**Jenkins and its integration with Nexus and Sonar**

Continuous Integration, also known as CI, is a cornerstone of modern software development. In fact it is a real game changer—when Continuous Integration is introduced into an organization, it radically alters the way teams think about the whole development process. It has the potential to enable and trigger a series of incremental process improvements, going from a simple scheduled automated build right through to continuous delivery into production. A good CI infrastructure can streamline the development process right through to deployment, help detect and fix bugs faster, provide a useful project dashboard for both developers and non-developers, and ultimately, help teams deliver more real business value to the end user. Every professional development team, no matter how small, should be practicing CI.

* 1. **Preparing Your Environment**

We will be using Jenkins to build a Java application using Maven. Maven is a widely-used build tool in the Java world, with many powerful features such as declarative dependency management, convention over configuration, and a large range of plugins. For our build, we will also be using recent versions of the Java Development Kit (JDK) and Maven.

**1.2.1 Installing Java**

The first thing you will need to install on your machine is Java. Jenkins is a Java web application, so you will need at least the Java Runtime Environment, or JRE to run it.

**1.2.2. Installing Git**

Since we will be using Git, you will need to install and configure Git on your machine. First of all, you need to install Git on your machine. This involves downloading the appropriate installer for your operating system from the Git website.

* 1. **Starting Up Jenkins**
* There are installers available for the principal operating systems available on the Jenkins website.Or, if you are an experienced Java user versed in the ways of WAR files, you may prefer to simply download the latest version of Jenkins and run it from the command line. Jenkins comes in the form of an executable WAR file—you can download the most recent version from the Jenkins website home page. Once downloaded, you can start Jenkins from the command line as shown here:

$ **java -jar jenkins.war**

Whether you have started Jenkins using Java Web Start or from the command line, Jenkins should now be running on your local machine. By default, Jenkins will be running on port 8080, so you can access Jenkins in your web browser on

<http://localhost:8080>.

* Alternatively, if you are familiar with Java application servers such as Tomcat, you can simply deploy the Jenkins WAR file to your application server—with Tomcat, for example, you could simply place the jenkins.war file in Tomcat’s webapps directory. If you are running Jenkins on an application server, the URL that you use to access Jenkins will be slightly different. On a default Tomcat installation, for example, you can access Jenkins in your web browser on

<http://localhost:8080/jenkins>.

When you open Jenkins in your browser, you should see a screen like the one shown in Figure 1.1

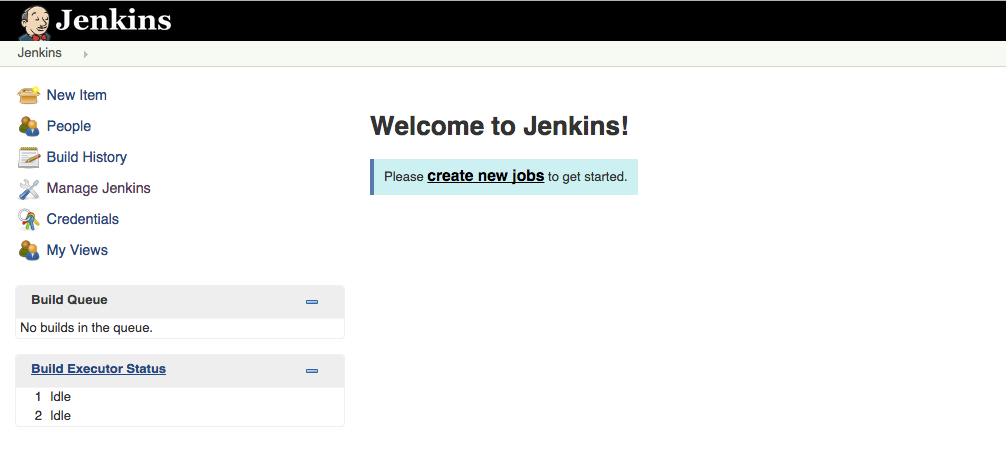


Figure 1.1. The Jenkins start page

* 1. **Configuring the Tools**

Before we get started, we do need to do a little configuration. More precisely, we need to tell Jenkins about the build tools and JDK versions we will be using for our builds. Click on the Manage Jenkins link on the home page (see Figure 1.1, “The Jenkins start page”). This will take you to the Manage Jenkins page, the central one-stop-shop for all your Jenkins configuration. From this screen, you can configure your Jenkins server, install and upgrade plugins, keep track of system load, manage distributed build servers, and more! For now, however, we’ll keep it simple. Just click on the Configuring System link at the top of the list (see Figure 1.2, “The Manage Jenkins screen”).

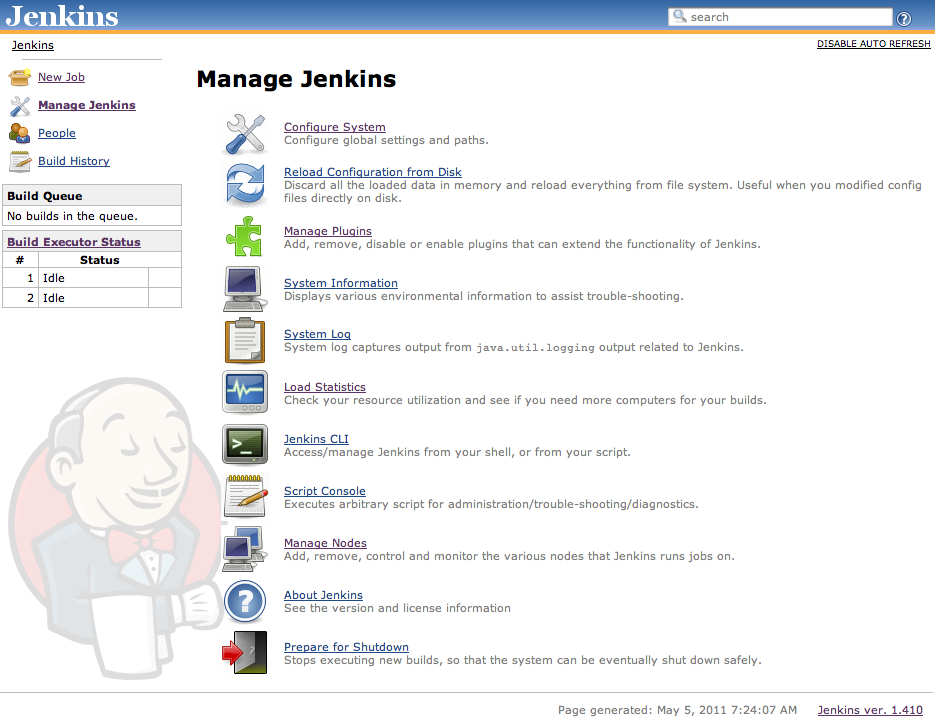


Figure 1.2. The Manage Jenkins screen

This will take you to Jenkins’s main configuration screen (see Figure 1.3, “The Configure Jenkins screen”). From here you can configure everything from security configuration and build tools to email servers, version control systems and integration with third-party software. The screen contains a lot of information, but most of the fields contain sensible default values.

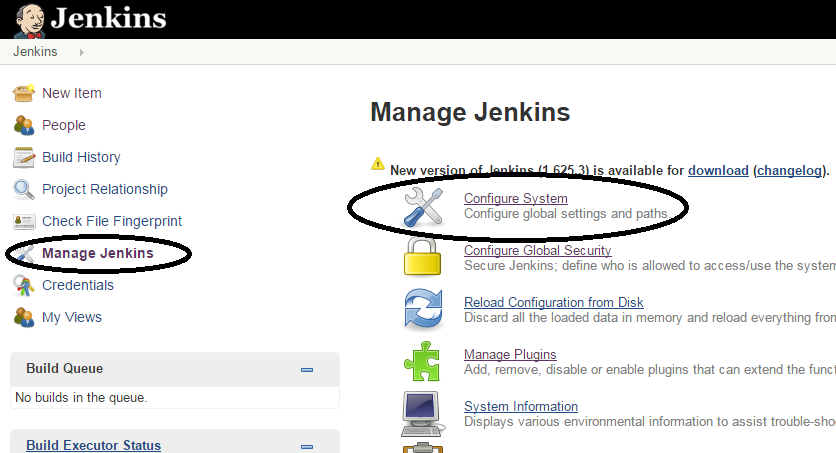


Figure 1.3, Configure Jenkins screen

**1.4.1 Configuring Your Maven Setup**

Jenkins provides several options when it comes to configuring Maven. If you already have Maven installed on your machine, you can simply provide the path in the MAVEN\_HOME field. Alternatively, you can install a Maven distribution by extracting a zip file located in a shared directory, or execute a home-rolled installation script. Or you can let Jenkins do all the hard work and download Maven for you. To choose this option, just tick the Install automatically checkbox. Jenkins will download and install Maven from the Apache website the first time a build job needs it. Just choose the Maven version you want to install and Jenkins will do the rest. You will also need to give a name for your Maven version.

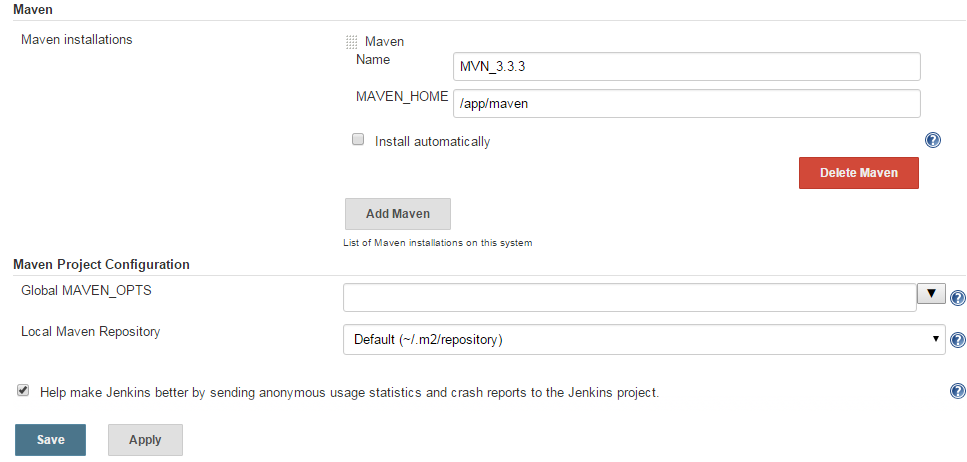


Figure 1.4. Configuring a Maven installation

**1.4.2. Configuring the JDK**

Once you have configured your Maven installation, you will also need to configure a JDK installation (see Figure 1.5, “Configuring a JDK installation”). Again, if you have a Java JDK (as opposed to a Java

Runtime Environment—the JDK contains extra development tools such as the Java compiler) already installed on your workstation, you can simply provide the path to your JDK in the JAVA\_HOME field. Otherwise, you can ask Jenkins to download the JDK from the Oracle website15 the first time a build job requires it.This is similar to the automatic Maven installation feature—just pick the JDK version you need and Jenkins will take care of all the logistics. However, for licensing reasons, you will also need to tick a checkbox to indicate that you agree with the Java SDK License Agreement.

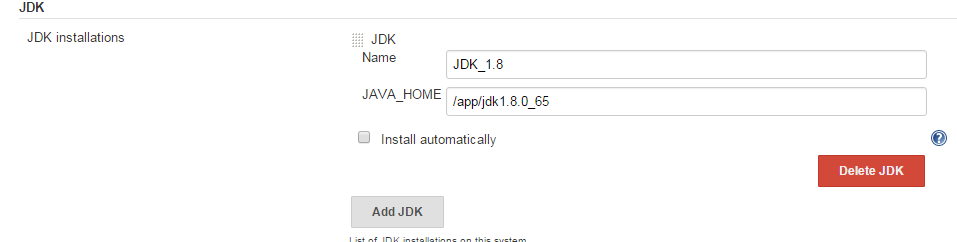


Figure 1.5. Configuring a JDK installation

**1.4.3. Notification**

Another important aspect you would typically set up is notification. When a Jenkins build breaks, and when it works again, it can send out email messages to the team to spread the word. Using plugins, you can also get it to send instant messages or SMS messages, post entries on Twitter, or get people notified in a few other ways. It all depends on what works best for your organizational culture. Email notification is easy enough to set up if you know your local SMTP server address—just provide this value in the

Email Notification section towards the bottom of the main configuration page.

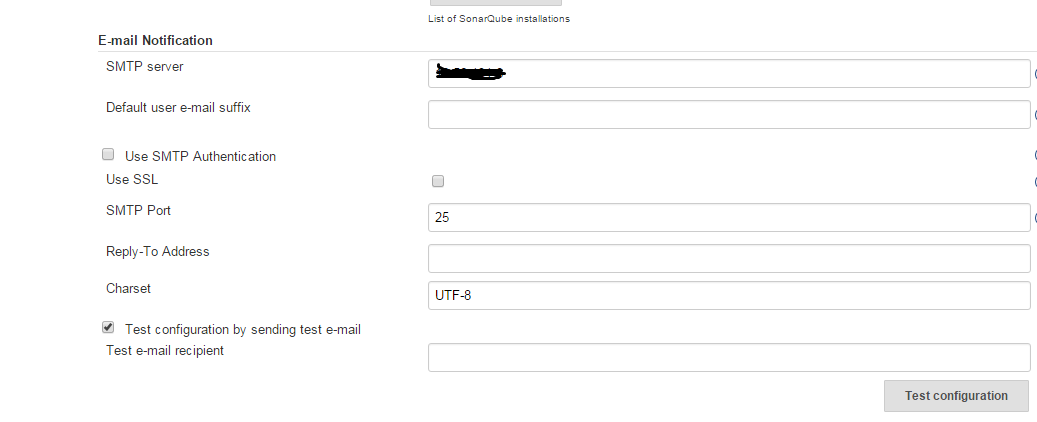


Figure 1.6. Setting up Email Notification

**1.4.4. Setting Up Git**

The last thing we need to configure for this is to get Jenkins working with Git. Jenkins comes with support for Subversion and CVS out of the box, but you will need to install the Jenkins Git plugin to be able use GIT Sourcecode

Manage Jenkins link to the left of the screen to go back to the main configuration screen (see Figure 1.2,“The Manage Jenkins screen”). Then click on Manage Plugins. This will open the plugin configuration screen, which is where you manage the extra features you want to install on your Jenkins server.

**2.1 Setting Up Your Build Jobs**

Build jobs are the basic currency of a Continuous Integration server.

A build job is a particular way of compiling, testing, packaging, deploying or otherwise doing something with your project. Build jobs come in a variety of forms; you may want to compile and unit test your application, report on code quality metrics related to the source code, generate documentation, bundle up an application for a release, deploy it to production, run an automated smoke test, or do any number of other similar tasks.

**2.2. Jenkins Build Jobs**

Creating a new build job in Jenkins is simple: just click on the “New Job” menu item on the Jenkins dashboard. Jenkins supports several different types of build jobs, which are presented to you when you choose to create a new job (see Figure 2.1, “Jenkins supports four main types of build jobs”).

**Freestyle software project**

Freestyle build jobs are general-purpose build jobs, which provide a maximum of flexibility.

**Maven project**

The “maven2/3 project” is a build job specially adapted to Maven projects. Jenkins understands Maven pom files and project structures, and can use the information gleaned from the pom file to reduce the work you need to do to set up your project.

**Monitor an external job**

The “Monitor an external job” build job lets you keep an eye on non-interactive processes, such as cron jobs.

**Multi configuration job**

The “multi configuration project” (also referred to as a “matrix project”) lets you run the same build job in many different configurations

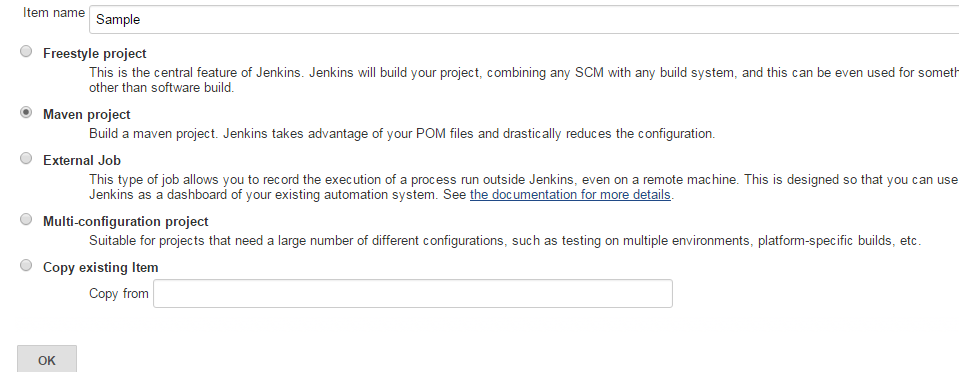


Figure 2.1, Jenkins supports four main types of build jobs

**2.2. Creating a new build job**

you create a new Maven job contains general information about the project, such as a unique name and description, and other information about how and where the build job should be executed (see Figure 2.2, “Creating a new build job”).

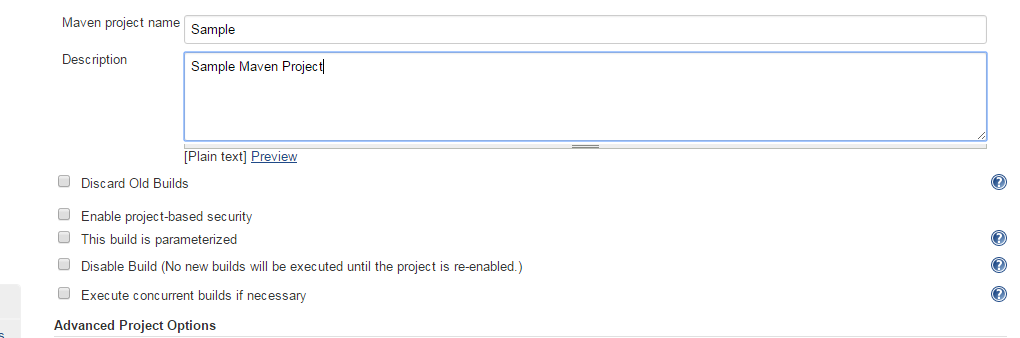


Figure 2.2. Creating a new build job

The project name can be anything you like, but it is worth noting that it will be used for the project directory and the build job URL, so I generally avoid names with spaces. The project description will go on the project home page—use this to provide an overview of the build job’s goals and context. HTML tags will work fine in this field.

In addition, Jenkins will never delete the last stable and successful builds, no matter how old they are.

For example, if you limit Jenkins to only keep the last twenty builds, and your last successful build was thirty builds ago, Jenkins will still keep the successful build job as well as the last twenty failing builds. You also have the option to disable the build. A disabled build will not be executed until you enable it again. Using this option when you create a new build job is quite rare. On the other hand, this option often comes in handy to temporarily suspend a build during maintenance work or major refactoring, when notification of the build failures will not be useful for the team.

**2.3. Configuring Source Code Management**

In its most basic role, a Continuous Integration server monitors your version control system, and checks out the latest changes as they occur. The server then compiles and tests the most recent version of the code. Alternatively, it may simply check out and build the latest version of your source code on a regular basis. In either case, tight integration with your version control system is essential. Because of its fundamental role, SCM configuration options in Jenkins are identical across all sorts of build jobs. Jenkins supports CVS and Subversion out of the box, with built-in support for Git, and also integrates with a large number of other version control systems via plugins. At the time of writing, SCM plugin support includes Accurev, Bazaar, BitKeeper, ClearCase, CMVC, Dimensions, Git, CA Harvest, Mercurial, Perforce, PVCS, StarTeam, CM/Synergy, Microsoft Team Foundation Server, and

even Visual SourceSafe. In the rest of this section, we will look at how to configure some of the more common SCM tools.

**2.3.1. Working with Subversion**

Subversion is one of the most widely used version control systems, and Jenkins comes bundled with full Subversion support (see Figure 2.3.1, “Jenkins provides built-in support for Subversion”). To use source code from a Subversion repository, you simply provide the corresponding Subversion URL—it will work fine with any of the three Subversion protocols of (http, svn, or file). Jenkins will check that the URL is valid as soon as you enter it. If the repository requires authentication, Jenkins will prompt you for the corresponding credentials automatically, and store them for any other build jobs that access this repository.

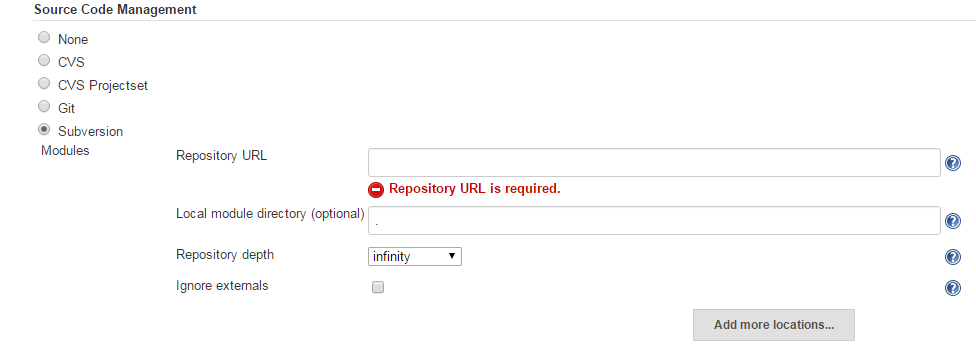


Figure 2.3.1. Jenkins provides built-in support for Subversion

By default, Jenkins will check out the repository contents into a subdirectory of your workspace, whose name will match the last element in the Subversion URL. So if your Subversion URL is svn://localhost/gameoflife/trunk, Jenkins will check out the repository contents to a directory called trunk in the build job workspace. If you would prefer another directory name, just enter the directory name you want in the **Local module directory** field. Place a period (“.”) here if you want Jenkins to check the source code directly into the workspace.

Occasionally you may need to get source code from more than one Subversion URL. In this case, just use the “Add more locations...” button to add as many additional repository sources as you need. A well-designed build process should not modify the source code, or leave any extra files that might confuse your version control system or the build process. Both generated artifacts and temporary files (such as log files, reports, test data or file-based databases) should go in a directory set aside for this purpose (such as the target directory in Maven builds), and/or be configured to be ignored by your version control repository. They should also be deleted as part of the build process, once the build has finished with them. This is also an important part of ensuring a clean and reproducible build process

—for a given version of your source code, your build should behave in exactly the same way, no matter where or when it is run. Locally changed source code files, and the presence of temporary files, both have the potential of compromising this.

You can fine-tune the way Jenkins obtains the latest source code from your Subversion repository by selecting an appropriate value in the Check-out Strategy drop-down list. If your project is well-behaved, however, you may be able to speed things up substantially by selecting “Use ‘svn update’ as much as possible”. This is the fastest option, but may leave artifacts and files from previous builds in your workspace. To be on the safe side, you may want to use the second option (“Use ‘svn update’ as much as possible, with ‘svn revert’ before update”), which will systematically run svn revert before running svn update. This will ensure that no local files have been modified, though it will not remove any new files that have been created during the build process. Alternatively, you can ask Jenkins to delete any unversioned or ignored files before performing an svn update, or play it safe by checking out a full clean copy for each build.

**2.3.2. Working with Git**

Git is a popular distributed version control system that is a logical successor to Subversion and a mind-share competitor to Mercurial. Git support in Jenkins is both mature and full-featured. There are a number of plugins that can contribute to the overall story of Git in Jenkins. We will begin by looking at the Git plugin, which provides core Git support in Jenkins.

**2.3.2.1. Installing the plugin**

Already discussed in section 1.4.4

**2.3.2.2. SSH key setup**

If the Git repository you are accessing uses SSH passphrase-less authentication—for example, if the access address is similar to [git@github.com:matthewmccullough/some-repo.git—](mailto:git@github.com:matthewmccullough/some-repo.git—)

you’ll need to provide the private half of the key as file ~/.ssh/id\_rsa where ~ is the home directory of the user account under which Jenkins is running.

The fingerprint of the remote server will additionally need to be placed in ~/.ssh/known\_hosts to prevent Jenkins from invisibly prompting for authorization to access this Git server for the first time.

Alternatively, if logging-in is enabled for the jenkins user, SSH into the Jenkins machine as Jenkins and manually attempt to Git clone a remote repository. This will test your private key setup and establish

the known\_hosts file in the ~/.ssh directory. This is probably the simplest option for users unfamiliar with the intricacies of SSH configuration.

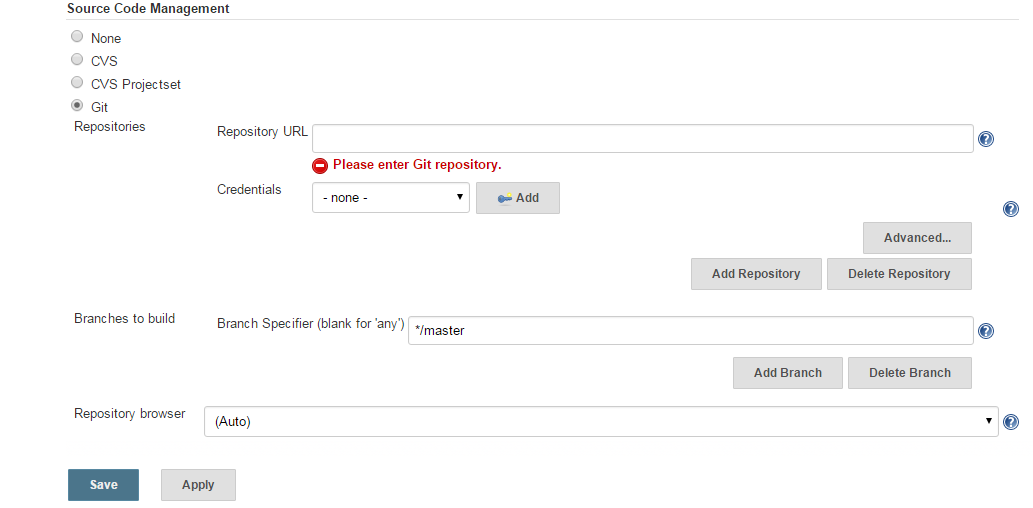


Figure 2.3.2. GIT source code management

**2.4. Build Triggers**

Once you have configured your version control system, you need to tell Jenkins when to kick off a build.

You set this up in the Build Triggers section.

In a Freestyle build, there are three basic ways a build job can be triggered (see Figure 2.4, “There are many ways that you can configure Jenkins to start a build job”):

• Start a build job once another build job has completed

• Kick off builds at periodical intervals

• Poll the SCM for changes

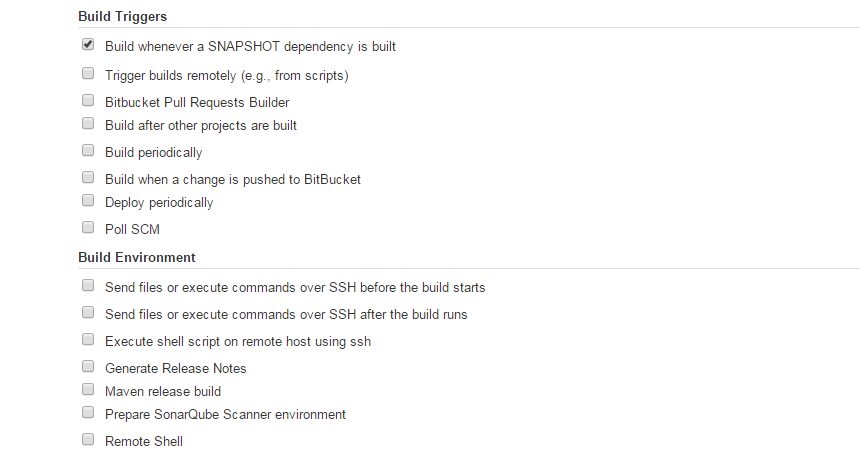


Figure 2.4. There are many ways that you can configure Jenkins to start a build job

**2.4.1. Triggering a Build Job Once Another Build Job Has Finished**

The first option lets you set up a build that will be run whenever another build has finished. This is an easy way to set up a build pipeline. For example, you might set up an initial build job to run unit and integration tests, followed by another separate build job to run more CPU-intensive code quality metrics.

You simply enter the name of the preceding build job in this field. If the build job can be triggered by several other build jobs, just list their names here, separated by commas. In this case, the build job will be triggered once any of the build jobs in the list finish.

**2.4.2. Scheduled Build Jobs**

Another strategy is simply to trigger your build job at regular intervals. It is important to note that this is not actually Continuous Integration—it is simply scheduled builds, something you could also do, for example, as a Unix cron job. In the early days of automated builds, and even today in many shops, builds are not run in response to changes committed to version control, but simply on a nightly basis.

However, to be effective, a Continuous Integration server should provide feedback much more quickly than once a day.

There are nevertheless a few cases where scheduled builds do make sense. This includes very long running build jobs, where quick feedback is less critical. For example, intensive load and performance tests which may take several hours to run or Sonar build jobs. Sonar is an excellent way to keep tabs on code quality metrics across your projects and over time, but the Sonar server only stores one set of data per day, so running Sonar builds more frequently than this is not useful. For all scheduling tasks, Jenkins uses a cron-style syntax, consisting of five fields separated by white space in the following format:

MINUTE HOUR DOM MONTH DOW

with the following values possible for each field:

102

MINUTE

Minutes within the hour (0–59)

HOUR

The hour of the day (0–23) DOM

DOM

The day of the month (1–31)

MONTH

The month (1–12)

DOW

The day of the week (0–7) where 0 and 7 are Sunday.

There are also a few short-cuts:

• “\*” represents all possible values for a field. For example, “\* \* \* \* \*” means “once a minute.”

• You can define ranges using the “M–N” notation. For example “1-5” in the DOW field would mean “Monday to Friday.”

• You can use the slash notation to defined skips through a range. For example, “\*/5” in the MINUTE field would mean “every five minutes.”

• A comma-separated list indicates a list of valid values. For example, “15,45” in the MINUTE field would mean “at 15 and 45 minutes past every hour.”

• You can also use the shorthand values of “@yearly”, “@annually”, “@monthly”, “@weekly”, “@daily”, “@midnight”, and “@hourly”.

Typically, you will only have one line in this field, but for more complicated scheduling setups, you may need multiple lines.

**2.4.3. Polling the SCM**

As we have seen, scheduled build jobs are usually not the best strategy for most CI build jobs. The value of any feedback is proportional to the speed in which you receive that feedback, and Continuous

Integration is no exception. That is why polling the SCM is generally a better option. Polling involves asking the version control server at regular intervals if any changes have been committed. If any changes have been made to the source code in the project, Jenkins kicks off a build.

Polling is usually a relatively cheap operation, so you can poll frequently to ensure that a build kicks off rapidly after changes have been committed. The more frequent the polling is, the faster the build jobs will start, and the more accurate the feedback about what change broke the build will be.

In Jenkins, SCM polling is easy to configure, and uses the same cron syntax we discussed previously.The natural temptation for SCM polling is to poll as often as possible (for example, using “\* \* \* \*

\*”, or once every minute). Since Jenkins simply queries the version control system, and only kicks off a build if the source code has been modified, this approach is often reasonable for small projects.

**2.5. Build Steps**

Now Jenkins should know where and how often to obtain the project source code. The next thing youneed to explain to Jenkins is what it what to do with the source code. In a freestyle build, you do this by defining build steps. Build steps are the basic building blocks for the Jenkins freestyle build process.

They are what let you tell Jenkins exactly how you want your project built.

A build job may have one step, or more. It may even occasionally have none. In a freestyle build, you can add as many build steps as you want to the Build section of your project configuration (see Figure 2.5,

“Adding a build step to a build job”). In a basic Jenkins installation, you will be able to add steps to invoke Maven and Ant, as well as running OS-specific shell or Windows batch commands. And by installing additional plugins, you can also integrate other build tools, such as Groovy, Gradle, Grails,Jython, MSBuild, Phing, Python, Rake, and Ruby, just to name some of the more well-known tools.

**2.5.1. Maven Build Steps**

Jenkins has excellent Maven support, and Maven build steps are easy to configure and very flexible.

Just pick “Invoke top-level Maven targets” from the build step lists, pick a version of Maven to run(if you have multiple versions installed), and enter the Maven goals you want to run. Jenkins freestyle build jobs work fine with both Maven 2 and Maven 3.

Just like on the command line, you can specify as many individual goals as you want. You can also provide command-line options. A few useful Maven options in a CI context are:

-B, --batch-mode

This option tells Maven not to prompt for any input from the user, just using the default values if any are required. If Maven does prompt for any input during the Jenkins build, the build will

get stuck indefinitely.

-U, --update-snapshots

Forces Maven to check for updated releases and snapshot dependencies on the remote repository. This makes sure you are building with the latest and greatest snapshot dependencies, and not just

using older local copies which may not by in sync with the latest version of the source code.

-Dsurefire.useFile=false

This option forces Maven to write JUnit output to the console, rather than to text files in the target directory as it normally would. This way, any test failure details are directly visible in the build job console output. The XML files that Jenkins needs for its test reporting will still be generated.

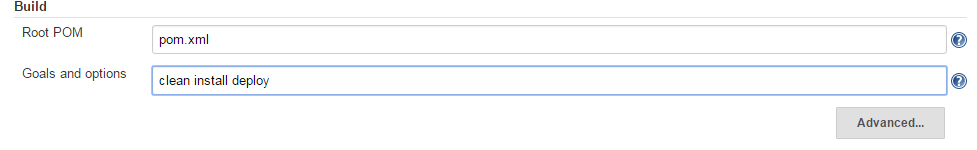


Figure 2.5. Adding a build step to a build job

**2.5.3. Executing a Shell or Windows Batch Command**

Occasionally you may need to execute a command directly at the Operating System level. Some legacy build processes rely on OS-specific scripts, for example. In other cases, you may need to perform a lowleveloperation that is most easily done with an OS-level command.

You can do this in Jenkins with the Execute Shell (for Unix) or Execute Windows Batch command (for Windows).

**2.5.4. Using Jenkins Environment Variables in Your Builds**

One useful trick that can be used in virtually any build step is to obtain information from Jenkins about

the current build job. In fact, when Jenkins starts a build step, it makes the following environment

variables available to the build script:

BUILD\_NUMBER

The current build number, such as “153”.

BUILD\_ID

A timestamp for the current build id, in the format YYYY-MM-DD\_hh-mm-ss.

JOB\_NAME

The name of the job, such as game-of-life.

BUILD\_TAG

A convenient way to identify the current build job, in the form of jenkins-${*JOB\_NAME*}-${*BUILD\_NUMBER*} (e.g., jenkins-game-of-life-2010-10-30\_23-59-59).

EXECUTOR\_NUMBER

A number identifying the executor running this build among the executors of the same machine. This is the number you see in the “build executor status”, except that the number starts from 0, not 1.

NODE\_NAME

The name of the slave if the build is running on a slave, or "" if the build is running on master.

NODE\_LABELS

The list of labels associated with the node that this build is running on.

JAVA\_HOME

If your job is configured to use a specific JDK, this variable is set to the JAVA\_HOME of the specified JDK. When this variable is set, PATH is also updated to have $JAVA\_HOME/bin.

WORKSPACE

The absolute path of the workspace.

HUDSON\_URL

The full URL of the Jenkins server, for example <http://ci.acme.com:8080/jenkins/>.

JOB\_URL

The full URL for this build job

BUILD\_URL

The full URL for this build, for example http://ci.acme.com:8080/jenkins/game-oflife/

SVN\_REVISION

For Subversion-based projects, this variable contains the current revision number.

**2.6. Post-Build Actions**

Once the build is completed, there are still a few things you need to look after. You might want to archive some of the generated artifacts, to report on test results, and to notify people about the results. In this section, we look at some of the more common tasks you need to configure after the build is done.

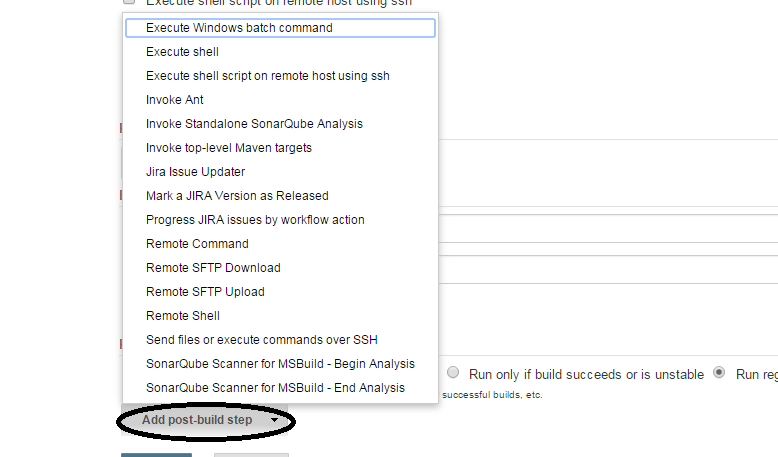


Figure 2.6. Adding a post build step

**2.7. Running Your New Build Job**

Now all you need to do is save your new build job. You can then trigger the first build manually, or just wait for it to kick off by itself. Once the build is finished, you can click on the build number to see the results of your work.

**2.8. Working with Maven Build Jobs**

In this section, we will have a look at the other most commonly used build job: Maven 2/3 build jobs.

Maven build jobs are specifically adapted to Maven 2 and Maven 3 builds. Creating a Maven build job requires considerably less work than configuring the equivalent freestyle build job. Maven build jobs support advanced Maven-related features such as incremental builds on multi module projects and triggering builds from changes in snapshot dependencies, and make configuration and reporting much simpler.

**2.8.1. Building Whenever a SNAPSHOT Dependency Is Built**

At first glance, the Maven 2/3 build job configuration screen is very similar to the freestyle builds in the previous section. The first difference you may notice is in the Build Triggers section. In this section, an extra option is available: “**Build whenever a SNAPSHOT dependency is built**”. If you select this option, Jenkins will examine your pom.xml file (or files) to see if any

SNAPSHOT dependencies are being built by other build jobs. If any other build jobs update a SNAPSHOT dependency that your project uses, Jenkins will build your project as well.

Typically in Maven, SNAPSHOT dependencies are used to share the latest bleeding-edge version of a library with other projects within the same team. Since they are by definition unstable, it is not recommended practice to rely on SNAPSHOT dependencies from other teams or from external sources.

**2.8.2. Configuring the Maven Build**

The next area where you will notice a change is in the Build section. In a Maven build job, the build section is entirely devoted to running a single Maven goal.

In this section, you specify the version of Maven you want to execute (remember, at the time of Maven, this will only work with Maven), the location of the pom.xml file, and the Maven goal (or goals) to invoke. You can also add any command-line options you need here.

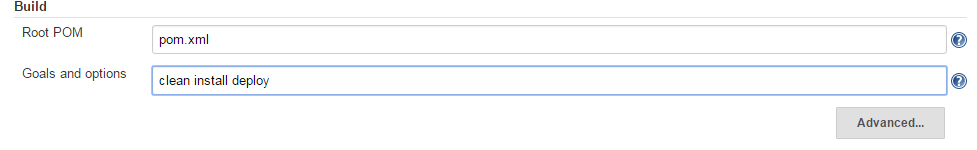


Figure 2.8. Specifying the Maven goals

In many cases, this is all you need to get your Maven build job configured. However, if you click on the “Advanced...” button, you can take your pick of some more advanced features

The Incremental Build option comes in very handy for large, multimodule Maven builds. If you tick this option, when a change is made to one of the project modules, Jenkins will only rebuild that module and any modules that use the changed module. It performs this magic by using some new Maven features introduced in Maven 2.1 (so it won’t work if you are using Maven 2.0.x). Jenkins detects which modules have been changed, and then uses the -pl (--project-list) option to build only the updated modules, and the -amd (--also-make-dependents) option to build the modules that use the updated modules.

If nothing has been changed in the source code, all of the modules are built.

By default, Jenkins will archive all of the artifacts generated by a Maven build job. This can come in handy at times, but it can also be very expensive in disk storage. If you want to turn off this option, just tick the “Disable automatic artifact archiving” option. Alternatively, you can always limit the artifacts stored by using the “Discard Old Builds” option at the top of the configuration page.

Another useful option is “Use [a] private Maven repository”. Normally, when Jenkins runs Maven, it will behave in exactly the same way as Maven on the command line: it will store artifacts in, and retrieve artifacts from the local Maven repository (found in ~/.m2/repository if you haven’t reconfigured it in the settings.xml file). This is efficient in terms of disk space, but not always ideal for CI builds.

Indeed, if several build jobs are working on and with the same snapshot artifacts, the builds may end up interfering with each other.

When this option is checked, Jenkins will tell Maven to use $WORKSPACE/.repository as the local Maven repository. This means each job will get its own isolated Maven repository just for itself. It fixes the above problems, at the expense of additional disk space consumption.

With this option, Maven will use a dedicated Maven repository for this build job, located in the $WORKSPACE/.repository directory. This takes more disk space, but guarantees a better isolation between build jobs.

Another way of addressing this problem is to override the default repository location by using the maven.repo.local property, as shown here:

$ **mvn install -Dmaven.repo.local=~/.m2/staging-repository**

This approach has the advantage of being able to share a repository across several build jobs, which is useful if you need to do a series of related builds. It will also work with freestyle jobs.

**2.8.3. Post-Build Actions**

The Post-Build actions in a Maven build job are considerably simpler to configure than in a freestyle job. This is simply because, since this is a Maven build, Jenkins knows where to look for a lot of the build output. Artifacts, test reports, Javadoc, and so forth, are all generated in standard directories, which means you don’t have to tell Jenkins where to find things. So Jenkins will find, and report on, Junit test results automatically, for example. Later on in the book, we will see how the Maven projects also simplify the configuration of many code quality metrics tools and reports.

Most of the other Post-build Actions are similar to those we saw in the freestyle build job.

**2.8.4. Deploying to an Enterprise Repository Manager(Nexus)**

One extra option does appear in the Maven build jobs is the ability to deploy your artifacts to a Maven repository (see Figure 2.8.4, “Deploying artifacts to a Maven repository”). An Enterprise Repository Manager is a server that acts as both a proxy/cache for public Maven artifacts, and as a central storage server for your own internal artifacts. Open Source Enterprise Repository Managers like Nexus (from Sonatype) and Artifactory (from JFrog) provide powerful maintenance and administration features that make configuring and maintaining your Maven repositories a lot simpler. Both these products have commercial versions, with additional features aimed at more sophisticated or high-end build infrastructures.

The advantage of getting Jenkins to deploy your artifacts (as opposed to simply running mvn deploy) is that, if you have a multimodule Maven build, the artifacts will only be deployed once the entire build has finished successfully. For example, suppose you have a multimodule Maven project with five modules.

If you run mvn deploy, and the build fails after three modules, the first two modules will have been deployed to your repository, but not the last three, which leaves your repository in an instable state.

Getting Jenkins to do the deploy ensures that the artifacts are only deployed as a group once the build has successfully finished.

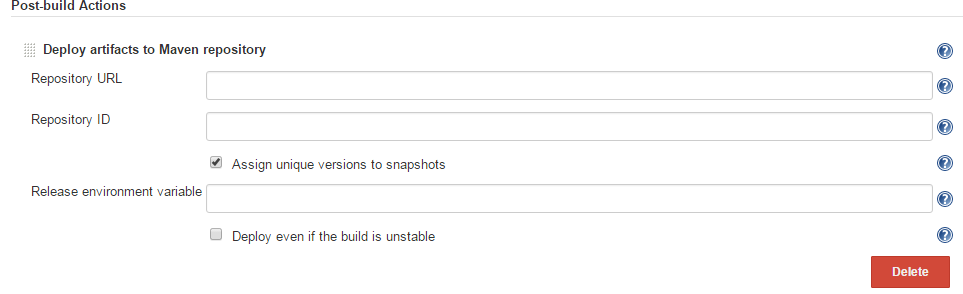


Figure 2.8.4. Deploying artifacts to a Maven repository

To do this, just tick the “Deploy artifacts to Maven repository” option in the “Post-Build actions”. You will need to specify the URL of the repository you want to deploy to. This needs to be the full URL to the repository (e.g., <http://nexus.acme.com/nexus/content/repositories/snapshots>

and not just <http://nexus.acme.com/nexus>

Most repositories need you to authenticate before letting you deploy artifacts to them. The standard Maven way to do this is to place a <server> entry in your local settings.xml file, as shown here:

<settings...>

<servers>

<server>

<id>nexus-snapshots</id>

<username>scott</username>

<password>tiger</password>

</server>

<server>

<id>nexus-releases</id>

<username>scott</username>

<password>tiger</password>

</server>

</servers>

</settings>

For the more security-minded, you can also encrypt these passwords if required. Then, enter the corresponding ID value in the Repository ID field in Jenkins. Jenkins will then be able to look up the right username and password, and deploy your artifacts. Once the build is finished, your

artifacts should be available in your Maven Enterprise Repository

**3.1. Integrating with Sonar**

Sonar is a tool that centralizes a range of code quality metrics into a single website (see Figure 3.1,“Code quality reporting by Sonar”). It uses several Maven plugins (Checkstyle, PMD, FindBugs, Cobertura or Clover, and others) to analyse Maven projects and generate a comprehensive set of code quality metrics reports. Sonar reports on code coverage, rule compliance, and documentation, but also on more high-level metrics such as complexity, maintainability and even technical debt. You can use plugins to extend its features and add support for other languages (such as support for CodeNarc for Groovy source code). The rules used by the various tools are managed and configured centrally on the Sonar website, and the Maven projects being analyzed don’t require any particular configuration. This makes Sonar a great fit for working on Maven projects where you have limited control over the pom files.

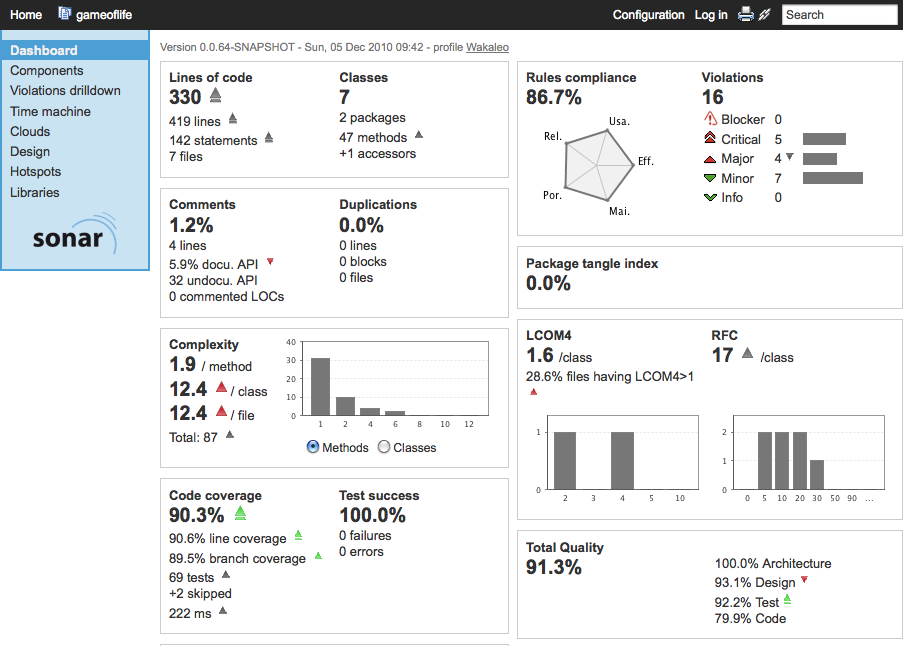


Figure 3.1. Code quality reporting by Sonar

In one of the most common usages of Sonar, Sonar automatically runs a set of Maven code quality related plugins against your Maven project, and stores the results into a relational database. The Sonar server, which you run separately, then analyzes and displays the results as shown in Figure 3.1, “Code quality reporting by Sonar”.

Jenkins integrates well with Sonar. The Jenkins Sonar Plugin lets you define Sonar instances for all of your projects, and then activate Sonar in particular builds. You can run your Sonar server on a different machine to your Jenkins instance, or on the same. The only constraint is that the Jenkins instance must have JDBC access to the Sonar database, as it injects code quality metrics directly into the database, without going through the Sonar website (see Figure 3.2, “Jenkins and Sonar”).

**Jenkins Sonar database**

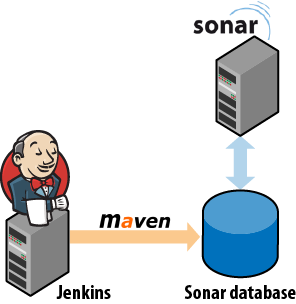
****

Figure 3.2. Jenkins and Sonar

Sonar also has an Ant bootstrap (with a Gradle bootstrap in the making at the time of writing) for non- Maven users.

You install the plugin in the usual way, via the Plugin Manager. Once installed, you configure the Jenkins Sonar plugin in the Configure System screen, in the Sonar section. This involves defining your

Sonar instances—you can configure as many instances of Sonar as you need. The default configuration assumes that you are running a local instance of Sonar with the default embedded database. This is useful for testing purposes but not very scalable. For a production environment, you will typically run Sonar

on a real database such as MySQL or Postgres, and you will need to configure the JDBC connection to the production Sonar database in Jenkins. You do this by clicking on the Advanced button and filling in the appropriate fields (see Figure 3.3, “Configuring Sonar in Jenkins”).

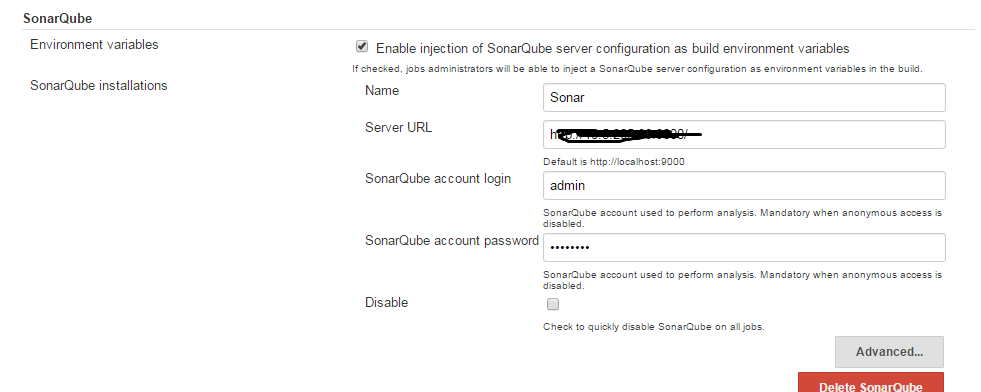


Figure 3.3. Configuring Sonar in Jenkins

The other thing you need to configure is when the Sonar build will kick off in a Sonar-enabled build job. You usually configure Sonar to run with one of the long-running Jenkins build jobs, such as the code quality metrics build. It is not very useful to run the Sonar build more than once a day, as Sonar stores metrics in 24-hour slices. The default configuration will kick off a Sonar build in a Sonar-enabled build job whenever the job is triggered by a periodically scheduled build or by a manual build.

To activate Sonar in your build job with the system-wide configuration options, just check the Sonar option in the Post-build Actions Sonar will run whenever your build is started by one of the trigger mechanisms defined above.

You typically set up Sonar to run on a regular basis, for example every night or once a week. So you can activate Sonar on your normal unit/integration test build job, simply by adding a schedule. This avoids duplicated configuration details between jobs. Or, if you already have a scheduled build job that runs with an appropriate frequency (such as a dedicated code quality metrics build), you can activate Sonar on this build job.

**4. Securing Jenkins**

**4.1. Introduction**

Jenkins supports several security models, and can integrate with several user repositories. In smaller organizations, where developers work in close proximity, security on your Jenkins machine may not be a large concern—you may simply want to prevent unidentified users tampering with your build job configurations. For larger organizations, with multiple teams, a stricter approach might be required, where only team members and system administrators are allowed to modify their build job configurations. And in situations where the Jenkins server may be exposed to a broader audience, such as on an internal corporate website, or even on the Internet, certain build jobs may be visible to all users whereas others will need to be hidden to unauthorized users.

In this chapter, we will look at how to configure different security configurations in Jenkins, for different environments and circumstances.

**4.2. Activating Security in Jenkins**

Setting up basic security in Jenkins is easy enough. Go to the main configuration page and check the Enable security checkbox (see Figure 4.1, “Enabling security in Jenkins”). This will display a number of options, that we will investigate in detail in this chapter. The first section, Security Realms, determines where Jenkins will look for users during authentication, and includes options such as using users stored in an LDAP server, using the underlying Unix user accounts (assuming, of course, that Jenkins is running on a Unix machine), or using a simple built-in user database managed by Jenkins.

The second section, Authorization, determines what users can do once they are logged in. This ranges from simple options like “Anyone can do anything” or “Logged-in users can do anything,” to more sophisticated role and project-based authorization policies.

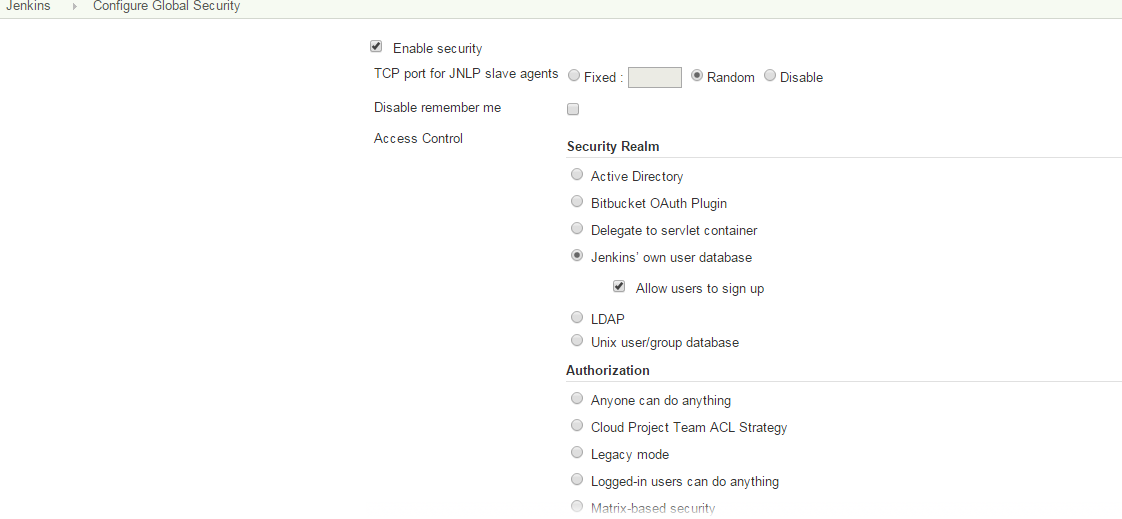


Figure 4.1. Enabling security in Jenkins

In the remainder of this chapter, we will look at how to configure Jenkins security for a number of common scenarios.

**4.3. Simple Security in Jenkins**

The most simple usable security model in Jenkins involves allowing authenticated users to do anything,whereas non-authenticated users will just have a read-only view of the build jobs. This is great for small teams—developers can manage the build jobs, whereas other users (testers, BAs, project managers and so on) can view the build jobs as required to view the status of the project. Indeed, certain build jobs may be set up just for this purpose, displaying the results of automated acceptance tests or code quality metrics, for example.

You can set up this sort of configuration to choose “Logged-in users can do anything” in the Authorization section. There are several ways that Jenkins can authenticate users but for this example, we will be using the simplest option, which is to use Jenkins’s own built in database. This is the configuration illustrated in Figure 4.1, “Enabling security in Jenkins”.

Make sure you tick the “Allow users to sign up” option. This option will display a Sign up link at the top of the screen to let users create their own user account as required (see Figure 4.2, “The Jenkins Sign up page”). It is a good idea for developers to use their SCM username here: in this case, Jenkins will be able to work out what users contributed to the SCM changes that triggered a particular build.

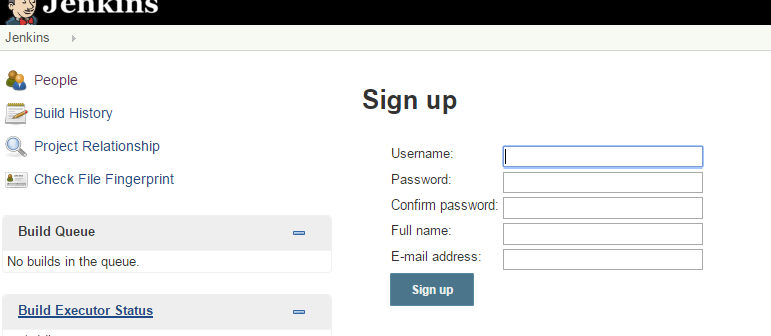


Figure 4.2. The Jenkins Sign up page

**4.4. Security Realms—Identifying Jenkins Users**

Jenkins lets you identify and manage users in a number of ways, ranging from a simple, built-in user database suitable for small teams to integration with enterprise directories, with many other options in between.

**4.4.1. Using Jenkins’s Built-in User Database**

The easiest way to manage user accounts in Jenkins is to use Jenkins’s internal user database. This is a good option if you want to keep things simple, as very little setup or configuration is required. Users who need to log on to the Jenkins server can sign up and create an account for themselves, and, depending on the security model chosen, an administrator can then decide what these users are allowed to do.

Jenkins automatically adds all SCM users to this database whenever a change is committed to source code monitored by Jenkins. These user names are used mainly to record who is responsible for each build job. You can view the list of currently known users by clicking on the People menu entry (see

Figure 4.3, “The list of users known to Jenkins”). Here, you can visualize the users that Jenkins currently knows about, and also see the last project they committed changes to. Note that this list contains all of the users who have ever committed changes to the projects that Jenkins monitors—they may not be (and usually aren’t) all active Jenkins users who are able to log on to the Jenkins server.



Figure 4.3. The list of users known to Jenkins

A user appearing in this list cannot necessarily log on to Jenkins. To be able to log on to Jenkins, the user account needs to be set up with a password. There are essentially two ways to do this. If you

have activated the “Allow users to sign up” option, users can simply sign up with their SCM user name and provide their email address and a password (see Section 4.3, “Simple Security in Jenkins”).

Another way to manage the current active users (those who can actually log on to Jenkins) is by clicking on the Manage Users link in the main Jenkins configuration page (see Figure 4.4, “You can also manage Jenkins users from the Jenkins configuration page”).

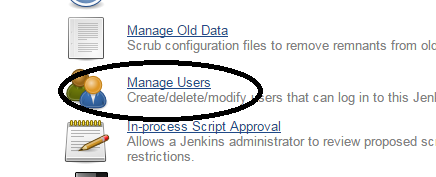


Figure 4.4. You can also manage Jenkins users from the Jenkins configuration page

The internal Jenkins database is sufficient for many teams and organizations. However, for larger organizations, it may become tedious and repetitive to manage large numbers of user accounts by hand, especially if this information already exists elsewhere. In the following sections, we will look at how to hook Jenkins up to other user management systems, such as LDAP repositories and Unix users and groups.

**4.4.2. Using an LDAP Repository**

Many organizations use LDAP directories to store user accounts and passwords across applications. Jenkins integrates well with LDAP, with no special plugins required. It can authenticate users using the LDAP repository, check group membership, and retrieve the email address of authenticated users.

To integrate Jenkins with your LDAP repository, Just select “LDAP” in the Security Realm section, and fill in the appropriate details about your LDAP server (see Figure 4.5, “Configuring LDAP in Jenkins”).

The most important field is the repository server. If you are using a non-standard port, you will need to provide this as well (for example, ldap.acme.org:1389). Or, if you are using LDAPS, you will need to specify this as well (for example, ldaps://ldap.acme.org)

If your server supports anonymous binding, this will probably be enough to get you started. If not, you can use the Advanced options to fine-tune your configuration.

Most of the Advanced fields can safely be left blank unless you have a good reason to change them. If your repository is extremely large, you may want to specify a root DN value (e.g., dc=acme, dc=com) and/or a User and Group search base (e.g., ou=people) to narrow down the scope of user queries.

This is not usually required unless you notice performance issues. Or, if your server does not support anonymous binding, you will need to provide a Manager DN and a Manager DN password, so that

Jenkins can connect to the server to perform its queries.

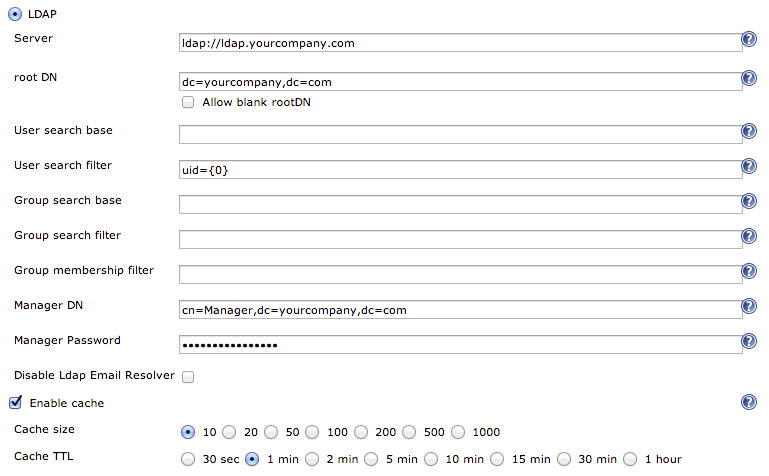


Figure 4.5. Configuring LDAP in Jenkins

Once you have set up LDAP as your Security Realm, you can configure your favorite security model as described previously. When users log on to Jenkins, they will be authenticated against the LDAP repository.

You can also use LDAP groups, though the configuration is not immediately obvious. Suppose you have defined a group called JenkinsAdmin in your LDAP repository, with a DN of cn=JenkinsAdmin, ou-

Groups, dc=acme, dc=com. To refer to this group in Jenkins, you need to take the common name (cn) in uppercase, and prefix it with ROLE\_. So cn=JenkinsAdmin becomes ROLE\_JENKINSADMIN. You can see an example of LDAP groups used in this way in Figure 4.6, “Using LDAP Groups in Jenkins”.

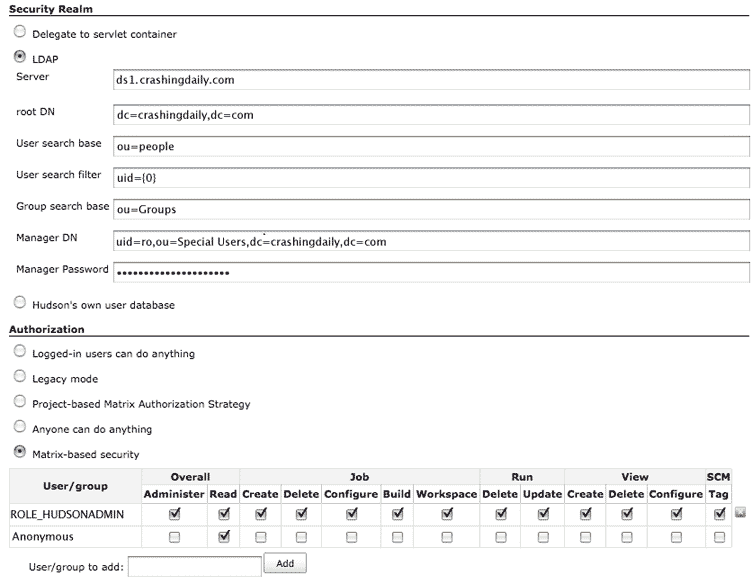


Figure 4.6. Using LDAP Groups in Jenkins

**4.4.3. Using Unix Users and Groups**

If you are running Jenkins on a Unix machine, you can also ask Jenkins to use the user and group accounts defined on this machine. In this case, users will log into Jenkins using their Unix account logins and passwords. This uses Pluggable Authentication Modules (PAM), and also works fine with NIS.

In its most basic form, this is somewhat cumbersome, as it requires new user accounts to be set up and configured for each new Jenkins user. It is only really useful if these accounts need to be set up for other purposes.

**4.5. Authorization—Who Can Do What**

Once you have defined how to identify your users, you need to decide what they are allowed to do. Jenkins supports a variety of strategies in this area, ranging from a simple approach where a logged-in user can do anything to more involved roles and project-based authentication strategies.

**4.5.1. Matrix-based Security**

Letting signed-in users do anything is certainly flexible, and may be all you need for a small team. For larger or multiple teams, or cases where Jenkins is being used outside the development environment, a more sophisticated approach is generally required.

Matrix-based security is a more sophisticated approach, where different users are assigned different rights, using a role-based approach.

**4.5.1.1. Setting up matrix-based security**

The first step in setting up matrix-based security in Jenkins is to create an administrator. This is an essential step, and must be done before all others. Now your administrator can be an existing user, or one created especially for the purpose. If you want to create a dedicated administrator user, simply create

one by signing up in the usual way (see Figure 4.2, “The Jenkins Sign up page”). It doesn’t have to be associated with an SCM user.

Once you have your admin user ready, you can activate matrix-based security by selecting “Matrixbased security” in the Authorization section of the main configuration page. Jenkins will display a table

containing authorized users, and checkboxes corresponding to the various permissions that you can assign to these users (see Figure 4.7, “Matrix-based security configuration”).

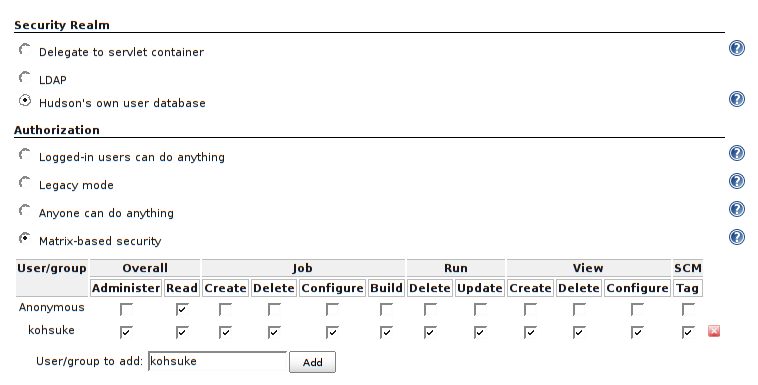


Figure 4.7. Matrix-based security configuration

The special “anonymous” user is always present in the table. This user represents unauthenticated users.

Typically, you only grant very limited rights to unauthenticated users, such as read-only access, or no access at all (as shown in Figure 4.7, “Matrix-based security configuration”).

The first thing you need to do now is to grant administration rights to your administrator. Add your administration user in the “User/group to add” field and click on Add

**4.5.1.2. Fine-tuning user permissions**

Once you have set up your administrator account, you can add any other users that need to access your Jenkins instance. Simply add the user names and tick the permissions you want to grant them (see

Figure 4.8, “Setting up other users”).

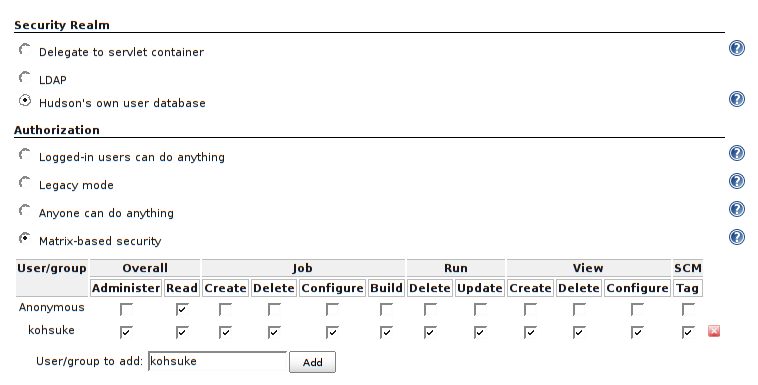


Figure 4.8. Setting up other users

You can grant a range of permissions, which are organized into several groups: Overall, Slave, Job, Run,View and SCM. Most of the permissions are fairly obvious, but some need a little more explanation.

The individual permissions are as follows:

Overall

This group covers basic system-wide permissions:

Administer

Lets a user make system-wide configuration changes and other sensitive operations, for example in the main Jenkins configuration pages. This should be reserved for the Jenkins administrator.

Read

This permission provides read-only access to virtually all of the pages in Jenkins. If you want anonymous users to be able to view build jobs freely, but not to be able to modify or start them, grant the Read role to the special “anonymous” user. If not, simply revoke this

permission for the Anonymous user. And if you want all authenticated users to be able to see build jobs, then add a special user called “authenticated”, and grant this user Overall/Read permission.

Slave

This group covers permissions about remote build nodes, or slaves:

Configure

Create and configure new build nodes.

Delete

Delete build nodes.

Job

This group covers job-related permissions:

Create

Create a new build job.

Delete

Delete an existing build job.

Configure

Update the configuration of an existing build jobs.

Read

View build jobs.

Build

Start a build job.

Workspace

View and download the workspace contents for a build job. Remember, the workspace contains source code and artifacts, so if you want to protect these from general access, you should revoke this permission.

Release

Start a Maven release for a project configured with the M2Release plugin.

Run

This group covers rights related to particular builds in the build history:

Delete

Delete a build from the build history.

Update

Update the description and other properties of a build in the build history. This can be useful if a user wants to leave a note about the cause of a build failure, for example.

View

This group covers managing views:

Create

Create a new view.

Delete

Delete an existing view.

Configure

Configure an existing view.

SCM

Permissions related to your version control system:

Tag

Create a new tag in the source code repository for a given build.

Others

There can also be other permissions available, depending on the plugins installed. One useful

one is:

Promote

If the Promoted Builds plugin is installed, this permission allows users to manually promote a build.

**4.5.2. Project-based Security**

Project-based security lets you build on the matrix-based security model we just discussed, and apply it to individual projects. Not only can you assign system-wide roles for your users, you can also configure more specific rights for certain individual projects.

To activate project-level security, select “Project-based Matrix Authorization Strategy” in the Authorization section of the main configuration screen (see Figure 4.9, “Project-based security”).

Here, you set up the default rights for users and groups, as we saw with Matrix-based security (see Section 4.5.1, “Matrix-based Security”).

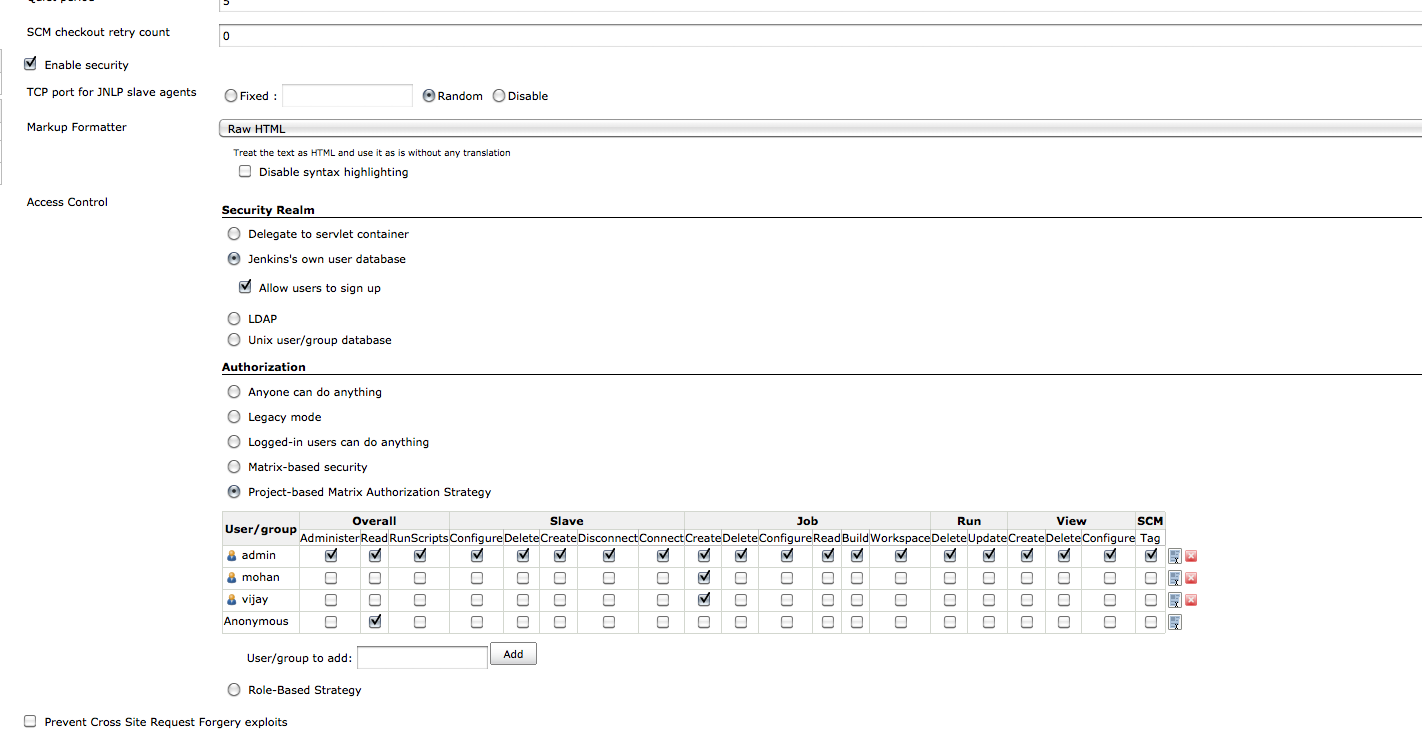


Figure 4.9. Project-based security

These are the default permissions that apply to all projects that have not been specially configured. However, when you use project-based security, you can also set up special project-specific permissions.

You do this by selecting “Enable project-based security” in the project configuration screen (see Figure 4.10, “Configuring project-based security”). Jenkins will display a table of project-specific permissions. You can configure these permissions for different users and groups just like on the system wide configuration page. These permissions will be added to the system-wide permissions to produce a project-specific set of permissions applicable for this project.

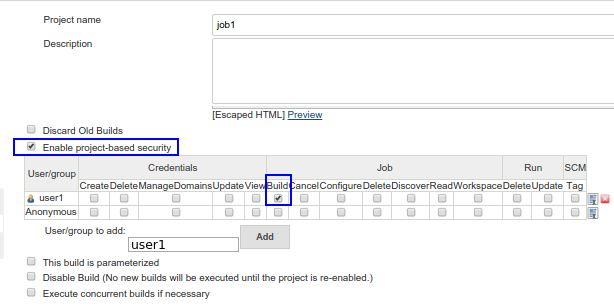


Figure 4.10. Configuring project-based security

**4.5.3. Role-based Security**

Sometimes managing user permissions individually can be cumbersome, and you may not want to integrate with an LDAP server to set up groups that way. A more recent alternative option is to use the Role Strategy plugin, which allows you to define global and project-level roles, and assign these roles to users. You install the plugin in the usual way, via the Plugin Manager. Once installed, you can activate this authorization strategy in the main configuration page (see Figure 4.11, “Setting up Role-based security”).

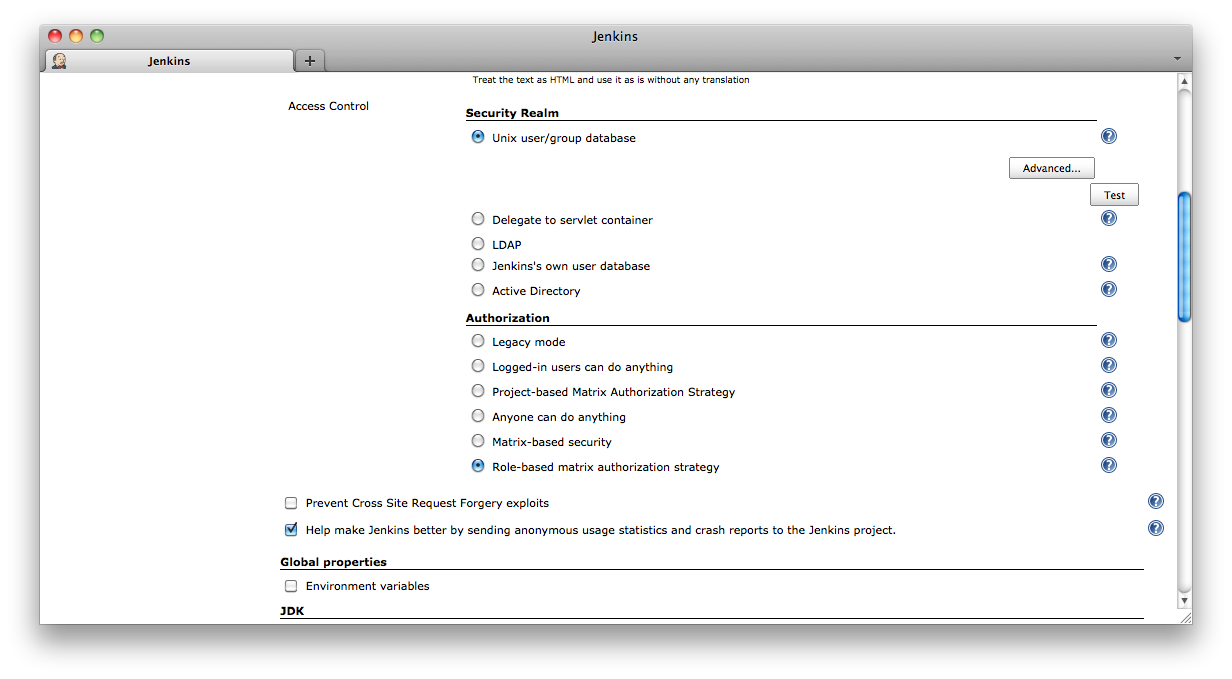


Figure 4.11. Setting up Role-based security

Once you have set this up, you can define roles that regroup sets of related permissions. You set up and configure your roles, and assign these roles to your users, in the Manage Roles screen, which you can access in the Manage Jenkins screen (see Figure 7.24, “The Manage Roles configuration menu”).

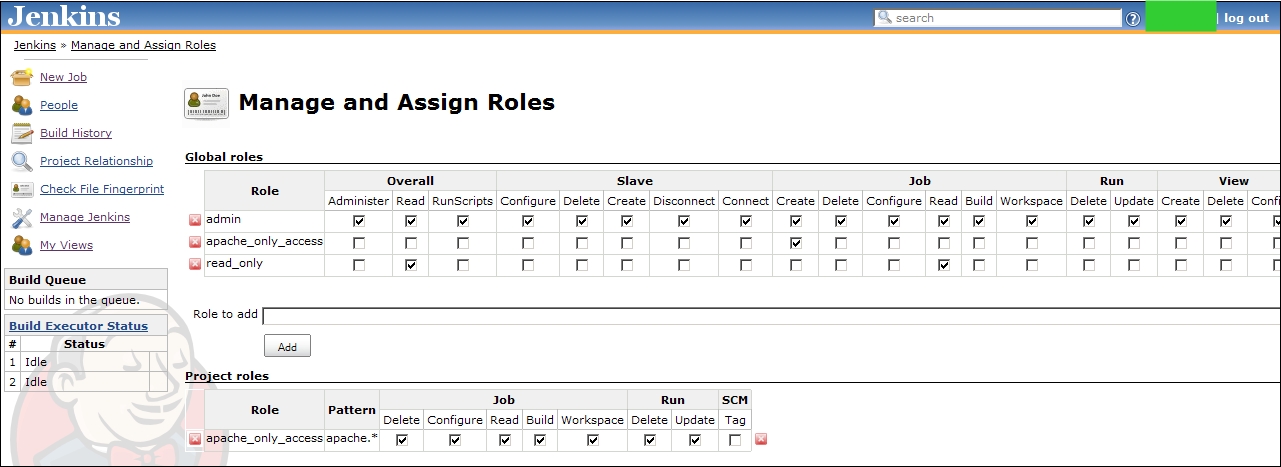


Figure 4.12. The Manage Roles configuration menu

In the Manage Roles screen, you can set up global and project-level permissions. Global permissions apply across all projects, and are typically system-wide administration or general access permissions Setting these roles up is intuitive and similar to setting up user permissions in the other security models we have seen.

Project roles are slightly more complicated. A project role regroups a set of permissions that are applicable to one or more (presumably related) projects. You define the relevant projects using a regular expression, so it helps to have a clear and consistent set of naming conventions in place for your project names . For example, you may wish to create roles distinguishing developers with full configuration rights on their own project from users who can simply trigger a build and view the build results, or create roles where developers can configure certain automated deployment build jobs, but only production teams are allowed to execute these jobs.

Once you have defined these roles, you can go to the Assign Roles screen to set up individual users or groups with these roles

Role-based strategy is relatively new in Jenkins, but it is an excellent way to simplify the task of managing permissions in large, multi-team and multiproject organizations.