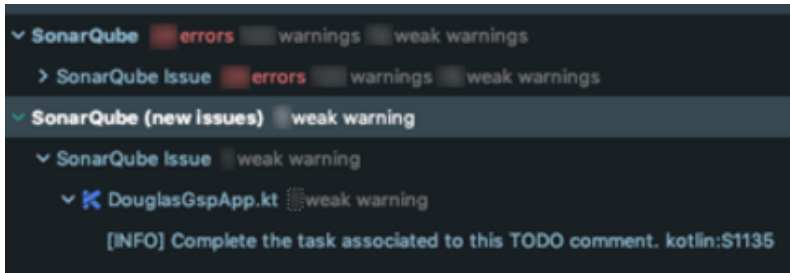
In der build.gradle Datei des Moduls wird SonarQube dann applied und konfiguriert.

```
apply plugin: 'org.sonarqube'  
  
sonarqube {  
    properties {  
        property 'sonar.projectName', 'Douglas Beauty Tab (Android) - From Local'  
        property "sonar.host.url", "https://tde-b-apps-01p.tde.thalia.de/sonarqube"  
        property "sonar.jacoco.reportPaths", "build/jacoco/testDebugUnitTest.exec"  
        property "sonar.projectVersion", version  
        property "sonar.scm.provider", "git"  
    }  
}
```

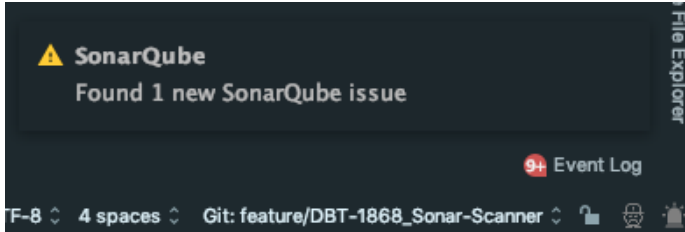
Nachdem SonarQube konfiguriert wurde, kann das Projekt unter Analyze -> Inspect Code analysiert werden. Hierzu muss nur noch das Sonar-Profil (mit Sonar Issues + new Sonar Issues) ausgewählt und mit OK bestätigt werden.



Nach der Analyse werden die Sonar-Issues vom Server und die aus der lokalen Analyse unten im Reiter Inspection Results angezeigt.



Neu gefundenen lokalen SonarQube Issues werden zusätzlich noch in einem kleinen Popup unten rechts von Android Studio angezeigt.



Nach der Einführung von SonarQube in unser Android-Projekt haben wir eine zentrale Anlaufstelle für unsere Code Qualität und kontrollieren regelmäßig diese in der SonarQube-Benutzer-Oberfläche.

Desweiteren bekommen wir über Jenkins ein schnelles Feedback ob unser Quality Gate eingehalten wurde.

Mit dem Community-Plugin laden wir uns die SonarQube-Issues vom Server herunter und beheben diese direkt in Android Studio. Bevor wir ein Feature pushen, kontrollieren wir mittels des Plugins im Vorfeld das Einhalten der Programmier Richtlinien und können neu aufgetretene Issues direkt in Android Studio beheben.