SONARQUBE – HOW TO SETUP JAVA AND ANGULAR APPLICATIONS WITH JENKINS AND MAVEN

**WHAT YOU’LL LEARN**

* How to install Jenkins and SonarQube on your Linux system
* How to integrate SonarQube with Jenkins
* Sonar Scanner integration with Maven plugins and NPM
* How to scan Java and Angular projects
* SonarQube multi module Maven project integration
* Identifying Bugs, Vulnerabilities, Debt, Code Coverage and Code smells in Projects

**Requirements**

Linux and DevOps basics

Basic knowledge in Maven

Any development experience with Java and/or Angular is a bonus

**DESCRIPTION**

This is a hands-on practical course that will guide you through the process of installing, integrating and using SonarQube.

In the process you will get to know DevOps tools like Jenkins and learn how to setup and manage them.

Configuring SonarQube for continuous code inspection is a must for any team that delivers high-quality maintainable products.

Many courses present the functionality of SonarQube. Knowing all the details of a product requires time and dedication. In my opinion you should use SonarQube in order to continuously examine your product and spend your time not on configuring third-party tools, but on delivering a better product. That’s the idea behind the Sonar team.

This course is focused on Maven projects. It reflects all scenarios that can occur during the integration of SonarQube with your single- or multi module Maven applications. We will briefly go over maven basics, and we will discuss single module Java applications, single module Angular applications, as well as multi-module mixed applications, where your Java and Angular codebase is placed in different modules.  
Finding the right pieces to the puzzle is often times demanding, but using the resources I’ve prepared, you will be able to configure your projects without wasting any valuable time on research.

**DURING THE COURSE, YOU WILL LEARN TO:**

* Install and setup Jenkins and SonarQube
* Integrate your Maven Projects with SonarQube
* Pass code coverage reports manually
* Create a Jenkins pipeline that automates scanning for you
* Identify bugs, code smells, vulnerabilities and code debt in SonarQube
* Create quality gates and quality profiles
* Understand maintainability, reliability and security ratings in SonarQube

**WHO THIS COURSE IS FOR:**

* Angular and Java developers that want to build quality and maintainable products
* Managers and Architects that are part of the decision-making staff
* DevOps teams

**PRESENTATION PART**

INTRODUCTION

If you’re a developer, a manager, or a person that’s dealing with the development of a software product, you’ll very well know the meaning of technical debt, low test coverage and continuous code inspection – those things take time. And you often times don’t have it.

My name is George, I’m a software engineer, and I’ll share with you my knowledge on SonarQube.

You need a reliable tool that can visualize the state of your code, so that different technical or managerial decisions could be made.

If you don’t need to deliver a high-quality product, that’s fine. This course is not suitable for you. But if you want to integrate a solution that can help you out throughout the lifecycle of your project, and do it quickly and seamlessly, then this is the right course for you.

SonarQube fits with your existing tools and simply raises a hand when the quality or security of your codebase is impaired. It’s really flexible and can be integrated with many DevOps tools. In our case, we’ll be using Jenkins to automate SonarQube scans.

If you haven’t dealt with rigorous software development before, this is a perfect starting point for you – SonarQube is the tool to catch tricky bugs, help you visualize code or dependency vulnerabilities and assist you in keeping your codebase clean and maintainable, so that developer velocity could be increased.

SonarQube supports over 20 programming languages. It this series, we’ll be focusing on Java and Angular.

After installing SonarQube and Jenkins, we will take a look what a multi-module Maven project looks like and what are the best scan configurations. We’ll also cover individual Java and Angular projects, and Sonar advanced options.

My target as an engineer is to deliver high quality software increments as fast as possible, and SonarQube helps me with that. I’m sure this course will help out your team as well.

Let’s cover SonarQube’s architecture and integration options first.

SONARQUBE ARCHITECTURE AND INTEGRATION

SonarQube Community edition, the one we’re going to be using, is free and open-source, and it consists in the following components:

* SonarQube Scanner – Generally speaking an application that processes scan results to your SonarQube server. You can use the standalone scanner or make use of the ones available for Maven, Ant or Gradle. As an input, the sonar scanner receives the codebase and test coverage report of your projects and processes them to the SonarQube Server
* SonarQube Server – The brains of SonarQube consists in a webserver, which visualizes the scan reports; A ComputeEngine or different services that allow a scan to be processed and stored in a database, and Elasticsearch – the well-known distributed JSON-based search and analytics engine. The server is extensible, you can use different plugins to add functionality to SonarQube, such as SCM Plugin, language plugin etc.
* After the data is processed, it is stored in a database. SonarQube supports Oracle, PostgreSQL and MS SQL. Previous versions – like version 6.4 used MySQL, but the team decided to deprecate and discontinue its usage. We’re going to be using SonarScanner version 8.2

A typical Sonar scan looks like this:

* Whenever a change is pushed in a source-controlled system – git, mercurial, perforce, it should trigger a ci-build. The build typically executes unit tests and end-to-end tests.
* After a successful integration, an automated job can run the SonarScanner in the source-controlled system, so that a report could be generated and pushed in the SonarQube server.
* After the report is pushed, it’s processed, persisted in the database, and visualized in the Sonar Web server.
* Teams can then access the web server and address the issues SonarQube has visualized for them; or use another tracking system and benefit from the reporting API or email distribution support features SonarQube presents.

Things to have in mind:

* The SonarQube Platform cannot have more than one SonarQube Server and one SonarQube Database.
* The SonarQube Server and the SonarQube Database must be located in the same network – in our case, it will be on the same machine
* SonarScanners don't need to be on the same network as the SonarQube Server - you can have a dedicated Jenkins server that is used to trigger Sonar scans and nothing will stop them. Well, only network failures.

Our development environment:

* We’ll use an Ubuntu 18.04 server, on which we’ll install Jenkins, SonarQube and PostgreSQL. We’ll not be wasting any additional time, as I’ve prepared a script that automates the task for you. But if you have specific preferences, feel free to use the script and update it any way you like.
* I have the VMware workstation virtualizing software, on which I’ll create my virtual machine. If you have a license, you can use this software, or install and use VirtualBox, or any other piece of virtualizing software.
* The script can also be installed on Ubuntu Desktop.

Without further ado, let’s get to the installation.

INSTALLING SONARQUBE AND JENKINS

As I’ve mentioned earlier, I’ve prepared a script for your convenience.

You can clone it from <https://github.com/WhiteLord/sonar-jenkins>

I’ve currently logged in my Ubuntu 18.04 server, and I’ll execute the script. But before that, let’s have a quick look.

…  
As I’ve mentioned earlier, feel free to modify any parts of the script as you see fit.

…

Now, as we’ve finished installing Jenkins and Sonar, the first thing I’d do is activate Jenkins. The script prints out a secret, which you’ll be using to open access Jenkins for the first time. After that, installing plugins depends more or less on your personal preferences.

Since I’ll be using this Jenkins instance for automating other jobs, I’d install a few plugins.

…

The SonarQube configuration is pretty straightforward as well.   
You can find the Sonar Server configuration options under /opt/sonar/conf folder.  
So, these are the settings we’ll be using throughout the course.

Now, let’s go through the project we’re going to scan.

MAVEN BASICS

If you haven’t worked much with Maven, now is a great time to understand the basic concept.

Maven is one of the best build tools out there. Philosophically speaking, it’s more than just a build tool.  
Maven is centered around the concept of POM files (Project Object Model). A POM file is an XML representation of project resources like source code, test code, dependencies (external JARs used) etc. The POM contains references to all of these resources. The POM file should be located in the root directory of the project it belongs to.

A maven command needs to be executed in a folder, where a pom.xml file resides. Maven will then execute pom.xml file input.

The build process in Maven is split up into build life cycles, phases and goals. A build life cycle consists of a sequence of build phases, and each build phase consists of a sequence of goals. When you run Maven you pass a command to Maven. This command is the name of a build life cycle, phase or goal. If a life cycle is requested and executed, all build phases in that life cycle are executed. If a build phase is requested and executed, all build phases before it in the pre-defined sequence of build phases are executed too.

Unless your project is small, your project may need external Java APIs or frameworks which are packaged in their own JAR files. These JAR files are needed on the classpath when you compile your project code.

Keeping your project up-to-date with the correct versions of these external JAR files can be a comprehensive task. Each external JAR may again also need other external JAR files etc. Downloading all these external dependencies (JAR files) recursively and making sure that the right versions are downloaded is cumbersome. Especially when your project grows big, and you get more and more external dependencies.

Luckily, Maven has built-in dependency management. You specify in the POM file what external libraries your project depends on, and which version, and then Maven downloads them for you and puts them in your local Maven repository. If any of these external libraries need other libraries, then these other libraries are also downloaded into your local Maven repository– in Linux and MacOs systems, that’s the hidden .m2 folder in the ~ directory (or /home/<user>), and user is the current privileged user you’re executing the task with.

I presume you’ve already installed maven on your development system. If not, the apache maven website presents a handy guideline for MacOs, Linux and Windows users.

Now, if we navigate to a directory, containing a pom.xml file, we can run mvn install from the command prompt.

This command executes the build phase called install (part of the default build life cycle), which builds the project and copies the packaged JAR file into the local Maven repository. Actually, this command executes all build phases before install in the build phase sequence, before executing the install build phase.

You can execute multiple build life cycles or phases by passing more than one argument to the mvn command. Here is an example: mvn clean install

This command first executes the clean build life cycle, which removes compiled classes from the Maven output directory, and then it executes the install build phase.

The project I’ve prepared is a multi-module Maven project. It consists of a root pom.xml file which describes its modules.

Take a closer look at the packaging option in the root pom.xml. It says pom – that’s because this kind of packaging represents a specification that states that the primary artifact that is going to be produced is not a war or a jar, but the pom.xml itself.   
This way of packaging is used when you don’t want to produce an artifact.  
In our case, build artifacts are going to be generated in the target folders of every leaf child project.

As a real world parent-to-child connection where the child inherits its parents’ DNA, the child projects in Maven inherit all properties, settings and profiles from their parent’s pom.xml, if they’re not overridden. Soon you’ll understand why this is so important.

Our example projects have this parent-child relationship.

As we continue our voyage through the parent-child relationships, we see that there are some pretty easy to understand project pom.xml files, such as the register-service pom. It declares two dependencies – for infra mapper, which is one of our projects, and junit. I assume you’re familiar with junit, but if not – that’s a unit testing framework Java uses.

The frontend project has modules of its own. There is the Angular application, under frontend-client, and two java applications, respectively frontend-service and frontend util.

Next, let’s see how we can include a scan report in SonarQube.

SONARQUBE REPORT GENERATION OVERVIEW

SonarQube provides several different options for scanning a project.

Firstly, you have the sonar scanner, which is the default option if you haven’t integrated any other forms of scanning.

You’ve got plugins for Ant, Jenkins and Grade.

The one we’re going to look at closely, however, is the maven sonar plugin.

I suggest we open the SonarQube documentation now.

In order to use the generated test results for our scan, we need to provide some sonar properties to the root pom.xml.

Secondly, and you’re not going to see this in the documentation, you need to generate unit test reports of every module before executing sonar’s maven plugin. Otherwise you’ll have an incomplete sonar report with no test coverage data. And that’s no good for you.

We need something that will create test coverage reports for Java, and for our Angular projects. I couldn’t find a maven plugin that does both things, and is supported by SonarQube, that’s why I implemented the following approach:

* I will use a maven plugin for java code to generate test coverage reports. The only rule is to include its configuration in the root (parent) pom of the project. Here we can benefit from JaCoCo - that’s a free code test coverage library for java, which reports are supported by SonarQube.
* I decided to go with another maven plugin for our Angular project. The test coverage reports will be generated the default way (executing ng test –codeCoverage=true) with karma coverage Istanbul reporter.

Before we create our implementation, let’s quickly go over how jacoco works.

CODE COVERAGE WITH JACOCO

Before Jacoco emerged as a code coverage tool, the team developed EMMA and its successor, EclEmma, which became a part of Eclipse foundation sometime around 2016 or 2017.

Evgeny Mandrikov and Marc Hoffmann built upon what they previously learned and started the Jacoco project in 2010.

Speaking about code coverage, you can have runtime profiling, like JVMPI and JVMTI, or do instrumentation of source code or byte code.

Jacoco team decided to go with byte code instrumentation, both offline and on the fly. Both approaches have their positive and negative sides.

Offline instrumentation for example is used whenever our test implementaion include Powermock or Powermockio. If we use on the fly instrumentation with tests that contain Powermock in one way or another, Jacoco won’t be able to produce valid results, and the code coverage report will show 0% coverage, despite our best efforts.

Jacoco uses the .class files to create execution files, relative to the test coverage task at the specific moment, compare them with the source code, and execute analytic functions to verify whether function calls or field modifications of the examined class, have taken place. Those analytic functions include the test scenarios, you as a developer write.

So, how does the magic happen?

As .class files are decompiled during analysis, Jacoco inserts probes at specific places. Those probes serve as a guarantee that a specific function call or action has taken place.

For example, if we have a function, and the function includes a return statement, a probe is placed right before the return statement. If there’s a function that calls another function, or there’s a conditional step, the probe is inserted right before the call of the second function.

A question for you – what kind of probe does Jacoco use? What could be fast, thread safe (as there are only write, and no read operations done) and small enough to be used? An array of Booleans. You wouldn’t care if one thread overrides the n-th element of the array with a true value, while it previously was initialized and set to true via another thread. At the end of the day, the goal is to verify whether a specific action took place.

Jacoco has a tough time dealing with reflections and respectively, Powermock, using on-the-fly instumentation. Depending on the JDK implementation (but often time pretty much all of them), Powermock uses similar approaches to instrument or modify Java bytecode. This is something you should not spend so much time on as a developer, as the compiler implementation and results should be eventually trusted. The end product you produce is not abstract knowledge, but a concreate jar or war file.   
As a dev myself, I trust the Java compiler, and I trust Jacoco, as they’ve proven to be in the top 5 bug reporting entities that contribute to the development of JDK, along with apache maven, intellij and other.  
And since I found out that Jacoco produced the same test coverage results whenever I use offline or on-the-fly instrumentation, excluding the cases of Powermock including tests, I decided to go with using offline instrumentation for all the java examples in this course.   
Yes, there could be some minor differences in some corner cases that a normal everyday startup or corporation won’t even come across in 1 million lines of code, but that’s the approach I decided to take.  
And finally, since I’m working with other people, and I can’t enforce the use of one tool over another in the context of a specific project (talking Powermock here), I decided to go with offline instrumentation.

In the next video, we’ll see how to actually implement offline instrumentation in our root pom.xml file.

SONARQUBE JAVA REPORT GENERATION WITH JACOCO

Since we’ve gone through the basics of Jacoco, let’s get to their website and learn something practical.

As we can see, we’ve got to add a plugin declaration containing the corresponding jacoco group id and artifact id.

So, we’ve already added an entry. Great.

From now on we’ll have to use the documentation, provided in the jacoco website, as well as usitlize the following command when we need to: mvn help:describe -Dplugin=org.jacoco:jacoco-maven-plugin -Ddetail

This command will list all goals and available parameters for use with a specific plugin.

I’ve added an execution entry with id prepare-agent. If we type in the command we’ve just discussed, and head to the description, we can see that the prepare-agent prepares a property, pointing to the jacoco runtime agent that can be passed as a vm argument to the application under test.  
Meaning – Jacoco will create and agent BEFORE we execute the code coverage reports. We’ve got to have this agent, otherwise no analysis would be made and no test report would be generated.

Maven does this automatically.

As I mentioned in the previous video, I’m going for the approach of using offline instrumentation for test coverage report generation. As we’ve discussed, there are positive and negative sides to this, and the one thing that makes me want to implement an offline instrumentation is the fact that whenever we have a powermock inclusion in our tests, we won’t have to double-check test results and verify that the code coverage report actually shows real values.

So, the instrumentation consists in two parts – the actual analysis of the .class files and source code, and that includes the probing part where Jacoco finds out which code parts were actually tested, and restoring instrumented classes.

Next, we need to make sure that we actually generate test coverage reports.

Well, a base project, or a parent maven project that does not produce and jar/war files, like in the case of the base pom.xml does not have tests. It could not have coverage reports.

The one thing that it can have, though, is a folder that can hold the test coverage report of all modules. That’s the reason we have a report-aggregate execution inclusion. In the base folder, where our root pom.xml resides, jacoco/maven will create a folder that contains all test coverage reports.

But we still have not told our modules that include source code and tests to gererate coverage reports.

That we can do by adding an entry with the id of generate-report and goal report.

Since we have parent to child relationships between projects, the child projects will inherit the configuration of their parents.

The generate-report execution will create test coverage reports only in the modules it finds source code and tests. It will not execute in those modules where we’ve stated that our maven target is a pom. Thus we’re not having any byproducts that are malicious, show false test coverage reports (meaning zero coverage) or introduce direct or orthogonal problems we have to deal with.

Having this in mind, we should be able to execute mvn clean install, and still get the test report results, as the default phase for prepare-agent is initialize, for report is verify, and for instrument is process classes, which comes directly after compile.

Yes, but, sometimes we’ve got to be a bit more specific.

I found out that I don’t want to create test coverage report on maven install. That’s why, using the following command does the job perfectly:

mvn org.jacoco:jacoco-maven-plugin:prepare-agent verify org.jacoco:jacoco-maven-plugin:report

The command prepares the jacoco agent, as we’ve specified, and then runs the report on the verify phase. The result of the command will be the newly created test reports in our directory.

Let’s double check if that really works.

SONARQUBE ANGULAR REPORT GENERATION WITH JACOCO

As we’ve already discussed test report generation with Java, it’s time for Angular now.

Usually developers would have different project structures. In the context of a big application, you might have several angular applications nested inside the big project.

For that reason, I like keeping Angular projects in another module, per project – if we have 3 Angular projects inside our big project, I’d have three maven modules – one for each Angular project.

The reason I’m having a maven project is the fact that the administration is much easier, you can have different profiles, associated with the current project, based on your personal preferences, and you can directly execute the build of the application from base pom.xml. Not that you can’t execute npm commands all the time, but it’s tedious and you’ve got to always look up what’s the next command you need to execute. And maven can automate this for you.

Now, if we go ahead and check our angular project, we can see the following configuration.

We’ve got a profile, which is going to be activated with sonar-scan. The project includes frontend-maven-plugin which allows us to execute different npm commands directly from maven. The cool thing about this plugin is that it can use a different npm and node version from the ones that are globally installed on your system. This is a perfect opportunity if you’re having limited access to the server where your team’s automation software resides and you’ve got limited capability when it comes to updating npm or node versions. By specifying nodeVersion in the configuration options, you can chose which node version you will use. In our case, it’s 10.16.3. Make sure that your version is higher than 10. Versions 10 and below can cause serious code coverage mismatches.

Following the pom declaration, we’re running npm install, and npm build, followed by npm coverage, which calls the test-coverage script from package.json, which executes ng test with code coverage. One more thing. Don’t let those id entries fool you. You can type whatever you wish in there. I’ve done it so that you’d understand better what’s happening. And the last execution step is npm run sonar. If we check the package.json again, we’ll see that the script is executed from the node modules folder. So far, so good.

Now, let’s talk about configuration.

The sonar-project.properties file is required as an argument for the npm run sonar command.

In there we describe the sonar host url, we give a project key, which can be anything you like, we give a name to the project we scan.

The sonar sources and tests entries require you to enter the root directory where test and sources reside. In our case, that’s the src folder. Here you can add different folders if they contain any other sources or tests. Be sure to add them with commas, and don’t include any spaces around commas.

Next the exclusions – You don’t want sonar to include the following types at all– spec.ts, test.ts and .js.

The test inclusions however should contain spec.ts and test.ts

With coverage exclusions we define files or patterns for sonar – the ones you specify will not be treated as sources. Meaning – you can add here a file you don’t want covered and shown in SonarQube. In our case we’ve got this configuration.

The last entry specifies the file path that we need to pass to sonarqube. It’s an lcov.info file, containing test coverage report data.

It’s really easy to generate test coverage data and post it to sonarqube from our development environment. But it’s also tedious. Imagine if we have to generate reports once per week on Saturday and our wifi’s down or our workstation is damaged, or there is another factor that keeps you from executing the maven command.

That is why we rely on automation tools, like Jenkins.

In the next part we’ll take a look how to integrate test coverage report generation and posting the results to sonarqube from Jenkins.