**Chapter 3. Installing Jenkins**

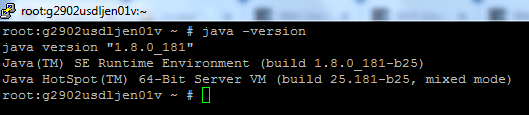
**Introduction**

One of the first things you will probably notice about Jenkins is how easy it is to install. Indeed, in less than five minutes, you can have a Jenkins server up and running. However, as always, in the real world, things aren’t always that simple, and there are a few details you should take into account when installing your Jenkins server for production use. In this chapter, we look at how to install Jenkins onto both your local machine and onto a fully fledged build server. We will also look at how to take care of your Jenkins installation once it’s up and running, and how to perform basic maintenance tasks such as backups and upgrades.

# Downloading and Installing Jenkins

Jenkins is easy to install, and can run just about anywhere. You can run it either as a stand-alone application, or deployed on a conventional Java application server such as Tomcat or JBoss. This first option makes it easy to install and try out on your local machine, and you can be up and running with a bare-bones installation in a matter of minutes.

Since Jenkins is a Java application, you will need a recent version of Java on your machine. More precisely, you will need at least Java 5. In fact, on your build server, you will almost certainly need the full features of the Java Development Kit (JDK) 5.0 or better to execute your builds. If you’re not sure, you can check the version of Java on your machine by executing the java -version command:



# Preparing a Build Server for Jenkins

Installing Jenkins on your local development machine is one thing, but installing Jenkins on a proper build server deserves a little more forethought and planning.

Before you start your installation, the first thing you will need is a build server. To work well, Jenkins needs both processor power and memory. Jenkins itself is a relatively modest Java web application. However, in most configurations, at least some of the builds will be run on the principal build server. Builds tend to be both memory and processor-intensive operations, and Jenkins can be configured to run several builds in parallel. Depending on the number of build jobs you are managing, Jenkins will also need memory of its own for its own internal use. The amount of memory required will depend largely on the nature of your builds, but memory is cheap these days (at least in non-hosted environments), and it’s best not to be stingy.

A build server also needs CPU horsepower. As a rule of thumb, you will need one processor per parallel build, though, in practice, you can capitalize on I/O delays to do a little better than this. It is also in your best interest to dedicate your build server as much as possible to the task of running continuous builds. In particular, you should avoid memory or CPU-intensive applications such as test servers, heavily-used enterprise applications, enterprise databases such as Oracle, enterprise mail servers, and so on.

One very practical option available in many organizations today is to use a virtual machine. This way, you can choose the amount of memory and number of processors you think appropriate for your initial installation, and easily add more memory and processors later on as required. However, if you are using a virtual machine, make sure that it has enough memory to support the maximum number of parallel builds you expect to be running. The memory usage of a Continuous Integration server is best described as spiky—Jenkins will be creating additional JVMs as required for its build jobs, and these need memory.

Another useful approach is to set up multiple build machines. Jenkins makes it quite easy to set up “slaves” on other machines that can be used to run additional build jobs. The slaves remain inactive until a new build job is requested—then the main Jenkins installation dispatches the build job to the slave and reports on the results. This is a great way to absorb sudden spikes of build activity, for example just before a major release of your principal product. It is also a useful strategy if certain heavy-weight builds tend to “hog” the main build server—just put them on their own dedicated build agent! We will look at how to do this in detail later on in the book.

If you are installing Jenkins on a Linux or Unix build server, it is a good idea to create a special user (and user group) for Jenkins. This makes it easier to monitor at a glance the system resources being used by the Jenkins builds, and to troubleshoot problematic builds in real conditions. The native binary installation packages discussed below do this for you. If you did not use one of these, you can create a dedicated Jenkins user from the command line as shown here:

$ **sudo groupadd build**

$ **sudo useradd --create-home --shell /bin/bash --groups build jenkins**

In most environments, you will need to configure Java correctly for this user. For example, you can do this by defining the JAVA\_HOME and PATH variables in the *.bashrc* file, as shown here:

export JAVA\_HOME=/usr/local/java/jdk1.6.0

export PATH=$JAVA\_HOME/bin:$PATH

You will now be able to use this user to run Jenkins in an isolated environment.

Before we install Jenkins, however, there are some things you need to know about how Jenkins stores its data. Indeed, no matter where you store the Jenkins WAR file, Jenkins keeps all its important data in a special separate directory called the Jenkins home directory. Here, Jenkins stores information about your build server configuration, your build jobs, build artifacts, user accounts, and other useful information, as well as any plugins you may have installed. The Jenkins home directory format is backward compatible across versions, so you can freely update or reinstall your Jenkins executable without affecting your Jenkins home directory.

Needless to say, this directory will need a lot of disk space

You can force Jenkins to use a different directory as its home directory by defining the JENKINS\_HOMEenvironment variable. You may need to do this on a build server to conform to local directory conventions or to make your system administrator happy. For example, if your Jenkins WAR file is installed in */usr/local/jenkins*, and the Jenkins home directory needs to be in the */data/jenkins* directory, you might write a startup script along the following lines:

export JENKINS\_BASE=/usr/local/jenkins

export JENKINS\_HOME=/var/jenkins-data

java -jar ${JENKINS\_BASE}/jenkins.war

# Installing Jenkins on Redhat, Fedora, or CentOS

There are also native binary packages available for Redhat, Fedora, and CentOS. First you need to set up the repository as follows:

$ **sudo wget -O /etc/yum.repos.d/jenkins.repo \**

**http://jenkins-ci.org/redhat/jenkins.repo**

$ **sudo rpm --import http://pkg.jenkins-ci.org/redhat/jenkins-ci.org.key**

On a fresh installation, you may need to install the JDK:

$ **sudo yum install java-1.6.0-openjdk**

Next, you can install the package as shown here:

$ **sudo yum install jenkins**

This will install the latest version of Jenkins into the /usr/lib/jenkins directory. The default Jenkins home directory will be in /var/lib/jenkins.

Now you can start Jenkins using the service command:

$ **sudo service jenkins start**

Jenkins will now be running on the default port of 8080 (<http://localhost:8080/>).

Jenkins’s configuration parameters are placed in the /etc/sysconfig/jenkins file. However at the time of writing the configuration options are more limited than those provided by the Ubuntu package: you can define the HTTP port using the JENKINS\_PORT parameter, for example, but to specify an application context you need to modify the startup script by hand. The principal configuration options are listed here:

JENKINS\_JAVA\_CMD

The version of Java you want to use to run Jenkins

JENKINS\_JAVA\_OPTIONS

Command-line options to pass to Java, such as memory options

JENKINS\_PORT

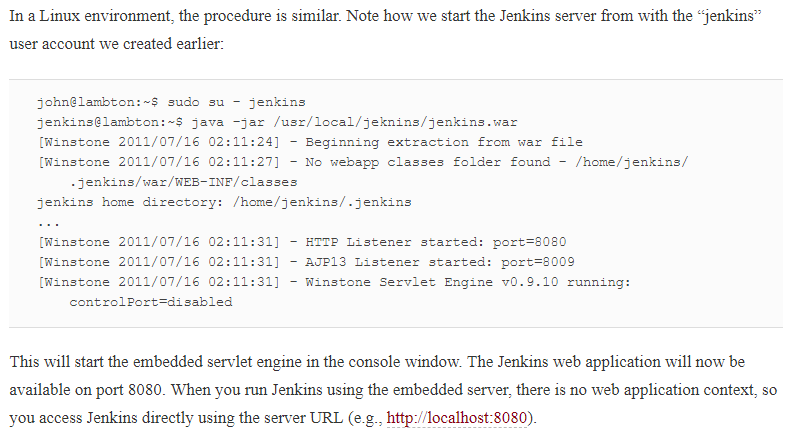
The port that Jenkins will to run on

# Running Jenkins as a Stand-Alone Application

You can run the Jenkins server in one of two ways: either as a stand-alone application, or deployed as a standard web application onto a Java Servlet container or application server such as Tomcat, JBoss, or GlassFish. Both approaches have their pros and cons, so we will look at both here.

Jenkins comes bundled as a WAR file that you can run directly using an embedded servlet container. Jenkins uses the lightweight Winstone servlet engine to allow you to run the server out of the box, without having to configure a web server yourself. This is probably the easiest way to get started, allowing you to be up and running with Jenkins in a matter of minutes. It is also a very flexible option, and provides some extra features unavailable if you deploy Jenkins to a conventional application server. In particular, if you are running Jenkins as a stand-alone server, you will be able to install plugins and upgrades on the fly, and restart Jenkins directly from the administration screens.

To run Jenkins using the embedded servlet container, just go to the command line and type the following:



To stop Jenkins, just press Ctrl-C

By default, Jenkins will run on the 8080 port. If this doesn’t suit your environment, you can specify the port manually, using the --httpPort option:

$ java -jar jenkins.war --httpPort=8081

In a real-world architecture, Jenkins may not be the only web application running on your build server. Depending on the capacity of your server, Jenkins may have to cohabit with other web applications or Maven repository managers, for example. If you are running Jenkins along side another application server, such as Tomcat, Jetty, or GlassFish, you will also need to override the ajp13 port, using the --ajp13Port option

$ **java -jar jenkins.war --httpPort=8081 --ajp13Port=8010**

Some other useful options are:

--prefix

This option lets you define a context path for your Jenkins server. By default Jenkins will run on the port 8080 with no context path ([http://localhost:8080](http://localhost:8080/)). However, if you use this option, you can force Jenkins to use whatever context path suits you, for example:

$ **java -jar jenkins.war --prefix=jenkins**

--daemon

If you are running Jenkins on a Unix machine, you can use this option to start Jenkins as a background task, running as a unix daemon.

--logfile

By default, Jenkins writes its logfile into the current directory. However, on a server, you often need to write your log files into a predetermined directory. You can use this option to redirect your messages to some other file:

$ **java -jar jenkins.war --logfile=/var/log/jenkins.log**

Stopping Jenkins using Ctrl-C is a little brutal, of course—in practice, you would set up a script to start and stop your server automatically.

Uploaded the Jenkins stop & start scripts for reference in github

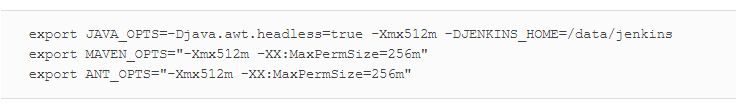
# Memory Considerations

Continuous Integration servers use a lot of memory. This is the nature of the beast—builds will consume memory, and multiple builds being run in parallel will consume still more memory. So you should ensure that your build server has enough RAM to cope with however many builds you intend to run simultaneously.

Jenkins naturally needs RAM to run, but if you need to support a large number of build processes, it is not enough just to give Jenkins a lot of memory. In fact Jenkins spans a new Java process each time it kicks off a build, so during a large build, the build process needs the memory, not Jenkins.

You can define build-specific memory options for your Jenkins build jobs—we will see how to do this later on in the book. However if you have a lot of builds to maintain, you might want to define the JAVA\_OPTS,MAVEN\_OPTS and ANT\_OPTS environment variables to be used as default values for your builds. TheJAVA\_OPTS options will apply for the main Jenkins process, whereas the other two options will be used when Jenkins kicks off new JVM processes for Maven and Ant build jobs respectively.

Here is an example of how these variables might be configured on a Unix machine in the .profile file:

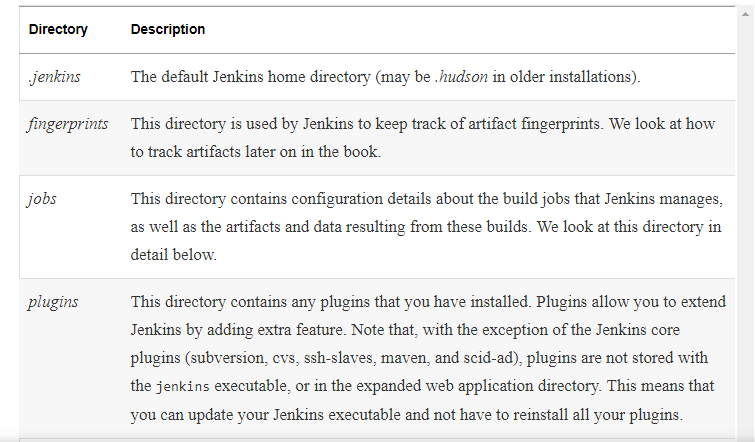


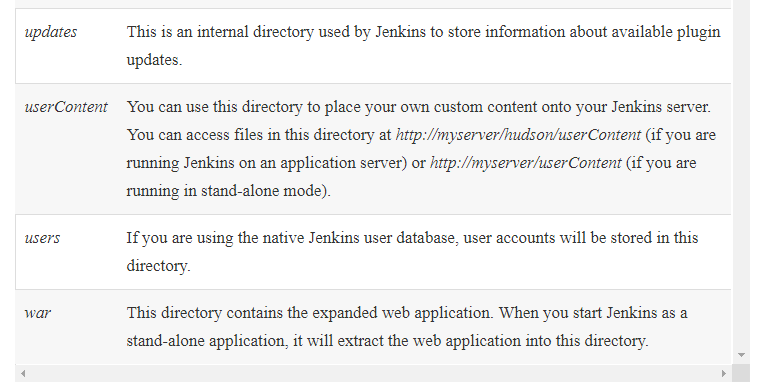
# What’s in the Jenkins Home Directory

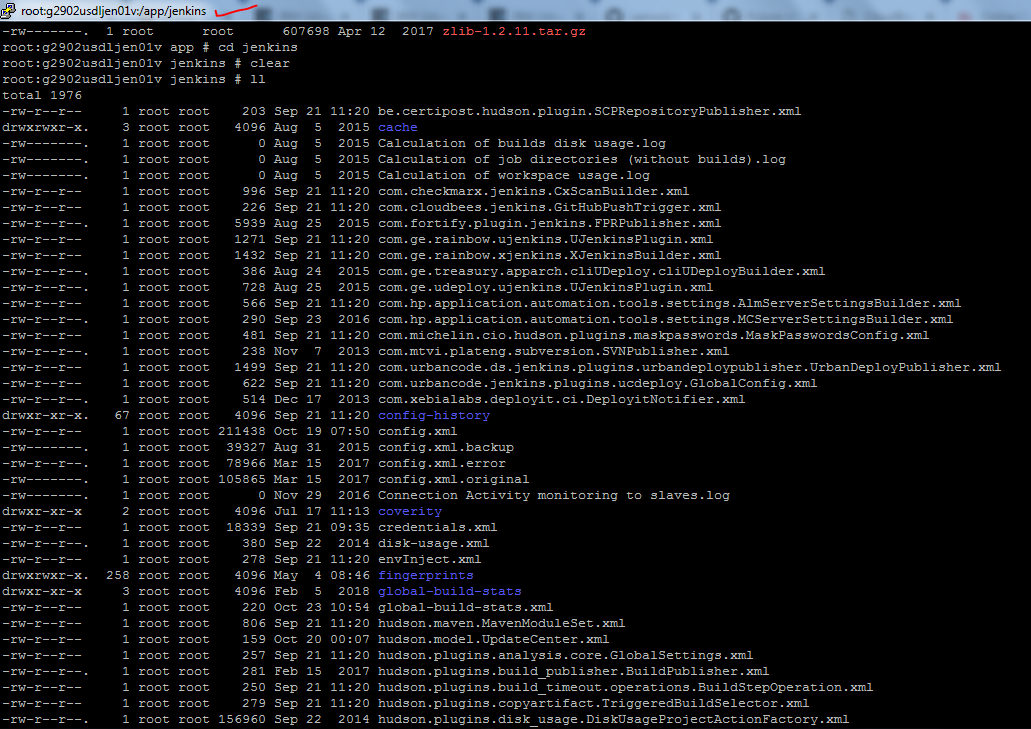
The Jenkins home directory contains all the details of your Jenkins server configuration, details that you configure in the Manage Jenkins screen. These configuration details are stored in the form of a set of XML files. Much of the core configuration, for example, is stored in the config.xml file. Other tools-specific configuration is stored in other appropriately-named XML files: the details of your Maven installations, for example, are stored in a file called hudson.tasks.Maven.xml. You rarely need to modify these files by hand, though occasionally it can come in handy.

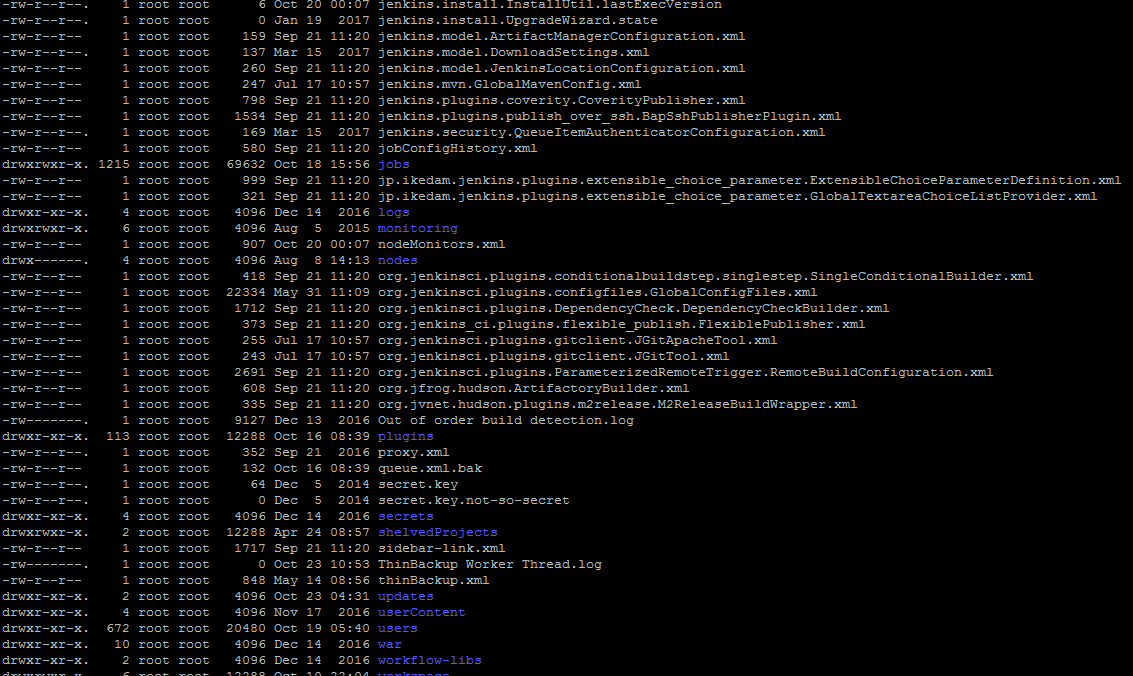
The Jenkins home directory also contains a number of subdirectories (see [Figure 3-7](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch03s13.html#fig-hudson-home-directory)). Not all of the files and directories will be present after a fresh installation, as some are created when required by Jenkins. And if you look at an existing Jenkins installation, you will see additional XML files relating to Jenkins configuration and plugins.

The main directories are described in more detail in

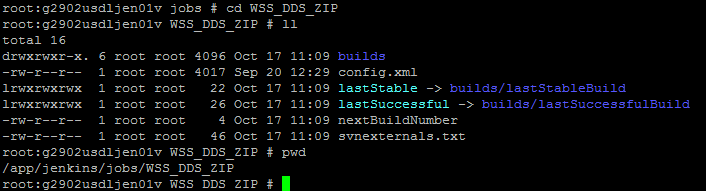








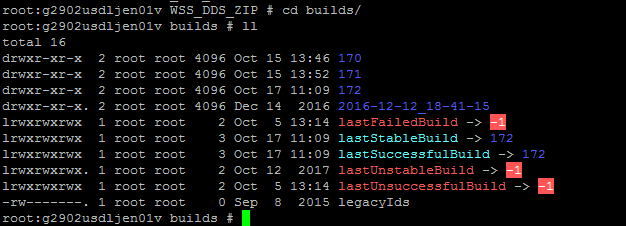
The jobs directory is a crucial part of the Jenkins directory structure, and deserves a bit more attention. You can see an example of a real Jenkins jobs directory in



This directory contains a subdirectory for each Jenkins build job being managed by this instance of Jenkins. Each job directory in turn contains two subdirectories: builds and workspace(In our case modified the default workspace to /udeploynas/Jenkins/workspace), along with some other files. In particular, it contains the build job config.xml file, which contains, as you might expect, the configuration details for this build job. There are also some other files used internally by Jenkins, that you usually wouldn’t touch, such as the nextBuildNumber file (which contains the number that will be assigned to the next build in this build job), as well as symbolic links to the most recent successful build and the last stable one. A successful build is one that does not have any compilation errors. A stable build is a successful build that has passed whatever quality criteria you may have configured, such as unit tests, code coverage and so forth.

Both the build and the workspace directories are important. The workspace directory is where Jenkins builds your project: it contains the source code Jenkins checks out, plus any files generated by the build itself. This workspace is reused for each successive build—there is only ever one workspace directory per project, and the disk space it requires tends to be relatively stable.

The builds directory contains a history of the builds executed for this job. You rarely need to intervene directly in these directories, but it can be useful to know what they contain. You can see a real example of the builds directory in [Figure 3-9](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch03s13.html#fig-hudson-jobs-directory-details), where three builds have been performed. Jenkins stores build history and artifacts for each build it performs in a directory labeled with a timestamp (“2016-12-12\_18-41-15” and so forth in [Figure 3-9](https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/ch03s13.html#fig-hudson-jobs-directory-details)). It also contains symbolic links with the actual build numbers that point to the build history directories.



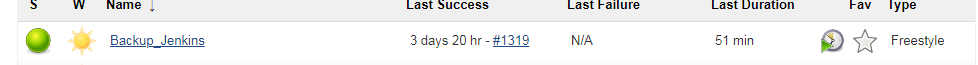
Each build directory contains information such as the build result log file, the Subversion revision number used for this build (if you are using Subversion), the changes that triggered this build, and any other data or metrics that you have asked Jenkins to keep track of. For example, if your build job keeps track of unit test results or test coverage metrics, this data will be stored here for each build. The build directory also contains any artifacts you are storing—binary artifacts, but also other generated files such as javadoc or code coverage metrics. Some types of build jobs, such as the Jenkins Maven build jobs, will also archive binary artifacts by default.

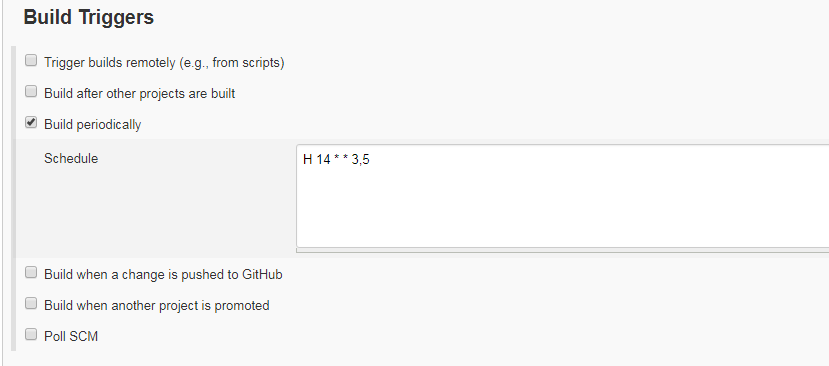
The size of the *build* directory will naturally grow over time, as the build history cumulates. You will probably want to take this into account when designing your build server directory structure, especially if your build server is running in a Unix-style environment with multiple disk partitions. A lot of this data takes the form of text or XML files, which does not consume a large amount of extra space for each build. However, if your build archives some of your build artifacts, such as JAR or WAR files, they too will be stored here. The size of these artifacts should be factored into your disk space requirements. We will see later on how to limit the number of builds stored for a particular build job if space is an issue. Limiting the number of build jobs that Jenkins stores is always a trade-off between disk space and keeping useful build statistics, as Jenkins does rely on this build history for its powerful reporting features.

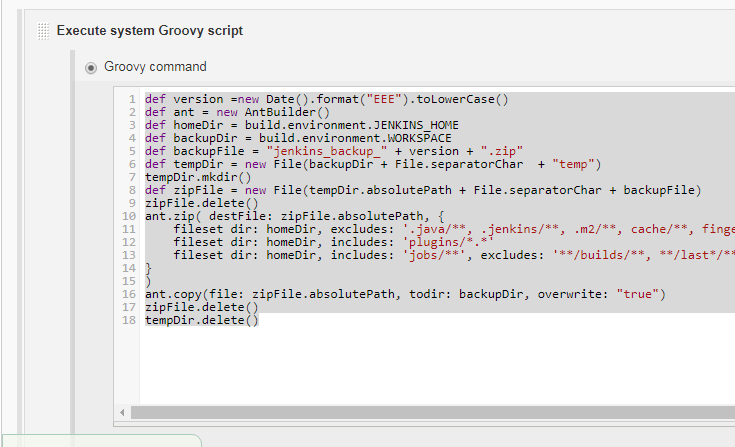
Jenkins uses the files in this directory extensively to display build history and metrics data, so you should be particularly careful not to delete any of the build history directories without knowing exactly what you are doing.

# Backing Up Your Jenkins Data

It is important to ensure that your Jenkins data is regularly backed up. This applies in particular to the Jenkins home directory, which contains your server configuration details as well as your build artifacts and build histories. This directory should be backed up frequently and automatically. The Jenkins executable itself is less critical, as it can easily be reinstalled without affecting your build environment.







File is uploaded to github

