Jenkins Continuous Integration Cookbook Second Edition

Jenkins is an award-winning and one of the most popular Continuous Integration servers in the market today. It was designed to maintain, secure, communicate, test, build, and improve the software development process.

This book starts by examining the most common maintenance tasks. This is followed by steps that enable you to enhance the overall security of Jenkins. You will then explore the relationship between Jenkins builds and Maven pom.xml. Then, you will learn how to use plugins to display code metrics and fail builds to improve quality, followed by how to run performance and functional tests against a web application and web services. Finally, you will see what the available plugins are, concluding with best practices to improve quality.

What this book will do for you...

- Integrate Jenkins with LDAP and SSO solutions
- Maintain and secure Jenkins
- Run an integration server firing automatic functional and performance tests
- Communicate through social media and by plotting custom data
- Skin Jenkins to your corporate look and feel
- Refine the use of code metrics to improve quality
- Write your first custom Jenkins plugin
- Apply tweaks to optimize your use of Jenkins

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions for solving the problem efficiently
- Clear explanations of what you did
- Apply the solution to other situations

Over 90 recipes to produce great results from Jenkins using pro-level practices, techniques, and solutions


Alan Mark Berg

7.50 x 9.25
835 mm x 191 mm

Content Type: Black & White
Paper Type: White
Page Count: 488
File Type: InDesign
Request ID: CSS1330767
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 "Maintaining Jenkins"
- A synopsis of the book’s content
- More information on *Jenkins Continuous Integration Cookbook* Second Edition

**About the Author**

**Alan Mark Berg**, BSc, MSc, PGCE, has been the lead developer at Central Computer Services at the University of Amsterdam since 1998. He is currently working in an Innovation Work Group that accelerates the creation of new and exciting services. In his famously scarce spare time, he writes. Alan has a bachelor's degree, two master's degrees, a teaching qualification, and quality assurance certifications. He has also coauthored two Packt Publishing books about Sakai (http://sakaiproject.org), a highly successful open source learning management platform used by millions of students around the world. He has won a couple of awards, including the Sakai Fellowship and Teaching With Sakai Innovation Award (TWSIA).

Alan enjoys working with talent; this forces him to improve his own competencies. This motivation is why Alan enjoys working in energetic, open source communities of interest. At the time of writing, he is on the board of directors of the Apereo Foundation and is the community officer for its Learning Analytics Initiative (https://confluence.sakaiproject.org/display/LAI/Learning+Analytics+Initiative).

In previous incarnations, Alan was a QA director, a technical writer, an Internet/Linux course writer, a product line development officer, and a teacher. He likes to get his hands dirty with building, gluing systems, exploring data, and turning it into actionable information. He remains agile by ruining various development and acceptance environments and generally rampaging through the green fields of technological opportunity.
Acknowledgments

I would like to warmly thank my gentle wife, Hester. Without your unwritten understanding that 2 a.m. is a normal time to work, I would not have finished this or any other large-scale project. I would also like to thank my strong-willed and stubborn teenage sons, Nelson and Lawrence, for no particular reason, possibly for all those very interesting, vaguely dangerous moments.

I would also like to thank the Packt Publishing team, whose consistent behind-the-scenes effort greatly improved the quality of this book.

Finally, I would also like to thank Hester van der Heijden for the book's cover image. The picture was shot in Texel, a Dutch Waddensee island. This book's cover represents Jenkins' ability to work with many types of systems. The entangled branches are Jenkins' plugins reaching out to the complexity of large organizations. In the distance, you can see the sunny landscape of production, practiced to perfection through Continuous Integration.
Jenkins Continuous Integration Cookbook Second Edition

Jenkins is a Java-based Continuous Integration (CI) server that supports the discovery of defects early in the software cycle. Thanks to a rapidly growing number of plugins (currently over 1,000), Jenkins communicates with many types of systems, building and triggering a wide variety of tests.

CI involves making small changes to software and then building and applying quality assurance processes. Defects do not only occur in the code, but also appear in naming conventions, documentation, how the software is designed, build scripts, the process of deploying the software to servers, and so on. CI forces the defects to emerge early, rather than waiting for software to be fully produced. If defects are caught in the later stages of the software development life cycle, the process will be more expensive. The cost of repair radically increases as soon as the bugs escape to production. Estimates suggest it is 100 to 1,000 times cheaper to capture defects early. Effective use of a CI server, such as Jenkins, could be the difference between enjoying a holiday and working unplanned hours to heroically save the day. And as you can imagine, in my day job as a senior developer with aspirations for quality assurance, I like long boring days, at least for mission-critical production environments.

Jenkins can automate the building of software regularly and trigger tests pulling in the results and failing based on defined criteria. Failing early via build failure lowers the costs, increases confidence in the software produced, and has the potential to morph subjective processes into an aggressive metrics-based process that the development team feels is unbiased.

Jenkins is not just a CI server, it is also a vibrant and highly active community. Enlightened self-interest dictates participation. There are a number of ways to do this:

- Participate in the mailing lists and Twitter (https://wiki.jenkins-ci.org/display/JENKINS/Mailing+Lists). First, read the postings and as you get to understand what is needed, then participate in the discussions. Consistently reading the lists will generate many opportunities to collaborate.
- Test Jenkins and especially the plugins and write bug reports, donating your test plans.
- Improve documentation by writing tutorials and case studies.
What This Book Covers

Chapter 1, Maintaining Jenkins, describes common maintenance tasks such as backing up and monitoring. The recipes in this chapter outline methods for proper maintenance that in turn lowers the risk of failures.

Chapter 2, Enhancing Security, details how to secure Jenkins and the value of enabling single sign-on (SSO). This chapter covers many details, ranging from setting up basic security for Jenkins, deploying enterprise infrastructure such as a directory service, and a SSO solution to automatically test for the OWASP top 10 security.

Chapter 3, Building Software, reviews the relationship between Jenkins and Maven builds and a small amount of scripting with Groovy and Ant. The recipes include checking for license violations, controlling the creation of reports, running Groovy scripts, and plotting alternative metrics.

Chapter 4, Communicating Through Jenkins, reviews effective communication strategies for different target audiences from developers and project managers to the wider public. Jenkins is a talented communicator, with its hordes of plugins notifying you by e-mail, dashboards, and Google services. It shouts at you through mobile devices, radiates information as you walk pass big screens, and fires at you with USB sponge missile launchers.

Chapter 5, Using Metrics to Improve Quality, explores the use of source code metrics. To save money and improve quality, you need to remove defects in the software life cycle as early as possible. Jenkins test automation creates a safety net of measurements. The recipes in this chapter will help you build this safety net.

Chapter 6, Testing Remotely, details approaches to set up and run remote stress and functional tests. By the end of this chapter, you will have run performance and functional tests against a web application and web services. Two typical setup recipes are included. The first is the deployment of a WAR file through Jenkins to an application server. The second is the creation of multiple slave nodes, ready to move the hard work of testing away from the master node.

Chapter 7, Exploring Plugins, has two purposes. The first is to show a number of interesting plugins. The second is to review how plugins work.

Appendix, Processes that Improve Quality, discusses how the recipes in this book support quality processes and points to other relevant resources. This will help you form a coherent picture of how the recipes can support your quality processes.
Maintaining Jenkins

In this chapter, we will cover the following recipes:

- Using a test Jenkins instance
- Backing up and restoring
- Modifying the Jenkins configuration from the command line
- Installing Nginx
- Configuring Nginx as a reverse proxy
- Reporting overall storage use
- Deliberately failing builds through log parsing
- Adding a job to warn of storage use violations through log parsing
- Keeping in contact with Jenkins through Firefox
- Monitoring via JavaMelody
- Keeping track of script glue
- Scripting the Jenkins command-line interface
- Global modifications of jobs with Groovy
- Signaling the need to archive
Introduction

Jenkins is feature-rich and is vastly extendable through plugins. Jenkins talks to numerous external systems and its jobs work with many diverse technologies. Maintaining Jenkins in a rich environment running 24 x 7 with uneven scheduling is a challenge. You have to pay attention to the detail. It is easy to add new jobs and you are unlikely to remove old projects quickly. Load increases, passwords expire, storage fills. Further, Jenkins and its plugins improve rapidly. There is a new minor version of Jenkins released weekly, mostly with improvements, occasionally with bugs. For a stable system in a complex environment, you need to monitor, clean up storage, back up, keep control of your Jenkins scripts, and consistently clean and polish. This chapter has recipes for the most common tasks. Proper maintenance lowers the risk of failures such as:

- **New plugins causing exceptions**: There are a lot of good plugins being written with rapid version changes. In this situation, it is easy for you to accidentally add new versions of plugins with new defects. There have been a number of occasions during upgrades when suddenly the plugin does not work. To combat the risk of plugin exceptions, consider using a test Jenkins instance before releasing to a critical system.

- **Storage over-flowing with artifacts**: If you keep a build history that includes artifacts such as war files, large sets of JAR files, or other types of binaries and source code, then your storage space is consumed at a surprising rate. Storage costs have decreased tremendously, but storage usage equates to longer backup times and more communication from slave to master. To minimize the risk of disk overflowing, you will need to consider your backup and restore policy and the associated build retention policy expressed in the advanced options of jobs.

- **Script spaghetti**: As jobs are written by various development teams, the location and style of the included scripts vary. This makes it difficult for you to keep track. Consider using well-defined locations for your scripts and a scripts repository managed through a plugin.

- **Resource depletion**: As memory is consumed or the number of intense jobs increases, then Jenkins slows down. Proper monitoring and quick reactions reduce impact.

- **A general lack of consistency between jobs due to organic growth**: Jenkins is easy to install and use. The ability to seamlessly turn on plugins is addictive. The pace of adoption of Jenkins within an organization can be breathtaking. Without a consistent policy, your teams will introduce lots of plugins and also lots of ways of performing the same work. Conventions improve consistency and readability of jobs and thus decrease maintenance.
The recipes in this chapter are designed to address the risks mentioned. They represent only one set of approaches. If you have comments or improvements, feel free to contact me at bergsmooth@gmail.com or better still add tutorials to the Jenkins community wiki.

The Jenkins community is working hard on your behalf. There are weekly minor releases of Jenkins and many of the plugins are incrementally improved occasionally because of the velocity of change, bugs are introduced. If you see an issue, please report it back.

Signing up to the community

To add community bug reports or modify wiki pages, you will need to create an account at https://wiki.jenkins-ci.org/display/JENKINS/Issue+Tracking.

Using a test Jenkins instance

Continuous Integration (CI) servers are critical in the creation of deterministic release cycles. Any long-term instability in the CI will reflect in a slowing down of the rate at which milestones are reached in your project plans. Incremental upgrading is addictive and mostly straightforward, but should be seen in the light of a Jenkins critical role—a software project's life cycle.

Before releasing plugins to your Jenkins production server, it is worth aggressively deploying to a test Jenkins instance and then sitting back and letting the system run jobs. This gives you enough time to react to any minor defects found.

There are many ways to set up a test instance. One is to use a virtual image of Ubuntu and share the workspace with the host server (the server that the virtual machine runs on). There are a number of advantages to this approach:

- **Saving state**: At any moment, you can save the state of the running virtual image and return to that running state later. This is excellent for short-term experiments that have a high risk of failure.

- **Ability to share images**: You can run your virtual image anywhere that a player can run. This may include your home desktop or a hardcore server.

- **Use a number of different operating systems**: This is good for node machines running integration tests or functional tests with multiple browser types.

- **Swap workspaces**: By having the workspace outside the virtual image on the host of the virtual server, you can test different version levels of OSes against one workspace. You can also test one version of Jenkins against different host workspaces with different plugin combinations.
The long-term support release

The community manages core stability via the use of a long-term support release of Jenkins, which is mature and less feature rich when compared to the latest version. However, it is considered the most stable platform to upgrade (http://mirrors.jenkins-ci.org/war-stable/latest/jenkins.war).

The test instance is normally of lower specification than the acceptance and production systems. By starving a test instance, you can expose certain types of issues such as memory leaks early. As you move your configuration to production, you want to scale up capacity, which might involve moving from virtual to hardware.

This recipe details the use of VirtualBox (http://www.virtualbox.org/), an open source virtual image player with a Ubuntu image (http://www.ubuntu.com/). The virtual image will mount a directory on the host server. You will then point Jenkins to the mounted directory. When the guest OS is restarted, then Jenkins will automatically run against the shared directory.

Throughout this book, recipes will be cited using Ubuntu as the example OS.

Getting ready

You will need to download and install VirtualBox. You can find detailed instructions to download a recent version of VirtualBox at https://www.virtualbox.org/manual/UserManual.html. At the time of writing this book, Ubuntu 11.04 was the latest version available from the VirtualBox image SourceForge site. Unpack an Ubuntu 11.04 virtual image from http://sourceforge.net/projects/virtualboximage/files/Ubuntu%20Linux/11.04/ubuntu_11.04-x86.7z/download.

If you run into problems, then the manual is a good starting point; in particular, refer to Chapter 12, Troubleshooting, at http://www.virtualbox.org/manual/ch12.html.

Note that newer images will be available at the time of reading. Feel free to try the most modern version; it is probable that the recipe still works with this.

You will find an up-to-date series of Ubuntu virtual images at http://virtualboxes.org/images/ubuntu-server/.

Security Considerations

If you consider using other’s OS images, which is a bad security practice, then you should create a Ubuntu image from a boot CD as mentioned at https://wiki.ubuntu.com/Testing/VirtualBox.
1. Run VirtualBox and click on the **New** icon in the top left-hand corner. You will now see a wizard for installing virtual images.

2. Set **Name** to **Jenkins_Ubuntu_11.04**. The OS type will be automatically updated. Click on the **Next** button.

3. Set **Memory** to **2048 MB** and then click on **Next**.
   
   Note that the host machine requires 1 GB more RAM than the total allocated to its guest images. In this example, your host machine requires 3 GB of RAM. For more details, visit [http://www.oracle.com/us/technologies/virtualization/oraclevm/oracle-vm-virtualbox-ds-1655169.pdf](http://www.oracle.com/us/technologies/virtualization/oraclevm/oracle-vm-virtualbox-ds-1655169.pdf).

4. Select **Use existing hard disk**. Browse and select the unpacked VDI image by clicking on the folder icon:

5. Press the **Create** button.

6. Start the virtual image by clicking on the **Start** icon:

7. Log in to the guest OS with username and password as **Ubuntu reverse**.

8. Change the password of user Ubuntu from a terminal as follows:
   ```
   sudo passwd
   ```


10. Update the OS with regard to security patches (this may take some time depending on bandwidth):
    ```
    sudo apt-get update
    sudo apt-get upgrade
    ```

11. Install the kernel's **dkms** module:
    ```
    sudo apt-get install dkms
    ```

    Note that the **dkms** module supports installing other kernel modules such as the modules needed by VirtualBox. For more details, visit [https://help.ubuntu.com/community/DKMS](https://help.ubuntu.com/community/DKMS).
12. Install Jenkins:
   
   ```bash
   sudo apt-get install jenkins
   ```

13. Install the kernel modules for VirtualBox:
   
   ```bash
   sudo /etc/init.d/vboxadd setup
   ```

14. Install guest additions using the **Devices** menu option in the VirtualBox window:
   
   ![Insert Guest Additions CD image... Host4D]

15. Add the `jenkins` user to the `vboxsf` group, as follows:
   
   ```bash
   sudo gedit /etc/group
   vboxsf:x:1001:Jenkins
   ```

16. Modify the `JENKINS_HOME` variable in `/etc/default/jenkins` to point to the mounted shared directory:
   
   ```bash
   sudo gedit /etc/default/jenkins
   JENKINS_HOME=/media/sf_workspacej
   ```

17. On the host OS, create the directory named `workspacej`.

18. Within VirtualBox, right-click on the Ubuntu image and select **Settings**.

19. Update the **Folder Path** field to point to the directory that you previously created. In the following screenshot, you can see that the folder was created under my `home` directory:
20. Restart VirtualBox and then start the Ubuntu guest OS.


**How it works...**

First, you installed a virtual image of Ubuntu, changed the password so that it is harder for others to log in, and updated the guest OS for security patches.

The Jenkins repository was added to the list of known repositories in the guest OS. This involved installing a repository key locally. The key is used to verify that the packages automatically downloaded belong to a repository that you have agreed to trust. Once the trust is enabled, you can install the most current version of Jenkins via standard package management and later update it aggressively.

You need to install some additional code called guest additions so that VirtualBox can share folders from the host. Guest additions depend on Dynamic Kernel Module Support (DKMS). DKMS allows bits of code to be dynamically added to the kernel. When you ran the /etc/init.d/vboxadd setup command, VirtualBox added guest addition modules through DKMS.

[Warning: If you forget to add the DKMS module, then sharing folders will fail without any errors being shown.]

The default Jenkins instance now needs a little reconfiguration:

- The jenkins user needs to belong to the vboxsf group to have permission to use the shared folder
- The /etc/init.d/jenkins startup script points to /etc/default/jenkins and thereby picks up the values of specific properties such as JENKINS_HOME

Next you added a shared folder to the guest OS from the VirtualBox GUI, and finally you restarted VirtualBox and the guest OS to guarantee that the system was in a fully configured and correctly initialized state.

There are a number of options for configuring VirtualBox with networking. You can find a good introduction at http://www.virtualbox.org/manual/ch06.html.

**See also**

- The Monitoring via JavaMelody recipe
Maintaining Jenkins

Back up and restoring

A core task for the smooth running of Jenkins is the scheduled backing up of its home directory (within Ubuntu /var/lib/jenkins), not necessarily all the artifacts, but at least its configuration and the history of testing that plugins need to make reports.

Backups are not interesting unless you can restore. There is a wide range of stories on this subject. My favorite (and I won't name the well-known company involved) is that somewhere in the early 70s, a company brought a very expensive piece of software and a tape backup facility to back up all the marketing results being harvested through their mainframes. However, not everything was automated. Every night a tape needed to be moved into a specific slot. A poorly paid worker was allocated the task. For a year, the worker would professionally fulfill the task. One day a failure occurred and a backup was required. The backup failed to restore. The reason was that the worker also needed to press the record button every night, but this was not part of the tasks assigned to him. There was a failure to regularly test the restore process. The process failed, not the poorly paid person. Hence, learning the lessons of history, this recipe describes both backup and restore.

Currently, there is more than one plugin for backups. I have chosen the thinBackup plugin (https://wiki.jenkins-ci.org/display/JENKINS/thinBackup) as it allows scheduling.

The rapid evolution of plugins and the validity of recipes

Plugins improve aggressively and you may need to update them weekly. However, it is unlikely that the core configuration changes, but quite likely that extra options will be added, increasing the variables that you input in the GUI. Therefore, the screenshots shown in this book may be slightly different from the most modern version, but the recipes should remain intact.

Getting ready

Create a directory with read and write permissions for Jenkins and install the thinBackup plugin.

Murphy as a friend

You should assume the worst for all of the recipes in this book: aliens attacking, coffee on motherboard, cat eats cable, cable eats cat, and so on. Make sure that you are using a test Jenkins instance.
How to do it...

1. Click on the ThinBackup link in the Manage Jenkins page:

2. Click on the link to Settings by the Toolset icon.

3. Add the details as shown in the following screenshot where /data/jenkins/backups is a placeholder for the directory you have previously created. Notice the warning about using the H syntax; this will be explained later.

4. Click on Save.
5. Then, click on the **Backup now** icon.

6. From the command line, visit your backup directory. You should now see an extra subdirectory named **FULL-\{timestamp\}** where \{timestamp\} is the time in seconds needed to create a full backup.

7. Click on the **Restore** icon.

8. A drop-down menu named **restore backup from** will be shown with the dates of the backups. Select the backup just created and click on the **Restore** button:

```
How it works...

The backup scheduler uses cron notation ([http://en.wikipedia.org/wiki/Cron](http://en.wikipedia.org/wiki/Cron)).

1 0 * * 7 means every seventh day of the week at 00:01 A.M. 1 1 * * * implies that differential backups occur once per day at 1.01 A.M. Every seventh day, the previous differentials are deleted.

Remember the warning when configuring? Replacing the time symbols with \H allows Jenkins to choose when to run the thinBackup plugin. \H \* * * will trigger a job at a random time in the day, which spreads the load.

Waiting until Jenkins/Hudson is idle to perform a backup is a safety method and helps Jenkins to spread the load. It is recommended that this option is enabled; otherwise there is a risk of corruption of the backups due to builds locking files.

Force Jenkins to quiet mode after specified minutes ensures that no jobs are running while backing up. This option forces quiet mode after waiting for Jenkins to be quiet for a specific amount of time. This avoids problems with backups waiting on Jenkins naturally reaching a quiet moment.
```
Differential backups contain only files that have been modified since the last full backup. The plugin looks at the last modified date to work out which files need to be backed up. The process can sometimes go wrong if another process changes the last modified date without actually changing the content of the files.

61 is the number of directories created with backups. As we are cleaning up the differentials via the Clean up differential backups option, we will get to around 54 full backups, roughly a year of archives before cleaning up the oldest.

Backup build results were selected as we assume that we are doing the cleaning within the job. There will not be much extra added to the full archive. However, in case of misconfiguration, you should monitor the archive for storage usage.

Cleaning up differential backups saves you doing the clean-up work by hand. Moving old backups to ZIP files saves space, but might temporarily slow down your Jenkins server.

For safety’s sake, regularly copy the archives off your system.

The backup options called Backup build archive, Backup ‘userContent’ folder, and Backup next build number file, increase the amount of content and system state backed up.

Restoring is a question of returning to the restore menu and choosing the date. Extra options include restoring the build number file and plugins (downloaded from an external server to decrease backup size).

I cannot repeat this enough; you should practice a restore occasionally to avoid embarrassment.

Full backups are the safest as they restore to a known state. Therefore, don’t generate too many differential backups between full backups.

There’s more...

Here are a couple more points for you to think about.

Checking for permission errors

If there are permission issues, the plugin fails silently. To discover these types of issues, you will need to check the Jenkins logfile, /var/log/jenkins/jenkins.log for *NIX distributions, for the log level SEVERE:

SEVERE: Cannot perform a backup. Please be sure jenkins/hudson has write privileges in the configured backup path {0}.
Testing exclude patterns

The following Perl script will allow you to test the exclude pattern. Simply replace the $content value with your Jenkins workspace location and $exclude_pattern with the pattern you wish to test. The following script will print a list of the excluded files:

```perl
#!/usr/bin/perl
use File::Find;
my $content = "/var/lib/jenkins";
my $exclude_pattern = "^.*\.(war)|(class)|(jar)$";
find( \&excluded_file_summary, $content );
sub excluded_file_summary {
    if ((-f $File::Find::name)&& ( $File::Find::name =~/$exclude_pattern/)) {
        print "$File::Find::name\n";
    }
}
```

You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

You can find the documentation for the standard Perl module File::Find at http://perldoc.perl.org/File/Find.html.

For every file and directory under the location mentioned in $content, the find( \&excluded_file_summary, $content ); line calls the excluded_file_summary function.

The exclude pattern "^.*\.(war)|(class)|(jar)$" ignores all WAR, class, and JAR files.

EPIC Perl

If you are a Java developer who occasionally writes Perl scripts, then consider using the EPIC plugin for Eclipse (http://www.epic-ide.org/).

See also

- The Reporting overall storage use recipe
- The Adding a job to warn of storage use violations through log parsing recipe
Modifying the Jenkins configuration from the command line

You may well be wondering about the XML files at the top level of the Jenkins workspace. These are configuration files. The `config.xml` file is the main one that deals with the default server values, but there are also specific ones for any plugins that have values set through the GUI.

There is also a `jobs` subdirectory underneath the workspace. Each individual job configuration is contained in a subdirectory with the same name as the job. The job-specific configuration is then stored in `config.xml` within the subdirectory. It's a similar situation for the `users` directory: one subdirectory per user with the personal information stored in `config.xml`.

Under a controlled situation where all the Jenkins servers in your infrastructure have the same plugins and version levels, it is possible for you to test on one test machine and then push the configuration files to all the other machines. You can then restart the Jenkins servers with the command-line interface (CLI) or scripts under `/etc/init.d`, as follows:

```
sudo /etc/init.d/jenkins restart
```

This recipe familiarizes you with the main XML configuration structure and then provides hints about the plugin API based on the details of the XML.

**Getting ready**

You will need a Jenkins server with security enabled and the ability to edit files either by logging in and working from the command line or through editing with a text editor.

**How to do it...**

1. In the top level directory of Jenkins, look for the `config.xml` file. Edit the line with `numExecutors`, changing the number 2 to 3:

   ```xml
   <numExecutors>3</numExecutors>
   ```

2. Restart the server. You will see that the number of executors has increased from the default two to three:

   ![Build Queue](image)

   No builds in the queue.

   ![Build Executor Status](image)

<table>
<thead>
<tr>
<th>#</th>
<th>Master</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Idle</td>
</tr>
<tr>
<td>2</td>
<td>Idle</td>
</tr>
<tr>
<td>3</td>
<td>Idle</td>
</tr>
</tbody>
</table>
3. Plugins persist their configuration through XML files. To prove this point, look for the thinBackup.xml file. You will not find it unless you have installed the thinBackup plugin.

4. Look at the Backing up and restoring recipe again. You will now find the following XML file:

```xml
<?xml version='1.0' encoding='UTF-8'?>
<org.jvnet.hudson.plugins.thinbackup.ThinBackupPluginImpl
 plugin="thinBackup@1.7.4">
  <fullBackupSchedule>1 0 * * 7</fullBackupSchedule>
  <diffBackupSchedule>1 1 * * *</diffBackupSchedule>
  <backupPath>/data/jenkins/backups</backupPath>
  <nrMaxStoredFull>61</nrMaxStoredFull>
  <excludedFilesRegex></excludedFilesRegex>
  <waitForIdle>false</waitForIdle>
  <forceQuietModeTimeout>120</forceQuietModeTimeout>
  <cleanupDiff>true</cleanupDiff>
  <moveOldBackupsToZipFile>true</moveOldBackupsToZipFile>
  <backupBuildResults>true</backupBuildResults>
  <backupBuildArchive>true</backupBuildArchive>
  <backupUserContents>true</backupUserContents>
  <backupNextBuildNumber>true</backupNextBuildNumber>
  <backupBuildsToKeepOnly>true</backupBuildsToKeepOnly>
</org.jvnet.hudson.plugins.thinbackup.ThinBackupPluginImpl>
```

**How it works...**

Jenkins uses XStream (http://xstream.codehaus.org/) to persist its configuration into a readable XML format. The XML files in the workspace are configuration files for plugins, tasks, and an assortment of other persisted information. The config.xml file is the main configuration file. Security settings and global configuration are set here and reflect changes made through the GUI. Plugins use the same structure and the XML values correspond to member values in the underlying plugin classes. The GUI itself is created from XML via the Jelly framework (http://commons.apache.org/jelly/).

By restarting the server, you are certain that any configuration changes are picked up during the initialization phase.

It is also possible to use **Reload configuration** from a storage feature from the Manage Jenkins page, to load an updated configuration without restarting.

**There's more...**

Here are a few things for you to consider.
Turning off security
When you are testing new security features, it is easy to lock yourself out of Jenkins. You will not be able to log in again. To get around this problem, modify useSecurity to false by editing config.xml and restart Jenkins; the security features are now turned off.

Finding JavaDoc for custom plugin extensions
The following line of code is the first line of the thin plugin configuration file named thinBackup.xml, mentioning the class from which the information is persisted. The class name is a great Google search term. Plugins can extend the functionality of Jenkins and there may well be useful methods exposed for administrative Groovy scripts:

```xml
<org.jvnet.hudson.plugins.thinbackup.ThinBackupPluginImpl>
```

The effects of adding garbage
Jenkins is great at recognizing invalid configurations as long as they are recognizable as a valid XML fragment. For example, add the following line of code to config.xml:

```xml
<garbage>yeuchblllllllllaaaaaa</garbage>
```

When you reload the configuration, you will see this at the top of the Manage Jenkins screen:

You have data stored in an older format and/or unreadable data.

Pressing the Manage button will return you to a detailed page of debug information, including the opportunity to reconcile the data:

You can see from this that Jenkins is developer-friendly when reading corrupted configuration that it does not understand.

See also
- The Using a test Jenkins instance recipe
Installing Nginx

This recipe describes the steps required to install a basic Nginx installation.

Nginx (pronounce as engine-x) is a free, open source, high-performance HTTP server and reverse proxy, as well as an IMAP/POP3 proxy server. Igor Sysoev started development of Nginx in 2002, with the first public release in 2004. Nginx is known for its high performance, stability, rich feature set, simple configuration, and low resource consumption.

Placing an Nginx server in front of your Jenkins server has a number of advantages:

- **Easy configuration**: The syntax is straightforward. Configuring the basic details of a new server requires only a few lines of easily readable text.
- **Speed and resource consumption**: Nginx has a reputation for running faster than its competitors and with fewer resources.
- **URL rewriting**: Powerful configuration options allow you to straightforwardly manage the URL namespace for multiple servers sitting behind Nginx.
- **Offsetting SSL**: Nginx can take on the responsibility for secure connections, diminishing the number of certificates needed in an organization and decreasing the CPU load of the Jenkins server.
- **Caching**: Nginx can cache much of the content from Jenkins, decreasing the number of requests that the Jenkins server has to return.
- **Monitoring**: When Nginx sits in front of many Jenkins servers, its central log files can act as a clear point of monitoring.

**Getting ready**


**How to do it...**

1. From a terminal, type:
   
   ```bash
   sudo apt-get install nginx
   ```

2. Browse to the localhost location. You will now see the Nginx welcome page:
3. From a terminal, type `sudo /etc/init.d/nginx` and you'll get the following output:

```
Usage: nginx {start|stop|restart|reload|force-reload|status|config test|rotate|upgrade}
```

Note that not only can you stop and start the server, you can also check the status and run configuration tests.

4. Check the status of the server by typing the `sudo /etc/init.d/nginx status` command:

```
* nginx is running
```

5. Edit the welcome page within gedit:

```
```

6. After the `<body>` tag, add `<h1>Welcome to nginx working with Jenkins</h1>`.

7. Save the file.

8. Browse to the localhost location. You will see a modified welcome page:

```
Welcome to nginx working with Jenkins
If you see this page, the nginx web server is successfully installed and working. Further configuration is required.
For online documentation and support please refer to nginx.org. Commercial support is available at nginx.com.
Thank you for using nginx.
```

9. Review the `/etc/nginx/nginx.conf` configuration file, especially the following lines:

```
include /etc/nginx/conf.d/*.*.conf;
include /etc/nginx/sites-enabled/*/;
access_log /var/log/nginx/access.log;
error_log /var/log/nginx/error.log;
```
10. Edit and save `/etc/nginx/sites-available/default`. For the two listen stanzas, change the number 80 to 8000:

```
listen 8000 default_server;
listen [::]:8000 default_server ipv6only=on;
```

If port 8000 is already in use by another server, then feel free to change to another port number.

11. Test the configuration by running the following command from the terminal:

```
sudo /etc/init.d/nginx configtest
* Testing nginx configuration   [ OK ]
```

12. From the terminal, restart the server:

```
sudo /etc/init.d/nginx restart
* Restarting nginx nginx
```

13. Browse to the localhost location. You will see a warning that you are unable to connect:

![Unable to connect](image)

14. Browse to `localhost:8000` and you will see the welcome page.

### How it works...

You installed Nginx with default settings using the `apt` command. The `/etc/init.d/nginx` command controls the server. You edited the welcome page, which resides at `/usr/share/nginx/html/index.html`, and restarted Nginx.

The master configuration file is `/etc/nginx/nginx.conf`. The `include /etc/nginx/conf.d/*.conf; line collects configuration settings from any file in the `/etc/nginx/conf.d` directory that has the `.conf` extension. It also collects any configuration file from the `/etc/nginx/sites-enabled` directory through the `include /etc/nginx/sites-enabled/*;` command.
You changed the port number that the Nginx server was listening to through the `listen` directives in the default configuration file named `/etc/nginx/sites-available/default`. To avoid embarrassment, we tested the configuration before deploying the changes. You did this through the terminal with the `/etc/init.d/nginx configtest` command.

**Support information**


A sample chapter on configuration is available online at http://www.packtpub.com/sites/default/files/0868-chapter-3-basic-nginx-configuration_1.pdf.

**There's more...**

Here are a couple more points for you to think about.

**Naming logfiles**

Nginx allows you run multiple virtual hosts on multiple ports. To help you with maintenance of the servers, it is advisable to separate the logfiles. To do this, you will need to change the following lines in `/etc/nginx/nginx.conf`:

```plaintext
access_log /var/log/nginx/access.log;
error_log /var/log/nginx/error.log;
```

Make it easier for others. Consider using consistently naming conventions such as including the hostname and port numbers:

```plaintext
access_log /var/log/nginx/HOST_PORT_access.log;
error_log /var/log/nginx/HOST_PORT_error.log;
```

**Backing up configuration**

I cannot over-emphasize this enough. Backing up changes to configurations is vital to the smooth running of your infrastructure. Personally, I back up all configuration changes to a version control system. I can go through the commit logs and see exactly when I have made mistakes or used a clever tweak. However, revision control is not always feasible because sensitive information such as passwords may be contained. At least automatically back up the configuration locally.

**See also**

- The *Configuring Nginx as a reverse proxy* recipe
Maintaining Jenkins

Configuring Nginx as a reverse proxy

This recipe configures Nginx to act as a reverse proxy for Jenkins. You will modify logfiles and port locations and tweak buffer sizes and the request headers passed through. I will also acquaint you with the best practice of testing configurations before restarting Nginx. This best practice has saved me a number of embarrassing moments.

Getting ready

You need to have followed the Installing Nginx recipe and have a Jenkins instance running on localhost:8080.

How to do it...

1. Create /etc/nginx/proxy.conf with the following lines of code:

```
proxy_redirect off;
proxy_set_header Host $host;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
client_max_body_size 10m;
client_body_buffer_size 128k;
proxy_connect_timeout 90;
proxy_send_timeout 90;
proxy_read_timeout 90;
proxy_buffers 32 4k;
```

2. Create the /etc/nginx/sites-enabled/jenkins_8080 proxypass file with the following lines of code:

```
server {
  listen 80;
  server_name localhost;
  access_log /var/log/nginx/jenkins_8080_proxypass_access.log;
  error_log /var/log/nginx/jenkins_8080_proxypass_access.log;

  location / {
    proxy_pass http://127.0.0.1:7070/;
    include /etc/nginx/proxy.conf;
  }
}
```
3. From the terminal, run `sudo /etc/init.d/nginx configtest`. You will see the following output:
   * Testing nginx configuration [ OK ]

4. In a terminal, restart the server by running the following command:
   `sudo /etc/init.d/nginx restart`

5. Browse to the localhost location. The connection will time out, as shown in the following screenshot:

![Unable to connect](image)

6. Review the access log `/var/log/nginx/jenkins_8080_proxypass_access.log`. You will see a line similar to the following line (note that 499 is the status code):
   ```
   127.0.0.1 - - [25/Jun/2014:17:50:50 +0200] "GET / HTTP/1.1" 499 0 "-" "Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:30.0) Gecko/20100101 Firefox/30.0"
   ```

7. Edit `/etc/nginx/sites-enabled/jenkins_8080_proxypass` by changing 7070 to 8080:
   ```
   location / {
     proxy_pass      http://127.0.0.1:8080/;
     include         /etc/nginx/proxy.conf;
   }
   ```

8. Test the configuration changes:
   ```
   sudo /etc/init.d/nginx configtest
   * Testing nginx configuration [ OK ]
   ```

9. From a terminal, restart the Nginx server by running the following command:
   `sudo /etc/init.d/nginx restart`

10. Browse to the localhost location. You will see the Jenkins main page.
How it works...

It is a tribute to Nginx configuration syntax that you configured Nginx with only a few lines.

By default, Nginx acts on any configuration in the files in the /etc/nginx/sites-enabled/directory. During the recipe, you added a file to this directory; it was then added to the configuration settings of Nginx on the next restart.

The configuration file includes a server block with a port and the server name localhost. You can have multiple servers defined in the configuration listening on different ports and with different server names. However, in our case, we needed only one server:

```
server {
  listen    80;
  server_name  localhost;
}
```

You also defined the location of the log files, as follows:

```
access_log  /var/log/nginx/Jenkins_8080_proxypass_access.log;
error_log   /var/log/nginx/jenkins_8080_proxypass_access.log;
```

Nginx tests the URI specified in the request's header against the parameters of the location directives defined inside the server block. In this case, you had only one location command that points at the top level /:

```
location / {
}
```

There can be more than one location configured. However, in our example, there is only one that passed on all requests to the Jenkins server found at 127.0.0.1:8080:

```
proxy_pass      http://127.0.0.1:8080/;
```

As noted, when proxy_pass is pointing to a non-existent location, then a 499 HTTP status code is returned. This is an Nginx-specific way of flagging the issue.

Note that proxy_pass works with both the HTTP and HTTPS protocols.

We loaded in a second configuration file that deals with the detailed settings of the proxying. This is useful as you can repeat the same settings consistently across a number of server configurations, keeping the details central. This approach aids readability and maintenance.

```
include         /etc/nginx/proxy.conf;
```
Nginx configuration allows you to use embedded variables such as the remote address of the $remote_addr client. The Nginx reference manual details the embedded variables. You can find the manual at http://nginx.com/wp-content/uploads/2014/03/nginx-modules-reference-r3.pdf.

Within proxy.conf, you set headers. You set X-REAL-IP and X-Forwarded-For to the remote address of the requester. You need both headers for the smooth running of backend servers and load balancers:

```
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
```


Other performance-related details you configured included the maximum size of the client body (10 megabytes), time out values (90 seconds), and internal buffer sizes (324 kilobytes):

```
client_max_body_size 10m;
client_body_buffer_size 128k;
proxy_connect_timeout 90;
proxy_send_timeout 90;
proxy_read_timeout 90;
proxy_buffers 32 4k;
```

For more information on Nginx as a reverse proxy server, visit http://nginx.com/resources/admin-guide/reverse-proxy/.

There's more...

Here are a couple more points for you to think about.

Testing complex configuration

Modern computers are cheap and powerful. They are able to support multiple test Jenkins and Nginx servers. There are a number of ways of testing complex configurations. One is to run multiple virtual machines on a virtual network. Another is to use different loopback addresses and/or different ports (127.0.0.1:8080, 127.0.0.2:8080, and so on). Both approaches have the advantage of keeping your network traffic off Ethernet cards and local to your computer.
As mentioned in the preface, you can run Jenkins from the command line with commands similar to:

```
java -jar jenkins.war -httpsport=8443 -httpPort=-1
```

Jenkins will start to run over HTTPS on port 8443. The `-httpPort=-1` turned off the HTTP port.

To choose a separate home directory, you will need first to set the `JENKINS_HOME` environment variable.

You would use the following command to run Jenkins on `127.0.0.2`, port 80:

```
sudo -jar jenkins.war --httpPort=80 --httpListenAddress=127.0.0.2
```

### Offloading SSL

One of the advantages of Nginx is that you can allow it to service SSL requests and then pass them onto multiple Jenkins servers as HTTP requests. You can find the basic configuration for this at [https://wiki.jenkins-ci.org/display/JENKINS/Jenkins+behind+an+nginx+reverse+proxy](https://wiki.jenkins-ci.org/display/JENKINS/Jenkins+behind+an+nginx+reverse+proxy).

First, you need to redirect requests on port 80 to an HTTPS URL. In the following example, the `301` status code is used:

```nginx
server {
    listen 80;
    return 301 https://$host$request_uri;
}
```

This states that the link has been permanently moved. This allows the redirect to be cached. You will then need to set a server up on port 443, the standard port for HTTPS, and load in a certificate for the server and its associated key:

```nginx
server {
    listen 443;
    server_name localhost;

    ssl on;
    ssl_certificate /etc/nginx/ssl/server.crt;
    ssl_certificate_key /etc/nginx/ssl/server.key;
}
```

Finally, you will need to use `location` and `proxy_pass` within the server configured for port 443 to pass on to Jenkins servers running over HTTP:

```nginx
location / {
    proxy_pass http://127.0.0.1:8080;
}
```
Despite its simplicity, there are well-known configuration pitfalls, some of which are mentioned at http://wiki.nginx.org/Pitfalls.

See also

- The Installing Nginx recipe

## Reporting overall storage use

Organizations have their own way of dealing with increasing disk usage. Policy ranges from no policy, depending on ad hoc human interactions, to the most state-of-the-art software with central reporting facilities. Most organizations sit between these two extremes with mostly ad hoc intervention with some automatic reporting for the more crucial systems. With minimal effort, you can make Jenkins report disk usage from the GUI and periodically run Groovy scripts that trigger helpful events.

This recipe highlights the disk usage plugin and uses the recipe as a vehicle to discuss the cost of keeping archives stored within the Jenkins workspace.

The disk usage plugin is strongest in combination with an early warning system that notifies you when soft or hard disk limits are reached. The Adding a job to warn of storage use violations through log parsing recipe details a solution. This recipe show that configuring Jenkins requires little effort. Each step might even seem trivial. The power of Jenkins is that you can build complex responses out of a series of simple steps and scripts.

### Getting ready

You will need to install the disk usage plugin.

### How to do it...

1. Click on the Disk usage link under the Manage Jenkins page:
2. Jenkins displays a page with each project name, builds, and a workspace disk usage summary. Click at the top of the table to sort the workspace by file usage:

<table>
<thead>
<tr>
<th>Project name</th>
<th>Jobs</th>
<th>Builds all</th>
<th>Builds locked</th>
<th>Workspace</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAKAI Trunk</td>
<td>13 MB</td>
<td>13 MB</td>
<td>-</td>
<td>2 GB</td>
</tr>
<tr>
<td>Log.git</td>
<td>65 KB</td>
<td>54 KB</td>
<td>-</td>
<td>25 MB</td>
</tr>
<tr>
<td>TestX</td>
<td>38 KB</td>
<td>18 KB</td>
<td>-</td>
<td>8 KB</td>
</tr>
<tr>
<td>ch7.plugins.job_export</td>
<td>14 KB</td>
<td>6 KB</td>
<td>-</td>
<td>5 KB</td>
</tr>
</tbody>
</table>

**How it works...**

Adding a plugin in Jenkins is simplicity itself. The question is what you are going to do with the information.

It is easy for you to forget a checkbox in a build; perhaps an advanced option is enabled where it should not be. Advanced options can at times be problematic as they are not displayed directly in the GUI, so you will need to hit the **Advanced** button first before reviewing them. On a Friday afternoon, this might be one step too far.

Advanced options include artifact retention choices that you will need to correctly configure to avoid overwhelming disk usage. In the previous example, the workspace for **Sakai Trunk** is **2 GB**. The size is to do with the job having its own local Maven repository as defined by the **Use private Maven repository** advanced option. The option is easy for you to miss. In this case, there is nothing to be done as trunk pulls in snapshot jars that might cause instability for other projects. The advanced options shown in the following screenshot include artifact:
Reviewing the advanced options of a project after looking at the disc usage of the project helps you to find unnecessary private repositories.

**There's more...**

If you are keeping a large set of artifacts, it is an indicator of a failure of purpose of your use of Jenkins. Jenkins is the engine that pushes a product through its life cycle. For example, when a job builds snapshots every day, then you should be pushing the snapshots out to where developers find them most useful. That is not Jenkins but a Maven repository or a repository manager such as Artifactory (http://www.jfrog.com/products.php), Apache Archiva (http://archiva.apache.org/), or Nexus (http://nexus.sonatype.org/). These repository managers have significant advantages over dumping to disk, such as:

- **Speed builds by acting as a cache**: Development teams tend to work on similar or the same code. If you build and use the repository manager as a mirror, then the repository manager will cache the dependencies; when job Y asks for the same artifact, the download will be local.
- **Acts as a mechanism to share snapshots locally**: Perhaps some of your snapshots are only for local consumption. The repository manager has facilities to limit access.
- **A GUI interface for ease of artifact management**: All three repository managers have intuitive GUIs, making your management tasks as easy as possible.

With these considerations in mind, if you are seeing a buildup of artifacts in Jenkins where they are less accessible and beneficial than deployed to a repository, consider this a signal of the need to upgrade your infrastructure.

For further reading, visit http://maven.apache.org/repository-management.html.

**Retention policy**

Jenkins can be a significant consumer of disk space. In the job configuration, you can decide to either keep artifacts or remove them automatically after a given period of time. The issue with removing artifacts is that you will also remove the results from any automatic testing. Luckily, there is a simple trick for you to avoid this. When configuring a job, click on **Discard Old Builds**, check the **Advanced** checkbox, and define **Max # of builds to keep with artifacts**. The artifacts are then removed after the number of builds specified, but the logs and results are kept. This has one important consequence: you have now allowed the reporting plugins to keep displaying a history of tests even though you have removed the other more disk-consuming artifacts.

**See also**

- The **Backing up and restoring** recipe
Deliberately failing builds through log parsing

Let's imagine you have been asked to clean up code that does not have any unit tests run during its build. There is a lot of code. To force the improvement of quality if you miss some residual defects, then you will want the Jenkins build to fail.

What you need is a flexible log parser that can fail or warn about issues found in the build output. To the rescue, this recipe describes how you can configure a log parsing plugin that spots unwanted patterns in the console output and fails jobs if the pattern is spotted. For example, a warning from Maven when there are no unit tests.

Getting ready

You will need to install the log-parser plugin as mentioned at https://wiki.jenkins-ci.org/display/JENKINS/Log+Parser+Plugin.

How to do it...

1. Create the log_rules directory owned by Jenkins under the Jenkins workspace.
2. Add the no_tests.rule file to the log_rules directory with one line:
   
   error /no tests/

3. Create a job with source code that gives deprecated warnings on compilation. In the following example, you are using the CLOG tool from the Sakai project:

   - **Job name**: Sakai_CLOG_Test
   - **Maven 2/3 Project**
   - **Source code Management**: Git
   - **Repository URL**: https://source.sakaiproject.org/contrib/clog/trunk
   - **Build**
   - **Maven Version**: 3.2.1 (or whatever your label is for your current version)
   - **Goals and options**: clean install

4. Run the build. It should not fail.
5. As shown in the following screenshot, visit the Manage Configuration page for Jenkins and add a description and the location of parsing rules file in the Console Output Parsing section:
6. Check the **Console output (build log) parsing** box in the **Post-build Actions** section of your *Sakai_CLOG_Test* job.

7. Check the **Mark build Failed on Error** checkbox:

![Console Output Parsing](image)

Select **Stop on no tests** for **Select Parsing Rules**.

Build the job and it should now fail.

8. Click on the **Parsed Console Output** link in the left-hand menu. You will now be able to see the parsed errors, as shown in the following screenshot:

![Parsed Console Output](image)
How it works...

The global configuration page allows you to add files each with a set of parsing rules. The rules use regular expressions mentioned in the home page of the plugin (https://wiki.jenkins-ci.org/display/JENKINS/Log+Parser+Plugin).

The rule file you used is composed of one line: `error /no tests/`.

If the **no tests** pattern (a case-sensitive test) is found in the console output, then the plugin considers this an **error** and the build fails. More lines to test can be added to the file. The first rule found wins. Other levels include **warn** and **ok**.

The source code was pulled in from the Sakai (http://www.sakaiproject.org) areas where no unit tests exist.

The rules file has the distinct `.rules` extension in case you want to write an exclude rule during backups.

Once the plugin is installed, you can choose per job between the rule files previously created.

This plugin empowers you to periodically scan for obvious lint and adapt to new circumstances. You should consider systematically sweeping through a series of rule files that fail suspect builds, until a full cleanup to the in-house style has taken place.

There's more...

Two other examples of common log patterns that are an issue but do not normally fail a build are:

- **MD5 checksums**: If a Maven repository has an artifact but not its associated MD5 checksum file, then the build will download the artifact even if it already has a copy. Luckily, the process will leave a **warn** message in the console output.

- **Failure to start up custom integration services**: These failures might be logged at the **warn** or **info** level when you really want them to fail the build.

See also

- The *Adding a job to warn of storage use violations through log parsing* recipe
Adding a job to warn of storage use violations through log parsing

The disk usage plugin is unlikely to fulfill all your disk maintenance requirements. This recipe will show you how you can strengthen disk monitoring by adding a custom Perl script to warn about disk usage violations.

The script will generate two alerts: a hard error when disk usage is above an acceptable level and a soft warning when the disk is getting near to that limit. The log-parser plugin will then react appropriately.

Getting ready

If you have not already done so, create a directory owned by Jenkins under the Jenkins workspace named log_rules. Also make sure that the Perl scripting language is installed on your computer and is accessible by Jenkins. Perl is installed by default on Linux distributions. ActiveState provides a decent Perl distribution for Mac and Windows (http://www.activestate.com/downloads).

How to do it...

1. Add a file named disk.rule to the log_rules directory with the following two lines:
   
   error /HARD_LIMIT/
   warn /SOFT_LIMIT/

2. Visit the Manage Configuration page for Jenkins and add the description DISC_USAGE to the Console Output section. Point to the location of the parsing rules file.

3. Add the following Perl script named disk_limits.pl to a location of choice, making sure that the Jenkins user can read the file:
   
   use File::Find;
   my $content = "/var/lib/jenkins";
   if ($#ARGV != 1) {
     print "[MISCONFIG ERROR] usage: hard soft (in Bytes)\n";
     exit(-1);
Maintaining Jenkins

```perl
my $total_bytes=0;
my $hard_limit=$ARGV[0];
my $soft_limit=$ARGV[1];

find( \&size_summary, $content );

if ($total_bytes>= $hard_limit){
    print "[HARD_LIMIT ERROR] $total_bytes>= $hard_limit (Bytes)\n";
}elsif ($total_bytes>= $soft_limit){
    print "[SOFT_LIMIT WARN] $total_bytes>= $soft_limit (Bytes)\n";
}else{
    print "[SUCCESS] total bytes = $total_bytes\n";
}

sub size_summary {
    if (-f $File::Find::name){
        $total_bytes+= -s $File::Find::name;
    }
}
```

4. Modify the $content variable to point to the Jenkins workspace.
5. Create a free-style software project job.
6. Under the Build section, add Build Step / Execute Shell. For the command, add `perl disk_limits.pl 9000000 2000000`.
7. Feel free to change the hard and soft limits (9000000 2000000).
8. Check Console output (build log) parsing in Post-build Actions.
9. Check the Mark build Unstable on Warning checkbox.
10. Check the Mark build Failed on Error checkbox.
11. Select the Parsing rules file as DISC_USAGE:
12. Run the build a number of times.

13. Under Build History on the left-hand side, select the trend link. You can now view trend reports and see a timeline of success and failure, as shown in the following screenshot:

![Build History](image)

### How it works...

The Perl script expects two command-line inputs: hard and soft limits. The hard limit is the value in bytes that the disk utilization under the `content` directory should not exceed. The soft limit is a smaller value in bytes that triggers a warning rather than an error. The warning gives time to administrators to clean up before the hard limit is reached.

The Perl script transverses the Jenkins workspace and counts the size of all the files. The script calls the `size_summary` method for each file or directory underneath the workspace.

If the hard limit is less than the content size, then the script generates the log output `[HARD_LIMIT ERROR]`. The parsing rules will pick this up and fail the build. If the soft limit is reached, then the script will generate the output `[SOFT_LIMIT WARN]`. The plugin will spot this due to the `warn /SOFT_LIMIT/` rule and then signal a job `warn`.

### There's more...

Welcome to the wonderful world of Jenkins. You can now utilize all the installed features at your disposal. The job can be scheduled and e-mails can be sent out on failure. You can also tweet, add entries to Google calendar, and trigger extra events such as disk cleaning builds and more. You are mostly limited by your imagination and twenty-first century technologies.

### See also

- The `Backing up and restoring` recipe
Keeping in contact with Jenkins through Firefox

If you are a Jenkins administrator, then it is your role to keep an eye on the ebb and flow of build activity within your infrastructure. Builds can occasionally freeze or break due to non-coding reasons. If a build fails and this is related to infrastructural issues, then you will need to be warned quickly. Jenkins can do this in numerous ways. Chapter 4, Communicating Through Jenkins, is dedicated to different approaches for different audiences. From e-mail, Twitter, and speaking servers, you can choose a wide range of prods, kicks, shouts, and pings. I could even imagine a Google Summer of Code project with a remotely controlled buggy moving to the sleeping administrator and then tooting.

This recipe is one of the more pleasant ways for you to be reached. You will pull in Jenkins RSS feeds using a Firefox add-on. This allows you to view the build process while going about your everyday business.

Getting ready

You will need Jenkins installed on your computer and an account on at least one Jenkins instance with a history of running jobs. You will also need to add the Status-4-Evar plugin, which you can get from https://addons.mozilla.org/en-US/firefox/addon/status-4-evar/.

How to do it...

1. Select the open menu icon at the top right-hand side of the browser:
2. Click on the add-ons button:

3. In Search bulk (top right), the Search all add-ons title searches for Jenkins.
4. Click on the Install button for Jenkins build monitor.
5. Restart Firefox.
6. Now, at the bottom right-hand side of Firefox, you will see a small Jenkins icon:

7. Right-click on the icon.
8. Select Preferences and the Feeds screen appears.
9. Add a recognizable but short name for your Jenkins instance. For example, Plugin test Server.
10. Add a URL using the following structure for Feed URL http://host:port/rssAll for example, http://localhost:8080/rssAll:

11. Check Enable executor monitoring.
12. Click on the OK button. An area in the add-on toolbar will appear with the Plugin test Server name of the feed URL(s) displayed and a health icon. If you hover your mouse over the name, more detailed status information will be displayed:

![Plugin test Server](http://host:port/job/job name/rssAll)

How it works...

Jenkins provides RSS feeds to make its status information accessible to a wide variety of tools. The Firefox add-on polls the configured feed and displays the information in a digestible format.

To configure for a specific crucial job, you will need to use the following structure:

http://host:port/job/job name/rssAll

To view only build failures, replace rssAll with rssFailed. To view only the last build, replace rssAll with rssLatest.

There's more...

Here are a few more things to consider.

**RSS credentials**

If security is enabled on your Jenkins instances, then most of your RSS feeds will be password-protected. To add a password, you will need to modify the feed URL to the following structure:

http://username:password@host:port/path

**Warning**

The negative aspect of using this add-on is that any feed URL password is displayed in plain text during editing.

Alternatives to Firefox

Firefox runs on a number of operating systems. This allows you to use one plugin for notifications across those operating systems. However, the disadvantage is that you have to keep a Firefox browser running in the background. An alternative is OS-specific notification software that pops up in the system tray. Examples of this type of software include CCMenu for Mac OSX (http://ccmenu.org) or CCTray for Windows (http://en.sourceforge.jp/projects/sfnet_ccnet/releases/).
Monitoring via JavaMelody

JavaMelody (http://code.google.com/p/javamelody/) is an open source project that provides comprehensive monitoring. The Jenkins plugin monitors both the master instance of Jenkins and also its nodes. The plugin provides a detailed wealth of important information. You can view evolution charts ranging for 1 day or week to months of the main quantities such as CPU or memory. Evolution charts are very good at pinpointing the scheduled jobs that are resource-hungry. JavaMelody allows you to keep a pulse on incremental degradation of resources. It eases the writing of reports by exporting statistics in a PDF format. Containing over 25 person years of effort, JavaMelody is feature-rich.

This recipe shows you how easy it is to install the Monitoring plugin (https://wiki.jenkins-ci.org/display/Jenkins/Monitoring) and then discusses troubleshooting strategies and their relationship to the generated metrics.

Community partnership

If you find this plugin useful, consider contributing back to either the plugin or the core JavaMelody project.

Getting ready

You will need to have installed the Monitoring plugin.
How to do it...

1. Click on the **Monitoring Jenkins master** link on the **Manage Jenkins** page. You will now see the detailed monitoring information, as shown in the following screenshot:


How it works...

JavaMelody has the advantage of running as the Jenkins user and can gain access to all the relevant metrics. Its main disadvantage is that it runs as part of the server and will stop monitoring as soon as there is a failure. Because of this disadvantage, you should consider JavaMelody as part of the monitoring solution and not the whole solution.

There's more...

Monitoring is the foundation for comprehensive testing and troubleshooting. This section explores the relation between these issues and the measurements exposed in the plugin.

Troubleshooting with JavaMelody – memory

Your Jenkins server can at times have memory issues due to greedy builds, leaky plugins, or some hidden complexity in the infrastructure. JavaMelody has a comprehensive range of memory measurements, including a heap dump and a memory histogram.

The Java virtual machine divides memory into various areas and, to clean up, it removes objects that have no references to other objects. Garbage collection can be CPU-intensive when it is busy and the nearer to full memory, the busier garbage collection becomes. To an external monitoring agent, this looks like a CPU spike that is often difficult to track down. Just because the garbage collector manages memory, it is also a fallacy to believe there is no potential for memory leakage in Java. Memory can be held too long by many common practices such as custom caches or calls to native libraries.

Slow burning memory leaks will show up as gentle slopes on the memory-related evolution graphs. If you suspect that you have a memory leak, then you can get the plugin to force a full garbage collection through the Execute the garbage collector link. If it is not a memory leak, the gentle slope will abruptly fall.

Memory issues can also express themselves as large CPU spikes as the garbage collector frantically tries to clean up, but can barely clean enough space. The garbage collector can also pause the application while comprehensively looking for no longer referenced objects ("Stop the world garbage collection") and thus cause large response times for web browser requests. This can be seen through the mean and max times in Statistics http – 1 day.
Maintaining Jenkins

Troubleshooting with JavaMelody – painful jobs

You should consider the following points:

- **Offload work**: For a stable infrastructure, offload as much work from the master instance as possible. If you have scheduled tasks, keep the heaviest ones separate in time. Time separation not only evens out load, but also makes finding the problematic build easier through the observation of the evolution charts of JavaMelody. Also consider spatial separation; if a given node or a labeled set of nodes shows problematic issues, then start switching around machine location of jobs and view their individual performance characteristics through http://host:port/monitoring/nodes.

- **Hardware is cheap**: Compared to paying for human hours, buying an extra 8 GB is roughly equivalent to one man’s hour effort.

  A common gotcha is to add memory to the server, while forgetting to update the init scripts to allow Jenkins to use more memory.

- **Review the build scripts**: Javadoc generation and custom Ant scripts can fork JVMs and reserve memory defined within their own configuration. Programming errors can also be the cause of the frustration. Don’t forget to review JavaMelody’s report on Statistic system error log and Statistic http system errors.

- **Don’t forget external factors**: Factors include backups, cron jobs, updating the locate database, and network maintenance. These will show up as periodic patterns in the evolution charts.

- **Strength in numbers**: Use JavaMelody in combination with the disk usage plugin and others to keep a comprehensive overview of the vital statistics. Each plugin is simple to configure, but their usefulness to you will grow quicker than the maintenance costs of adding extra plugins.

See also

- The Using Groovy hook scripts and triggering events on startup recipe in Chapter 7, Exploring Plugins
Keeping track of script glue

There are negative implications for backing up and especially restoring if maintenance scripts are scattered across the infrastructure. It is better to keep your scripts in one place and then run them remotely through the nodes. Consider placing your scripts under the master Jenkins home directory and back up to a Git repository. It would be even better for the community if you can share the less sensitive scripts online. Your organization can reap the benefits; the scripts will then get some significant peer review and improvements. For the communities repository details, review the support information at http://localhost:8080/scriptler.git/.

In this recipe, we will explore the use of the Scriptler plugin to manage your scripts locally and download useful scripts from an online catalogue.

Getting ready

You will need to install the Scriptler plugin (https://wiki.jenkins-ci.org/display/JENKINS/Scriptler+Plugin).

How to do it...

1. Click on the Scriptler link under the Manage Jenkins page. You will notice the text in bold. Currently you do not have any scripts available; you can import scripts from a remote catalog or create your own.

2. Click on the Remote Script catalogs link on the left-hand side.

3. Click on the ScriptierWeb tab.

4. Click on the icon of the floppy disk for getThreadDump. If the script is not available, then choose another script of your choice.

5. Click on the Submit button.

6. You have now returned to the Scriptler main page. You will see three icons. Choose the furthest right to execute the script:

7. You are now in the Run a script page. Select a node and then hit the Run button.

If the script fails with a message startup failed, then please add a new line between entry.key and for and the script will then function correctly.
8. To write a new Groovy script or upload the one that you have on your local system, click on the Add a new Script link on the left-hand side.

**How it works...**

This plugin allows you to easily manage your Groovy scripts and enforces a standard place for all Jenkins administrators to keep their code, making it easier for you to plan backups and indirectly share knowledge.

The plugin creates a directory named scriptler under the Jenkins workspace and persists the metainformation about the files you created in the scriptler.xml file. A second file named scriptlerweb-catalog.xml mentions the list of online files that you can download.

All local scripts are contained in the subdirectory scripts.

**There's more...**

If enough people use this plugin, then the list of online scripts will radically increase the process of generating a significant library of reusable code. Therefore, if you have interesting Groovy scripts, then upload them. You will need to create a new account the first time to log in at http://scriptlerweb.appspot.com/login.gtpl.

Uploading your scripts allows people to vote on them and to send you feedback. The free peer review can only improve your scripting skills and increase your recognition in the wider community.

**See also**

- The Scripting the Jenkins CLI recipe
- The Global modifications of jobs with Groovy recipe

**Scripting the Jenkins CLI**

The Jenkins CLI (https://wiki.jenkins-ci.org/display/JENKINS/Jenkins+CLI) allows you to perform a number of maintenance tasks on remote servers. Tasks include moving Jenkins instances on-and off-line, triggering builds, and running Groovy scripts. This makes for easy scripting of the most common chores.

In this recipe, you will log on to a Jenkins instance, run a Groovy script that looks for files greater than a certain size, and log off. The script represents a typical maintenance task. After reviewing the output, you can run a second script to remove a list of files you want deleted.
At the time of writing this chapter, the interactive Groovy shell was not working from the CLI. This is mentioned in the bug report at https://issues.jenkins-ci.org/browse/JENKINS-5930.

### Getting ready


Add the following script to a directory under the control of Jenkins and call it `large_files.groovy`:

```groovy
root = jenkins.model.Jenkins.instance.getRootDir()
count = 0
size = 0
maxsize=1024*1024*32
root.eachFileRecurse() { file ->
count++
size+=file.size();
if (file.size() > maxsize) {
    println "Thinking about deleting: " + file.getPath()
    // do things to large files here
}
}
println "Space used ${size/(1024*1024)} MB Number of files ${count}"
```

### How to do it...

1. Run the following command from a terminal replacing `http://host` with the real address of your server, for example, `http://localhost:8080`.
   ```bash
   java -jar jenkins-cli.jar -s http://host login --username username
   ```
   2. Input your password.
   3. Look at the online help:
   ```bash
   java -jar jenkins-cli.jar -s http://host help
   ```
   4. Run the Groovy script. The output will now mention all the oversized files:
   ```bash
   java -jar jenkins-cli.jar -s http://host groovy large_files.groovy
   ```
   5. Logout by running the following command:
   ```bash
   java -jar jenkins-cli.jar -s http://host logout.
   ```
How it works...

The CLI allows you to work from the command line and perform standard tasks. Wrapping the CLI in a shell script such as Bash allows you to script maintenance tasks and a large number of Jenkins instances at the same time. This recipe performs a lot of maintenance. In this case, it reviews x number of files for oversized artifacts, saving you time that you can better spend on more interesting tasks.

Before performing any commands, you needed to first authenticate via the login command.

Reviewing the `root = jenkins.model.Jenkins.instance.getRootDir()` script uses the Jenkins framework to obtain a `java.io.File` that points to the Jenkins workspace.

The maximum file size is set to 32 MB via `maxsize=1024*1024*32`.

The script visits every file under the Jenkins workspace using the standard `root`. `eachFileRecurse(){ file -> Groovy method.`

You can find the current JavaDoc for Jenkins at http://javadoc.jenkins-ci.org/.

There's more...

The authentication used in this recipe can be improved. You can add your SSH public key under http://localhost:8080/user/{username}/configure (where username is your username) by cutting and pasting into the SSH Public Keys section. You can find detailed instructions at https://wiki.jenkins-ci.org/display/JENKINS/Jenkins+CLI.

At the time of writing this book, there were some issues with the key approach. For more information, visit https://issues.jenkins-ci.org/browse/JENKINS-10647. Feel free to resort to the method used in this recipe that has proven to work stably, though less securely.

The CLI is easily extendable and therefore, over time, the CLI's command list increases. It is thus important that you occasionally check the help option.

See also

- The Global modifications of jobs with Groovy recipe
- The Scripting global build reports recipe
Global modifications of jobs with Groovy

Jenkins is not only a Continuous Integration server, but also a rich framework with an exposed internal structure available from within the script console. You can programmatically iterate through the jobs, plugins, node configuration, and a variety of rich objects. As the number of jobs increases, you will notice the scripting becomes more valuable. For example, imagine that you need to increase custom memory settings across 100 jobs. A Groovy script can do that in seconds.

This recipe is a representative example: you will run a script that iterates through all jobs. The script then finds one specific job by its name and then updates the description of that job with a random number.

Getting ready

Log in to Jenkins with an administrative account.

How to do it...

1. Create an empty job named MyTest.
2. Within the Manage Jenkins page, click on the Script Console link.
3. Click on Add new script.
4. Cut and paste the following script into the Script text area input:

```java
import java.util.Random
Random random = new Random()

hudson.model.Hudson.instance.items.each{ job ->
  println ("Class: 
job.class")
  println ("Name: 
job.name")
  println ("Root Dir: 
job.rootDir")
  println ("URL: 
job.url")
  println ("Absolute URL: 
job.absoluteUrl")

  if ("MyTest".equals(job.name)){
    println ("Description: 
job.description")
    job.setDescription("This is a test id: 
random.nextInt(99999999)")
  }
}
```
5. Click on the Run button. The results should be similar to the following screenshot:

```
Result
Class: class hudson.matrix.MatrixProject
Name: MyTest
Root Dir: /var/lib/jenkins/jobs/MyTest
URL: job/MyTest/
Absolute URL: http://localhost:8080/job/MyTest/
Description: This is a test id: 75447531
Result: [hudson.matrix.MatrixProject@5575b132[MyTest]]
```

6. Run the script again; you will notice that the random number in the description has now changed.

7. Copy and run the following script:

```java
for (slave in hudson.model.Hudson.instance.slaves) {
    println "Slave class: \
    println "Slave name: \
    println "Slave URL: \
    println "Slave URL: \
}
```

If you have no slave instances on your Jenkins master, then no results are returned. Otherwise, the output will look similar to the following screenshot:
Jenkins has a rich framework that is exposed to the script console. The first script iterates through jobs whose parent is `AbstractItem` ([http://javadoc.jenkins-ci.org/hudson/model/AbstractItem.html](http://javadoc.jenkins-ci.org/hudson/model/AbstractItem.html)). The second script iterates through instances of `slave` objects ([http://javadoc.jenkins-ci.org/hudson/slaves/SlaveComputer.html](http://javadoc.jenkins-ci.org/hudson/slaves/SlaveComputer.html)).

For the hard core Java developer: if you don't know how to do a programmatic task, then excellent sources of example code are the Jenkins subversion directories for plugins ([https://svn.jenkins-ci.org/trunk/hudson/plugins/](https://svn.jenkins-ci.org/trunk/hudson/plugins/)) and the more up-to-date Github location ([https://github.com/jenkinsci](https://github.com/jenkinsci)).

If you're interested in donating your own plugin, review the information at [https://wiki.jenkins-ci.org/display/JENKINS/Hosting+Plugins](https://wiki.jenkins-ci.org/display/JENKINS/Hosting+Plugins).

Each development team is unique. Teams have their own way of doing business. In many organizations, there are one-off tasks that need to be done periodically. For example, at the end of each year, making a full backup of the entire filesystem.

This recipe details a script that checks for the last successful run of any job; if the year is different to the current year, then a warning is set at the beginning of the jobs description. Thus it is hinting to you it’s time to perform some action, such as archiving and then deleting. You can of course programmatically do the archiving. However, for high-value actions it is worth forcing interceding, letting the Groovy scripts focus your attention.

Log in to Jenkins with an administrative account.
How to do it...

1. Within the Manage Jenkins page, click on the Script Console link and run the following script:
   ```java
   Import hudson.model.Run;
   Import java.text.DateFormat;

   def warning='\n   def now=new Date()
   for (job in hudson.model.Hudson.instance.items) {
      println "\n      Run lastSuccessfulBuild = job.getLastSuccessfulBuild()
      if (lastSuccessfulBuild != null) {
         def time = lastSuccessfulBuild.getTimestamp().getTime()
         if (now.year.equals(time.year)){
            println("Project has same year as build");
         }else {
            if (job.description.startsWith(warning)){
               println("Description has already been changed");
            }else{
               job.setDescription("${warning}${job.description}"
            }
         }
      }
   }
   
   Any project that had its last successful build in another year than this will have
   the word [ARCHIVE] in red added at the start of its description, as shown in the
   following screenshot:
   
   Project Simple Job
   [ARCHIVE] Yet another project
   ```

How it works...

Reviewing the code listing:

- A warning string is defined and the current date is stored in `now`. Each job in Jenkins
  is programmatically iterated via the `for` statement.
Jenkins has a class to store build run information. The runtime information is retrieved via \texttt{job.getLastSuccessfulBuild()} and is stored in the \texttt{lastSuccessfulBuild} instance. If no successful build has occurred, then \texttt{lastSuccessfulBuild} is set to \texttt{null}; otherwise, it has the runtime information.

The time of the last successful build is retrieved and then stored in the \texttt{time} instance via \texttt{lastSuccessfulBuild.getTimestamp().getTime()}.

The current year is compared with the year of the last successful build and, if they are different and the warning string has not already been added to the front of the job description, then the description is updated.

\begin{quote}
\textbf{Javadoc}

You will find the job API mentioned at \url{http://javadoc.jenkins-ci.org/hudson/model/Job.html} and the Run information at \url{http://javadoc.jenkins-ci.org/hudson/model/Run.html}.
\end{quote}

\begin{quote}
\textbf{There's more...}

Before writing your own code, you should review what already exists. With 1,000 plugins and expanding, Jenkins has a large, freely available, and openly licensed example code base. Although in this case, the standard API was used, it is well worth reviewing the plugin code base. In this example, you will find part of the code reused from the \texttt{lastsuccessversioncolumn} plugin. (\url{http://tinyurl.com/pack-jenkins-1}).

If you find any defects while reviewing the plugin code base, please contribute to the community via patches and bug reports.
\end{quote}

\begin{quote}
\textbf{See also}

- The \textit{Scripting the Jenkins CLI} recipe
- The \textit{Global modifications of jobs with Groovy} recipe
\end{quote}
Where to buy this book

You can buy Jenkins Continuous Integration Cookbook Second Edition from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.