**Chapter 5. Setting Up Your Build Jobs**

**Introduction**

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.

A software project will usually have several related build jobs. For example, you might choose to start off with a dedicated build job that runs all of your unit tests. If these pass, you might proceed to a build job that executes longer-running integration tests, runs code quality metrics, or generates technical documentation, before finally bundling up your web application and deploying it to a test server.

In Jenkins, build jobs are easy to set up. In this chapter, we will look at the main types of build jobs and how to configure them. In later chapters, we will take things further, looking at how to organize multiple build jobs, how to set up build promotion pipelines, and how to automate the deployment process. But, for now, let’s start off with how to set up your basic build jobs in Jenkins.

**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 5-1](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch05s02.html#fig-hudson-build-types)).

Freestyle software project

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

Maven project

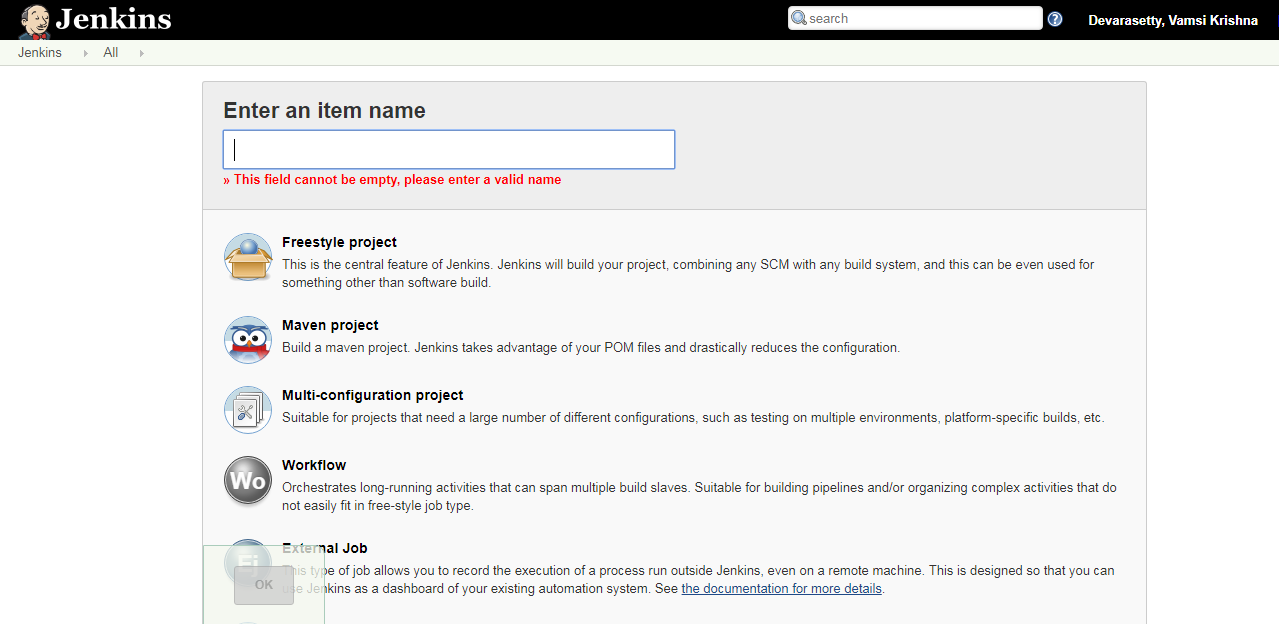
The “maven2/3 project” is a build job specially adapted to Maven projects. Jenkins understands Maven *pom*files and project structures, and can use the information gleaned from the *pom* file to reduce the work you need to do to set up your project.

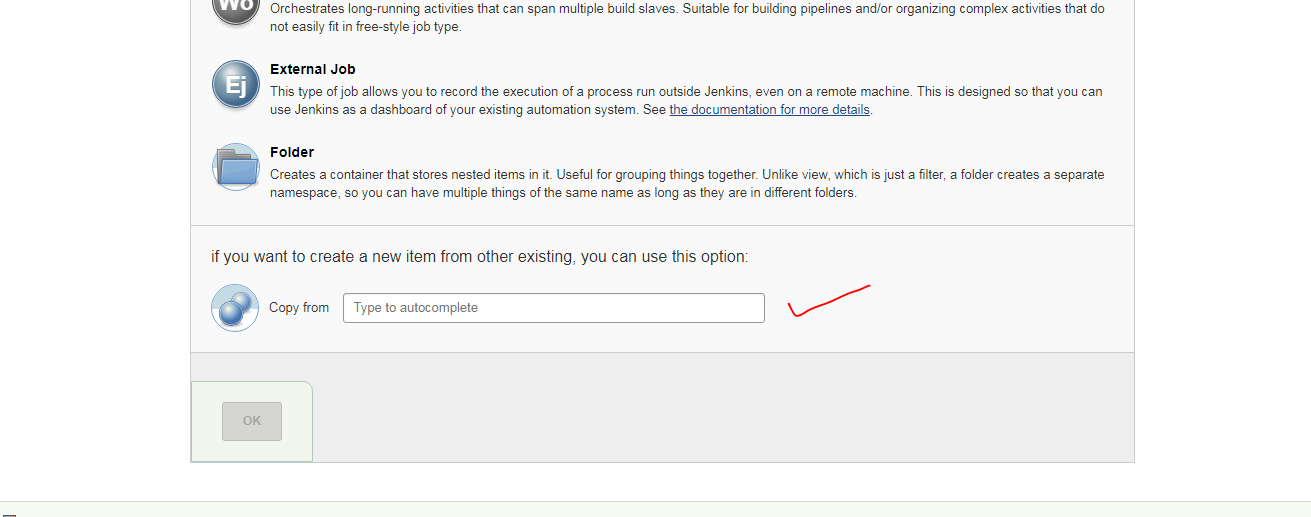
Monitor an external job

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

Multiconfiguration job

The “multiconfiguration project” (also referred to as a “matrix project”) lets you run the same build job in many different configurations. This powerful feature can be useful for testing an application in many different environments, with different databases, or even on different build machines. We will be looking at how to configure multiconfiguration build jobs later on in the book.





You can also copy an existing job, which is a great way to create a new job that is very similar to an existing build job, except for a few configuration details.

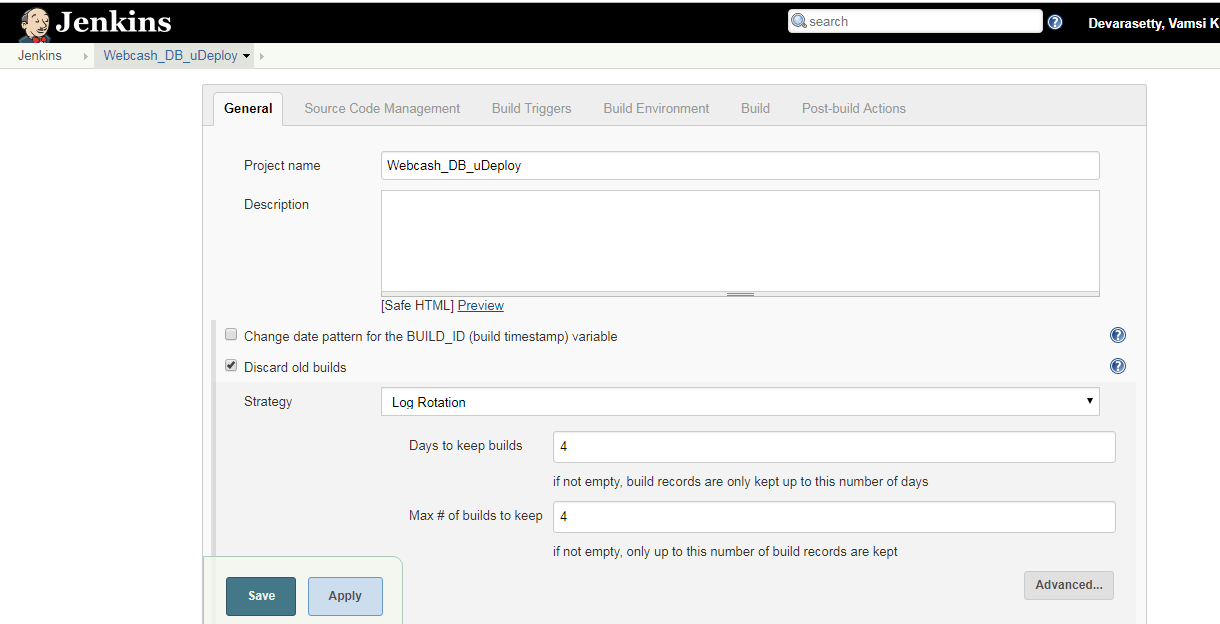
In this chapter, we will focus on the first two types of build jobs, which are the most commonly used. We will discuss the others later on. Let’s start with the most flexible option: the freestyle build job.

# Creating a Freestyle Build Job

The freestyle build job is the most flexible and configurable option, and can be used for any type of project. It is relatively straightforward to set up, and many of the options we configure here also appear in other build jobs.

## General Options

The first section you see when you create a new freestyle 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



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.

One important aspect that you should think about upfront is how you want to handle build history. Build jobs can consume a lot of disk space, especially if you store the build artifacts (the binary files, such as JARs, WARs, TARs, etc., generated by your build job). Even without artifacts, keeping a record of every build job consumes additional disk space and memory, which may or may not be justified, depending on the nature of your build job. For example, for a code quality metrics build that reports on static analysis and code coverage metrics over time, you might want to keep a record of the builds for the duration of the project, whereas, for a build job that automatically deploys an application to a test server, keeping the build history and artifacts for posterity might be less important.

The Discard Old Builds option lets you limit the number of builds you record in the build history. You can either tell Jenkins to only keep recent builds (Jenkins will delete builds after a certain number of days), or to keep no more than a specified number of builds. If a certain build has particular sentimental value, you can always tell Jenkins to keep it forever by using the Keep forever button on the build details page (see [Figure 5-3](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch05s03.html#fig-jenkins-keep-forever)). Note that this button will only appear if you have asked Jenkins to discard old builds.

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.

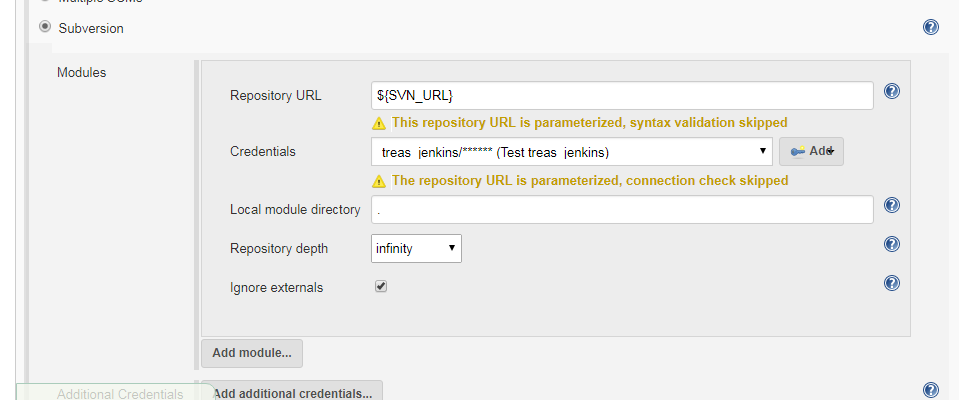
# Configuring Source Code Management

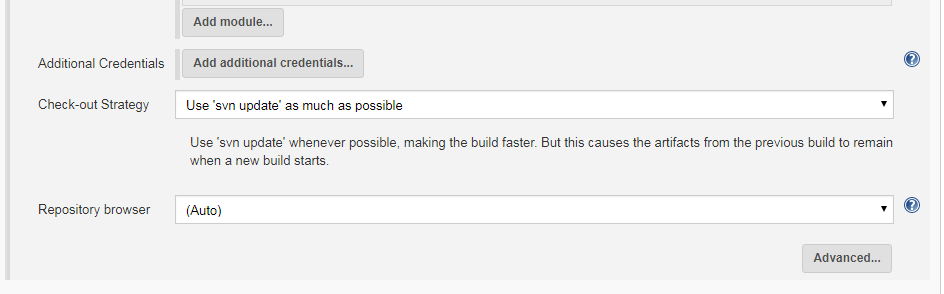
In its most basic role, a Continuous Integration server monitors your version control system, and checks out the latest changes as they occur. The server then compiles and tests the most recent version of the code. Alternatively, it may simply check out and build the latest version of your source code 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 includesAccurev, 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.

## Working with Subversion

Subversion is one of the most widely used version control systems, and Jenkins comes bundled with full Subversion support .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.

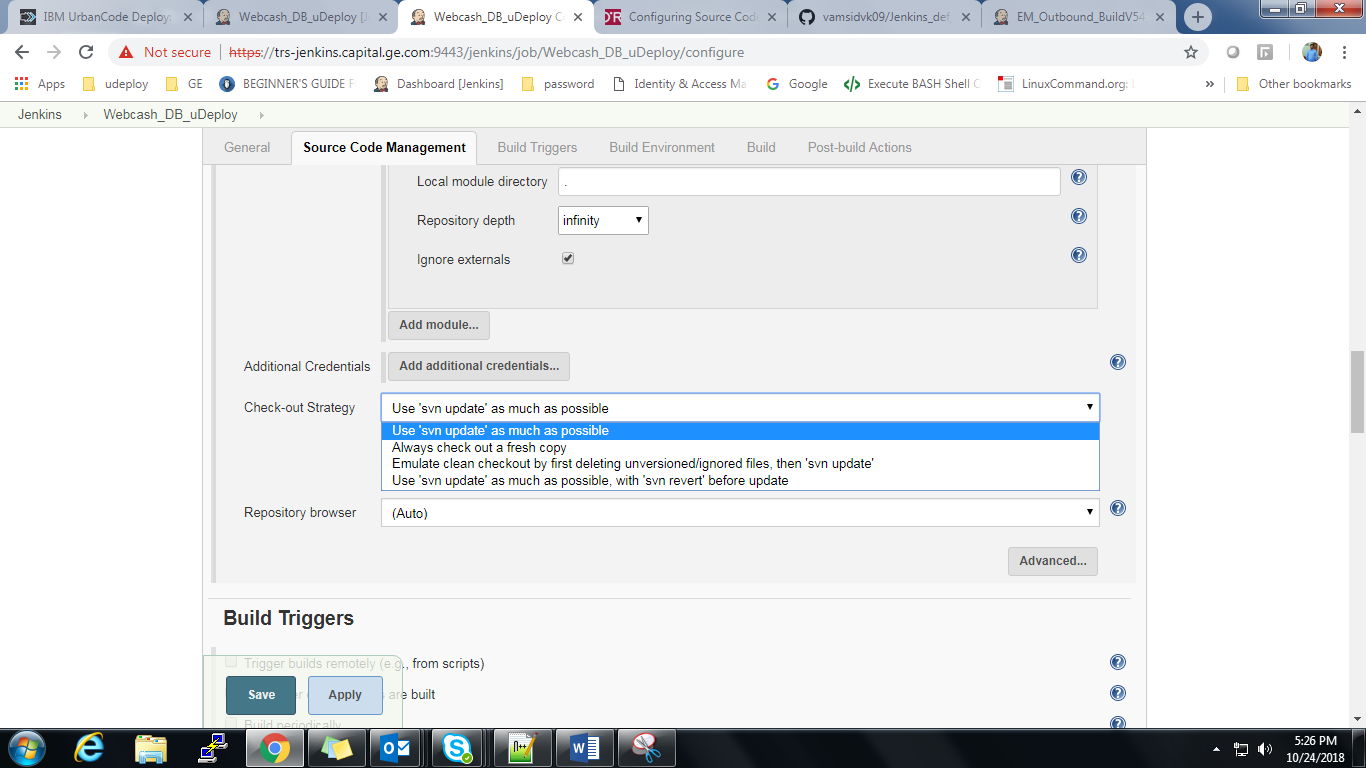




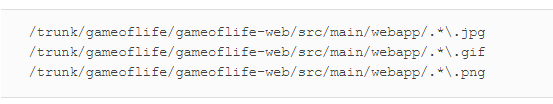
By default, Jenkins will check out the repository contents into a subdirectory of your workspace, whose name will match the last element in the Subversion URL. So if your Subversion URL is <svn://localhost/gameoflife/trunk>, Jenkins will check out the repository contents to a directory called trunk in 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 targetdirectory 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”), whichwill 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.

Jenkins also lets you refine the changes that will trigger a build. In the Advanced section, you can use the Excluded Regions field to tell Jenkins not to trigger a build if only certain files were changed. This field takes a list of regular expressions, which identify files that should not trigger a build. For example, suppose you don’t want Jenkins to start a new build if only images have been changed. To do this, you could use a set of regular expressions like the following:



Alternatively, you can specify the Included Regions, if you are only interested in changes in part of the source code directory structure. You can even combine the Excluded Regions and Included Regions fields—in this case a modified file will only trigger a build if it is in the Included Regions but not in the Excluded Regions.

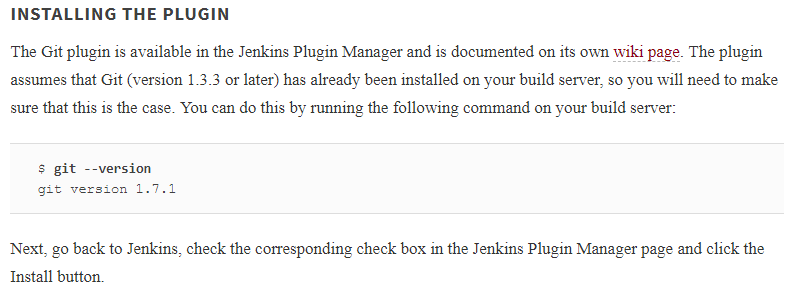


You can also ignore changes coming from certain users (Excluded Users), or with certain commit messages (Excluded Commit Messages). For example, if your project uses Maven, you may want to use the Maven Release Plugin to promote your application from snapshot versions to official releases. This plugin will automatically bump up the version number of your application from a snapshot version used during development (such as 1.0.1-SNAPSHOT) to a release (1.0.1), bundles up and deploys a release of your application with this versionnumber, and then moves the version on to the next snapshot number (e.g., 1.0.2-SNAPSHOT) for ongoing development. During this process Maven takes care of many SCM bookkeeping tasks, such as committing the source code with the release version number and creating a tag for the released version of your application, and then committing the source code with the new snapshot version number.



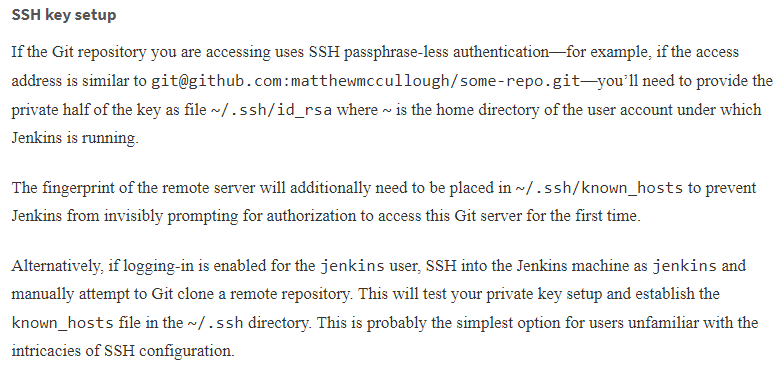
## Working with Git

[Git](http://git-scm.com/) is a popular distributed version control system that is a logical successor to [Subversion](http://subversion.tigris.org/) and a mind-share competitor to [Mercurial](http://mercurial.selenic.com/). 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. We’ll discuss the supplemental plugins shortly.



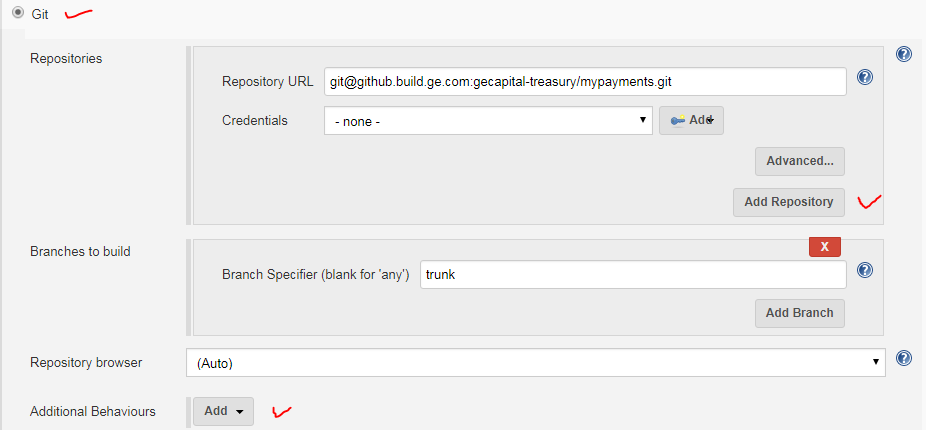
After installing the Git plugin, a small new set of configuration options will be available on the Manage Jenkins→Global tool configuration page . In particular, you need to provide the path to your Git executable. If Git is already installed on the system path, just put “git” here.





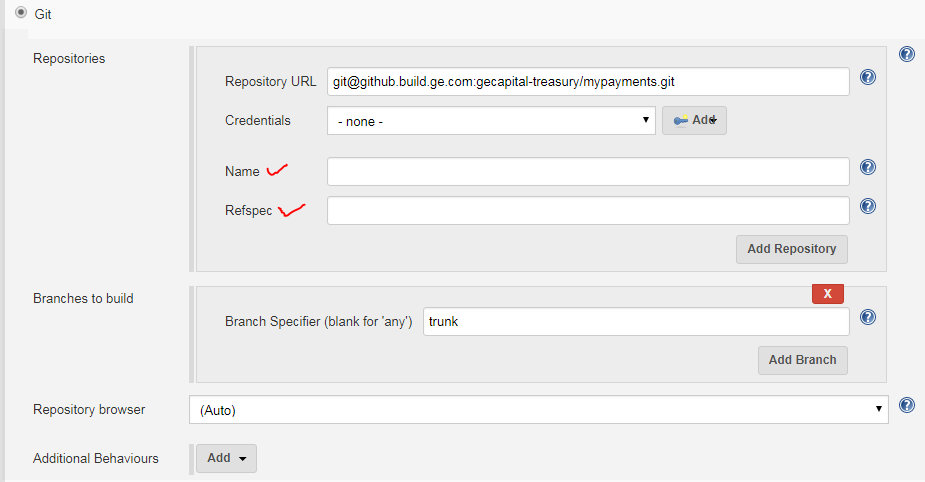
### USING THE PLUGIN

On either an existing or a new Jenkins project, a new Source Code Management option for Git will be displayed. From here, you can configure one or more repository addresses (see [Figure 5-9](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch05s04.html#I_figure1_id2245479)). One repository is usually enough for most projects: adding a second repository can be useful in more complicated cases, and lets you specify distinct named locations for pull and push operations.



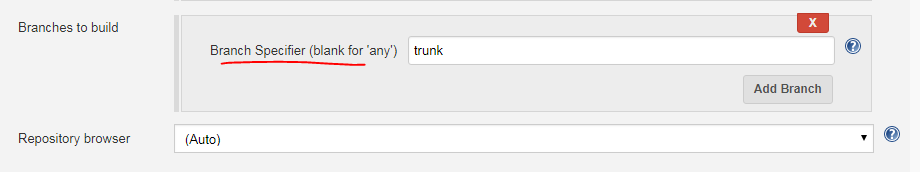
The Name of repository is a shorthand title (a.k.a. remote in Git parlance) for a given repository, that you can refer to later on in the merge action configuration.

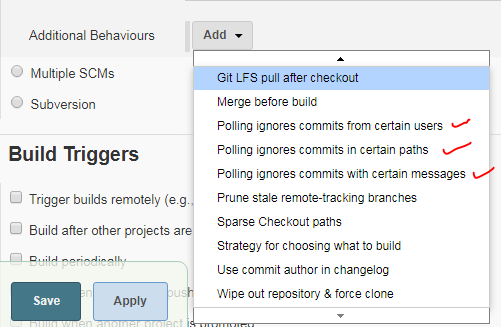
The Refspec is a Git-specific [language](http://progit.org/book/ch9-5.html) for controlling precisely what is retrieved from remote servers and under what namespace it is stored locally.



#### Branches to build

The branch specifier ([Figure 5-11](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch05s04.html#I_figure1_id2245638)) is the wildcard pattern or specific branch name that should be built by Jenkins. If left blank, all branches will be built. At the time of this writing, after the first time saving a job with a blank branches to build setting, it is populated with \*\*, which means “build all branches.”





#### Excluded regions

Regions (seen in [Figure 5-12](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch05s04.html#I_figure1_id2245692)) are named specific or wildcard paths in the codebase that, even when changed, should not trigger a build. Commonly these are noncompiled files such as localization bundles or images, which, understandably might not have an effect on unit or integration tests.

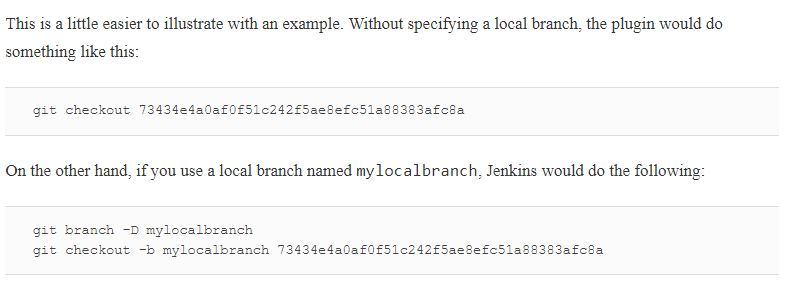
#### Excluded users

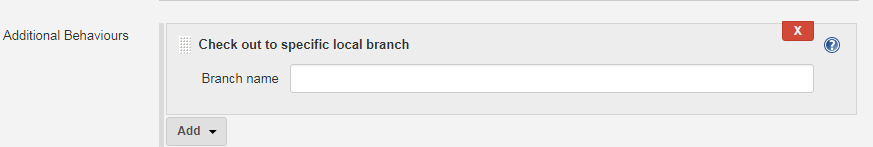
The Git plugin also lets you ignore certain users, even if they make changes to the codebase that would typically trigger a build.

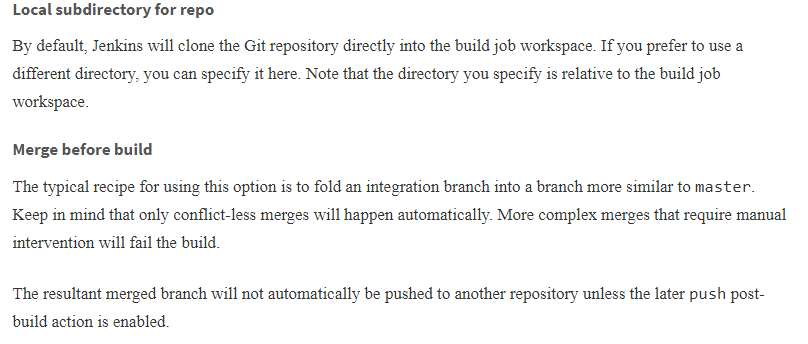
This is not as spiteful as it sounds: excluded users are typically automated users, not human developers, that happen to have distinct accounts with commit rights in the source control system. These automated users often are performing small numeric changes such as bumping up version numbers in a pom.xml file, rather than making actual logic changes. If you want to exclude several users, just place them on separate lines.

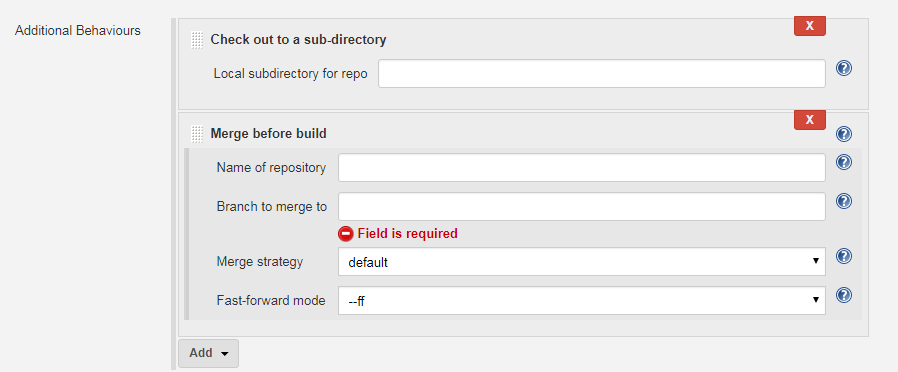
#### Checkout/merge to local branch

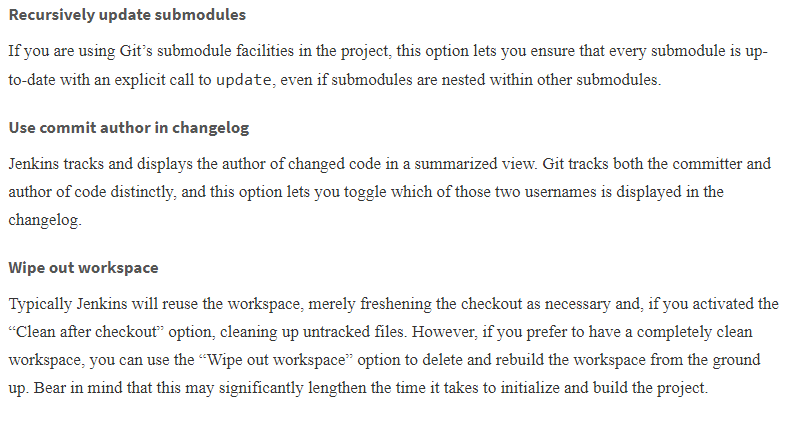
There are times when you may want to create a local branch from the tree you’ve specified, rather than just using a direct detached HEAD checkout of the commit’s hash. In this case, just specify your local branch in the “Checkout/merge to a local branch” field.

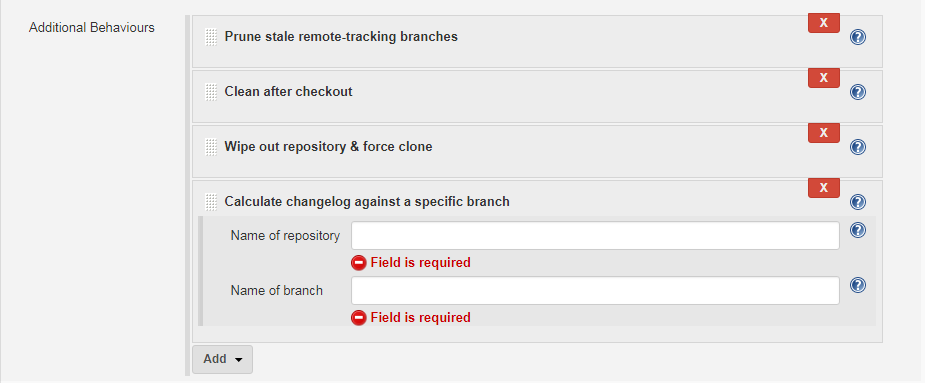






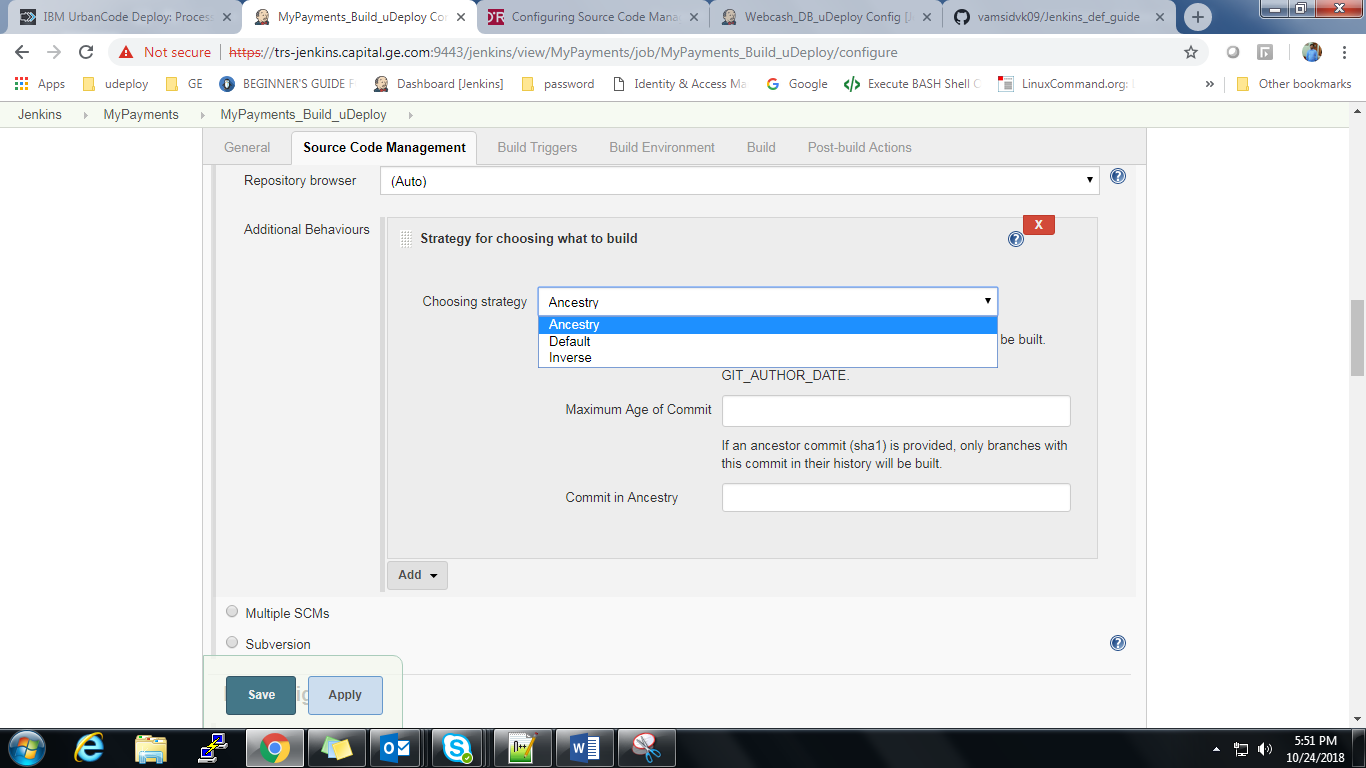






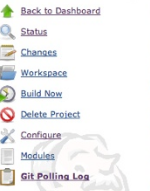
#### Choosing strategy

Jenkins decides which branches to build based on a strategy (see [Figure 5-13](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch05s04.html#I_figure1_id2245980)). Users can influence this branch-search process.



### BUILD TRIGGERS

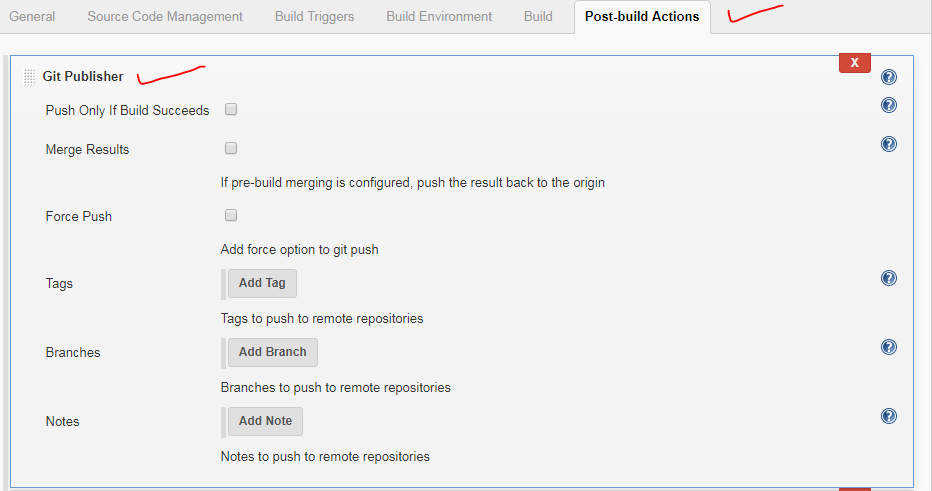
The basic Git plugin offers the ability to Poll SCM on a timed basis, looking for changes since the last inquiry. If changes are found, a build is started. The polling log (shown in [Figure 5-16](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch05s04.html#I_figure1_id2246150)) is accessible via a link on the left hand side of the page in the navigation bar when viewing a specific job. It offers information on the last time the repository was polled and if it replied with a list of changes



The Git polling is distilled into a more developer-useful format that shows commit comments as well as hyperlinking usernames and changed files to more detailed views of each.

### POST-BUILD ACTIONS

The Git plugin for Jenkins adds Git-specific capabilities to the post-processing of the build artifacts. Specifically, the Git Publisher (shown in [Figure 5-19](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch05s04.html#I_figure1_id2246534)) offers merging and pushing actions. Check the Git Publisher checkbox to display four options.



#### Push only if build succeeds

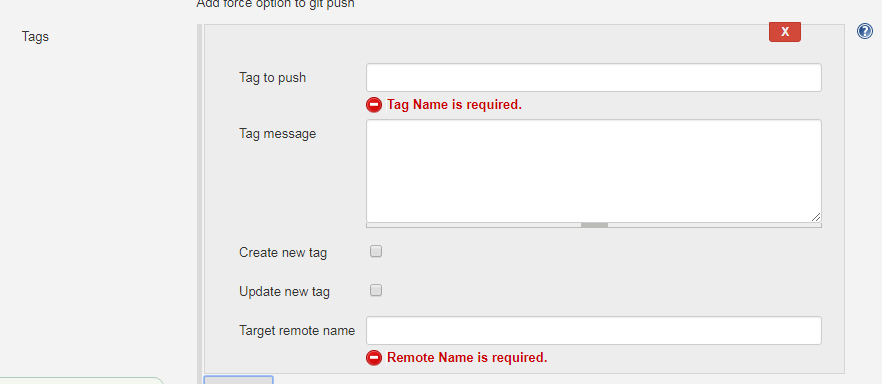
If a merge or other commit-creating action has been taken during the Jenkins build, it can be enabled to push to a remote.

#### Merge results

If prebuild merging is configured, push the merge-resultant branch to its origin

#### Tags

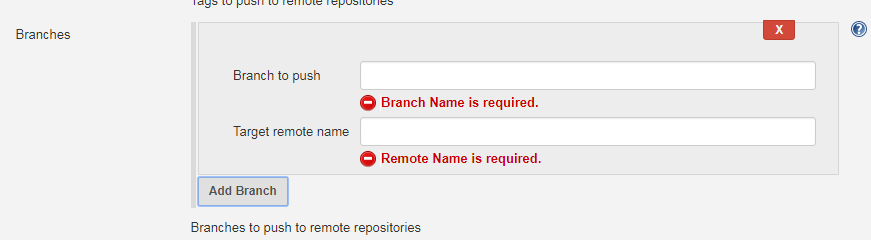
When pushing tags, each tag can be named and chosen to be created if it does not exist (which fails if it does already exist). Environment variables can be embedded in the tag name.



#### Branches

The current HEAD used in the Jenkins build of the application can be pushed to other remotes as an after-step of the build. You only need to provide the destination branch name and remote name.

Names of remotes are validated against the earlier configuration of the plugin. If the remote doesn’t exist, a warning is displayed.



# 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 5-23](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch05s05.html#fig-hudson-new-job-triggers)):

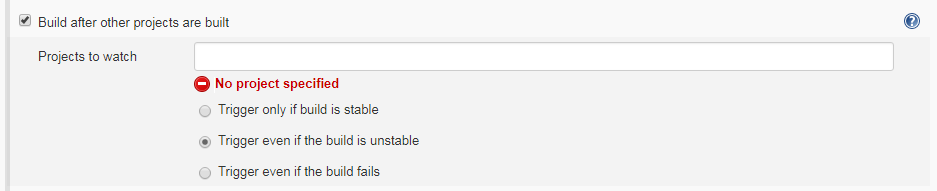
* Start a build job once another build job has completed
* Kick off builds at periodical intervals
* Poll the SCM for changes



## 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

There is a symmetrical field in the Post-build actions section of the preceding build job called (appropriately enough) “Build other projects”. This field will be automatically updated in the corresponding build jobs whenever you modify the “Build after other projects are built” field. However, unlike the “Build after other projects are built” field, this field gives you the option to trigger a build even if the build is unstable (see [Figure 5-24](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch05s05.html#fig-hudson-new-job-trigger-build-after)). This is useful, for example, if you want to run a code quality metrics build job even if there are unit test failures in the default build job.

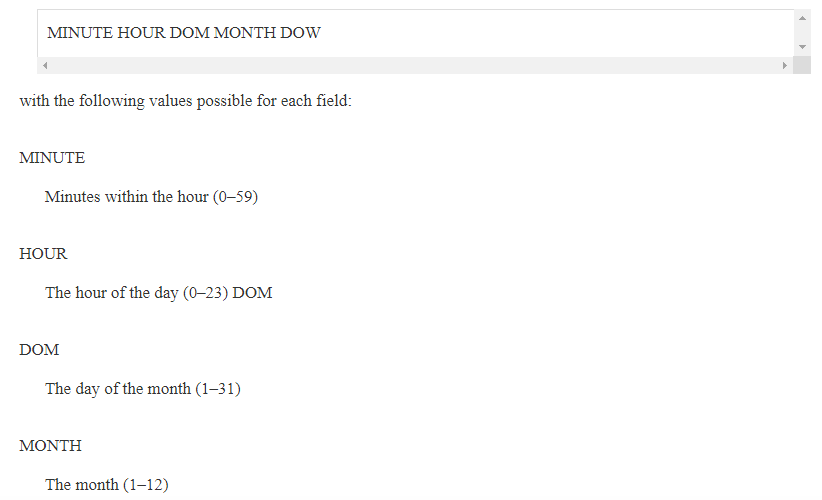


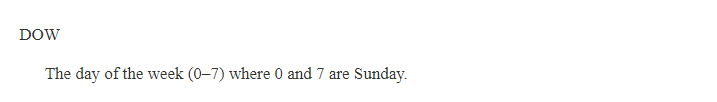
## Scheduled Build Jobs

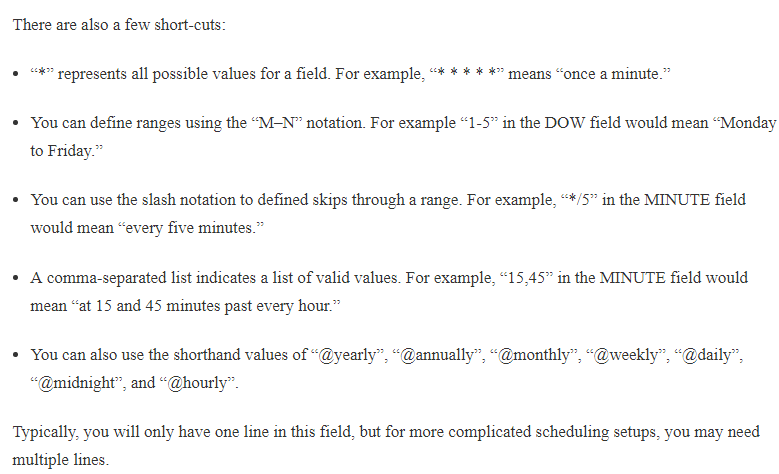
Another strategy is simply to trigger your build job at regular intervals. It is important to note that this is not actually Continuous Integration—it is simply scheduled builds, something you could also do, for example, as a Unix cron job. In the early days of automated builds, and even today in many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

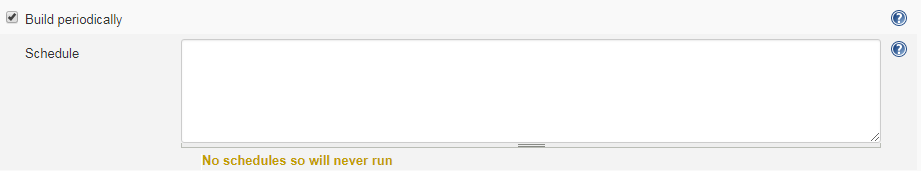
There are nevertheless a few cases where scheduled builds do make sense. This includes very long running build jobs, where quick feedback is less critical. For example, intensive load and performance tests which may take several hours to run, or Sonar build jobs. Sonar is an excellent way to keep tabs 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:



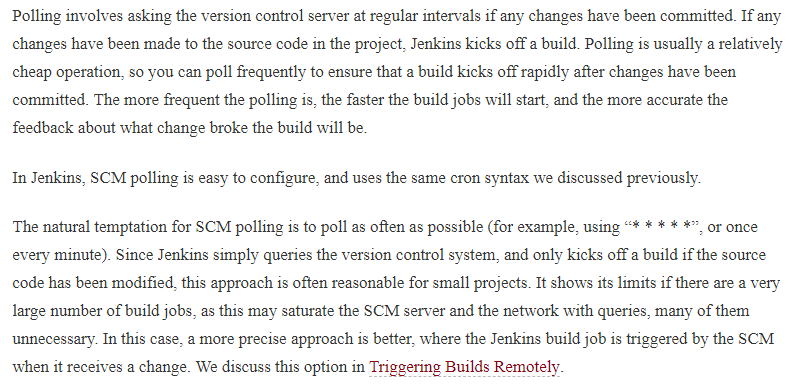


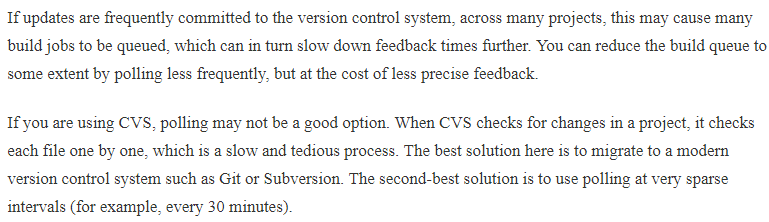


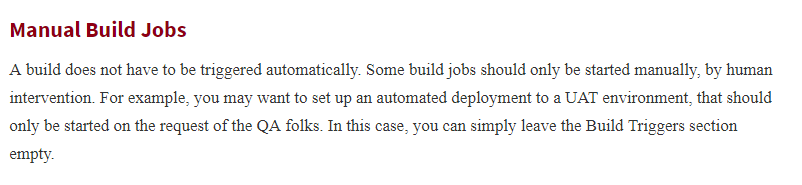


## Polling the SCM

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.







# Build Steps

Now Jenkins should know where and how often to obtain the project source code. The next thing you need 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 5-26](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch05s06.html#fig-hudson-build-steps)). 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.

In the remainder of this section, we will delve into some of the more common types of build steps

## 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

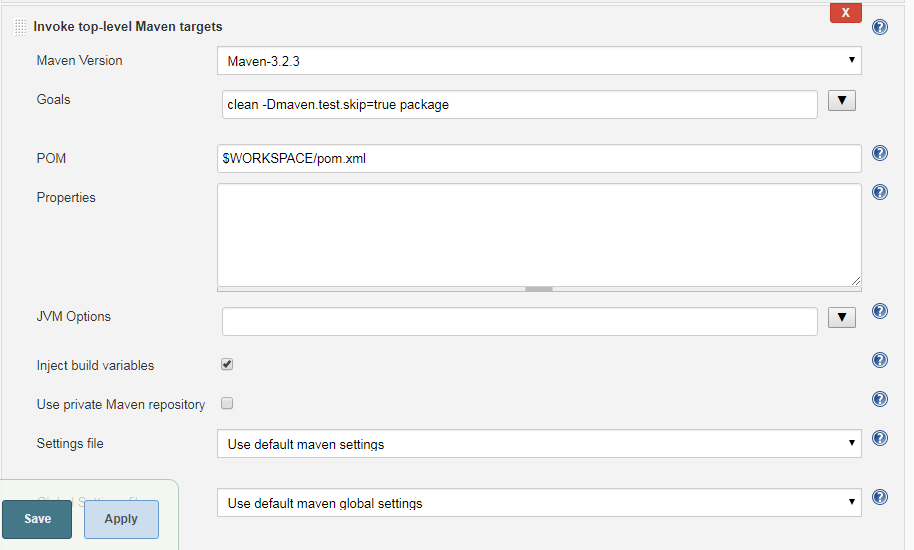
This option tells Maven not to prompt for any input from the user, just using the default values if any are required. If Maven does prompt for any input during the Jenkins build, the build will get stuck indefinitely.

-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

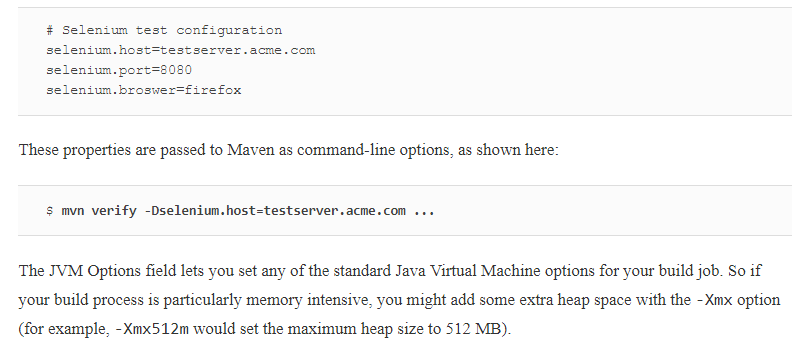
This option forces Maven to write JUnit output to the console, rather than to text files in the target directory as it normally would. This way, any test failure details are directly visible in the build job console output. The XML files that Jenkins needs for its test reporting will still be generated.



The advanced options are also worth investigating (click on the Advanced button).

The optional *POM* field lets you override the default location of the Maven *pom.xml* file. This is the equivalent of running Maven from the command line with the -f or --file option. This is useful for some multimodule Maven projects where the aggregate *pom.xml* file (the one containing the <modules> section) is located in a subdirectory rather than at the top level.

The Properties field lets you set property values that will be passed into the Maven build process, using the standard property file format illustrated here:

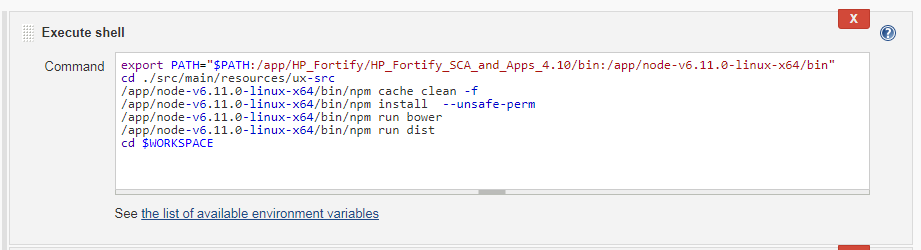


The final option lets you configure a private Maven repository for this build job. Normally, Maven will just use the default Maven repository (usually in the .m2/repository folder in the user’s home directory). Occasionally, this can lead to build jobs interfering with each other, or use inconsistent snapshot versions from one build to another. To be sure that your build is run in clean laboratory conditions, you can activate this option. Your build job will get its own private repository, reserved for its own exclusive use. On the downside, the first time the build job runs a build, this may take some time to download all of the Maven artifacts, and private repositories can take up a lot of space. However, it is the best way of guaranteeing that your build is run in a truly isolated environment.

## Executing a Shell or Windows Batch Command

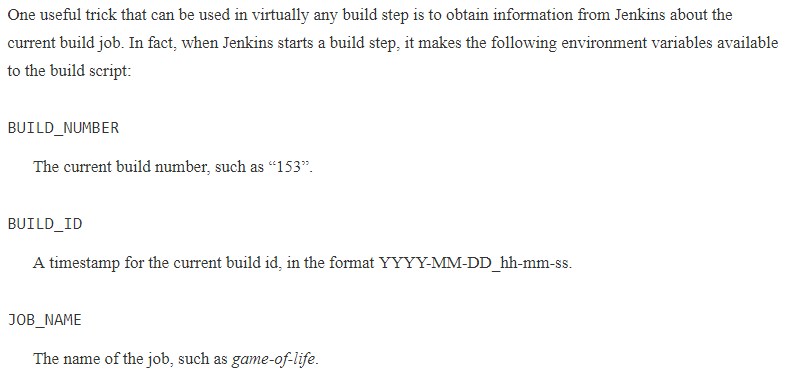
Occasionally you may need to execute a command directly at the Operating System level. Some legacy build processes rely on OS-specific scripts, for example. In other cases, you may need to perform a low-level operation that is most easily done with an OS-level command.

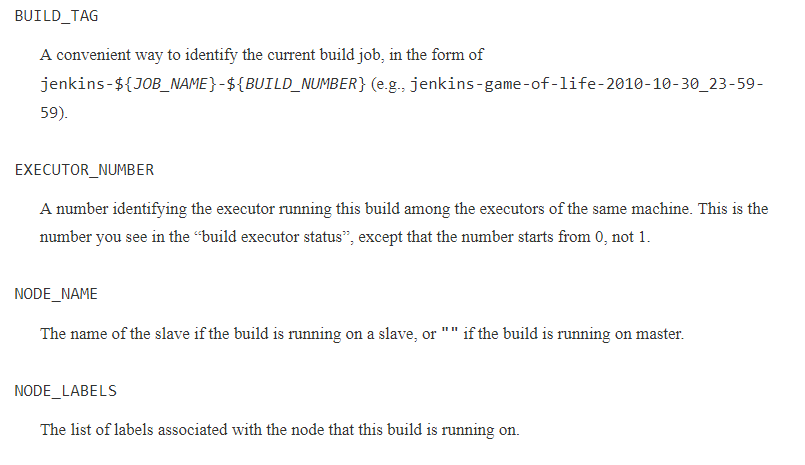
You can do this in Jenkins with the Execute Shell (for Unix) or Execute Windows Batch command (for Windows).

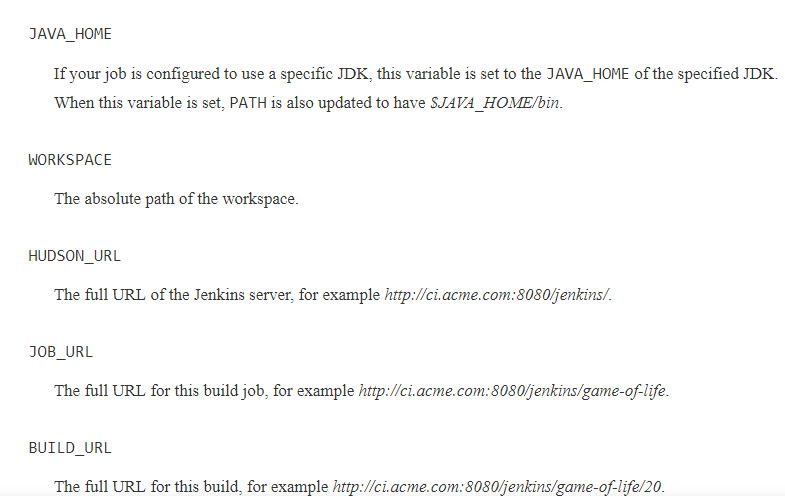


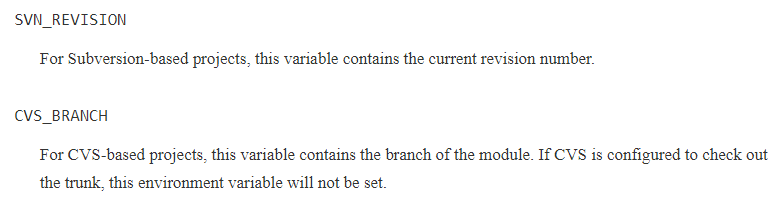
Shell scripts are executed using the -ex option—the commands are printed to the console, as is the resulting output. If any of the executed commands return a nonzero value, the build will fail.

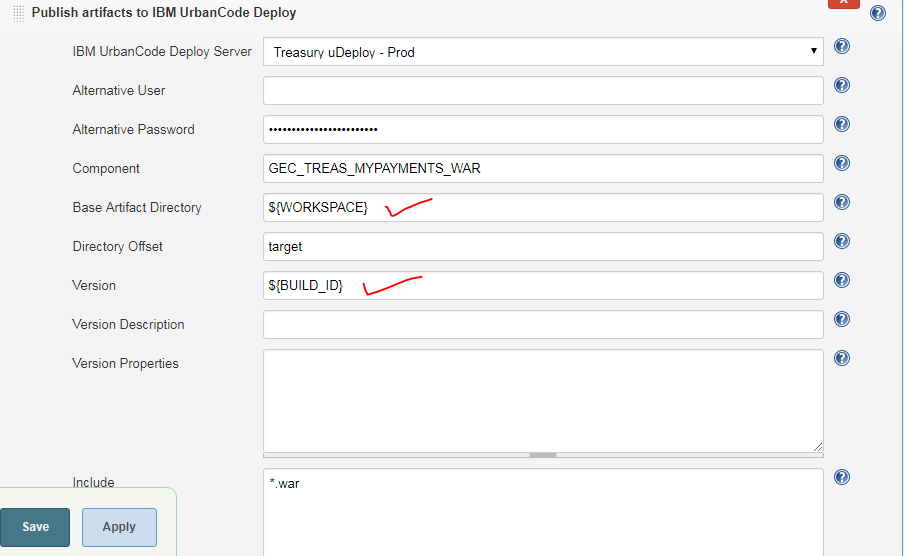
## Using Jenkins Environment Variables in Your Builds

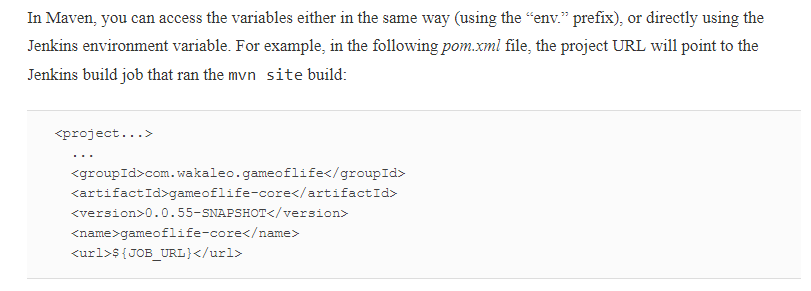






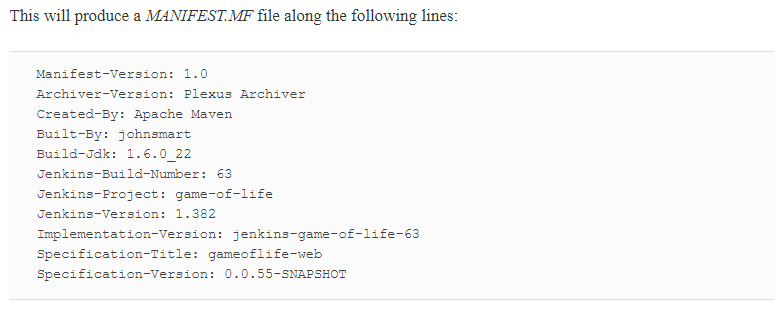






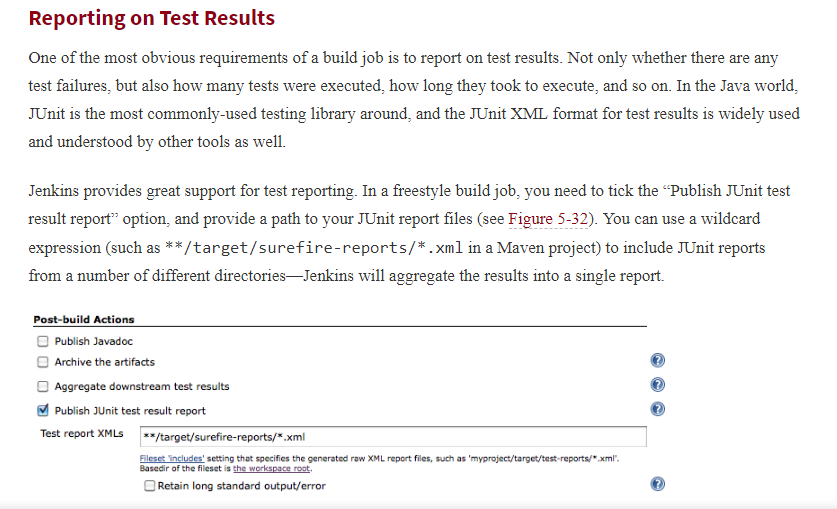
Alternatively, if you are building a web application, you can also use the maven-war-plugin to insert the build job number into the web application manifest, e.g.:

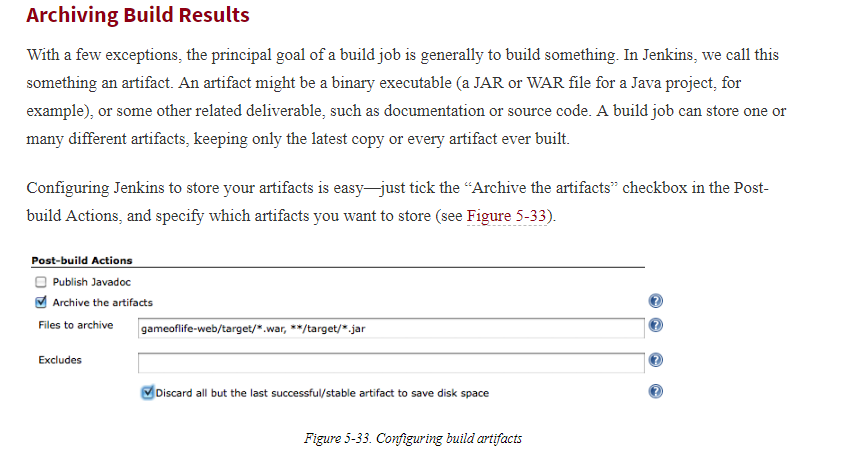


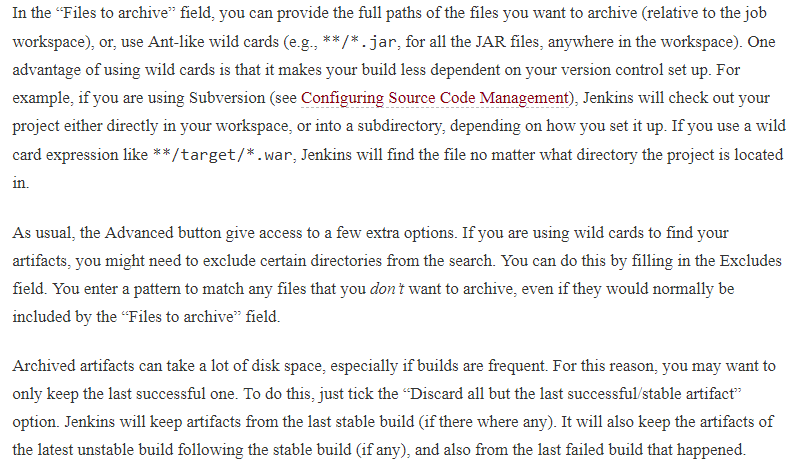


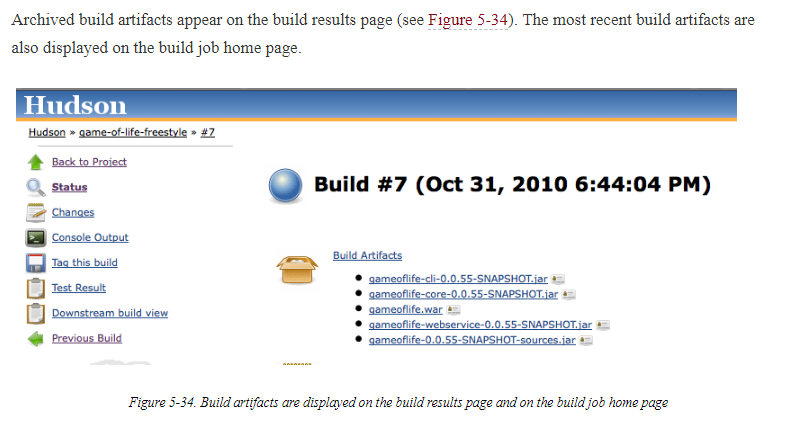
# Post-Build Actions

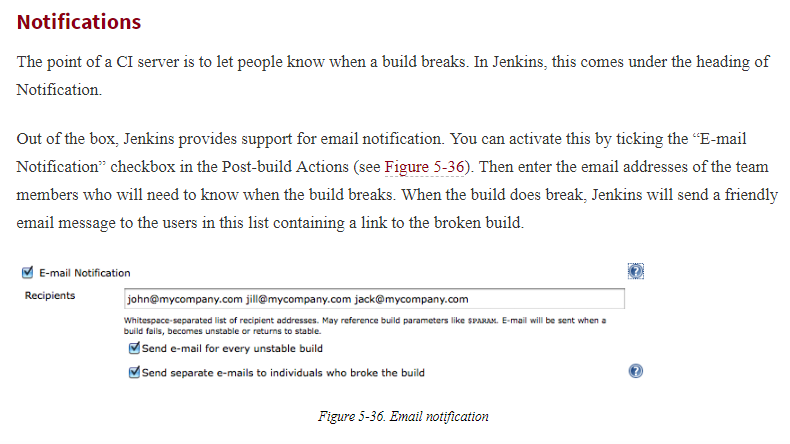
Once the build is completed, there are still a few things you need to look after. You might want to archive some of the generated artifacts, to report on test results, and to notify people about the results. In this section, we look at some of the more common tasks you need to configure after the build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

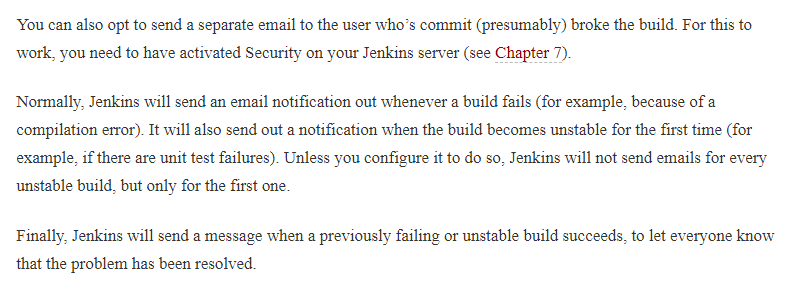












# Working with Maven Build Jobs

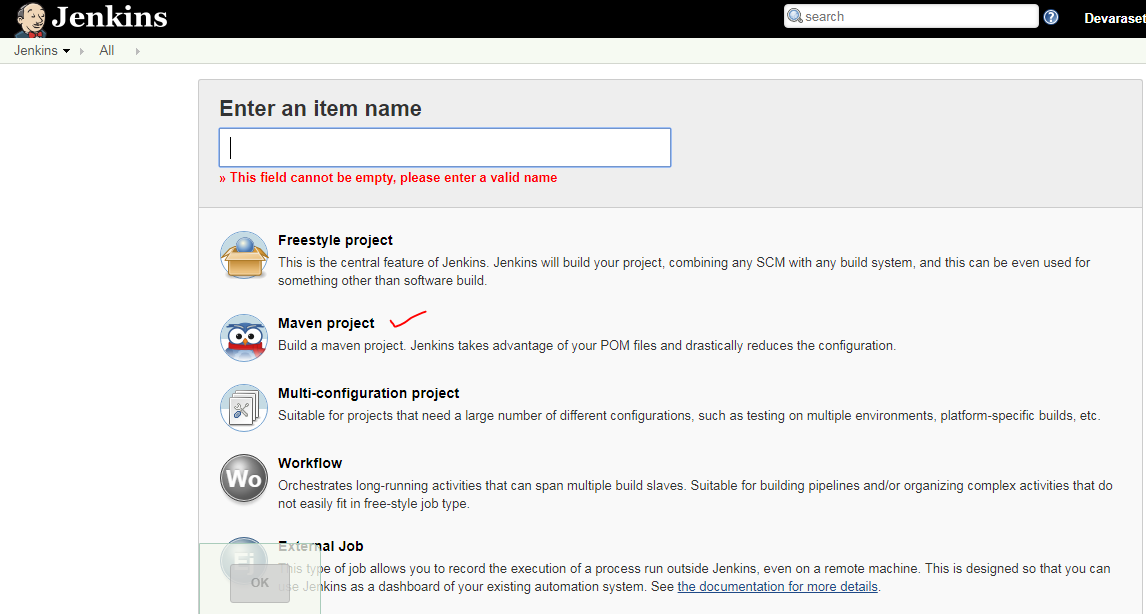
In this section, we will have a look at the other most commonly used build job: Maven 2/3 build jobs.

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 multimodule projects and triggering builds from changes in snapshot dependencies, and make configuration and reporting much simpler.

However, there is a catch: Maven 2/3 build jobs are less flexible than freestyle build jobs, and don’t support multiple build steps within the same build job. Some users also report that large Maven projects tend to run more slowly and use more memory when configured as Maven build jobs rather than as Freestyle ones.

In this section, we will investigate how to configure Maven 2/3 builds, when you can use them, as well as their advantages and limitations

To create a new Maven build job, just choose the “”Build a maven project” option in the New Job page



## Building Whenever a SNAPSHOT Dependency Is Built

At first glance, the Maven 2/3 build job configuration screen is very similar to the one we saw for 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 yourproject 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.

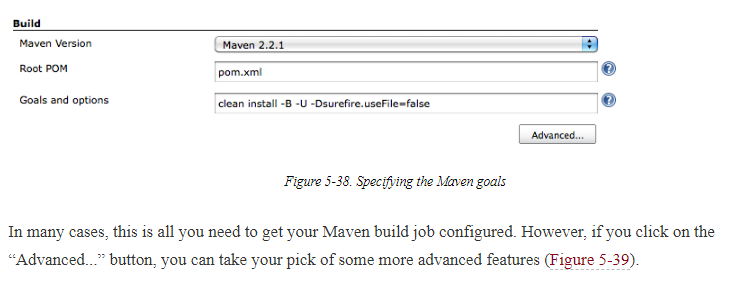
For example, imagine that you are working on a new *game-of-life* web application. You are using Maven for this project, so you can use a Maven build job in Jenkins. Your team is also working on a reusable library called*cooltools*. Since these two projects are being developed by the same team, you are using some of the latest*cooltools* features in the *game-of-life* web application. So you have a SNAPSHOT dependency in the <dependencies> section of your *game-of-life* *pom.xml* file:

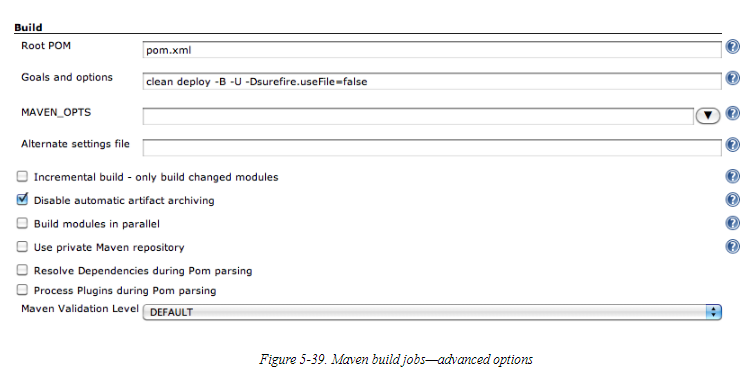


On your Jenkins server, you have set up Maven build jobs for both the cooltools and the game-of-lifeapplications. Since your game-of-life project needs the latest cooltools SNAPSHOT version, you tick the “Build whenever a SNAPSHOT dependency is built” option. This way, whenever the cooltools project is rebuilt, thegame-of-life project will automatically be rebuilt as well.

## 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 (see [Figure 5-38](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch05s09.html#fig-hudson-build-job-maven2)). 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.





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.

