# 코드 분석 도구 적용기 - 1편, 코드 커버리지(Code Coverage)가 뭔가요?

시작하기에 앞서 코드 커버리지와 밀접하게 관련 있는 **테스트 코드**에 대해 간단하게 설명하고 들어가도록 하겠습니다.

저희 팀은 직고래 프로젝트를 진행하면서 꾸준히 **테스트 코드를 작성**하고 있습니다.

테스트 코드를 작성하는 것이 그리 재밌는 일이 아님에도 불구하고 , 테스트 코드를 작성하는 이유는 테스트 코드를 작성하면 얻을 수 있는 **다양한 장점** 때문입니다.

몇 가지만 이야기해보면

* 제품의 **안정성**을 높여준다.
* 기능 추가 및 수정으로 인한 **부작용(Side-effect)**을 줄일 수 있다.
* **깔끔하고 재사용성이 좋은 코드 작성**을 가능하게 해준다.

등이 있습니다.

테스트 코드가 가지는 장점을 잘 알았으니 테스트 코드를 열심히 작성해야겠죠? 그런데, 테스트 코드를 잘 작성하고 있는지는 **어떻게 확인할 수 있을까요?**

**기능 추가 및 수정으로 인한 부작용을 줄여준다**라고 했는데, 혹시나 테스트 코드 작성을 빠트린 부분이 발생한다면 어떻게 할까요?

테스트 코드를 **정말 열심히** 작성하고 있었는데 하필 딱! 실수로 테스트 코드가 작성되지 않은 로직에 대한 **변경**이 발생했고, 이 변경으로 인해 **부작용이 발생**한다면?

과연 우리는 이런 상황이 발생하지 않으리란 보장을 할 수 있을까요? 만약 보장할 수 없다면, 우리는 언젠가 발생할지 모르는 일에 대해 항상 불안감에 떨며 개발을 해야 하는 걸까요?

지금부터 소개할 코드 커버리지는 이러한 우리의 **불안감을 해소**해 줄 수 있을 것입니다.

## 코드 커버리지(Code Coverage)란?

### 코드 커버리지(Code Coverage)

***In***[**computer science**](https://en.wikipedia.org/wiki/Computer_science)***, test coverage is a measure used to describe the degree to which the***[**source code**](https://en.wikipedia.org/wiki/Source_code)***of a***[**program**](https://en.wikipedia.org/wiki/Computer_program)***is executed when a particular***[**test suite**](https://en.wikipedia.org/wiki/Test_suite)***runs. A program with high test coverage, measured as a percentage, has had more of its source code executed during testing, which suggests it has a lower chance of containing undetected***[**software bugs**](https://en.wikipedia.org/wiki/Software_bug)***compared to a program with low test coverage. -***[**wikipedia**](https://en.wikipedia.org/wiki/Code_coverage)

코드 커버리지는 소프트웨어의 **테스트 케이스가 얼마나 충족되었는지를 나타내는 지표** 중 하나입니다. 테스트를 진행하였을 때 **‘코드 자체가 얼마나 실행되었느냐’**는 것이고, 이는 **수치**를 통해 확인할 수 있습니다.

### 코드 커버리지는 어떻게 측정할까?

코드 커버리지는 소스 코드를 기반으로 수행하는 **화이트 박스 테스트**를 통해 측정합니다.

***블랙 박스 테스트(*Black-box test*)***

***- 소프트웨어의 내부 구조나 작동 원리를 모르는 상태에서 동작을 검사하는 방식이다.***

***- 올바른 입력과 올바르지 않은 입력을 입력하여 올바른 출력이 나오는지 테스트하는 기법이다.***

***- 사용자 관점의 테스트 방법이라 볼 수 있다.***

***화이트 박스 테스트(*White-box test*)***

***- 응용 프로그램의 내부 구조와 동작을 검사하는 테스트 방식이다.***

***- 소프트웨어 내부 소스 코드를 테스트하는 기법이다.***

***- 개발자 관점의 단위 테스트 방법이라 볼 수 있다.***

그럼 측정하는 **기준**으로는 어떤 것들이 있을까요?

먼저 코드의 구조를 살펴보면 크게 **구문(Statement), 조건(Condition), 결정(Decision)**의 구조로 이루어져 있습니다. 코드 커버리지는 이러한 코드의 구조를 **얼마나 커버했느냐에 따라 측정기준이 나뉘게 됩니다.**

* **구문(Statement)**

***라인(*Line*) 커버리지라고 부르기도 합니다.***

**코드 한 줄이 한 번이상 실행**된다면 충족된다.

Copy

1void foo (int x) {

2 system.out("start line"); // 1번

3 if (x > 0) { // 2번

4 system.out("middle line"); // 3번

5 }

6 system.out("last line"); // 4번

7}

위의 코드를 테스트한다고 가정해보겠습니다. **x = -1**을 테스트 데이터로 사용할 경우, if 문의 조건을 통과하지 못하기 때문에 3번 코드는 실행되지 못합니다. 총 4개의 라인에서 1, 2, 4번의 라인만 실행되므로 **구문 커버리지**는 **3 / 4 \* 100 = 75(%)**가 됩니다.

* **조건(Condition)**

**모든 조건식의 내부 조건이 true/false**을 가지게 되면 충족된다.

Copy

1void foo (int x, int y) {

2 system.out("start line"); // 1번

3 if (x > 0 && y < 0) { // 2번

4 system.out("middle line"); // 3번

5 }

6 system.out("last line"); // 4번

7}

***내부 조건이라는 것이 헛갈릴 수 있는데 조건식 내부의 각각의 조건이라 생각하면 될 것 같습니다.***

***위 코드를 예시로 보면 모든 조건식으로는 2번 if 문이 있고, 그중 내부 조건은 조건식 내부의 x > 0, y < 0을 말합니다.***

위의 코드를 테스트한다고 가정해보겠습니다. **조건 커버리지**를 만족하는 테스트 케이스로는 **x = 1, y = 1**, **x = -1, y = -1**이 있습니다. 이는 **x > 0** 내부 조건에 대해 **true/false**를 만족하고, **y < 0** 내부 조건에 대해 **false/true**를 만족합니다. 그러나 테스트 케이스는 if 문은 조건에 대해 **false**만 반환합니다. if 문의 조건을 통과하지 못하기 때문에 3번 코드는 실행되지 못합니다.

조건 커버리지를 기준으로 테스트를 진행할 경우, **구문 커버리지와 결정 커버리지를 만족하지 못하는 경우가 존재**할 수 있습니다.

* **결정(Decision)**

***브랜치(*Branch*) 커버리지라고 부르기도 합니다.***

**모든 조건식이 true/false**을 가지게 되면 충족된다.

Copy

1void foo (int x, int y) {

2 system.out("start line"); // 1번

3 if (x > 0 && y < 0) { // 2번

4 system.out("middle line"); // 3번

5 }

6 system.out("last line"); // 4번

7}

위의 코드를 테스트한다고 가정해보겠습니다. if 문의 조건에 대해 **true/false** 모두 가질 수 있는 테스트 케이스로는 **x = 1, y = -1**, **x = -1, y = 1**이 있습니다. 첫 번째 테스트 데이터는 **x > 0**과 **y < 0** 모두 **true**이기 때문에 if 문의 조건에 대해 **true를 반환**합니다. 두 번째 테스트 데이터는 **x < 0**에서 이미 **false**이기 때문에 if 문의 조건에 대해 false를 반환합니다. 모든 조건식에 대해 **true**와 **false**를 반환하므로 **결정 커버리지를 충족**합니다.

위의 세 가지 코드 커버리지 중에서 구문 커버리지가 가장 대표적으로 많이 사용되고 있습니다.

### 코드 커버리지가 왜 중요하죠?

지금까지 **코드 커버리지가 무엇인지 그리고 어떻게, 어떤 기준으로 측정하는지**에 대해 알아보았습니다.

코드 커버리지가 어떤 것인지 알게 되었다면, 이제 이런 의문이 들 수 있습니다.

***코드 커버리지가 뭔지는 이제 알겠어요.***

***근데 코드 커버리지가 왜 중요하나요? 그리고 실제로 코드 커버리지를 많이 사용하는가요?***

먼저 **코드 커버리지의 중요성**은 **테스트 코드의 중요성과 일맥상통**한다고 생각합니다.

테스트 코드를 작성함으로써 얻을 수 있는 장점에 대해서는 앞서 설명해드렸습니다. 그렇다면 테스트 코드를 잘 작성하고 있는지는 어떻게 알 수 있을까요?

테스트 코드는 발생할 수 있는 **모든 시나리오에 대해 작성**되어야 합니다. 그런데 개발자도 사람인지라 테스트로 커버하지 못하는 부분이 발생할 수 있습니다. 비즈니스 코드는 때에 따라 매우 복잡하게 작성되기도 합니다. 단순한 분기문을 생각해보더라도 분기문의 조건에 들어갈 값이 1이 더 크고 작음에 따라 로직은 실행될 수도 있고 안될 수도 있습니다.

이렇게 테스트에서 놓칠 수 있는 부분들을 코드 커버리지를 통해 확인할 수 있습니다. 그리고 그에 따라 부족한 테스트를 추가할 수 있습니다. 코드 커버리지는 **휴먼 에러를 최대한 방지**할 수 있도록 도와주는 용도라고 생각해도 될 것 같습니다.

그럼 실제로도 코드 커버리지를 많이 사용하고 있을까요?

많은 서비스 기업에서는 테스트 코드의 중요성을 인지하고 **코드 커버리지를 최대한 유지 및 지속해서 상승**시키면서 개발을 하려고 노력합니다. 코드 커버리지 도구와 소나큐브(SonarQube)와 같은 **정적 코드 분석 도구**를 함께 활용하여 코드 커버리지가 기존보다 떨어지는 경우 커밋(commit)이 불가능하도록 제한하기도 합니다.

이처럼 코드 커버리지는 **코드의 안정성을 어느 정도 보장해 줄 수 있는 지표**이기 때문에 많은 프로젝트에서 커버리지를 확인하고 관리, 적용하려고 노력합니다.

## 자바 코드 커버리지

저희 프로젝트에서는 자바 기반의 프레임워크인 [스프링](https://spring.io/projects/spring-boot) 을 기반으로한 솔루션업체의 개발도구를 사용하고 있습니다.

그래서 이번에는 자바의 코드 커버리지 분석 도구로 **어떤 것들이 존재**하는지, 그리고 저희는 **어떤 분석 도구를 선택했는지** 소개하도록 하겠습니다.

### 자바 코드 커버리지 분석 도구는 어떤것들이

**코드 커버리지 분석 도구**는 앞서 설명한 **코드 커버리지를 개발자가 직접 확인하지 않고 분석할 수 있도록 도와주는 도구**입니다. 저희는 자바로 프로그램을 만들기 때문에 **자바 커버리지 분석 도구**에 대해 알아보도록 하겠습니다.

**자바 코드 커버리지 분석 도구**는 여러가지가 존재하는데, 대표적으로 **Cobertura, Jacoco, Clover** 등이 있습니다.

각각의 도구가 가지는 장단점을 확인하고 싶었으나, 아쉽게도 **Jacoco**를 제외한 나머지는 레퍼런스가 너무 부족하였습니다.

간단하게나마 Jacoco가 가지는 특징을 소개해보면

* 코드 커버리지에 대한 결과를 html이나 xml, csv와 같은 리포트로 생성한다.
* 설정한 커버리지를 만족하는지 여부를 확인할 수 있다.
* 사용 방법이 간단하다.

등이 있습니다.

***Atlassian에서***[**위 세 가지 도구의 차이점을 분석한 글**](https://confluence.atlassian.com/clover/comparison-of-code-coverage-tools-681706101.html)***이 있으니 참고해주시기 바랍니다.***

### 선택하게 된 도구는?

저희 프로젝트에서는 코드 커버리지 분석 도구로 **Jacoco**를 사용하기로 결정했습니다!



선택한 이유는 크게 2가지가 있었습니다.

1. **레퍼런스가 많다(다른 도구의 경우 레퍼런스가 너무 부족했다).**
2. 사용하기가 쉽다.

이 중에서 첫 번째 이유가 선택에 가장 큰 영향을 주었습니다.

새로운 기술을 처음 도입하는 상황에서 **레퍼런스의 유무**는 정말 중요하다고 생각합니다. 기술을 도입하는 과정에서는 **여러 시행착오들이 발생**하는데, 레퍼런스가 많이 존재한다면 이를 **빠르게 해결할 수 있기 때문**입니다. 그런 부분에서 Jacoco는 적용한 후기나 설명이 다른 도구들에 비해 월등하게 많았습니다.

이러한 이유로 인해 이번 프로젝트에서는 Jacoco를 사용하기로 결정하였습니다.

**코드 분석 도구 적용기 - 2편, JaCoCo 적용하기**

## [JaCoCo](https://www.jacoco.org/jacoco/)란?

### JaCoCo

***JaCoCo is a free code coverage library for Java, which has been created by the EclEmma team based on the lessons learned from using and integration existing libraries for many years. -***[**JaCoCo**](https://www.jacoco.org/jacoco/)

**JaCoCo**는 **자바 코드 커버리지를 체크하는 데에 사용되는 오픈소스 라이브러리**입니다.

***JaCoCo의 버전은***[**Maven Central Repository**](https://search.maven.org/search?q=g:org.jacoco)***를 통해 확인할 수 있습니다. 작성일(20.10.05) 기준, 최신 버전은 0.8.6입니다.***

### JaCoCo의 특징

JaCoCo가 가지는 **특징**으로는

* **Line, Branch Coverage를 제공**한다.
* **코드 커버리지 결과**를 보기 좋도록 **파일 형태로 저장**할 수 있다.
  + html, xml, csv 등으로 Report를 생성할 수 있다.
* 설정한 **커버리지 기준을 만족하는지 확인**할 수 있다.

등이 있습니다.

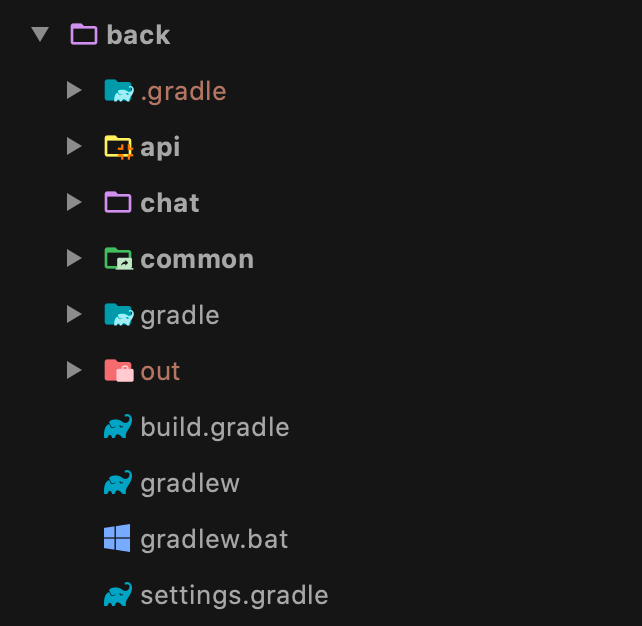
이러한 특징들은 **코드 커버리지를 쉽게 확인하고 관리할 수 있도록 도와줍니다.**

## 프로젝트에 JaCoCo 적용하기

### 개발 환경

* Java 8 🡪 OpenJDK 1.8
* Spring Boot 2.3.1
* Gradle 6.4.1 🡪 Gradle 3.0

현재 본 프로젝트의 개발 환경은 위와 같습니다.



저희 프로젝트 구조의 경우 **api, chat, common,** 총 3개의 모듈로 구성된 **멀티 모듈 프로젝트**입니다.

JaCoCo를 싱글 모듈 프로젝트에 적용하는 방법과 멀티 모듈 프로젝트에 적용하는 방법은 조금 차이가 있습니다. 아쉽지만 이번 글에서는 저희 프로젝트의 구조인 **멀티 모듈 프로젝트**에 적용하는 방법을 보여드리겠습니다.

***일반적인 싱글 모듈 프로젝트에 적용하는 방법은 우아한형제들 기술 블로그에 연철님께서 쓰신***[**Gradle 프로젝트에 JaCoCo 설정하기**](https://woowabros.github.io/experience/2020/02/02/jacoco-config-on-gradle-project.html)***글을 참고하시면 될 것 같습니다.***

### JaCoCo 플러그인 추가

저희 루트 프로젝트의 build.gradle 파일은 아래와 같습니다.

Copy

1plugins {

2 id 'java'

3 id 'org.springframework.boot' version '2.3.1.RELEASE'

4 id 'io.spring.dependency-management' version '1.0.9.RELEASE'

5 id "org.asciidoctor.convert" version "1.5.9.2"

6}

7

8allprojects {

9 group = 'com.jikgorae'

10 version = '0.0.1-SNAPSHOT'

11}

12

13subprojects {

14 apply plugin: 'java'

15 apply plugin: 'org.springframework.boot'

16 apply plugin: 'io.spring.dependency-management'

17 apply plugin: 'org.asciidoctor.convert'

18

19 sourceCompatibility = '1.8'

20

21 repositories {

22 mavenCentral()

23 }

24}

25

26project(':api') {

27 dependencies {

28 compile project(':common')

29 }

30}

31

32project(':chat') {

33 dependencies {

34 compile project(':common')

35 }

36}

저희는 모든 모듈의 테스트에 JaCoCo를 적용하고 싶기 때문에 **JaCoCo 플러그인**을 subprojects 블록에 설정값으로 추가해야 합니다.

apply plugin의 값으로 'jacoco'를 추가하면 됩니다.

Copy

1...

2

3subprojects {

4 apply plugin: 'java'

5 apply plugin: 'org.springframework.boot'

6 apply plugin: 'io.spring.dependency-management'

7 apply plugin: 'org.asciidoctor.convert'

8 apply plugin: 'jacoco' // 추가

9

10 sourceCompatibility = '1.8'

11

12 repositories {

13 mavenCentral()

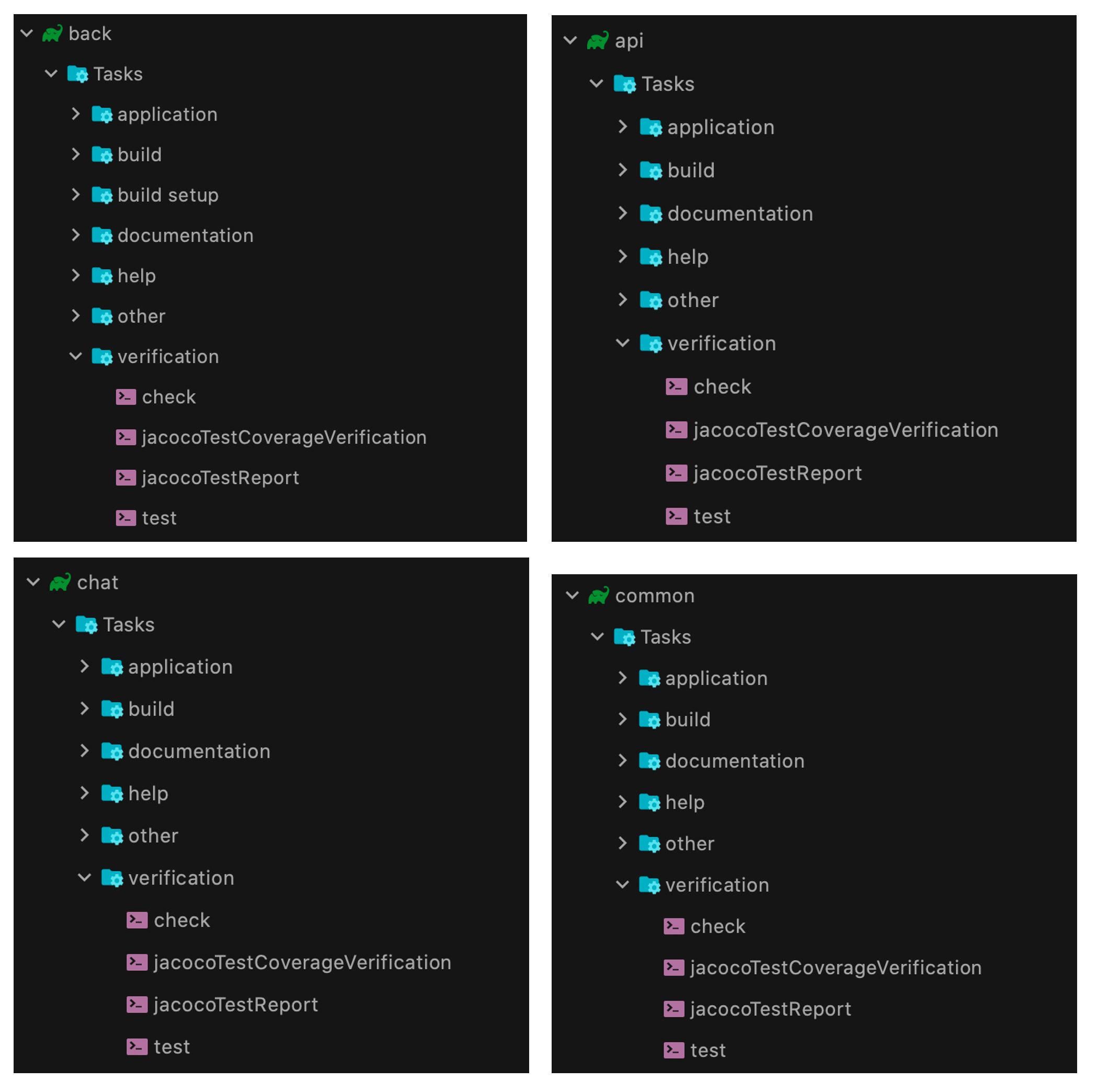
14 }

15}

16

17...

위 설정을 추가한 후, gradle 새로 고침을 실행하면 의존성이 추가되면서 서브 모듈의 Tasks/verification에 JaCoCo의 Task가 추가됩니다.



추가된 jacocoTestReport와 jacocoTestCoverageVerification은 **JaCoCo 플러그인**의 Task입니다. 각 Task의 역할을 간단하게 살펴보면

* jacocoTestReport : 바이너리 커버리지 결과를 사람이 **읽기 좋은 형태의 리포트로 저장**해주는 Task이다.
* jacocoTestCoverageVerification : **원하는 커버리지 기준을 만족하는지 확인**해 주는 Task이다.

정도로 볼 수 있습니다. 추가적인 내용은 각 Task에 대한 설정을 진행하면서 소개하도록 하겠습니다.

### JaCoCo 플러그인 설정하기

위의 두 가지 Task에 대한 설정을 진행하기 전에, [JaCoCo 플러그인 설정](https://docs.gradle.org/current/dsl/org.gradle.testing.jacoco.plugins.JacocoPluginExtension.html)을 먼저 해주어야 합니다.

jacoco라는 이름을 가지는 JaCoCoPluginExtension 타입의 **project extension**을 통해 추가적인 설정을 해줄 수 있으며, 설정해줄 수 있는 속성으로는 reportsDir과 toolVersion이 있습니다.

* reportsDir : 사용할 JaCoCo의 JAR 버전
* toolVersion : Report가 생성될 디렉토리 경로

저희는 여기서 toolVersion만 설정하고 넘어가도록 하겠습니다. toolVersion은 작성일(20.10.05)을 기준으로 가장 최신 버전인 **0.8.6**로 설정하겠습니다.

***reportsDir을 설정하지 않을 경우, ${project.reporting.baseDir}/jacoco 가 기본 경로입니다.***

Copy

1...

2

3subprojects {

4 apply plugin: 'java'

5 apply plugin: 'org.springframework.boot'

6 apply plugin: 'io.spring.dependency-management'

7 apply plugin: 'org.asciidoctor.convert'

8 apply plugin: 'jacoco'

9

10 sourceCompatibility = '1.8'

11

12 repositories {

13 mavenCentral()

14 }

15

16 jacoco {

17 toolVersion = '0.8.6' // 작성일(20.10.05) 기준

18 // reportsDir = ${project.reporting.baseDir}/jacoco

19 }

20}

21

22...

### jacocoTestReports Task 설정하기

이번에는 테스트 결과를 리포트 파일로 저장하는 jacocoTestReports Task의 [설정](https://docs.gradle.org/current/dsl/org.gradle.testing.jacoco.tasks.JacocoReport.html#org.gradle.testing.jacoco.tasks.JacocoReport:reports)을 해야 합니다.

jacocoTestReports Task는 테스트 결과를 html, csv, xml 형태로 저장해줍니다. html의 경우 **사용자가 읽기 편한 파일 형식**이고, csv나 xml의 경우 추후 연동할 **소나큐브(SonarQube) 등에서 사용되는 파일 형식**입니다.

**테스트 결과를 받을 파일 형식**은 jacocoTestReport의 reports 메서드를 통해 설정해 줄 수 있습니다.

Copy

1reports {

2 html {

3 enabled false

4 }

5 csv {

6 enabled true

7 }

8}

추가적으로 **파일 형식에 따라 저장하는 경로를 다르게** 할 수 있는 방법도 있는데 설정 방법은 아래와 같습니다.

Copy

1reports {

2 html {

3 enable true

4 destination file('build/reports/myReport.html')

5 }

6}

***destination은 File 타입의 값만 받기 때문에 file('저장할 디렉토리 경로')과 같은 형식으로 작성해야 합니다.***

Copy

1reports {

2 html.enabled false

3 html.destination file('build/reports/myReport.html')

4 csv.enabled true

5}

또는 위의 예제와 같이 좀 더 간단한 방법으로도 설정이 가능합니다.

저희 프로젝트에서는 팀원이 테스트 결과를 보기 쉽고 추후 소나큐브에서 사용할 수 있게 **html과 csv**, 두 가지 파일 형식을 설정하도록 하겠습니다.

Copy

1subprojects {

2 apply plugin: 'java'

3 apply plugin: 'org.springframework.boot'

4 apply plugin: 'io.spring.dependency-management'

5 apply plugin: 'org.asciidoctor.convert'

6 apply plugin: 'jacoco'

7

8 sourceCompatibility = '1.8'

9

10 repositories {

11 mavenCentral()

12 }

13

14 jacoco {

15 toolVersion = '0.8.6'

16 }

17

18 jacocoTestReport {

19 reports {

20 html.enabled true // html 설정

21 csv.enabled true // csv 설정

22 xml.enabled false // xml 미설정

23 }

24 }

25}

### jacocoTestCoverageVerification Task 설정하기

이번에는 **원하는 코드 커버리지를 설정하고, 커버리지를 만족하는지 여부를 확인**할 수 있는 jacocoTestCoverageVerification Task의 [설정](https://docs.gradle.org/current/dsl/org.gradle.testing.jacoco.tasks.JacocoCoverageVerification.html)을 살펴보겠습니다.

jacocoTestCoverageVerification Task는 **최소 코드 커버리지 수준을 설정**할 수 있고, 이를 통과하지 못할 경우 Task가 **실패**하게 됩니다.

jacocoTestCoverageVerification의 violationRules 메서드를 통해 **커버리지 기준을 설정하는 룰을 정의**할 수 있고, 각각의 룰에 대한 설정은 violationRules 메서드에 전달할 rule 메서드를 통해 정의할 수 있습니다.

depth가 깊다 보니 설명만으로는 이해하기 힘든 것 같습니다. 실제 코드를 통해 rule 메서드에 적용할 수 있는 값들이 어떤 것이 있는지 알아보도록 하겠습니다.

Copy

1jacocoTestCoverageVerification {

2 violationRules {

3 rule {

4 enable = true

5 element = 'CLASS'

6 // includes = []

7

8 limit {

9 counter = 'BRANCH'

10 value = 'COVEREDRATIO'

11 minimum = 0.90

12 }

13

14 limit {

15 counter = 'LINE'

16 value = 'COVEREDRATIO'

17 minimum = 0.80

18 }

19

20 limit {

21 counter = 'LINE'

22 value = 'TOTALCOUNT'

23 maximum = 200

24 }

25

26 // excludes = []

27 }

28

29 // 여러 rule을 생성할 수 있습니다.

30 rule {

31 ...

32 }

33 }

34}

#### enable

해당하는 rule의 **활성화 여부**를 boolean으로 나타냅니다. 값을 지정하지 않는 경우 Default 값은 **true**입니다.

#### [element](https://www.eclemma.org/jacoco/trunk/doc/api/org/jacoco/core/analysis/ICoverageNode.ElementType.html)

**커버리지를 체크할 기준(단위)**을 정할 수 있으며, 총 6개의 기준이 존재합니다.

* BUNDLE : 패키지 번들(프로젝트 모든 파일을 합친 것)
* CLASS : 클래스
* GROUP : 논리적 번들 그룹
* METHOD : 메서드
* PACKAGE : 패키지
* SOURCEFILE : 소스 파일

값을 지정하지 않는 경우 Default 값은 **BUNDLE**입니다.

#### includes

해당하는 rule을 **적용 대상**을 package 수준으로 정의할 수 있습니다. 값을 지정하지 않는 경우 Default 값은 **전체 package** 입니다.

#### [counter](https://www.eclemma.org/jacoco/trunk/doc/api/org/jacoco/core/analysis/ICoverageNode.CounterEntity.html)

counter는 limit 메서드를 통해 지정할 수 있으며 **커버리지 측정의 최소 단위**를 말합니다. 이때 측정은 java byte code가 실행된 것을 기준으로 측정되고, 총 6개의 단위가 존재합니다.

* BRANCH : 조건문 등의 분기 수
* CLASS : 클래스 수, 내부 메서드가 한 번이라도 실행된다면 실행된 것으로 간주한다.
* COMPLEXITY : [복잡도](https://www.eclemma.org/jacoco/trunk/doc/counters.html)
* INSTRUCTION : Java 바이트코드 명령 수
* METHOD : 메서드 수, 메서드가 한 번이라도 실행된다면 실행된 것으로 간주한다.
* LINE : 빈 줄을 제외한 실제 코드의 라인 수, 라인이 한 번이라도 실행되면 실행된 것으로 간주한다.

값을 지정하지 않는 경우 Default 값은 **INSTRUCTION**입니다.

#### [value](https://www.eclemma.org/jacoco/trunk/doc/api/org/jacoco/core/analysis/ICounter.CounterValue.html)

value는 limit 메서드를 통해 지정할 수 있으며 **측정한 커버리지를 어떠한 방식으로 보여줄 것**인지를 말합니다. 총 5개의 방식이 존재합니다.

* COVEREDCOUNT : 커버된 개수
* COVEREDRATIO : 커버된 비율, 0부터 1사이의 숫자로 1이 100%이다.
* MISSEDCOUNT : 커버되지 않은 개수
* MISSEDRATIO : 커버되지 않은 비율, 0부터 1사이의 숫자로 1이 100%이다.
* TOTALCOUNT : 전체 개수

값을 지정하지 않은 경우 Default 값은 **COVEREDRATIO**입니다.

#### minimum

minimum은 limit 메서드를 통해 지정할 수 있으며 counter 값을 value에 맞게 표현했을 때 **최솟값**을 말합니다. 이 값을 통해 jacocoTestCoverageVerification의 **성공 여부가 결정**됩니다.

해당 값은 BigDecimal 타입이고 **표기한 자릿수만큼 value가 출력**됩니다. 만약 커버리지를 80%를 원했는데 0.80이 아니라 0.8을 입력하면 커버리지가 0.87이라도 0.8로 표시됩니다.

minimum은 Default 값이 존재하지 않습니다.

#### excludes

커버리지를 측정할 때 **제외할 클래스**를 지정할 수 있습니다. 패키지 레벨의 경로로 지정하여야 하고 경로에는 \*와 ?를 사용할 수 있습니다.

지금까지 rule 메서드에 적용할 수 있는 **설정값**들에 대해 알아보았습니다. 학습한 내용을 바탕으로 저희 프로젝트의 jacocoTestCoverageVerification Task를 설정해보겠습니다.

Copy

1subprojects {

2 apply plugin: 'java'

3 apply plugin: 'org.springframework.boot'

4 apply plugin: 'io.spring.dependency-management'

5 apply plugin: 'org.asciidoctor.convert'

6 apply plugin: 'jacoco'

7

8 sourceCompatibility = '1.8'

9

10 repositories {

11 mavenCentral()

12 }

13

14 jacoco {

15 toolVersion = '0.8.6'

16 }

17

18 jacocoTestReport {

19 reports {

20 html.enabled true

21 csv.enabled true

22 xml.enabled false

23 }

24 }

25

26 jacocoTestCoverageVerification {

27 violationRules {

28 rule {

29 enabled = true // 활성화

30 element = 'CLASS' // 클래스 단위로 커버리지 체크

31 // includes = []

32

33 // 라인 커버리지 제한을 80%로 설정

34 limit {

35 counter = 'LINE'

36 value = 'COVEREDRATIO'

37 minimum = 0.80

38 }

39

40 // 브랜치 커버리지 제한을 80%로 설정

41 limit {

42 counter = 'BRANCH'

43 value = 'COVEREDRATIO'

44 minimum = 0.80

45 }

46

47 excludes = []

48 }

49 }

50 }

51}

### test Task 설정하기

JaCoCo 플러그인은 모든 test 타입의 Task에 JacocoTaskExtension을 추가하고, **test Task에서 그 설정을 변경**할 수 있게 합니다.

test Task에 Default로 설정된 값은 아래와 같습니다.

Copy

1test {

2 jacoco {

3 enabled = true

4 destinationFile = file("$buildDir/jacoco/${name}.exec")

5 includes = []

6 excludes = []

7 excludeClassLoaders = []

8 includeNoLocationClasses = false

9 sessionId = "<auto-generated value>"

10 dumpOnExit = true

11 classDumpDir = null

12 output = JacocoTaskExtension.Output.FILE

13 address = "localhost"

14 port = 6300

15 jmx = false

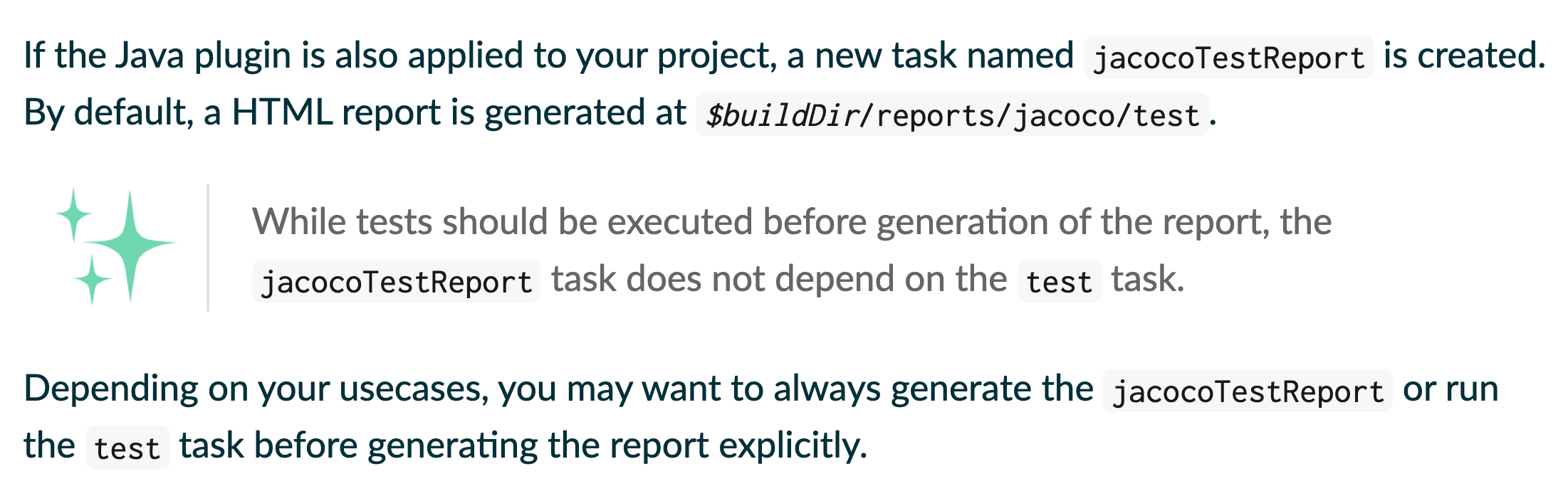
16 }

17}

각 설정값은 오버라이드가 가능합니다. 저희 프로젝트에는 기본값에서 변경할 내용이 없기 때문에 넘어가도록 하겠습니다.

### JaCoCo 플러그인 Task 순서 설정하기

이제 마지막 관문에 도착했습니다! 지금까지 살펴본 JaCoCo 플러그인의 Task들에 **의존성을 설정**해야 하는데 어떻게 해야하는지 알아보겠습니다.



JaCoCo 플러그인의 [User Guide 문서](https://docs.gradle.org/current/userguide/jacoco_plugin.html#sec:jacoco_getting_started)를 보면 jacocoTestReport Task로 리포트를 생성하기 **이전에** test Task가 실행되어야 하는데, jacocoTestReport Task에는 test Task와의 **의존성이 설정되어있지 않다**고 합니다.

위의 의존성 설정은 test Task의 설정에 **한 줄만 추가**하면 해결할 수 있습니다. 이 설정만 추가하면 끝인..줄 알았지만 아직 한 가지 설정이 더 남았습니다.

바로 **jacocoTestReport Task와 jacocoTestCoverageVerification Task 간의 순서**입니다.

앞서 jacocoTestReport는 **리포트를 생성하는 Task**, jacocoTestCoverageVerification는 **설정한 커버리지를 만족하는지 확인하는 Task**라고 설명했습니다. 만약, 이 둘의 순서를 지정하지 않아서 jacocoTestCoverageVerification Task가 jacocoTestReport Task보다 **먼저 실행되면** 어떤 일이 발생할까요?

사실 엄청나게 큰일이 나는 것은 아닙니다 🤣

단지 jacocoTestCoverageVerification Task가 먼저 실행되고 설정한 커버리지를 통과하지 못하면 **gradle 빌드가 멈추게 됩니다.** 그렇다면 jacocoTestReport Task는 실행되지 않게 되고, **리포트가 생성되지 않으면서 저희는 이전 테스트에서 생성된 리포트를 보는 경우가 발생**할 수 있기 때문입니다.

그렇다면 저희는 **test -> jacocoTestReport -> jacocoTestCoverageVerification 순서로 Task를 실행**하면 위의 모든 조건을 만족할 수 있습니다. 이 순서는 finalizedBy 라는 메서드를 사용하여 설정할 수 있습니다.

저희 프로젝트의 Task를 위 순서대로 설정해보도록 하겠습니다.

Copy

1subprojects {

2 apply plugin: 'java'

3 apply plugin: 'org.springframework.boot'

4 apply plugin: 'io.spring.dependency-management'

5 apply plugin: 'org.asciidoctor.convert'

6 apply plugin: 'jacoco'

7

8 sourceCompatibility = '1.8'

9

10 repositories {

11 mavenCentral()

12 }

13

14 test {

15 useJUnitPlatform() // JUnit5를 사용하기 위한 설정

16 finalizedBy 'jacocoTestReport' // 추가

17 }

18

19 jacoco {

20 toolVersion = '0.8.6'

21 }

22

23 jacocoTestReport {

24 reports {

25 html.enabled true

26 csv.enabled true

27 xml.enabled false

28 }

29 finalizedBy 'jacocoTestCoverageVerification' // 추가

30 }

31

32 jacocoTestCoverageVerification {

33 violationRules {

34 rule {

35 enabled = true

36 element = 'CLASS'

37

38 limit {

39 counter = 'LINE'

40 value = 'COVEREDRATIO'

41 minimum = 0.80

42 }

43

44 limit {

45 counter = 'BRANCH'

46 value = 'COVEREDRATIO'

47 minimum = 0.80

48 }

49

50 excludes = []

51 }

52 }

53 }

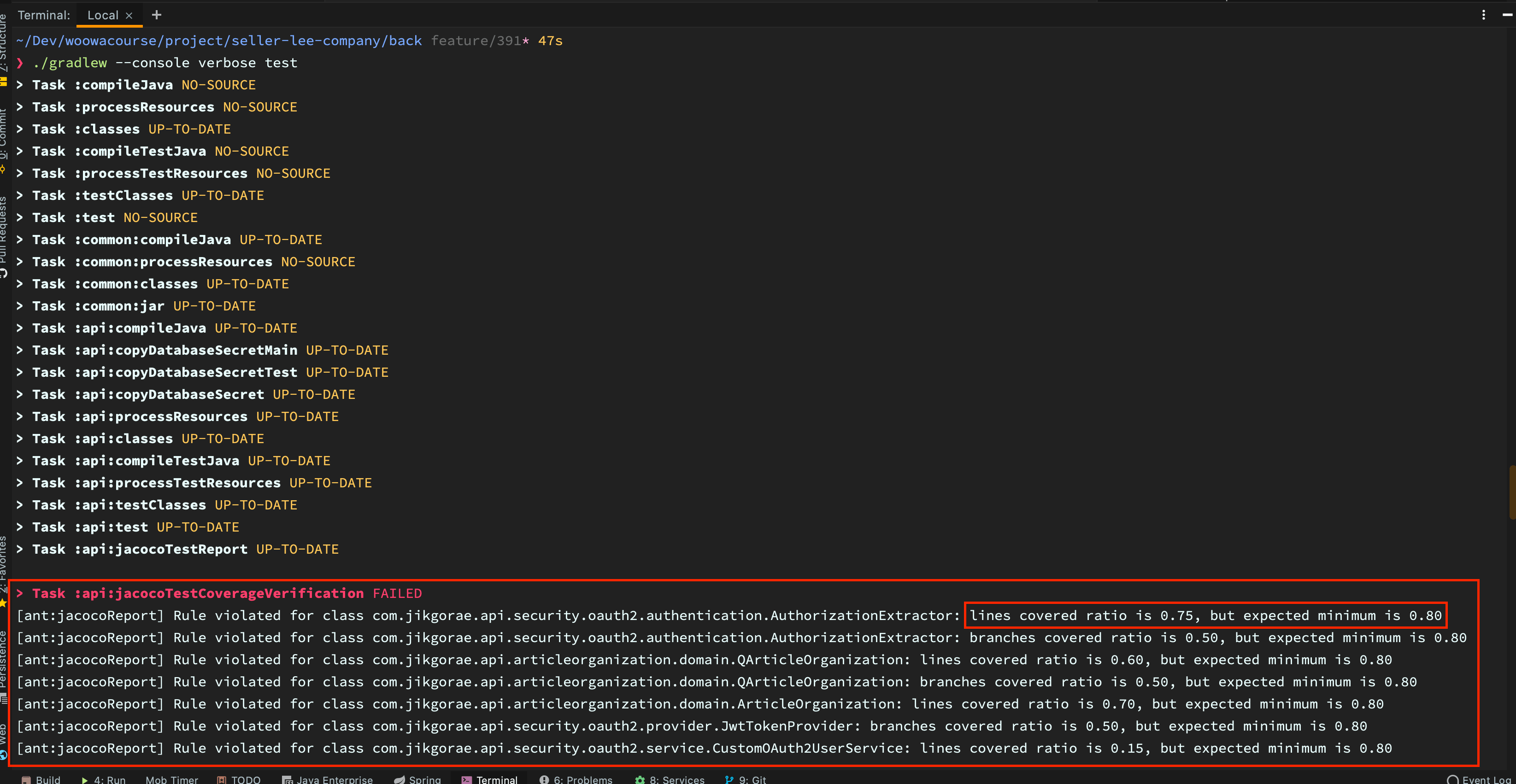
54}

### 테스트 실행하기

드디어, 긴 설정을 마무리하고 JaCoCo 테스트를 돌려볼 시간입니다.

./gradlew test 명령을 사용해서 test Task를 실행해보도록 하겠습니다.

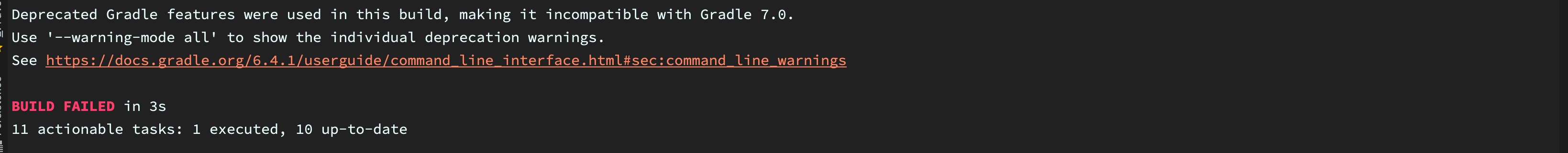
***명령어에 --console verbose 옵션을 추가하면 실행되는 Task를 함께 볼 수 있습니다.***



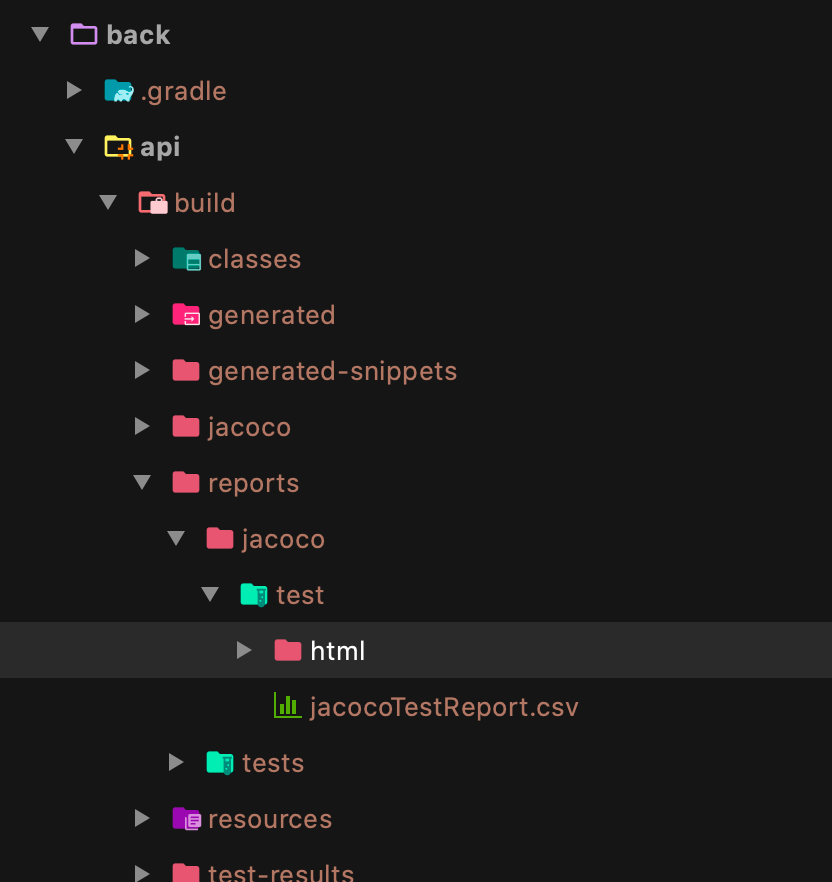
실행한 결과 jacocoTestCoverageVerification Task가 실패하였습니다. 결과를 살펴보면 **설정한 커버리지를 만족하지 못한 클래스와 해당 클래스의 커버리지 결과를 함께 보여줍니다.**

실행 결과에 표시해둔 빨간 네모 속 네모를 보면 AuthorizationExtractor 클래스의 라인 커버리지가 0.75이고 제가 설정한 커버리지는 0.80이기 때문에 통과하지 못한 것을 볼 수 있습니다.

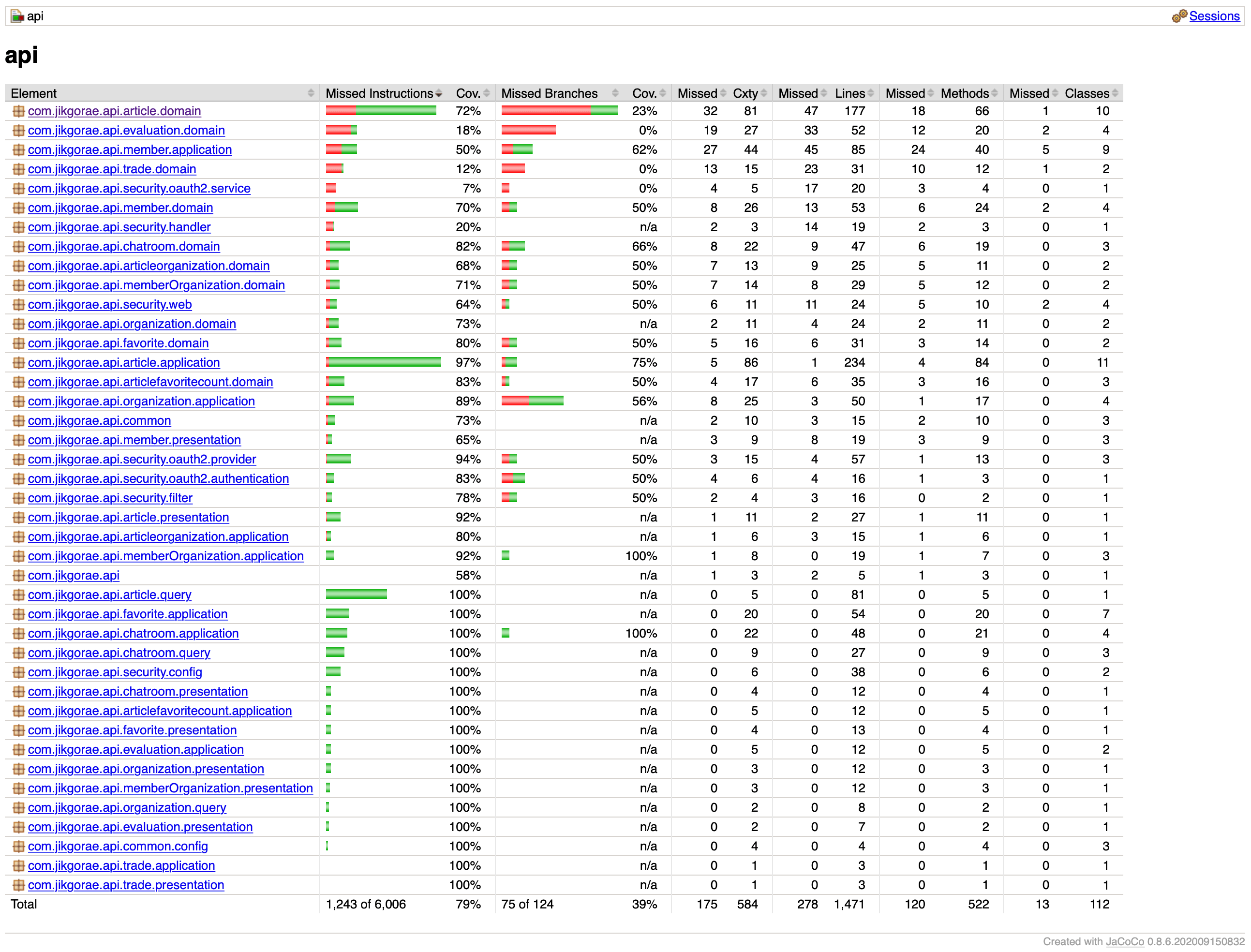
이렇게 커버리지를 통과하지 못한 클래스가 존재하면 **gradle 빌드는 실패**하게 됩니다.

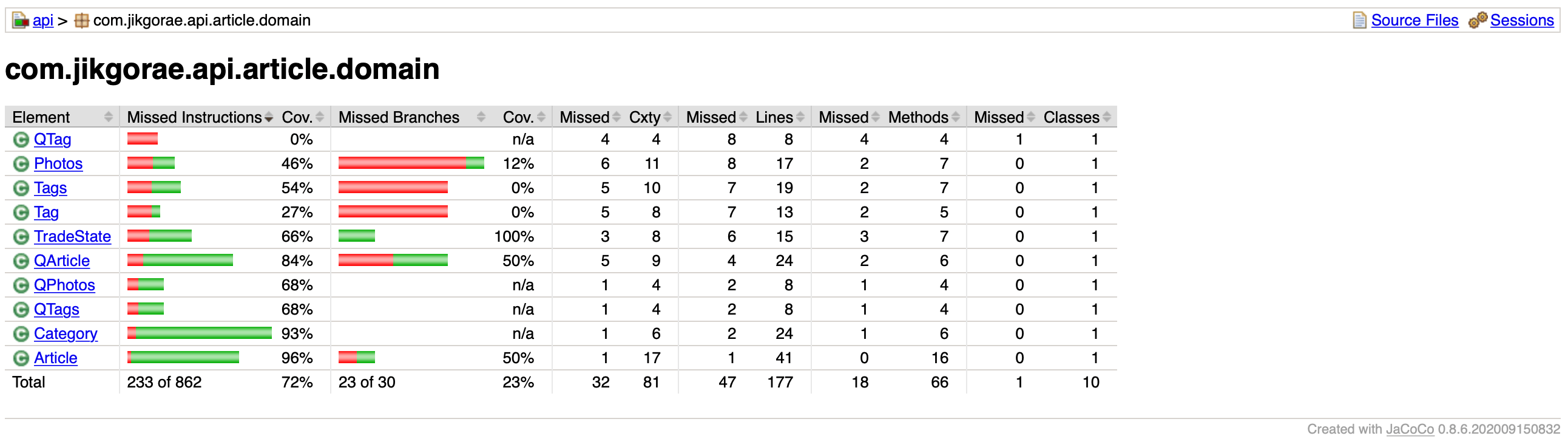


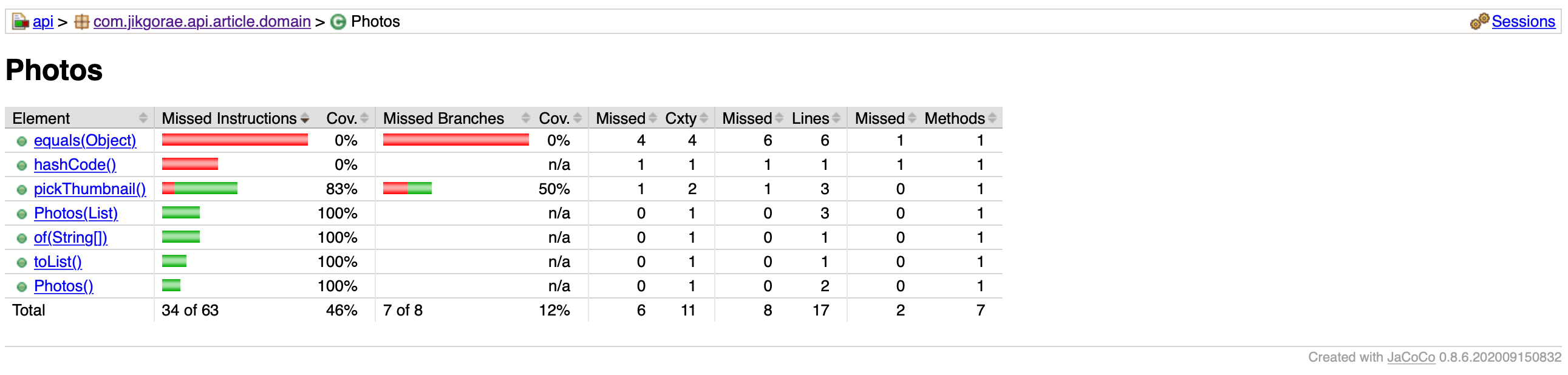
빌드는 실패했지만 앞서 Task의 순서를 설정해두었기 때문에 리포트는 정상적으로 생성되었을 것입니다.



리포트는 생성되는 위치를 설정하지 않았기 때문에 Default 경로인 build/reports/jacoco/test/html/index.html에 저장되었습니다.

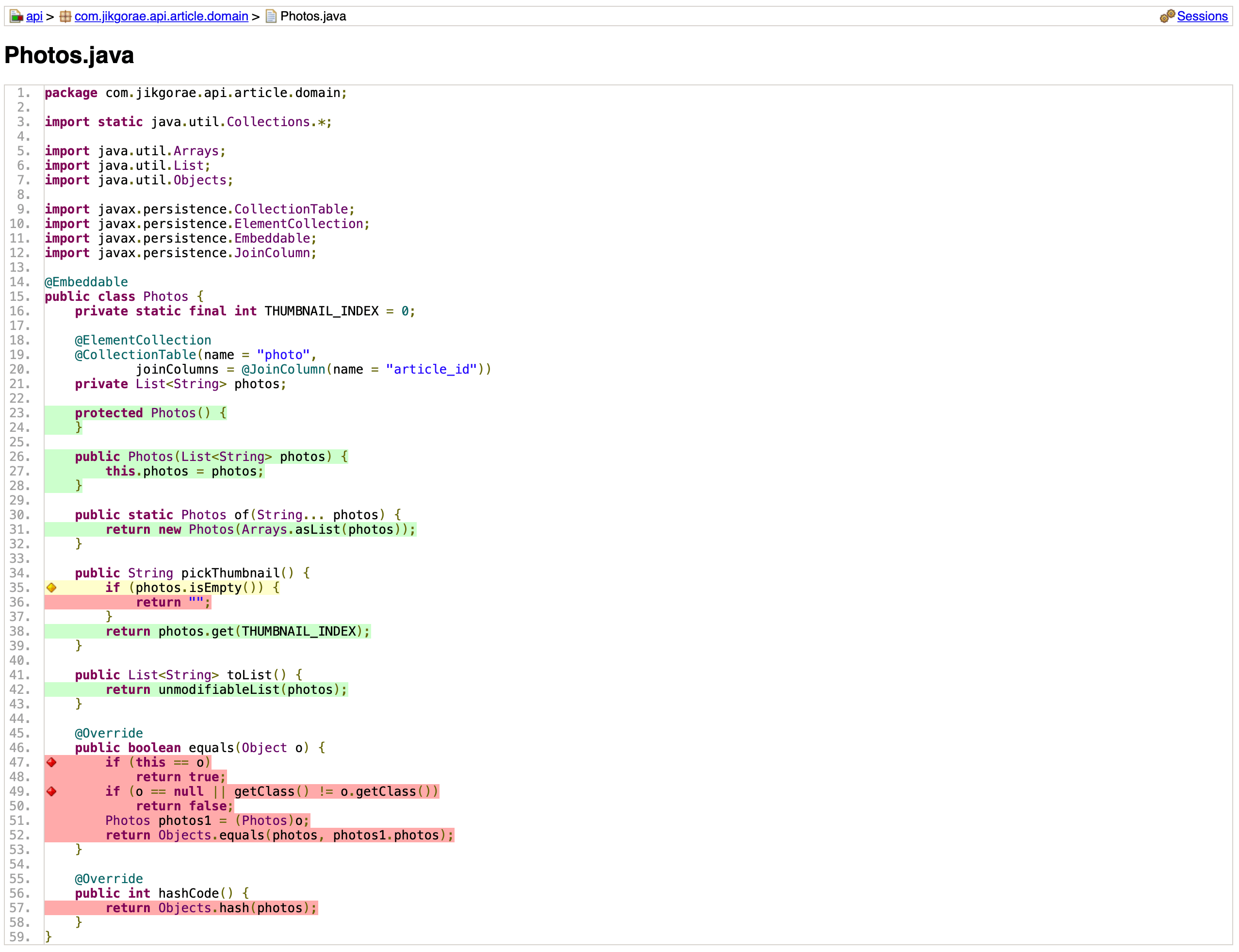




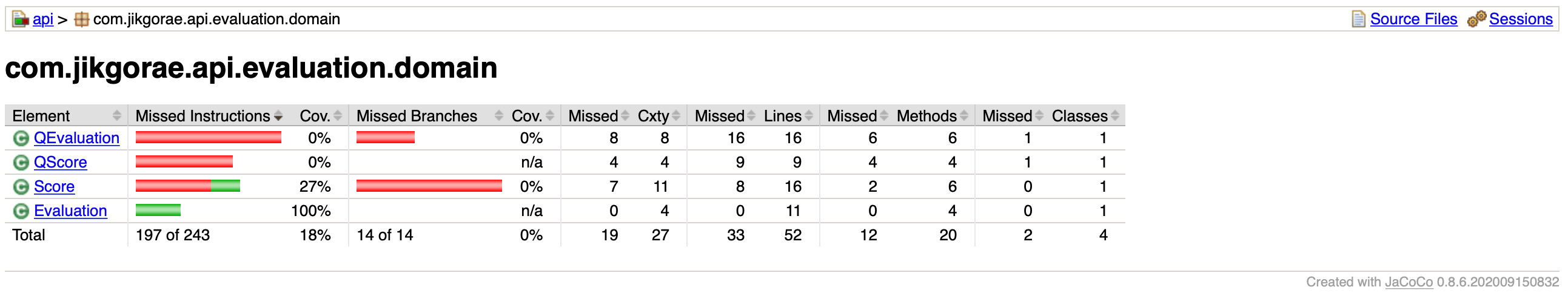


만들어진 html 리포트는 각 커버리지 항목마다 **총 개수와 놓친 개수**를 표시해줍니다. 저희 프로젝트에서는 라인 커버리지와 브랜치 커버리지를 모두 확인하기 때문에 각각의 커버리지 정보도 함께 표시되어 있습니다.

리포트에서 클래스의 메서드를 클릭하면 해당 메서드에서 커버된 라인들을 확인할 수도 있습니다.



### 커버리지에 제외할 클래스 설정하기



api 서브 모듈에 evaluation.domain 패키지의 커버리지를 보면 QEvaluation 클래스와 QScore 클래스가 **전혀 커버되지 않음**을 볼 수 있습니다.

저희 프로젝트의 api 서브 모듈에서는 **QueryDSL**을 사용하고 있습니다. 위 예시의 QEvaluation 클래스와 QScore 클래스는 QueryDSL가 자동으로 생성한 **Qdomain 클래스**입니다.

해당 클래스는 커버리지에서 확인할 필요가 없기 때문에 저희 프로젝트에서 Qdomain 클래스를 **커버리지 측정 대상에서 제외하는 설정**을 추가해보겠습니다.

***지금 소개할 QueryDSL의 Qdomain 클래스를 커버리지 측정 대상에서 제외하는 설정은 우테코 크루인***[**비밥의 블로그**](https://bottom-to-top.tistory.com/36)***의 내용을 참고했습니다.***

#### 커버리지 측정에서 제외하는 설정

먼저 제외할 Qdomain 클래스 명을 **리스트로 만드는 방법**에 대해 알아보겠습니다.

***해당 리스트는 jacocoTestCoverageVerification Task에 excludes 속성의 값으로 사용됩니다.***

***이때 사용할 값은 \*와 ?를 통해 패턴처럼 설정할 수 있습니다.***

**Qdomain 클래스 명**의 **Prefix는 Q**입니다. 이를 보고 단순하게 '\*.Q\*'를 excludes 값을 지정하면 **예상치 못한 문제**가 발생하게 됩니다.

바로 **Q로 시작하는 domain 클래스**들, 예를 들어 Question, Quality, Quantity 같은 domain 클래스도 **함께 제외**된다는 것입니다.

우테코 크루인 [비밥의 블로그](https://bottom-to-top.tistory.com/36)에서는 이를 **Qdomain 클래스 명의 특징**을 통해 센스있게 해결하였는데, 저희도 이 방법을 사용하도록 하겠습니다.

Qdomain의 경우 기존의 domain 클래스 명에 Prefix로 Q가 붙는다고 하였습니다. 기존의 domain 클래스 명도 **첫문자는 대문자**이기 때문에 QScore와 같은 Qdomain 클래스 명을 가지게 됩니다. 즉 **Q + 알파벳 대문자**의 형식으로 Qdomain 클래스 명을 가지게 됩니다.

그래서 **'\*.QA\*'부터 '\*.QZ\*'까지** 모두 excludes 값으로 지정하면 앞서 설명한 Side-effect를 방지할 수 있습니다.

또한 Gradle의 Groovy 문법은 이 방법을 생각보다 쉽게 구현할 수 있습니다.

Copy

1def Qdomains = []

2

3for (qPattern in '\*.QA'..'\*.QZ') { // qPattern = '\*.QA', '\*.QB', ... '\*.QZ'

4 Qdomains.add(qPattern + '\*')

5}

위의 코드는 결론적으로 **'\*.QA\*'부터 '\*.QZ\*'까지의 모든 값**을 만들어서 **Qdomains 리스트에 저장**합니다(이 코드는 아래의 **리포트를 작성할 때 제외하는 설정**에서도 조금만 변경해서 사용합니다).

그럼 이제 jacocoTestCoverageVerification Task에 excludes 값으로 지정하면 됩니다.

Copy

1subprojects {

2 apply plugin: 'java'

3 apply plugin: 'org.springframework.boot'

4 apply plugin: 'io.spring.dependency-management'

5 apply plugin: 'org.asciidoctor.convert'

6 apply plugin: 'jacoco'

7

8 sourceCompatibility = '1.8'

9

10 repositories {

11 mavenCentral()

12 }

13

14 test {

15 useJUnitPlatform()

16 finalizedBy 'jacocoTestReport'

17 }

18

19 jacoco {

20 toolVersion = '0.8.6'

21 }

22

23 jacocoTestReport {

24 reports {

25 html.enabled true

26 csv.enabled true

27 xml.enabled false

28 }

29 finalizedBy 'jacocoTestCoverageVerification'

30 }

31

32 jacocoTestCoverageVerification {

33 def Qdomains = []

34

35 for (qPattern in '\*.QA'..'\*.QZ') { // qPattern = '\*.QA', '\*.QB', ... '\*.QZ'

36 Qdomains.add(qPattern + '\*')

37 }

38

39 violationRules {

40 rule {

41 enabled = true

42 element = 'CLASS'

43

44 limit {

45 counter = 'LINE'

46 value = 'COVEREDRATIO'

47 minimum = 0.80

48 }

49

50 limit {

51 counter = 'BRANCH'

52 value = 'COVEREDRATIO'

53 minimum = 0.80

54 }

55

56 excludes = [] + Qdomains // 제외할 Qdomains 패턴 추가

57 }

58 }

59 }

60}

#### 리포트를 작성할 때 제외하는 설정

**리포트를 작성할 때 제외**하기 위해서는 리포트를 작성해주는 jacocoTestReport Task에 설정을 추가하여야 합니다.

아래 설정에 대한 코드를 보면서 설명해드리도록 하겠습니다.

***Qdomains 리스트를 생성하는 로직은 '\*.QA'..'\*.QZ'을 '\*\*/QA'..'\*\*/QZ'로만 변경하면 됩니다.***

Copy

1jacocoTestReport {

2 reports {

3 html.enabled true

4 csv.enabled true

5 xml.enabled false

6 }

7

8 def Qdomains = []

9

10 for (qPattern in '\*\*/QA'..'\*\*/QZ') { // qPattern = '\*.QA', '\*.QB', ... '\*.QZ'

11 Qdomains.add(qPattern + '\*')

12 }

13

14 // 여기부터

15 afterEvaluate {

16 classDirectories.setFrom(

17 files(classDirectories.files.collect {

18 fileTree(dir: it, excludes: [] + Qdomains)

19 })

20 )

21 }

22

23 finalizedBy 'jacocoTestCoverageVerification'

24}

***이 부분은 제가 정확히 이해하지 못한 부분이 많이 있습니다. 틀린 부분이 보이면 언제든지 지적해주시면 감사하겠습니다 🙇🏻‍♂️***

[afterEvaluate](https://docs.gradle.org/current/userguide/build_lifecycle.html#sec:project_evaluation)는 gradle의 빌드 라이프 사이클에 대한 메서드입니다. 프로젝트가 평가된 후 실행할 수 있도록 도와줍니다.

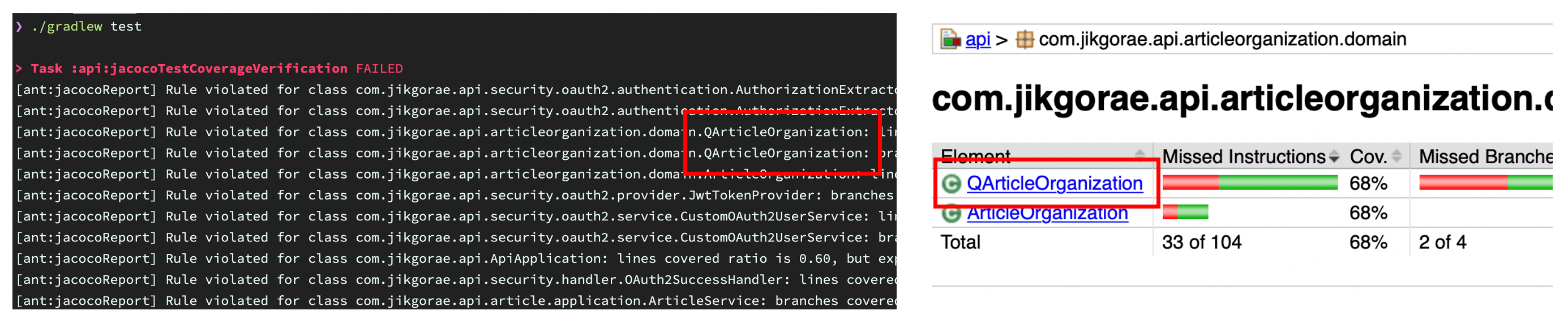
[classDirectories](https://docs.gradle.org/current/dsl/org.gradle.testing.jacoco.tasks.JacocoReport.html#org.gradle.testing.jacoco.tasks.JacocoReport:classDirectories)는 **커버리지가 리포트로 작성할 소스 파일**을 말합니다. 여기서는 setFrom 메서드를 통해 이를 설정해줍니다.

[files](https://docs.gradle.org/current/javadoc/org/gradle/api/Project.html#files-java.lang.Object...-)는 지정된 파일을 포함하는 ConfigurableFileCollection 타입을 반환합니다.

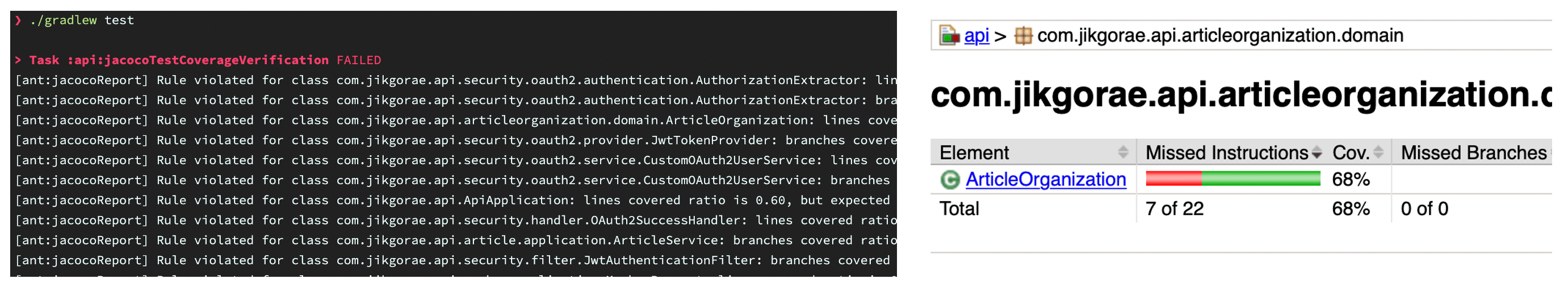
files 내부 코드는 정확하지는 않지만, 기존의 classDirectories의 파일들을 돌면서 각 파일을 통해 계층 구조로 된 파일 컬렉션인 fileTree을 생성합니다. 그리고 excludes로 **Qdomains 리스트를 지정**합니다.

위 과정을 거치면 저희가 받는 리포트에는 **Qdomain 클래스가 모두 제외**되게 됩니다. 그럼 ./gradlew test를 실행 시켜 결과를 확인해보도록 하겠습니다.

먼저 위의 설정을 적용하기 전은



위의 설정을 적용한 결과는



와 같습니다.

실행 결과를 살펴보면 기존에 존재한 QArticleOrganization 클래스가 test Task를 실행한 결과와 리포트 결과에서 **제외되었음**을 알 수 있습니다.

#### 참고. @Generated로 생성된 코드 제외하기

현재 저희 프로젝트에서는 [lombok](https://projectlombok.org/)을 사용하고 있지 않습니다. 그러나 lombok을 사용하는 프로젝트의 경우에는 generated code를 커버리지에서 제외하고 싶을 수 있습니다.

이를 해결하는 방법은 매우 간단합니다.

프로젝트의 루트에서 lombok.config 파일을 생성하고 lombok.addLombokGeneratedAnnotation = true 코드를 한 줄을 추가해주면 끝!

### 최종 설정 확인하기

드디어 기나긴 여정이 끝났습니다! 지금까지의 모든 설정을 적용한 gradle.build 파일은 아래와 같습니다.

Copy

1...

2

3subprojects {

4 apply plugin: 'java'

5 apply plugin: 'org.springframework.boot'

6 apply plugin: 'io.spring.dependency-management'

7 apply plugin: 'org.asciidoctor.convert'

8 apply plugin: 'jacoco'

9

10 sourceCompatibility = '1.8'

11

12 repositories {

13 mavenCentral()

14 }

15

16 test {

17 useJUnitPlatform()

18 finalizedBy 'jacocoTestReport'

19 }

20

21 jacoco {

22 toolVersion = '0.8.6'

23 }

24

25 jacocoTestReport {

26 reports {

27 html.enabled true

28 csv.enabled true

29 xml.enabled false

30 }

31

32 def Qdomains = []

33

34 for (qPattern in '\*\*/QA'..'\*\*/QZ') { // qPattern = '\*.QA', '\*.QB', ... '\*.QZ'

35 Qdomains.add(qPattern + '\*')

36 }

37

38 afterEvaluate {

39 classDirectories.setFrom(

40 files(classDirectories.files.collect {

41 fileTree(dir: it, excludes: [] + Qdomains)

42 })

43 )

44 }

45

46 finalizedBy 'jacocoTestCoverageVerification'

47 }

48

49 jacocoTestCoverageVerification {

50 def Qdomains = []

51

52 for (qPattern in '\*.QA'..'\*.QZ') { // qPattern = '\*.QA', '\*.QB', ... '\*.QZ'

53 Qdomains.add(qPattern + '\*')

54 }

55

56 violationRules {

57 rule {

58 enabled = true

59 element = 'CLASS'

60

61 limit {

62 counter = 'LINE'

63 value = 'COVEREDRATIO'

64 minimum = 0.00

65 }

66

67 limit {

68 counter = 'BRANCH'

69 value = 'COVEREDRATIO'

70 minimum = 0.00

71 }

72

73 excludes = [] + Qdomains

74 }

75 }

76 }

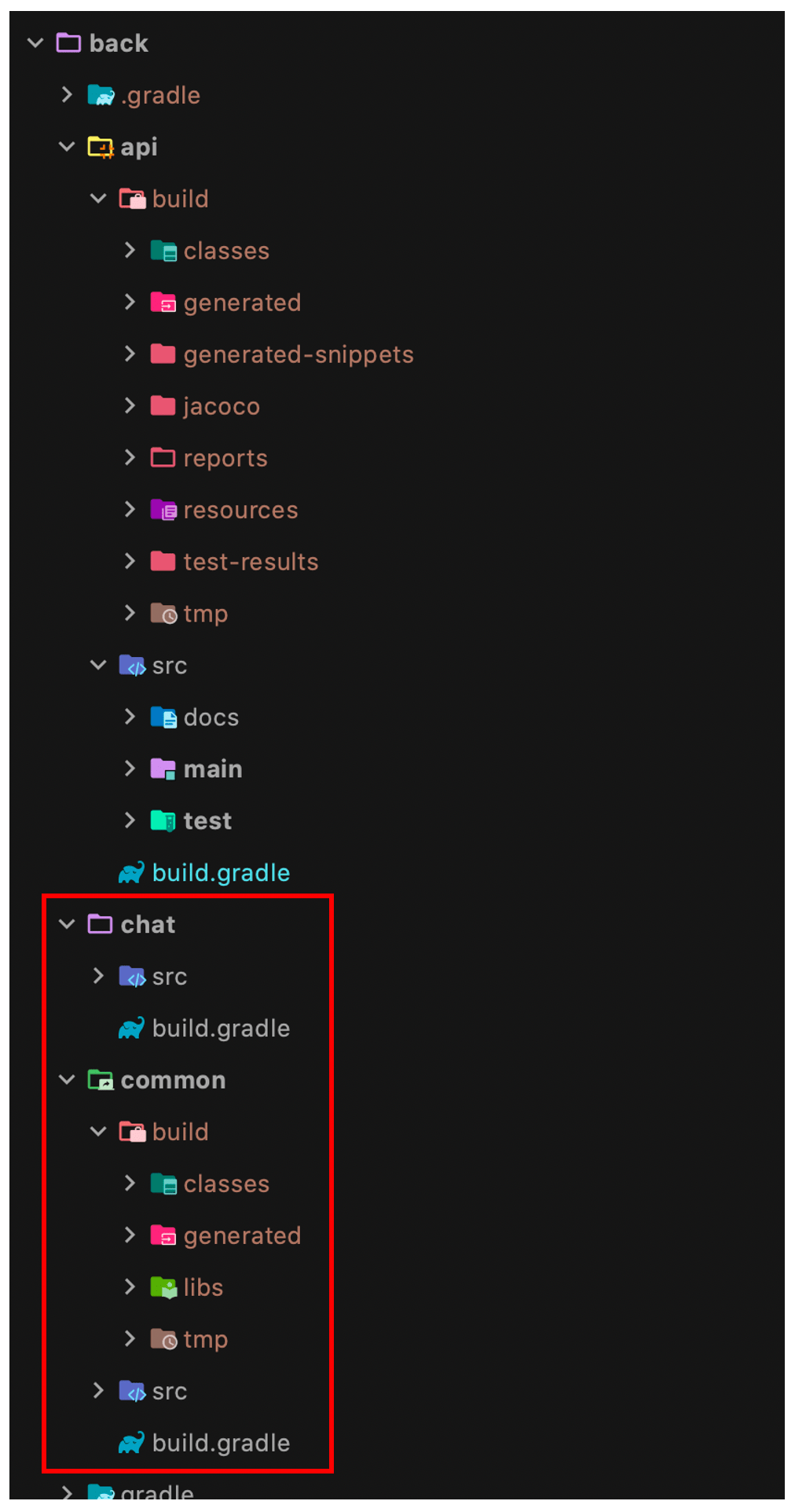
77}

78

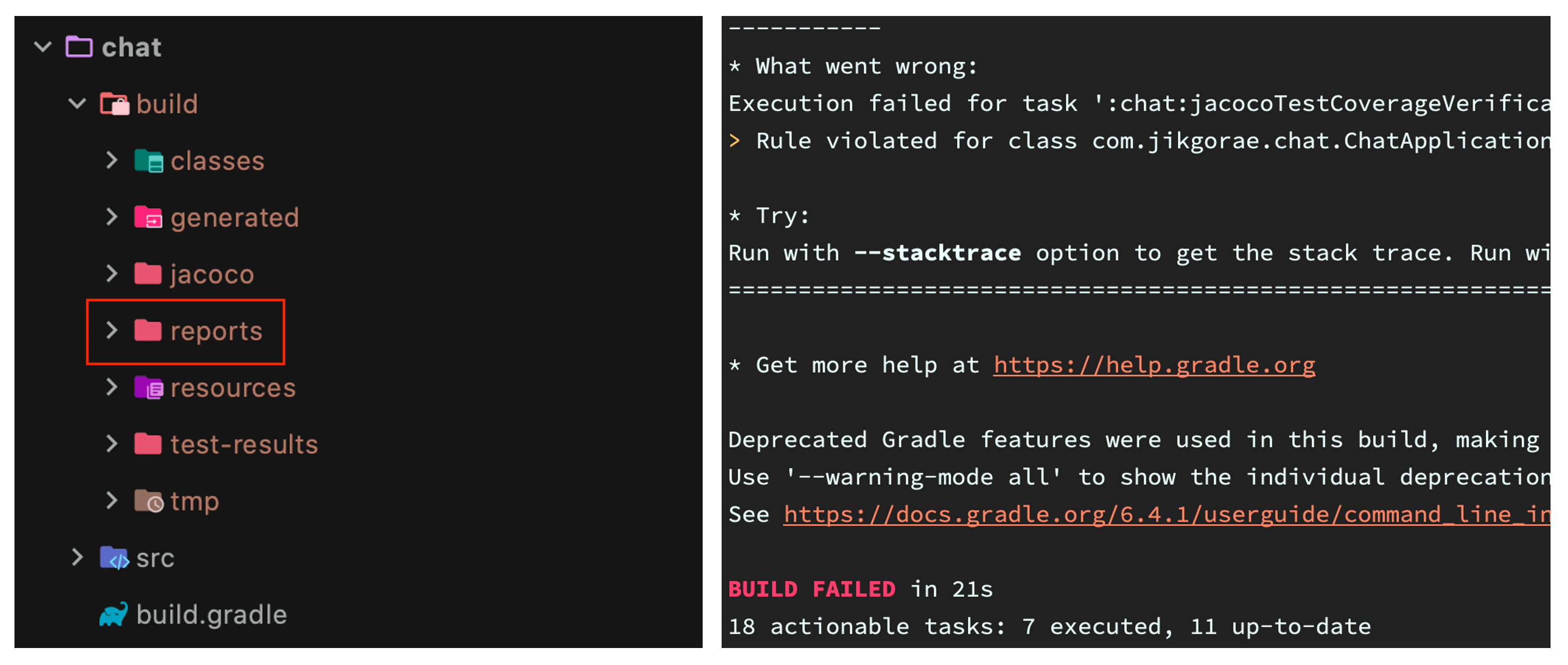
79...

현재 저희 프로젝트에서는 테스트 커버리지를 만족하지 못하는 클래스가 존재하기 때문에 **빌드가 실패**합니다. 그런데 gradle의 Task는 하나의 테스크가 실패하면 뒤에 실행되야 할 나머지 Task가 실행되지 않습니다.

저희 프로젝트는 **멀티 모듈 프로젝트**인데 **api** 모듈의 test Task가 실패하면 이후 **chat** 모듈과 **common** 모듈에 대한 test Task는 실행되지 않게 됩니다.



이는 간단하게 해결할 수 있습니다. 이전에 실행하던 ./gradlew test 명령에 --continue 옵션을 추가해주면 이전 Task의 실패 여부와 상관없이 모든 Task를 수행할 수 있습니다.



빌드가 실패하지만 **chat** 모듈에 **report가 생성**된 것을 볼 수 있습니다.

### 관련 포스팅

* [코드 분석 도구 적용기 - 1편, 코드 커버리지(Code Coverage)가 뭔가요?](https://seller-lee.github.io/java-code-coverage-tool-part1)
* [코드 분석 도구 적용기 - 3편, SonarQube 적용하기](https://seller-lee.github.io/static-code-analysis-part3)

### 참고 링크

* [Gradle 프로젝트에 JaCoCo 설정하기 - 우아한형제들 기술 블로그](https://woowabros.github.io/experience/2020/02/02/jacoco-config-on-gradle-project.html)
* [좌충우돌 jacoco 적용기 - 바닥부터 천천히](https://bottom-to-top.tistory.com/36)
* [The JaCoCo Plugin - Gradle](https://docs.gradle.org/current/userguide/jacoco_plugin.html#header)
* [Gradle 파일 다루기 - 권남](https://kwonnam.pe.kr/wiki/gradle/files)
* [Gradle Build Lifecycle - 권남](https://kwonnam.pe.kr/wiki/gradle/buildlifecycle)

# 코드 분석 도구 적용기 - 3편, SonarQube 적용하기

## SonarQube란?



### 정적 코드 분석 도구

***정적 프로그램 분석(static program analysis)은 실제 실행 없이 컴퓨터 소프트웨어를 분석하는 것을 말한다. 대부분의 경우에 분석은***[**소스 코드**](https://ko.wikipedia.org/wiki/%EC%86%8C%EC%8A%A4_%EC%BD%94%EB%93%9C)***의 버전 중 하나의 형태로 수행되며, 가끔은***[**목적 파일**](https://ko.wikipedia.org/wiki/%EB%AA%A9%EC%A0%81_%ED%8C%8C%EC%9D%BC)***형태로 분석된다. 이에 반하여 실행 중인 프로그램을 분석하는 것을***[**동적 프로그램 분석**](https://ko.wikipedia.org/wiki/%EB%8F%99%EC%A0%81_%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%A8_%EB%B6%84%EC%84%9D)***이라고 한다. -***[**wikipedia**](https://ko.wikipedia.org/wiki/%EC%A0%95%EC%A0%81_%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%A8_%EB%B6%84%EC%84%9D)

간단하게 정적 분석은 프로그램을 **실행하지 않은 상태**에서 소스 코드나 컴파일된 코드를 이용해 프로그램을 분석하는 방법이며, 동적 분석은 프로그램을 실제 환경이나 가상 환경에서 **실행해 보면서** 분석하는 방법입니다.

정적 분석은 소스 코드의 모든 부분을 확인할 수 있지만, 실행 환경에서의 상태를 정확히 알 수 없기 때문에 실행할 때에만 알 수 있는 데이터가 필요한 경우 정확히 분석할 수 없습니다. 반대로 동적 분석은 실제로 실행해 보면서 분석하기 때문에 실행 환경에서의 상태를 잘 알 수 있지만, 프로그램을 실행할 수 있는 환경을 구축하기 어려울 때가 많고 소스 코드의 모든 부분을 테스트해 보기 힘들다는 문제가 있습니다.

이렇게 각 분석 방식은 장단점이 존재하기 때문에 정적 분석은 주로 개발 단계에서 소스 코드의 구조적인 문제나 실수를 찾아내는 데 사용하며 동적 분석은 테스트나 모니터링할 때 사용합니다.

|  | **정적 분석(Static analysis)** | **동적 분석(Dynamic analysis)** |
| --- | --- | --- |
| 분석 대상 | 소스 코드 또는 컴파일된 코드 | 프로그램 실행 환경 |
| 테스트 범위 | 소스 코드의 모든 부분 | 실행 가능한 경로 |
| 활용 | 코드 상의 문제나 실수를 찾음 | 테스트, 모니터 |

### SonarQube

***소나큐브(SonarQube, 이전 이름: 소나/Sonar)[***[**2]**](https://ko.wikipedia.org/wiki/%EC%86%8C%EB%82%98%ED%81%90%EB%B8%8C#cite_note-2)***는 20개 이상의***[**프로그래밍 언어**](https://ko.wikipedia.org/wiki/%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D_%EC%96%B8%EC%96%B4)***에서***[**버그**](https://ko.wikipedia.org/wiki/%EC%86%8C%ED%94%84%ED%8A%B8%EC%9B%A8%EC%96%B4_%EB%B2%84%EA%B7%B8)***,***[**코드 스멜**](https://ko.wikipedia.org/wiki/%EC%BD%94%EB%93%9C_%EC%8A%A4%EB%A9%9C)***, 보안 취약점을 발견할 목적으로 정적***[**코드 분석**](https://ko.wikipedia.org/wiki/%EC%A0%95%EC%A0%81_%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%A8_%EB%B6%84%EC%84%9D)***으로 자동 리뷰를 수행하기 위한 지속적인***[**코드 품질**](https://ko.wikipedia.org/wiki/%EC%86%8C%ED%94%84%ED%8A%B8%EC%9B%A8%EC%96%B4_%ED%92%88%EC%A7%88)***검사용***[**오픈 소스**](https://ko.wikipedia.org/wiki/%EC%98%A4%ED%94%88_%EC%86%8C%EC%8A%A4_%EC%86%8C%ED%94%84%ED%8A%B8%EC%9B%A8%EC%96%B4)***플랫폼이다.***[**소나소스**](https://ko.wikipedia.org/w/index.php?title=%EC%86%8C%EB%82%98%EC%86%8C%EC%8A%A4&action=edit&redlink=1)***(SonarSource)가 개발하였다. 소나큐브는***[**중복 코드**](https://ko.wikipedia.org/wiki/%EC%A4%91%EB%B3%B5_%EC%BD%94%EB%93%9C)***,***[**코딩 표준**](https://ko.wikipedia.org/wiki/%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D_%EC%BD%94%EB%93%9C_%EC%9E%91%EC%84%B1)***,***[**유닛 테스트**](https://ko.wikipedia.org/wiki/%EC%9C%A0%EB%8B%9B_%ED%85%8C%EC%8A%A4%ED%8A%B8)***,***[**코드 커버리지**](https://ko.wikipedia.org/wiki/%EC%BD%94%EB%93%9C_%EC%BB%A4%EB%B2%84%EB%A6%AC%EC%A7%80)***,***[**코드 복잡도**](https://ko.wikipedia.org/w/index.php?title=%EC%88%9C%ED%99%98_%EB%B3%B5%EC%9E%A1%EB%8F%84&action=edit&redlink=1)***,***[**주석**](https://ko.wikipedia.org/wiki/%EC%A3%BC%EC%84%9D_(%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D))***,***[**버그**](https://ko.wikipedia.org/wiki/%EB%B0%A9%EC%96%B4%EC%A0%81_%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D)***및 보안 취약점의 보고서를 제공한다.***

**SonarQube**는 위에서 소개한 **정적 코드 분석 도구** 중 하나입니다.

정적 코드 분석 도구에는 PMD, FindBugs, CheckStyle 등이 있습니다. 저희 팀에서 SonarQube를 선택하게 된 주된 이유는 **레퍼런스가 많고** Github이나 Jenkins와의 연동을 통해 자동 정적 코드 분석을 구성할 수 있기 때문입니다.

그렇다면 SonarQube의 장점에는 어떤 것이 있을까요?

* 지속적인 인스펙션
  + 지속적인 통합과 같이 빌드와 연동하여 지속적으로 코드에 대한 인스펙션을 수행합니다.
* 품질 중앙화
  + 개발된 조직의 코드의 품질을 중앙 저장소에서 가시화하고 단일 위치에서 관리합니다.
* DevOps와의 통합
  + 다양한 빌드 시스템, CI 엔진과 통합되어 DevOps 실천을 지원합니다.
* 품질 요구사항 설정
  + 품질 게이트를 통해 표준화된 코드 품질 요구사항을 설정합니다.
* 다중 언어 분석
  + 20개가 넘는 프로그램 언어에 대한 코드 분석을 지원합니다.
* 플러그인을 통한 확장
  + 다수의 플러그인을 통해 SonarQube의 기능을 확장할 수 있습니다.
* 오픈소스 프로젝트
  + 오픈소스 프로젝트로 특정 범위까지 무료로 사용 가능하다.

지금부터는 이런 장점이 있는 SonarQube를 프로젝트에 적용하는 방법에 대해 소개하도록 하겠습니다!

## 프로젝트에 SonarQube 적용하기

### 개발 환경

#### 프로젝트

* Java 8
* Spring Boot 2.3.1
* Gradle 6.4.1

#### 인프라

* Ubuntu 18.04
* Jenkins 2.249.3
* SonarQube Server 8.5.1
* SonarQube Scanner 4.5.0

현재 본 프로젝트의 개발 환경은 위와 같습니다.

### 설치 환경 설정하기

SonarQube를 설치하는 방법은 **시스템에 직접 설치하는 방법**과 **Docker를 사용하여 설치하는 방법**이 있습니다. 저희는 프로젝트 전반에 걸처 Docker를 사용했기 때문에 SonarQube도 Docker를 통해 설치를 진행하겠습니다.

***시스템 환경은 AWS EC2 인스턴스를 기반으로 설명하도록 하겠습니다. OS로는 Ubuntu 18.04를 사용하였습니다.***

***SonarQube의 설치 방법에 대해 좀 더 자세히 알아보고 싶은 분들은 SonarQube 공식 문서의***[**Try Out SonarQube**](https://docs.sonarqube.org/latest/setup/get-started-2-minutes/)***를 참고해주시기 바랍니다.***

#### Docker 설치하기

[Docker Document](https://docs.docker.com/engine/install/ubuntu/)를 참고하여 설치를 진행했습니다. 공식 문서에는 환경 설정을 직접 하는 방법과 스크립트로 하는 방법이 있는데, SonarQube의 경우 실제 Product가 올라가는 상황은 아니기 때문에 **스크립트를 통해 진행**하였습니다.

[**Docker Document**](https://docs.docker.com/engine/install/ubuntu/#install-using-the-convenience-script)***를 보면 Using these scripts is not recommended for production environments 경고문이 있습니다. 실제 Product 환경에 Docker를 설치한다면 직접 환경 설정을 하면서 설치하기를 추천드립니다.***

먼저 EC2 인스턴스의 패키지를 업데이트하고 스크립트를 다운받기 위해 curl을 설치하겠습니다.

Copy

1sudo apt-get update

2sudo apt-get curl

그 후 [스크립트](https://get.docker.com/)를 EC2 인스턴스에 설치하고 실행하면 됩니다.

Copy

1curl -fsSL https://get.docker.com -o get-docker.sh

2sudo sh get-docker.sh

이 상태로 마치면 Docker를 사용할 때마다 매번 sudo를 입력해야 하는 불편함이 있습니다. 로그인 계정을 docker 그룹에 추가하면 sudo를 입력하지 않고도 사용할 수 있습니다.

Copy

1sudo usermod -aG docker $USER

#### SonarQube 이미지 설치하기

Docker 설치가 완료됐다면 이제 SonarQube 이미지를 가져오면 됩니다. [Docker Hub](https://hub.docker.com/_/sonarqube)에서 SonarQube의 이미지를 가져와서 설치합니다.

Copy

1docker pull sonarqube

명령어 한 줄로 sonarqube의 설치는 끝이 납니다!

이미지가 잘 설치되었는지는 아래의 명령어를 통해 확인할 수 있습니다.

Copy

1docker images



SonarQube가 정상적으로 설치됨을 확인할 수 있습니다.

### SonarQube 실행하기

이제 Docker로 설치한 SonarQube를 실행해보겠습니다.

Copy

1docker run -d --name sonarqube -p 8080:9000 sonarqube

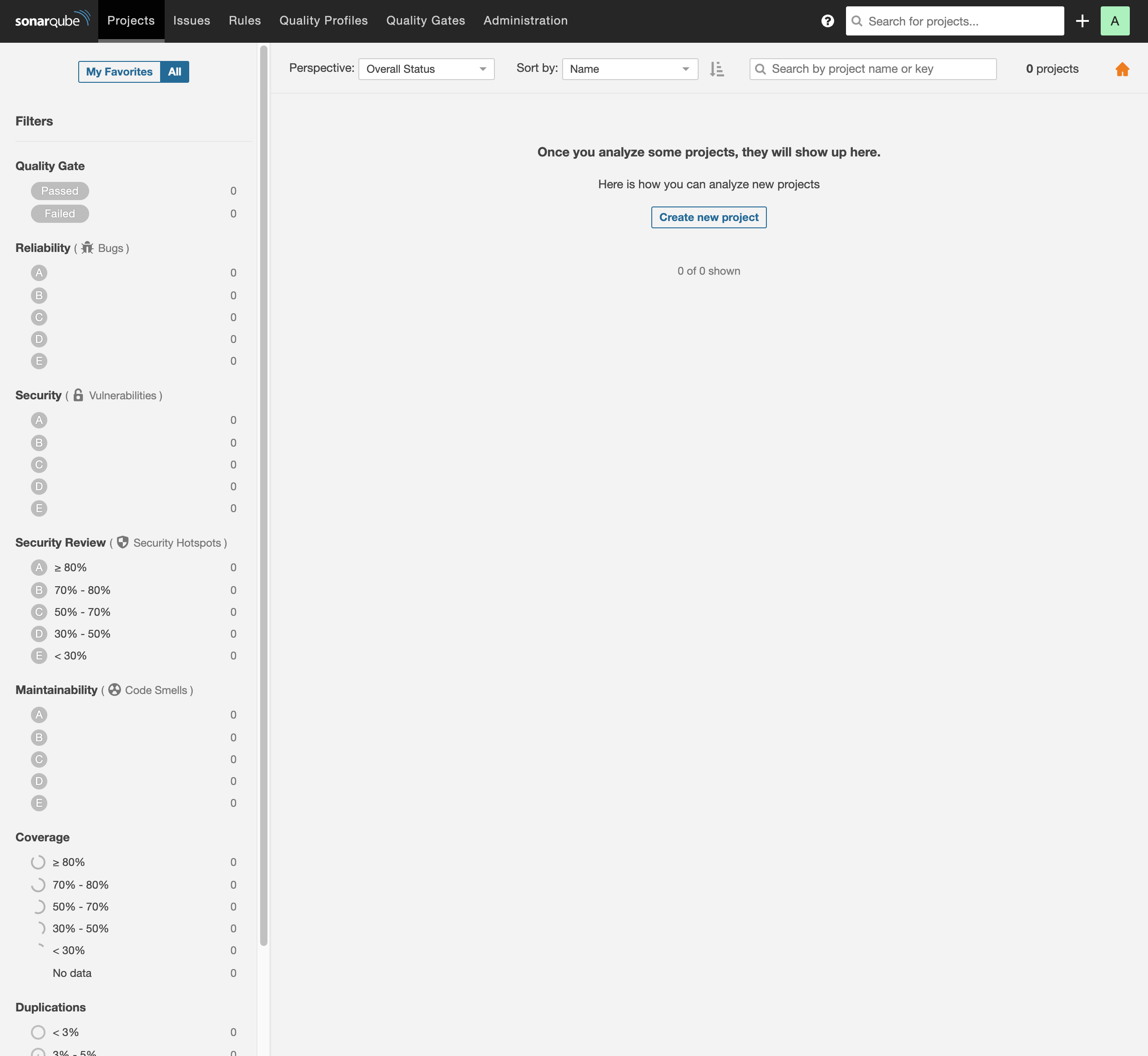
* -d : 컨테이너를 일반 프로세스가 아닌 데몬 프로세스 형태로 실행하여 프로세스가 끝나도 유지되도록 한다.
* --name : 실행할 컨테이너의 이름을 설정한다.
* -p : 컨테이너의 포트를 호스트의 포트와 바인딩해 연결할 수 있다. -p [host(외부)의 port]:[container(내부)의 port]이다.

SonarQube의 경우 **기본이 9000 포트**입니다. 현재 사용 중인 EC2가 9000 포트는 열려있지 않고 8000 포트만 열려있어서 8000 포트로 바인딩하였습니다.

***SonarQube의 기본 포트를 변경하고 싶다면 SonarQube 컨테이너에 들어간 후 /config/sonar.properties 파일의 sonar.web.port 프로퍼티를 수정하면 됩니다.***

이렇게 실행하고 http://[EC2 인스턴스 ip 주소]:[port 번호]로 접속을 하면 실행된 SonarQube를 확인할 수 있습니다. 저희 프로젝트의 경우 8000 포트를 사용했기 때문에 http://x.x.x.x:8000로 접속하였습니다.

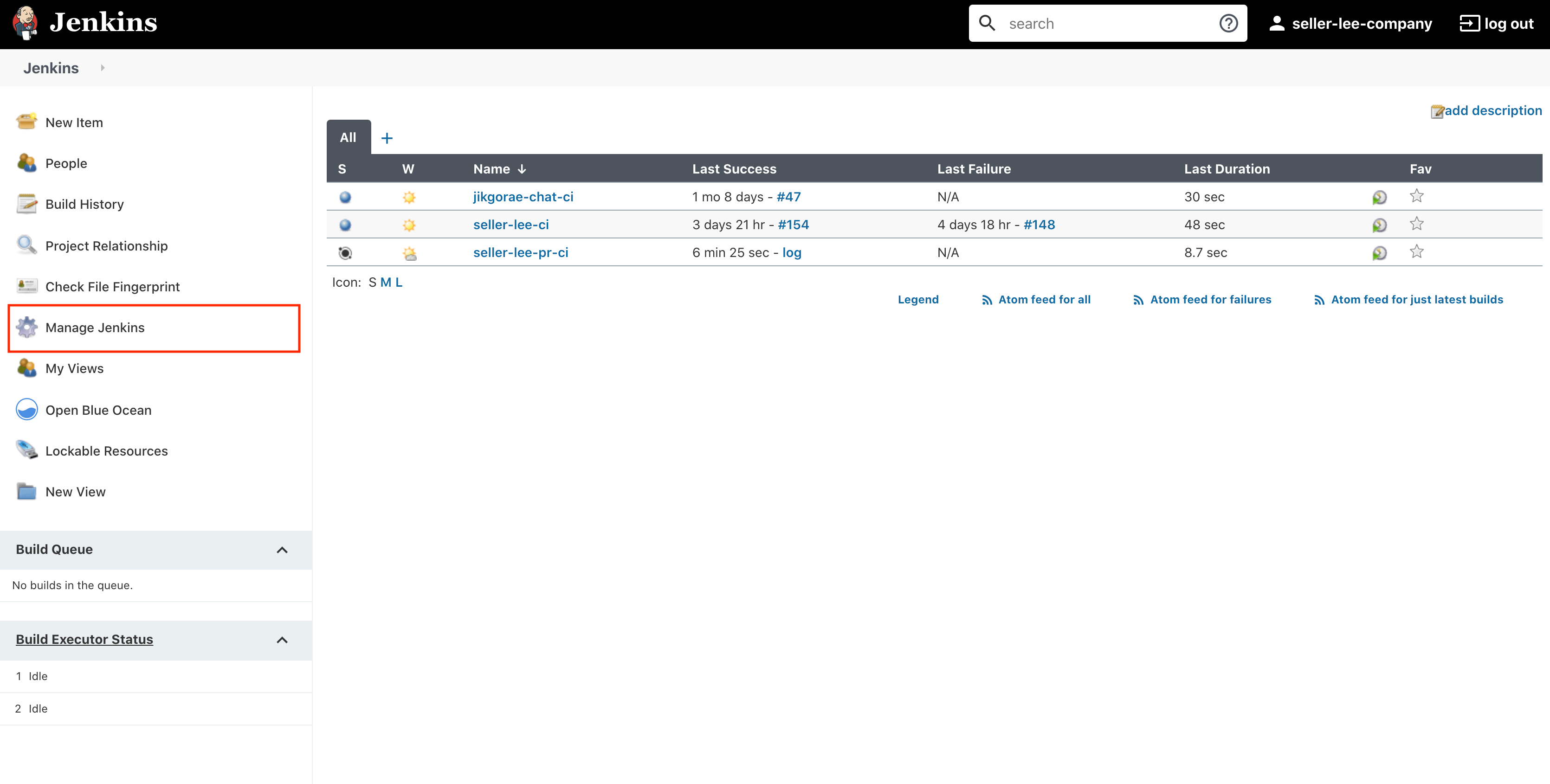
처음 접속하면 로그인을 해야하는데 기본 ID와 비밀번호는 모두 admin입니다. 이를 통해 접속하면 아래와 같은 페이지를 볼 수 있습니다.

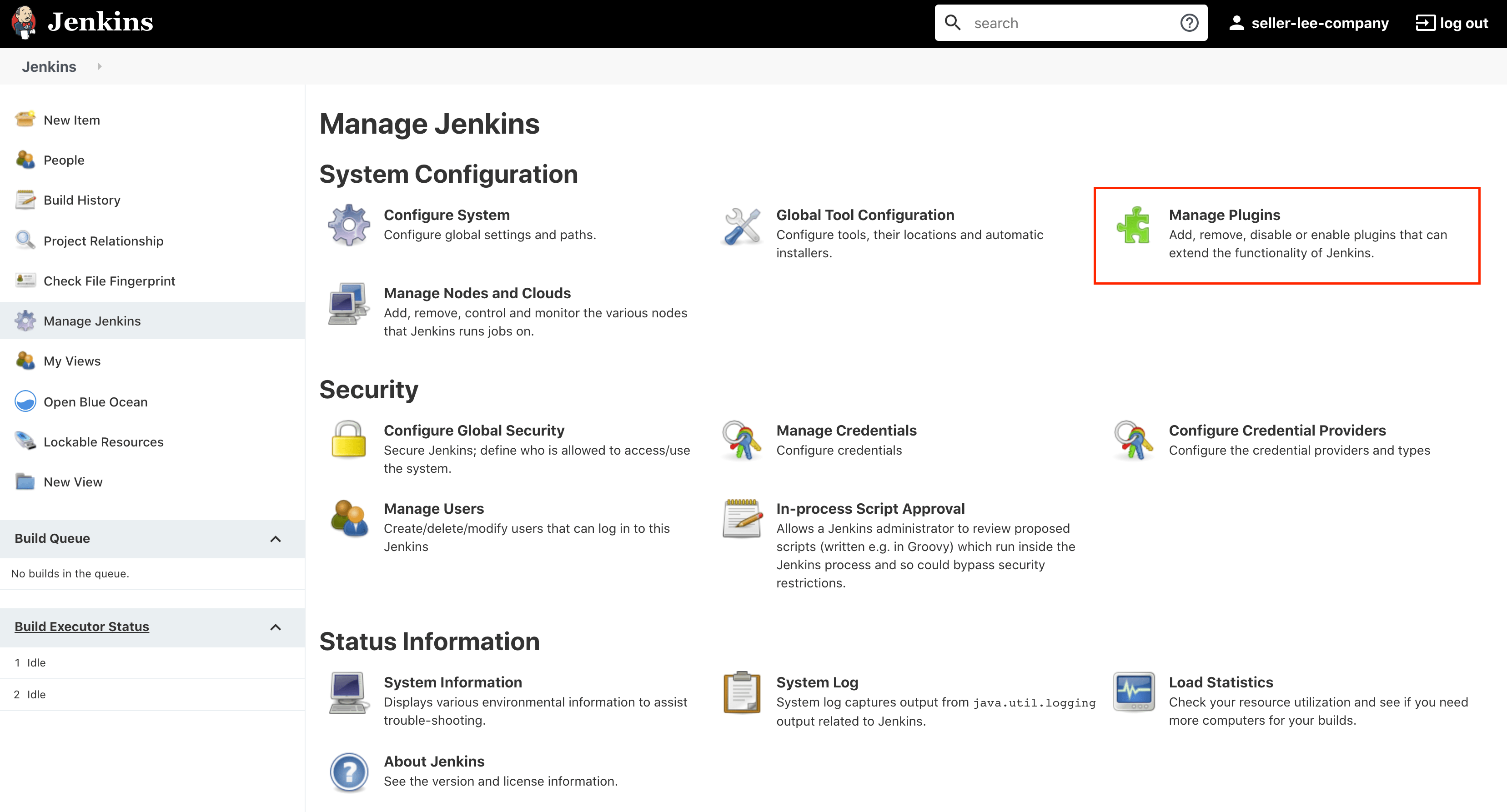


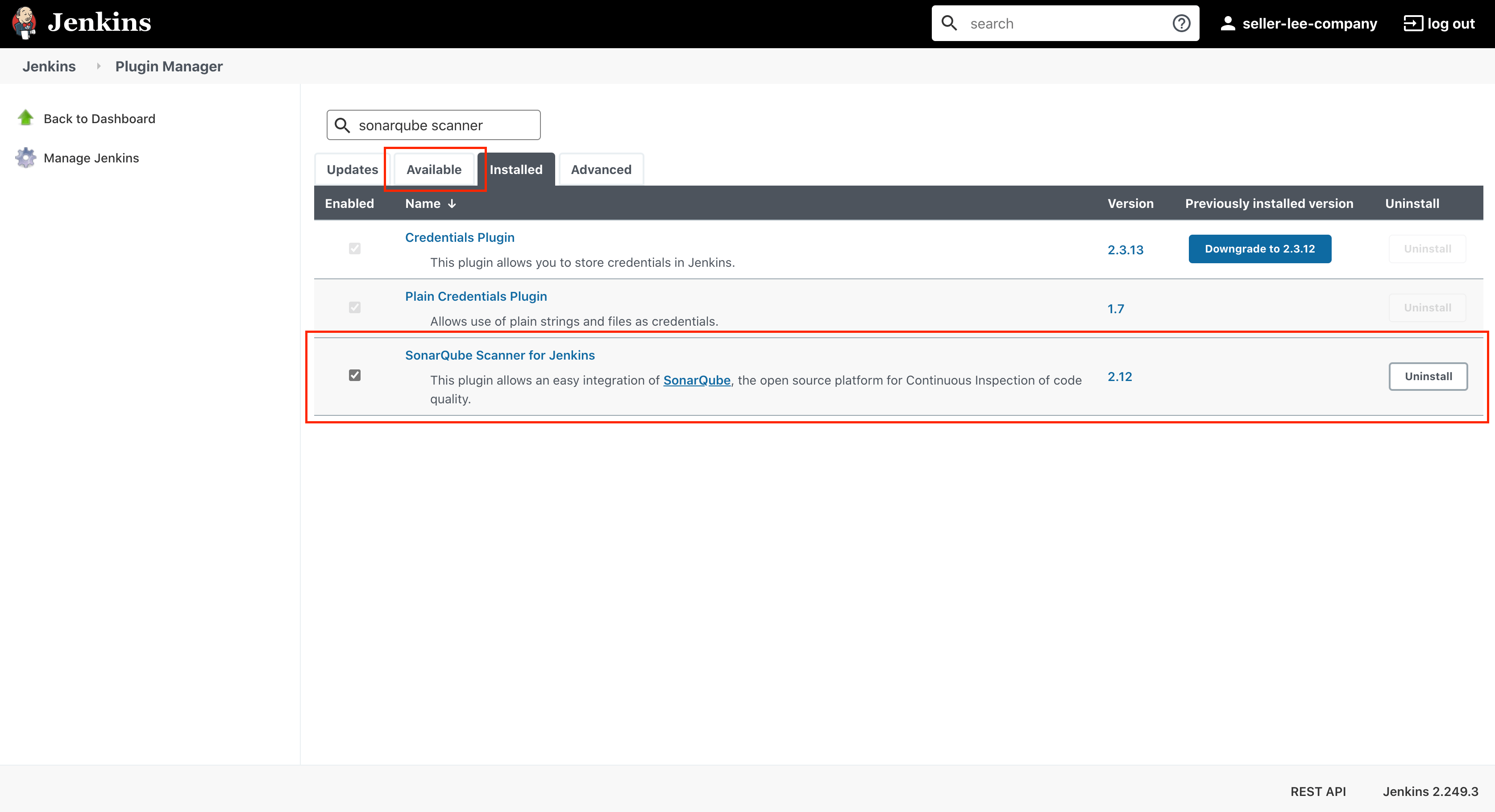
### Jenkins 설정하기

#### SonarQube Scanner 플러그인 설치하기

프로젝트에 적용한 Jenkins에 **SonarQube Scanner 플러그인**을 설치합니다.





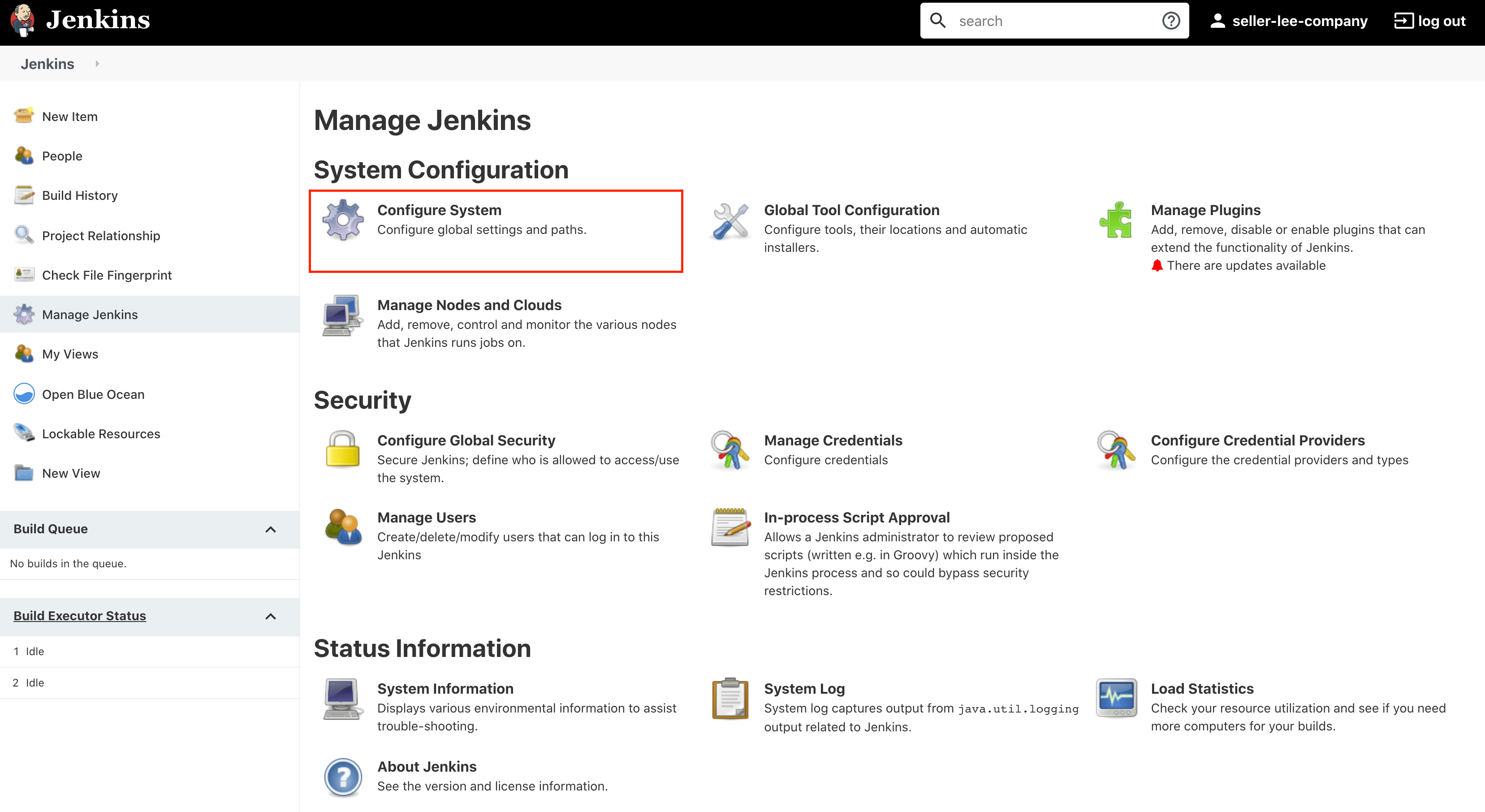


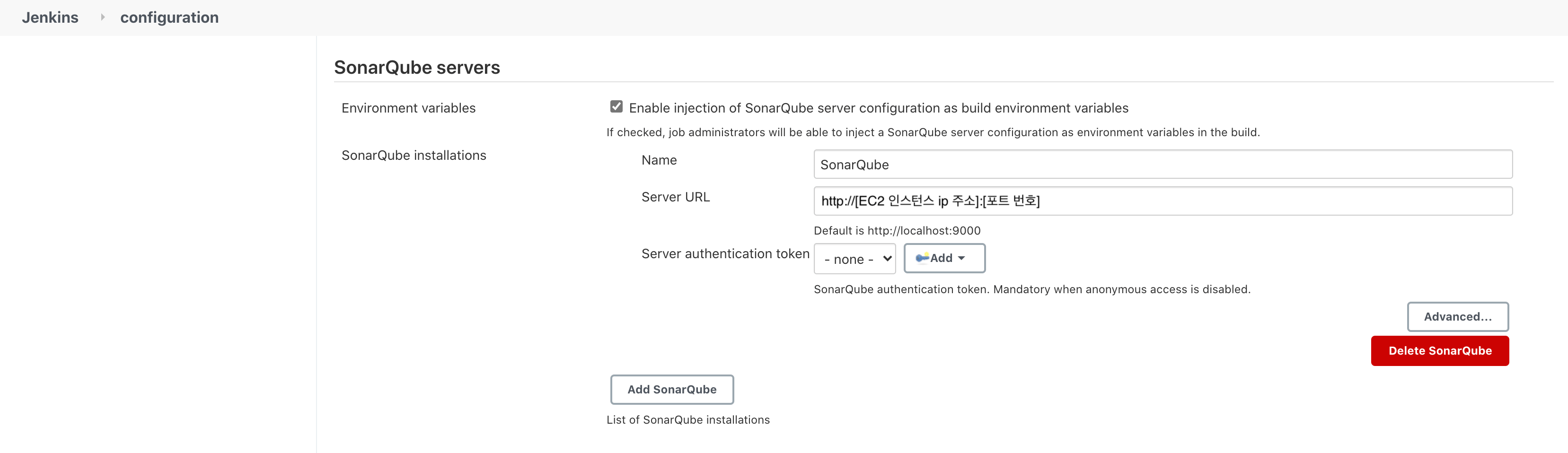
Jenkins의 Manage Jenkins > Manage Plugins > Available에서 SonarQube Scanner for Jenkins를 설치합니다.

***위 사진은 Jenkins에 이미 SonarQube Scanner를 설치했기 때문에 Installed에서 조회가 된 모습입니다.***

#### SonarQube Server 설정하기

플러그인 설치가 완료되면 Jenkins에 SonarQube Server에 대한 설정을 해줘야 합니다. 이번에는 Manage Jenkins > Configure System > SonarQube Server에서 설정을 진행하겠습니다.



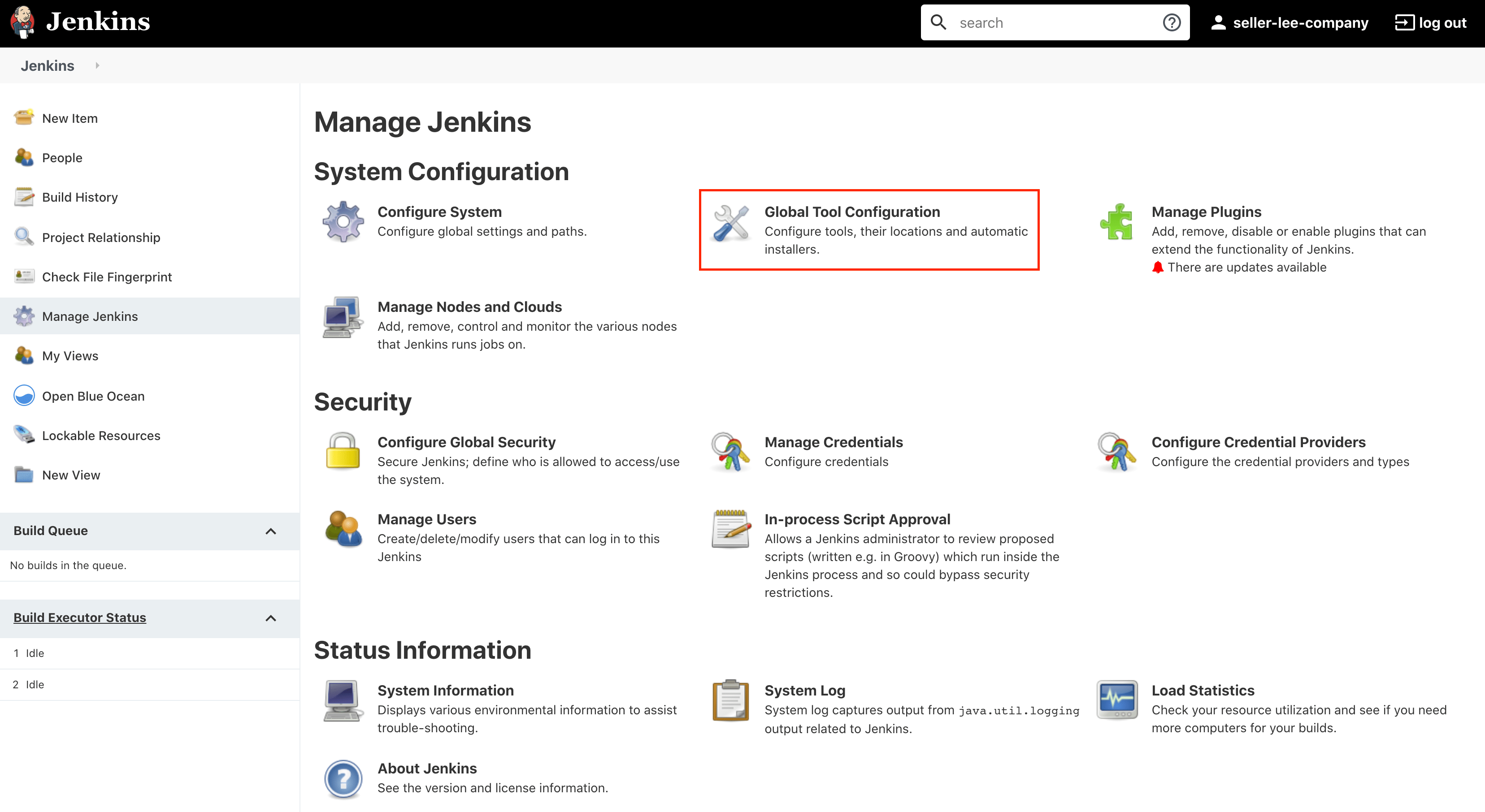


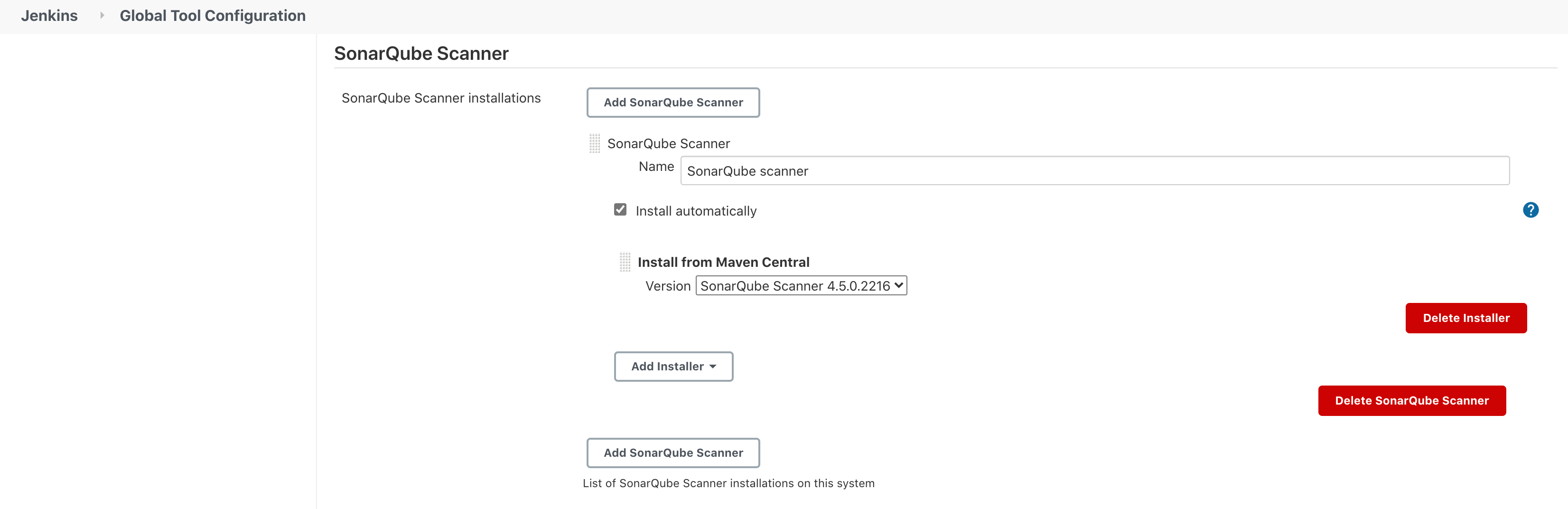
먼저 Enable injection of SonarQube server configuration as build environment variables를 체크합니다.

그리고 Name에는 설정하고 싶은 이름을 입력하고 Server URL은 설치한 SonarQube Server의 주소를 입력합니다. Server authentication token의 경우는 따로 설정하지 않아도 됩니다.

#### SonarQube Scanner 설정하기

Jenkins 설정의 마지막 단계로 SonarQube Scanner를 설정합니다. Manage Jenkins > Global Tool Configuration > SonarQube servers에서 설정을 진행하겠습니다.





이때 Name의 경우 추후 Jenkinsfile을 설정할 때 사용해야 하기 때문에 잘 기억해두어야 합니다.

SonarQube Scanner Installer로 여러 종류가 있지만 그 중에서 가장 간단한 Maven Central을 사용하였습니다.

### Gradle 설정하기

#### SonarQube 플러그인 의존성 추가

이제 SonarQube로 소스 코드를 분석하는 설정을 하겠습니다. SonarQube로 소스 코드를 분석하는 방법은 여러가지가 있는데 그 중에서 프로젝트의 Gradle Task를 통해 분석하는 방법을 사용하겠습니다.

[**SonarQube Document - Analyzing Source Code**](https://docs.sonarqube.org/latest/analysis/overview/)***에 여러가지 방법이 소개되어 있습니다.***

먼저 프로젝트의 기본 build.gradle에 [SonarQube 플러그인 의존성](https://plugins.gradle.org/plugin/org.sonarqube)을 추가해줘야 합니다.

Copy

1plugins {

2 id 'org.sonarqube' version '3.0'

3}

#### SonarQube Property 설정

현재 프로젝트 구조가 멀티 모듈이기 때문에 build.gradle의 subprojects에도 설정을 추가해줘야 합니다.

***프로젝트의 멀티 모듈 구조에 대한 내용은***[**이전 글**](https://seller-lee.github.io/java-code-coverage-tool-part2)***을 참고해주시기 바랍니다.***

Copy

1subprojects {

2 apply plugin: 'org.sonarqube'

3

4 sonarqube {

5 properties {

6 property 'sonar.host.url', 'http://[EC2 인스턴스 ip 주소]:[포트 번호]'

7 property 'sonar.login', 'SonarQube 로그인 토큰'

8 property 'sonar.sources', 'src'

9 property 'sonar.language', 'java'

10 property 'sonar.projectVersion', '1.1.0-SNAPSHOT'

11 property 'sonar.sourceEncoding', 'UTF-8'

12 property 'sonar.coverage.jacoco.xmlReportPaths', '${buildDir}/reports/jacoco/test/jacocoTestReport.xml'

13 property 'sonar.java.binaries', '${buildDir}/classes'

14 property 'sonar.test.inclusions', '\*\*/\*Test.java'

15 property 'sonar.exclusions', '\*\*/test/\*\*, \*\*/Q\*.java, \*\*/\*Doc\*.java, \*\*/resources/\*\*'

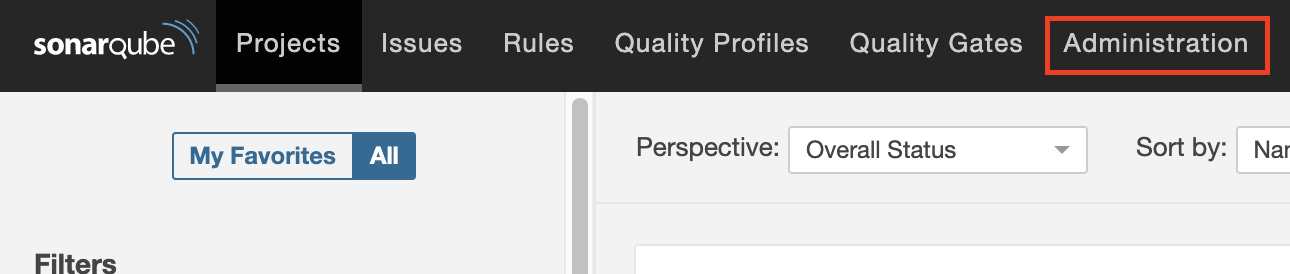
16 }

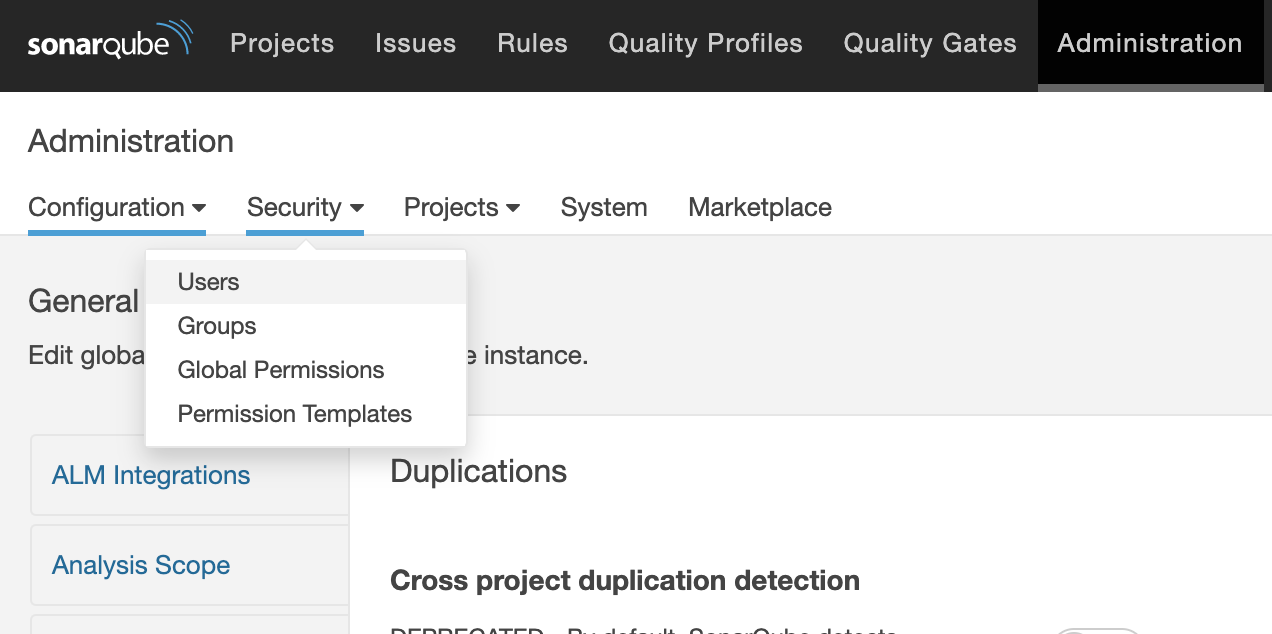
17 }

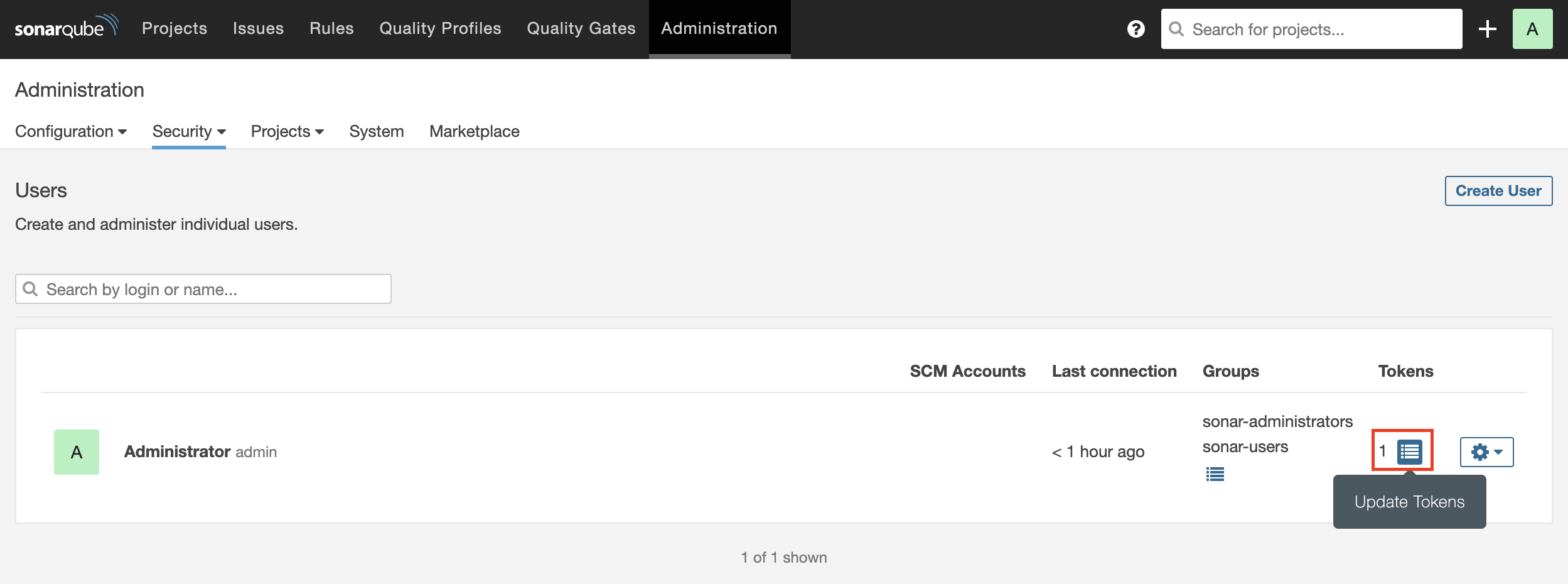
18}

sonarqube.property에 대해 간단히 알아보면

* sonar.host.url : 설치한 SonarQube의 주소이다. 앞서 젠킨스의 SonarQube Server에 입력한 주소와 동일하다.
* sonar.login : SonarQube Server에 로그인하여 발급받은 토큰이다.







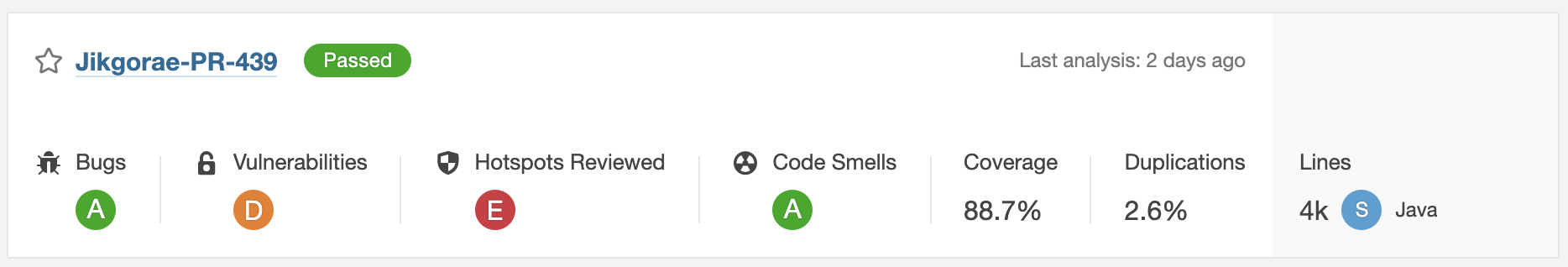
SonarQube Server 페이지에서 Administration > Security > Users > Tokens에서 토큰을 생성할 수 있습니다. 토큰의 경우 한 번 발급받으면 이후에 다시 토큰을 확인할 수 없기 때문에 잘 저장해둬야 합니다.

여기서 발급받은 토큰을 해당 프로퍼티의 값으로 설정하면 됩니다.

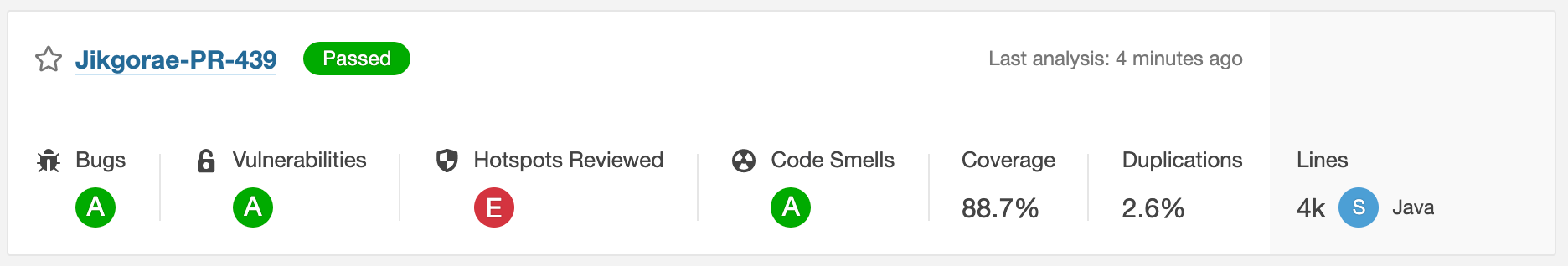
* sonar.sources : 분석할 소스 파일의 공통 경로를 지정한다.
* sonar.language : 작성한 코드의 언어를 선택한다. java의 경우 커뮤니티 버전에서 무료로 정적 분석을 할 수 있다.
* sonar.projectVersion : SonarQube에서 분석한 소스의 버전을 지정한다.
* sonar.coverage.jacoco.xmlReportPaths : JaCoCo가 분석한 결과에 해당하는 xml 파일을 사용하여 커버리지를 분석한다. JaCoCo가 생성하는 xml report의 기본 경로가 ${buildDir}/reports/jacoco/test/jacocoTestReport.xml이다.
* sonar.java.binaries : Java가 컴파일되면서 생성된 바이너리 파일을 분석한다.

해당 프로퍼티를 설정하면 보다 정확한 코드 분석이 가능합니다. 이번 프로젝트에서도 해당 프로퍼티의 설정만으로 코드 분석의 정확도가 증가함을 확인할 수 있었습니다.

* + 프로퍼티 적용 전



* + 프로퍼티 적용 후(코드 변경 X)



* sonar.test.inclusion : 분석에 사용할 테스트 코드의 위치를 지정한다. \*\*/\*Test.java 의 경우 src 하위의 모든 디렉토리에서 Test.java로 끝나는 파일을 의미한다.
* sonar.exclusions : 커버리지 분석에서 제외할 파일의 위치를 지정한다.

#### 서브 모듈 Property 설정하기

멀티 모듈 구조에서 모듈별로 프로퍼티를 적용하고 싶다면 해당 모듈 하위의 build.gradle 파일에 설정을 추가하면 됩니다.

Copy

1sonarqube {

2 properties {

3 property 'sonar.exclusions', '\*\*/test/\*\*'

4 }

5}

추가적으로 프로퍼티에 폴더 경로나 파일 경로를 패턴을 통해 표현할 수 있습니다.

* \* : 0개 또는 그 이상의 문자를 의미한다.
* \*\* : 0개 또는 그 이상의 디렉토리를 의미한다.
* ? : 하나의 문자에 해당한다.

### Jenkins Stage에 SonarQube 분석 추가하기

이제 위에서 설정한 SonarQube Gradle Task를 실행하는 Jenkins Stage를 추가합니다. 이번에는 SonarQube 코드 분석을 빌드 시점에서 실행하지 않고 Jenkins가 분석하는 시점에서 실행하도록 설정하였습니다.

***빌드 시점에서 실행하려면 JaCoCo Gradle Task 설정에서 사용한 finalizedBy와 같은 함수로 실행 순서를 규정하면 가능합니다.***

이는 프로젝트의 jenkinsfile을 통해 설정 가능합니다.

Copy

1node {

2 stage ('clone') {

3 checkout scm

4 }

5 stage('build') {

6 sh 'cd back && ./gradlew api:clean api:build && ./gradlew chat:clean chat:build'

7 }

8 stage('SonarQube analysis') {

9 withSonarQubeEnv('SonarQube') {

10 sh 'cd back && ./gradlew --info sonarqube' +

11 ' -Dsonar.projectKey=Jikgorae-' + env.BRANCH\_NAME +

12 ' -Dsonar.projectName=Jikgorae-' + env.BRANCH\_NAME

13 }

14 }

15}

clone stage와 build stage는 기존에 존재하던 설정이고 SonarQube analysis stage 설정을 이번에 추가하였습니다.

withSonarQubeEnv() 함수의 인자로 들어가는 'SonarQube'는 앞에서 설정한 Jenkins의 SonarQube Scanner 이름입니다. 해당 이름과 같은 SonarQube Scanner를 사용하여 아래의 스크립트를 실행합니다.

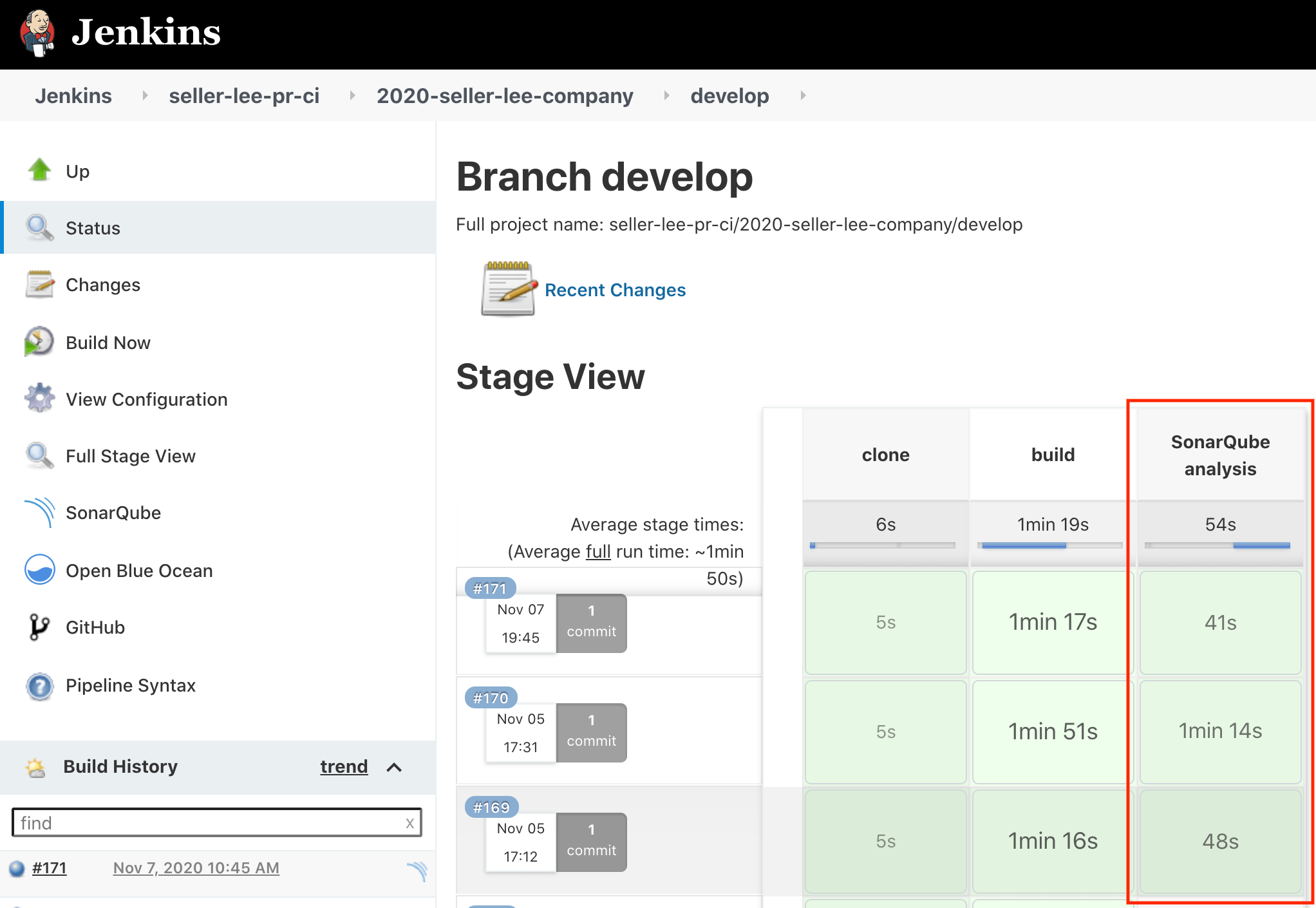
이때 -D 옵션을 사용하여 **스크립트 실행 시점에서 프로퍼티를 추가**할 수 있습니다.

위 설정의 경우 projectKey와 projectName을 현재 브랜치의 이름으로 설정하도록 작성하였습니다. 각 프로퍼티 사이에는 공백이 있어야지 정상적으로 실행됩니다.

위의 설정이 완료되면 Jenkins가 분석하면서 함께 SonarQube analysis Stage를 실행합니다.

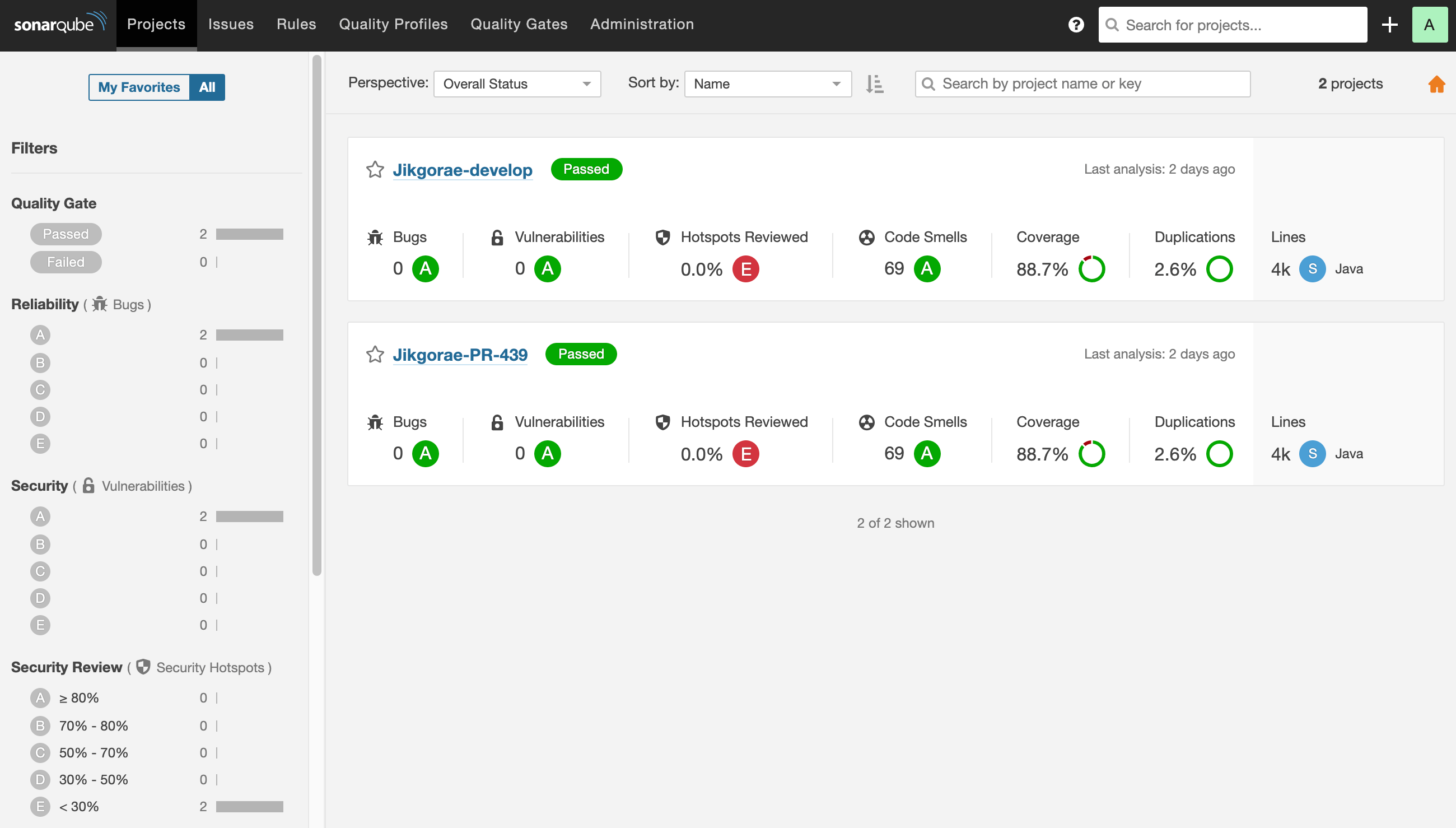
***저희 프로젝트의 경우 PR을 통해 기능을 구현하고, 이를 develop에 merge하는(그리고 develop을 master로 merge하는) 전형적인 git-flow 브랜치 전략을 사용하고 있습니다.***

***따라서 PR이 생성되는 시점(브랜치가 새로 생성 & 새로운 커밋, 푸쉬), develop 또는 master에 머지되는 시점에 SonarQube analysis Stage가 실행됩니다.***



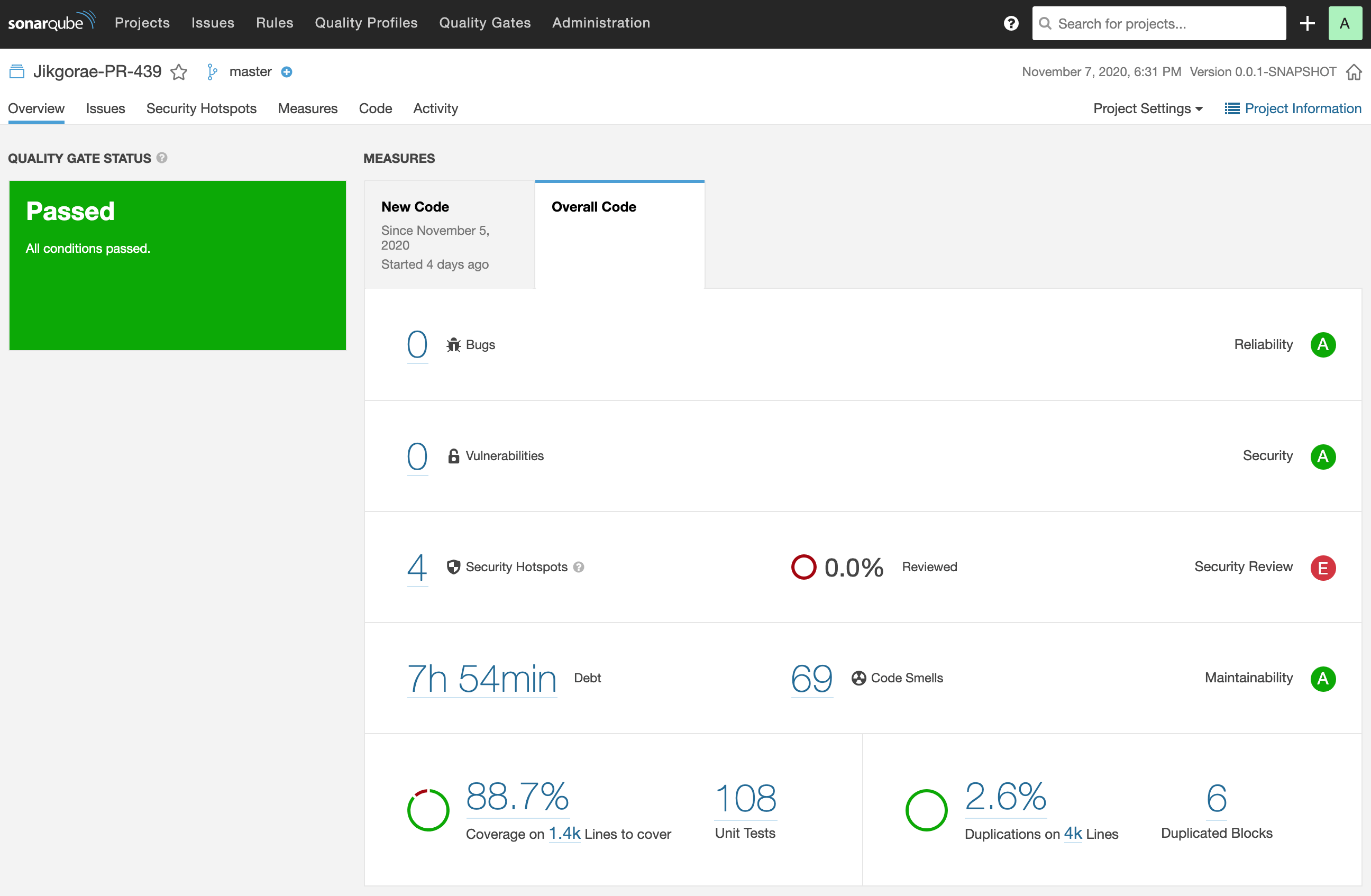
### SonarQube Server 프로젝트

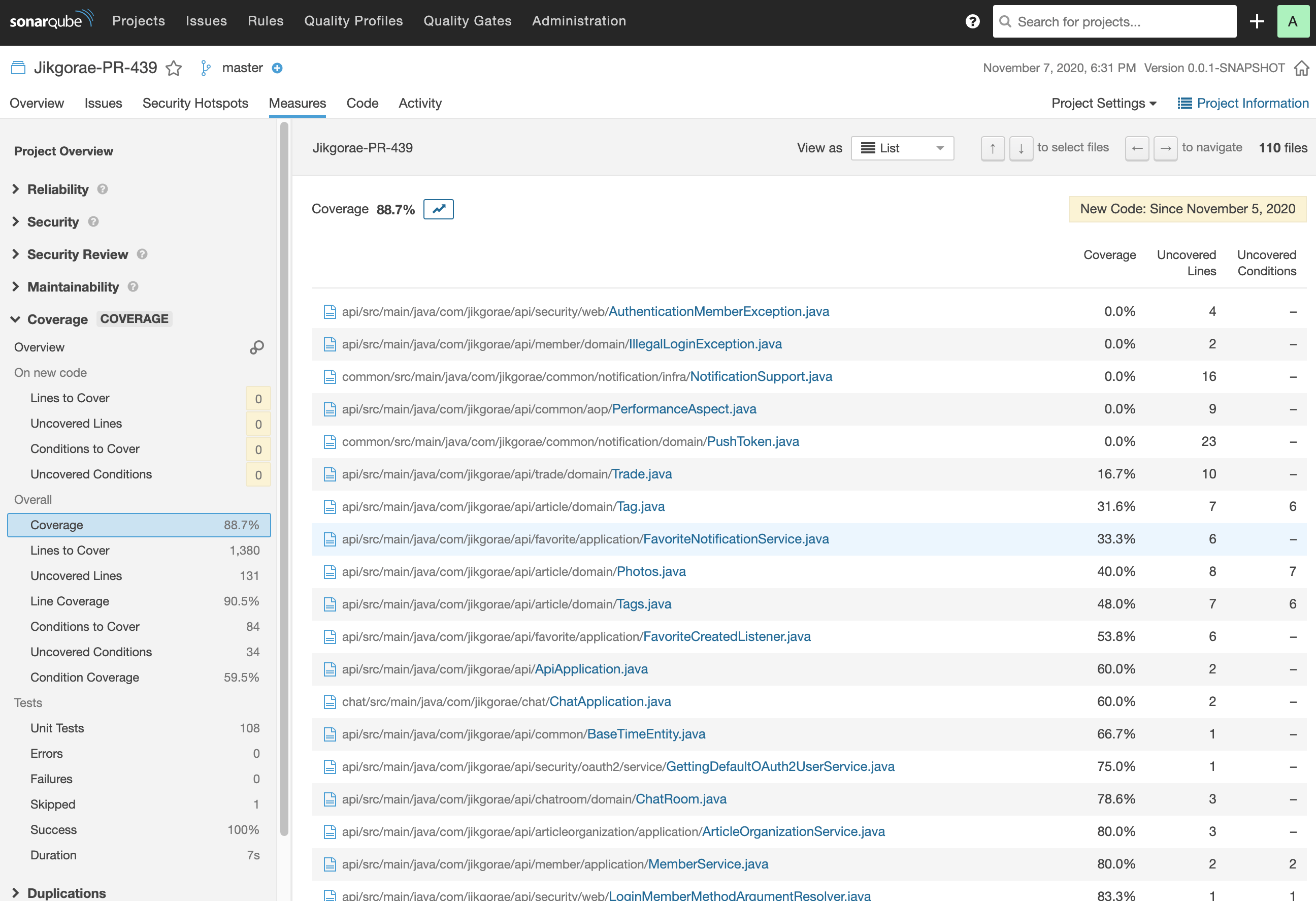
SonarQube 코드 분석이 정상적으로 마무리 됐다면 SonarQube 프로젝트가 생성될 것입니다.

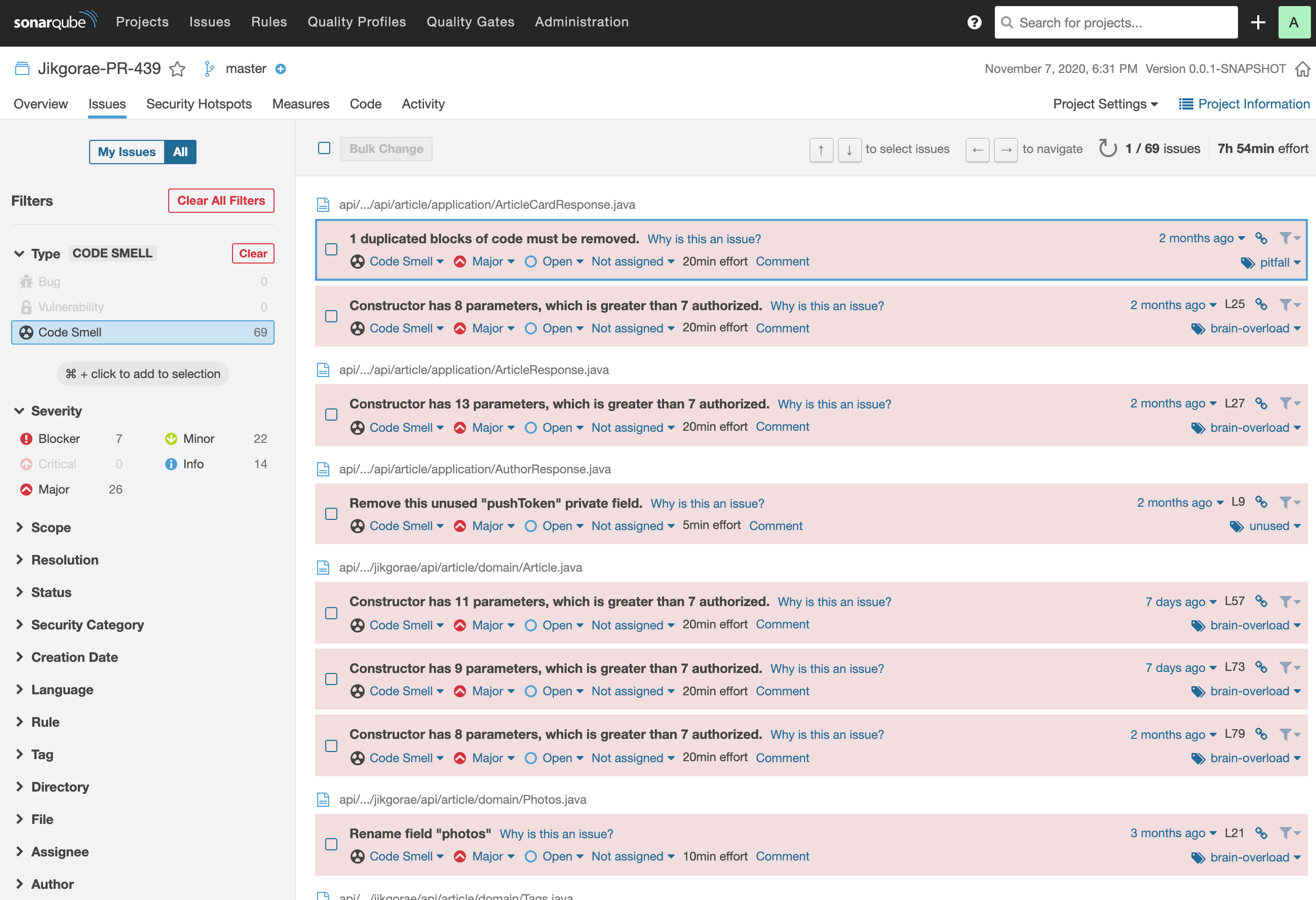


제가 한 설정은 브랜치(PR, develop) 별로 하나의 프로젝트를 생성합니다.

***프로젝트를 하나만 두고 한 프로젝트에서 여러 브랜치로 관리할 수도 있지만 이는 무료 버전인 Community 버전에서는 지원되지 않는 기능이어서 적용하지 못하였습니다.***







SonarQube 프로젝트를 클릭하면 위와 같이 상세한 정보들을 볼 수 있습니다. SonarQube에서 관리해주는 소프트웨어 품질에 대해 간단하게 소개해보면

* Code Smell : 심각한 이슈는 아니지만 베스트 프렉티스에서 사소한 이슈들로 모듈성(modularity), 이해가능성(understandability), 변경가능성(changeability), 테스트용의성(testability), 재사용성(reusability) 등이 포함된다.
* Bugs : 일반적으로 잠재적인 버그 혹은 실행시간에 예상되는 동작을 하지 않는 코드를 나타낸다.
* Vulnerabilities : 해커들에게 잠재적인 약점이 될 수 있는 보안상의 이슈를 말한다. SQL 인젝션, 크로스 사이트 스크립팅과 같은 보안 취약성을 찾아낸다.
* Duplications : 코드 중복은 코드의 품질을 저해시키는 가장 큰 요인 중 하나이다.
* Unit Test : 단위테스트 커버리지를 통해 단위 테스트의 수행 정도와 수행한 테스트의 성공/실패 정보를 제공한다.
* Complexity : 코드의 순환 복잡도, 인지 복잡도를 측정합니다.
* Size : 소스코드 사이즈와 관련된 다양한 지표를 제공합니다.

## 정리하며

저희 바람개비에서는 금융프로젝트에서 **정적 코드 분석 도구인 SonarQube를 어떻게 적용하면 좋은지**를 간단히 소개해보았습니다.

코드의 품질 개선은 프로젝트가 오랜 기간 유지보수를 가능하게 합니다. 코드 커버리지(JaCoCo)와 같은 동적 코드 분석, 코드 품질 분석(SonarQube)과 같은 정적 코드 분석은 프로젝트의 코드 품질 개선에 많은 도움을 줄 수 있습니다.

# 코드 분석 도구 적용기 - 4편, 개발 도구에서 SonarQube 활용하기

# IntelliJ

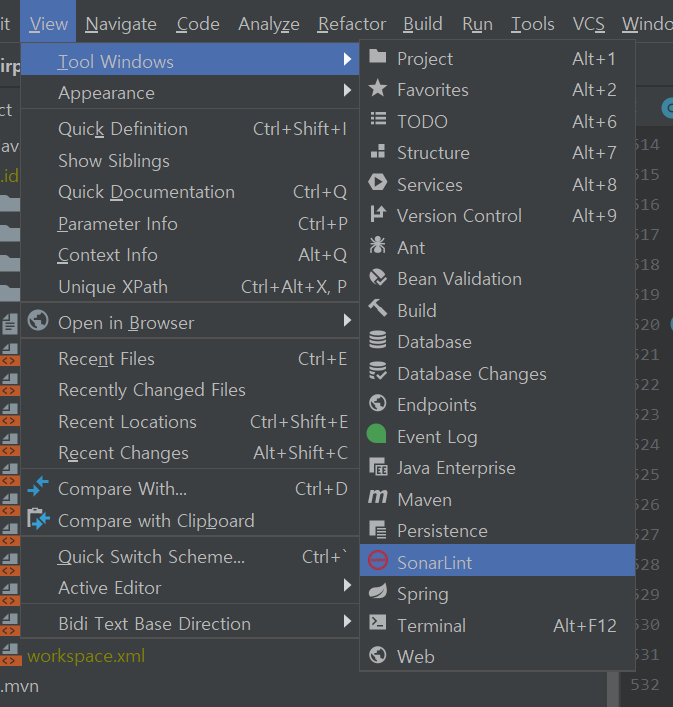
### **SonarLint**

SonarLint는 intelliJ에서 코드를 분석하게 도와주는 플러그인으로,

SonarQube와 연결하기 위해**필수적인 단계가 아니므로,** 넘어가도 무방합니다.

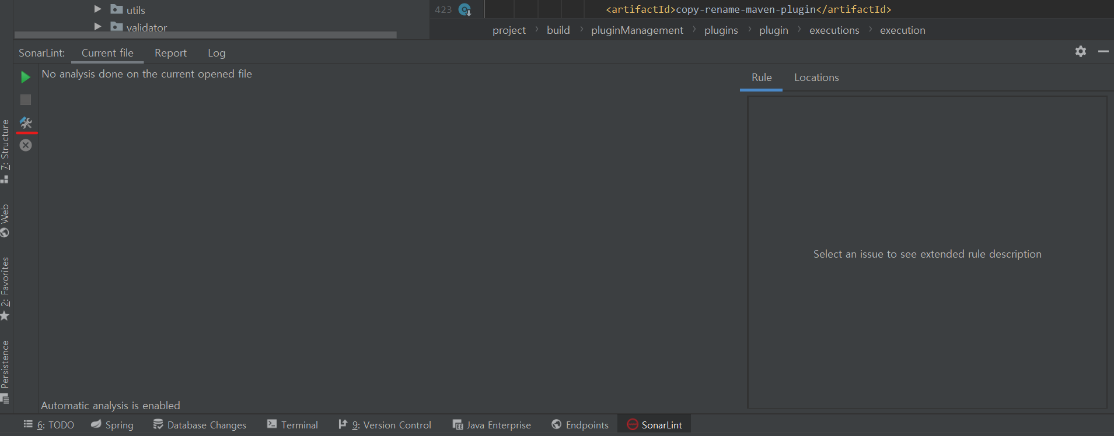


intelliJ에서 File - Setting - Plugins에서 **SonarLint**를 설치해줍니다.

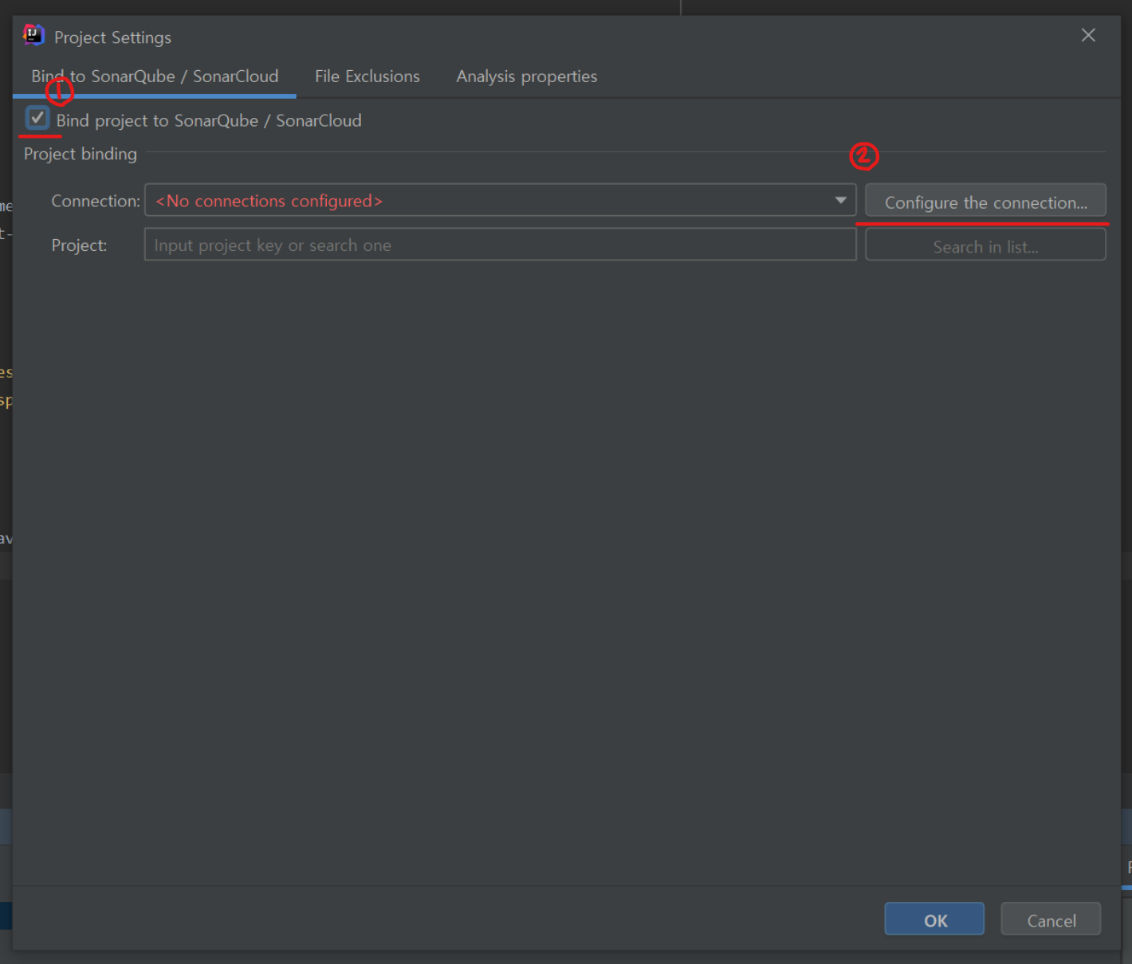


적용을 위해 intelliJ를 재시작하고

**View - Tool Windows - SonarLint**를 누르면 아래쪽에 사이드바같은게 뜨는것을 확인할 수 있습니다.

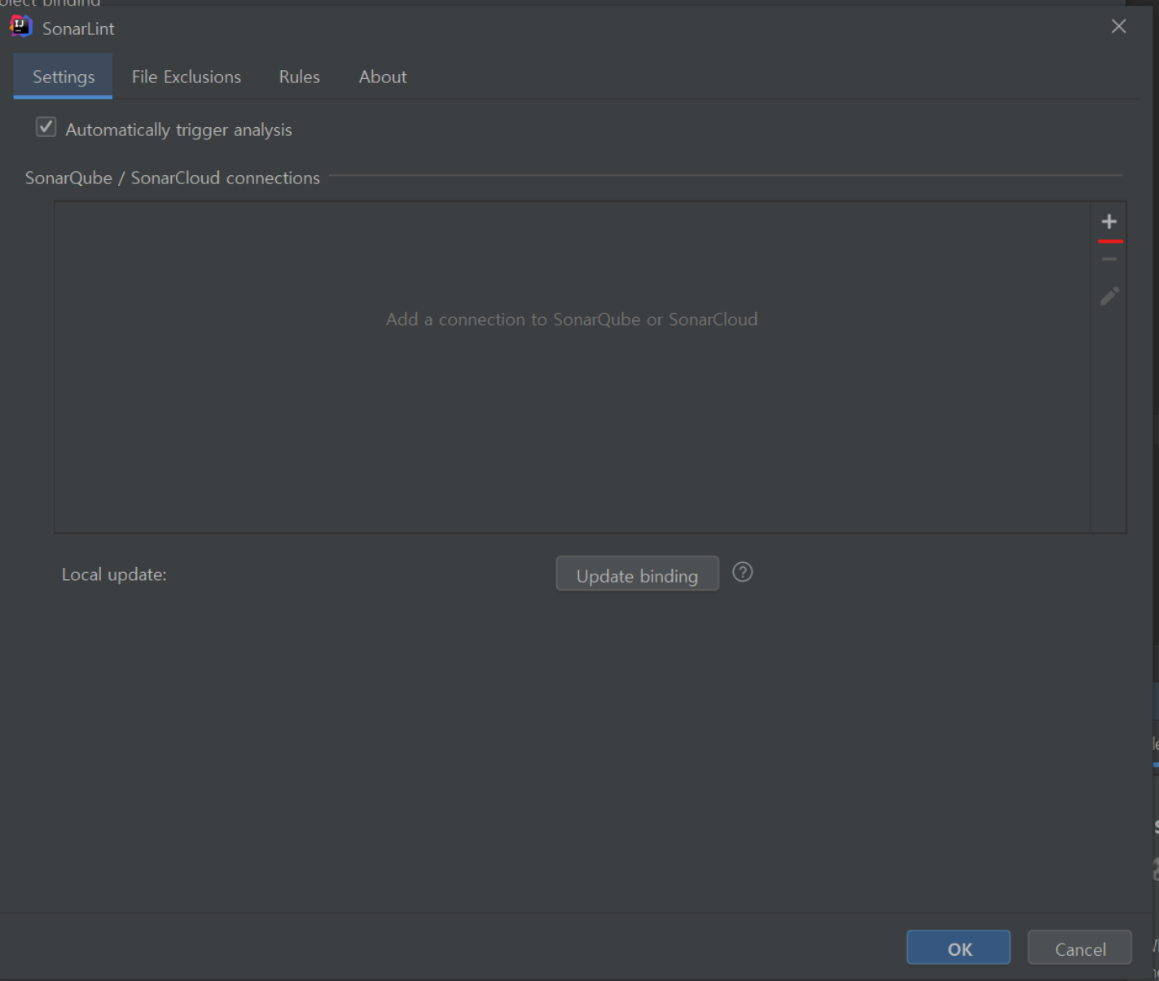


여기서 가장 왼쪽의 세번째 아이콘을 클릭해줍니다.

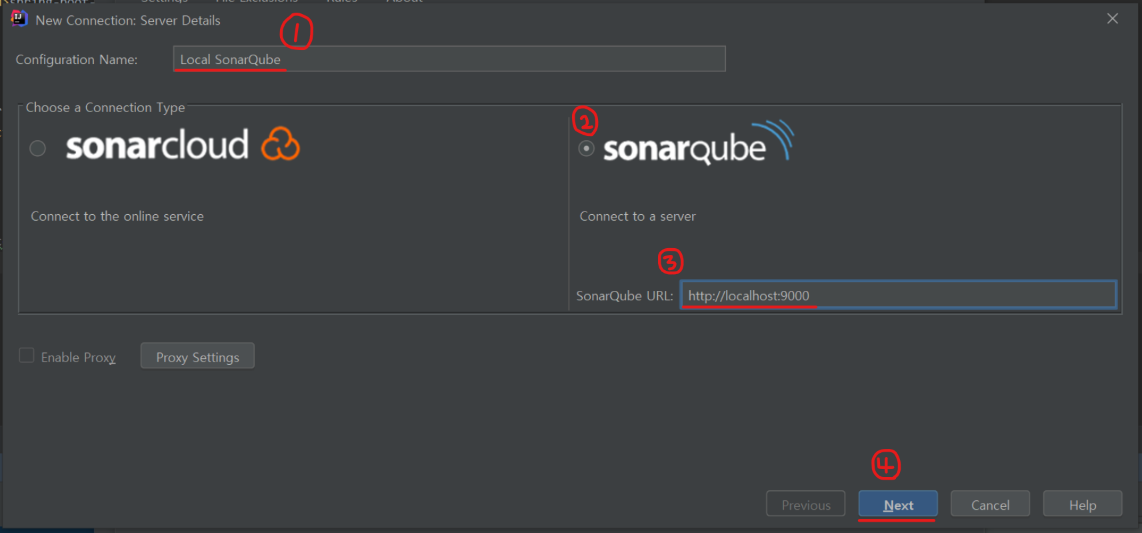


**Bind project to SonarQube / SonarCloud**의 체크박스를 누르고,

활성화되는 **Configure the connection...** 버튼을 눌러줍니다.



플러스(+)버튼을 누릅니다.



**이름**을 입력한 후에(1),

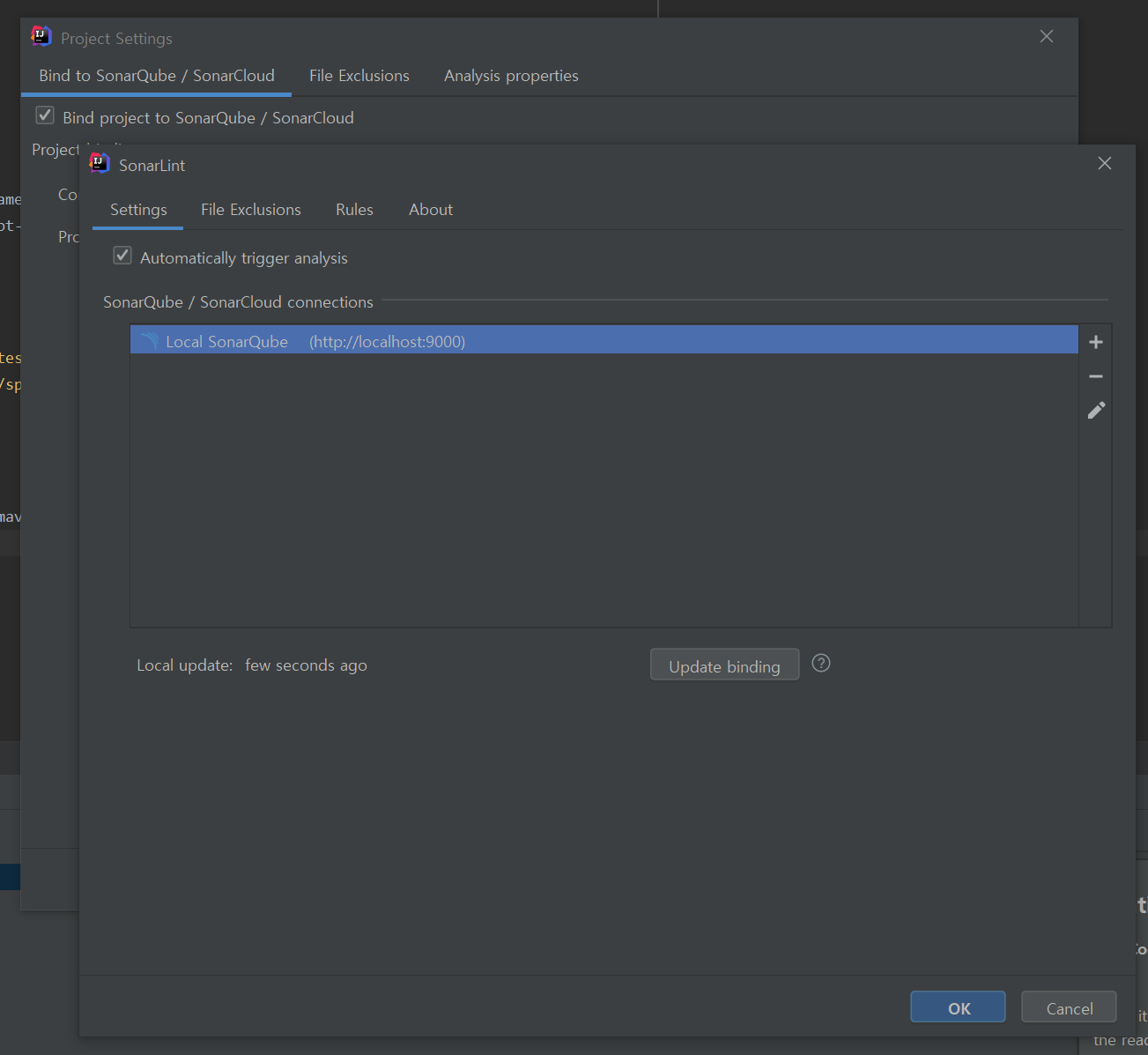
**SonarQube**를 선택해주고(2),

SonarQube **URL**을 [http://localhost:9000으로](http://localhost:9000%EC%9C%BC%EB%A1%9C/) 입력해 준 뒤에(3),

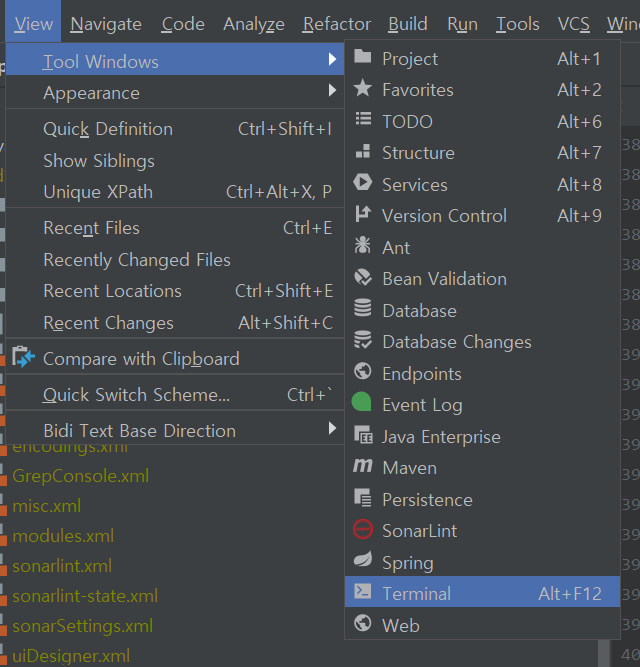
**다음**을 눌러줍니다.(4)



아까 복사해둔 토큰을 입력하고 **Next**를 눌러줍니다.



### **Terminal**



터미널을 열어준 후에 아래 명령어를 타이핑합니다.

mvn clean verify sonar:sonar

만약 아래같은 오류가 뜬다면 소나큐브와 intelliJ를 재부팅했다가 다시 실행하면 됩니다.

(소나큐브는 cmd창에서 **StopNTService.bat** - **StartSonar.bat**를 순서대로 입력하면 됩니다.)

ERROR: Error during SonarQube Scanner execution

ERROR: The 'report' parameter is missing

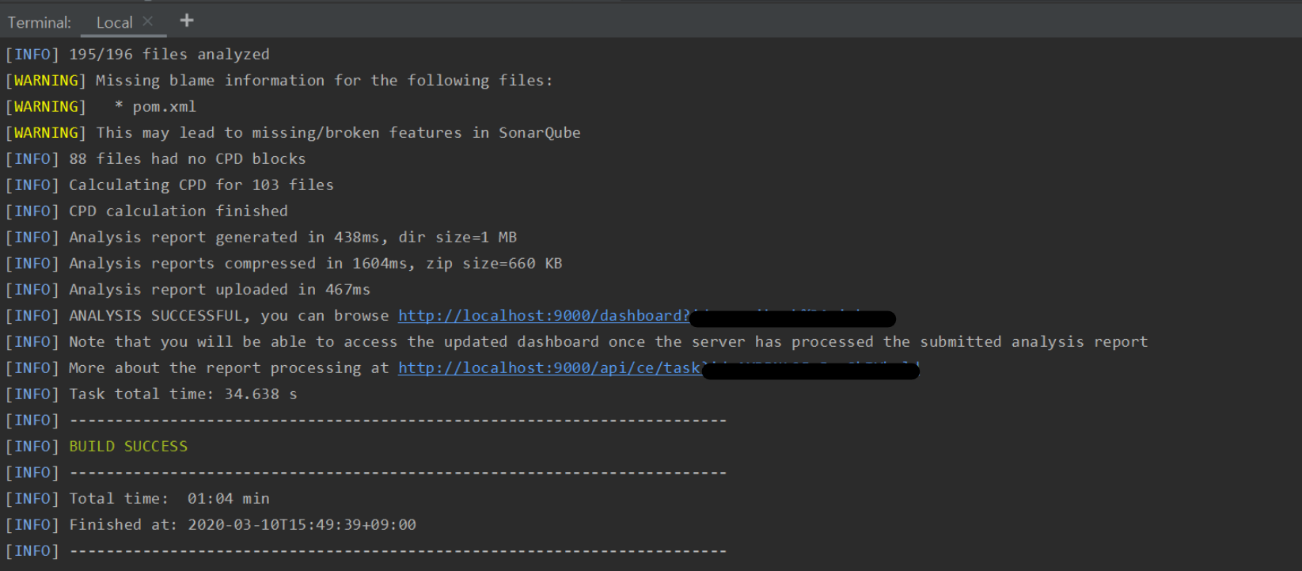
ERROR:

ERROR: Re-run SonarQube Scanner using the -X switch to enable full debug logging.

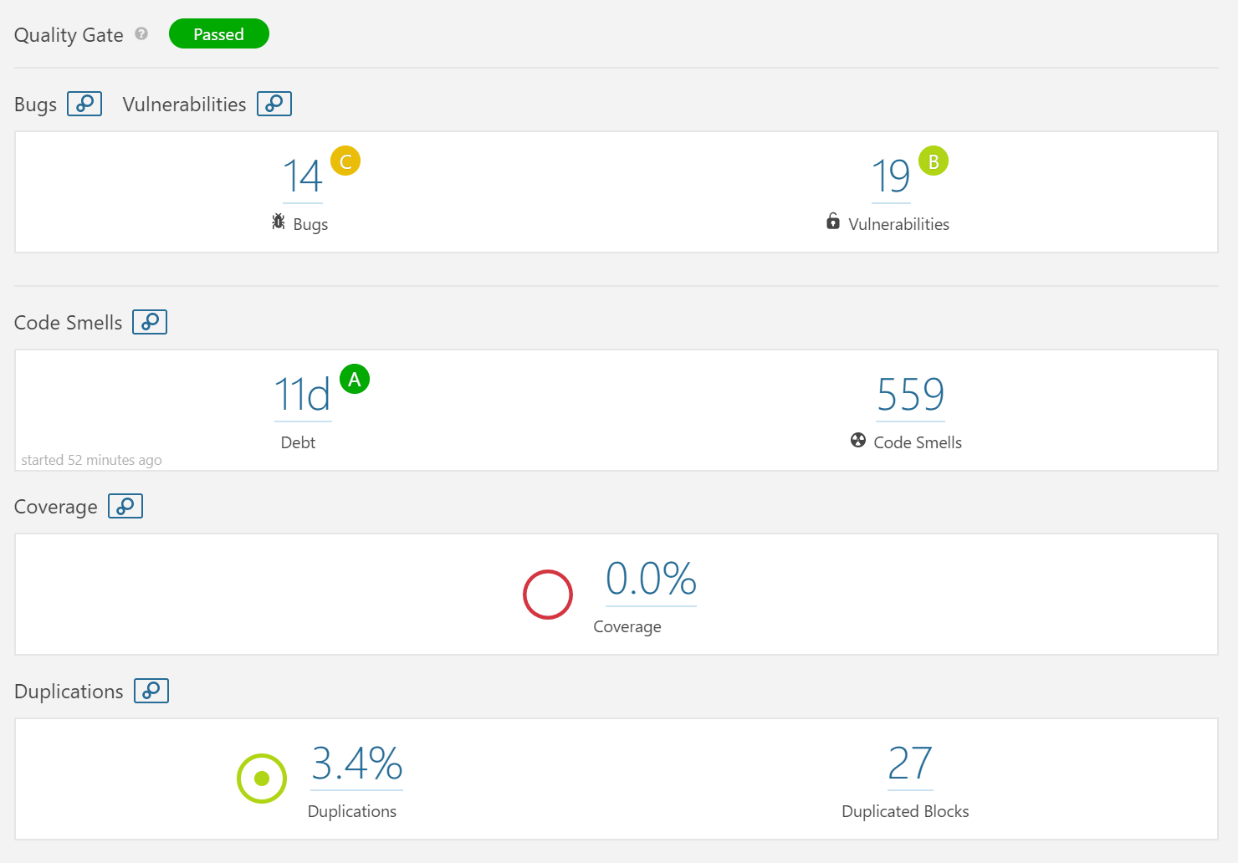
The SonarQube Scanner did not complete successfully

12:53:21.909 Creating a summary markdown file...

12:53:21.918 Post-processing failed. Exit code: 1



빌드가 성공하면 <http://localhost:9000/dashboard>에 접속합니다.



와! 드디어 소나큐브 접속에 성공했습니다!

초기에 진행했던 프로젝트라 그런지 테스트가 하나도 없어서 Coverage가 0퍼인 처참한 모습도 보입니다 🧐

# jacoco

## **Snippets(Gradle)**

implementation group: 'me.proton.core.gradle-plugins', name: 'jacoco', version: '1.1.2'

## **POM Files**

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<!-- This module was also published with a richer model, Gradle metadata, -->

<!-- which should be used instead. Do not delete the following line which -->

<!-- is to indicate to Gradle or any Gradle module metadata file consumer -->

<!-- that they should prefer consuming it instead. -->

<!-- do\_not\_remove: published-with-gradle-metadata -->

<modelVersion>4.0.0</modelVersion>

<groupId>me.proton.core.gradle-plugins</groupId>

<artifactId>jacoco</artifactId>

<version>1.1.2</version>

<name>jacoco</name>

<description>Proton gradle plugins for Android</description>

<url>https://github.com/ProtonMail/protoncore\_android</url>

<licenses>

<license>

<name>GNU GENERAL PUBLIC LICENSE, Version 3.0</name>

<url>https://www.gnu.org/licenses/gpl-3.0.en.html</url>

</license>

</licenses>

<developers>

<developer>

<id>opensource@proton.me</id>

<name>Open Source Proton</name>

<email>opensource@proton.me</email>

</developer>

</developers>

<scm>

<connection>git@gitlab.protontech.ch:proton/mobile/android/proton-libs.git</connection>

<developerConnection>https://gitlab.protontech.ch/proton/mobile/android/proton-libs.git</developerConnection>

<url>https://gitlab.protontech.ch/proton/mobile/android/proton-libs</url>

</scm>

<dependencies>

<dependency>

<groupId>org.jetbrains.kotlin</groupId>

<artifactId>kotlin-gradle-plugin</artifactId>

<version>1.7.10</version>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>org.jacoco</groupId>

<artifactId>org.jacoco.core</artifactId>

<version>0.8.7</version>

<scope>runtime</scope>

</dependency>

</dependencies>

</project>

Apache Maven

## **Snippets**

<dependency>

<groupId>me.proton.core.gradle-plugins</groupId>

<artifactId>jacoco</artifactId>

<version>1.1.2</version>

</dependency>

## **POM Files**

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<!-- This module was also published with a richer model, Gradle metadata, -->

<!-- which should be used instead. Do not delete the following line which -->

<!-- is to indicate to Gradle or any Gradle module metadata file consumer -->

<!-- that they should prefer consuming it instead. -->

<!-- do\_not\_remove: published-with-gradle-metadata -->

<modelVersion>4.0.0</modelVersion>

<groupId>me.proton.core.gradle-plugins</groupId>

<artifactId>jacoco</artifactId>

<version>1.1.2</version>

<name>jacoco</name>

<description>Proton gradle plugins for Android</description>

<url>https://github.com/ProtonMail/protoncore\_android</url>

<licenses>

<license>

<name>GNU GENERAL PUBLIC LICENSE, Version 3.0</name>

<url>https://www.gnu.org/licenses/gpl-3.0.en.html</url>

</license>

</licenses>

<developers>

<developer>

<id>opensource@proton.me</id>

<name>Open Source Proton</name>

<email>opensource@proton.me</email>

</developer>

</developers>

<scm>

<connection>git@gitlab.protontech.ch:proton/mobile/android/proton-libs.git</connection>

<developerConnection>https://gitlab.protontech.ch/proton/mobile/android/proton-libs.git</developerConnection>

<url>https://gitlab.protontech.ch/proton/mobile/android/proton-libs</url>

</scm>

<dependencies>

<dependency>

<groupId>org.jetbrains.kotlin</groupId>

<artifactId>kotlin-gradle-plugin</artifactId>

<version>1.7.10</version>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>org.jacoco</groupId>

<artifactId>org.jacoco.core</artifactId>

<version>0.8.7</version>

<scope>runtime</scope>

</dependency>

</dependencies>

</project>