**Chapter 2. Your First Steps with Jenkins**

**Introduction**

In this chapter, we are going to take a quick guided tour through some of Jenkins’s key features. You’ll get to see first-hand just how easy it is to install Jenkins and set up your first Jenkins automated build job. We won’t dwell on the details too much—there are more details to come in the following chapters, as well as a detailed chapter on Jenkins Administration at the end of the book ([Chapter 13](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch13.html)). This chapter is just an introduction. Still, by the end of the chapter, you will also be keeping tabs on test results, generating javadoc and publishing code coverage reports! We’ve got a lot of ground to cover, so let’s get started!

# Preparing Your Environment

There are two ways you can tackle this chapter. You can read through it without touching a keyboard, just to get an overview of what Jenkins is about. Or you can get your hands dirty, and follow along on your own machine.

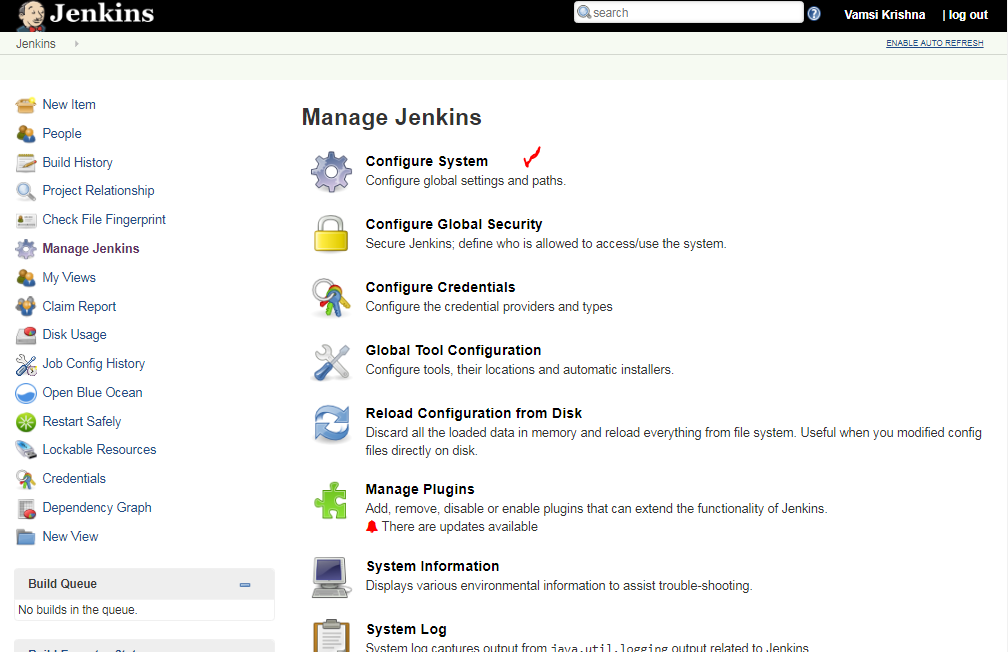
If you do want to follow along at home, you may need to set up some software on your local machine. Remember, the most basic function of any Continuous Integration tool is to monitor source code in a version control system and to fetch and build the latest version of your source code whenever any changes are committed. So you’ll need a version control system. In our case, we’ll be using [Git](http://git-scm.com/). The central source code repository for our simple project is stored on [GitHub](https://github.com/). Don’t worry about messing up this repository with your own changes, though: you’ll be creating your own fork of the repository that you can use as you wish. If you haven’t used Git and/or don’t have an account on GitHub yet, don’t worry, we’ll walk through the basics, and the whole installation process is well documented on the GitHub website. We’ll explain how to set it all up in great detail further on.

In this chapter, we’ll 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, but if you don’t have these installed on your machine, don’t fret! As we will see, Jenkins will install them for you.

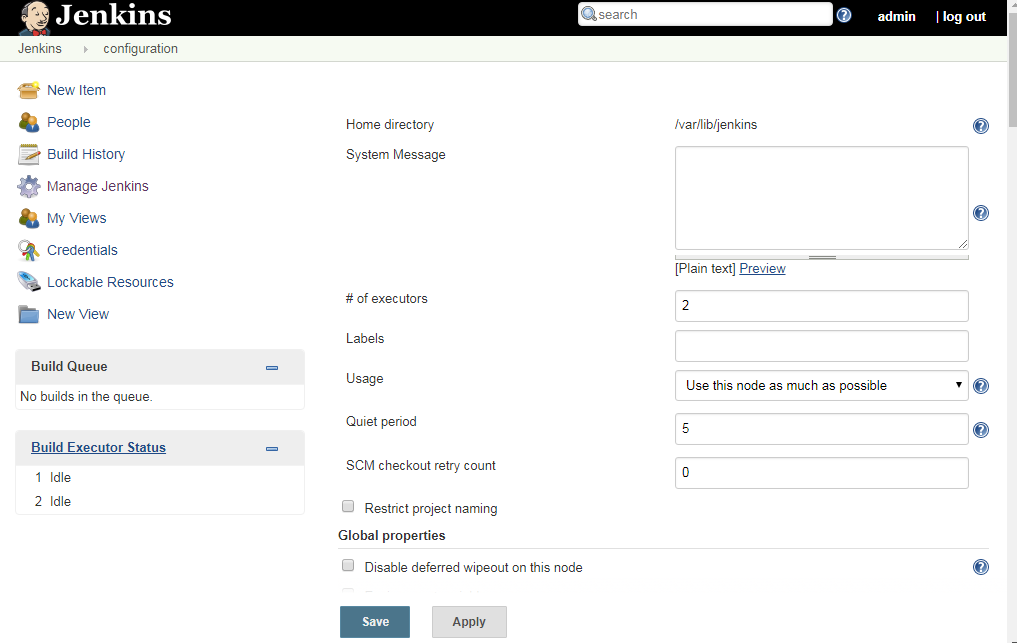
# Configuring the Tools

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 2-7](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s03.html#fig-hudson-start)). 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 2-8](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s04.html#fig-hudson-manage)).



This will take you to Jenkins’s main configuration screen (see [Figure 2-9](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s04.html#fig-hudson-configure)). 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, so you can safely ignore them for now.



For now, you will just need to configure the tools required to build our sample project. The application we will be building is a Java application, built using Maven. So in this case, all we need to do is to set up a recent JDK and Maven installation.

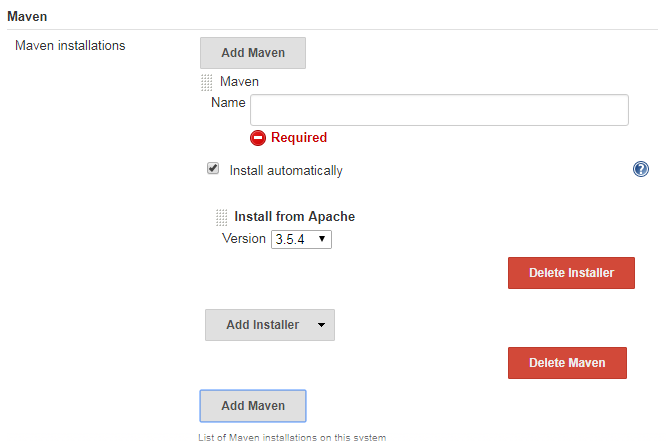
However before we start, take a look at the little blue question mark icons lined to the right of the screen. These are Jenkins’s contextual help buttons. If you are curious about a particular field, click on the help icon next to it and Jenkins will display a very detailed description about what it is and how it works.

## Configuring Your Maven Setup

Our sample project uses Maven, so we will need to install and configure Maven first. Jenkins provides great out-of-the-box support for Maven. Scroll down until you reach the Maven section in the Configure System screen

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 (imaginatively called “Maven 2.2.1” in the example), so that you can refer to it in your build jobs.

For this to work, you need to have an Internet connection. If you are behind a proxy, you’ll need to provide your proxy information—we discuss how to set this up in [Configuring a Proxy](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch04s09.html).



Above picture is from Global Tool Configuration

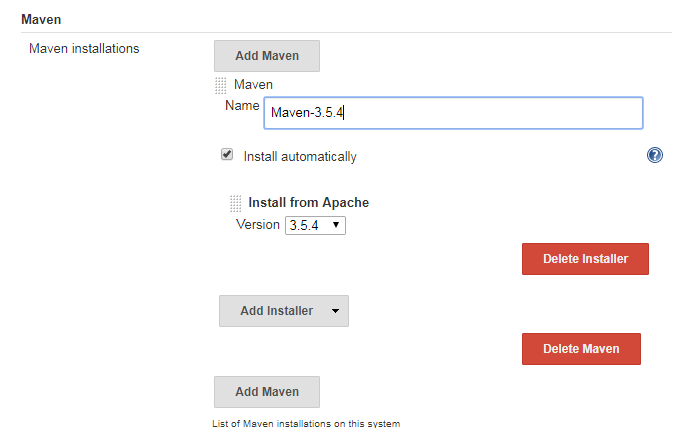
One of the nice things about the Jenkins Maven installation process is how well it works with remote build agents. Later on in the book, we’ll see how Jenkins can also run builds on remote build servers. You can define a standard way of installing Maven for all of your build servers (downloading from the Internet, unzipping a distribution bundle on a shared server, etc.)—all of these options will work when you add a new remote build agent or set up a new build server using this Jenkins configuration.

## Configuring Your Maven Setup

Our sample project uses Maven, so we will need to install and configure Maven first. Jenkins provides great out-of-the-box support for Maven. Scroll down until you reach the Maven section in the Configure System screen (see [Figure 2-10](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s04.html#fig-hudson-maven)).

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 (imaginatively called “Maven 2.2.1” in the example), so that you can refer to it in your build jobs.

For this to work, you need to have an Internet connection. If you are behind a proxy, you’ll need to provide your proxy information—we discuss how to set this up in [Configuring a Proxy](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch04s09.html).



works with remote build agents. Later on in the book, we’ll see how Jenkins can also run builds on remote build servers. You can define a standard way of installing Maven for all of your build servers (downloading from the Internet, unzipping a distribution bundle on a shared server, etc.)—all of these options will work when you add a new remote build agent or set up a new build server using this Jenkins configuration.

## Configuring the JDK

Once you have configured your Maven installation, you will also need to configure a JDK installation (see [Figure 2-11](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s04.html#fig-hudson-jdk)). 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 website](http://www.oracle.com/technetwork/java/index.html) 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.



## 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. However, to keep things simple, we’re not going to worry about notifications just yet.

# Your First Jenkins Build Job

Build jobs are at the heart of the Jenkins build process. Simply put, you can think of a Jenkins build job as a particular task or step in your build process. This may involve simply compiling your source code and running your unit tests. Or you might want a build job to do other related tasks, such as running your integration tests, measuring code coverage or code quality metrics, generating technical documentation, or even deploying your application to a web server. A real project usually requires many separate but related build jobs.

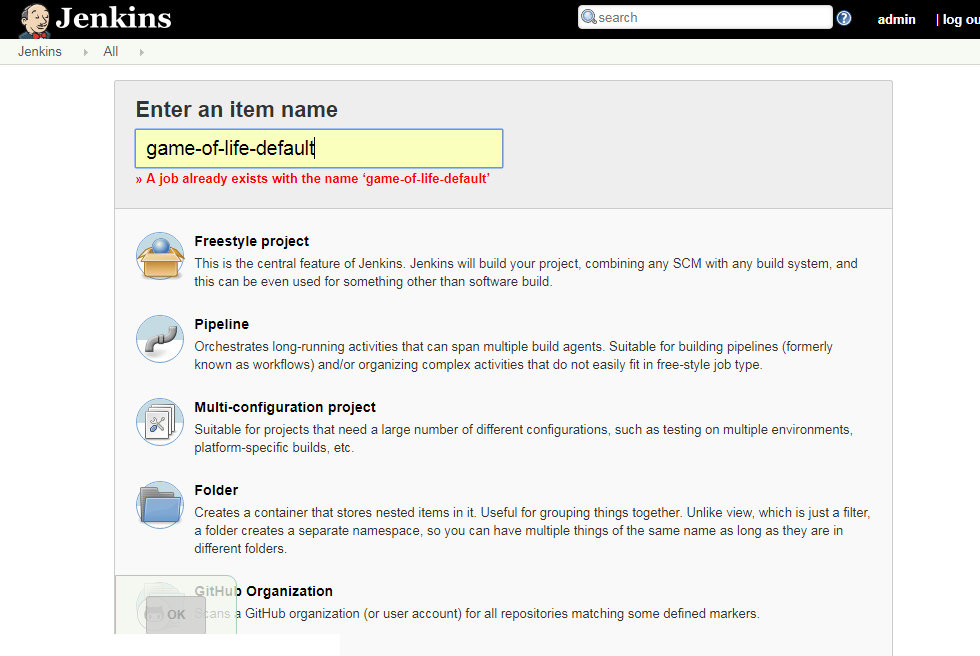
Our sample application is a simple Java implementation of John Conway’s “Game of Life.”[[1](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s05.html#ftn.footnote-game-of-life)] The Game of Life is a mathematical game which takes place on a two dimensional grid of cells, which we will refer to as the Universe. Each cell can be either alive or dead. Cells interact with their direct neighbors to determine whether they will live or die in the next generation of cells. For each new generation of cells, the following rules are applied:

* Any live cell with fewer than two live neighbors dies of underpopulation.
* Any live cell with more than three live neighbors dies of overcrowding.
* Any live cell with two or three live neighbors lives on to the next generation.
* Any dead cell with exactly three live neighbors becomes a live cell.

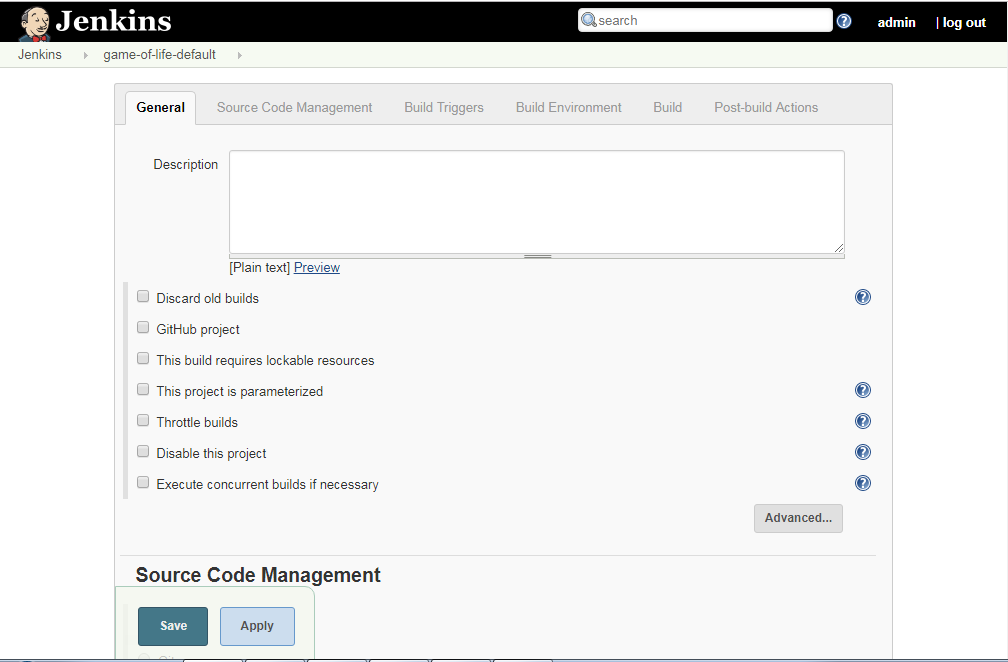
Our application is a Java module, built using Maven, that implements the core business logic of the Game of Life. We’ll worry about the user interfaces later on. For now, let’s see how we can automate this build in Jenkins. If you are not familiar with Maven, or prefer Ant or another build framework—don’t worry! The examples don’t require much knowledge of Maven, and we’ll be looking at plenty of examples of using other build tools later on in the book.

For our first build job, we will keep it simple: we are just going to compile and test our sample application. Click on the New Job link. You should get to a screen similar to [Figure 2-14](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s05.html#fig-first-build-job). Jenkins supports several different types of build jobs. The two most commonly-used are the freestyle builds and the Maven 2/3 builds. The freestyle projects allow you to configure just about any sort of build job: they are highly flexible and very configurable. The Maven 2/3 builds understand the Maven project structure, and can use this to let you set up Maven build jobs with less effort and a few extra features. There are also plugins that provide support for other types of build jobs. Nevertheless, although our project does use Maven, we are going to use a freestyle build job, just to keep things simple and general to start with. So choose “Build a freestyle software project”, as shown in

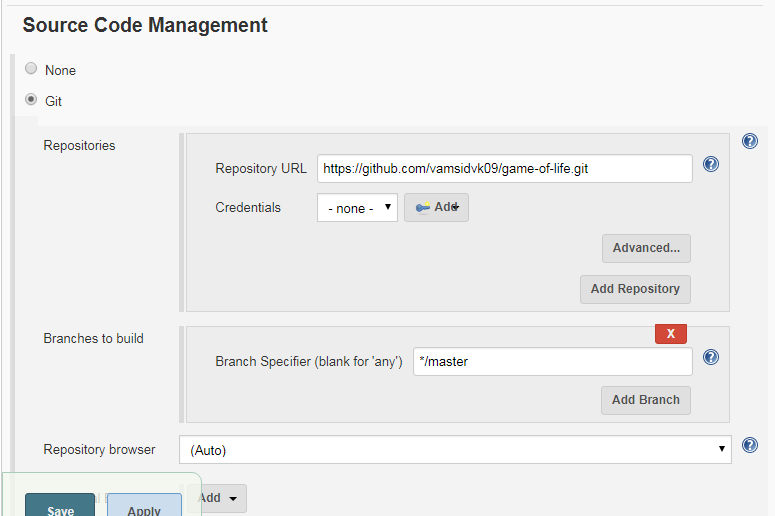
You’ll also need to give your build job a sensible name. In this case, call it gameoflife-default, as it will be the default CI build for our Game of Life project.



Once you click on OK, Jenkins will display the project configuration screen as below

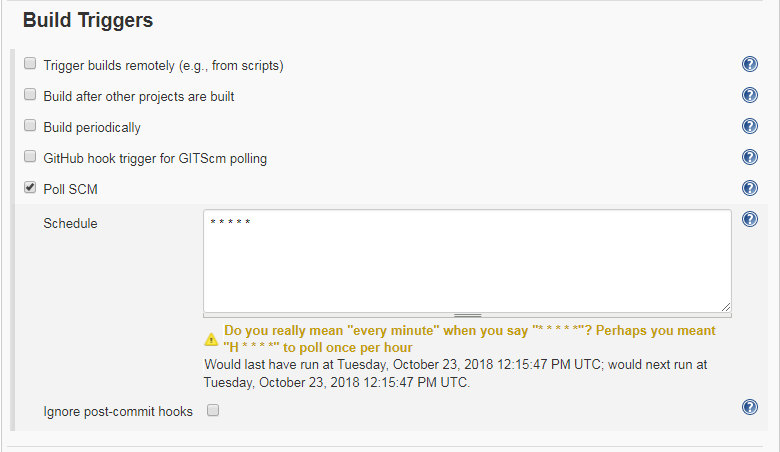


In a nutshell, Jenkins works by checking out the source code of your project and building it in its own workspace. So the next thing you need to do is to tell Jenkins where it can find the source code for your project. You do this in the Source Code Management section, For this project, we will be getting the source code from the GitHub repository



Once we have told Jenkins where to find the source code for our application, we need to tell it how often it should check for updates. We want Jenkins to monitor the repository and start a build whenever any changes have been committed. This is a common way to set up a build job in a Continuous Integration context, as it provides fast feedback if the build fails. Other approaches include building on regular intervals (for example, once a day), requiring a user to kick of the build manually, or even triggering a build remotely using a “post-commit” hook in your SCM.

We configure all of this in the Build Triggers section (see [Figure 2-16](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s05.html#fig-new-job-triggers)). Pick the Poll SCM option and enter “\* \* \* \* \*” (that’s five asterisks separated by spaces) in the Schedule box. Jenkins schedules are configured using the cron syntax, well-known in the Unix world. The cron syntax consists of five fields separated by white space, indicating respectively the minute (0–59), hour (0–23), day of the month (1–31), month (1–12) and the day of the week (0–7, with 0 and 7 being Sunday). The star is a wildcard character which accepts any valid value for that field. So five stars basically means “every minute of every hour of every day.” You can also provide ranges of values: “\* 9-17 \* \* \*” would mean “every minute of every day, between 9am and 5pm.” You can also space out the schedule using intervals: “\*/5 \* \* \* \*” means “every 5 minutes,” for example. Finally, there are some other convenient short-hands, such as “@daily” and “@hourly”.



The next step is to configure the actual build itself. In a freestyle build job, you can break down your build job into a number of build steps. This makes it easier to organize builds in clean, separate stages. For example, a build might run a suite of functional tests in one step, and then tag the build in a second step if all of the functional tests succeed. In technical terms, a build step might involve invoking an Ant task or a Maven target, or running a shell script. There are also Jenkins plugins that let you use additional types of build steps: Gant, Grails, Gradle, Rake, Ruby, MSBuild and many other build tools are all supported.

For now, we just want to run a simple Maven build. Scroll down to the Build section and click on the “Add build step” and choose “Invoke top-level Maven targets” (see [Figure 2-17](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s05.html#fig-new-job-build-step)). Then enter “clean package” in the Goals field. If you are not familiar with Maven, this will delete any previous build artifacts, compile our code, run our unit tests, and generate a JAR file.

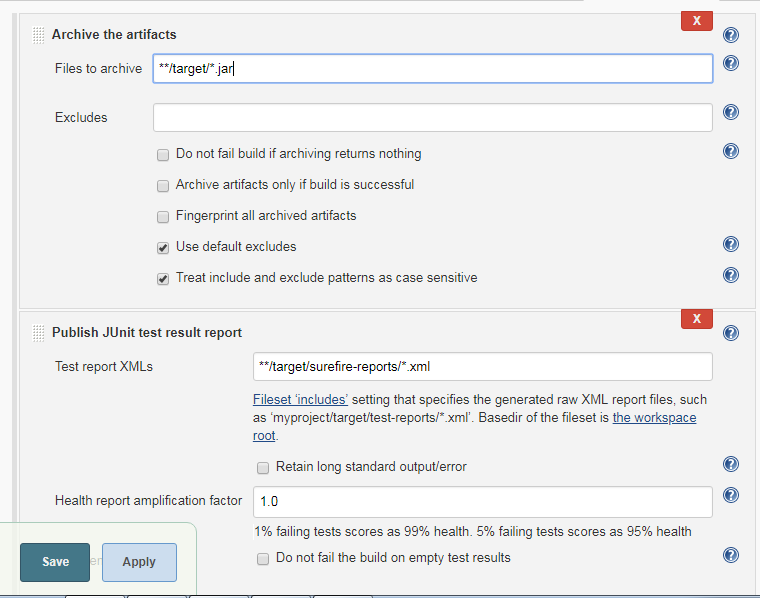


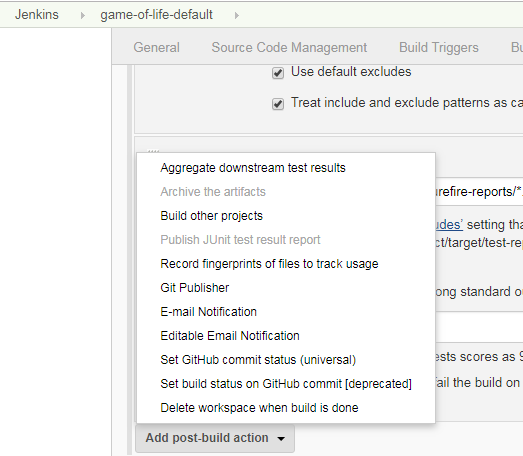
By default, this build job will fail if the code does not compile or if any of the unit tests fail. That’s the most fundamental thing that you’d expect of any build server. But Jenkins also does a great job of helping you display your test results and test result trends.

The de facto standard for test reporting in the Java world is an XML format used by JUnit. This format is also used by many other Java testing tools, such as TestNG, Spock and Easyb. Jenkins understands this format, so if your build produces JUnit XML test results, Jenkins can generate nice graphical test reports and statistics on test results over time, and also let you view the details of any test failures. Jenkins also keeps track of how long your tests take to run, both globally, and per test—this can come in handy if you need to track down performance issues.

So the next thing we need to do is to get Jenkins to keep tabs on our unit tests.

Go to the Post-build Actions section (see [Figure 2-18](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s05.html#fig-new-job-test-reports)) and tick “Publish JUnit test result report” checkbox. When Maven runs unit tests in a project, it automatically generates the XML test reports in a directory called surefire-reports in the target directory. So enter “\*\*/target/surefire-reports/\*.xml” in the “Test report XMLs” field. The two asterisks at the start of the path (“\*\*”) are a best practice to make the configuration a bit more robust: they allow Jenkins to find the target directory no matter how we have configured Jenkins to check out the source code.





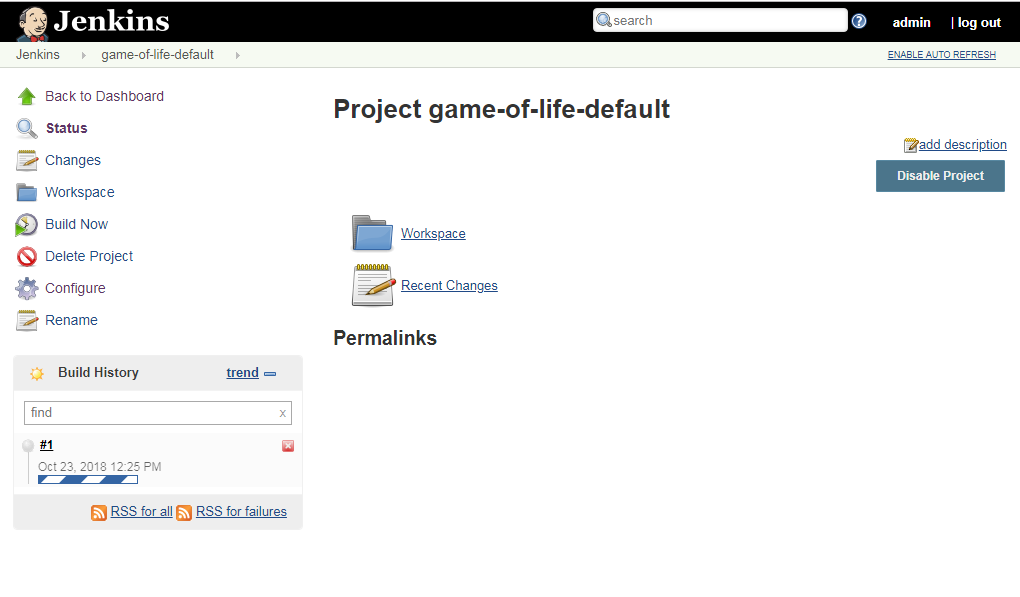
Another thing you often want to do is to archive your build results. Jenkins can store a copy of the binary artifacts generated by your build, allowing you to download the binaries produced by a build directly from the build results page. It will also post the latest binary artifacts on the project home page, which is a convenient way to distribute the latest and greatest version of your application. You can activate this option by ticking the “Archive the artifacts” checkbox and indicating which binary artifacts you want Jenkins to archive. In [Figure 2-18](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s05.html#fig-new-job-test-reports), for example, we have configured Jenkins to store all of the JAR files generated by this build job.

Now we’re done—just click on the Save button at the bottom of the screen. Our build job should now be ready to run. So let’s see it in action!

# Your First Build Job in Action

Once you save your new build job, Jenkins will display the home page for this job (see [Figure 2-19](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s06.html#fig-new-job-running)). This is where Jenkins displays details about the latest build results and the build history.

If you wait a minute or so, the build should kick off automatically—you can see the stripy progress bar in the Build History section in the bottom left hand corner of [Figure 2-19](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s06.html#fig-new-job-running). Or, if you are impatient, you can also trigger the build manually using the Build Now button.

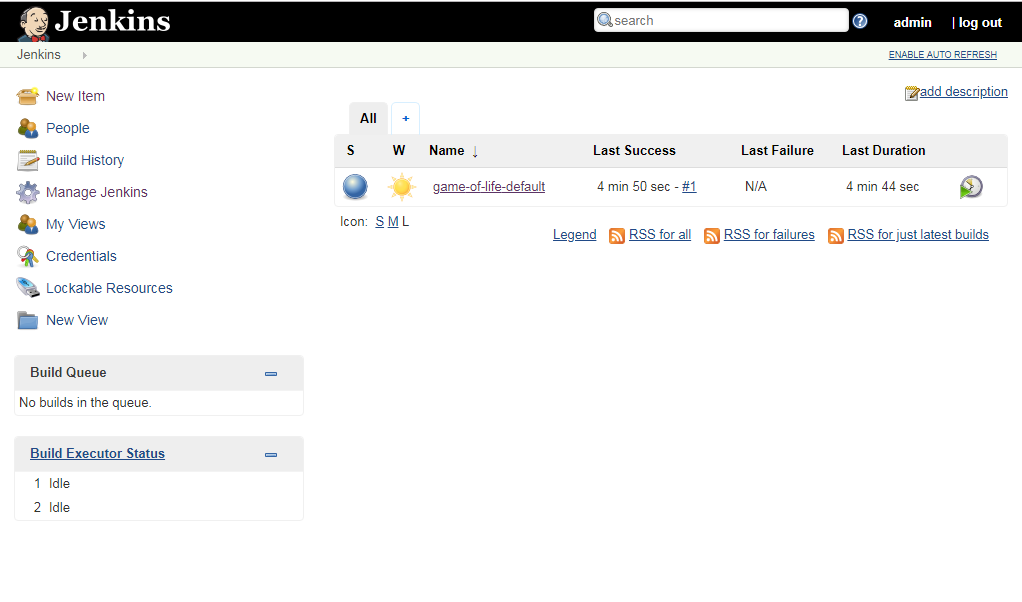


The build will also now figure proudly on your Jenkins server’s home page (see [Figure 2-20](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s06.html#fig-hudson-dashboard)). This page shows a summary of all of your build jobs, including the current build status and general state of heath of each of your builds. It tells you when each build ran successfully for the last time, and when it last failed, and also the result of the last build.

Once of Jenkins’s specialities is the way it lets you get an idea of build behavior over time. For example, Jenkins uses a weather metaphor to help give you an idea of the stability of your builds. Essentially, the more your builds fail, the worse the weather gets. This helps you get an idea of whether a particular broken build is an isolated event, or if the build is breaking on a regular basis, in which case it might need some special attention.

You can also manually trigger a build job here, using the build schedule button (that’s the one that looks a bit like a green play button on top of a clock).

When the build finishes, the ball in the Build History box becomes solid blue. This means the build was a success. Build failures are generally indicated by a red ball. For some types of project, you can also distinguish between a build error (such as a compiler error), indicated by a red ball, and other sorts of build failures, such as unit test failures or insufficient code coverage, which are indicated by a yellow ball. There are also some other details about the latest test results, when the last build was run, and so on. But before we look at the details, let’s get back to the core business model of a Continuous Integration server—kicking off builds when someone changes the code!

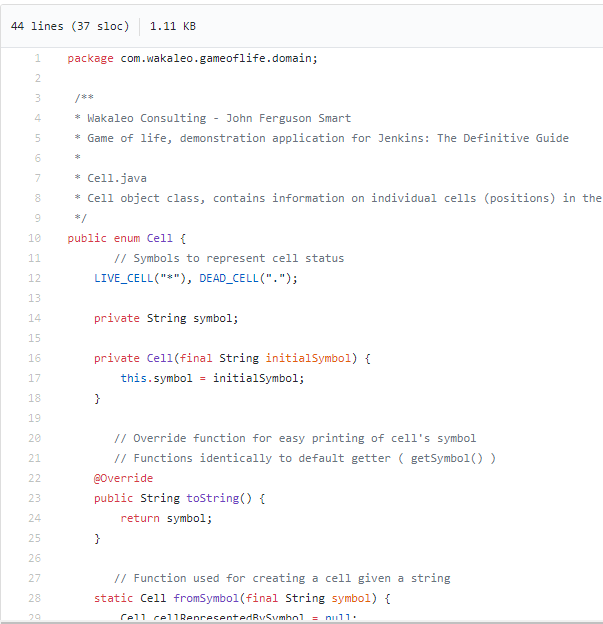


We are going to commit a code change to GitHub and see what happens, using the source code we checked out in [Forking the Sample Repository](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s02.html#sect-git-fork). We now have Jenkins configured to monitor our GitHub fork, so if we make any changes, Jenkins should be able to pick them up.

So let’s make a change. The idea is to introduce a code change that will cause the unit tests to fail. If your Java is a bit rusty, don’t worry, you won’t need to know any Java to be able to break the build—just follow the instructions!

Now in normal development, you would first modify the unit test that describes this behaviour. Then you would verify that the test fails with the existing code, and implement the code to ensure that the test passes. Then you would commit your changes to your version control system, allowing Jenkins to build them. However this would be a poor demonstration of how Jenkins handles unit test failures. So in this example, we will, against all best practices, simply modify the application code directly.

First of all, open the Cell.java file, which you will find in the gameoflife-core/src/main/java/com/wakaleo/gameoflife/domain directory. Open this file in your favorite text editor. You should see something like this:



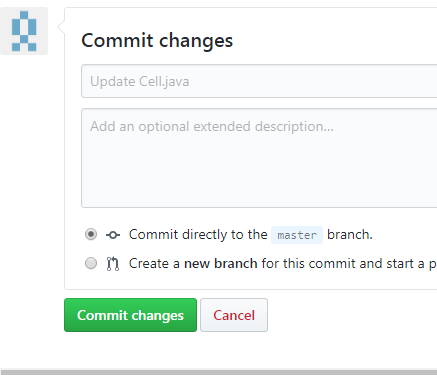
The application can print the state of the grid as a text array. Currently, the application prints our live cells as an asterisk (\*), and dead cells appear as a minus character (–). So a five-by-five grid containing a single living cell in the center would look like this:



Now users have asked for a change to the application—they want pluses (+) instead of stars! So we are going to make a slight change to the Cell class method, and rewrite it as follows (the modifications are in **bold**):

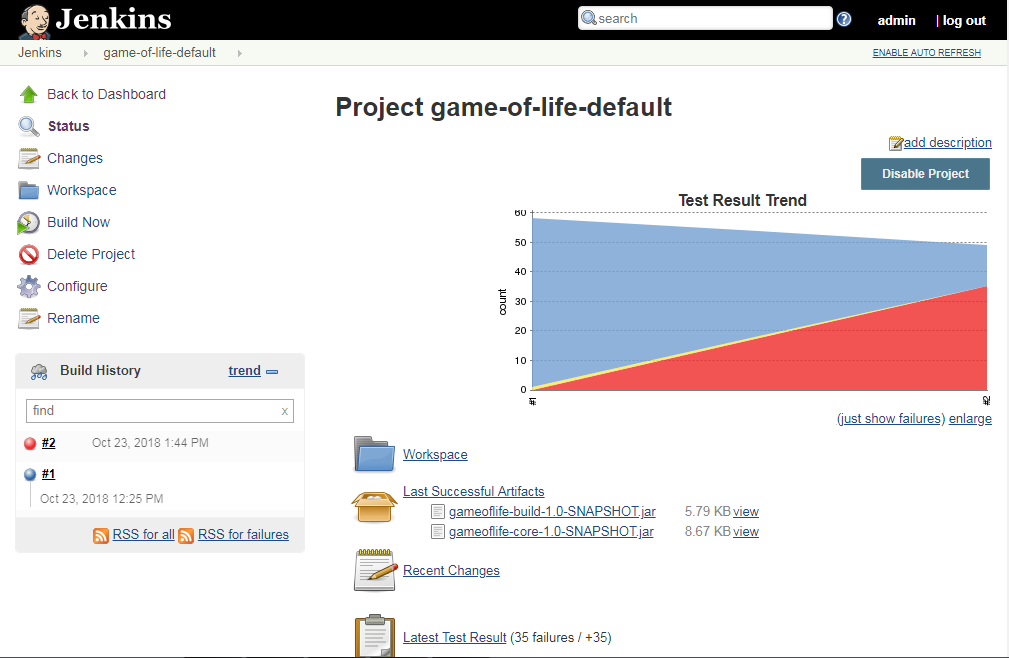


Save this change, and then commit them

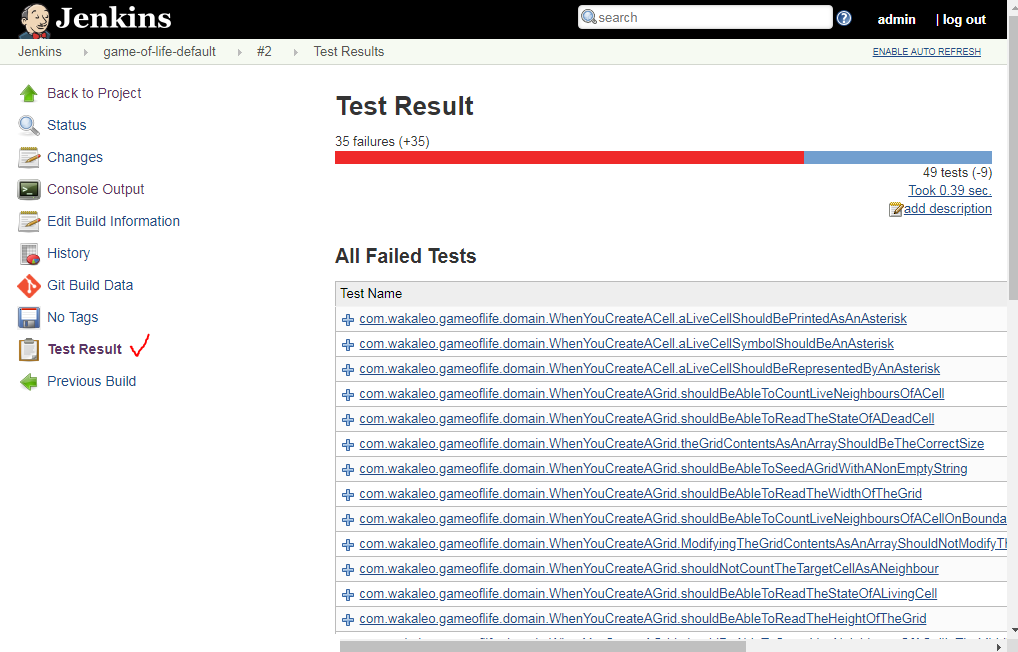


Now go back to the Jenkins web page. After a minute or so, a new build should kick off, and fail. In fact, there are several other places which are affected by this change, and the regression tests related to these features are now failing. On the build job home page, you will see a second build in the build history with an ominous red ball (see [Figure 2-21](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s06.html#fig-new-job-failed))—this tells you that the latest build has failed. ( IF POLL SCM configured)

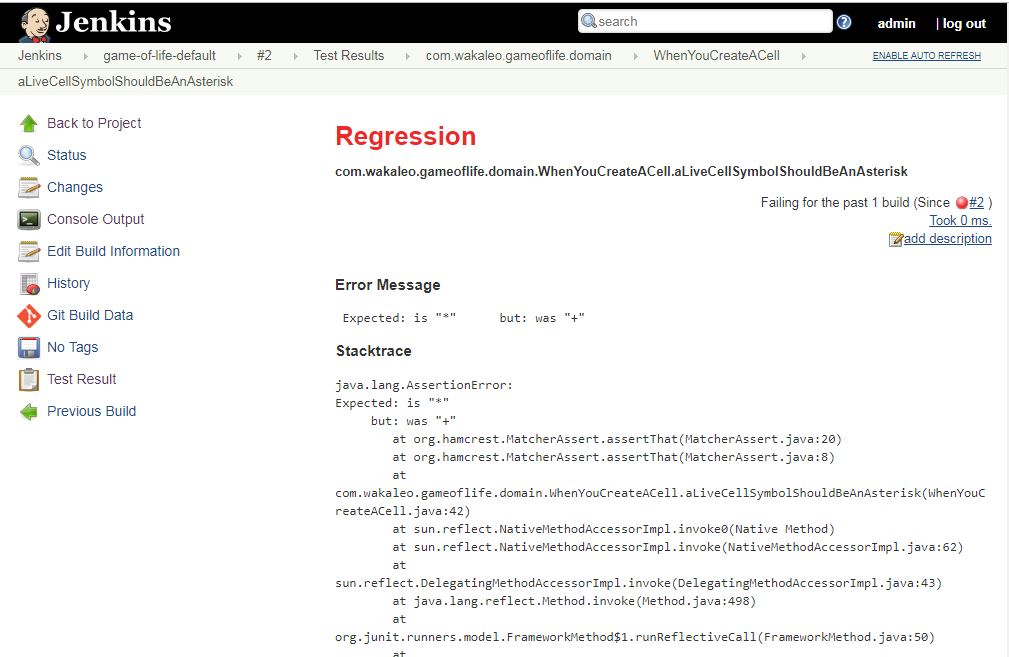
You might also notice some clouds next to the Build History title—this is the same “weather” icon that we saw on the home page, and serves the same purpose—to give you a general idea of how stable your build is over time.



If you click on the new build history entry, Jenkins will give you some more details about what went wrong (see [Figure 2-22](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s06.html#fig-new-job-failure-details)). Jenkins tells us that there were 11 new test failures in this build, something which can be seen at a glance in the Test Result Trend graph—red indicates test failures. You can even see which tests are failing, and how long they have been broken.

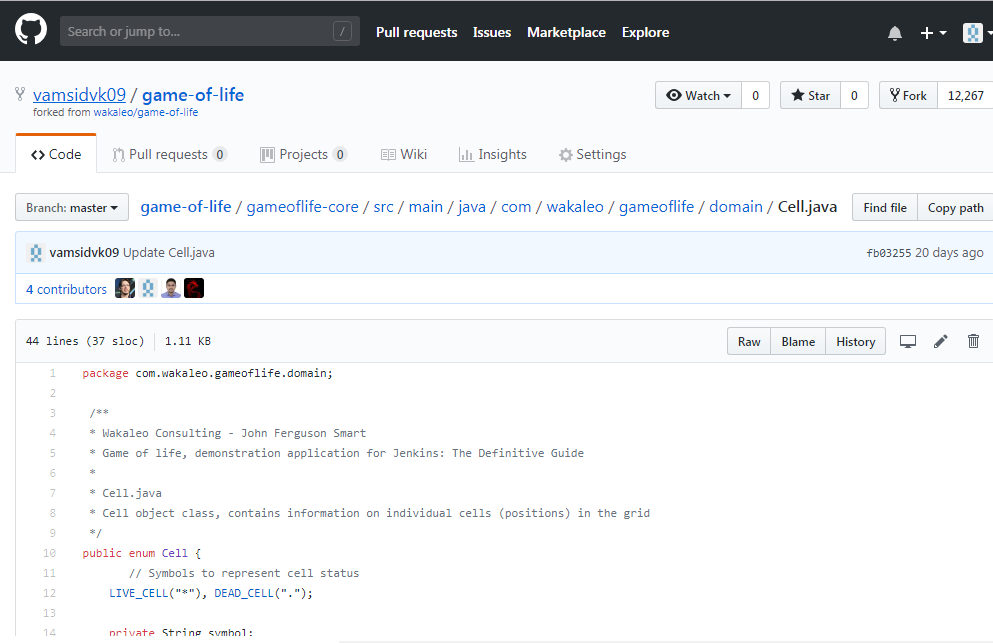


If you want to know exactly what went wrong, that’s easy enough to figure out as well. If you click on the failed test classes, Jenkins brings up the actual details of the test failures (see [Figure 2-23](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s06.html#fig-hudson-new-job-test-failure-details)), which is a great help when it comes to reproducing and fixing the issue.

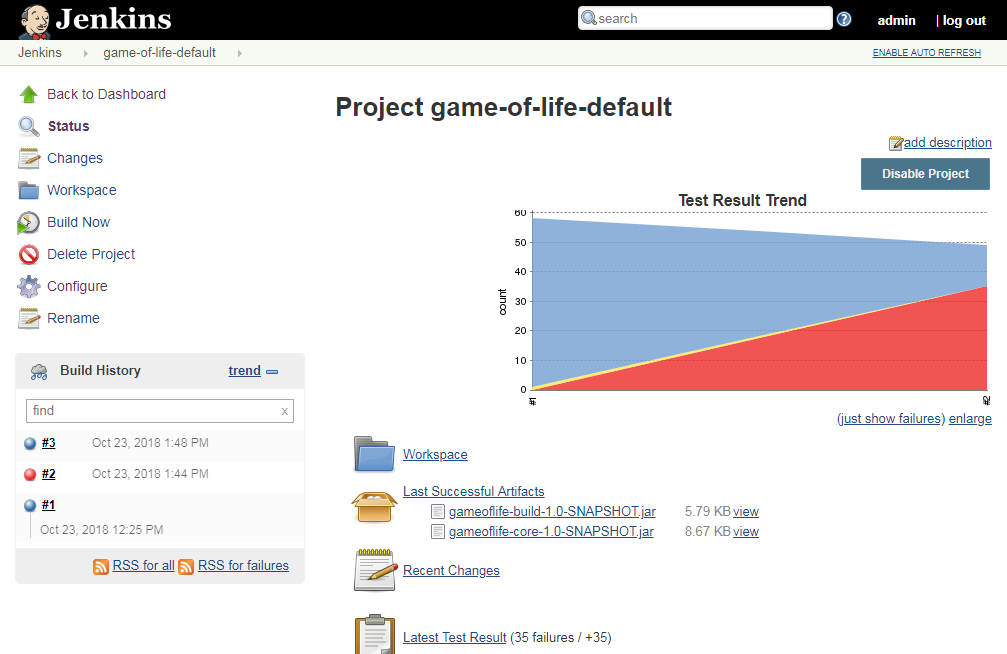


Jenkins displays a host of information about the failed test in a very readable form, including the error message the test produced, the stack trace, how long the test has been broken, and how long it took to run. Often, this in itself is enough to put a developer on the right track towards fixing the issue.

Now let’s fix the build. To make things simple, we’ll just back out our changes and recommit the code in its original state (the end users just changed their mind about the asterisks, anyway). So just undo the changes you made to the Cell class (again, the changes are highlighted in **bold**)



Once you’ve committed these changes, Jenkins should pick them up and kick off a build. Once this is done, you will be able to see the fruit of your work on the build job home page (see [Figure 2-24](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s06.html#fig-new-job-fixed-build))—the build status is blue again and all is well. Also notice the way we are building up a trend graph showing the number of succeeding unit tests over time—this sort of report really is one of Jenkins’s strong points.

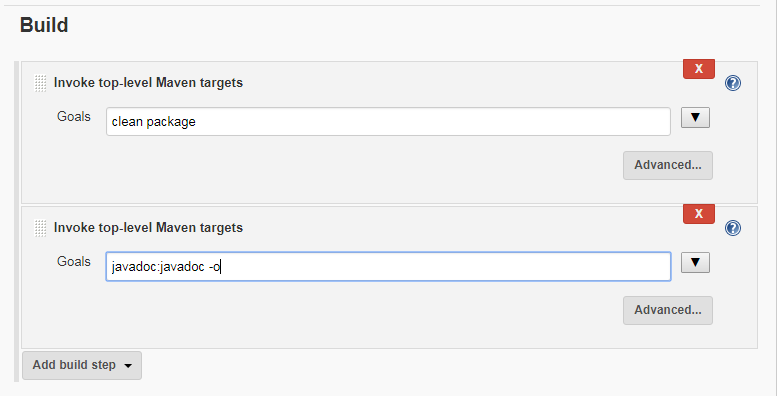


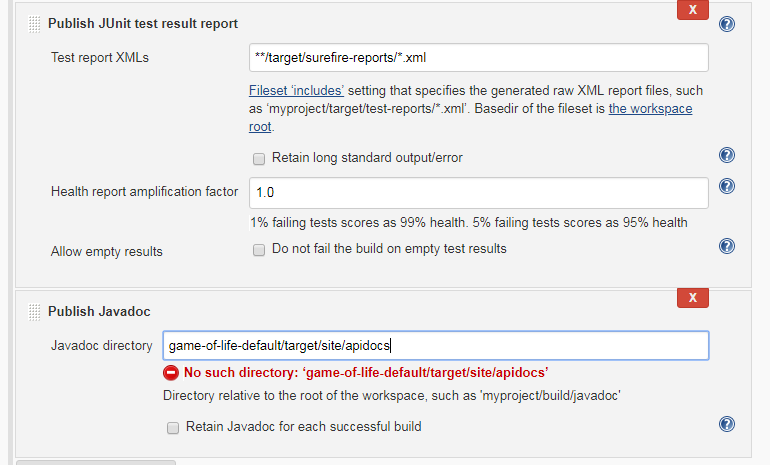
# More Reporting—Displaying Javadocs

For many Java projects, Javadoc comments are an important source of low-level technical documentation. There are even tools, such as UmlGraph, that let you produce Javadoc with embedded UML diagrams to give you a better picture of how the classes fit together in the application. This sort of technical documentation has the advantage of being cheap to produce, accurate and always up-to-date.

Jenkins can integrate Javadoc API documentation directly into the Jenkins website. This way, everyone can find the latest Javadoc easily, in a well known place. Often, this sort of task is performed in a separate build job, but for simplicity we are going to add another build step to the gameoflife-default build job to generate and display Javadoc documention for the Game of Life API.

Start off by going into the “gameoflife-default” configuration screen again. Click on “Add build step”, and add a new build step to “Invoke top level Maven targets” .  In the Goals field, place javadoc:javadoc—this will tell Maven to generate the Javadoc documentation.





Now go to the “Post-build Action” and tick the “Publish Javadoc” checkbox. This project is a multimodule project, so a separate subdirectory is generated for each module (core, services, web and so forth). For this example, we are interested in displaying the documentation for the core module. In the Javadoc directory field, enter **gameoflife-core/target/site/apidocs**—this is where Maven will place the Javadocs it generates for the core module. Jenkins may display an error message saying that this directory doesn’t exist at first. Jenkins is correct—this directory won’t exist until we run the javadoc:javadoc goal, but since we haven’t run this command yet we can safely ignore the message at this stage.

If you tick “Retain Javadoc for each successful build”, Jenkins will also keep track of the Javadocs for previous builds—not always useful, but it can come in handy at times.

Now trigger a build manually. You can do this either from the build job’s home page (using the Build Now link), or directly from the server home page. Once the build is finished, open the build job summary page. You should now see a Javadoc link featuring prominently on the screen—this link will open the latest version of the Javadoc documentation (see [Figure 2-26](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s07.html#fig-hudson-javadoc-results)). You will also see this link on the build details page, where it will point to the Javadoc for that particular build, if you have asked Jenkins to store Javadoc for each build.

# Adding Code Coverage and Other Metrics

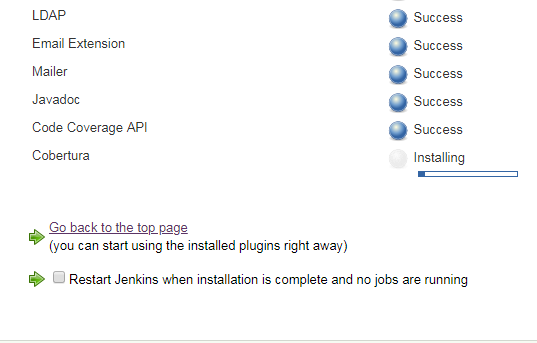
As we mentioned earlier, reporting is one of Jenkins’s strong points. We have seen how easy it is to display test results and to publish Javadocs, but you can also publish a large number of other very useful reports using Jenkins’s plugins.

Plugins are another one of Jenkins’s selling points—there are plugins for doing just about anything, from integrating new build tools or version control systems to notification mechanisms and reporting. In addition, Jenkins plugins are very easy to install and integrate smoothly into the existing Jenkins architecture.

To see how the plugins work, we are going to integrate code coverage metrics using the Cobertura plugin. Code coverage is an indication of how much of your application code is actually executed during your tests—it can be a useful tool in particular for finding areas of code that have not been tested by your test suites. It can also give some indication as to how well a team is applying good testing practices such as Test-Driven Development or Behavior-Driven Development.

[Cobertura](http://cobertura.sourceforge.net/) is an open source code coverage tool that works well with both Maven and Jenkins. Our Maven demonstration project is already configured to record code coverage metrics, so all we need to do is to install the Jenkins Cobertura plugin and generate the code coverage metrics for Jenkins to record and display.

To install a new plugin, go to the Manage Jenkins page and click on the Manage Plugins entry. This will display a list of the available plugins as well as the plugins already installed on your server (see [Figure 2-27](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s08.html#fig-hudson-plugin-manager)). If your build server doesn’t have an Internet connection, you can also manually install a plugin by downloading the plugin file elsewhere and uploading it to your Jenkins installation (just open the Advanced tab in [Figure 2-27](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s08.html#fig-hudson-plugin-manager)), or by copying the plugin to the $JENKINS\_HOME/pluginsdirectory.



In our case, we are interested in the Cobertura plugin, so go to the Available tab and scroll down until you find the Cobertura Plugin entry in the Build Reports section. Click on the checkbox and then click on the Install button at the bottom of the screen.

This will download and install the plugin for you. Once it is done, you will need to restart your Jenkins instance to see the fruits of your labor. When you have restarted Jenkins, go back to the Manage Plugins screen and click on the Installed tab—there should now be a Cobertura Plugin entry in the list of installed plugins on this page.

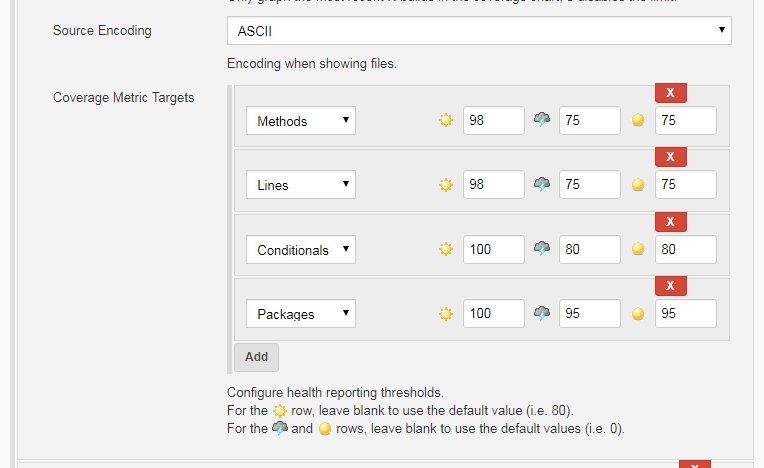
Once you have made sure the plugin was successfully installed, go to the configuration page for the *gameoflife-default* build job.

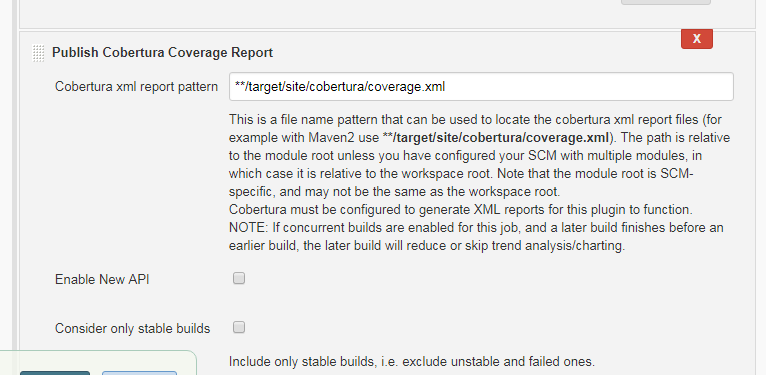
To set up code coverage metrics in our project, we need to do two things. First we need to generate the Cobertura coverage data in an XML form that Jenkins can use; then we need to configure Jenkins to display the coverage reports.

Our Game of Life project already has been configured to generate XML code coverage reports if we ask it. All you need to do is to run mvn cobertura:cobertura to generate the reports in XML form. Cobertura can also generate HTML reports, but in our case we will be letting Jenkins take care of the reporting, so we can save on build time by not generating the For this example, for simplicity, we will just add the cobertura:cobertura goal to the second build step (see [Figure 2-28](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s08.html#fig-hudson-cobertura-build-step)). You could also add a new build step just for the code coverage metrics. In a real-world project, code quality metrics like this are typically placed in a distinct build job, which is run less frequently than the default build.

Next, we need to tell Jenkins to keep track of our code coverage metrics. Scroll down to the “Post-build Actions” section. You should see a new checkbox labeled Publish Cobertura Reports. Jenkins will often add UI elements like this when you install a new plugin. When you tick this box, Jenkins will display the configuration options for the Cobertura plugin that we installed earlier

Like most of the code-quality related plugins in Jenkins, the Cobertura plugin lets you fine-tune not only the way Jenkins displays the report data, but also how it interprets the data. In the Coverage Metrics Targets section, you can define what you consider to be the minimum acceptable levels of code coverage. In [Figure 2-29](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s08.html#fig-hudson-cobertura-config), we have configured Jenkins to list any builds with less than 50% test coverage as “unstable” (indicated by a yellow ball), and notify the team accordingly.





This fine-tuning often comes in handy in real-world builds. For example, you may want to impose a special code coverage constraint in release builds, to ensure high code coverage in release versions. Another strategy that can be useful for legacy projects is to gradually increase the minimum tolerated code coverage level over time. This way you can avoid having to retro-fit unit tests on legacy code just to raise the code coverage, but you do encourage all new code and bug fixes to be well tested.

Now trigger a build manually. The first time you run the build job with Cobertura reporting activated, you will see coverage statistics for your build displayed on the build home page, along with a Coverage Report link when you can go for more details (see [Figure 2-30](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch02s08.html#fig-hudson-initial-coverage-report)). The Cobertura report shows different types of code coverage for the build we just ran. Since we have only run the test coverage metrics once, the coverage will be displayed as red and green bars.