



JENKINS

Continuous Integration Tool



Chandra Shekhar Reddy

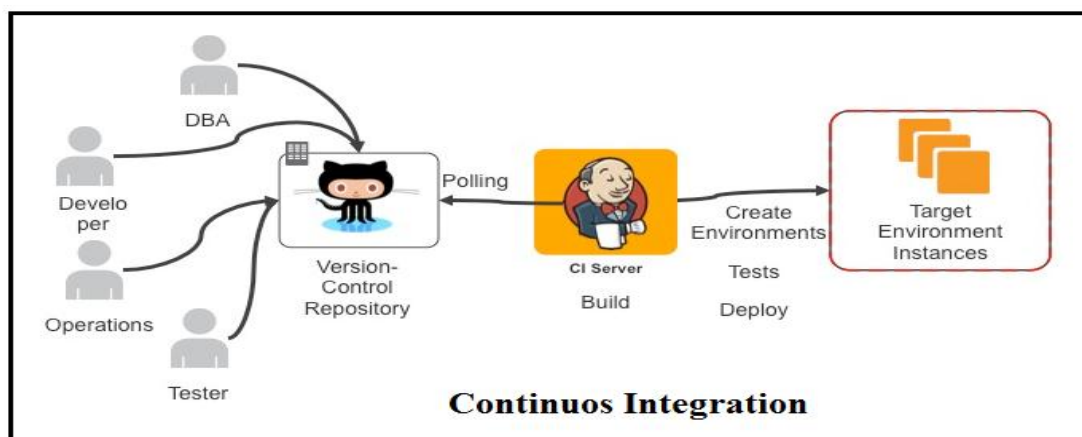
INTRODUCTION

Jenkins (née Hudson) is a continuous integration server written in Java. It can be downloaded from <http://www.jenkins-ci.org> as an executable war. It means that you can run it with **java -jar jenkins.war**. Jenkins comes with an embedded Servlet Container (Winstone) but you also have the option to deploy the war to an application server like Tomcat, Jetty, JBoss, etc. Jenkins does not use any database to store its configuration. Jenkins uses XStream to save data as XML files.

With Jenkins you can create, monitor and schedule jobs. There are plug-ins for almost anything that you may think about, from different SCMs (git, mercury, SVN, CVS) to plug-ins for integrating your Jenkins with Selenium, Gerrit, TestLink and other tools.

Continuous Integration:

Continuous Integration is a software development practice where members of a team integrate their work frequently; usually each person integrates at least daily—leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.



Jenkins features:

- Easy Installation
- Change set support
- Permanent links
- RSS/E-mail/IM Integration
- After-the-fact-tagging
- JUnit/TestNG test reporting
- Distributed builds
- File fingerprinting
- Plugin support
- Easy configuration

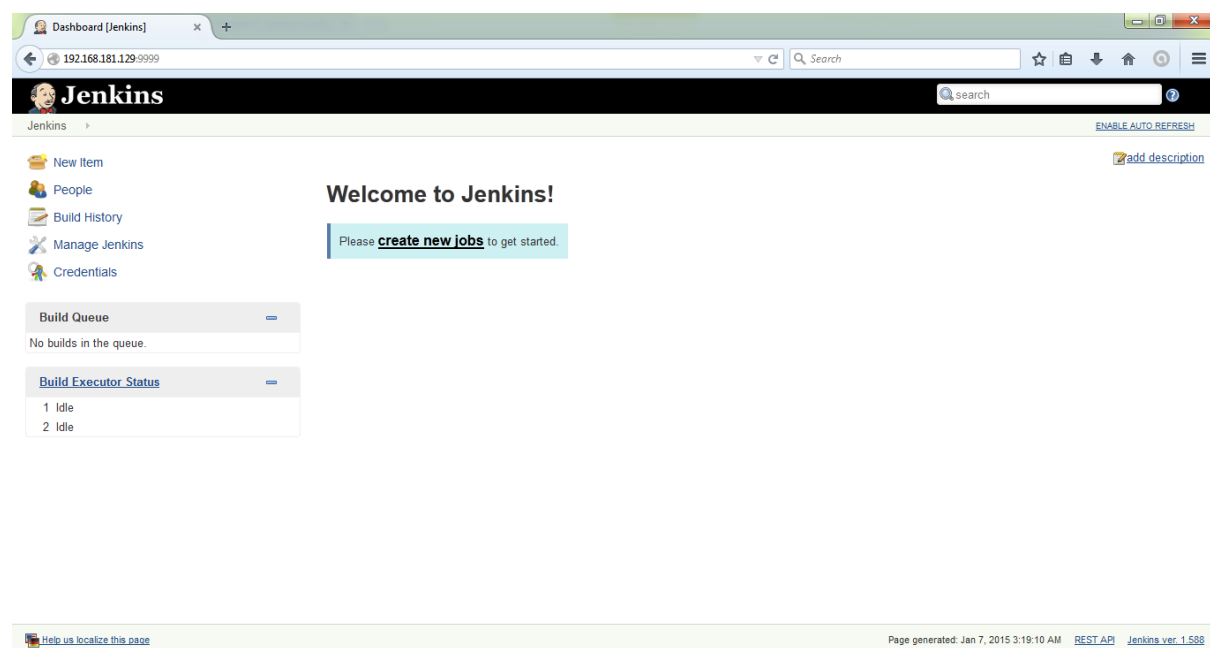
Installing Jenkins:

Download *jenkins.war* from <http://www.jenkins-ci.org>. Now open a terminal, navigate to the directory where that *jenkins.war* is available and execute **java -jar jenkins.war**. This will initiate Jenkins in port 8080 by default (if you need to change that port, use `--httpPort=9999`).

Jenkins creates a default workspace for you at `~/.jenkins` or it uses the folder specified in `JENKINS_HOME` environment variable.

```
root@localhost:~/backup
[root@localhost backup]# java -jar jenkins.war --httpPort=9999
Running from: /root/backup/jenkins.war
webroot: $user.home/.jenkins
Jan 07, 2015 3:14:40 AM winstone.Logger logInternal
INFO: Beginning extraction from war file
Jan 07, 2015 3:14:40 AM org.eclipse.jetty.util.log.JavaUtilLog info
INFO: jetty-winstone-2.8
Jan 07, 2015 3:14:44 AM org.eclipse.jetty.util.log.JavaUtilLog info
INFO: NO JSP Support for , did not find org.apache.jasper.servlet.JspServlet
Jenkins home directory: /root/.jenkins found at: $user.home/.jenkins
Jan 07, 2015 3:14:45 AM org.eclipse.jetty.util.log.JavaUtilLog info
INFO: Started SelectChannelConnector@0.0.0.0:9999
Jan 07, 2015 3:14:45 AM winstone.Logger logInternal
INFO: Winstone Servlet Engine v2.0 running: controlPort=disabled
Jan 07, 2015 3:14:45 AM jenkins.InitReactorRunner$1 onAttained
INFO: Started initialization
Jan 07, 2015 3:14:45 AM jenkins.InitReactorRunner$1 onAttained
INFO: Listed all plugins
Jan 07, 2015 3:14:46 AM jenkins.InitReactorRunner$1 onAttained
INFO: Prepared all plugins
Jan 07, 2015 3:14:46 AM jenkins.InitReactorRunner$1 onAttained
INFO: Started all plugins
Jan 07, 2015 3:14:48 AM jenkins.InitReactorRunner$1 onAttained
INFO: Augmented all extensions
Jan 07, 2015 3:14:48 AM jenkins.InitReactorRunner$1 onAttained
INFO: Loaded all jobs
Jan 07, 2015 3:14:51 AM org.jenkinsci.main.modules.sshd.SSHD start
INFO: Started SSHD at port 42235
Jan 07, 2015 3:14:51 AM jenkins.InitReactorRunner$1 onAttained
INFO: Completed initialization
Jan 07, 2015 3:14:51 AM hudson.WebAppMain$3 run
INFO: Jenkins is fully up and running
```

Go to <http://localhost:8080> to check if your installation is working. We will call this page as main screen from now on. The examples in this tutorial require you to have administrator rights in Jenkins. In this scenario I have used remote server for Jenkins <http://192.168.181.129:9999/>.



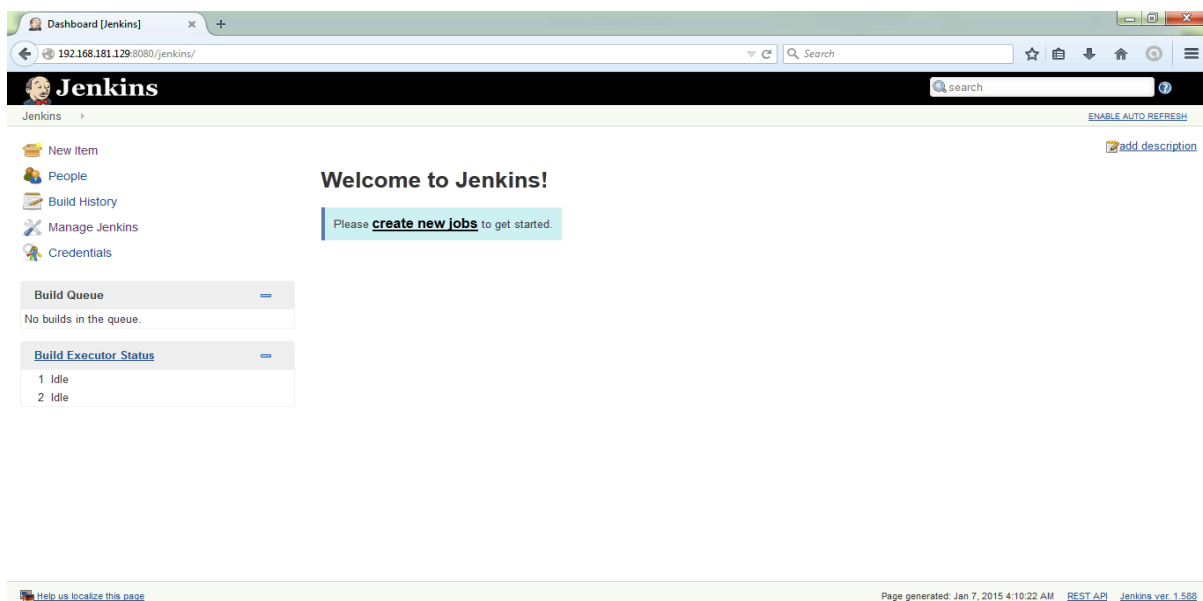
In other way you can deploy/copy Jenkins.war to Web or Application Server. Here I am going to use free and open source web server **Apache Tomcat Server**.

```
root@localhost:~/backup
[root@localhost backup]# cp jenkins.war /opt/tomcat7/webapps/
[root@localhost backup]# ls /opt/tomcat7/webapps/
atlassian-jira-4.2      docs      helloWorld-0.1-dev    host-manager  manager
atlassian-jira-4.2.war  examples  helloWorld-0.1-dev.war jenkins.war   ROOT
[root@localhost backup]#
```

To execute or access the Jenkins you need to start the tomcat server.

```
root@localhost:~/backup
[root@localhost backup]# /opt/tomcat7/bin/startup.sh
Using CATALINA_BASE:   /opt/tomcat7
Using CATALINA_HOME:   /opt/tomcat7
Using CATALINA_TMPDIR: /opt/tomcat7/temp
Using JRE_HOME:        /opt/java/jdk1.8.0_25
Using CLASSPATH:       /opt/tomcat7/bin/bootstrap.jar:/opt/tomcat7/bin/tomcat-juli.jar
Tomcat started.
[root@localhost backup]#
```

Now you can start Jenkins. **URL:** <http://192.168.181.129:8080/jenkins/>



Once you get the main page you can configure the Jenkins by clicking on **Manage Jenkins**. In this section you will get many options to configure and extend the functionality of the Jenkins.

Securing Jenkins:

In the default configuration, Jenkins does not perform any security check. This means any person accessing the website can configure Jenkins and jobs, and perform builds. While this configuration is acceptable during initial evaluation of the software, Jenkins should be

configured to authenticate users and enforce access control in most other situations, especially when exposed to the Internet.

This setting is controlled mainly by two axes:

1. **Security Realm**, which determines users and their passwords, as well as what groups the users belong to.
2. **Authorization Strategy**, which determines who has access to what.

These two axes are orthogonal, and need to be individually configured. For example, you might choose to use external LDAP or Active Directory as the security realm, and you might choose "everyone full access once logged in" mode for authorization strategy. Or you might choose to let Jenkins run its own user database, and perform access control based on the permission/user matrix.

Configure Global Security

☒ Enable security

TCP port for JNLP slave agents: Fixed: Random ☒ Disable ☐

Disable remember me ☐

Access Control

Security Realm

☐ Delegate to servlet container

☐ Jenkins' own user database

☐ LDAP

☒ Unix user/group database

Authorization

☐ Anyone can do anything

☐ Legacy mode

☐ Logged-in users can do anything

☐ Matrix-based security

☒ Project-based Matrix Authorization Strategy

User/group	Overall	Credentials
Anonymous	<input type="checkbox"/> Administer <input type="checkbox"/> Read <input type="checkbox"/> Run Scripts <input type="checkbox"/> Upload Plugins <input type="checkbox"/> Configure <input type="checkbox"/> Update Center <input type="checkbox"/> Create <input type="checkbox"/> Update View <input type="checkbox"/> Delete <input type="checkbox"/> Manage Domains <input type="checkbox"/> Configure	<input type="checkbox"/>
chandu	<input checked="" type="checkbox"/> Administer <input checked="" type="checkbox"/> Read <input checked="" type="checkbox"/> Run Scripts <input checked="" type="checkbox"/> Upload Plugins <input checked="" type="checkbox"/> Configure <input checked="" type="checkbox"/> Update Center <input checked="" type="checkbox"/> Create <input checked="" type="checkbox"/> Update View <input checked="" type="checkbox"/> Delete <input checked="" type="checkbox"/> Manage Domains <input checked="" type="checkbox"/> Configure	<input checked="" type="checkbox"/>

User/group to add: Add

Markup Formatter

Escaped HTML

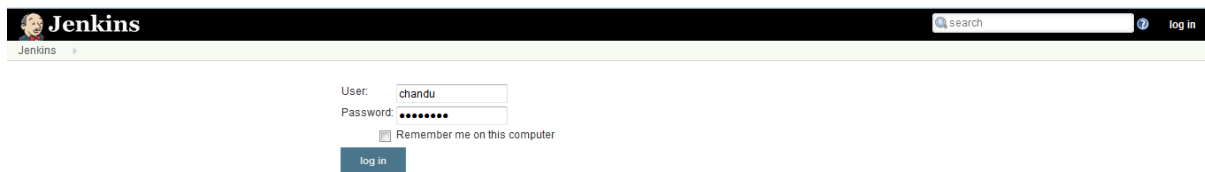
☐ Prevent Cross Site Request Forgery exploits

Save Apply

In the above situation I have selected Unix user/group database, In the Authorization section I have selected Project based Matrix Authorization Strategy, and I have given the username chandu, this user must be existed in your Linux/Unix System.

After finishing this configurations, click on apply and save buttons, then you will get the username and logout options in the right upper corner of the page.

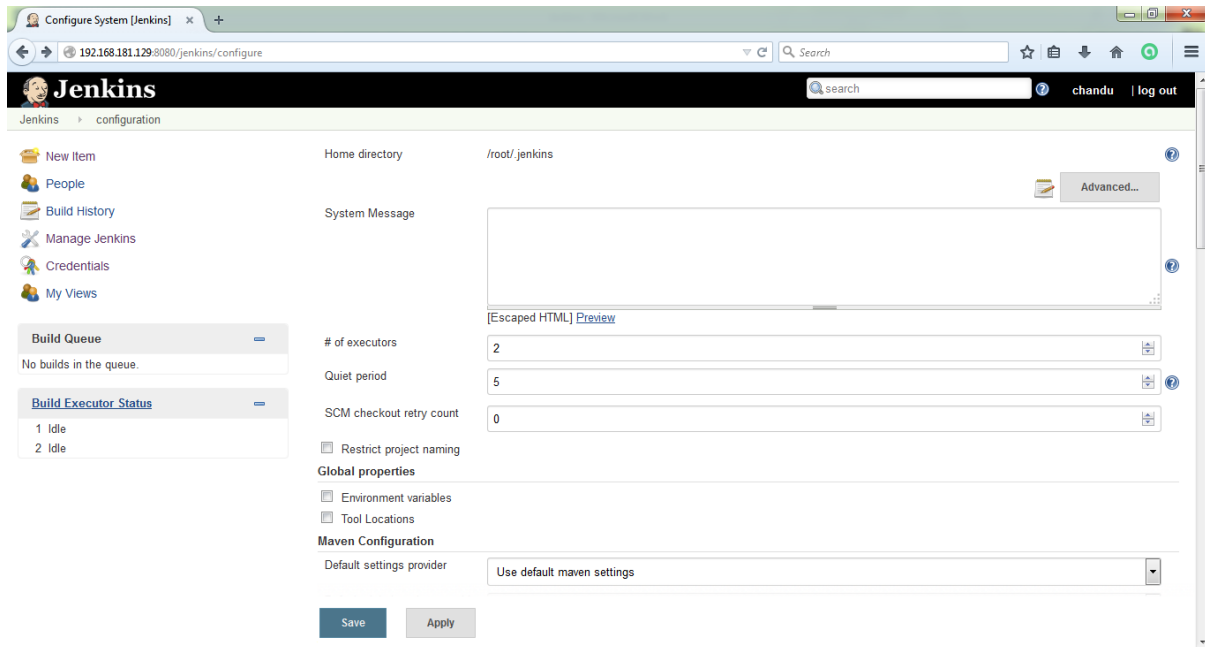
Next time onwards it will ask for the login when you access the Jenkins.



Click on 'Configure System' to make basic configurations in Jenkins. Here we can change the Jenkins Home Directory, No. of Executors and many other options.

1. **Home directory:** This is the basic directory which Jenkins uses for all the build related jobs. It is the working directory for Jenkins and all the changes and jobs configured in Jenkins are created inside this directory. Default Home directory is set to /root/.jenkins. This can be changed as per the user requirement. PS – If you plan to change the work directory from default to some custom location, make sure the directory is owned by the user who is running Jenkins. By default, Jenkins is triggered and executed by a self-created user 'jenkins'. This can be changed from the configuration file which will be covered later.
2. **Number of Executors:** This refers to the total concurrent job executions that can take place on the Jenkins box. We can change this as per the need. Recommended – No. of Executors should be set to equal the number of CPU that you have for optimal performance. PS – Changing No of executors from this place will make change in the master (Jenkins Host) only. For any slave machine, you need to manually change the No. of Executors for each slave.
3. **Environment Variables:** To add custom environment variables which apply to all the jobs, check the environment variables check-box and add the required environment variables. These are key-value pairs and can be accessed and used in Builds wherever required.
4. **Jenkins URL:** By default Jenkins URL points to localhost. If you have a domain name setup for your machine, set this to the domain name else overwrite localhost with IP of machine. This will help in setting up slaves and while sending out links using the email as you can directly access the Jenkins URL using the environment variable JENKINS_URL which can be accessed as \${JENKINS_URL}.
5. **E-mail Notification:** In the email Notification area you can configure the SMTP settings for sending out emails. This is required for Jenkins to connect to the SMTP

mail server and send out emails to the recipient list. PS – to send out emails from Jenkins you need to specify an email address (Sender Email) in System Admin e-mail address. Jenkins will use this email address to send out the emails to the recipient list.



Jenkins Home Directory:

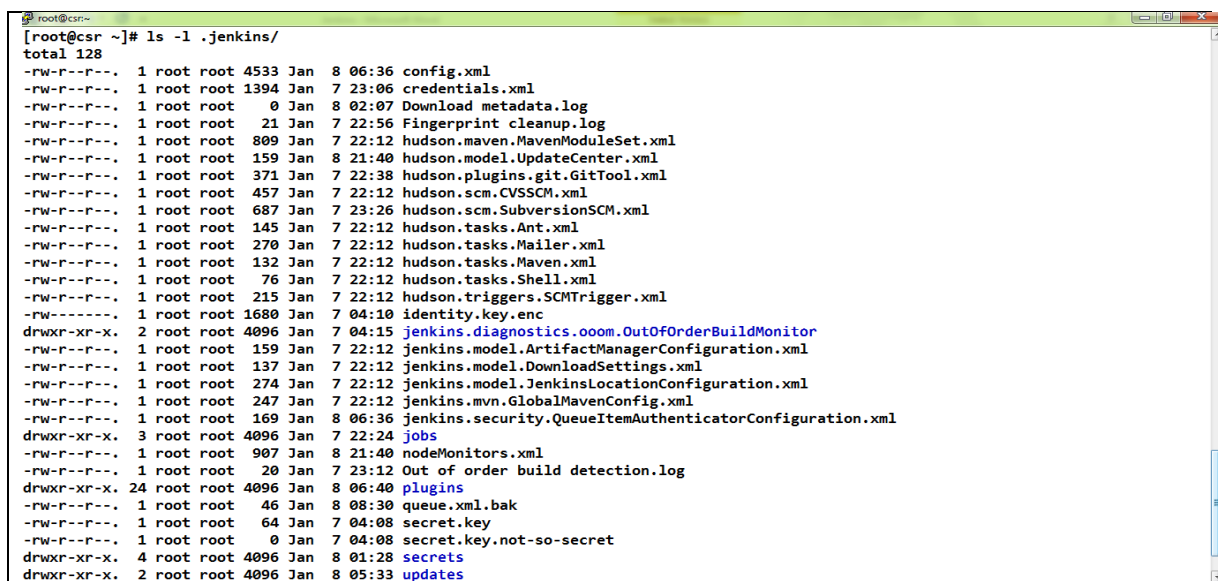
The Jenkins home directory contains all the details of your Jenkins server configuration, details that you configure in the Manage Jenkins screen. These configuration details are stored in the form of a set of XML files. Much of the core configuration, for example, is stored in the config.xml file. Other 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.

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.

Directory	Description
.jenkins	The default Jenkins home directory is .jenkins (may be .hudson in older installations).
fingerprints	This directory is used by Jenkins to keep track of artifact fingerprints. We look at how to track artifacts later on in the book.

jobs	This directory contains configuration details about the build jobs that Jenkins manages, as well as the artifacts and data resulting from these builds. We look at this directory in detail below.
plugins	This directory contains any plugins that you have installed. Plugins allow you to extend Jenkins by adding extra feature. Note that, with the exception of the Jenkins core plugins (subversion, cvs, ssh-slaves, maven, and scid-ad), plugins are not stored with the jenkins executable, or in the expanded web application directory. This means that you can update your Jenkins executable and not have to reinstall all your plugins.
updates	This is an internal directory used by Jenkins to store information about available plugin updates.
userContent	You can use this directory to place your own custom content onto your Jenkins server. You can access files in this directory at http://myserver/hudson/userContent (if you are running Jenkins on an application server) or http://myserver/userContent (if you are running in stand-alone mode).
users	If you are using the native Jenkins user database, user accounts will be stored in this directory.
war	This directory contains the expanded web application. When you start Jenkins as a stand-alone application, it will extract the web application into this directory.
workspace	It is the place where all the work is stored.

Below figure is illustrating the home directory contents of Jenkins (.jenkins directory).



```

root@csr:~# ls -l .jenkins/
total 128
-rw-r--r-- 1 root root 4533 Jan 8 06:36 config.xml
-rw-r--r-- 1 root root 1394 Jan 7 23:06 credentials.xml
-rw-r--r-- 1 root root 0 Jan 8 02:07 Download metadata.log
-rw-r--r-- 1 root root 21 Jan 7 22:56 Fingerprint cleanup.log
-rw-r--r-- 1 root root 809 Jan 7 22:12 hudson.maven.MavenModuleSet.xml
-rw-r--r-- 1 root root 159 Jan 8 21:40 hudson.model.UpdateCenter.xml
-rw-r--r-- 1 root root 371 Jan 7 22:38 hudson.plugins.git.GitTool.xml
-rw-r--r-- 1 root root 457 Jan 7 22:12 hudson.scm.CVSSCM.xml
-rw-r--r-- 1 root root 687 Jan 7 23:26 hudson.scm.SubversionSCM.xml
-rw-r--r-- 1 root root 145 Jan 7 22:12 hudson.tasks.Ant.xml
-rw-r--r-- 1 root root 270 Jan 7 22:12 hudson.tasks.Mailer.xml
-rw-r--r-- 1 root root 132 Jan 7 22:12 hudson.tasks.Maven.xml
-rw-r--r-- 1 root root 76 Jan 7 22:12 hudson.tasks.Shell.xml
-rw-r--r-- 1 root root 215 Jan 7 22:12 hudson.triggers.SCMTrigger.xml
-rw-r--r-- 1 root root 1680 Jan 7 04:10 identity.key.enc
drwxr-xr-x 2 root root 4096 Jan 7 04:15 jenkins.diagnostics.outOfOrderBuildMonitor
-rw-r--r-- 1 root root 159 Jan 7 22:12 jenkins.model.ArtifactManagerConfiguration.xml
-rw-r--r-- 1 root root 137 Jan 7 22:12 jenkins.model.DownloadSettings.xml
-rw-r--r-- 1 root root 274 Jan 7 22:12 jenkins.model.JenkinsLocationConfiguration.xml
-rw-r--r-- 1 root root 247 Jan 7 22:12 jenkins.mvn.GlobalMavenConfig.xml
-rw-r--r-- 1 root root 169 Jan 8 06:36 jenkins.security.QueueItemAuthenticatorConfiguration.xml
drwxr-xr-x 3 root root 4096 Jan 7 22:24 jobs
-rw-r--r-- 1 root root 907 Jan 8 21:40 nodeMonitors.xml
-rw-r--r-- 1 root root 20 Jan 7 23:12 Out of order build detection.log
drwxr-xr-x 24 root root 4096 Jan 8 06:40 plugins
-rw-r--r-- 1 root root 46 Jan 8 08:30 queue.xml.bak
-rw-r--r-- 1 root root 64 Jan 7 04:08 secret.key
-rw-r--r-- 1 root root 0 Jan 7 04:08 secret.key.not-so-secret
drwxr-xr-x 4 root root 4096 Jan 8 01:28 secrets
drwxr-xr-x 2 root root 4096 Jan 8 05:33 updates

```


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.



```
[root@csr ~]# ls -l .jenkins/jobs/helloworld/
total 20
drwxr-xr-x. 6 root root 4096 Jan  8 07:23 builds
-rw-r--r--. 1 root root 1410 Jan  8 02:28 config.xml
lrwxrwxrwx. 1 root root   22 Jan  8 07:23 lastStable -> builds/lastStableBuild
lrwxrwxrwx. 1 root root   26 Jan  8 07:23 lastSuccessful -> builds/lastSuccessfulBuild
-rw-r--r--. 1 root root    3 Jan  8 07:23 nextBuildNumber
-rw-r--r--. 1 root root 506 Jan  7 23:26 subversion.credentials
-rw-r--r--. 1 root root  46 Jan  8 07:23 svnexternals.txt
[root@csr ~]#
```

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.

Upgrading Your Jenkins Installation:

Upgrading Jenkins is easy—you simply replace your local copy of the jenkins.war file and restart Jenkins. However you should make sure there are no builds running when you restart your server. Since your build environment configuration details, plugins, and build history are stored in the Jenkins home directory, upgrading your Jenkins executable will have no impact on your installation. You can always check what version of Jenkins you are currently running by referring to the version number in the bottom right corner of every screen. If you have installed Jenkins using one of the Linux packages, Jenkins can be upgraded using the same process as the other system packages on the server. If you are running Jenkins as a stand-alone instance, you can also upgrade your Jenkins installation directly from the web interface, in the Manage Jenkins section. Jenkins will indicate if a more recent version is available, and give you the option to either download it manually or upgrade automatically.

Configuring Your Jenkins Server

Before you can start creating your build jobs in Jenkins, you need to do a little configuration, to ensure that your Jenkins server works smoothly in your particular environment. Jenkins is highly configurable, and, although most options are provided with sensible default values, or are able to find the right build tools in the system path and environment variables, it is always a good idea to know exactly what your build server is doing. Jenkins is globally very easy to configure. The administration screens are intuitive, and the contextual online help (the blue question mark icons next to each field) is detailed and precise. In this chapter, we will look at how to configure your basic server setup in detail, including how to configure Jenkins to use different versions of Java, build tools such as Ant and Maven, and SCM tools such as CVS and Subversion. We will look at more advanced server configuration, such as using other version control systems or notification tools.

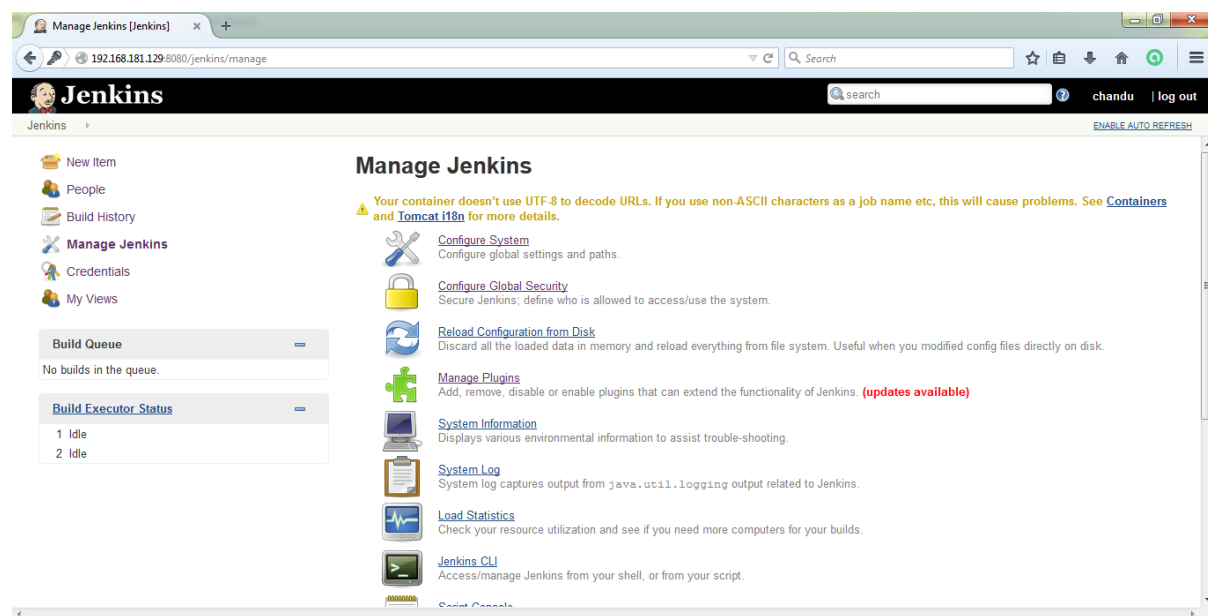
The Configuration Dashboard - The Manage Jenkins Screen:

In Jenkins, you manage virtually all aspects of system configuration in the Manage Jenkins screen. You can also get to this screen directly from anywhere in the application by typing “manage” in the Jenkins search box. This screen changes depending on what plugins you install, so don’t be surprised if you see more than what we show here.

This below screen lets you configure different aspects of your Jenkins server. Each link on this page takes you to a dedicated configuration screen, where you can manage different parts of the Jenkins server. Some of the more interesting options are discussed here:

Configure System

This is where you manage paths to the various tools you use in your builds, such as JDKs, and versions of Ant and Maven, as well as security options, email servers, and other system-wide configuration details. Many of the plugins that you install will also need to be configured here— Jenkins will add the fields dynamically when you install the plugins.



Reload Configuration from Disk

Jenkins stores all its system and build job configuration details as XML files stored in the Jenkins home directory. It also stores all of the build history in the same directory. If you are migrating build jobs from one Jenkins instance to another, or archiving old build jobs, you will need to add or remove the corresponding build job directories to Jenkins’s builds directory. You don’t need to take Jenkins offline to do this—you can simply use the “Reload Configuration from Disk” option to reload the Jenkins system and build job configurations directly. This process can be a little slow if there is a lot of build history, as Jenkins loads not only the build configurations but also all of the historical data as well.

Manage Plugins

One of the best features of Jenkins is its extensible architecture. There is a large ecosystem of third-party open source plugins available, enabling you to add extra features to your build server, from support for different SCM tools such as Git, Mercurial or ClearCase, to code quality and code coverage metrics reporting. We will be looking at many of the more popular and useful plugins throughout this book. Plugins can be installed, updated and removed through the Manage Plugins screen. Note that removing plugins needs to be done with some care, as it can sometimes affect the stability of your Jenkins instance.

System Information

This screen displays a list of all the current Java system properties and system environment variables. Here, you can check exactly what version of Java Jenkins is running in, what user it is running under, and so forth. You can also check that Jenkins is using the correct environment variable settings. Its main use is for troubleshooting, so that you can make sure that your server is running with the system properties and variables you think it is.

System Log

The System Log screen is a convenient way to view the Jenkins log files in real time. Again, the main use of this screen is for troubleshooting. You can also subscribe to RSS feeds for various levels of log messages. For example, as a Jenkins administrator, you might want to subscribe to all the ERROR and WARNING log messages.

Load Statistics

Jenkins keeps track of how busy your server is in terms of the number of concurrent builds and the length of the build queue (which gives an idea of how long your builds need to wait before being executed). These statistics can give you an idea of whether you need to add extra capacity or extra build nodes to your infrastructure.

Script Console

This screen lets you run Groovy scripts on the server. It is useful for advanced troubleshooting: since it requires a strong knowledge of the internal Jenkins architecture, it is mainly useful for plugin developers and the like.

Manage Nodes

Jenkins handles parallel and distributed builds well. In this screen, you can configure how many builds you want. Jenkins runs simultaneously, and, if you are using distributed builds, set up build nodes. A build node is another machine that Jenkins can use to execute its builds. We will look at how to configure distributed builds in detail later sections.

Prepare for Shutdown

If you need to shut down Jenkins, or the server Jenkins is running on, it is best not to do so when a build is being executed. To shutdown Jenkins cleanly, you can use the Prepare for

Shutdown link, which prevents any new builds from being started. Eventually, when all of the current builds have finished, you will be able to shut down Jenkins cleanly.

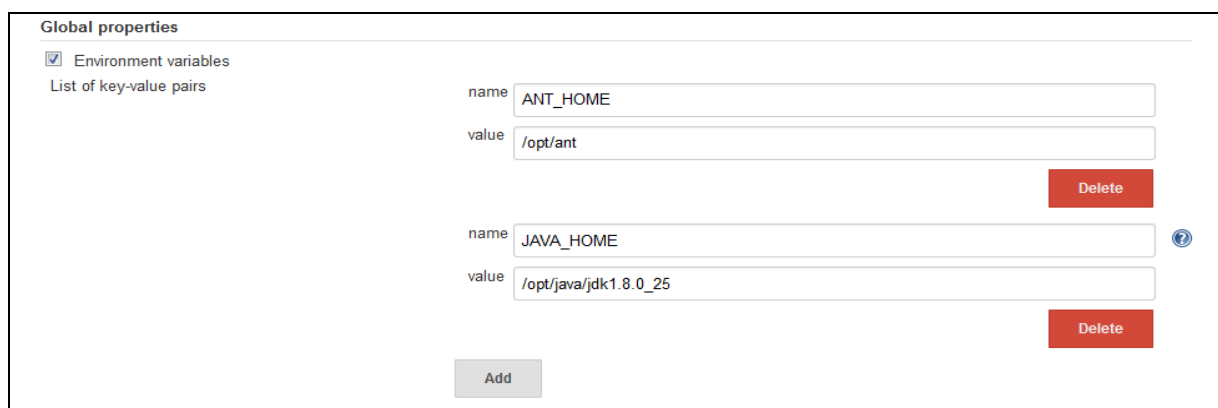
Configuring the System Environment

The most important Jenkins administration page is the Configure System screen. Here, you set up most of the fundamental tools that Jenkins needs to do its daily work. The default screen contains a number of sections, each relating to a different configuration area or external tool. In addition, when you install plugins, their system-wide configuration is also often done in this screen.

The Configure System screen lets you define global parameters for your Jenkins installation, as well as external tools required for your build process. The first part of this screen lets you define some general system-wide parameters. The Jenkins home directory is displayed, for reference. This way, you can check at a glance that you are working with the home directory that you expect. Remember, you can change this directory by setting the JENKINS_HOME environment variable in your environment.

Configuring Global Properties

The Global Properties section lets you define variables that can be managed centrally but used in all of your build jobs. You can add as many properties as you want here, and use them in your build jobs. Jenkins will make them available within your build job environment, so you can freely use them within your Ant and Maven build scripts. Note that you shouldn't put periods (".") in the property names, as they won't be processed correctly. So ldapserver or ldap_server is fine, but not ldap.server.



The screenshot shows the 'Global properties' section of the Jenkins configuration interface. It features a checkbox labeled 'Environment variables' which is checked. Below this, a text label reads 'List of key-value pairs'. The main area contains two rows of input fields. The first row has a 'name' field with 'ANT_HOME' and a 'value' field with '/opt/ant', followed by a red 'Delete' button. The second row has a 'name' field with 'JAVA_HOME', a 'value' field with '/opt/java/jdk1.8.0_25', and a red 'Delete' button. A blue help icon is visible to the right of the second row. At the bottom left, there is a grey 'Add' button.

Configuring Your JDKs:

Historically, one of the most common uses of Jenkins has been to build Java applications. So Jenkins naturally provides excellent built-in support for Java. You can able to configure existing java in your machine

JDK

JDK installations

JDK
Name
JAVA_HOME
<input type="checkbox"/> Install automatically

Delete JDK

Add JDK

List of JDK installations on this system

Configuring Your Build Tools:

Build tools are the bread-and-butter of any build server, and Jenkins is no exception. Out of the box, Jenkins supports three principal build tools: Ant, Maven, and the basic shell-script (or Batch script in Windows). Using Jenkins plugins, you can also add support for other build tools and other languages, such as Gant, Grails, MSBuild, and many more.

Configuring Ant:

You can configure Apache Ant either selecting the check box for automatic installation, or directly configure your existing installation.

Jenkins provides excellent build-in support for Ant—you can invoke Ant targets from your build job, providing properties to customize the process as required.

Ant

Ant
Name
ANT_HOME
<input type="checkbox"/> Install automatically

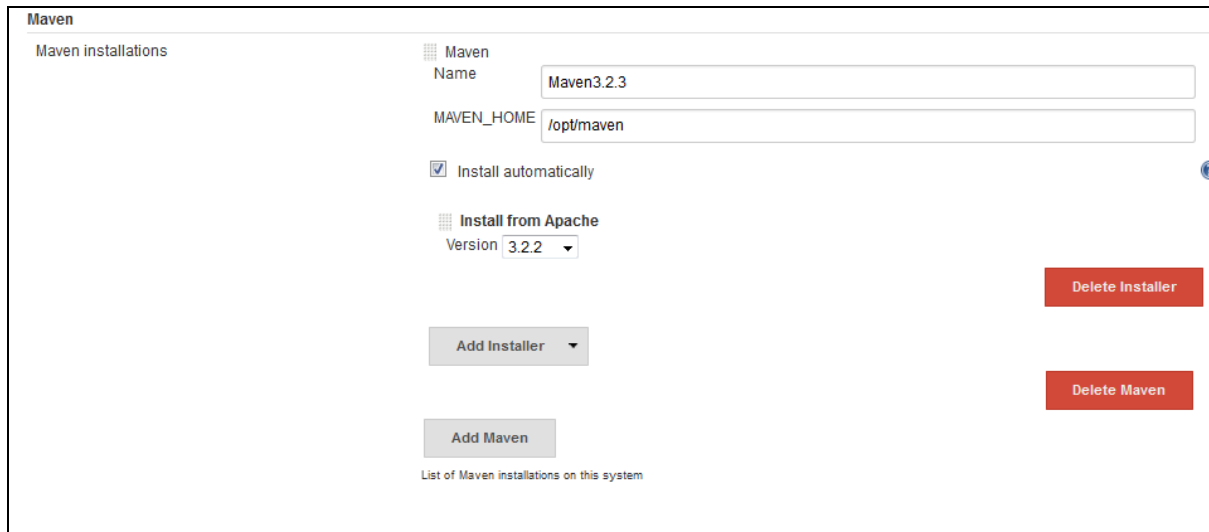
Delete Ant

Configuring Maven:

Maven is a high-level build scripting framework for Java that uses notions such as a standard directory structure and standard life cycles, Convention over Configuration, and Declarative Dependency Management to simplify a lot of the low-level scripting that you find in a typical Ant build script. In Maven, your project uses a standard, well-defined build life cycle—compile, test, package, deploy, and so forth. Each life cycle phase is associated with a Maven plugin. The various Maven plugins use the standard directory structure to carry out these tasks with a minimum of intervention on your part. You can also extend Maven by overriding the default plugin configurations or by invoking additional plugins.

Jenkins provides excellent support for Maven, and has a good understanding of Maven project structures and dependencies. You can either get Jenkins to install a specific version of Maven automatically or provide a path to a local Maven installation. You can configure as many

versions of Maven for your build projects as you want, and use different versions of Maven for different projects.



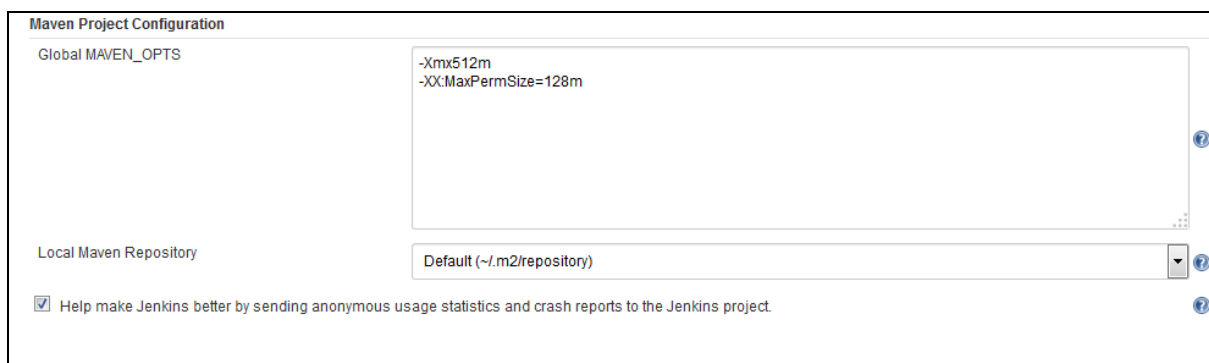
The screenshot shows the 'Maven' configuration page in Jenkins. On the left, there is a sidebar with 'Maven installations'. The main area is titled 'Maven' and contains the following fields and controls:

- Name:** A text input field containing 'Maven3.2.3'.
- MAVEN_HOME:** A text input field containing '/opt/maven'.
- Install automatically:** A checked checkbox.
- Install from Apache:** A section with a 'Version' dropdown menu set to '3.2.2'.
- Buttons:** 'Add Installer' (disabled), 'Add Maven' (disabled), 'Delete Installer' (red), and 'Delete Maven' (red).
- Footer:** A link 'List of Maven installations on this system'.

If you tick the Install automatically checkbox, Jenkins will download and install the requested version of Maven for you. You can either ask Jenkins to download Maven directly from the Apache site, or from a (presumably local) URL of your choice. This is an excellent choice when you are using distributed builds, and, since Maven is cross-platform, it will work on any machine. You don't need to install Maven explicitly on each build machine—the first time a build machine needs to use Maven, it will download a copy and install it to the tools directory in the Jenkins home directory.

Maven Project Configuration:

Sometimes you need to pass Java system options to your Maven build process. For instance it is often useful to give Maven a bit of extra memory for heavyweight tasks such as code coverage or site generation. Maven lets you do this by setting the MAVEN_OPTS variable. In Jenkins, you can set a system-wide default value, to be used across all projects (Configuring system-wide MVN_OPTS). This comes in handy if you want to use certain standard memory options (for example) across all projects, without having to set it up in each project by hand.



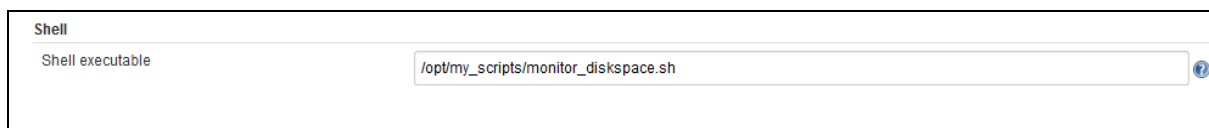
The screenshot shows the 'Maven Project Configuration' page in Jenkins. It contains the following fields and controls:

- Global MAVEN_OPTS:** A text area containing '-Xmx512m' and '-XX:MaxPermSize=128m'.
- Local Maven Repository:** A dropdown menu set to 'Default (~/.m2/repository)'.
- Help:** A checked checkbox with the text 'Help make Jenkins better by sending anonymous usage statistics and crash reports to the Jenkins project'.

Shell-Scripting Language

If you are running your build server on Unix or Linux, Jenkins lets you insert shell scripts into your build jobs. This is handy for performing low-level, OS-related tasks that you don't want to do in Ant or Maven. In the Shell section, you define the default shell that will be used when executing these shell scripts. By default, this is `/bin/sh`, but there are times you may want to modify this to another command interpreter such as `bash` or `Perl`.

In Windows, the Shell section does not apply—you use Windows batch scripting instead. So, on a Windows build server, you should leave this field blank.

A screenshot of the Jenkins configuration interface for the 'Shell' section. It shows a label 'Shell' above a text input field. The input field has a placeholder text 'Shell executable' and contains the value '/opt/my_scripts/monitor_diskspace.sh'. To the right of the input field is a small blue question mark icon.

Configuring Your Version Control Tools

Jenkins comes preinstalled with plugins for CVS and Subversion. Other version control systems are supported by plugins that you can download from the Manage Plugins screen.

Configuring Subversion

Subversion needs no special configuration, since Jenkins uses native Java libraries to interact with Subversion repositories. If you need to authenticate to connect to a repository, Jenkins will prompt you when you enter the Subversion URL in the build job configuration.

Configuring CVS

CVS needs little or no configuration. By default, Jenkins will look for tools like CVS on the system path, though you can provide the path explicitly if it isn't on the system path. CVS keeps login and password details in a file called `.cvspass`, which is usually in your home directory. If this is not the case, you can provide a path where Jenkins can find this file.

Configuring the Mail Server

The last of the basic configuration options you need to set up is the email server configuration. Email is Jenkins's more fundamental notification technique—when a build fails, it will send an email message to the developer who committed the changes, and optionally to other team members as well. So Jenkins needs to know about your email server.

The System Admin email address is the address from which the notification messages are sent. You can also use this field to check the email setup—if you click on the Test configuration button, Jenkins will send a test email to this address.

You also need to provide a proper base URL for your Jenkins server (one that does not use `localhost`). Jenkins uses this URL in the email notifications so that users can go directly from the email to the build failure screen on Jenkins.

Jenkins also provides for more sophisticated email configuration, using more advanced features such as SMTP authentication and SSL. If this is your case, click on the Advanced button to configure these options.

For example, many organizations use Google Apps for their email services. You can configure Jenkins to work with the Gmail service as shown in Figure 4.11, “Configuring an email server in Jenkins to use a Google Apps domain”. All you need to do in this case is to use the Gmail SMTP server, and provide your Gmail username and password in the SMTP Authentication (you also need to use SSL and the non-standard port of 465).

The screenshot shows the 'E-mail Notification' configuration page in Jenkins. It contains the following fields and settings:

- SMTP server:** smtp.gmail.com
- Default user e-mail suffix:** (empty)
- Use SMTP Authentication:** ☒
- User Name:** chandu.2035@gmail.com
- Password:** (masked with dots)
- Use SSL:** ☒
- SMTP Port:** 465
- Reply-To Address:** (empty)
- Charset:** UTF-8
- Test configuration by sending test e-mail:** ☒
- Test e-mail recipient:** chandu.2035@gmail.com
- Test configuration button:** (bottom right)

Configuring a Proxy

In most enterprise environments, your Jenkins server will be situated behind a firewall, and will not have direct access to the Internet. Jenkins needs Internet access to download plugins and updates, and also to install tools such as the JDK, Ant and Maven from remote sites. If you need to go through an HTTP proxy server to get to the Internet, you can configure the connection details (the server and port, and if required the username and password) in the Advanced tab on the Plugin Manager screen.

If your proxy is using Microsoft’s NTLM authentication scheme, then you will need to provide a domain name as well as a username. You can place both in the User name field: just enter the domain name, followed by a back-slash (\), and followed by the username, such as “MyDomain\chandu”.

Finally, if you are setting up Proxy access on your Jenkins build server, remember that all of the other tools running on this server will need to know about the proxy as well. In particular, this may include tools such as Subversion (if you are accessing an external repository) and Maven (if you are not using an Enterprise Repository Manager).

Updates

Available

Installed

Advanced

HTTP Proxy Configuration

Server

csrtarget.in

Port

8080

User name

chandu

Password

.....

No Proxy Host

Test URL

Validate Proxy

Submit

Upload Plugin

You can upload a .hpi file to install a plugin from outside the central plugin repository.

File:

Browse...

No file selected.

Upload

Update Site

URL

http://updates.jenkins-ci.org/update-center.json

Submit

Update information obtained: 1 hr 40 min ago

Check now

Setting Up Your Build Jobs

A build job is a particular way of compiling, testing, packaging, and 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.

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.

- **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.xml files and project structures, and can use the information gleaned from the pom.xml file to reduce the work you need to do to set up your project.

- **External job**

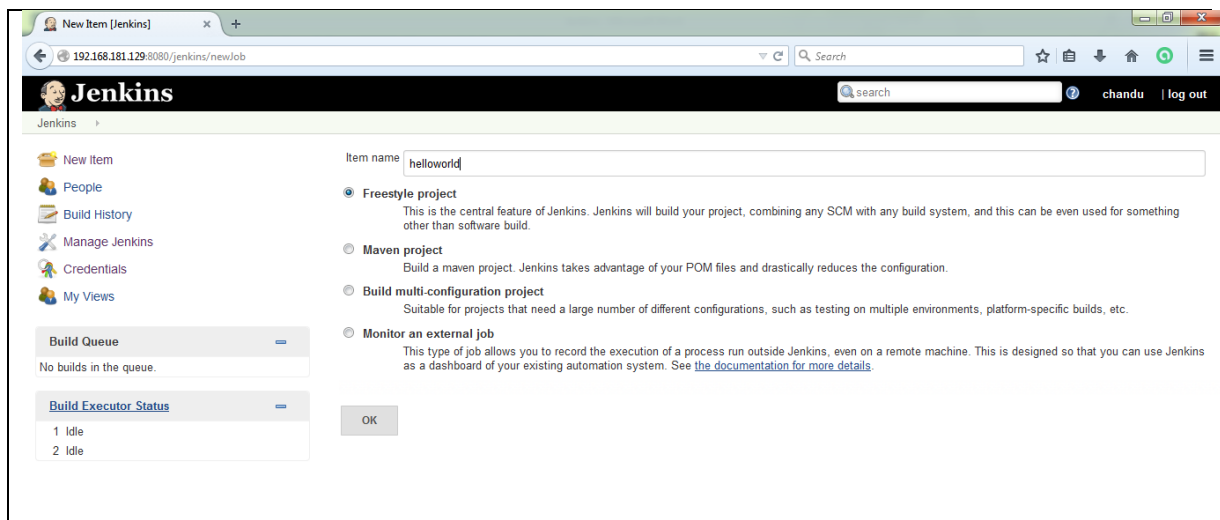
An “External job” build job lets you keep an eye on non-interactive processes, such as cron jobs.

- **Multi-configuration job**

The “multi-configuration project” (also referred to as a “matrix project”) lets you run the same build job in many different configurations. This powerful feature can be useful for testing an application in many different environments, with different databases, or even on different build machines.

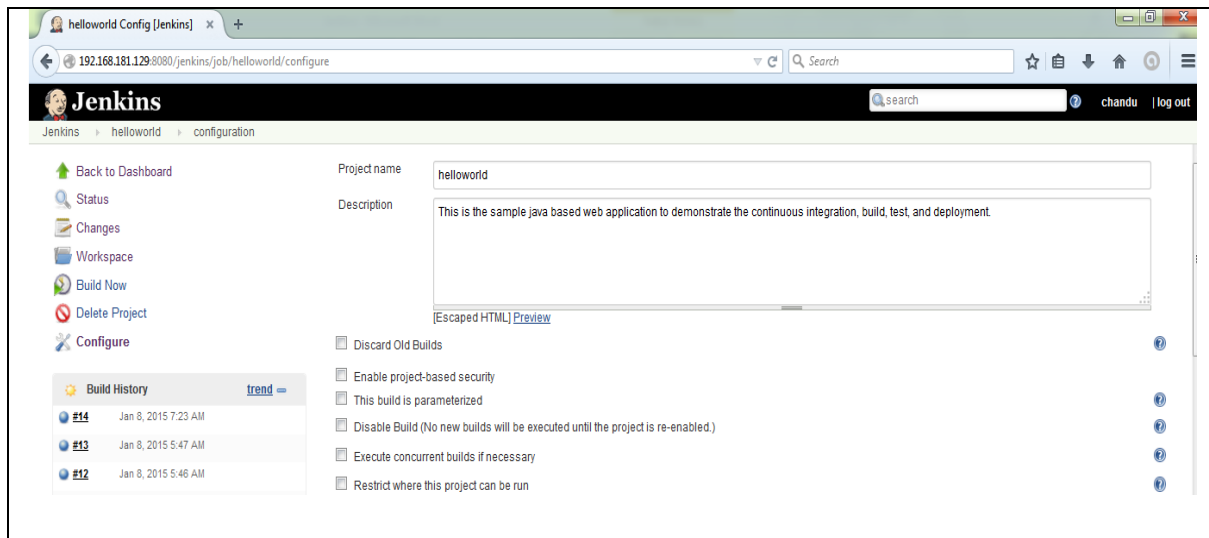
Creating a Jobs:

In order to create a new job all that you need to do is click on **New Job** or **New Item** and give it a name. Choose the option to create a *Free-style project*.



General Options

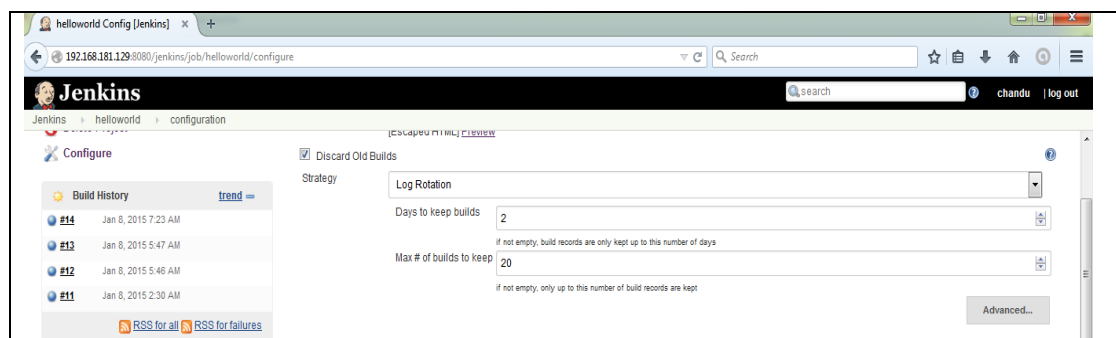
The first section you see when you create a new freestyle job contains general information about the project, such as a unique name and description, and other information about how and where the build job should be executed.



- **Discard Old Builds:**

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. 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.



- **Enable project based security:**

Different security realm has different conventions about group names. The best way to go about it is to login and see what group names you belong to, by going to [this diagnostics page](#).

A special group "authenticated" is also available, which represents all authenticated (logged in) users.

<input checked="" type="checkbox"/> Enable project-based security																
User/group	Credentials				Job				Run	SCM						
	Create	Delete	Manage	Domains	Update	View	Build	Cancel	Configure	Delete	Discover	Read	Workspace	Delete	Update	Tag
Anonymous	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
User/group to add: <input type="text"/> <input type="button" value="Add"/>																

Diagnostics page:

Jenkins | Who Am I

Who Am I?

Name: chandu
IsAuthenticated?: true
Authorities:
• "chandu"
• "authenticated"

Details:
toString: org.acegisecurity.ui.WebAuthenticationDetails@59b2; RemoteIpAddress: 192.168.181.1; SessionId: 184616D40A6CF98A5482A8B96F23DF7D
org.acegisecurity.providers.UsernamePasswordAuthenticationToken@6859cdf: Username: org.acegisecurity.userdetails.User@0; Username: chandu; Password: [PROTECTED]; Enabled: true; AccountNonExpired: true; credentialsNonExpired: true; AccountNonLocked: true; Granted Authorities: chandu, authenticated; Password: [PROTECTED]; Authenticated: true; Details: org.acegisecurity.ui.WebAuthenticationDetails@59b2; RemoteIpAddress: 192.168.181.1; SessionId: 184616D40A6CF98A5482A8B96F23DF7D; Granted Authorities: chandu, authenticated

Request Headers

host	192.168.181.129:8080
user-agent	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:34.0) Gecko/20100101 Firefox/34.0
accept	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
accept-language	en-US,en;q=0.5
accept-encoding	gzip, deflate
referer	http://192.168.181.129:8080/jenkins/job/helloworld/configure
cookie	iconSize=32x32; JSESSIONID=E54264348E3D56F0C5828BB179847373; screenResolution=1525x858
connection	keep-alive

- **This build is parameterized:**

Sometimes, it is useful/necessary to have your builds take several "parameters." Consider the following use case:

- You set up a test job on Jenkins, and it accepts a distribution bundle as a parameter and perform tests against it. You want to have developers do local builds and let them submit builds for test execution on Jenkins. In such a case, your parameter is a zip file that contains a distribution.
- Your test suite takes so much time to run that in normal execution you can't afford to run the entire test cycle. So you want to control the portion of the test to be executed. In such a case, your parameter is perhaps a string token that indicates that test suite to be run.

First, you need to define parameters for your job by selecting "This build is parameterized", then using the drop-down button to add as many parameters as you need.

File parameter allows a build to accept a file, to be submitted by the user when scheduling a new build. The file will be placed inside the workspace at the known location after the check-out/update is done, so that your build scripts can use this file.



- **Disable Build (No new builds will be executed until the project is re-enabled.)**
Sometimes you want to temporarily stop building a project. For example, maybe you are in the middle of a large migration and you know new builds are going to fail. Or maybe a project is built every hour but you know that the CVS server will be down for the next 24 hours.

When this option is set, no new build is performed on this project. This allows you to disable new builds without changing any of the build dependency chain or changing the notification set up.

- **Execute concurrent builds if necessary**
If this option is checked, Jenkins will schedule and execute multiple builds concurrently (provided that you have sufficient executors and incoming build requests.) This is useful on builds and test jobs that take a long time, as each build will only contain a smaller number of changes, and the total turn-around time decreases due to the shorter time a build request spends waiting for the previous build to complete. It is also very useful with parameterized builds, whose individual executions are independent from each other.

For other kinds of jobs, allowing concurrent executions of multiple builds may be problematic, for example if it assumes a monopoly on a certain resource, like database, or for jobs where you use Jenkins as a cron replacement.

If you use a custom workspace and enable this option, all your builds will run on the same workspace, thus unless a care is taken by your side, it'll likely to collide with each other. Otherwise, even when they are run on the same node, Jenkins will use different workspaces to keep them isolated.

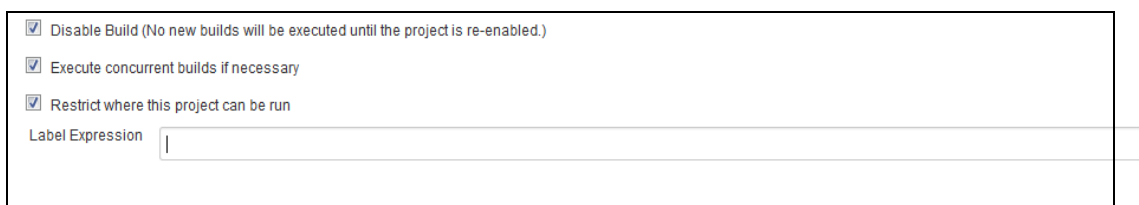
When Jenkins creates different workspaces for isolation, Jenkins appends "@num" to the workspace directory name, e.g. "@2". The separator "@" can be configured by setting the system property "hudson.slaves.WorkspaceList" to the desired separator string on the Jenkins command line. E.g. "-Dhudson.slaves.WorkspaceList=-" will use a dash as separator.

- **Restrict where this project can be run**

Sometimes a project can only be successfully built on a particular slave (or master). If so, this option forces Jenkins to always build this project on a specific computer. If there is a group of machines that the job can be built on, you can specify that label as the node to tie on, which will cause Jenkins to build the project on any of the machines with that label.

Otherwise, uncheck the box so that Jenkins can schedule builds on available nodes, which results in faster turn-around time.

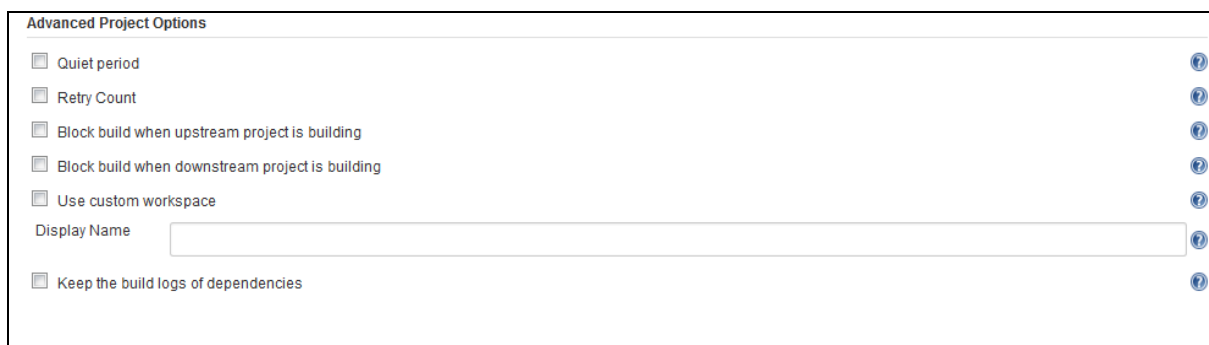
This option is also useful when you'd like to make sure that a project can be built on a particular node.



A screenshot of the Jenkins configuration interface. It shows three checked options: 'Disable Build (No new builds will be executed until the project is re-enabled.)', 'Execute concurrent builds if necessary', and 'Restrict where this project can be run'. Below these is a text input field labeled 'Label Expression' with a vertical cursor.

Advanced Project Options:

The Advanced Project options contains, as the name suggests, configuration options that are less frequently required. You need to click on the advanced button for them to appear. (To display the Advanced Options, you need to click on the Advanced button”).



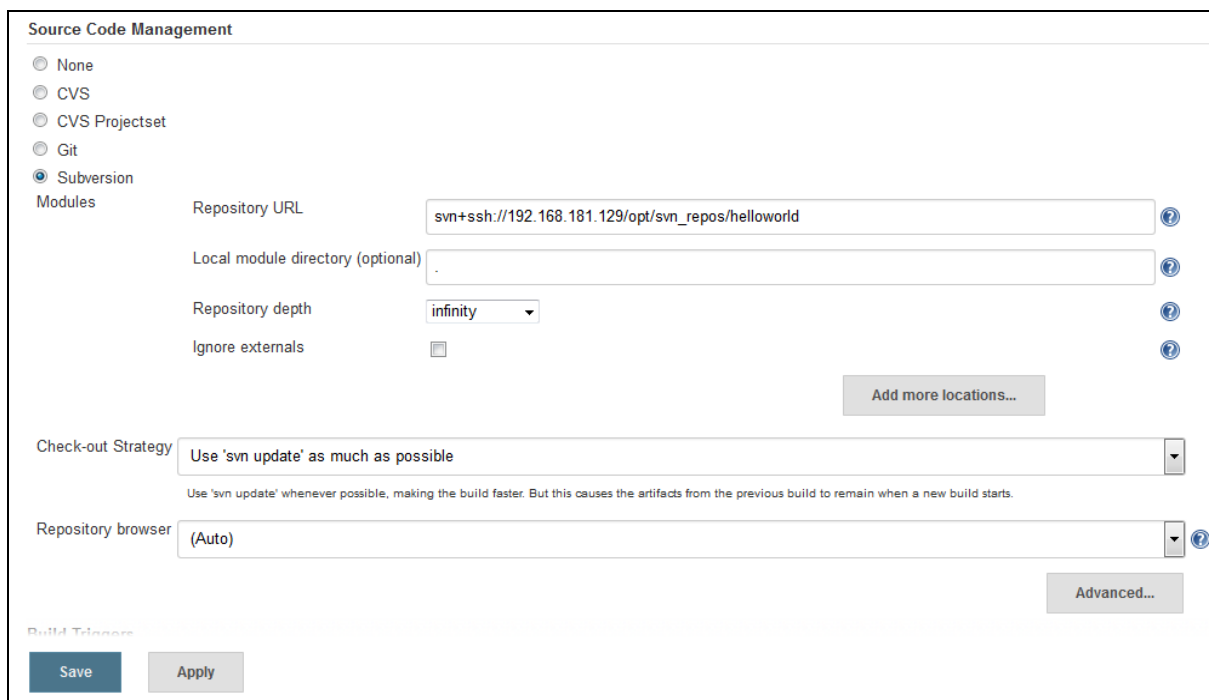
A screenshot of the 'Advanced Project Options' section in Jenkins. It contains several unchecked checkboxes: 'Quiet period', 'Retry Count', 'Block build when upstream project is building', 'Block build when downstream project is building', 'Use custom workspace', and 'Keep the build logs of dependencies'. There is also a text input field labeled 'Display Name'.

The **Quiet Period** option in the build job configuration simply lets you override the system-wide quiet period defined in the Jenkins System Configuration screen. This option is mainly used for version control systems that don't support atomic commits, such as CVS, but it is also sometimes used in teams where developers have the habit of committing their work in several small commits.

Source Code Management:

Our job will use Subversion (SVN) to retrieve/checkout the source code/project from the repository. In a real world project you probably will use a SCM (SVN, Git, CVS, etc) to store your test automation project.

To configure SCM tools, you need to have the Repository URL and credentials to checkout the project source code from the repository.



The screenshot shows the 'Source Code Management' configuration page in Jenkins. Under the 'Subversion' radio button, the 'Repository URL' is set to 'svn+ssh://192.168.181.129/opt/svn_repos/helloworld'. The 'Local module directory (optional)' is set to '.'. The 'Repository depth' is set to 'infinity'. The 'Ignore externals' checkbox is unchecked. There is an 'Add more locations...' button. The 'Check-out Strategy' is set to 'Use 'svn update' as much as possible', with a note below it: 'Use 'svn update' whenever possible, making the build faster. But this causes the artifacts from the previous build to remain when a new build starts.' The 'Repository browser' is set to '(Auto)'. There is an 'Advanced...' button. At the bottom, there are 'Save' and 'Apply' buttons.

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 four basic ways a build job can be triggered.



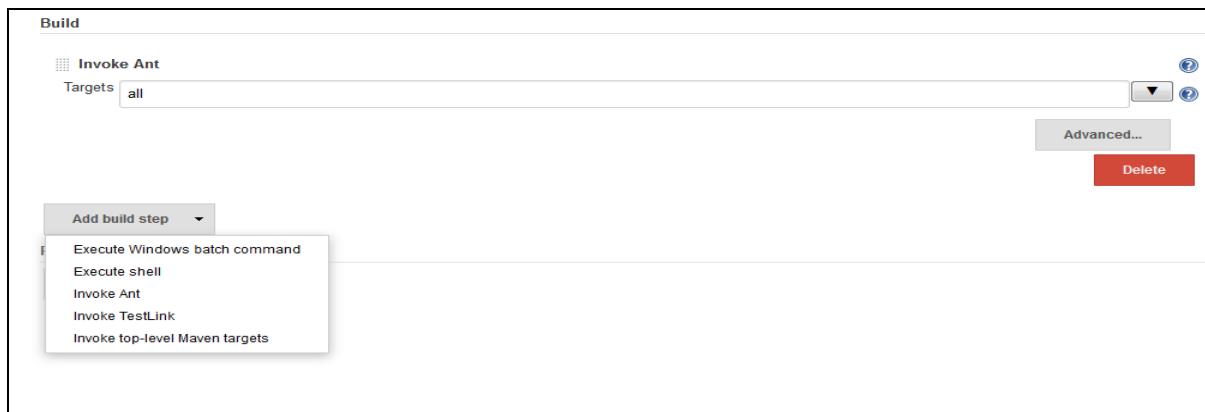
The screenshot shows the 'Build Triggers' configuration page in Jenkins. It contains four checkboxes, all of which are unchecked: 'Trigger builds remotely (e.g., from scripts)', 'Build after other projects are built', 'Build periodically', and 'Poll SCM'. Each checkbox has a help icon to its right.

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.

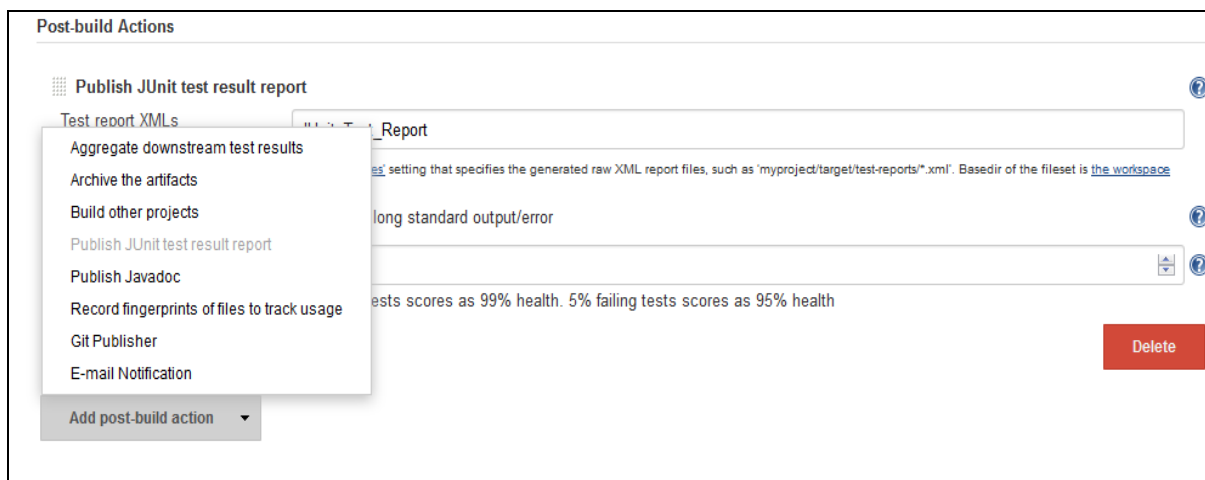
A build job may have one step, or more. It may even occasionally have none. In a freestyle build, you can add as many build steps as you want to the Build section of your project configuration. In a basic Jenkins installation, you will be able to add steps to invoke Maven and Ant, as well as running 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.

Click on *Add build step* button to expand its options and then click on *Invoke Ant*. It will show a new form with options to integrate your Jenkins job with Ant. The plug-in configuration screen contains text area to provide the target names.



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.

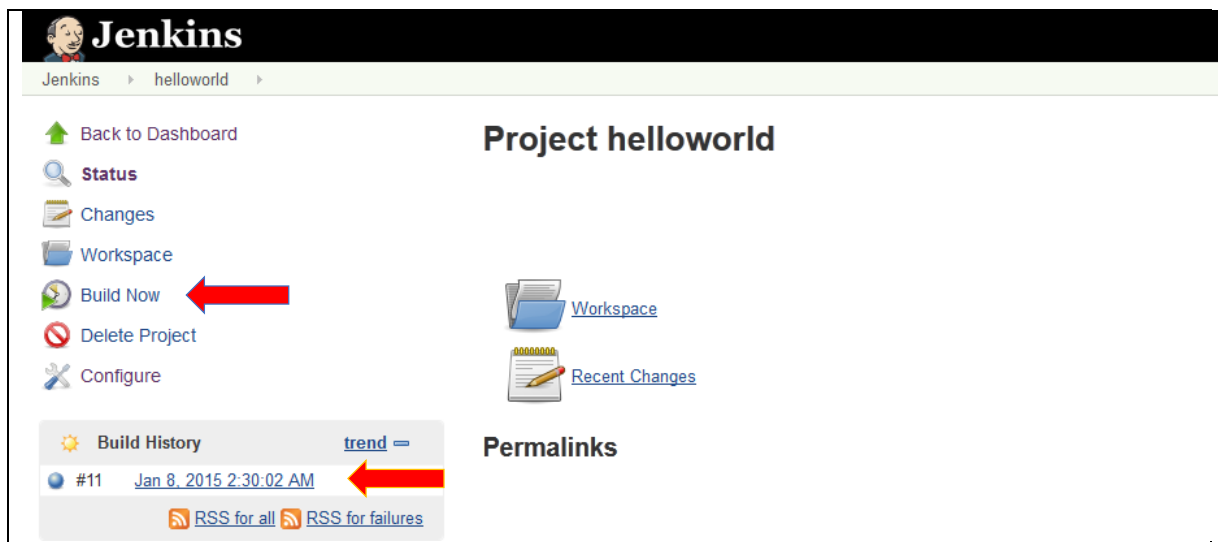


One of the most obvious requirements of a build job is to report on test results. Not only whether there are any test failures, but also how many tests were executed, how long they took to execute, and so on. In the Java world, JUnit is the most commonly-used testing library

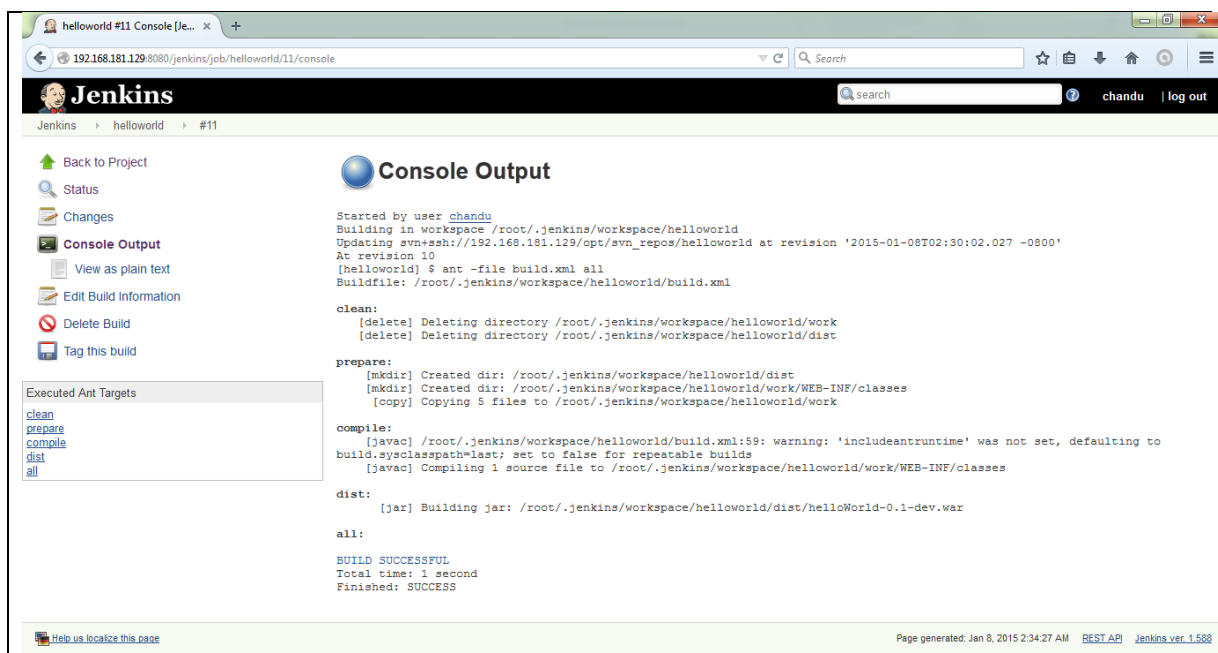
around, and the JUnit XML format for test results is widely used and understood by other tools as well.

Running your new Build Job:

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



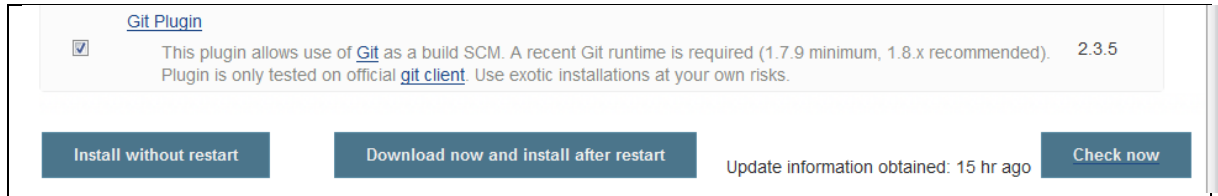
Click on the Build Now button to start the build process. Once build succeed click on the item that was build. If you want to know the execution process go to console output, here you will find the entire execution report for the job you executed and if build fails, you can find the error log here to resolve it.



Integrating GIT with Jenkins:

Go to **Manage Jenkins** → **Manage Plugin** → **Available**

In the available section you will get the list available plugins, here you can select the plugin that you want to install. In this scenario we are heading with Git Plugin, so select the git plugin and click on install without restart.

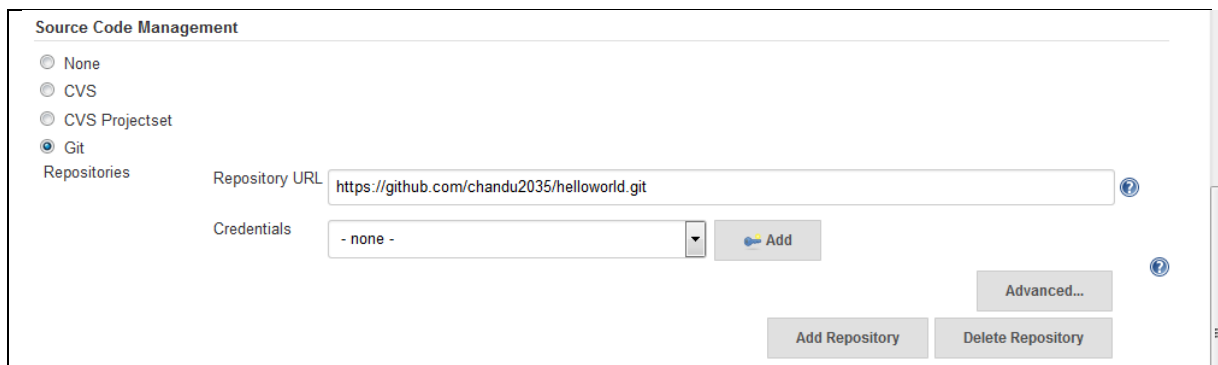


Once, if it installed successfully, then you need to restart the Jenkins to take effect changes.

Configuring Jobs using GIT Repository:

You can configure the jobs using git repository. Select the Git radio button in the Source Code Management section. Here you need to pass the URL of the git repository and credentials.

You can add multiple repositories by clicking on the Add Repository button.



Adding Build Step:

Build step involves to invoke the build scripts that are available for the current project. This build scripts will be available in the repository. In this demo project we are using Ant.



When you click on invoke Ant it will ask you to select the version of ant, and the targets, so if you have configured multiples ant versions with Jenkins, then here you need specify the version of ant to be used.

If you configured single Ant version with Jenkins, automatically it will consider it as a default.

The screenshot shows the 'Ant' configuration page in Jenkins. It has a title 'Ant' and a subtitle 'Ant installations'. There are two entries for Ant installations. Each entry has a 'Name' field, an 'ANT_HOME' field, and an 'Install automatically' checkbox. The first entry has 'Name' as 'Ant 1.9.4' and 'ANT_HOME' as '/opt/apps/ant'. The second entry has 'Name' as 'Ant 1.4' and 'ANT_HOME' as '/opt/apps/ant1.4'. Both entries have a 'Delete Ant' button. At the bottom, there is an 'Add Ant' button and a link 'List of Ant installations on this system'.

Deploying the war/ear files to Application Server:

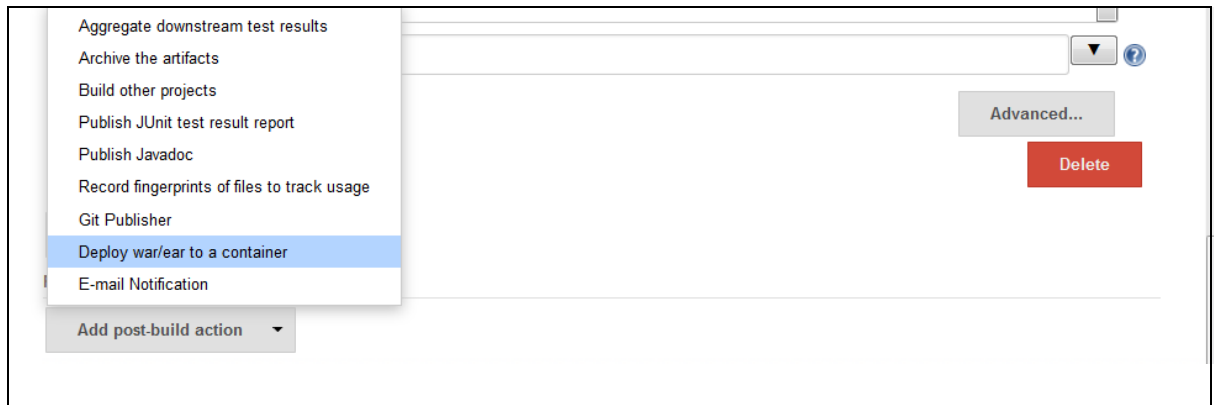
Once if build succeeded, you will get the projects artifacts like jar/war/ear. So if you want to deploy the war/ear files to your Application server, you need to install the deployment plugins depending on the Application server that you are using. In this case, Tomcat is the Application server, which is used to run the web applications.

Step 1: Install the Deploy plugin, and restart the Jenkins server.

The screenshot shows the 'Deploy Plugin' page in Jenkins. It has a title 'Deploy Plugin' and a subtitle 'This plugin takes a war/ear file and deploys that to a running remote application server at the end of a build'. There is a 'Deploy WebSphere Plugin' link. At the bottom, there are two buttons: 'Install without restart' and 'Download now and install after restart'. To the right, there is a link 'Update information obtained: 47 min ago' and a 'Check now' button.

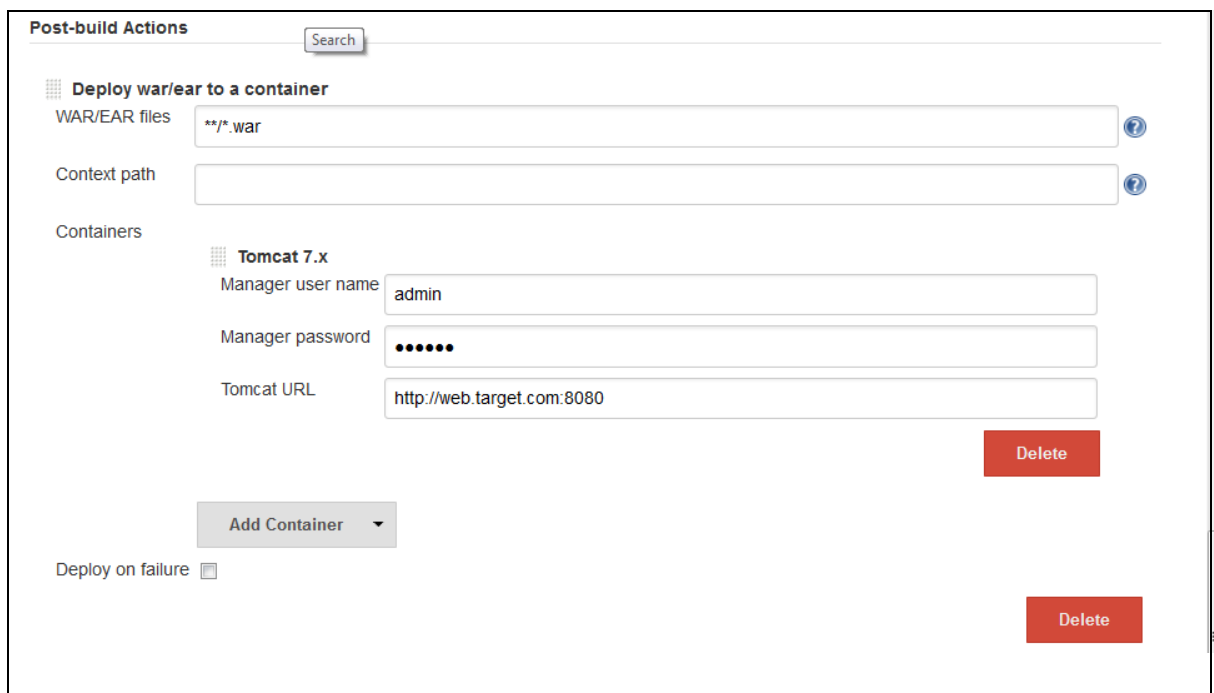
Now you will get the option called **Deploy war/ear to a container** in the post build actions.

Step 2: Click on Add post build action, then select the Deploy war/ear to a container.



Step 3:

- In war/ear file section, tell the Jenkins where your project artifacts are available after performing the build operation.
- Context path can be used to represent the name in the URL of your application.
- In the Containers section you need to select the type and version of the Application server.
- After selecting the server, it will ask the username, password and URL of the server.



In the similar way, you can add multiple containers to it, if you are using multiple type of servers or multiple instances of a single server.

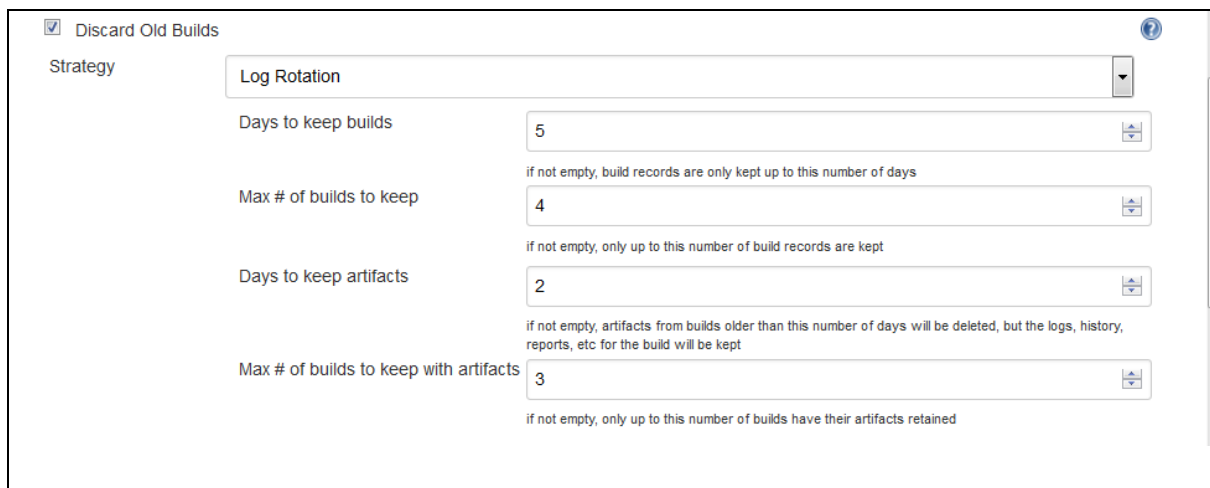
Now save your job configuration and click on the Build option, that will be available in the left pane of the Jenkins job page.

Once build is succeeded, it will deploy your war files to the Tomcat server.

Configuring builds with several options

- **Discard old builds:**

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. Note that this button will only appear if you have asked Jenkins to discard old builds.



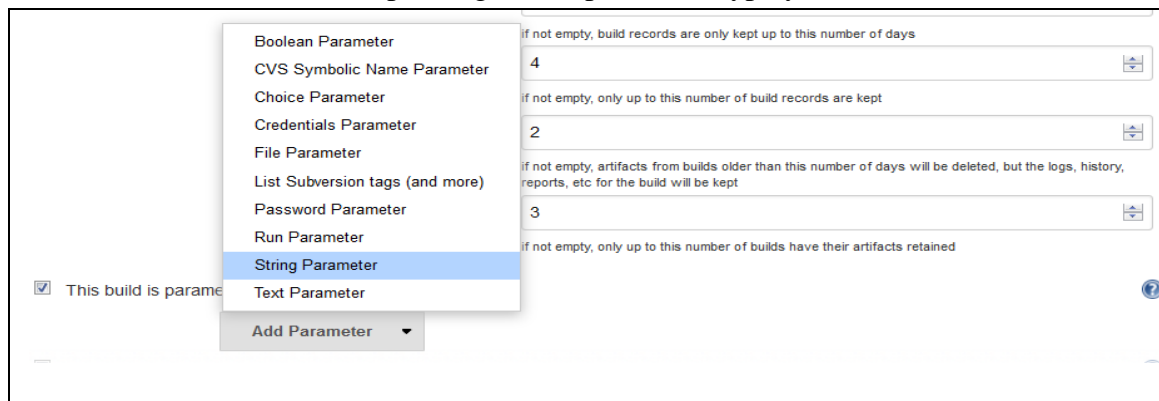
The screenshot shows the 'Discard Old Builds' configuration page in Jenkins. The 'Discard Old Builds' checkbox is checked. The 'Strategy' dropdown is set to 'Log Rotation'. There are four input fields with their respective values and descriptions:

Field	Value	Description
Days to keep builds	5	if not empty, build records are only kept up to this number of days
Max # of builds to keep	4	if not empty, only up to this number of build records are kept
Days to keep artifacts	2	if not empty, artifacts from builds older than this number of days will be deleted, but the logs, history, reports, etc for the build will be kept
Max # of builds to keep with artifacts	3	if not empty, only up to this number of builds have their artifacts retained

- **This build is parameterized:**

First, you need to define parameters for your job by selecting "This build is parameterized", then using the drop-down button to add as many parameters as you need.

There are different parameter types available, and it is extensible, too. The way parameters take effect is also different depending on the parameter type you choose.

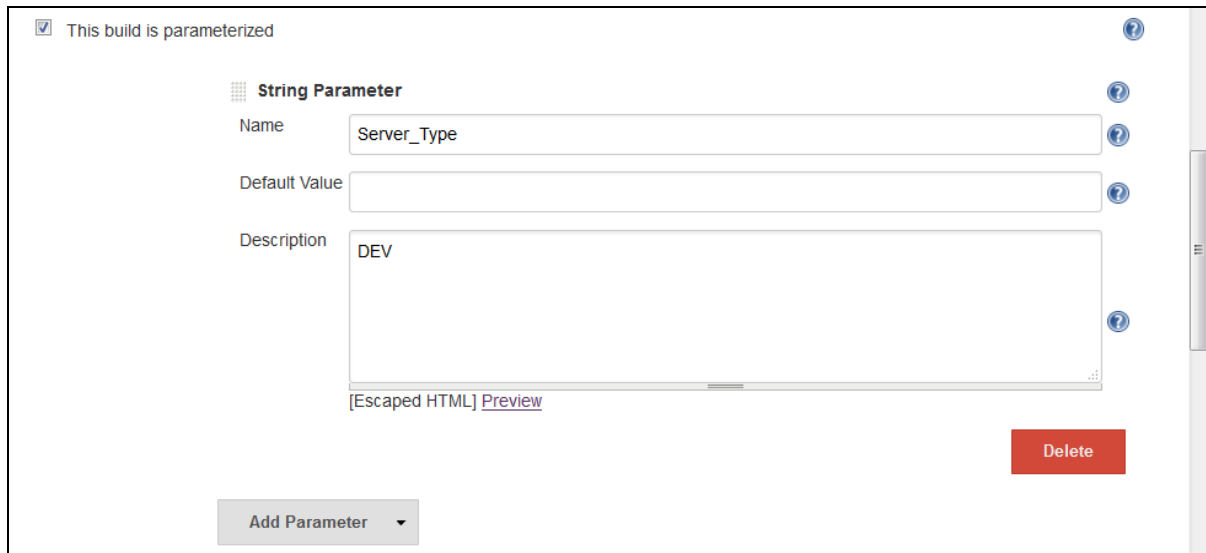


The screenshot shows the 'This build is parameterized' configuration page in Jenkins. The 'This build is parameterized' checkbox is checked. A dropdown menu is open, showing various parameter types. The 'String Parameter' option is highlighted. The background shows the same 'Discard Old Builds' configuration as the previous screenshot.

Parameter Type
Boolean Parameter
CVS Symbolic Name Parameter
Choice Parameter
Credentials Parameter
File Parameter
List Subversion tags (and more)
Password Parameter
Run Parameter
String Parameter
Text Parameter
Add Parameter

String parameters are exposed as environment variables of the same name. Therefore, a builder, like Ant and Shell, can use the parameters. Continuing the above example, the following is a simple example:

1. Reference parameter by name in builder. I'm using the "Server_Type" command to build this job on the remote servers.



☒ This build is parameterized

String Parameter

Name:

Default Value:

Description:

[Escaped HTML] [Preview](#)

[Delete](#)

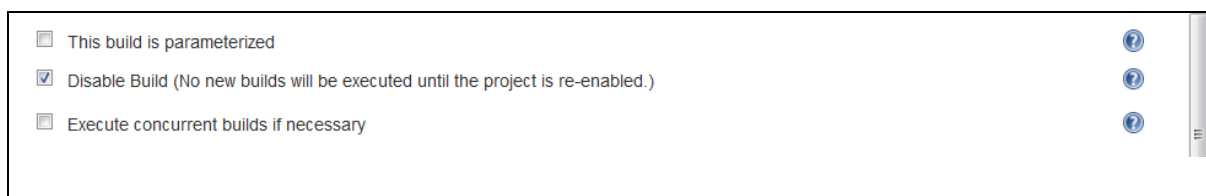
[Add Parameter](#)

In the similar way you can use different types of parameters within your job configuration according to your requirement.

- **Disable Build:**

Sometimes you want to temporarily stop building a project. For example, maybe you are in the middle of a large migration and you know new builds are going to fail. Or maybe a project is built every hour but you know that the GIT server will be down for the next 24 hours.

When this option is set, no new build is performed on this project. This allows you to disable new builds without changing any of the build dependency chain or changing the notification set up.



☐ This build is parameterized

☒ **Disable Build** (No new builds will be executed until the project is re-enabled.)

☐ Execute concurrent builds if necessary

Changes can be reflected in your job page, so here you will get an option to enable it. If click on enable option, automatically it will deselect / uncheck the option **Disable build** in job configuration.

Jenkins > Hello_World

[Back to Dashboard](#)

[Status](#)

[Changes](#)

[Workspace](#)

[Delete Project](#)

[Configure](#)

Project Hello_World

This project is currently disabled [Enable](#)

[Workspace](#)

[Recent Changes](#)

Build History

[trend](#)

Build Number	Timestamp
#14	May 7, 2015 7:53 AM
#11	May 7, 2015 7:49 AM
#10	May 7, 2015 2:48 AM

Permalinks

- [Last build \(#14\), 12 min ago](#)
- [Last stable build \(#14\), 12 min ago](#)
- [Last successful build \(#14\), 12 min ago](#)

- **Execute concurrent builds if necessary:**

If this option is checked, Jenkins will schedule and execute multiple builds concurrently (provided that you have sufficient executors and incoming build requests.) This is useful on builds and test jobs that take a long time, as each build will only contain a smaller number of changes, and the total turn-around time decreases due to the shorter time a build request spends waiting for the previous build to complete. It is also very useful with parameterized builds, whose individual executions are independent from each other.

For other kinds of jobs, allowing concurrent executions of multiple builds may be problematic, for example if it assumes a monopoly on a certain resource, like database, or for jobs where you use Jenkins as a cron replacement.

If you use a custom workspace and enable this option, all your builds will run on the same workspace, thus unless a care is taken by your side, it'll likely to collide with each other. Otherwise, even when they are run on the same node, Jenkins will use different workspaces to keep them isolated.

When Jenkins creates different workspaces for isolation, Jenkins appends "*@num*" to the workspace directory name, e.g. "@2". The separator "@" can be configured by setting the system property "hudson.slaves.WorkspaceList" to the desired separator string on the Jenkins command line. E.g. "-Dhudson.slaves.WorkspaceList=-" will use a dash as separator.

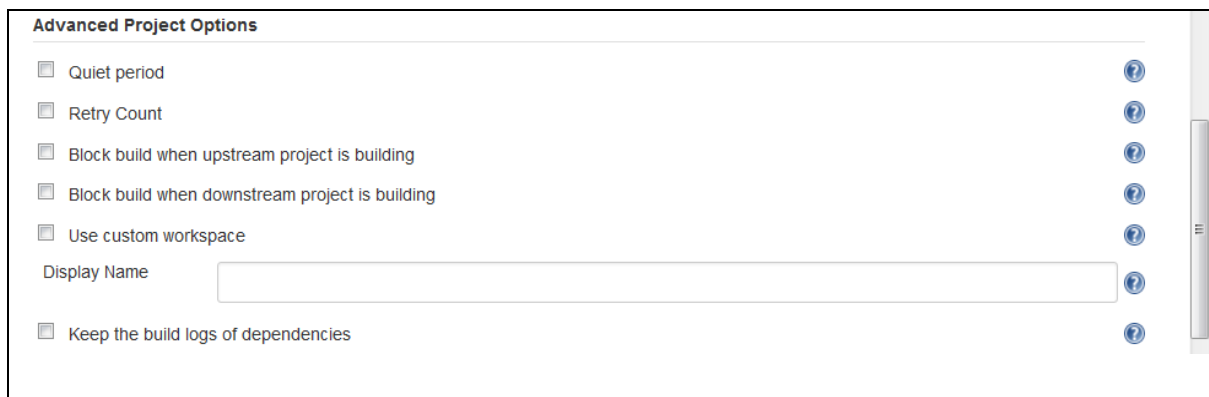
☒ Execute concurrent builds if necessary

[?](#)

Advanced Project Options

Advanced Project Options:

Next section is Advanced project options, this section is by default in hiding mode, but can able to the options by clicking on the advanced option in this section.




- **Quiet period:**

Commits often come in a burst. This seems to happen mainly for two reasons --- people sometimes forget to commit some files, and in the tranquility of waiting for your SCM to finish a commit, people sometimes realize the problems in the commit and they quickly make follow-up changes. The conventional wisdom is that the CI server should wait for the burst to finish before attempting a build. This is said to reduce the chance of having broken build, and it is also sometimes useful in reducing the average turn-around time for builds that take longer.

As such, Hudson is capable of waiting for a commit burst to be over before it triggers a new build, and this feature is called "quiet period." There are two parts in Hudson that interacts with the quiet period. One is the SCM polling behavior and the other is the queue.

The queue portion of the quiet period is straight-forward. When a build is scheduled into the queue with quiet period, the build will sit in the queue until the quiet period expires. If during this period, additional attempts are made to put the same build in the queue, the quiet period resets to its initial value. For example, if the quiet period is 5 minutes, and the build is put into the queue 9:00am and 9:03am, the actual build will only happen after 9:08am. Thus another way to think of the quiet period is that you are requiring a certain period of inactivity.



The above applies to all the mechanisms in Hudson that puts builds into the queue. This includes REST API call, CLI call, downstream triggers, and SCM pollings. So if you implement some kind of a "push" mechanism in your SCM to notify Hudson of a new commit, then you get the desired effect by just setting the quiet period in Hudson, and those push scripts don't have to do anything tricky.

It is also possible for some of those to override the quiet period configured in the project. For example, when you click "Build Now" button in your browser, your browser is making a REST API call, but with the quiet period of zero. I used to run a "push" script that looks into a commit message and overrides the quiet period by taking advantages of this feature.

- **Retry count:**

If a build fails to checkout from the repository, Jenkins will retry the specified number of times before giving up.



A screenshot of a Jenkins configuration panel. It features a checkbox labeled "Retry Count" which is checked. Below it is a text input field labeled "SCM checkout retry count" with the value "2" entered. A help icon is visible in the top right corner.

- **Block build when upstream project is building:**

When this option is checked, Jenkins will prevent the project from building when a dependency of this project is in the queue, or building. The dependencies include the direct as well as the transitive dependencies.

- **Block build when downstream project is building:**

When this option is checked, Jenkins will prevent the project from building when a child of this project is in the queue, or building. The children include the direct as well as the transitive children.



A screenshot of a Jenkins configuration panel showing two unchecked checkboxes: "Block build when upstream project is building" and "Block build when downstream project is building". Help icons are present in the top right corner.

- **Use custom workspace:**

By default Jenkins workspace will be user's home directory (.jenkins) or If you explicitly provide the environment variable \$JENKINS_HOME then Jenkins will store the all things under that directories.

But if you want to change the workspace for a particular job, then you can use the specific path for that.




A screenshot of a Jenkins configuration panel. The "Use custom workspace" checkbox is checked. Below it are two text input fields: "Directory" with the value "/home/chandu/" and "Display Name" with the value "chandu". Help icons are visible in the top right corner.

- **Keep the build logs of dependencies:**

If this option is enabled, all of the builds that are referenced from builds of this project (via fingerprint) will be protected from log rotation.

When your job depends on other jobs on Jenkins and you occasionally need to tag your workspace, it's often convenient/necessary to also tag your dependencies on Jenkins. The problem is that the log rotation could interfere with this, since the build your project is using might already be log rotated (if there have been a lot of builds in your dependency), and if that happens you won't be able to reliably tag your dependencies.

This feature fixes that problem by "locking" those builds that you depend on, thereby guaranteeing that you can always tag your complete dependencies.

A screenshot of a Jenkins configuration option. It shows a single row with a checked checkbox labeled "Keep the build logs of dependencies" and a help icon (question mark in a circle) to its right.

Build Triggers section:

This section allows you execute the jobs automatically depending upon the configuration made to them.

A screenshot of the "Build Triggers" section in Jenkins. It has a title bar "Build Triggers" with a help icon. Below it are three options, all with unchecked checkboxes: "Build after other projects are built", "Build periodically", and "Poll SCM". Each option has a help icon to its right.

- **Build after other projects are built:**

Set up a trigger so that when some other projects finish building, a new build is scheduled for this project. This is convenient for running an extensive test after a build is complete, for example.

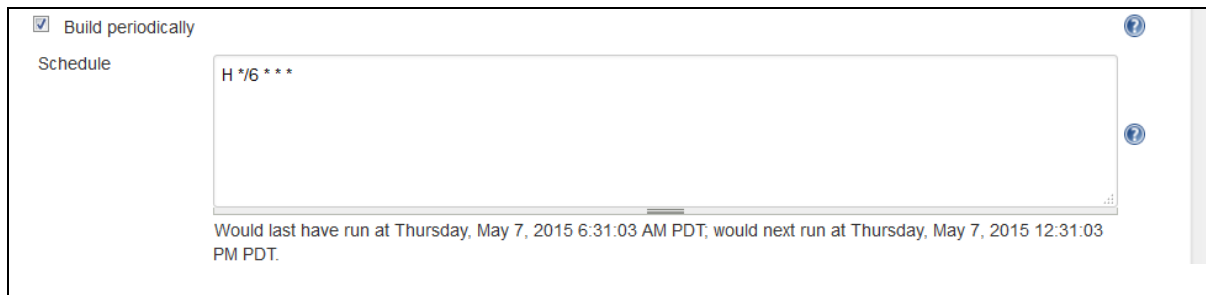
This configuration complements the "Build other projects" section in the "Post-build Actions" of an upstream project, but is preferable when you want to configure the downstream project.

A screenshot of the "Build Triggers" section in Jenkins, specifically showing the "Build after other projects are built" option selected. The title bar is "Build Triggers" with a help icon. The first option, "Build after other projects are built", is checked and has a help icon. Below it is a text input field labeled "Projects to watch" containing the text "Demo_Project". At the bottom are three radio button options: "Trigger only if build is stable" (which is selected), "Trigger even if the build is unstable", and "Trigger even if the build fails".

- **Build periodically:**

Provides a cron-like feature to periodically execute this project. So, before using this feature, stop and ask yourself if this is really what you want.

This feature is primarily for using Jenkins as a cron replacement, and it is **not ideal for continuously building software projects**. When people first start continuous integration, they are often so used to the idea of regularly scheduled builds like nightly/weekly that they use this feature. However, the point of continuous integration is to start a build as soon as a change is made, to provide a quick feedback to the change.

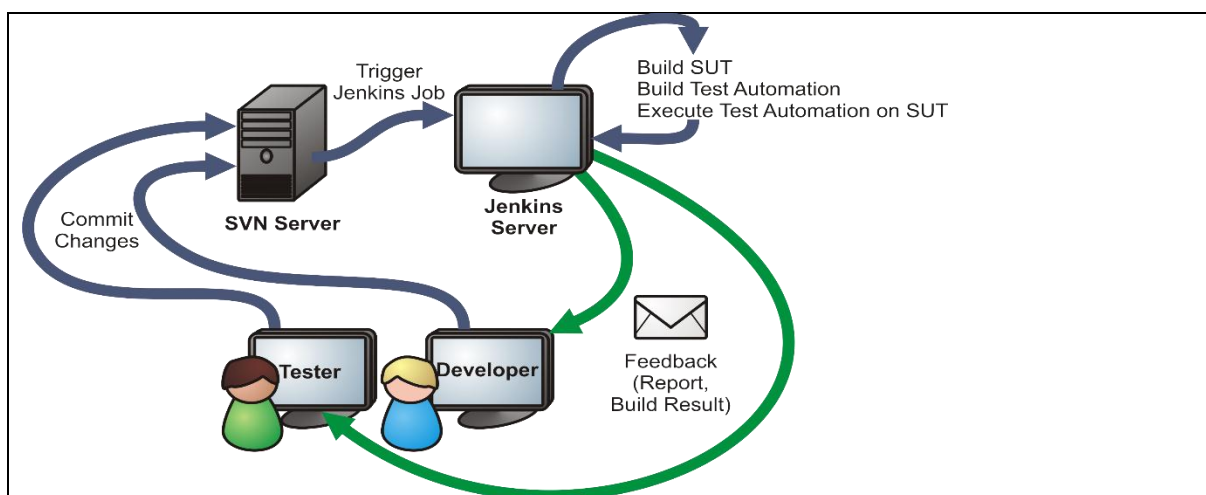


The screenshot shows the Jenkins configuration page for a job. The 'Build periodically' checkbox is checked. Below it, the 'Schedule' field contains the cron expression 'H */6 ***'. A status message at the bottom indicates the last run was on Thursday, May 7, 2015 at 6:31:03 AM PDT, and the next run is on Thursday, May 7, 2015 at 12:31:03 PM PDT.

In the above example, I have given a value to build this job automatically for every six hours.

- **Poll SCM:**

Configure Jenkins to poll changes in SCM. Note that this is going to be an expensive operation for CVS or SVN or GIT, as every polling requires Jenkins to scan the entire workspace and verify it with the server. Consider setting up a "push" trigger to avoid this overhead.



For example, if any commits happened to repository then Jenkins will checkup the SCM and start the build process. You can configure cron job syntax to check the SCM.

☒ Poll SCM

Schedule

H/15 * * * *

Would last have run at Thursday, May 7, 2015 9:30:21 AM PDT; would next run at Thursday, May 7, 2015 9:45:21 AM PDT.

☐ Ignore post-commit hooks

In the above screenshot I have mentioned the value **H/15 * * * ***, It means every 15 minutes it will check the SCM and if there are any commits happened then it will start the build.

Adding Build Step:

In this section we can invoke the various build tools like Ant, Maven, Windows Batch script and Shell.

Build

Add build step

- Execute Windows batch command
- Execute shell
- Invoke Ant
- Invoke top-level Maven targets

Invoking Ant:

Click on Add build step then select the Invoke Ant option in the drop down menu.

Build

Invoke Ant

Ant Version: Default

Targets:

Advanced...

Delete

In the Ant version, you need to select the version of Ant you want use, or default means it can be used the Ant which is configured in the configure system.

In Target section you need to specify which targets to be executed once build is triggered.

By default it will take build.xml as an Ant script, so if your Ant script file name is having some other name than build.xml you need specify that file name. To do that click on Advanced button

then you will get Build File, Properties, and Java Options. According to your requirement and configurations you need to use the options here.

The screenshot shows the 'Build' configuration page in Jenkins. It features a section titled 'Invoke Ant' with several input fields: 'Ant Version' (set to 'Default'), 'Targets' (set to 'all'), 'Build File', 'Properties', and 'Java Options'. Each field has a dropdown arrow and a help icon. A red 'Delete' button is located at the bottom right of the section.

Adding Post Build Actions:

Post build actions can be performed once build step is completed. In this section we can configure many useful things, like deployments, publishing the documents, publishing the reports, job dependencies, E-mail notifications and etc.

The screenshot shows the 'Post-build Actions' section in Jenkins. A dropdown menu is open, displaying various actions such as 'Aggregate downstream test results', 'Archive the artifacts', 'Build other projects', 'Publish JUnit test result report', 'Publish Javadoc', 'Record fingerprints of files to track usage', 'Git Publisher', 'Deploy war/ear to a container', and 'E-mail Notification'. The 'E-mail Notification' option is highlighted. Below the menu is a button labeled 'Add post-build action'. To the right, there are 'Advanced...' and 'Delete' buttons.

Configuring E-mail notifications:

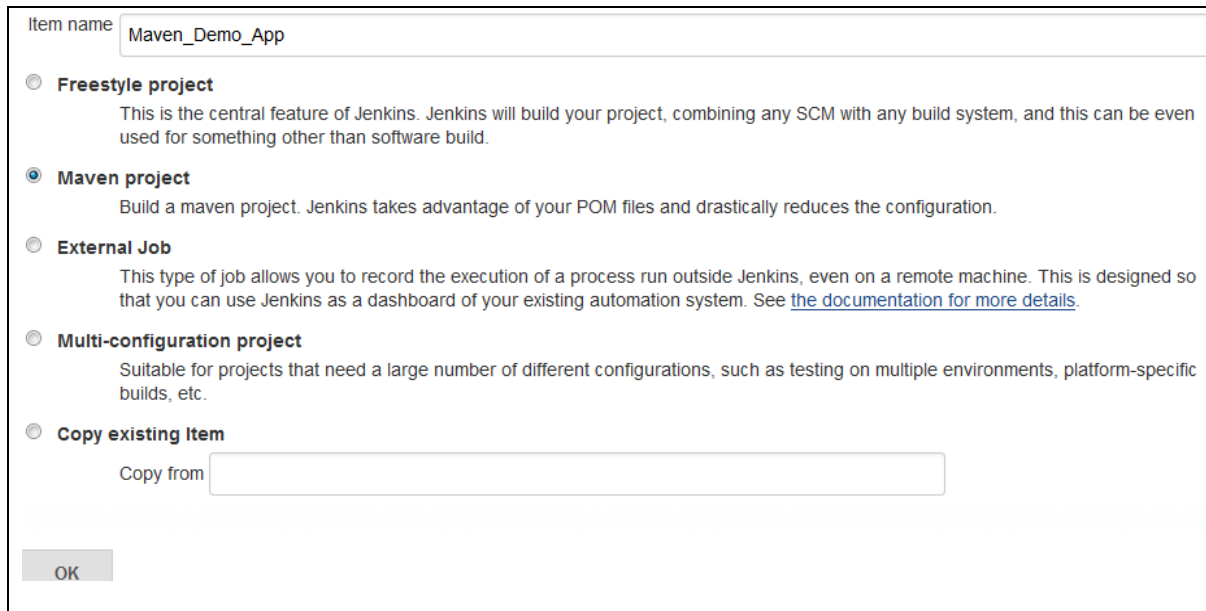
In the configure system section we will configure the E-mail server and all, so here you can simply specify the receptionist email. If build is failed, then the specified users can get the notifications via E-mail.

The screenshot shows the 'E-mail Notification' configuration page in Jenkins. It includes a 'Recipients' field with the text 'chandu.2035@gmail.com, guru123@gmail.com'. Below this, there is a note: 'Whitespace-separated list of recipient addresses. May reference build parameters like \$PARAM. E-mail will be sent when a build fails, becomes unstable or returns to stable.' There are two checkboxes: 'Send e-mail for every unstable build' (checked) and 'Send separate e-mails to individuals who broke the build' (unchecked). A red 'Delete' button is at the bottom right.

Configuring Maven Jobs

By default Maven-plugin will come with Jenkins, so directly we can able to configure the maven jobs without installing any plugins.

Whenever you click on new job / new item you will get the job types list, so you can select the job type that you want to create.



Item name: Maven_Demo_App

☐ Freestyle project
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

☒ **Maven project**
Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

☐ External Job
This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system. See [the documentation for more details](#).

☐ Multi-configuration project
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

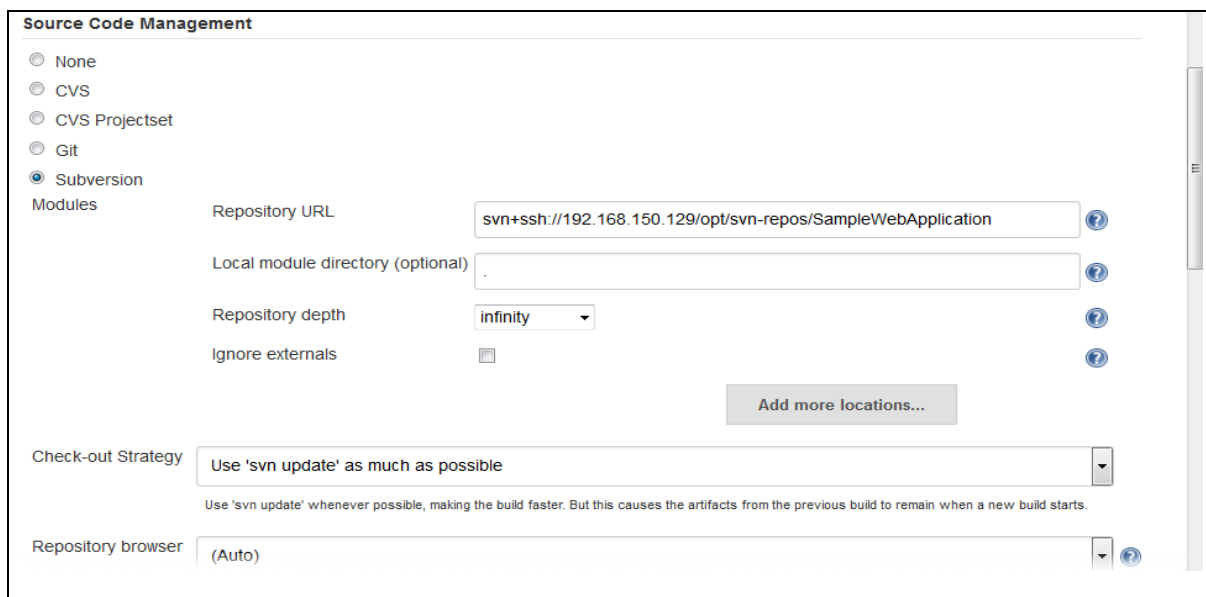
☐ Copy existing Item
Copy from:

OK

When you click on Ok button, job will be created and it will take you into the job configuration page.

In the previous section we have seen how to configure jobs, but here some extra options will be added for the maven projects.

Adding SVN (Subversion) Repository:



Source Code Management

☐ None
☐ CVS
☐ CVS Projectset
☐ Git
☒ **Subversion**

Modules

Repository URL:

Local module directory (optional):

Repository depth:

Ignore externals: ☐

Add more locations...

Check-out Strategy:

Repository browser:

Previously we have used GIT SCM Tool, now we have used SVN for the Maven Project.

Once if you pass the repository URL then will ask for the credentials, so keep watch it and add the credentials that you are using for repository.

Build Triggers:



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

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 cool tools. Since these two projects are being developed by the same team, you are using some of the latest cool tools 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.

```
<dependencies>
  <dependency>
    <groupId>com.target.web</groupId>
    <artifactId>cooltools</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <scope>test</scope>
  </dependency>
  ...
</dependencies>
```

On your Jenkins server, you have set up Maven build jobs for both the cooltools and the game-of-life applications. 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, the game-of-life project will automatically be rebuilt as well.

Adding Pre-Build setup:

This section allows you to perform pre-build activities, here you can invoke Ant, Shell and etc.



Pre Steps

Invoke top-level Maven targets

Maven Version: Maven 3.2.3

Goals: clean

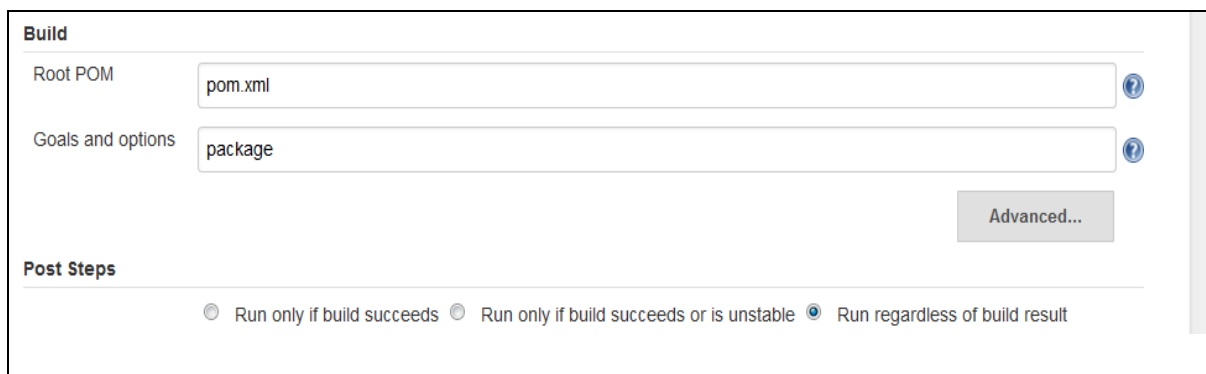
Advanced...

Delete

In the above example, before performing the build maven performs the clean operation in the workspace.

Build Step:

In the build step section you need to specify the maven build file (**pom.xml**), then which goals to be executed.



Build

Root POM: pom.xml

Goals and options: package

Advanced...

Post Steps

☐ Run only if build succeeds ☐ Run only if build succeeds or is unstable ☒ Run regardless of build result

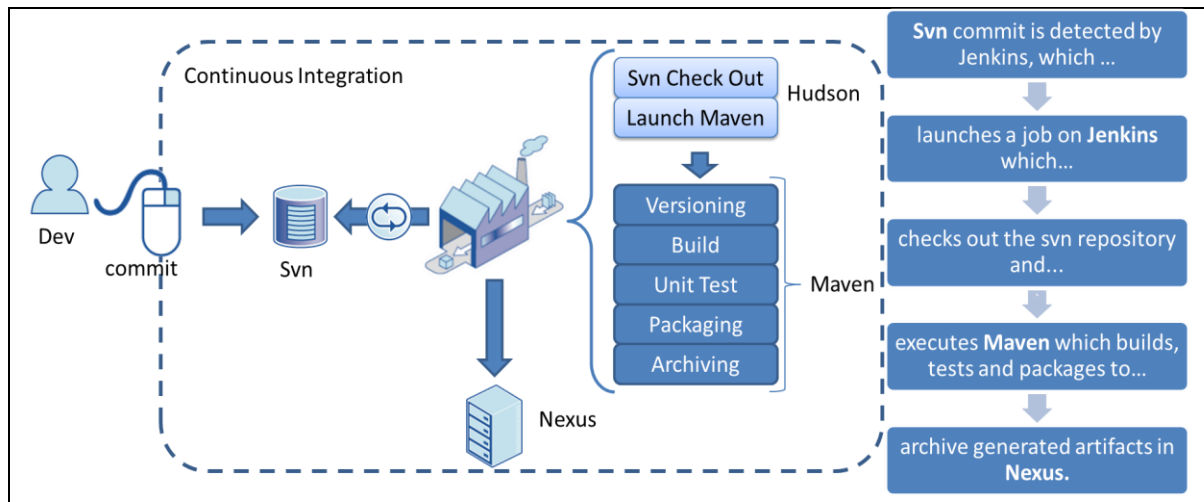
In Maven package goal will perform an action to build the project, if you have a good knowledge on maven it will be easier to configure maven jobs in Jenkins.

Post Steps:

This section allows to run the post build steps, depending on the build operation. In the above figure there are three options available to configure post steps.

Jenkins workflow:

Jenkins workflow with svn and maven.



Distributed Builds

Arguably one of the more powerful features of Jenkins is its ability to dispatch build jobs across a large number of machines. It is quite easy to set up a farm of build servers, either to share the load across multiple machines, or to run build jobs in different environments. This is a very effective strategy which can potentially increase the capacity of your CI infrastructure dramatically.

Distributed builds are generally used either to absorb extra load, for example absorbing spikes in build activity by dynamically adding extra machines as required, or to run specialized build jobs in specific operating systems or environments. For example, you may need to run particular build jobs on a particular machine or operating system. For example, if you need to run web tests using Internet Explorer, you will need to be use a Windows machine. Or one of your build jobs may be particularly resource-heavy, and need to be run on its own dedicated machine so as not to penalize your other build jobs.

Demand for build servers can also fluctuate over time. If you are working with product release cycles, you may need to run a much higher number of builds jobs towards the end of the cycle, for example, when more comprehensive functional and regression test suites may be more frequent.

Jenkins Distributed Build Architecture:

Jenkins uses a master/slave architecture to manage distributed builds. Your main Jenkins server (the one we have been using up until present) is the master. In a nutshell, the master's job is to handle scheduling build jobs, dispatching builds to the slaves for the actual execution, monitor the slaves (possibly taking them online and offline as required) and recording and presenting

the build results. Even in a distributed architecture, a master instance of Jenkins can also execute build jobs directly.

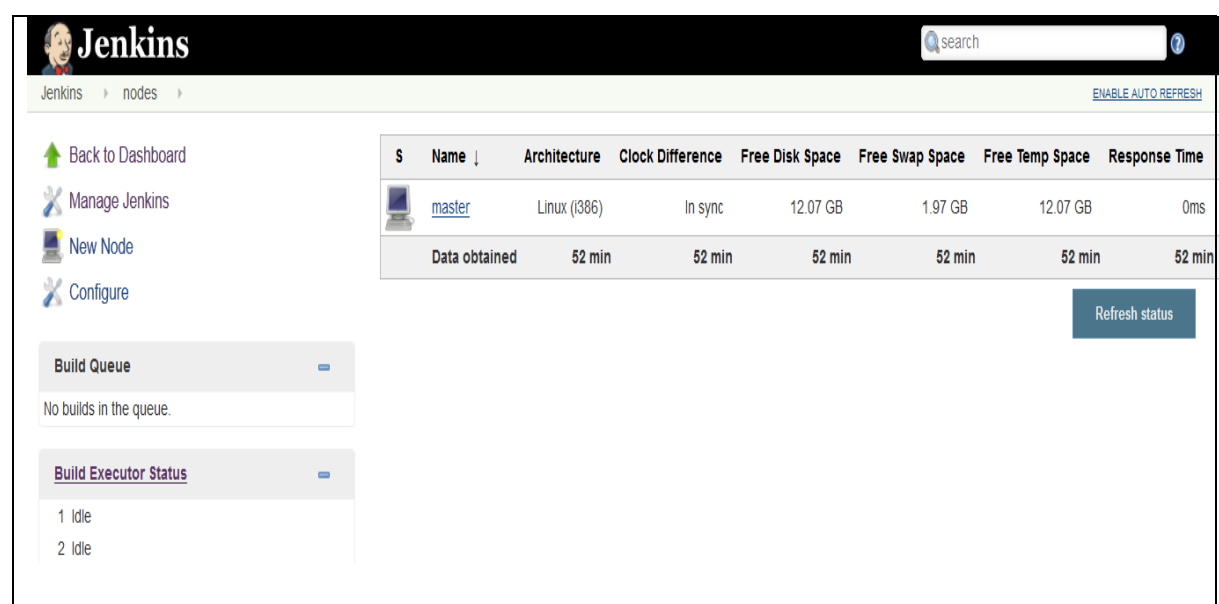
The job of the slaves is to do as they are told, which involves executing build jobs dispatched by the master. You can configure a project to always run on a particular slave machine, or a particular type of slave machine, or simply let Jenkins pick the next available slave.

A slave is a small Java executable that runs on a remote machine and listens for requests from the Jenkins master instance. Slaves can (and usually do) run on a variety of operating systems. The slave instance can be started in a number of different ways, depending on the operating system and network architecture. Once the slave instance is running, it communicates with the master instance over a TCP/ IP connection. We will look at different setups in the next steps.


The Master/Slave Strategies in Jenkins:

There are a number of different ways that you can configure set up a distributed build farm using Jenkins, depending on your operating systems and network architecture. In all cases, the fact that a build job is being run on a slave, and how that slave is managed, is transparent for the end-user: the build results and artifacts will always end up on the master server.

Creating a new Jenkins slave node is a straightforward process. First, go to the Manage Jenkins screen and click on Manage Nodes. This screen displays the list of slave agents (also known as “Nodes” in more politically correct terms), “Managing slave nodes”. From here, you can set up new nodes by clicking on the New Node button. You can also configure some of the parameters related to your distributed build setup.

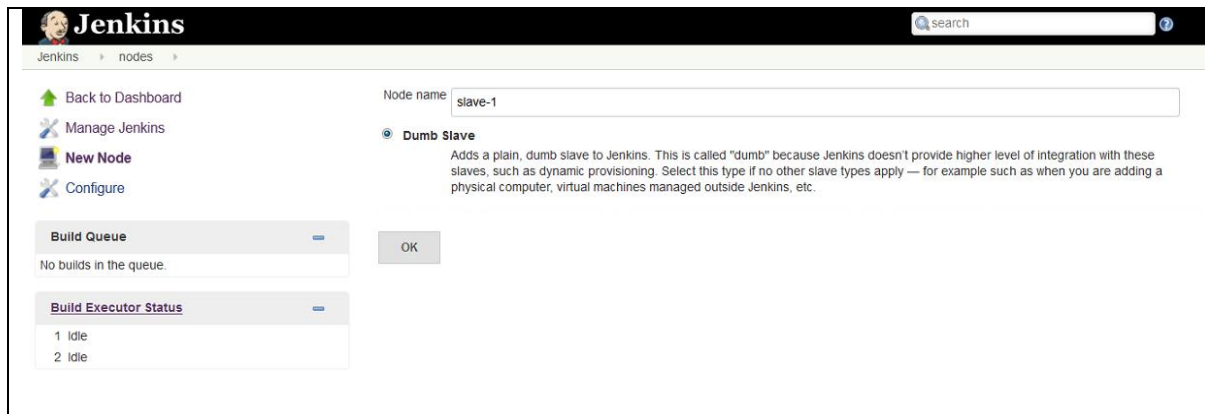


The screenshot displays the Jenkins interface for managing slave nodes. The top navigation bar includes the Jenkins logo, a search bar, and a link to 'ENABLE AUTO REFRESH'. The main content area is titled 'nodes' and contains a table of nodes. The table has columns for 'S', 'Name', 'Architecture', 'Clock Difference', 'Free Disk Space', 'Free Swap Space', 'Free Temp Space', and 'Response Time'. The first row shows the 'master' node with a Linux (i386) architecture, in sync clock difference, and 12.07 GB of free disk, swap, and temp space. The second row shows a 'Data obtained' node with a 52 min clock difference and 52 min of free disk, swap, and temp space. The left sidebar contains links for 'Back to Dashboard', 'Manage Jenkins', 'New Node', and 'Configure'. The bottom section shows the 'Build Queue' (empty) and 'Build Executor Status' (2 idle executors).

S	Name	Architecture	Clock Difference	Free Disk Space	Free Swap Space	Free Temp Space	Response Time
	master	Linux (i386)	In sync	12.07 GB	1.97 GB	12.07 GB	0ms
	Data obtained		52 min	52 min	52 min	52 min	52 min

Creating Node:

To create a Unix-based slave, click on the New Node button as we mentioned above. This will prompt you to enter the name of your slave, At the time of writing, only “dumb slaves” are supported out of the box; “dumb” slaves are passive beasts, that simply respond to build job requests from the master node. This is the most common way to set up a distributed build architecture, and the only option available in a default installation.



In this screen, you simply need to provide a name for your slave. When you click on OK, Jenkins will let you provide more specific details about your slave machine.

Creating a UNIX slave node

The name is simply a unique way of identifying your slave machine. It can be anything, but it may help if the name reminds you of the physical machine it is running on. It also helps if the name is file-system and URL-friendly. It will work with spaces, but you will make life easier for yourself if you avoid them. So “Slave-1” is better than “Slave 1”.

The description is also purely for human consumption, and can be used to indicate why you would use this slave rather than another. Like on the main Jenkins configuration screen, the number of executors lets you define how many concurrent build job this node can execute.

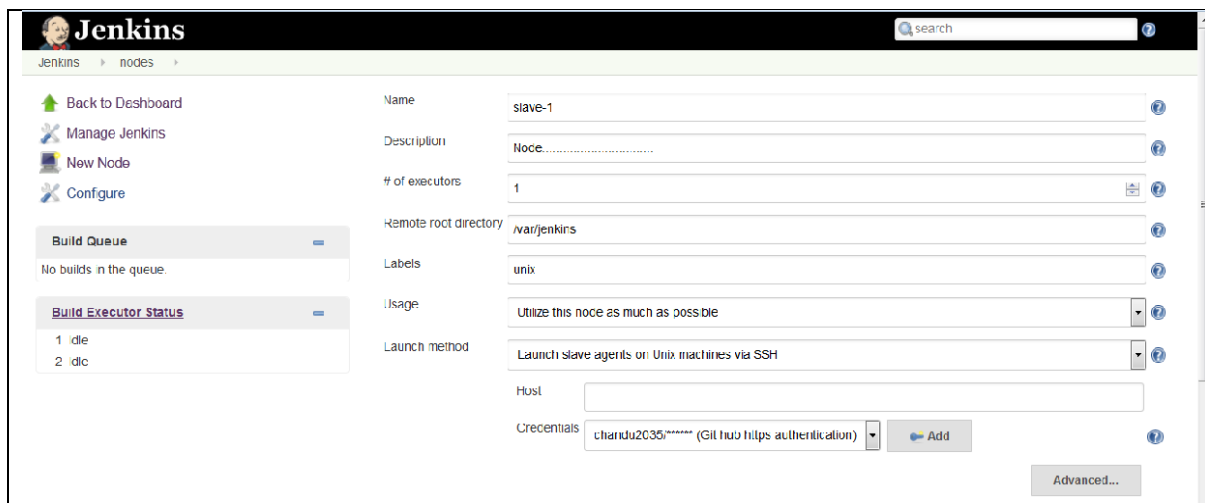
Every Jenkins slave node also needs a place that it can call home, or, more precisely, a dedicated directory on the slave machine that the slave agent can use to run build jobs. You define this directory in the Remote FS root field. You need to provide a local, OS-specific path, such as /var/jenkins for a Unix machine, or C:\jenkins on Windows. Nothing mission-critical is stored in this directory— everything important is transferred back to the master machine once the build is done. So you usually don’t need to be as concerned with backing up these directories as you should be with the master.

Labels are a particularly useful concept when your distributed build architecture begins to grow in size. You can define labels, or tags, to each build node, and then configure a build job to run only on a slave node with a particular label. Labels might relate to operating systems

(unix, windows, macosx, etc.), environments (staging, UAT, development, etc.) or any criteria that you find useful. For example, you could configure your automated WebDriver/Selenium tests to run using Internet Explorer, but only on slave nodes with the “windows” label.

The Usage field lets you configure how intensively Jenkins will use this slave. You have the choice of three options: use it as much as possible, reserve it for dedicated build jobs, or bring it online as required. The first option, “Utilize this slave as much as possible”, tells Jenkins to use this slave freely as soon as it becomes available, for any build job that it can run. This is by far the most commonly used one, and is generally what you want.

There are times, however, when this second option comes in handy. In the project configuration, you can tie a build job to a specific node—this is useful when a particular task, such as automated deployment or a performance test suite, needs to be executed on a specific machine. In this case, the “Leave this machine for tied jobs only” option makes good sense. You can take this further by setting the maximum number of Executors to 1. In this case, not only will this slave be reserved for a particular type of job, but it will only ever be able to run one of these build jobs at any one time. This is a very useful configuration for performance and load tests, where you need to reserve the machine so that it can execute its tests without interference.



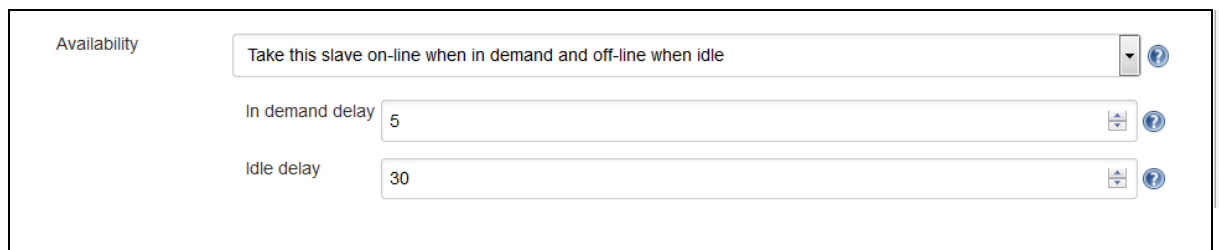
The screenshot shows the Jenkins web interface for configuring a slave node. The top navigation bar includes 'Jenkins' and a search bar. The left sidebar has links for 'Back to Dashboard', 'Manage Jenkins', 'New Node', and 'Configure'. The main content area is titled 'Configure' and shows the configuration for a node named 'slave-1'. The configuration fields include: Name (slave-1), Description (Node...), # of executors (1), Remote root directory (/var/jenkins), Labels (unix), Usage (Utilize this node as much as possible), and Launch method (Launch slave agents on Unix machines via SSH). There are also fields for Host and Credentials (chandu2035*****). An 'Add' button is next to the credentials field, and an 'Advanced...' button is at the bottom right. On the left, there is a 'Build Queue' section showing 'No builds in the queue' and a 'Build Executor Status' section showing two executors: '1 die' and '2 dic'.

The launch method is where you decide how Jenkins will start the node, as we mentioned earlier. For the configuration we are discussing here, you would choose “Launch slave agents on UNIX machines via SSH”. The Advanced button lets you enter the additional details that Jenkins needs to connect to the UNIX slave machine: a host name, a login and password, and a port number. You can also provide a path to the SSH private key file on the master machine (e.g., id_dsa or id_rsa) to use for “passwordless” Public/Private Key authentication.

You can also configure when Jenkins starts and stops the slave. By default, Jenkins will simply keep the slave running and use it whenever required (the “Keep this slave on-line as

much as possible” option). If Jenkins notices that the slave has gone offline (for example due to a server reboot), it will attempt to restart it if it can. Alternatively, Jenkins can be more conservative with your system resources, and take the slave offline when it doesn’t need it. To do this, simply choose the “Take this slave on-line when in demand and off-line when idle” option. This is useful if you have regular spikes and lulls of build activity, as an unused slave can be taken offline to conserve system resources for other tasks, and brought back online when required.

Jenkins also needs to know where it can find the build tools it needs for your build jobs on the slave machines. This includes JDKs as well as build tools such as Maven, Ant, and Gradle. If you have configured your build tools to be automatically installed, you will usually have no extra configuration to do for your slave machines; Jenkins will download and install the tools as required. On the other hand, if your build tools are installed locally on the slave machine, you will need to tell Jenkins where it can find them. You do this by ticking the Tool Locations checkbox, and providing the local paths for each of the tools you will need for your build jobs.

The image shows a screenshot of the 'Availability' section in the Jenkins slave configuration interface. It features a dropdown menu with the selected option 'Take this slave on-line when in demand and off-line when idle'. Below this, there are two input fields: 'In demand delay' with the value '5' and 'Idle delay' with the value '30'. Each input field has a small icon to its right, likely for incrementing or decrementing the value. There are also help icons (question marks) next to each field and the dropdown menu.

“Take this slave on-line when in demand and off-line when idle”

The third option is “Take this slave on-line when in demand and off-line when idle” (“Taking a slave off-line when idle”). As the name indicates, this option tells Jenkins to bring this slave online when demand is high, and to take it offline when demand subsides. This lets you keep some build slaves in reserve for periods of heavy use, without having to maintain a slave agent running on them permanently. When you choose this option, you also need to provide some extra details. The “In demand delay” indicates how many minutes jobs must have been waiting in the queue before this slave will be brought online. The Idle delay indicates how long the slave needs to be idle before Jenkins will take it off-line.

Node Properties:

You can also specify environment variables. These will be passed into your build jobs, and can be a good way to allow your build jobs to behave differently depending on where they are being executed.

Once you have done this, your new slave node will appear in the list of computers on the Jenkins Nodes.

Node Properties

☐ Environment variables
 ☒ Tool Locations

List of tool locations

Name

(JDK) Java 1.8

Home

/opt/apps/java

Delete

Name

(Ant) Ant

Home

/opt/apps/ant

Delete

Name

(Maven) Maven 3.2.3

Home

/opt/apps/maven

Delete

Add

Launch Node:

Once click on save button you will get the option to launch the slave machine. Now you can see the console output on the screen.

Jenkins

nodes

slave-1

Back to List

Status

Delete Slave

Configure

Build History

Load Statistics

Log

Build Executor Status

Slave slave-1 (your description about this node.....)

Mark this node temporarily offline

Disconnected by anonymous

Launch slave agent

Created by anonymous user

Labels

[unix](#)

Projects tied to slave-1

None

Jenkins

nodes

slave-1

Back to List

Status

Delete Slave

Configure

Build History

Load Statistics

Log

Build Executor Status

```

[05/11/15 23:59:00] [SSH] Opening SSH connection to :22.
[05/11/15 23:59:00] [SSH] Authentication successful.
[05/11/15 23:59:01] [SSH] The remote users environment is:
BASIC=/bin/bash
BASHOPTS=cmdhist:extquote:force_ignores:hostcomplete:interactive_comments:progcomp:promptvars:sourcepath
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_EXECUTION_STRING=ssh
BASH_LINENO=()
BASH_SOURCE=()
BASH_VERSION=([0]--"4" [1]--"1" [2]--"2" [3]--"1" [4]--"release" [5]--"1386-redhat-linux-gnu")
BASH_VERSION=4.1.2(1)-release
DIRSTACK=()
EUID=0
CIRCUTS=()
G_FROXPX_FTPNAMEPS=1

```

```
[05/11/15 23:59:01] [SSH] Checking java version of /opt/apps/java/bin/java
[05/11/15 23:59:01] [SSH] /opt/apps/java/bin/java -version returned 1.8.0_25.
[05/11/15 23:59:01] [SSH] Starting sftp client.
[05/11/15 23:59:01] [SSH] Copying latest slave.jar...
[05/11/15 23:59:01] [SSH] Copied 478,472 bytes.
Expanded the channel window size to 4MB
[05/11/15 23:59:01] [SSH] Starting slave process: cd "/var/jenkins" && /opt/apps/java/bin/java -jar slave.jar
<==[JENKINS REMOTING CAPACITY]==>channel started
Slave.jar version: 2.51
This is a Unix slave
Evacuated stdout
Slave successfully connected and online
```

Page generated: May 11, 2015 11:59:00 PM [REST API](#) [Jenkins ver. 1.611](#)

Once all configuration done properly you will notified in the console output as “Slave successfully connected and online”. It means your slave is ready to execute jobs.

Go to your job configuration and check the option “Restrict where this project can be run” and then provide the slave label name and save the job configuration. Now can you able run your jobs in slave machine by click on the “Build Now”

☐ This build is parameterized

☐ Disable Build (No new builds will be executed until the project is re-enabled.)

☐ Execute concurrent builds if necessary

☒ Restrict where this project can be run

Label Expression

Slaves in [label](#): 1

Advanced Project Options

Jenkins

- New Item
- People
- Build History
- Project Relationship
- Check File Fingerprint
- Manage Jenkins
- Credentials
- Disk usage

Build Queue
No builds in the queue.

Build Executor Status

- master**
 - 1 Idle
 - 2 Idle
- slave-1**
 - 1 Idle

ENABLE AUTO REFRESH

[add description](#)

S	W	Name ↓	Last Success	Last Failure	Last Duration
		Demo Job	1 day 3 hr - #1	N/A	9.9 sec
		Hello World	50 sec - #17	5 min 26 sec - #15	4.3 sec
		Maven Demo App	6 min 13 sec - #3	9 min 58 sec - #2	20 sec

Icon: [S](#) [M](#) [L](#)

[Legend](#) [RSS for all](#) [RSS for failures](#) [RSS for just latest builds](#)

[Help us localize this page](#)

Page generated: May 12, 2015 12:23:45 AM [REST API](#) [Jenkins ver. 1.611](#)

In the above figure you can find the slaves that are available in your Jenkins server and number of executors.

Turn – off your Slave Node:

Go to Manage Jenkins → Manage Nodes → Click on Slave which you want to turn – off.

Now below screen will appear in this section and you will have to use below mentioned options to turn – off the slave.

- Disconnect
- Mark this node temporarily offline.

Jenkins

nodes > slave-1

Slave slave-1 (your description about this node.....)

Created by anonymous user

Labels: unix

Projects tied to slave-1

S	W	Name ↓	Last Success	Last Failure	Last Duration
		Hello World	6 min 52 sec #17	11 min #15	4.3 sec

Icon: [S](#) [M](#) [L](#)

Legend: [RSS for all](#) [RSS for failures](#) [RSS for just latest builds](#)

Give some description about the reason why you are turning - off that slave and click on the button.

Jenkins

nodes > slave-1

Taking slave-1 Offline

You can optionally explain why you are taking this node offline, so that others can see why:

Not in using.

Mark this node temporarily offline

Build Queue

No builds in the queue.

Build Executor Status

S	Name ↓	Architecture	Clock Difference	Free Disk Space	Free Swap Space	Free Temp Space	Response Time
	master	Linux (i386)	In sync	12.06 GB	1.97 GB	12.06 GB	0ms
	slave-1	Linux (i386)	In sync	12.06 GB	1.97 GB	12.06 GB	3371ms
	Data obtained	10 min	10 min	10 min	10 min	10 min	10 min

Refresh status

master

- 1 die
- 2 die

slave-1 (offline)

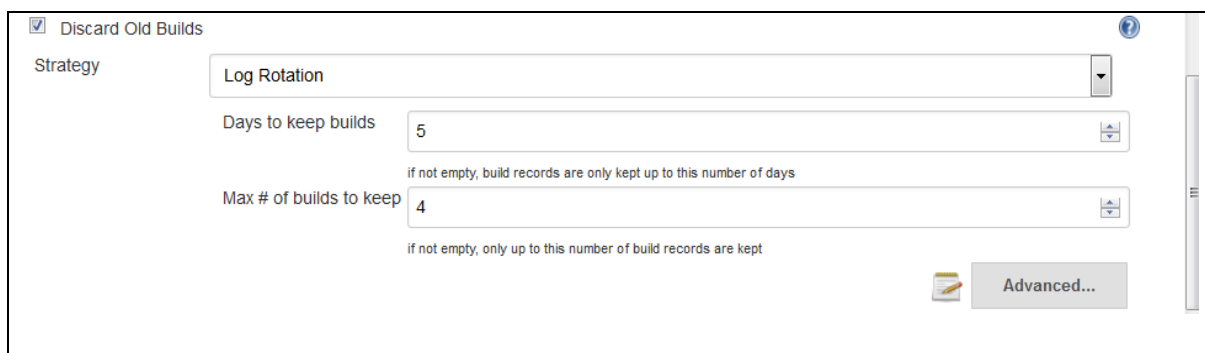
Maintaining Jenkins

We will be discussing a few tips and tricks that you might find useful when maintaining a large Jenkins instance. We will look at things like how to limit, and keep tabs on, disk usage, how to give Jenkins enough memory and how to archive build jobs or migrate them from one server to another. Some of these topics are discussed elsewhere in the book, but here we will be looking at things from the point of view of the system administrator.

Monitoring Disk Space:

Build History takes disk space. In addition, Jenkins analyzes the build records when it loads a project configuration, so a build job with a thousand archived builds is going to take a lot longer to load than one with only fifty. If you have a large Jenkins server with tens or hundreds of build jobs, multiply this accordingly.

Probably the simplest way to keep a cap on disk usage is to limit the number of builds a project maintains in its history. You can configure this by ticking the Discard Old Builds checkbox at the top of the project configuration page. If you tell Jenkins to only keep the last 20 builds, it will start discarding (and deleting) older build jobs once it reaches this number. You can limit them by number (i.e., no more than 20 builds) or by date (i.e., builds no older than 30 days). It does this intelligently, though: if there has ever been a successful build, Jenkins will always keep at least the latest successful build as part of its build history, so you will never lose your last successful build.



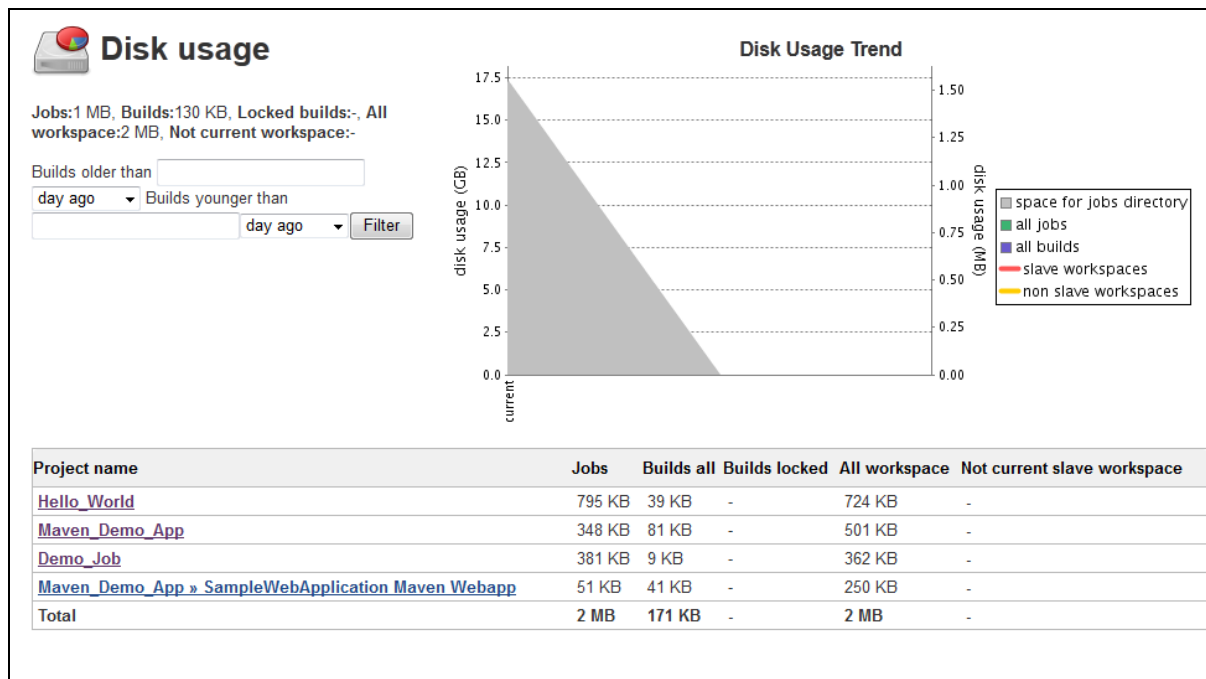
The screenshot shows the 'Discard Old Builds' configuration section in Jenkins. At the top, the checkbox 'Discard Old Builds' is checked. Below it, the 'Strategy' dropdown menu is set to 'Log Rotation'. Under 'Days to keep builds', the value '5' is entered. A small text note below this says 'if not empty, build records are only kept up to this number of days'. Under 'Max # of builds to keep', the value '4' is entered. Another small text note below this says 'if not empty, only up to this number of build records are kept'. At the bottom right of the configuration area, there is a small icon of a notepad and a button labeled 'Advanced...'.

Using the Disk Usage Plugin:

One of the most useful tools in the Jenkins administrator's tool box is the Disk Usage plugin. This plugin records and reports on the amount of disk space used by your projects. It lets you isolate and fix projects that are using too much disk space.

You can install the Disk Usage plugin in the usual way, from the Plugin Manager screen. Once you have installed the plugin and restarted Jenkins, the Disk Usage plugin will record the

amount of disk space used by each project. It will also add a Disk Usage link on the Manage Jenkins screen, which you can use to display the overall disk usage for your projects.



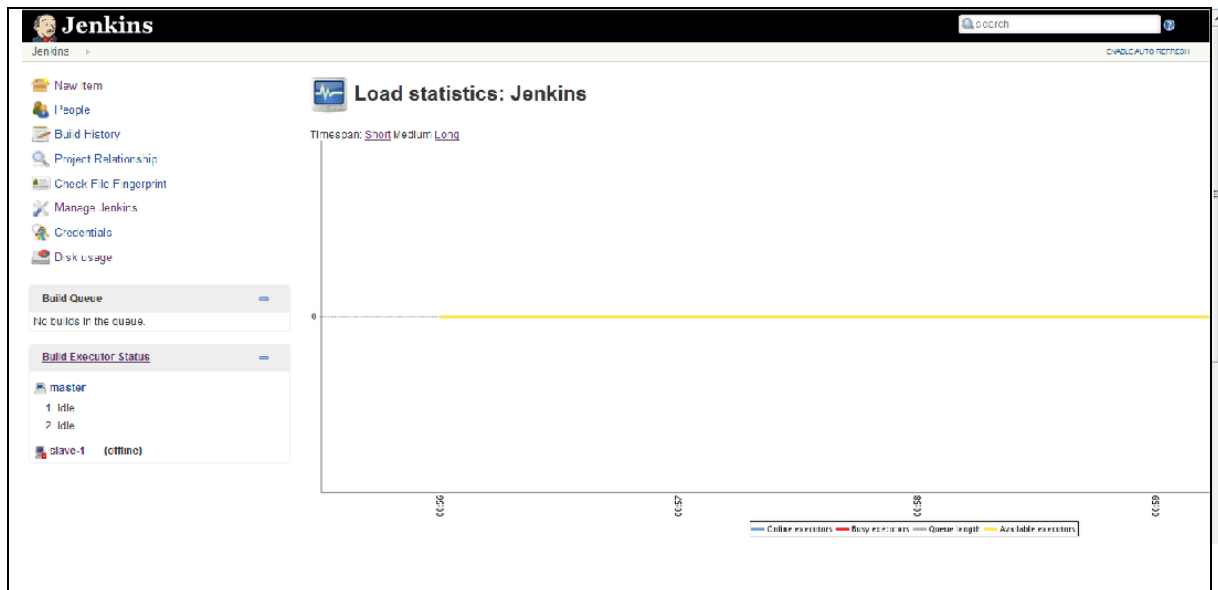
Monitoring the Server Load:

Jenkins provides build-in monitoring of server activity. On the Manage Jenkins screen, click on the Load Statistics icon. This will display a graph of the server load over time for the master node (“Jenkins Load Statistics”). This graph keeps track of three metrics: the total number of executors, the number of busy executors, and queue length.

The **total number of executors** (the blue line) includes the executors on the master and on the slave nodes. This can vary when slaves are brought on and offline, and can be a useful indicator of how well your dynamic provisioning of slave nodes is working.

The **number of busy executors** (the red line) indicates how many of your executors occupied executing builds are. You should make sure you have enough spare capacity here to absorb spikes in build jobs. If all of your executors are permanently occupied running build jobs, you should add more executors and/or slave nodes.

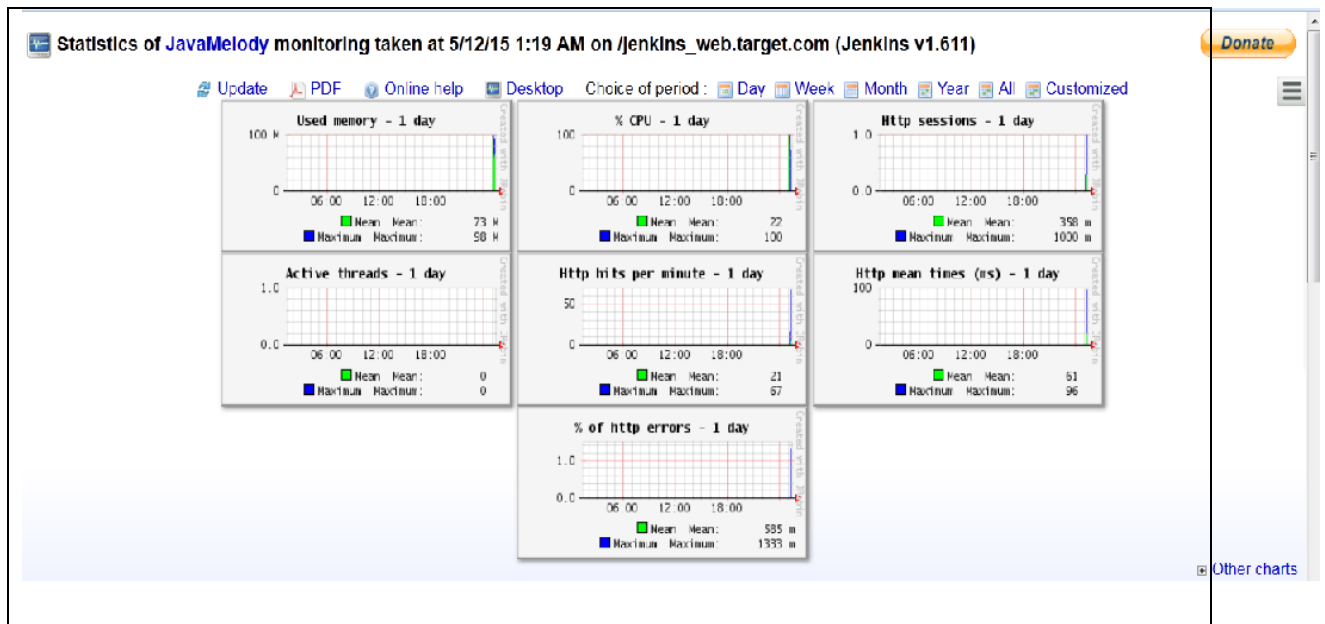
The **queue length** (the gray line) is the number of build jobs awaiting executing. Build jobs are queued when all of the executors are occupied. This metric does not include jobs that are waiting for an upstream build job to finish, so it gives a reasonable idea of when your server could benefit from extra capacity.



You can get a similar graph for slave nodes, using the Load Statistics icon in the slave node details page.

Another option is to install the Monitoring plugin. This plugin uses JavaMelody to produce comprehensive HTML reports about the state of your build server, including CPU and system load, average response time, and memory usage (“The Jenkins Monitoring plugin”). Once you have installed this plugin, you can access the JavaMelody graphs from the Manage Jenkins screen, using the “Monitoring of Jenkins/Jenkins master” or “Jenkins/Jenkins nodes” menu entries.

The image shows the Jenkins 'Monitoring' page for the 'master' node. The breadcrumb trail at the top is 'Jenkins > nodes > master > Monitoring'. The left sidebar contains links: 'Back to List', 'Status', 'Configure', 'Build History', 'Load Statistics', 'Script Console', and 'Monitoring' (which is highlighted). Below the sidebar is the 'Build Executor Status' section, showing '1 idle' and '2 idle' executors. The main content area is titled 'JavaMelody Monitoring' and includes a subtitle: 'This page provides access to the [JavaMelody](#) monitoring of the [master](#) node.' Under the 'System reports' section, there are four links with icons: 'View Threads' (gear icon), 'View OS processes' (gear icon), 'View memory usage histogram. JDK is required' (memory icon), and 'Display data collected by MBeans' (beans icon). Under the 'System actions' section, there are two links with icons: 'Runs the garbage collector on the node' (garbage can icon) and 'Warning! This operation may affect performance of this node' (warning icon).



Backing Up your Configuration:

Backing up your data is a universally recommended practice, and your Jenkins server should be no exception. Fortunately, backing up Jenkins is relatively easy. In this section, we will look at a few ways to do this.

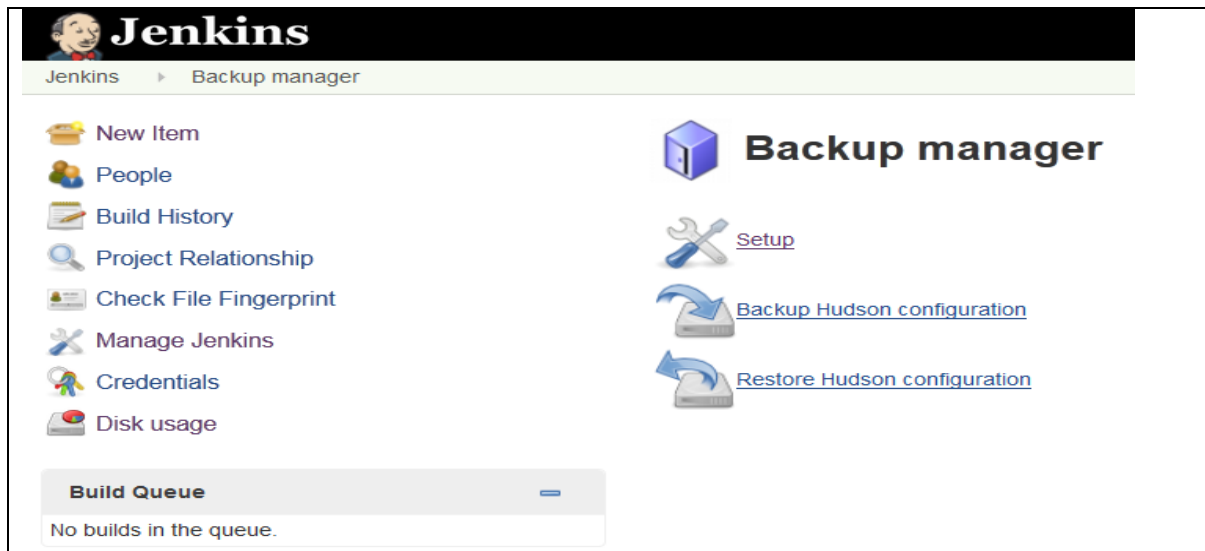
In the simplest of configurations, all you need to do is to periodically back up your JENKINS_HOME directory. This contains all of your build jobs configurations, your slave node configurations, and your build history. This will also work fine while Jenkins is running—there is no need to shut down your server while doing your backup.

```
$ export JENKINS_HOME=/tmp/jenkins-backup
```

Using the Backup Plugin:

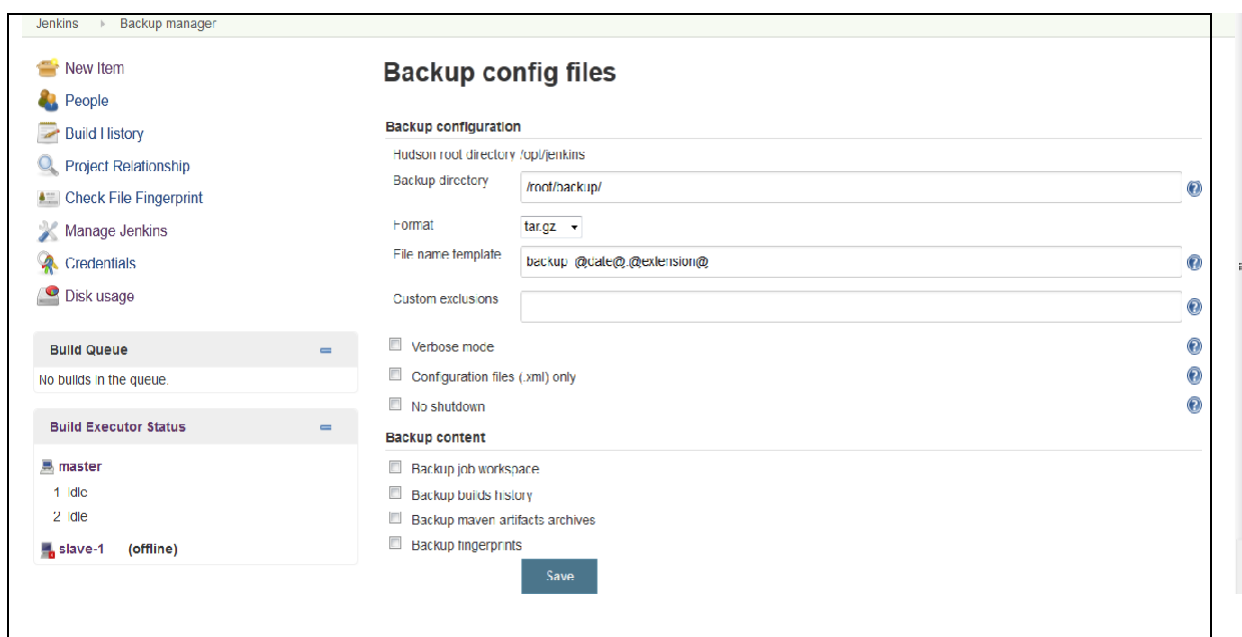
The approach described in the previous section is simple enough to integrate into your normal backup procedures, but you may prefer something more Jenkins-specific. The Backup plugin (“The Jenkins Backup Manager Plugin”) provides a simple user interface that you can use to back up and restore your Jenkins configurations and data.

This plugin lets you configure and run backups of both your build job configurations and your build history. The Setup screen gives you a large degree of control over exactly what you want backed up (Configuring the Jenkins Backup Manager”). You can opt to only back up the XML configuration files, or back up both the configuration files and the build history. You can also choose to backup (or not to backup) the automatically-generated Maven artifacts (in many build processes, these will be available on your local Enterprise Repository Manager). You can also back up the job workspaces (typically unnecessary, as we discussed above) and any generated fingerprints.



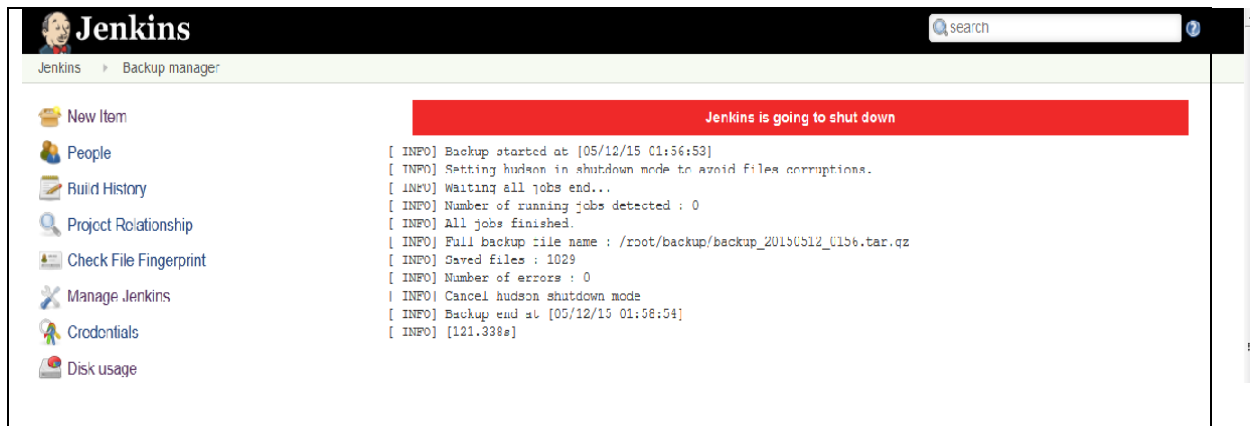
Backup config files:

Click on setup link, it will take you to the “Backup Manger” configuration area. Here you can configure Jenkins to take the backup.



Backup Hudson Configuration:

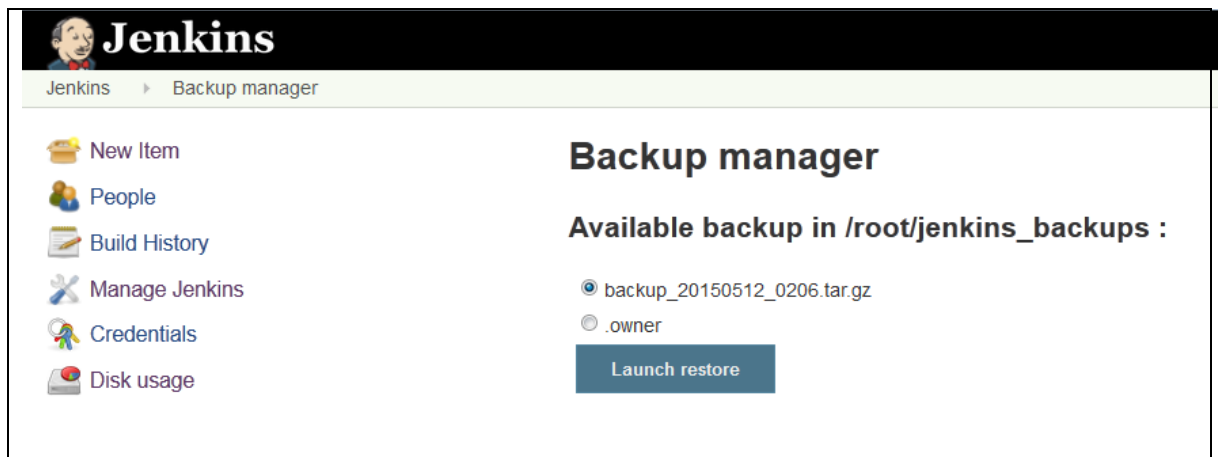
This option will perform the automated backup, compress the file as tar.gz and store it into your user home directory.



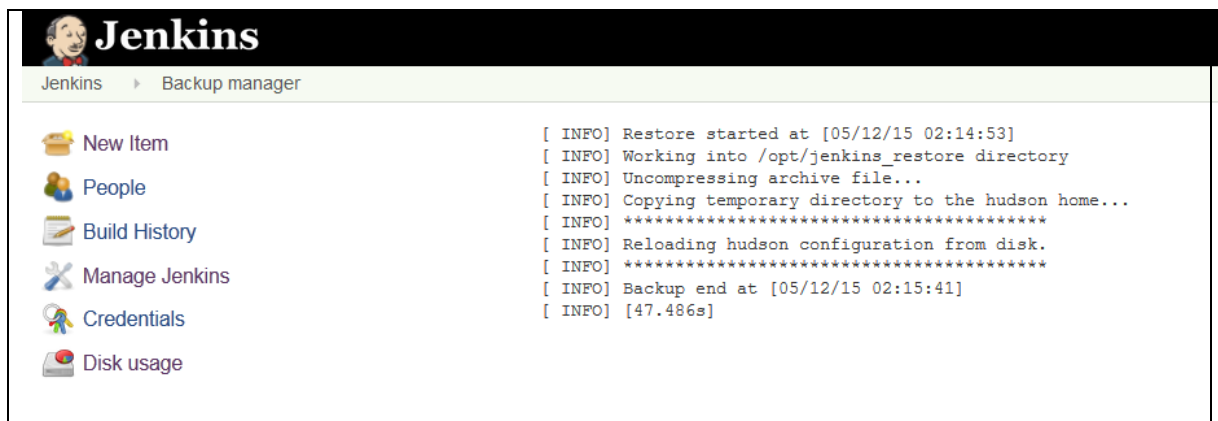
The screenshot shows the Jenkins Backup manager page. A red banner at the top states "Jenkins is going to shut down". Below this, a log shows the backup process: "Backup started at [05/12/15 01:56:53]", "Setting hudson in shutdown mode to avoid files corruptions.", "Waiting all jobs end...", "Number of running jobs detected : 0", "All jobs finished.", "Full backup file name : /root/backup/backup_20150512_0156.tar.gz", "Saved files : 1029", "Number of errors : 0", "Cancel hudson shutdown mode", "Backup end at [05/12/15 01:58:54]", and "[121.338s]". The left sidebar contains links: New Item, People, Build History, Project Relationship, Check File Fingerprint, Manage Jenkins, Credentials, and Disk usage.

Restore Configuration:

To restore a previous configuration, just go to the Restore page and choose the date of the configuration you wish to reinstate ("Restoring a previous configuration"). Once the configuration has been restored to the previous state, you need to reload the Jenkins configuration from disk or restart Jenkins.



The screenshot shows the Jenkins Backup manager page. The title is "Backup manager". Below it, the text says "Available backup in /root/jenkins_backups :". There are two radio buttons: "backup_20150512_0206.tar.gz" (selected) and ".owner". A "Launch restore" button is at the bottom right. The left sidebar contains links: New Item, People, Build History, Manage Jenkins, Credentials, and Disk usage.



The screenshot shows the Jenkins Backup manager page. A log shows the restore process: "Restore started at [05/12/15 02:14:53]", "Working into /opt/jenkins_restore directory", "Uncompressing archive file...", "Copying temporary directory to the hudson home...", "Reloading hudson configuration from disk.", "Backup end at [05/12/15 02:15:41]", and "[47.486s]". The left sidebar contains links: New Item, People, Build History, Manage Jenkins, Credentials, and Disk usage.

Frequently asked questions:

- **What is Continuous Integration**

Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.

- **What are the advantages of Jenkins over the traditional and automated build systems**

Open source, and good support also, and good community is there and lot of help from the Internet.

- **Where does Jenkins plugins resides**

In Jenkins workspace there will be directory called plugins, in this directory all plugins will be installed.

- **By default where get Jenkins installed**

In tomcat actually, inside webapps directory. In JENKINS_HOME.

- **What are the default Jenkins plugins**

Ant, credentials, CVS, javadoc, JUnit, LDAP, Mailer, Matrix Authorization strategy, Matrix project, Maven integration, OWASP markup formatter, PAM Authentication, SSH Credentials, SSH Slave, Subversion, Windows Slaves, External Monitor Job Type.

- **What is “ivy plugin” in Jenkins**

By using an Ivy project instead of a Freestyle project Jenkins can simplify the job configuration. It will auto-detect all the ivy modules you checked out, create builds for them, and automatically generate the upstream/downstream build triggers based on your Ivy dependencies.

- **What happened if build fails**

Automatically you will get the mail, if you configured the mail notifications in the post build actions.

- **What are global properties (Jenkins)**

The Global Properties section lets you define variables that can be managed centrally but used in all of your build jobs. You can add as many properties as you want here, and use them in your build jobs. Jenkins will make them available within your build job environment, so you can freely use them within your Ant and Maven build scripts. Note that you shouldn't put periods (".") in the property names, as they won't be processed correctly. So ldapserver or ldap_server is fine, but not ldap.server.

- **What is mean by 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

- ✓ Build after other builds are built.
- ✓ Build periodically (**cron job syntax**)
- ✓ Trigger builds remotely
- ✓ Poll the SCM for changes (***** for every check-in**)

- **What is parameterized build**

If you want pass some arguments during the run time you need to configure it.

Example: When you are doing a deployment which environment you want to deploy. Because same Jenkins job used to deploy QA, UAT, Performance. There is drop down menu, by selecting that you can call Jenkins shell script and then work it accordingly.

- **Build fails if one build having another build dependency, so what was the case Job A is running, job B is dependent on job A, What happens if first build fails.** Job B is dependent on job A, So first it will execute the job A and then B. otherwise it will fail.

- **What is distributed build**

There is Master/Slave concept is there, If java based applications there in no issue, but when comes to C and C++ like applications it will differ the OS to OS, in this scenario we will use the Distributed build.

Actually in which flavor of OS it should be execute it, because we configured the slaves in multiple flavors of OS.

- **What is mean by build promotion**

You need to deploy it to QA environment, if it is certified QA then you need to deploy it to higher environment.

- **What is the build pipeline**

This gives the ability to form a chain of jobs based on their upstream\downstream dependencies. Downstream jobs may, as per the default behaviors, be triggered automatically, or by a suitable authorized user manually triggering it.

- **Jenkins runs at what user**

It depends on.... JENIKINS_USER=

Or

Tomcat will be the Jenkins user.

- **What is post build actions**

After your job is done, like Email notifications, Tag this build, check-in artifact to svn, or executing some post build script.

- **Why Jenkins went to wait state**

Jenkins runs out of disk space, Go to Manage Jenkins → Manage nodes – there you can find it.

- **How do you resolve wait state in Jenkins**

By restarting the Jenkins or using Manage nodes options.

- **Do you know executors in Jenkins**

No of build executers will be used to define the how many jobs should be run parallely at a time.

Example: Let's say there are 3 jobs available, but executors are 2 only, then 3rd job will be pending in the queue list.

- **How do you configure email system in Jenkins**

Configure Jenkins → Manage Jenkins: There will be a options to set email notifications, here you need to specify SMTP server, port number and all.

- **How can you monitor Jenkins (Checking the status)**

Manage Jenkins → Load Statistics: Here you will get the number of executors, number of busy executors and queue length.

- **Do you know Poll SCM**

Poll SCM is like, Jenkins would always check your version control system, if there are any changes in that particular version control or not, if there are any changes it will trigger build automatically in Jenkins. **(Continuously checking the svn for changes for triggering)**

- **What value will you setup for Poll SCM**

If you want for every 5 minutes check whether any check-ins happened in svn then `*/5****`.

- **In real time we can develop the applications using different technologies like Java, .Net, PHP, etc.. Is same configuration/job creation for any technology**
Yes, we can do it by integrating the plugins.

- **How to debug if any build fails**

In Jenkins, we need to check the console output, by checking the console output logs the troubleshooting steps will vary.

- **Is installation of Jenkins means deploying war is enough for any OS like windows, linux, or Linux**

Deploying Jenkins.war file is enough or we can run that war file as a service.