**Chapter 12. Integrating Analysis Tools**

Most pipelines have some version of an “analysis” stage for doing things such as gathering code metrics, determining complexity, identifying bad coding practices and likely breaking points, and calculating potential resource costs such as technical debt. These analytics identify potential problems (some more serious than others), and fixing these “holes” can enhance key characteristics of the code such as readability, reliability, and maintainability.

In this chapter, we’ll look at how to integrate one of the most popular of these applications, SonarQube, into a Jenkins pipeline. We’ll also see how to integrate a separate tool, Jacoco, for code coverage analysis. Code coverage analysis is frequently integrated into a tool like SonarQube, but it’s worth understanding how to separate it out, given the important role that code coverage can often play in analyzing code.

For SonarQube, we’ll start by briefly discussing the tool and how it is integrated into a traditional pipeline. Then we’ll look at how that translates into a pipeline-as-code environment. Along the way, we’ll cover one of the most important aspects of using such a tool in a pipeline, as a way to pass or fail your pipeline stage based on selected thresholds set within the application.

Though we’ll utilize Gradle again here as a supporting technology, the approaches we use should be adaptable to most other technologies once you understand the basics.

Likewise, for Jacoco, we’ll briefly discuss the application, see how it has typically been integrated into a traditional pipeline, and then look at how we can migrate that into a pipeline-as-code.

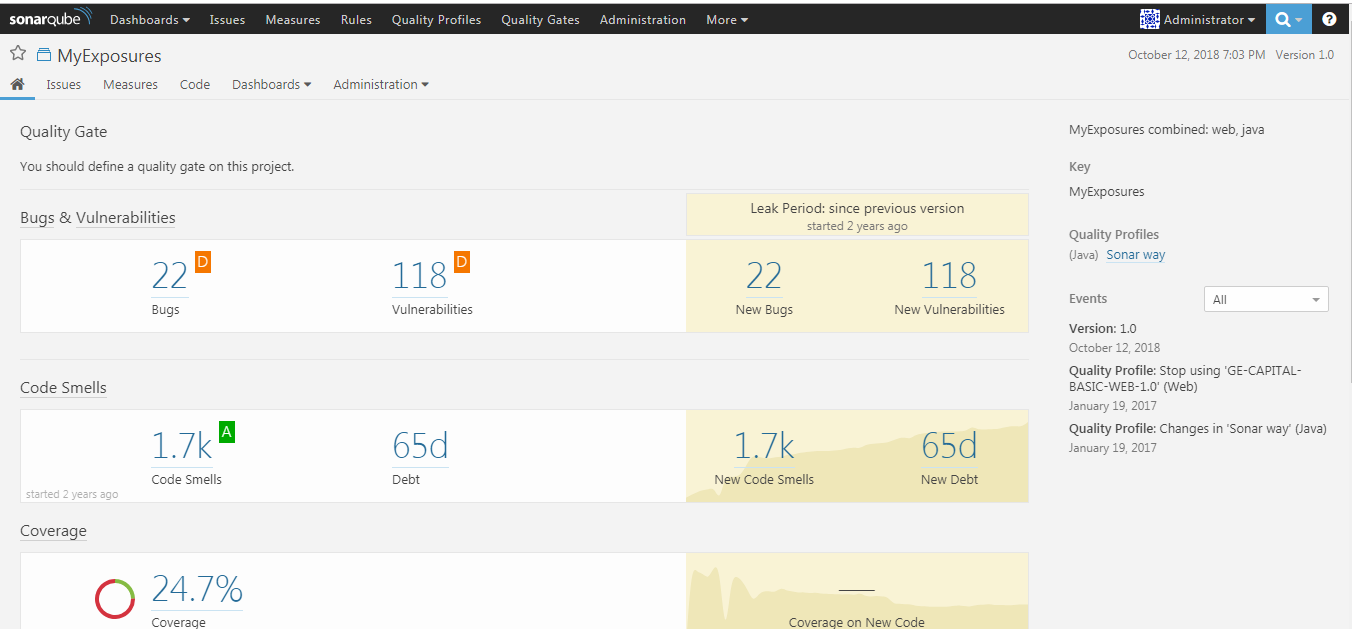
Let’s begin by discussing a little about what SonarQube offers for code quality analysis in a pipeline.

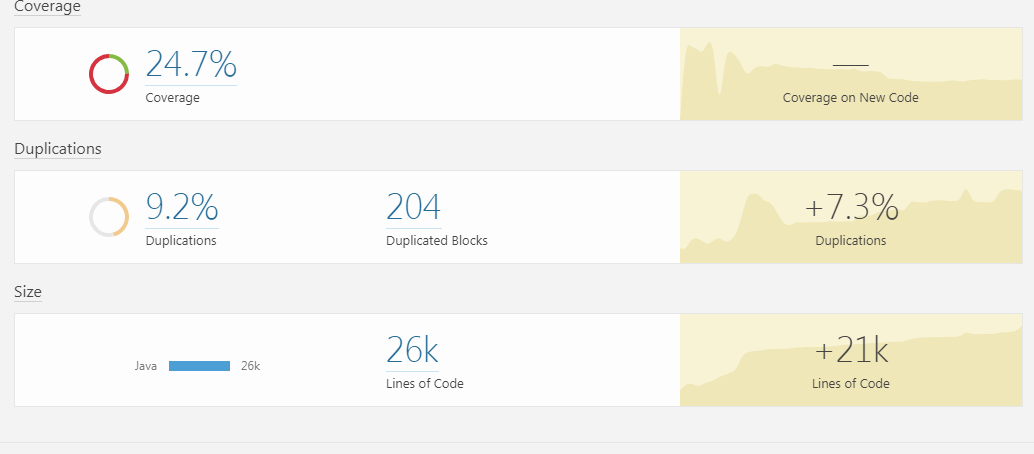
# SonarQube Survey

Per [its website](https://www.sonarqube.org/), SonarQube (formerly known as just “Sonar”) is an open platform for managing code quality in several key software areas, including:

* Architecture and design
* Comments
* Coding rules
* Potential bugs
* Duplications
* Unit tests
* Complexity

As you can see by this list, the core functionality covers a lot of territory and provides many metrics that can be beneficial. Within the SonarQube application itself, you can get a quick overview of how an analyzed project fared by looking at the dashboard. An example of that is shown in



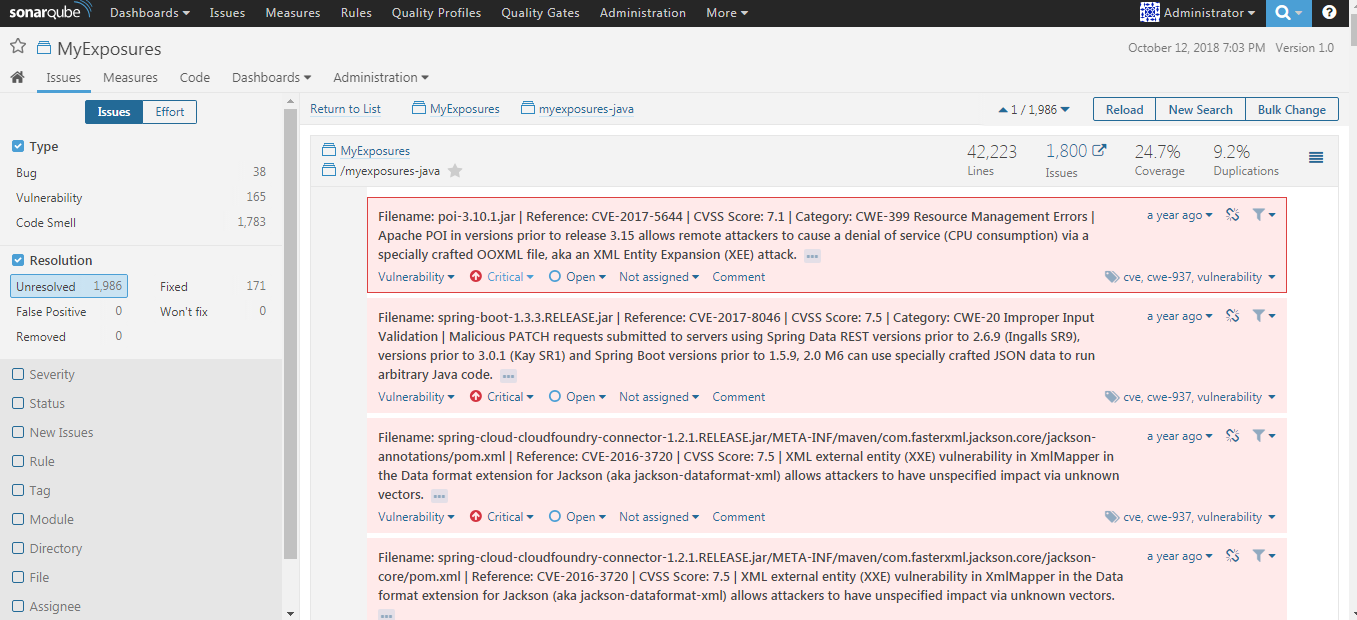


Beyond this core set, the “open” reference refers to the ability to extend the tool’s functionality with plugins to gather additional metrics, and to the ability to further define and tune the rules that govern the core functionality. This may be helpful as you start establishing criteria that your pipeline code needs to meet.

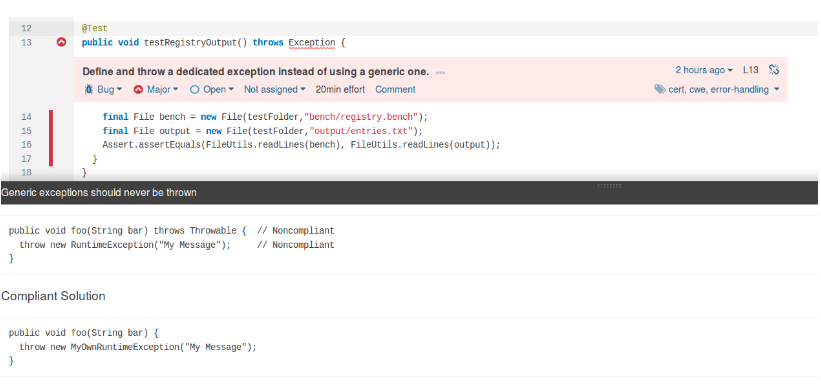
To begin to understand how it fits in with our pipelines, let’s take a quick look at individual violations flagged by SonarQube.

## **Working with Individual Rules**

SonarQube governs the conditions it checks based on a set of specified “rules.” When it analyzes source code and detects code that violates these rules, it flags the offending code and reports the rule violations. A simple example showing a set of violations that were found can be seen in

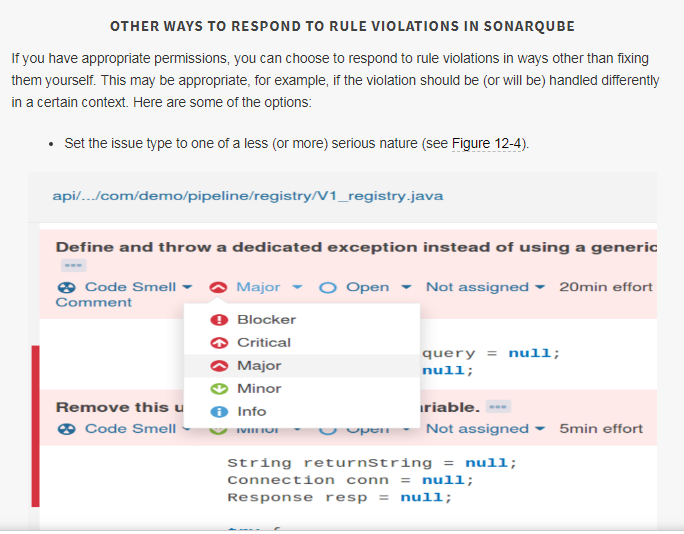


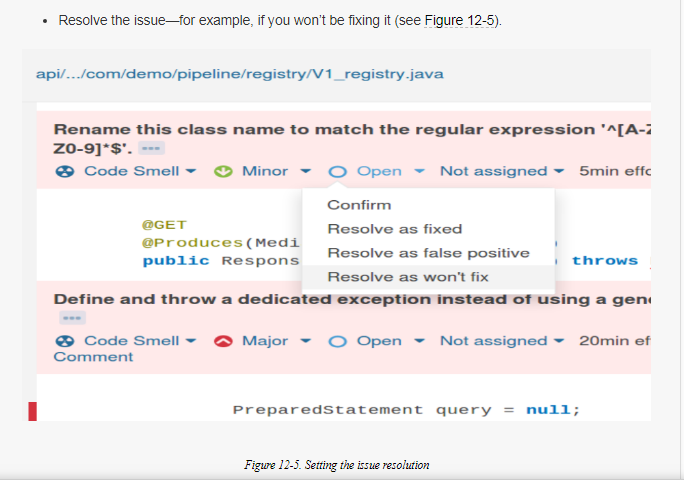
From here we can drill down into individual violations to get more information, as seen in

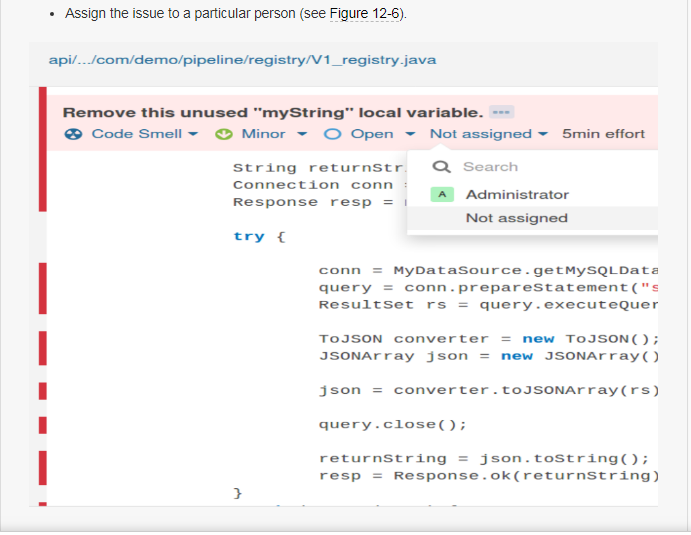


Note that if you go far enough, the explanation will not only show you the location of the offending source code, but also examples of solutions—ones that are noncompliant and ones that are compliant.

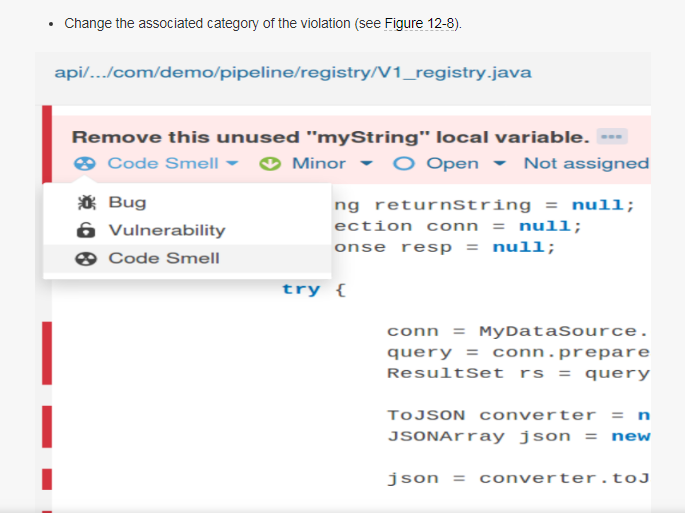
Assuming you agree with the analysis, you can go back and make a change to the source code to correct the issue, and then send the code back through another run of the pipeline (and thus SonarQube).











The amount of information generated by this type of analysis can be significant, depending on the size and scope of the code being analyzed. Analysis tools nearly always have ways to “turn down” the number of issues flagged, by ignoring certain types of items. But ultimately, in a continuous delivery environment, we want to establish thresholds for quality analysis. The goal is that our candidate release will only pass (be able to move further down the pipeline) if it meets or exceeds the minimum threshold for desirable characteristics, and falls below the maximum threshold for problematic issues. Examples might be that a minimum percentage of our code has to be covered by unit testing and we can only have a certain maximum amount of issues flagged as critical code alerts.

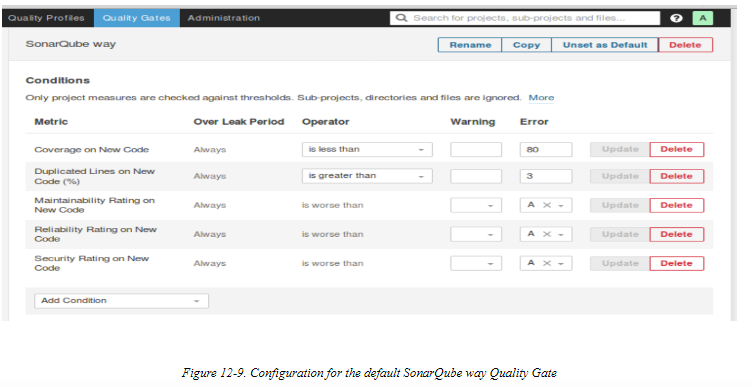
Within SonarQube, thresholds for various metrics and analyses can be set this way. Selected thresholds can be combined to form a single set of criteria for rendering a pass/fail judgment on the code being evaluated. In SonarQube, these pass/fail thresholds are called *Quality Gates* and the application of specific Quality Gates to different projects or technologies is done via *Quality Profiles*.

## **Quality Gates and Profiles**

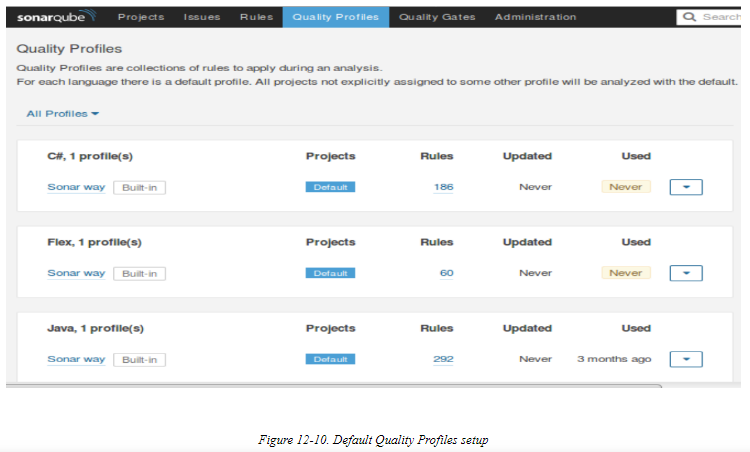
Quality Gates in SonarQube are made up of sets of conditions. Conditions, in turn, are made up of:

* Something to be measured (such as the number of critical issues or amount of code coverage)
* A period of time for the measurement (either current or over some defined period)
* A threshold value
* A comparison operation (such as “is less than,” “is greater than,” etc.)
* An error value or warning value, as needed

An example would be a condition in a Quality Gate that says we can have no more than two alert issues. Another might be a condition that requires a minimum of 80% code coverage by unit tests. Below pic  shows some of the conditions in the default “SonarQube way” Quality Gate that comes with SonarQube.



In short, we’re defining tests in different categories of quality analysis against threshold values. The implication is that we’re setting a baseline for how many “violations” we’re willing to tolerate to consider our code to be of good quality and suitable for production. The set of conditions functioning together as a Quality Gate allow us to translate thresholds into a single pass/fail status. Quality Gates can then be configured for different projects or technologies via Quality Profiles. For example, you might have one Quality Profile for your Java projects, another for your JavaScript projects, and another for your Python projects. Each of these could use the same Quality Gate(s) if the rules were widely applicable, or specialized Quality Gates for each language. [Figure 12-10](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch12.html#fig_DQ_pro_set)shows the default Quality Profiles setup for a SonarQube instance.

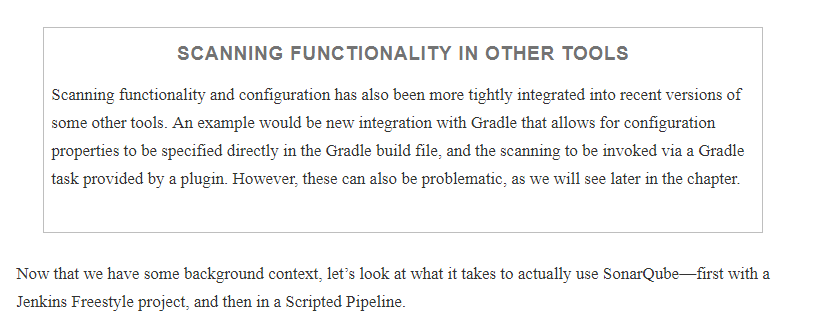


A full exploration of SonarQube, Quality Profiles, and Quality Gates is beyond the scope of this text. However, as noted previously, we can leverage the Quality Gate/Profile functionality as a pass/fail gate for an “analysis” stage in our pipeline. Before we can do that, though, there is one more aspect of using SonarQube that we need to understand—the SonarQube scanner.

## **The Scanner**

As the name suggests, a *scanner* for SonarQube is a program that scans the source code, checking for issues. The scanner is differentiated from the SonarQube server or instance that stores results, produces reports, etc., but both elements (some form of scanning and the server) are needed to form a complete analysis mechanism.

Historically, scanners (or “runners,” as they were sometimes known) were standalone, separate executables. The other piece that was needed, in conjunction with the scanner, was a configuration file of some kind that identified key SonarQube properties, such as the location of the source to be analyzed—including subprojects, the server, etc.

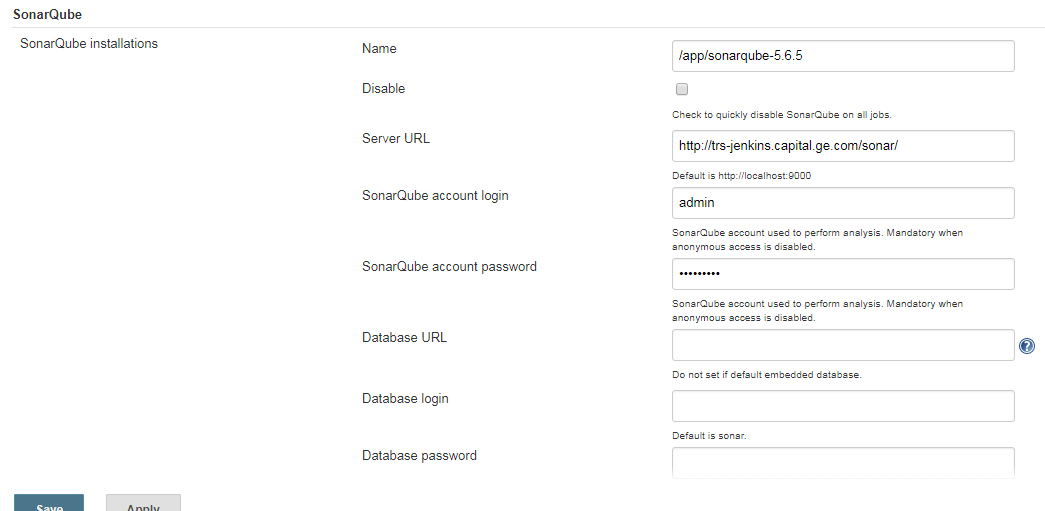


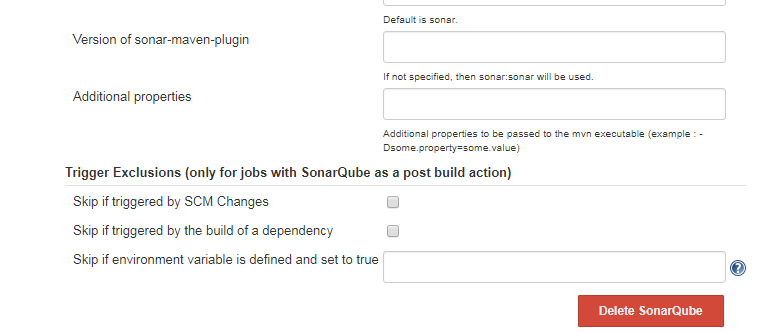
# Using SonarQube with Jenkins

Like with any other external application, integration of SonarQube with Jenkins requires a few pieces to be put in place. These include having the application up and running, having the plugin installed, global configuration of the server and (optionally) a scanner, and then invoking it within jobs. Let’s look at these areas in more detail.

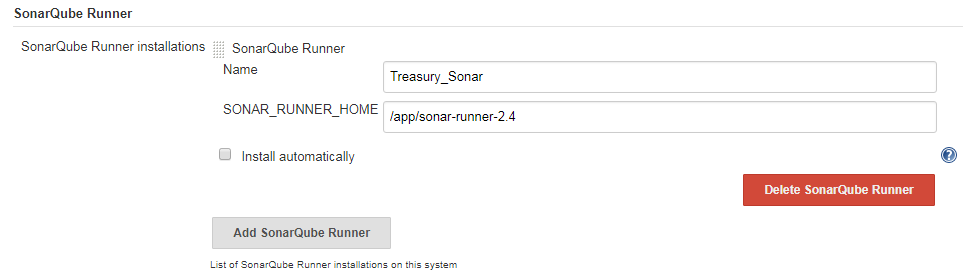
## **Global Configuration**

Below shows an example of the global configuration for a SonarQube server. (This is done on the Configure System page.)





We also need to have the SonarQube scanner installed and configured. [Below](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch12.html#fig_SQ_scanner_glob_config)  shows an example of this. (Note that it is configured on the Global Tool Configuration screen.)



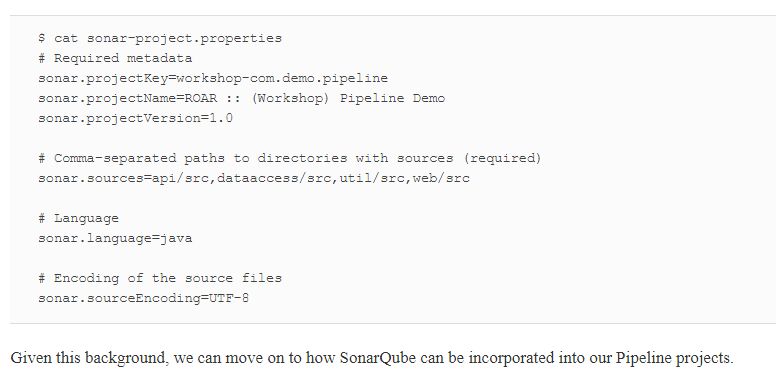
Once we have the application installed and running, the Jenkins plugin installed, and the global system and tool configuration done, we are ready to make use of SonarQube in our pipeline for the analysis. Consistent with our migration/conversion theme throughout the book, we’ll look first at how it would have been used in a Freestyle project.

## **Using SonarQube in a Freestyle Project**

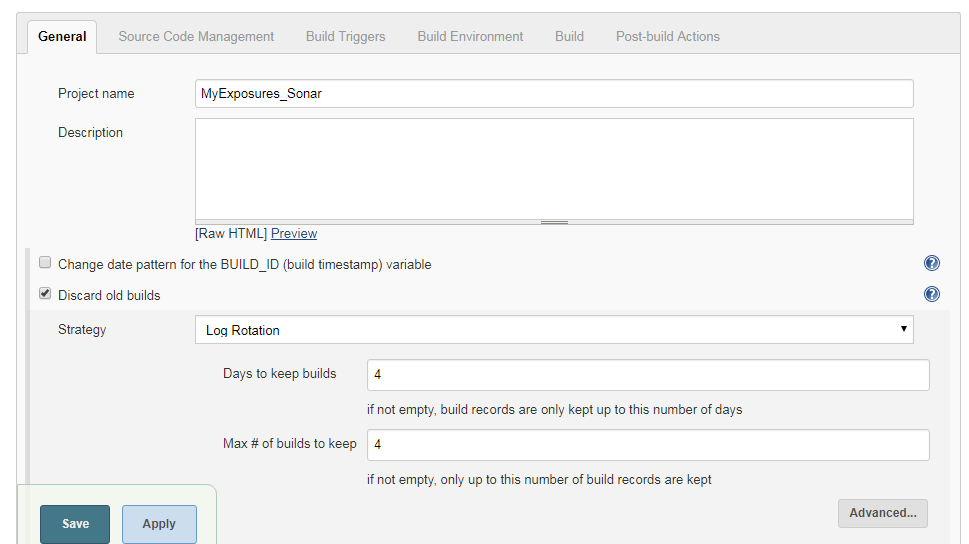
In a traditional Freestyle job environment, we might have had a separate “analysis” job that first called the scanner as a build step. The plugin would have provided some formal build steps that could be used to run the standard SonarQube scanner or an MSBuild scanner. Other applications will have relied on calls oriented around their specific syntax. We won’t go into all the options here since that isn’t our primary goal, but you can find examples of the different types of scanning for tools like Maven, Gradle, etc. on the SonarQube website.

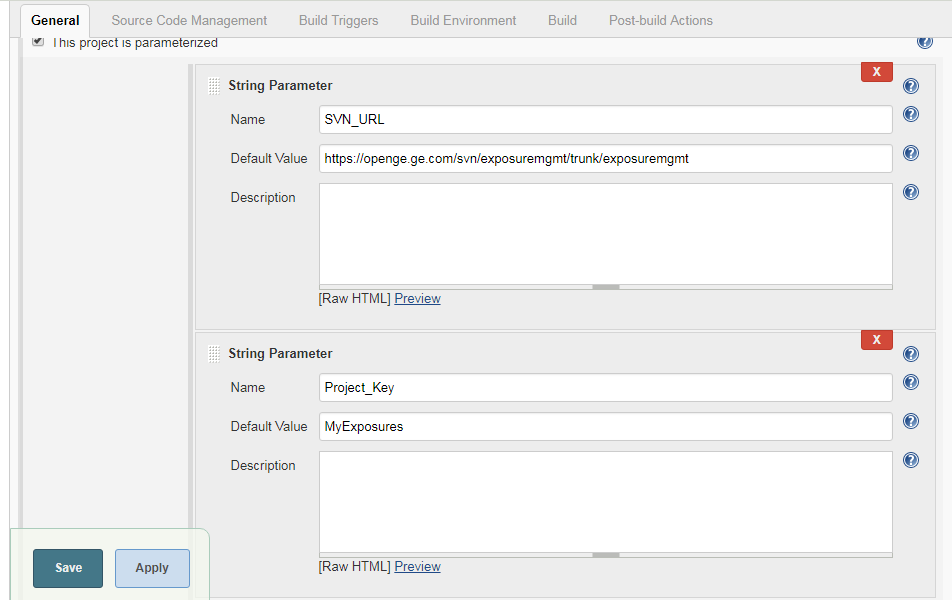
A simple fallback was to execute a shell step that simply invoked the scanner executable.

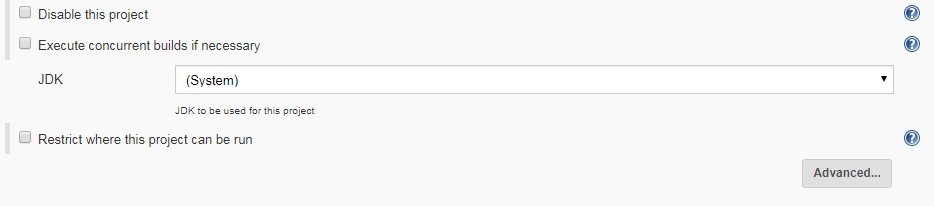
Any scanner invocation also needed to have some basic configuration values defined for it to know what to process, and how to reference it in SonarQube. These could be stored in a text file and the location pointed to in the Jenkins job. Or, in the case of some of the formalized build steps, they could be entered directly into a field in the web job. A simple example of a project configuration file might look like the following:

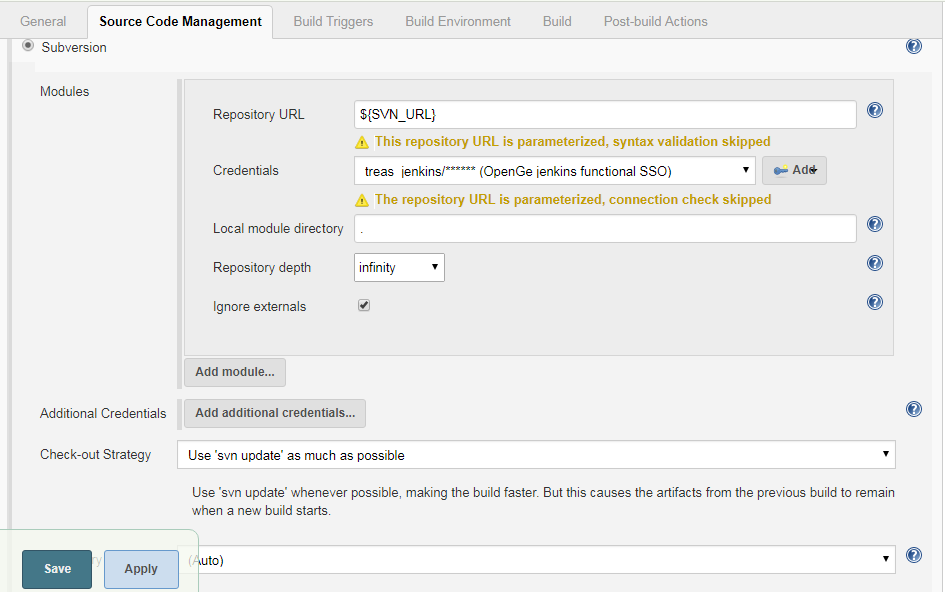


Now In our project how it is invoked (myexposures)

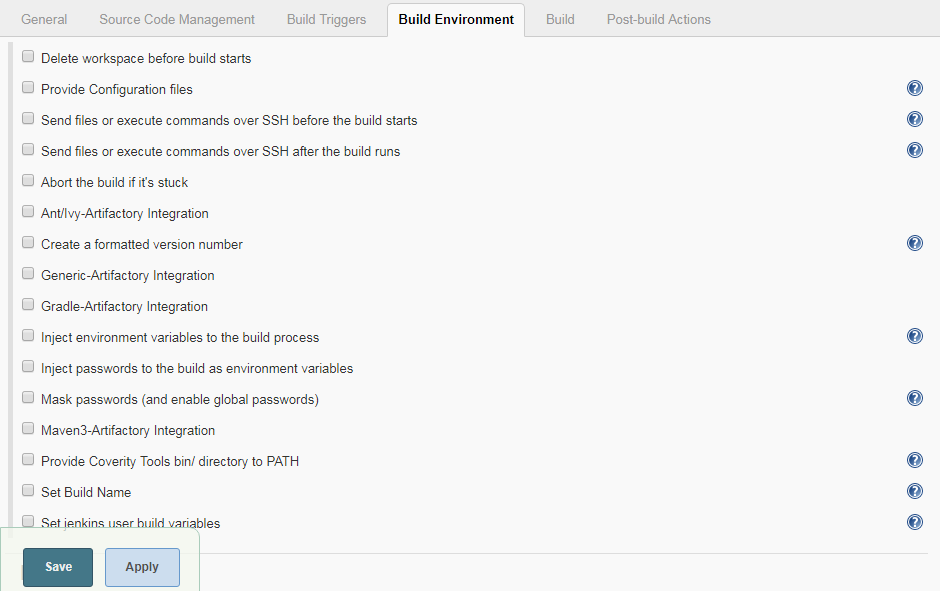


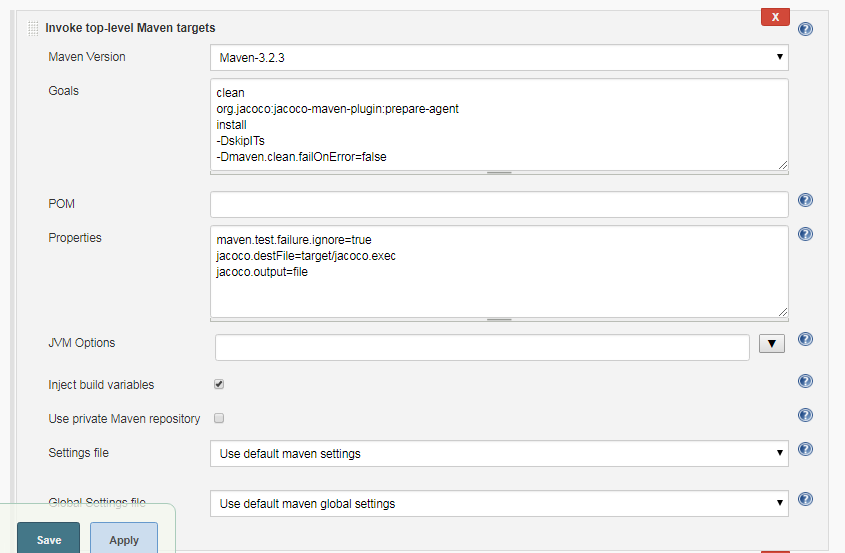


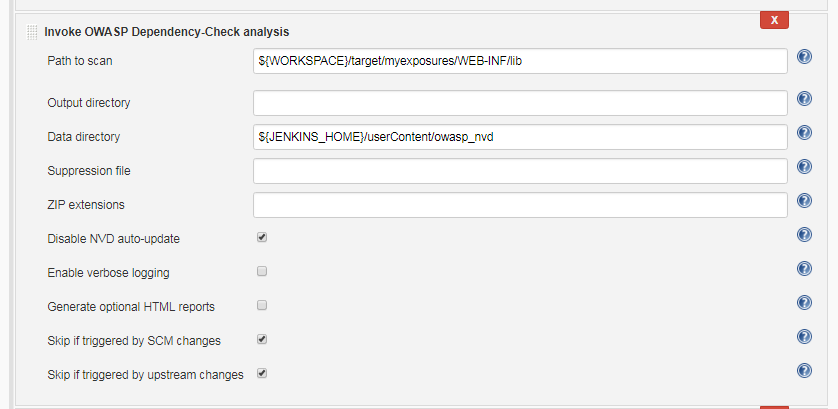


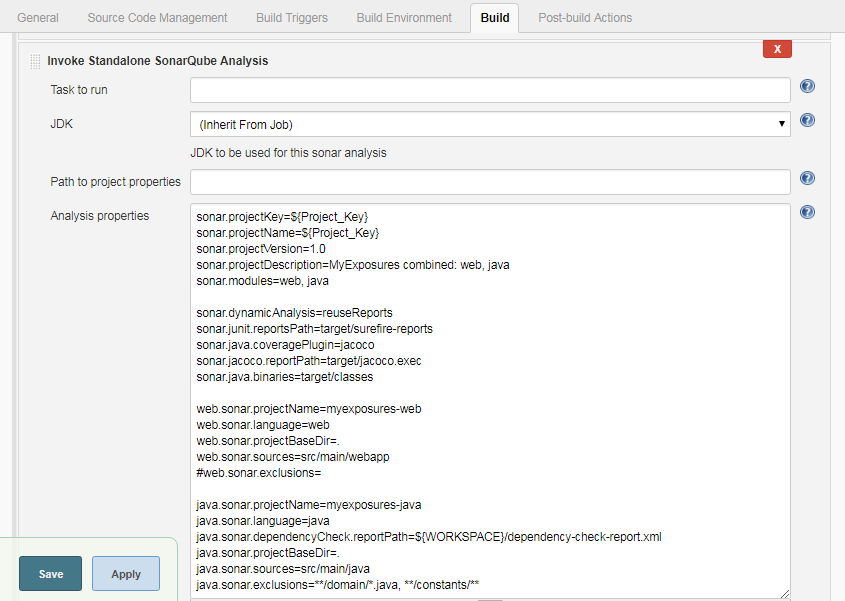


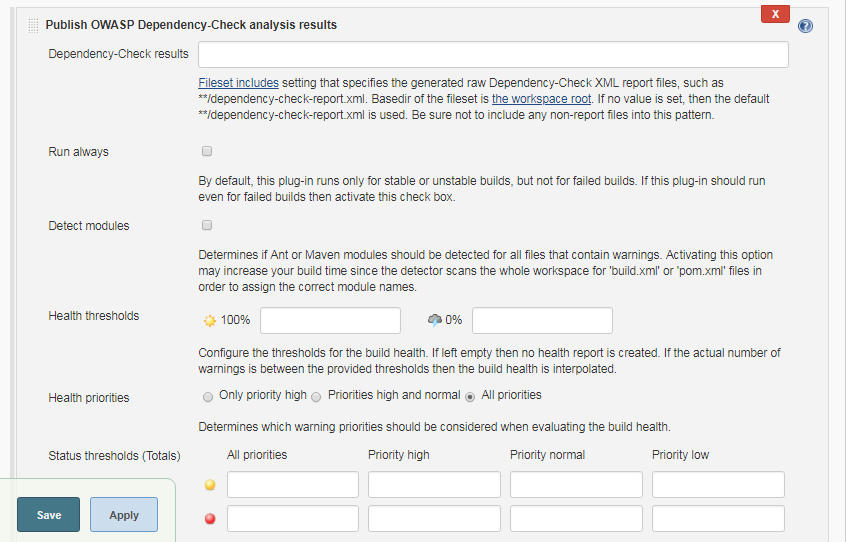


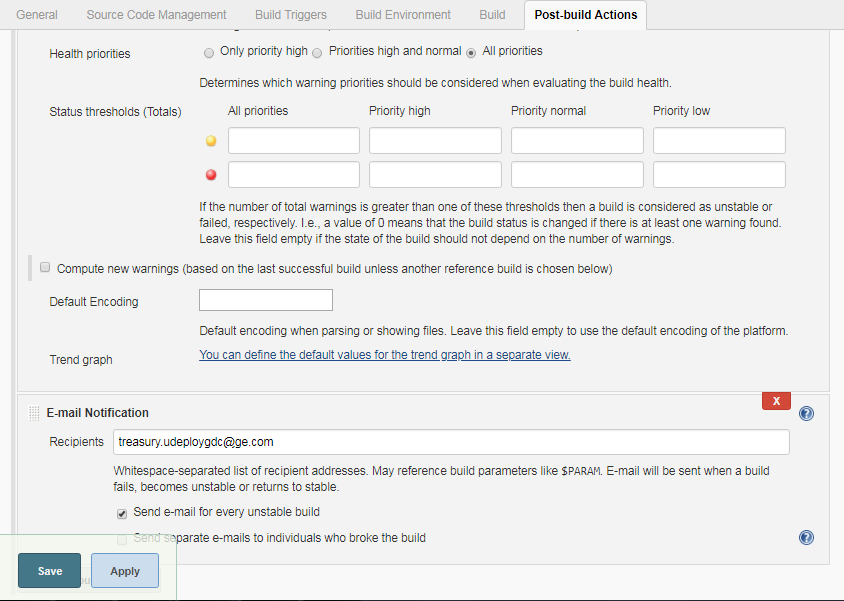


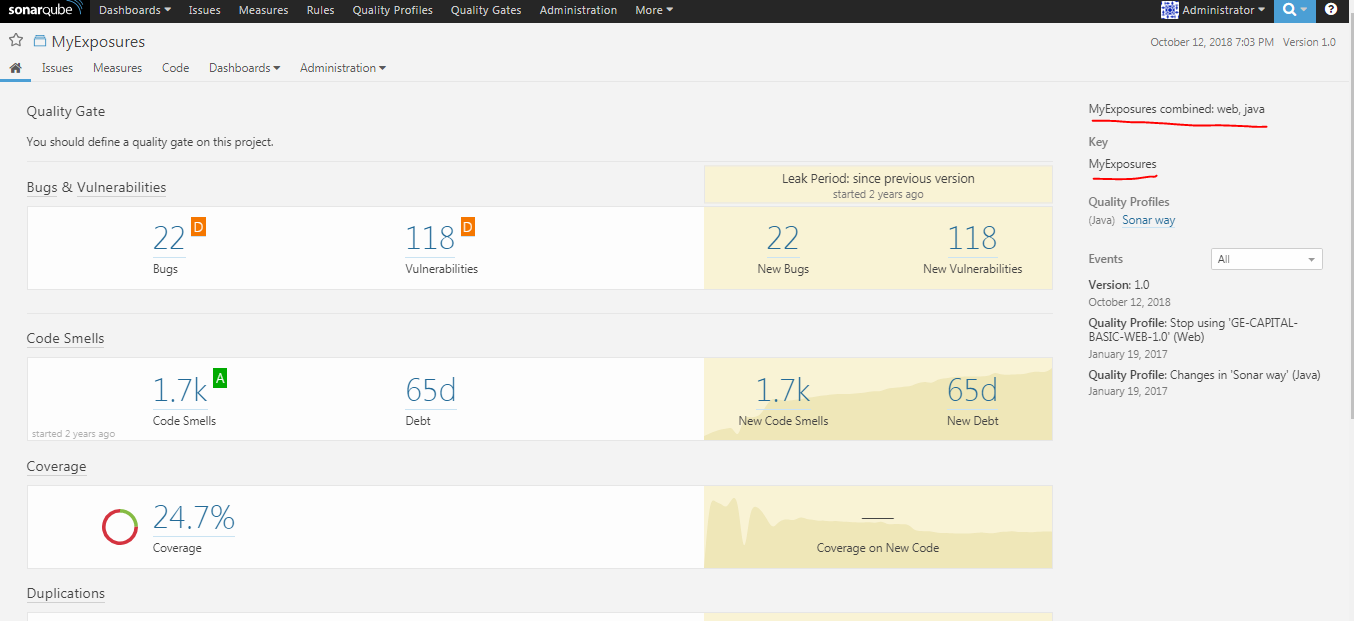




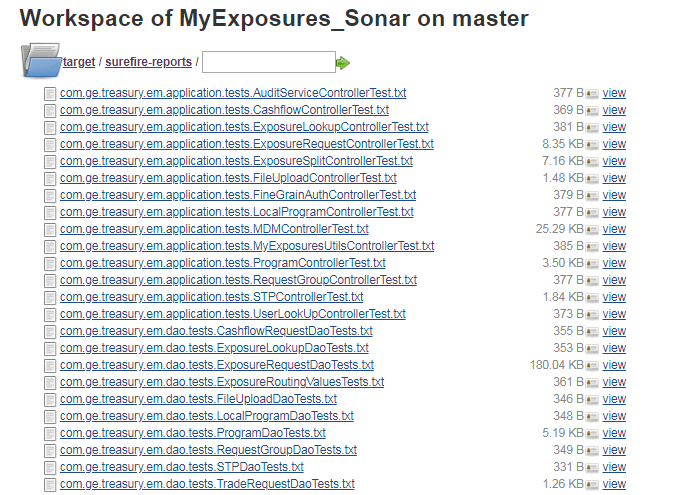


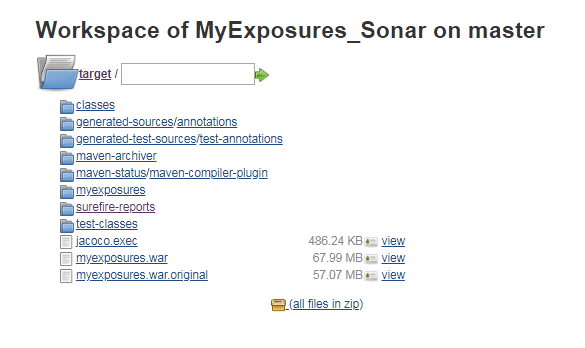


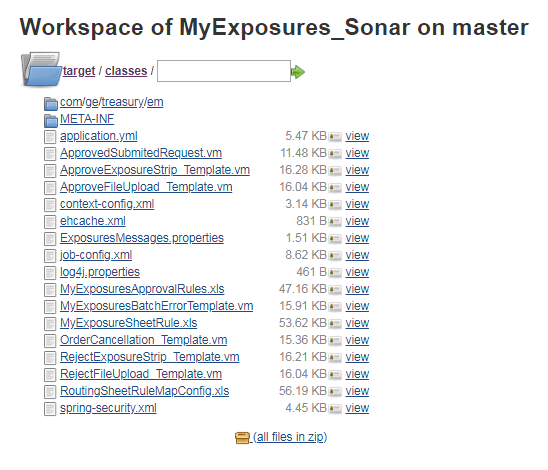




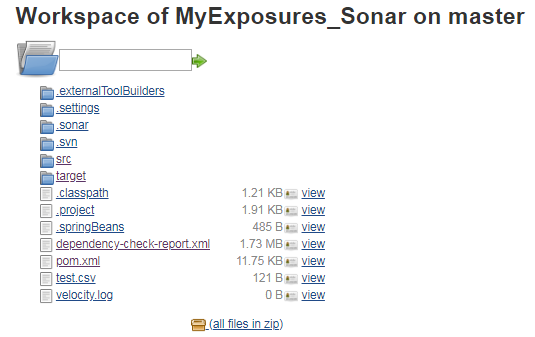
Project Workspace

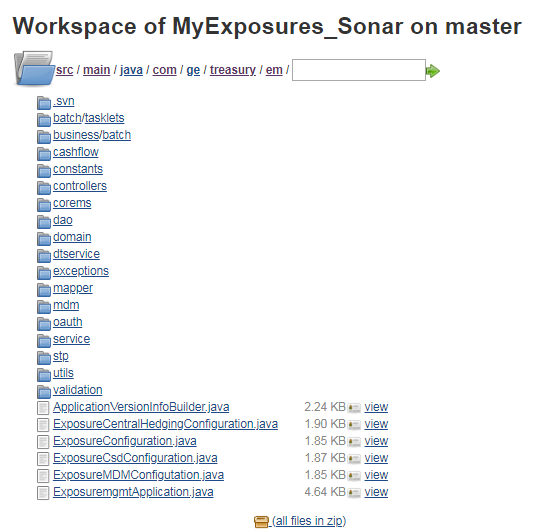












## **Using SonarQube in a Pipeline Project**

If we have or had an analysis job in our traditional pipeline, we can carry that idea forward to create an analysis stage in our Jenkins 2 pipeline. We just need to select the server, pass on the appropriate environment details, and call the scanner.

Fortunately, with SonarQube version 5.2 or greater, SonarQube Scanner version 2.8 or greater, and a recent version of the [SonarQube plugin](http://bit.ly/2HEe0NN) installed, the Jenkins pipeline DSL simplifies the process for us. It provides a withSonarQubeEnv block that allows us to select a globally configured SonarQube server to use. Further, it makes the connection details (associated with the global configuration for that server) available to operations done in that block. This simplifies providing an environment for a call to the scanner.

An example of using this block in a simple analysis stage would be:

stage('Analysis') {

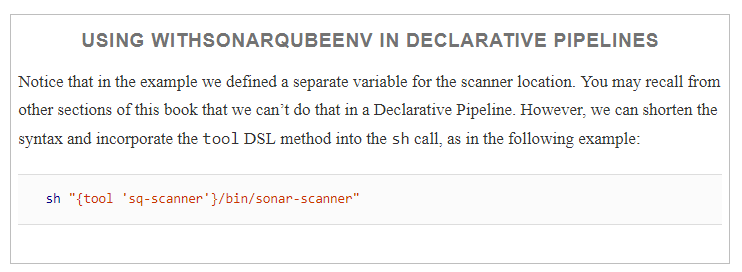
**def** scannerLoc = tool 'sq-scanner';

    withSonarQubeEnv('Local SonarQube') {

      sh "${scannerLoc}/bin/sonar-scanner"

    }

  }



With the use of the withSonarQubeEnv block, we can run the analysis and get the results back in the SonarQube application. However, that is not all we want to be able to do in the pipeline. We also want to have a way to use the results of the analysis to tell the pipeline whether the changes we are analyzing are of good enough quality to proceed to the next stage. How do we do that?

## **Leveraging the Outcome of the SonarQube Analysis**

One of the historical challenges with doing a SonarQube analysis as part of a pipeline process has been getting, and leveraging, the overall results of the analysis—that is, using the analysis results as a pass/fail indicator of whether to allow the code to proceed on to the next part of the pipeline.

Over the years, a number of solutions have been implemented and used. For example, one solution was a Groovy script that ran in a Jenkins job and accessed the SonarQube server via REST API calls. That custom script got the desired result values from SonarQube, and then evaluated them against thresholds entered as Jenkins job parameters to determine whether any were out of bounds. If any of the results were outside of the thresholds set by the parameters, the script would cause the Jenkins process to abort further processing.

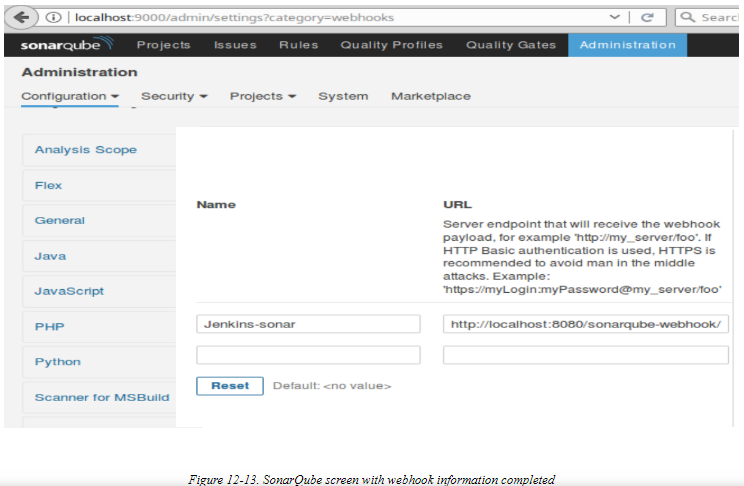
Better options are now available. For the Jenkins–SonarQube integration, we can set up a webhook in SonarQube and then have Jenkins wait for notification from that webhook before continuing. Let’s see how to do that.

### **SETTING UP THE SONARQUBE WEBHOOK**

To set up the SonarQube webhook, first sign in to the SonarQube application with administrator credentials. Then click on the Administration menu and select Configuration, then General Settings. From there, scroll down to the Webhooks section directly under that column. Click on that and fill in the fields as follows:

* **Name:** jenkins\_sonar
* **URL:** <jenkins-url>/sonarqube-webhook/ (don’t forget the trailing slash on the URL!)

Once you have filled in the fields, click the Save button to put the webhook is in place. [Figure 12-13](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch12.html#fig_SQ_screen_w_webhook_info_complete) shows the screen with the webhook information completed.



Now that we have the webhook set up, we are ready to set up the code to process it in the Jenkins pipeline

### **PROCESSING THE SONARQUBE WEBHOOK IN THE JENKINS DSL**

The SonarQube plugin provides a Jenkins DSL method to wait for the SonarQube webhook to process, named waitForQualityGate. This method pauses pipeline execution and waits for the previously submitted SonarQube analysis to complete, per the webhook notification from SonarQube. The method returns the status of the Quality Gate that was applied to the project in SonarQube. You can then check the return status to know whether the analysis was a pass or fail and whether it’s OK to proceed in the pipeline.

An implementation in a Scripted Pipeline might look like this:

**def** qg = waitForQualityGate();

**if** (qg.status != 'OK') {

    error "Pipeline aborted due to quality gate failure: ${qg.status}"

}

There is another important consideration around using this method. Any time you are using a method that pauses your pipeline, as this one does, you should consider the consequences if the method never gets the input, or the event that triggers it to continue. For example, in this case, what if your SonarQube server died or became inaccessible while this method was waiting? It is likely you would not want to hold up your entire pipeline until the problem was discovered and fixed.

A good approach to address this kind of potential issue is to surround the code with a DSL timeout block (as discussed in [Chapter 3](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#CH_Pipeline_Execution_Flow)). The syntax is straightforward. Here’s an example with the timeout value set to 5 minutes

timeout(time:5, unit:'MINUTES') {

**def** qg = waitForQualityGate()

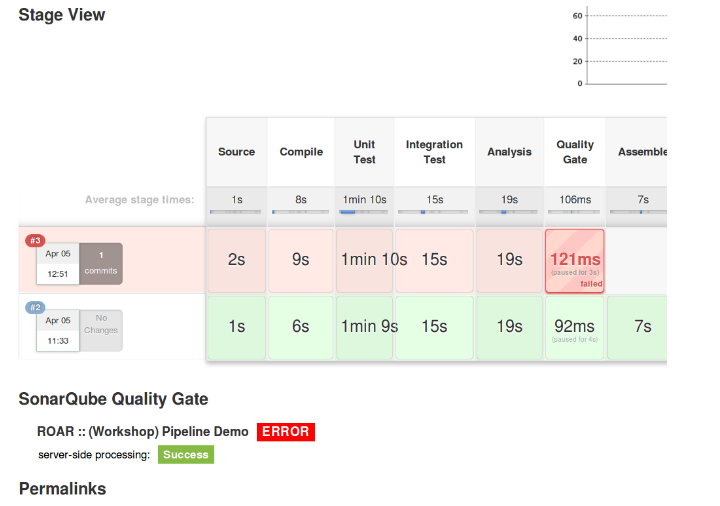
**if** (qg.status != 'OK') {

       error "Pipeline aborted due to quality gate failure: ${qg.status}"

    }

 }

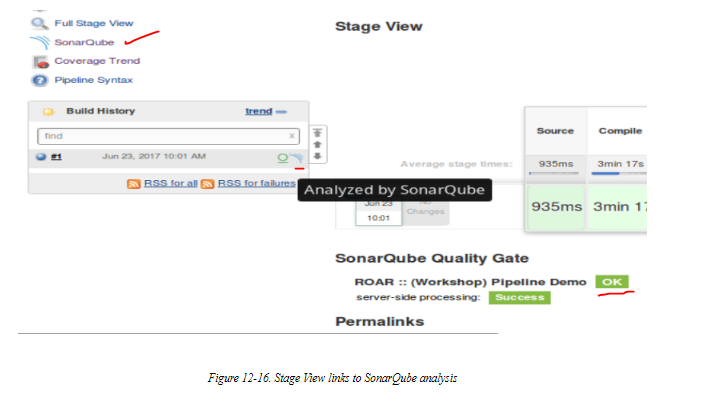
This code can be incorporated into the same analysis stage in the pipeline that we used earlier for the call to the scanner, or it can be put into its own separate stage. Having a separate stage to wait for the Quality Gate does allow you to easily determine (in an interface like the Jenkins Stage View) whether that particular piece failed, versus a failure in running the scanner. [Figure 12-14](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch12.html#fig_stage_view_failure_sep_stage_quality_gate) shows a representation of this.



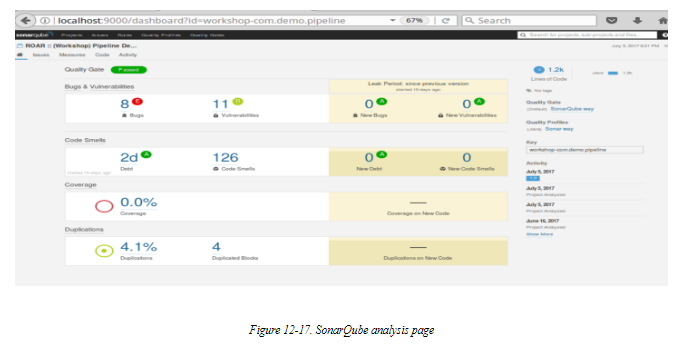
If this is not important to you, though, then it really just depends on how granular you want to make your pipeline stages.

## **SonarQube Integration Output with Jenkins**

SonarQube provides multiple ways to link to the analysis of a Jenkins project from within the Jenkins output itself. This is most noticeable in the Stage View of a pipeline or job. In [Figure 12-16](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch12.html#fig_stage_view_links_sonar_analysis), you can see several links to the SonarQube output for this project. In the lefthand menu, you see the item named SonarQube as one link. Notice the icon/symbol next to it with the three curved lines. That same symbol/badge shows up in the Build History area at the end of the line for the #1 run. Clicking on that badge there will take you to the same project analysis page in the SonarQube application. And finally, there is the OK button under the SonarQube Quality Gate label (after the project name). This is another link to the same location.



Clicking on any of these links takes you to the SonarQube analysis page for the project



# Code Coverage: Integration with JaCoCo

Typically, code coverage analysis is included with a tool like SonarQube. However, since it can be a significant factor on its own, here we’ll look at how to utilize one code coverage application, JaCoCo, separately in your pipeline. Even if you are not going to use JaCoCo, the example integration shown here may be useful to you as you incorporate other tools.

## **About JaCoCo**

The name JaCoCo is short for Java Code Coverage. As the name implies, the intended purpose of the tool is to provide code coverage information for Java source files—essentially, how much of your code your test cases are exercising. It does this by instrumenting the Java class files.

JaCoCo can provide information about a number of coverage aspects, including:

Instruction coverage

Basic info about how much code has been executed

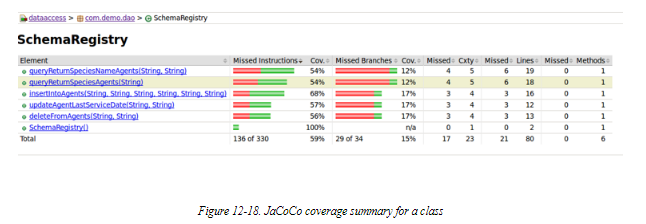
Branch coverage

For if and switch statements, will look at all the possible branches and figure out the number of executed and missed branches

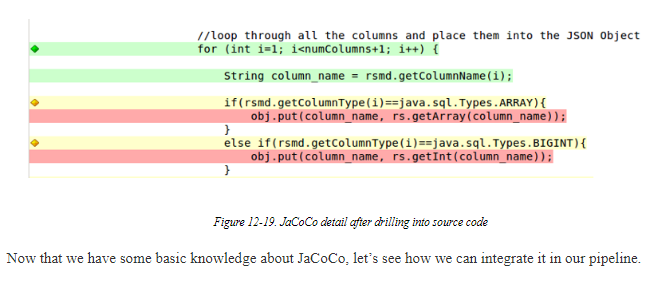
Cyclomatic complexity

Defined as the minimum number of paths that are capable of generating all possible paths through a method; basically, this can suggest the number of unit tests required to completely cover a piece of code

Let’s take a look at some example output. [Figure 12-18](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch12.html#fig_jacoco_coverage_summary_class) is a summary of missed instructions and missed branches in a class

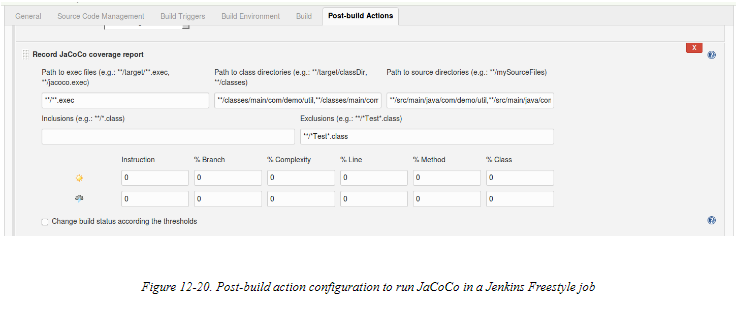


[Figure 12-19](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch12.html#fig_jacoco_detail_after_drilling_srccode) is a more detailed summary after drilling into the source code. Fully covered lines are represented as green, partially covered lines are colored yellow, and lines that haven’t been exercised yet are red. The diamonds here refer to decision points, and the colors have similar meanings as before: green means all branches executed, yellow means some branches executed, and red means none of the branches executed.



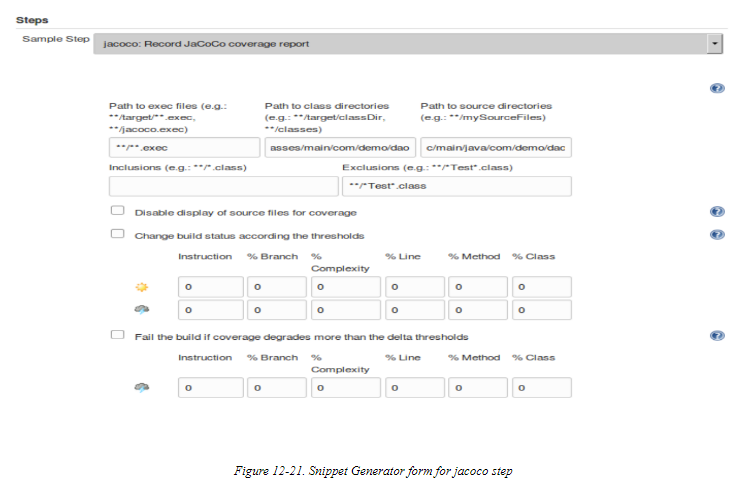
## **Integrating JaCoCo with the Pipeline**

To use JaCoCo, the application must be available and you must have the [JaCoCo plugin](https://plugins.jenkins.io/jacoco) installed in Jenkins. (This assumes that you are using JaCoCo separately from a code analysis application like SonarQube.) Unlike other applications, JaCoCo does not require any global configuration in Jenkins. Rather, it is made available as a post-build action in the traditional Jenkins model. [Figure 12-20](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch12.html#fig_PB_action_config_run_jacoco_Jenkfree) shows a job set up to run JaCoCo as a post-build action.



The fields in this section allow us to configure various aspects of the code coverage analysis. The paths define the locations of the various types of files JaCoCo needs access to. These are relative to the Jenkins workspace. The Inclusions and Exclusions fields allow excluding certain class files from instrumentation. (Recall that JaCoCo works by instrumenting class files.) And the bottom numeric fields allow for setting coverage thresholds. If the bottom checkbox is checked, this tells Jenkins to update the build status based on whether or not the threshold values were met.

We can translate this into code for our pipeline most easily using the Snippet Generator. In fact, as [Figure 12-21](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch12.html#fig_snip_gen_form_jacoco_step)shows, the form to fill in for the “jacoco” step using the Snippet Generator looks remarkably like the form from the traditional Jenkins job.



Filling in the form to match our traditional Jenkins job’s JaCoCo configuration and pressing the button to generate the Groovy script code yields the following pipeline code:

jacoco classPattern: '\*\*/classes/main/com/demo/util,

\*\*/classes/main/com/demo/dao', exclusionPattern: '\*\*/\*Test\*.class',

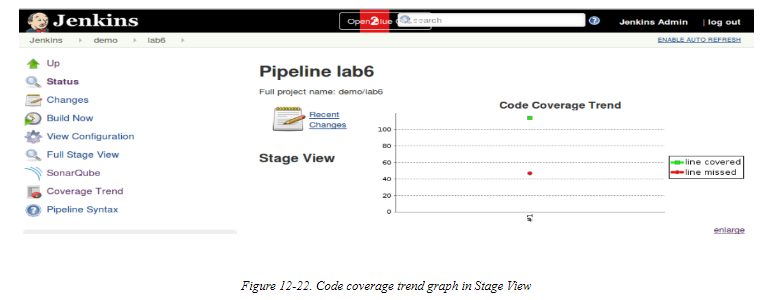
sourcePattern: '\*\*/src/main/java/com/demo/util,

\*\*/src/main/java/com/demo/dao'

This can then be placed in a stage block in the pipeline. Most commonly, it might just be added to the stage for the analysis functions.

## **JaCoCo Output Integration with Jenkins**

Finally, let’s take a quick look at how JaCoCo integrates its output with Jenkins. Once you have successfully run through an analysis with JaCoCo, Jenkins will add two things to the output page (Stage View) for the job. The first will be a large graph showing code coverage trends over time. The second will be an additional Coverage Trend menu item in the lefthand menu that, when clicked, will bring up a similar code coverage trend graph. (See [Figure 12-22](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch12.html#fig_jacoco_output_integration).)



Clicking on either graph will allow you to drill down further into the code coverage details, by packages and then eventually into files and methods.

Figures [12-23](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch12.html#fig_drill_jacoco_int_Jenkout), [12-24](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch12.html#fig_drill_as_above_more), and [12-25](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch12.html#fig_drill_jacoco_some_more), show some examples of further drilling down.

