

Quick answers to common problems

Jenkins Continuous Integration Cookbook

Over 80 recipes to maintain, secure, communicate, test, build, and improve the software development process with Jenkins

Alan Mark Berg

[PACKT] open source* PUBLISHING community experience distilled

Jenkins Continuous Integration Cookbook

Over 80 recipes to maintain, secure, communicate, test, build, and improve the software development process with Jenkins

Alan Mark Berg

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Jenkins Continuous Integration Cookbook

Copyright © 2012 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: June 2012

Production Reference: 1080612

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-849517-40-9

www.packtpub.com

Cover Image by Faiz Fattohi (faizfattohi@gmail.com)

Credits

Author

Alan Mark Berg

Reviewers

Dr. Alex Blewitt

Florent Delannoy

Michael Peacock

Acquisition Editor

Usha Iyer

Lead Technical Editor

Azharuddin Shaikh

Technical Editors

Merin Jose

Lubna Shaikh

Copy Editor

Brandt D'Mello

Project Coordinator

Leena Purkait

Proofreader

Jonathan Todd

Indexers

Tejal Daruwale

Hemangini Bari

Production Coordinator

Arvindkumar Gupta

Cover Work

Arvindkumar Gupta

About the Author

Alan Mark Berg, Bsc, MSc, PGCE, has been the lead developer at the Central Computer Services at the University of Amsterdam for the last 12 years. In his famously scarce spare time, he writes. Alan has a degree, two Master's, and a teaching qualification. He has also co-authored two books about **Sakai** (`http://sakaiproject.org`)—a highly successful open source learning management platform used by many millions of students around the world. Alan has also won a Sakai Fellowship.

In previous incarnations, Alan was a technical writer, an Internet/Linux course writer, a product line development officer, and a teacher. He likes to get his hands dirty with the building and gluing of systems. He remains agile by ruining various development and acceptance environments.

I would like to thank Hester, Nelson, and Lawrence. I felt supported and occasionally understood by my family. Yes, you may pretend you don't know me, but you do. Without your unwritten understanding that 2 a.m. is a normal time to work and a constant supply of sarcasm is good for my soul, I would not have finished this or any other large-scale project.

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

About the Reviewers

Dr. Alex Blewitt is a technical architect, working at an investment bank in London. He has recently won an Eclipse Community Award at EclipseCon 2012 for his involvement with the Eclipse platform over the last decade. He also writes for InfoQ and has presented at many conferences. In addition to being an expert in Java, he also develops for the iOS platform, and when the weather's nice, he goes flying. His blog is at `http://alblue.bandlem.com`, and he can be reached via `@alblue` on Twitter.

Florent Delannoy is a French software engineer, now living in New Zealand after graduating with honors from a MSc in Lyon. Passionate about open source, clean code, and high quality software, he is currently working on one of New Zealand's largest domestic websites with Catalyst I.T. in Wellington.

I would like to thank my family for their support and my colleagues at Catalyst for providing an amazingly talented, open, and supportive workplace.

Michael Peacock (`www.michaelpeacock.co.uk`) is an experienced senior/lead developer and Zend Certified Engineer from Newcastle, UK, with a degree in Software Engineering from the University of Durham.

After spending a number of years running his own web agency, managing the development team, and working for Smith Electric Vehicles on developing its web-based vehicle telematics platform, he currently serves as head developer for an ambitious new start-up: leading the development team and managing the software development processes.

He is the author of *Drupal 7 Social Networking*, *PHP 5 Social Networking*, *PHP 5 E-Commerce Development*, *Drupal 6 Social Networking*, *Selling online with Drupal E-Commerce*, and *Building Websites with TYPO3*. Other publications in which Michael has been involved include *Mobile Web Development* and *Drupal for Education and E-Learning*, both of which he acted as technical reviewer for.

Michael has also presented at a number of user groups and conferences including PHPNE, PHPNW10, CloudConnect, and ConFoo,

You can follow Michael on Twitter: `www.twitter.com/michaelpeacock`, or find out more about him through his blog: `www.michaelpeacock.co.uk`.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read, and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface 1

Chapter 1: Maintaining Jenkins 7

- Introduction 8
- Using a sacrificial Jenkins instance 9
- Backing up and restoring 13
- Modifying Jenkins configuration from the command line 18
- Reporting overall disc usage 21
- Deliberately failing builds through log parsing 24
- A Job to warn about the disc usage violations through log parsing 27
- Keeping in contact with Jenkins through Firefox 30
- Monitoring through JavaMelody 32
- Keeping a track of the script glue 36
- Scripting the Jenkins command-line interface 37
- Global modifications of Jobs with Groovy 40
- Signaling the need to archive 42

Chapter 2: Enhancing Security 45

- Introduction 45
- Testing for OWASP's top ten security issues 47
- Finding 500 errors and XSS attacks in Jenkins through fuzzing 50
- Improving security via small configuration changes 53
- Looking at the Jenkins user through Groovy 56
- Working with the Audit Trail plugin 58
- Installing OpenLDAP with a test user and group 61
- Using Script Realm authentication for provisioning 64
- Reviewing project-based matrix tactics via a custom group script 67
- Administering OpenLDAP 70
- Configuring the LDAP plugin 74

Installing a CAS server 77
Enabling SSO in Jenkins 83
Chapter 3: Building Software 85
Introduction 85
Plotting alternative code metrics in Jenkins 88
Running Groovy scripts through Maven 93
Manipulating environmental variables 97
Running AntBuilder through Groovy in Maven 102
Failing Jenkins Jobs based on JSP syntax errors 106
Configuring Jetty for integration tests 111
Looking at license violations with RATs 114
Reviewing license violations from within Maven 117
Exposing information through build descriptions 121
Reacting to the generated data with the Post-build Groovy plugin 123
Remotely triggering Jobs through the Jenkins API 126
Adaptive site generation 129
Chapter 4: Communicating Through Jenkins 135
Introduction 136
Skinning Jenkins with the Simple Theme plugin 137
Skinning and provisioning Jenkins using a WAR overlay 140
Generating a home page 145
Creating HTML reports 148
Efficient use of views 151
Saving screen space with the Dashboard plugin 153
Making noise with HTML5 browsers 155
An eXtreme view for reception areas 158
Mobile presentation using Google Calendar 160
Tweeting the world 163
Mobile apps for Android and iOS 166
Getting to know your audience with Google Analytics 169
Chapter 5: Using Metrics to Improve Quality 173
Introduction 174
Estimating the value of your project through Sloccount 176
Looking for "smelly" code through code coverage 179
Activating more PMD rulesets 183
Creating custom PMD rules 188
Finding bugs with FindBugs 193
Enabling extra FindBugs rules 197

Finding security defects with FindBugs 199
Verifying HTML validity 203
Reporting with JavaNCSS 205
Checking style using an external pom.xml 207
Faking checkstyle results 210
Integrating Jenkins with Sonar 213

Chapter 6: Testing Remotely 217
Introduction 217
Deploying a WAR file from Jenkins to Tomcat 219
Creating multiple Jenkins nodes 222
Testing with Fitnesse 226
Activating Fitnesse HtmlUnit Fixtures 230
Running Selenium IDE tests 234
Triggering Failsafe integration tests with Selenium Webdriver 240
Creating JMeter test plans 244
Reporting JMeter performance metrics 246
Functional testing using JMeter assertions 249
Enabling Sakai web services 253
Writing test plans with SoapUI 256
Reporting SoapUI test results 259

Chapter 7: Exploring Plugins 265
Introduction 265
Personalizing Jenkins 267
Testing and then promoting 270
Fun with pinning JS Games 274
Looking at the GUI Samples plugin 277
Changing the help of the file system scm plugin 280
Adding a banner to Job descriptions 283
Creating a RootAction plugin 288
Exporting data 291
Triggering events on startup 293
Triggering events when web content changes 295
Reviewing three ListView plugins 297
Creating my first ListView plugin 301

Appendix: Processes that Improve Quality 309
Avoiding group think 309
Considering test automation as a software project 310
Offsetting work to Jenkins nodes 311

Learning from history 311
Test frameworks are emerging 312
Starve QA/ integration servers 313
And there's always more 313
Final comments 314

Index 315

Preface

Jenkins is a Java-based **Continuous Integration** (**CI**) server that supports the discovery of defects early in the software cycle. Thanks to over 400 plugins, 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 the naming conventions, documentation, how the software is designed, build scripts, the process of deploying the software to servers, and so on. Continuous integration 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 lifecycle, 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 1000 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. As you can imagine, in my day job as a Senior Developer with aspirations to 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 through 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.

The following are the advantages of Jenkins:

- It is proven technology that is deployed at a large scale in many organizations.
- It is an open source technology, so the code is open to review and has no licensing costs.
- It has a simple configuration through a web-based GUI, which speeds up Job creation, improves consistency, and decreases the maintenance costs.
- It is a master slave topology that distributes the build and testing effort over slave servers with the results automatically accumulated on the master. This topology ensures a scalable, responsive, and stable environment.

- It has the ability to call slaves from the cloud. Jenkins can use Amazon services or an **Application Service Provider** (**ASP**), such as **CloudBees** (`http://www.cloudbees.com/`).
- There is no fuss in installation, as installation is as simple as running only a single download file named `jenkins.war`.
- It has over 400 plugins supporting communication, testing, and integration to numerous external applications (`https://wiki.jenkins-ci.org/display/JENKINS/Plugins`).
- It is a straightforward plugin framework—for Java programmers, writing plugins is straightforward. Jenkins plugin framework has clear interfaces that are easy to extend. The framework uses **Xstream** for persisting configuration information as XML (`http://xstream.codehaus.org/`) and Jelly for the creation of parts of the GUI (`http://commons.apache.org/jelly/`).
- It has the facility to support running Groovy scripts, both in the master and remotely on slaves. This allows for consistent scripting across operating systems. However, you are not limited to scripting in Groovy. Many administrators like to use Ant, Bash, or Perl scripts.
- Though highly supportive of Java, Jenkins also supports other languages.
- Jenkins is an agile project; you can see numerous releases in the year, pushing improvements rapidly (`http://jenkins-ci.org/changelog`). There is also a highly stable long-term support release for the more conservative. Hence, there is a rapid pace of improvement.
- Jenkins pushes up code quality by automatically testing within a short period after code commit, and then shouting loudly if build failure occurs.

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

- Participate on 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, participate in the discussions. Consistently reading the lists will generate many opportunities to collaborate.
- Improve code, write plugins (`https://wiki.jenkins-ci.org/display/JENKINS/Help+Wanted`).
- Test Jenkins, especially the plugins, and write bug reports, donating your test plans.
- Improve documentation by writing tutorials and case studies.
- Sponsor and support events.

What this book covers

Chapter 1, *Maintaining Jenkins*, describes common maintenance tasks, such as backing up and monitoring.

Chapter 2, *Enhancing Security*, details how to secure Jenkins and the value of enabling **Single Sign On** (**SSO**).

Chapter 3, *Building Software*, explores the relationship between Jenkins builds and the Maven `pom.xml` file.

Chapter 4, *Communicating Through Jenkins*, reviews effective communication strategies for different target audiences, from developers and project managers to the wider public.

Chapter 5, *Using Metrics to Improve Quality*, explores the use of source code metrics.

Chapter 6, *Testing Remotely*, details approaches to set up and run remote stress and functional tests.

Chapter 7, *Exploring Plugins*, reviews a series of interesting plugins and shows how easy it is to create your first plugin.

Appendix, *Processes That Improve Quality*, discusses how the recipes in this book support quality processes.

What you need for this book

This book assumes you have a running an instance of Jenkins.

In order to run the recipes provided in the book, you need to have the following software:

Recommended

- **Maven 2**: `http://maven.apache.org/docs/2.2.1/release-notes.html`
- **Jenkins**: `http://jenkins-ci.org/`
- **Java version 1.6**: `http://java.com/en/download/index.jsp`

Optional

- **VirtualBox**: `https://www.virtualbox.org/`
- **SoapUI**: `http://www.soapui.org/`
- **JMeter**: `http://jmeter.apache.org/`

Helpful

- **A local subversion repository**
- **OS of preference**: Linux/Ubuntu

There are numerous ways to install Jenkins: as a Windows service, using the repository management features of Linux such as `apt` and `yum`, using Java Web Start, or running directly from a WAR file. It is up to you to choose the approach that you feel is most comfortable. However, you could run Jenkins from a WAR file using HTTPS from the command line, pointing to a custom directory. If any experiments go astray, then you can simply point to another directory and start fresh.

To use this approach, first set the environment variable `JENKINS_HOME` to the directory you wish Jenkins to run under. Next, run a command similar to the following:

```
Java –jar jenkins.war –httpsPort=8443 –httpPort=-1
```

Jenkins will start to run over `https` on port `8443`. The `http` port is turned off by setting `httpPort=-1`, and the terminal will display logging information.

You can ask for help through the following command:

```
Java –jar jenkins.war –help
```

A wider range of installation instructions can be found at `https://wiki.jenkins-ci.org/display/JENKINS/Installing+Jenkins`.

For a more advanced recipe describing how to set up a virtual image under VirtualBox with Jenkins, you can use the *Using a sacrificial Jenkins instance recipe in Chapter 1*, *Maintaining Jenkins*.

Who this book is for

This book is for Java developers, software architects, technical project managers, build managers, and development or QA engineers. A basic understanding of the Software Development Life Cycle, some elementary web development knowledge, and basic application server concepts are expected to be known. A basic understanding of Jenkins is also assumed.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text are shown as follows: "On the host OS, create a directory named `workspacej`."

A block of code is set as follows:

```
<?xml version='1.0' encoding='UTF-8'?>
<org.jvnet.hudson.plugins.thinbackup.ThinBackupPluginImpl>
  <fullBackupSchedule>1 0 * * 7</fullBackupSchedule>
  <diffBackupSchedule>1 1 * * *</diffBackupSchedule>
```

```
    <backupPath>/home/aberg/backups</backupPath>
    <nrMaxStoredFull>61</nrMaxStoredFull>
    <cleanupDiff>true</cleanupDiff>
    <moveOldBackupsToZipFile>true</moveOldBackupsToZipFile>
    <backupBuildResults>true</backupBuildResults>
    <excludedFilesRegex></excludedFilesRegex>
  </org.jvnet.hudson.plugins.thinbackup.ThinBackupPluginImpl>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
#!/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";
  }
}
```

Any command-line input or output is written as follows:

```
mvn license:format -Dyear=2012
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Within the VirtualBox, right-click on the Ubuntu image, selecting **properties**."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. 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.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Maintaining Jenkins

This chapter provides recipes that help you to maintain the health of your Jenkins server.

In this chapter, we will cover the following recipes:

- Using a sacrificial Jenkins instance
- Backing up and restoring
- Modifying the Jenkins configuration from the command line
- Reporting about the overall disc usage
- Deliberately failing builds through log parsing
- A Job to warn about the disc usage violations
- Keeping in contact with Jenkins through Firefox
- Monitoring through JavaMelody
- Keeping a track of the 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 with numerous external systems, and its **Jobs** work with many diverse technologies. Maintaining Jenkins in a rich environment is a challenge. Proper maintenance lowers the risk of failures, a few of which are listed as follows:

- **New plugins causing exceptions**: There are a lot of good plugins being written with a rapid version change. In this situation, it is easy for you to accidentally add new versions of the plugins with new defects. There have been a number of times when the plugin suddenly stopped working, while it was being upgraded. To combat the risk of plugin exceptions, consider using a sacrificial Jenkins instance before releasing to a critical system.
- **Disks overflowing with artifacts**: If you keep a build history, which includes artifacts such as war files, large sets of JAR files or other types of binaries, then your disk space is consumed at a surprising rate. Disc costs have decreased tremendously, but disk usage equates to longer backup times and more communication from the slave to the 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 increase, Jenkins slows down. Proper monitoring and quick reaction reduce their 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 lots of ways of performing the same work. Conventions improve the consistency and readability of Jobs and thus decrease the 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 through my Packt Publishing e-mail address, or it would even be better if you still add tutorials to the Jenkins community wiki.



Signing up to the community

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

Using a sacrificial Jenkins instance

Continuous Integration (CI) servers are critical in the creation of deterministic release cycles. Any long-term instability in the CI server will reflect in the milestones of your project plans. Incremental upgrading is addictive and mostly straightforward, but should be seen in the light of the Jenkins wider role.

Before the release of plugins into the world of your main development cycle, it is worth aggressively deploying to a sacrificial Jenkins instance, and then sitting back and letting the system run the Jobs. This gives you enough time to react to any minor defects found.

There are many ways to set up a sacrificial 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 hard core server.
- **Use a number of different operating systems**: Good for node machines running integration tests or functional tests with multiple browser types.
- **Swap workspaces**: By having the workspace outside the virtual image, you can test different version levels of the OS against one workspace. You can also test one version of Jenkins against different workspaces, with different plugin combinations.



The long-term support release

The community manages support of the enterprise through the release of a long-term supported version of Jenkins. This stable release version is older than the newest version and thus misses out on some of the newer features. You can download it from: `http://mirrors.jenkins-ci.org/war-stable/latest/jenkins.war`.

This recipe details the use of **VirtualBox** (`http://www.virtualbox.org/`), an open source virtual image player with a guest Debian OS image. 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, it will automatically run against the shared directory.



Throughout the rest of 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 the detailed instructions at `http://www.virtualbox.org/manual`. To download and unpack a Ubuntu virtual image, you can refer to the following URL: `http://sourceforge.net/projects/virtualboximage/files/Ubuntu%20Linux/11.04/ubuntu_11.04-x86.7z/download`.

Note that the newer images will be available at the time of reading. Feel free to try the most modern version; it is probable that the recipes might still work.



Security considerations

If you consider using other OS images a bad security practice, then you should create a Ubuntu image from a boot CD as mentioned at: `https://wiki.ubuntu.com/Testing/VirtualBox`

How to do it...

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. On the **Welcome** screen, click on the **Next** button.
3. Set the **Name** to `Jenkins_Ubuntu_11.04`. The **OS Type** will be automatically updated. Click on the **Next** button.
4. Set **Memory** to `2048` MB, and click on **Next**.
5. Select **Use existing hard disk**. Browse and select the unpacked VDI image by clicking on the folder icon.



6. Press the **Create** button.

7. Start the virtual image by clicking on the **Start** icon.



8. Log in to the guest OS with username and password as `Ubuntu reverse`.
9. Change the password of user Ubuntu from a terminal.

```
passwd
```

10. Install the Jenkins repository, as explained at `http://pkg.jenkins-ci.org/debian/`.
11. Update the OS in case of security patches (this may take some time depending on bandwidth):

```
apt-get update
apt-get upgrade
```

12. Install the kernel `dkms` module.

```
sudo apt-get install dkms
```

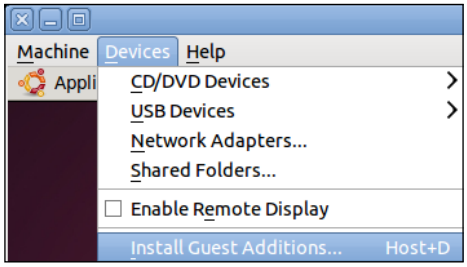
13. Install Jenkins.

```
sudo apt-get install jenkins
```

14. Install the kernel modules for VirtualBox.

```
/etc/init.d/vboxadd setup
```

15. Select **Install Guest additions** using the **Devices** menu option.



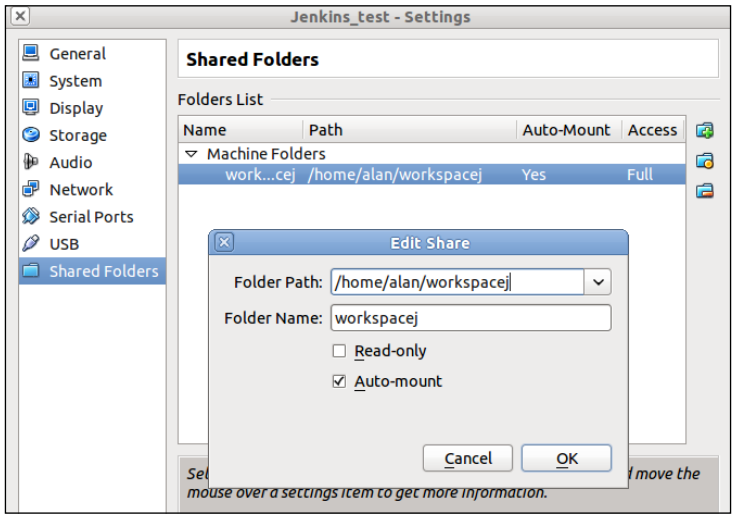
16. Add the Jenkins user to the `vboxsf` group.

```
sudo gedit /etc/group
vboxsf:x:1001:Jenkins
```

17. Modify the `JENKINS_HOME` variable in `/etc/default/Jenkins` to point to the mounted shared directory:

    ```
    sudo gedit /etc/default/Jenkins
    JENKINS_HOME=/media/sf_workspacej
    ```

18. On the host OS, create the directory `workspacej`.
19. Within the VirtualBox, right-click on the Ubuntu image, selecting **properties**.
20. Update the **Folder Path** to point to the directory that you have previously created. In the following screenshot, the folder was created under my home directory.



21. Restart VirtualBox, and start the Ubuntu Guest OS.
22. On the Guest OS, run a web browser, and visit `http://localhost:8080`. You will see a locally running instance of Jenkins, ready for your experiments.

How it works...

Your recipe first installs a virtual image of Ubuntu, changes the password so that it is harder for others to log in, and updates the guest OS for security patches.

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

You need to install the 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 command `/etc/init.d/vboxadd setup`, VirtualBox added guest addition modules through DKMS.



Warning: If you forget to add the DKMS module, then sharing folders will fail without any visual warning.

The default Jenkins instance now needs a little reconfiguration:

- The `jenkins` user needs to belong to the the `vboxsf` group to have permission to use the shared folder
- The `/etc/init.d/jenkins` startup script points to `/etc/default/jenkins` and 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 had 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 introductory text at: http://www.virtualbox.org/manual/ch06.html

There's more...

Two excellent sources of virtual images are:

- `http://virtualboximages.com/`
- `http://virtualboxes.org/images/`

See also

- *Monitoring through JavaMelody*

Backing up and restoring

A core task for the smooth running of Jenkins is the scheduled backing up of its workspace. Not necessarily backing up all the artifacts, but at least the Jenkins configuration and the testing history are recorded by individual plugins.

Backups are not interesting unless you can restore. There are a wide range of stories on this subject. My favorite (and I won't name the well-known company involved) is the one in which 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 time. 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 for 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, it's quite likely that extra options will be added, increasing the variables that you input in the GUI. Therefore, the screen grabs 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, write permissions for Jenkins, and install the `ThinBackup` plugin.

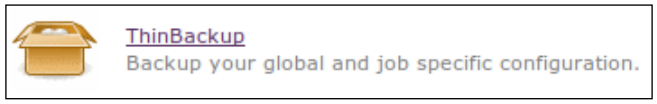


Murphy as a friend

You should assume the worst for all of the recipes in this book: aliens attacking, coffee on motherboard, cat eats cable, cable eats cat. Please make sure that you are using a sacrificial Jenkins instance.

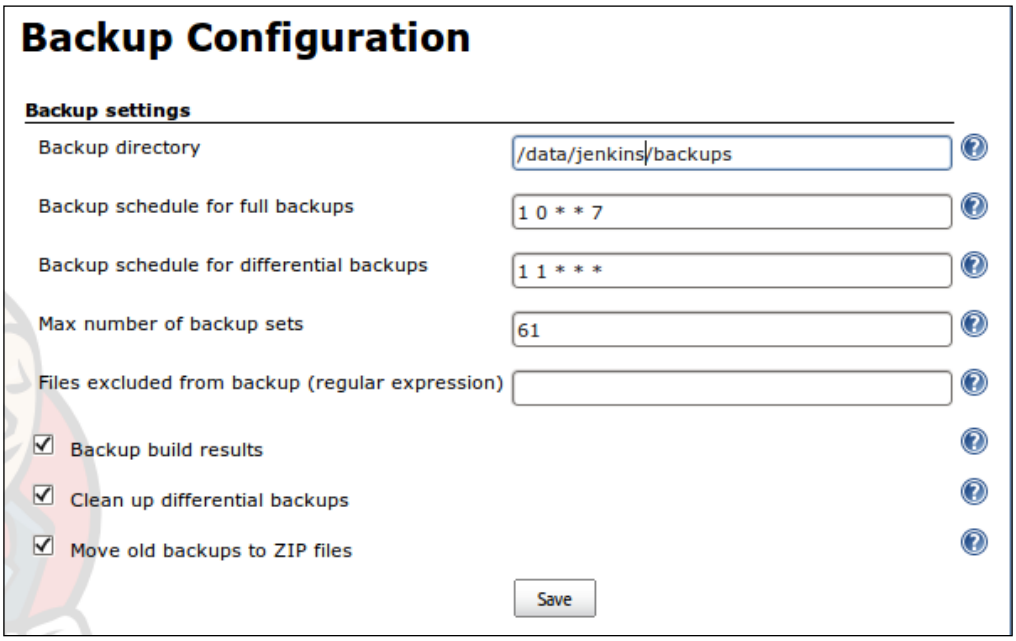
How to do it...

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

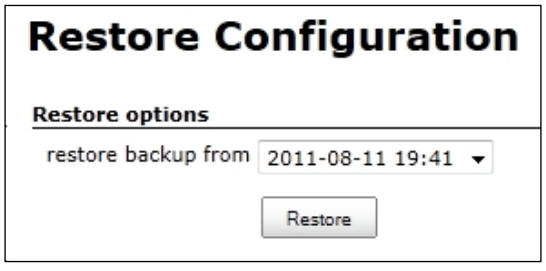


2. Click on the link to **Settings** by the **Toolset** icon.

3. Add the details as shown in the next screenshot. Here, `/data/Jenkins/backups` is a placeholder for the directory that you have previously created.



4. Click on **Save**.
5. Click on the **Backup now** icon.
6. From the command line, visit your backup directory. You should now see an extra sub-directory named `FULL-{timestamp}`, where `{timestamp}` is the time to the second that the full backup was created.
7. Click on the **Restore** icon.
8. A select box restore backup form will be shown with the dates of the backups. Select the backup just created. Click on the **Restore** button.



9. To guarantee the consistency, restart the Jenkins server.

How it works...

The backup scheduler uses the `cron` notation (`http://en.wikipedia.org/wiki/Cron`). `1 0 * * 7` means every seventh day of the week at `00:01 AM`. `1 1 * * *` implies that differential backup occurs once per day at `1.01 A.M`. Every seventh day, the previous differentials are deleted.

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 through the option `Clean up differential backups`, 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. Under these conditions, the build results should not take much space. However, in case of misconfiguration, you should monitor the archive for disc 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.

Restore is a question of returning to the restore menu and choosing the date. I can't 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; that's a false economy.

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 log file `/var/log/jenkins/jenkins.log`, for `*NIX` distributions and for log-level `SEVERE`. For example:

```
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 `$content` value with your Jenkins workspace location, and the `$exclude_pattern` with the pattern you wish to test. The script will print a list of the excluded files.

```
#!/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";
   }
}
```



Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. 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 at:

`http://perldoc.perl.org/File/Find.html`.

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

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

- *Reporting about the overall disc usage*
- *A Job to warn about the disc usage violations*

Modifying Jenkins configuration from the command line

You may be wondering about the XML files at the top level of the Jenkins workspace. These are configuration files. `config.xml` is the main one, dealing 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` sub-directory underneath the workspace. Each individual Job configuration is contained in a sub-directory with the same name as the Job. The Job-specific configuration is then stored in `config.xml` within the sub-directory. There is a similar situation for the user's directory with one sub-directory per user, with the user information stored in its own `config.xml` file.

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 sacrificial machine and then push the configuration files to all the other machines. You can then restart the servers with the **Command-Line Interface** (**CLI**).

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

Getting ready

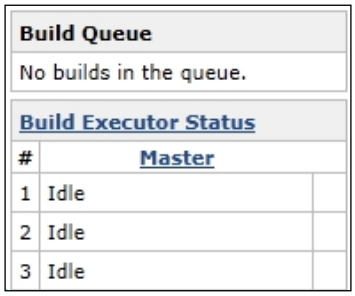
You will need a fresh install of Jenkins with security enabled.

How to do it...

1. In the top-level directory of Jenkins, look for the file named `config.xml`. Go to the line that has the `<numExecutor>` tag, and edit it by changing the number from `2` to `3`, as follows:

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

2. Restart the server. You will see that the number of executors has increased from a default of `2` to `3`.



3. Look for the file named `thinBackup.xml`. You will not find it unless you have installed the `thinBackup` plugin.
4. Replay the recipe *Back up and Restoring*, and look again. You will now find the following XML file.

```
<?xml version='1.0' encoding='UTF-8'?>
<org.jvnet.hudson.plugins.thinbackup.ThinBackupPluginImpl>
  <fullBackupSchedule>1 0 * * 7</fullBackupSchedule>
  <diffBackupSchedule>1 1 * * *</diffBackupSchedule>
  <backupPath>/home/aberg/backups</backupPath>
  <nrMaxStoredFull>61</nrMaxStoredFull>
  <cleanupDiff>true</cleanupDiff>
  <moveOldBackupsToZipFile>true</moveOldBackupsToZipFile>
  <backupBuildResults>true</backupBuildResults>
  <excludedFilesRegex></excludedFilesRegex>
</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. `config.xml` 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 underlying plugin classes. The GUI itself is created from XML through the **Jelly framework** (`http://commons.apache.org/jelly/`).

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

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` in `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 `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 be useful methods exposed for administrative Groovy scripts.

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

The effects of adding garbage

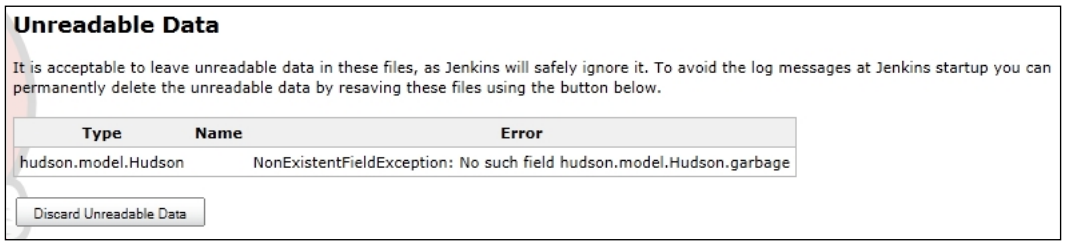
Jenkins is great at recognizing rubbish configuration as long as it is recognizable as a valid XML fragment. For example, add the following line of code to `config.xml`:

```
<garbage>yeuch bllllllaaaaaa</garbage>
```

When you reload the configuration, you will see the following error at the top of the manage Jenkins screen:



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



From this, you can notice that Jenkins is developer-friendly when reading corrupted configuration that it does not understand.

See also

- *Using a sacrificial Jenkins instance*
- *Participating in the community - Maven archetypes and plugins*, *Chapter 7*, *Exploring Plugins*

Reporting overall disc usage

Organizations have their own way of dealing with increasing disc 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 disc 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 disc usage plugin is the strongest in combination with an early warning system that notifies you when soft or hard disc limits are reached. The recipe: *A Job to warn of disc usage violations through log parsing* details a solution. Both the recipes 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

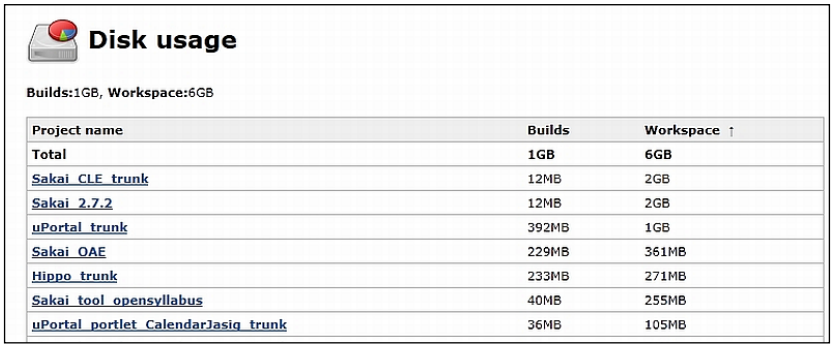
You will need to install the disc usage plugin.

How to do it...

1. Press the **Disk usage** link under the **Manage Jenkins** page.



2. After clicking on the **Disk Usage** link, Jenkins displays a page with each project and the builds and **Workspace** disc usage summary. Click on the top of the table to sort the workspace by file usage.



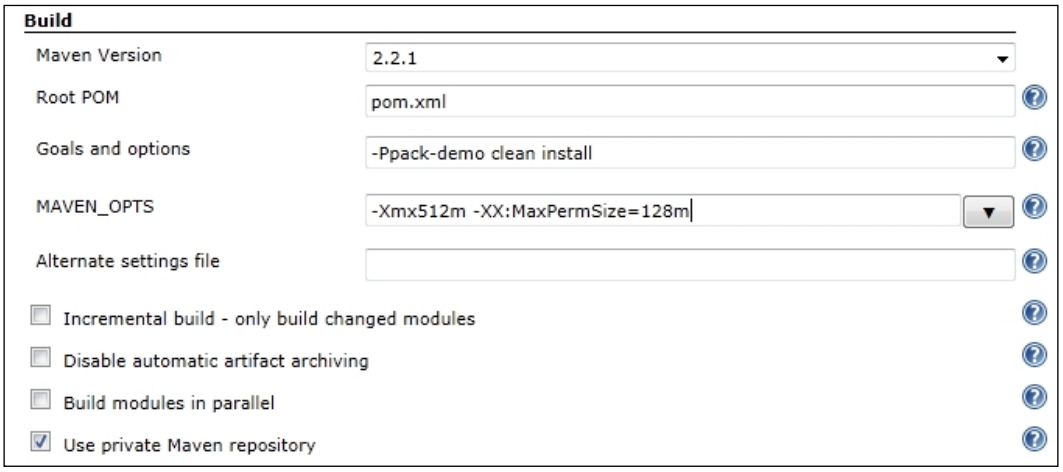
Project name	Builds	Workspace ↑
Total	1GB	6GB
Sakai_CLE_trunk	12MB	2GB
Sakai_2.7.2	12MB	2GB
uPortal_trunk	392MB	1GB
Sakai_OAE	229MB	361MB
Hippo_trunk	233MB	271MB
Sakai_tool_opensyllabus	40MB	255MB
uPortal_portlet_CalendarJasig_trunk	36MB	105MB

How it works...

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

It is easy for you to forget a tick box 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. You will need to hit the advanced button first, before reviewing. On a Friday afternoon, this might be one step too far.

Advanced options include artifact retention choices, which you will need to correctly configure to avoid overwhelming disc usage. In the previous example, the workspace for the **Sakai CLE** is **2GB**. The size is to do with the Job having its own local Maven repository, as defined by the advanced option Use a private Maven repository. The option is easy for you to miss. In this case, there is nothing to be done, as trunk pulls in snapshot JARs, which might cause instability for other projects.



The simple act of being able to sort disc usage points the offending Jobs out to you, ready for further inspection of their advanced configuration.

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 disc. They have the following advantages:

- **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, and 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 the access.
- **GUI interface for ease of artifact management**: All the three repository managers have intuitive GUIs, making your management tasks as easy as possible.

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

For further reading, see: `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**, and then the **Advanced** checkbox, define the **Max #** of builds to keep with the 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 disc consuming artifacts.

See also

- *Backing up and restoring*

Deliberately failing builds through log parsing

Scenario: You have been asked to clean up the code removing depreciated Java methods across all the source contained under a Jenkins Jobs; that is a lot of code. If you miss some residue 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.

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 file named `depreciated.rule` to the `log_rules` directory with one line:
   ```
   error /DEPRECATED/
   ```
3. Create a Job with a source code that gives deprecated warnings on compilation. In the following example, you are using the **Roster** tool from the `Sakai` project:
   - **Jobname**: `Sakai_Roster2_Test`
   - Check **Maven 2/3 Project**
   - **Sourcecode Management**: `Subversion`
   - **Repository URL**: `https://source.sakaiproject.org/contrib/roster2/trunk`
   - Build
   - **Maven Version**: `2.2.1` (or whatever your label is for this version)
   - **Goals and options**: `clean install`
4. Run the build. It should not fail.

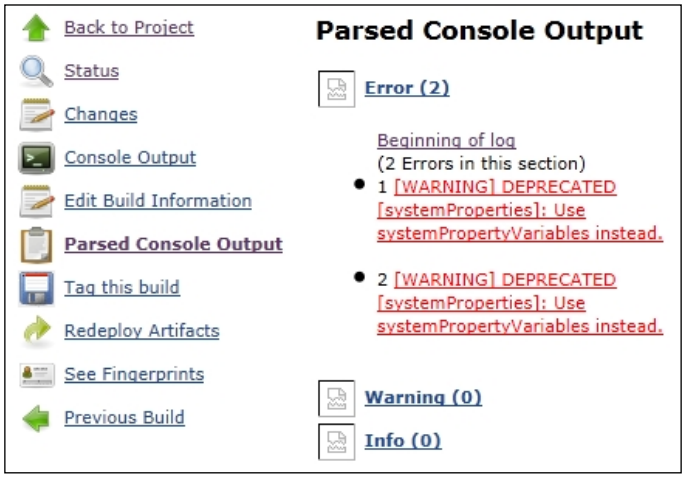
5. As shown in the next screenshot, visit the **Manage configuration** page for Jenkins, and to the **Console Output** section, add a description and location of the parsing rules file that was mentioned in *step 2*.



6. Check the **Console output (build log) parsing** box in the **Post-build Actions** section of your Job.
7. Check the **Mark build Failed on Error** checkbox.
8. Select **Kill Depreciated from** the **Select Parsing Rules** list box.



9. Build the Job; it should now fail.
10. Click on the **Parsed Console Output** link in the left-hand menu. You will now be able to see the parsed errors.



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 /DEPRECATED/`.

If the pattern `DEPRECATED` (a case-sensitive test) is found in the console output, then the plugin considers this as 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 pulled in from Sakai (`http://www.sakaiproject.org`) contains deprecated method and triggers the pattern.

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 a Job between the rule files previously created.



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

There's more...

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

- **MD5 check sums**: 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 warning 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

- *A Job to warn about the disc usage violations through log parsing*

A Job to warn about the disc usage violations through log parsing

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

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



Using `Perl` is typical for a Jenkins Job, as Jenkins plays well and adapts to most environments. You can expect `Perl`, `Bash`, `Ant`, `Maven`, and a full range of scripts and binding code to be used in the battle to get the work done.

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 to the `log_rules` directory named `disc.rule` with the following two lines:

```
error /HARD_LIMIT/
warn /SOFT_LIMIT/
```

2. Visit the **Manage configuration** page for Jenkins, and add a description as `DISC_USAGE` to the **Console Output** section. Point to the location of the **Parsing Rules** file.
3. Add the following `Perl` script to a location of choice named `disc_limits.pl`, 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);
}
```

```
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 the build `Step / Execute Shell`. For the command, add `perl disc_limits.pl 9000000 2000000`.
7. Feel free to change the hard and soft limits (`9000000 2000000`).
8. Check the **Console output (build log) parsing** in the **Post-build Actions** section.
9. Check the **Mark build Unstable on Warning** checkbox.
10. Check the **Mark build Failed on Error** checkbox.
11. Select **DISC_USAGE** from the **Select Parsing Rules** combo box.



12. Run the build a number of times.

13. Under build history on the left-hand, select the **trend** link. You can now view trend reports and see a timeline of success and failure.



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 disc 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 the administrators time 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 method `size_summary` 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 rule warn `/SOFT_LIMIT/`, and then signal a Job warn.

There's more...

Welcome to the wonderful world of Jenkins. You can now utilize all of the installed features at your disposal. The Job can be scheduled, e-mails can be sent out on failure. You can also tweet, add entries to Google calendar, trigger extra events, for example disc-cleaning builds, and so on. You are mostly limited by your imagination and 21st century technologies.

See also

- *Backing up and restoring*

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 the 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 5*, *Communicating Through Jenkins* is dedicated to the 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 toting.

This recipe is one of the more pleasant ways for you to be reached. You will pull in the 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 Firefox 5 or later installed on your computer and an account on at least one Jenkins instance, with a history of running Jobs.



A plug for the developers

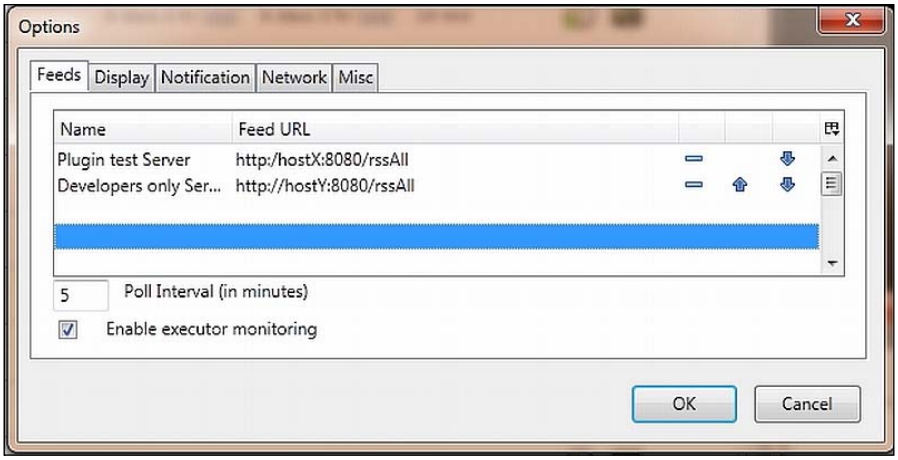
If you like the add-on and want more features in the future, then it is enlightened in the self-interest to donate a few bucks at the add-on author's website.

How to do it...

1. Select the **Firefox** tab at the top-left hand side of the browser.
2. In the **Search** box (top-right) with the title **Search all add-ons**, search for Jenkins.
3. Click on the **Install** button for the Jenkins **Build** monitor.
4. Restart Firefox.
5. Select the **Firefox** tab at the top left-hand side of the browser.
6. Enable the **Add-On Bar** by selecting **Options**, and then **Add-On Bar**. Now, at the bottom right-hand side of Firefox, you will see a small Jenkins icon.



7. Right-click on the icon.
8. Select the 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 e.g.: http://localhost:8080/rssAll`.



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



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

There's more...

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.

See also

Chapter 5, Communicating Through Jenkins:

- *Visualizing schedules – the Google calendar*
- *Shouting at developers through Twitter*

Monitoring through 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 the important information. You can view the evolution charts ranging from a day or weeks to months of the main quantities, such as the CPU or the memory. **Evolution charts** are very good at pinpointing the scheduled Jobs that are resource-hungry. JavaMelody allows you to keep a pulse on the incremental degradation of resources. It eases the writing of reports by exporting statistics in a PDF format. Containing over 25 years of human effort, JavaMelody is feature-rich.

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



Community partnership

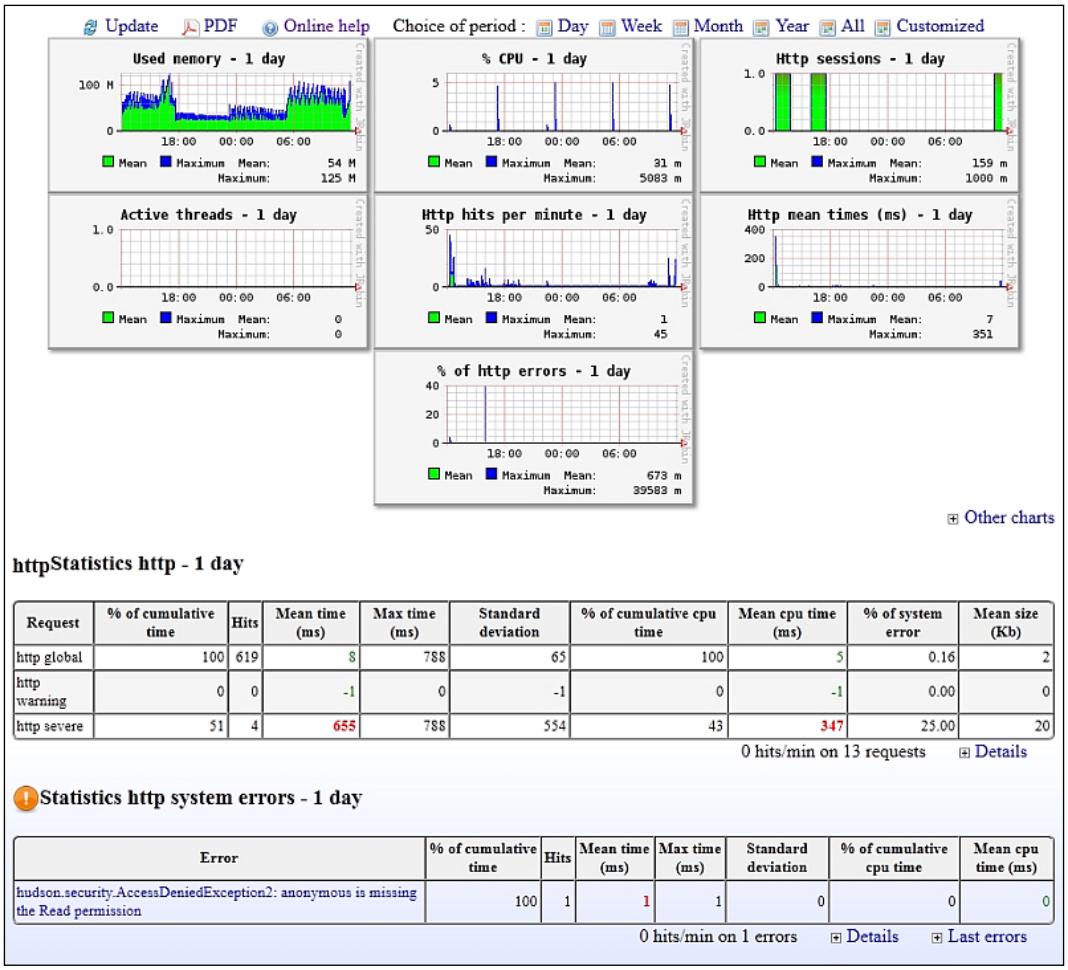
If you find this plugin useful, consider contributing back to either the plugin or the core JavaMelody project.

Getting ready

You will need to have installed the JavaMelody plugin.

How to do it...

1. Click on the **Monitoring Hudson/Jenkins** master link on the **Manage Jenkins** page. You will now see the detailed monitoring information.



2. Read the online help at the URL `http://host:port/monitoring?resource=help/help.html`, where the host and port point to your server.
3. Review the monitoring of the node processes directly, by visiting `http://host:port/monitoring/nodes`.

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.

There's more...

Monitoring is the foundation for comprehensive testing and troubleshooting. This section explores the relationship 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 the 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 you get to full memory, the busier the 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 that 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 link **Execute the garbage collector**. If it is not a memory leak, then 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, and cause large response times for web browser requests. This can be seen through the `mean` and `max` times under the Statistics labeled `http - 1 day`.

Troubleshooting with JavaMelody - painful Jobs

You should consider the following points:

- **Offload work**: For a stable infrastructure, offload as much work as possible from the master instance. If you have scheduled the 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 show 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 8GB is cheap.



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

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

Chapter 7, *Testing Remotely*:

- *Running a script to obtain the monitoring information*

Keeping a track of the 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. 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.

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

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 link on the left-hand side of **Remote Script catalogs**.
3. Click on the icon of the floppy disk for `getThreadDump`. If the script is not available, then choose another script of your choice.
4. You have now returned to the **Scriptler** main page. You will see three icons. Choose the furthest right to execute the script.



5. You are now in the **Run a script** page. Select a node and 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.

6. To write a new Groovy script or to 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 meta information about the files that you have created in the `scriptler.xml` file. A second file, `scriptlerweb-catalog.xml`, mentions the list of online files that you can download.

All the local scripts are contained in the sub-directory 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 a wider community.

See also

- *Scripting the Jenkins command-line interface*
- *Global modifications of Jobs with Groovy*
- *Scripting the global build reports*

Scripting the Jenkins command-line interface

The Jenkins **Command-Line Interface** (**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 the Jenkins instances on and offline, 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 and run a Groovy script that looks for files greater than a certain size, and log off. The script represents a typical maintenance task. You can imagine chaining a second script to the first, to remove the large files found.



At the time of writing this chapter, the interactive Groovy shell was not working from the CLI. This is mentioned in the bug report: `http://issues.hudson-ci.org/browse/HUDSON-5930`.

Getting ready

Download the CLI JAR file from http://host/jnlpJars/jenkins-cli.jar.

Add the following script to a directory under the control of Jenkins and call it `large_files.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 next command from a terminal, replacing `http://host` with the real address of your server, for example `http://localhost:8080`.

   `java -jar _jenkins-cli.jar -s http://host  login --username username`

2. Input your password.
3. Look at the online help:

   `java -jar _jenkins-cli.jar -s http://host   help`

4. Run the Groovy script. The command-line output will now mention all the oversize files.

   `java -jar _jenkins-cli.jar -s http://host groovy look.groovy`

5. Log out.

   `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 a large number of Jenkins instances at the same time. This recipe performs a lot of drudgework. In this case, it reviews X thousand files for oversized artifacts, saving you time that you can better spend on more interesting tasks.

Before performing any commands, you need to first authenticate through the `login` command.

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

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

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



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, there were some issues with the key approach. See `https://issues.jenkins-ci.org/browse/JENKINS-10647`. Feel free to resort back to the method used in this recipe, which 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 therefore important that you occasionally check the in-built help.

See also

- *Global modifications of Jobs with Groovy*
- *Scripting global build reports*

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 increase, you will notice that 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

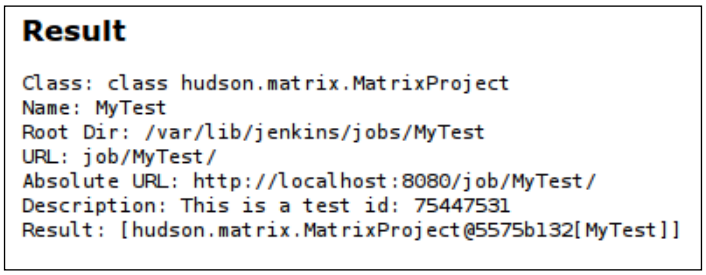
Log in to Jenkins with an administrative account.

How to do it...

1. Create an empty Job named `MyTest`.
2. Within the Manage Jenkins page, click on the **Script console** link.
3. Cut and paste the following script into the text area input.

```
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)}")
   }
}
```

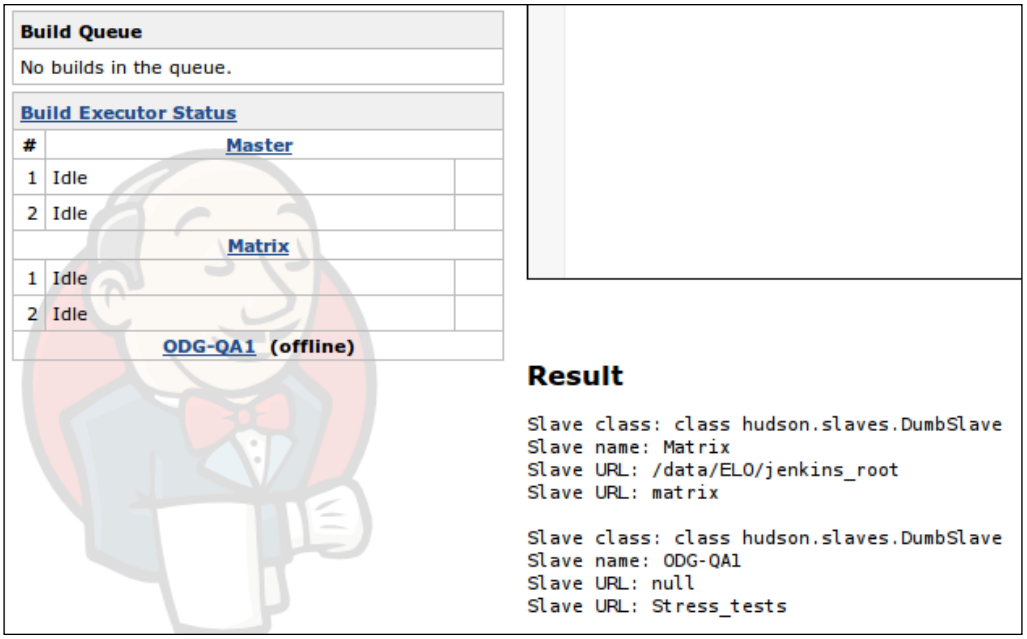
4. Click on the **run** button. The results should be similar to the following screenshot:



5. Run the script a second time, and you will notice that the random number in the description has now changed.
6. Copy and run the following script:

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

7. 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:



How it works...

Jenkins has a rich framework, which 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`). The second script iterates through instances of slave objects (`http://javadoc.jenkins-ci.org/hudson/slaves/SlaveComputer.htm`).

There's more...

For the hardcore Java developer: If you don't know how to do a programmatic task, then an excellent source of example code is the Jenkins subversion directories for plugins (`https://svn.jenkins-ci.org/trunk/hudson/plugins/`).



If you are interested in donating your own plugin, review the information at: `https://wiki.jenkins-ci.org/display/JENKINS/Hosting+Plugins`

See also

- *Scripting the Jenkins command-line interface*
- *Scripting the global build reports*

Signaling the need to archive

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.

This recipe details a script that checks for the last successful run of any Job, and if the year is different to the current year, then a warning is set at the beginning of the Jobs description. Thus, hinting to you it is 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.

Getting ready

Log in to Jenkins with an administrative account.

How to do it...

Within the Manage Jenkins page, click on the **Script console** link, and run the following script:

```
import hudson.model.Run;
import java.text.DateFormat;

def warning='<font color=\'red\'>[ARCHIVE]</font> '
def now=new Date()

for (job in hudson.model.Hudson.instance.items) {
    println "\nName: ${job.name}"
    Run lastSuccessfulBuild = job.getLastSuccessfulBuild()
    if (lastSuccessfulBuild != null) {
      def time = lastSuccessfulBuild.getTimestamp().getTime()
      if (now.year.equals(time.year)){
         println("Project has same year as build");
      }else {
        if (job.description.startsWith(warning)){
            println("Description has already been changed");
        }else{
            job.setDescription("${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.



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 through the `for` statement.

Jenkins has a class to store information about the running of builds. The runtime information is retrieved through `job.getLastSuccessfulBuild()`, and is stored in the `lastSuccessfulBuild` instance. If no successful build has occurred, then `lastSuccessfulBuild` is set to `null`, otherwise it has the runtime information.

The time of the last successful build is retrieved, and then stored in the `time` instance through `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.



Javadoc

You will find the Job API mentioned at `http://javadoc.jenkins-ci.org/hudson/model/Job.html` and the Run information at `http://javadoc.jenkins-ci.org/hudson/model/Run.html`.

There's more...

Before writing your own code, you should review what already exists. With 300 plugins, 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 re-used from the `lastsuccessversioncolumn` plugin (`https://github.com/jenkinsci/lastsuccessversioncolumn-plugin/blob/master/src/main/java/hudson/plugins/lastsuccessversioncolumn/LastSuccessVersionColumn.java`).



If you find any defects while reviewing the plugin code base, please contribute to the community through patches and bug reports.

See also

- *Scripting the Jenkins command-line interface*
- *Global modifications of Jobs with Groovy*

2
Enhancing Security

In this chapter, we will cover:

- Testing for OWASP's top ten security issues
- Finding 500 errors and XSS attacks in Jenkins through fuzzing
- Improving security via small configuration changes
- Looking at the Jenkins user through Groovy
- Working with the Audit Trail plugin
- Installing OpenLDAP with a test user and group
- Using the Script Realm authentication for provisioning
- Reviewing Project-based Matrix tactics via a custom group script
- Administering OpenLDAP
- Configuring the LDAP plugin
- Installing a CAS server
- Enabling SSO in Jenkins

Introduction

In this chapter, we will discuss the security of Jenkins, taking into account that it can live in a rich variety of infrastructures.

The only perfectly secure system is the system that does not exist. For real services, you will need to pay attention to the different surfaces available to attack. The primary surfaces of Jenkins are its web-based graphical user interface and its trust relationships with its slave nodes.

Online services need vigorous attention to their security surface. For Jenkins, there are two main reasons why:

- Jenkins has the ability to talk to a wide range of infrastructure, either through its plugins or the master slave topology
- The rate of code change around the plugins is high and open to accidental inclusion of security-related defects

A counterbalance is that the developers using the Jenkins frameworks apply well-proven technologies, such as **Xstream** (`http://xstream.codehaus.org/`) for configuration persistence and **Jelly** (`http://commons.apache.org/jelly/`) for rendering the GUI. This use of well-known frameworks minimizes the number of lines of supporting code, and the code that is used is well tested, limiting the scope of vulnerabilities.

Another positive is that Jenkins code is freely available for review, and the core community keeps a vigilant eye. It is unlikely that anyone contributing to a code would deliberately add defects or unexpected license headers. However, trust but verify.

The first half of this chapter is devoted to the Jenkins environment. In the second half, you will see how Jenkins fits into the wider infrastructure.

LDAP is widely available and the de facto standard for Enterprise directory services. We shall use LDAP for Jenkins authentication and authorization, and later **Single Sign On** (**SSO**) by JASIG's **Central Authentication Server** (**CAS**, `http://www.jasig.org/cas`). CAS allows you to sign on once and then go to other services without logging in again. This is useful for when you want to link from Jenkins to other password-protected services such as an organization's internal wiki or code browser. Just as importantly, CAS can connect behind the scenes to multiple types of authentication providers, such as LDAP, databases, textfiles, and an increasing number of other methods. This indirectly allows Jenkins to use many logon protocols on top of the ones its plugins already provide.



Security advisories

There is an e-mail list and RSS feed for Jenkins-related security advisories. You can find the link to the advisory feeds at `https://wiki.jenkins-ci.org/display/JENKINS/Security+Advisories`.

Testing for OWASP's top ten security issues

This recipe details the automatic testing of Jenkins for well-known security issues with **w3af**, a penetration testing tool from the **Open Web Application Security Project** (**OWASP**, `http://w3af.sourceforge.net`). The purpose of OWASP is to make application security visible. The OWASP top ten list of insecurities includes:

- **A2-Cross Site Scripting (XSS)**: An XSS attack can occur when an application returns an unescaped input to a client's browser. The Jenkins administrator can do this by default, through the Job description.
- **A6-Security Misconfiguration**: A Jenkins plugin gives you the power to write custom authentication scripts. It is easy to get the scripts wrong by misconfiguration.
- **A7-Insecure Cryptographic Storage**: There are over 300 plugins for Jenkins, each storing their configuration in separate XML files. It is quite possible that there is a rare mistake with the storage of passwords in plain text. You will need to double check.
- **A9-Insufficient Transport Layer Protection**: Jenkins runs by default over HTTP. It can be a hassle and involves extra costs to obtain a trusted certificate. You might be tempted to not implement TLS, leaving your packets open.

Jenkins has a large set of plugins written by a motivated, diffuse, and hardworking community. It is possible, due to the large churn of code, that security defects are inadvertently added. Examples include leaving passwords in plain text in configuration files or using unsafe rendering that does not remove suspicious JavaScript. You can find the first type of defect by reviewing the configuration files manually. The second type is accessible to a wider audience, and thus is more readily crackable.

You can attack the new plugins by hand. There are helpful cheat sheets available on the Internet (`http://ha.ckers.org/xss.html`). The effort is tedious; automated tests can cover more ground and be scheduled as part of a Jenkins Job.



OWASP Store front

Each year, OWASP publishes a list of the top ten most common security attack vectors for web applications. They publish this document and a wide range of books through `http://lulu.com`. At `lulu.com`, you have free access to the PDF versions of OWASP's documents or you can buy cheap on-demand printed versions. You can find the official store front at `http://www.lulu.com/spotlight/owasp`.

Getting ready

Penetration tests have the potential to damage a running application. Make sure that you have a backed up copy of your Jenkins workspace; you might have to reinstall. Please also turn off any enabled security within Jenkins; this allows w3af to freely roam the security surface.

Please download the newest version of w3af from **SourceForge** (`http://w3af.sourceforge.net/`), and also download and read the OWASP top 10 list of well-known attacks at `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`.

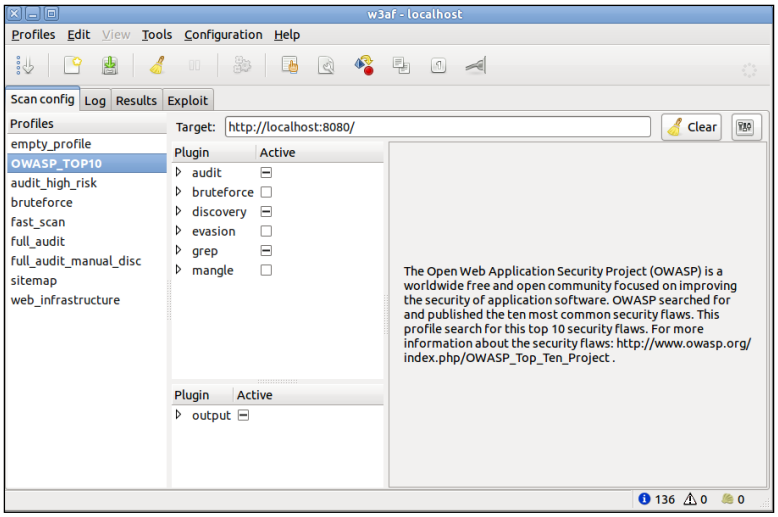
w3af has both a Windows and *NIX installation package; use the OS install of choice.



Warning: The Debian package for w3af is older and more unstable than the SourceForge package for Linux. Therefore, please do not use the apt-get and yum methods of installation, but rather use the downloaded package from SourceForge.

How to do it...

1. Run w3af.
2. Under the **Profiles** tab, select **OWASP_TOP10**.
3. Under the **Target:** address window, fill in `http://localhost:8080/`, changing the hostname to suit your environment.
4. Click on the **Start** button. The penetration tests will now take place and the **Start** button will change to **Stop**. At the end of the scan, the **Stop** button will change to **Clear**.



5. View the attack history by selecting the **Log** tab.
6. Review the results by clicking on the **Results** tab.
7. After the first scan, select **full_audit** under **Profiles**.
8. Click on the **Clear** button.
9. Add `http://localhost:8080/` as **Target:**.
10. Click on the **Start** button.
11. Wait until the scan has finished, and review the results in the Results tab.

How it works...

w3af is written by security professionals. It is a pluggable framework with extensions written for different types of attacks. The profiles define which plugins and their associated configurations you are going to use in the penetration test.

You first attack using the **OWASP_TOP10** profile, and then attack again with a fuller set of plugins.

The results will vary according to your setup. Depending on the plugin, security issues that do not exist are occasionally flagged. You will need to verify by hand any issues mentioned.

At the time of writing, no significant defects were found using this approach. However, the tool pointed out slow links and generated server-side exceptions. This is the sort of information you would want to note in the bug reports.

There's more...

Consistently securing your applications requires experienced attention to detail. Here are a few more things for you to review:

Target practice with Webgoat

The top ten list of security defects can at times seem difficult to understand. If you have some spare time and you like practicing against a deliberately insecure application, you should try **Webgoat** (`https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project`).

Webgoat is well documented, with a hints system and links to video tutorials; it leaves little room for misunderstanding the attacks.

More tools of the trade

w3af is a powerful tool but works better in conjunction with other tools, including:

- **Nikto** (`http://cirt.net/nikto2`): A Perl script that quickly summarizes system details and looks for the most obvious of defects.
- **Skipfish** (`http://code.google.com/p/skipfish/`): A C program that bashes away with many requests over a prolonged period. You can choose from different dictionaries of attacks. This is an excellent poor man's stress test. If your system stays up, you know that it has reached a minimal level of stability.
- **Wapiti** (`http://wapiti.sourceforge.net/`): A Python-based script, it discovers attackable URLs and then cycles through a list of evil parameters.

Jenkins is flexible so that you can call a wide range of tools through scripts running in Jobs, including the security tools mentioned.

See also

- *Finding 500 errors and XSS attacks in Jenkins through fuzzing*
- *Improving security via small configuration changes*

Finding 500 errors and XSS attacks in Jenkins through fuzzing

This recipe describes using a **fuzzer** to find server-side errors and XSS attacks in your Jenkins servers.

A fuzzer goes through a series of URLs, appends different parameters blindly, and checks the response from servers. The inputted parameters are variations of scripting commands such as `<script>alert("random string");</script>`. An attack vector is found if the server's response includes the unescaped version of the script.

Cross Site Scripting attacks are currently one of the more popular forms of attack (`http://en.wikipedia.org/wiki/Cross-site_scripting`). The attack involves injecting script fragments into the client's browser so that the script runs as if it comes from a trusted website. For example, once you have logged in to an application, it is probable that your session ID is stored in a cookie. The injected script might read the value in the cookie and then send the information to another server ready for an attempt at reuse.

A fuzzer discovers the links on the site it is attacking and the form variables that exist within the site's web pages. For the web pages discovered, it repeatedly sends input based on historic attacks and lots of minor variations. If responses are returned with the same random strings sent, then the fuzzer knows it has found an **evil URL**.

Getting ready

Back up your sacrificial Jenkins server and turn off its security. Expect the application to be unstable by the end of the attack.

You will need the Python programming language installed on your computer. To download and install Wapiti, you will need to follow the instructions found at `http://www.ict-romulus.eu/web/wapiti/home`.



If you are attacking your local machine from your local machine, then you can afford to turn off its networking. The attack will stay in the Loopback network driver, and no packets should escape to the Internet.

How to do it...

1. Within the `src` directory of Wapiti, run the following command:

```
python wapiti.py http://localhost:8080 -m
  "-all,xss,exec" -x http://localhost:8080/pluginManager/* -v2
```

2. A number of server-side errors will be reported to the console. You can confirm that the URL is causing an error by using your favorite web browser to visit the URL mentioned. For example, `http://localhost:8080/computer/createItem?name=%2Fe%00&Submit=OK&mode=dummy2`.

3. View the result of your browsing, for example:

```
java.lang.NullPointerException
  at hudson.model.ComputerSet.doCreateItem
  (ComputerSet.java:246)
```

4. Run the following command:

```
python wapiti.py http://localhost:8080 -m
  "-all,xss,permanentxss" -x
  http://localhost:8080/pluginManager/*
```

5. View the results, as follows:

```
[*] Loading modules : mod_crlf, mod_exec,
  mod_file, mod_sql, mod_xss, mod_backup,
  mod_htaccess, mod_blindsql, mod_permanentxss, mod_nikto
[+] Launching module xss
```

```
[+] Launching module permanentxss
  Found permanent XSS in
    http://localhost:8080/?auto_refresh=true
    attacked by
    http://localhost:8080/view/All/editDescription
    with fields description=
    <script>alert('5npc2bivvu')</script>&Submit=diwan3xigb
    injected from
    http://localhost:8080/view/All/submitDescription
```

How it works...

Wapiti loads in different modules. By default, all modules are used. You will have to be selective for version 2.2.1 on Ubuntu Linux, as this causes Wapiti to crash or time out.

To load Wapiti in specific modules, use the `-m` option.

`-m "-all,xss,exec"` tells Wapiti to ignore all modules, except the `xss` and `exec` modules.

The `exec` module is very good at finding 500 errors in Jenkins. This is mostly due to unexpected input that Jenkins does not handle well. This is purely a cosmetic set of issues. However, if you start to see errors associated with resources such as files or database services, you should give the issues higher priority and send in bug reports.

The `-x` option specifies which URLs to ignore. In this case, we don't want to create work for the plugin manager. If we do, it will then generate a lot of requests to an innocent external service.

`-v2` sets the verbosity of logging up to its highest so that you can see all the attacks.

In the second run of Wapiti, you also used the `permanentxss` module, which finds a bona fide XSS attack through the editing of descriptions. You will be eliminating this issue in the next recipe, *Improving security via small configuration changes*.

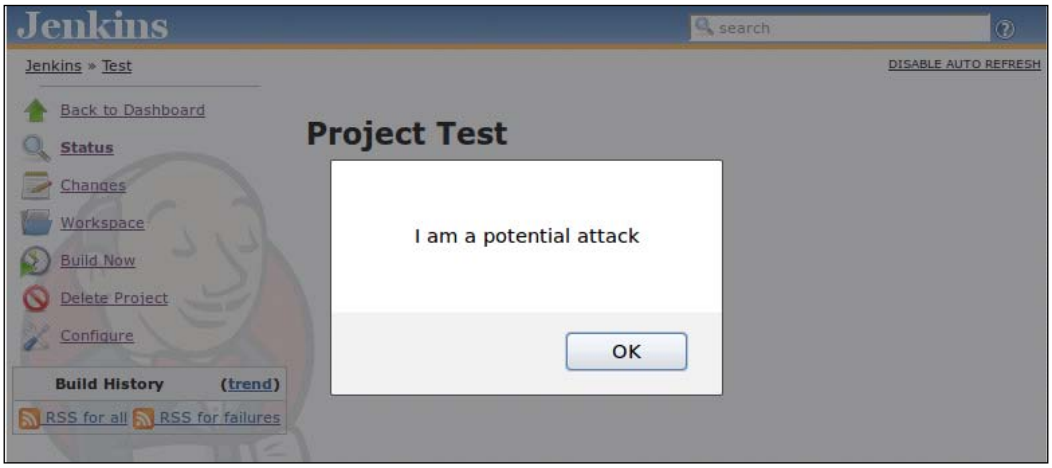


Poor man's quality assurance

Fuzzers are good at covering a large portion of an application's URL space, triggering errors that would be costly, in terms of time, to search out. Consider automating through a Jenkins job as a part of a project's QA process.

There's more...

To confirm that the attack found is possible, add `<script>alert('I am a potential attack');</script>` to the description of a Job. When you next visit the Job, you will see an alert box pop up. This implies that you have managed to inject JavaScript into your browser.



See also

- *Testing for OWASP's top ten security issues*
- *Improving security via small configuration changes*

Improving security via small configuration changes

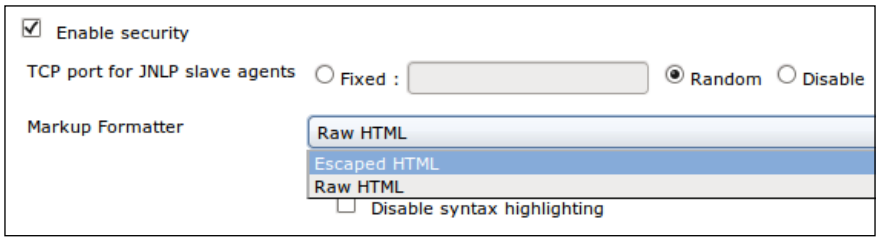
This recipe describes modest configuration changes that strengthen the default security settings of Jenkins. The reconfiguration includes removing the ability to add JavaScript or HTML tags to descriptions, masking passwords in console output, and adding a one-time random number, which makes it more difficult for a form input to be forged. The combination of tweaks strengthens the security of Jenkins considerably.

Getting ready

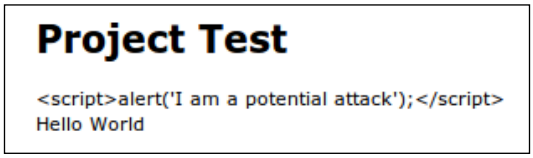
You will need to install the **Escaped Markup** plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Escaped+Markup+Plugin`) and the **Mask Passwords** plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Mask+Passwords+Plugin`).

How to do it...

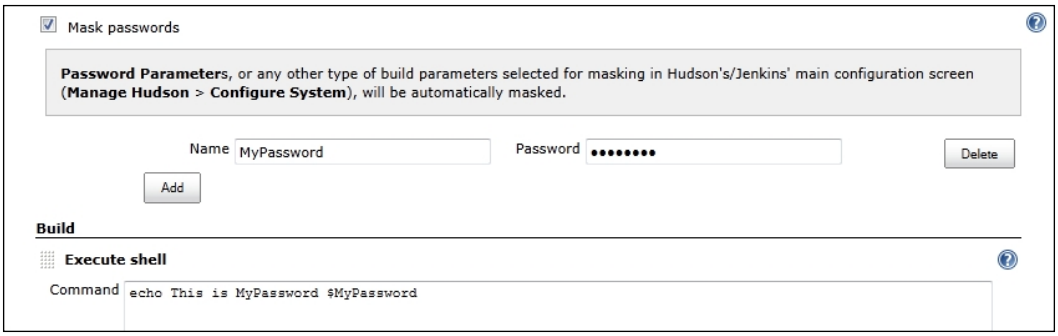
1. Add `<script>alert('I am a potential attack');</script>` to the description of a job. When you next visit the job, you will see an alert box pop up.
2. Visit the Jenkins configuration page (`http://localhost:8080/configure`).
3. Tick the **Enable security** box.
4. Select **Escaped HTML**, under **Markup Formatter**.



5. Press the **Save** button.
6. To confirm that the attack has been removed, visit the job whose description you have previously modified in step 1. You will now see escaped HTML.



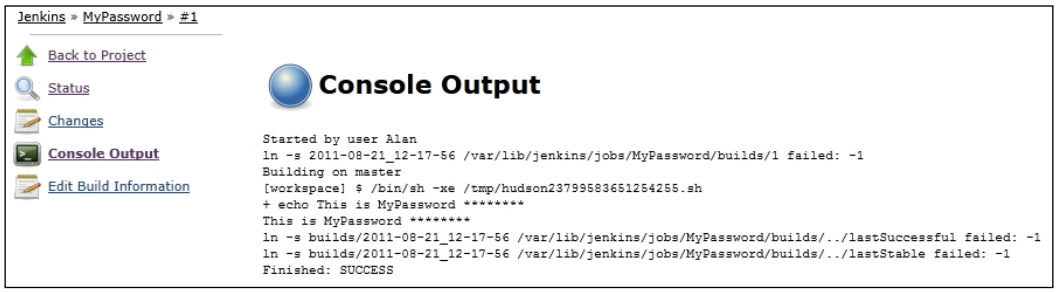
7. Create a job.
8. Click on the **Mask passwords** tickbox and add the following variables:
   - **Name**: `MyPassword`
   - **Password**: `changeme`



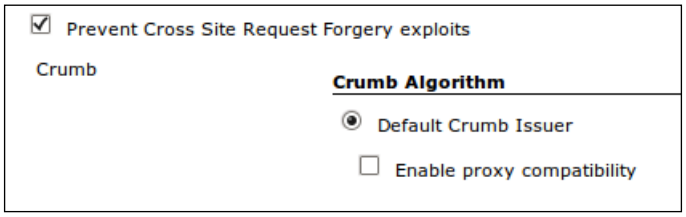
9. Add an **Execute shell** build with the following **Command**:

```
echo This is MyPassword $MyPassword
```

10. Run the job.
11. Review the **Console Output**.



12. Return to the Jenkins configuration page and click on **Prevent Cross Site Request Forgery exploits**, making sure the **Default Crumb Issuer** is selected.



How it works...

The escaped HTML plugin takes its input from your Job description and escapes any tags by parsing the text through a Jenkins utility class. This action not only removes the risk of running unsolicited JavaScript, but also removes some flexibility for you as the administrator of the Job. You can no longer add formatting tags, such as `font`.

The Mask Passwords plugin removes the password from the screen or the console, replacing each character of the password with the letter "x", thus avoiding accidental reading. You should also always keep this plugin turned on, unless you find undocumented side effects or need to debug a Job.

Cross Site Request Forgery (`http://en.wikipedia.org/wiki/Cross-site_request_forgery`) occurs, for example, if you accidentally visit a third-party location. A script at that location then tries to make your browser perform an action (such as delete a Job) by making your web browser visit a known URL within Jenkins. Jenkins, thinking that the browser is doing your bidding, then complies with the request. Once the **nonce** feature is turned on, Jenkins avoids CSRF by generating a random one-time number called a nonce that is returned as part of the request. The number is not easily known and is also invalidated after a short window of time, limiting the risk of **replay attacks**.

There's more...

Jenkins is a pleasure to use. This is because Jenkins makes it easy to get the work done and can talk through plugins with a multitude of infrastructure. This implies that in many organizations, the number of administrators increases rapidly as the service organically grows. Think about turning on the HTML escaping early, before the group of administrators get used to the flexibility of being able to add arbitrary tags.

Consider occasionally replaying the recipe *Finding 500 errors and XSS attacks in Jenkins through fuzzing* to verify the removal of this source of potential XSS attacks.

See also

- *Testing for OWASP's top ten security issues*
- *Finding 500 errors and XSS attacks in Jenkins through fuzzing*

Looking at the Jenkins user through Groovy

Groovy scripts run as the user `jenkins`. This recipe highlights the power of, and danger to, the Jenkins application.

Getting ready

Log in to your sacrificial Jenkins instance as an administrator.

How to do it...

1. Run the following script from the **Script Console** (`http://localhost:8080/script`):

```
def printFile(location) {
  pub = new File(location)
  if (pub.exists()){
```

```
    println "Location ${location}"
    pub.eachLine{line-> println line}
}
else{
  println "${location} does not exist"
}

printFile("/etc/passwd")
printFile("/var/lib/jenkins/.ssh/id_rsa")
printFile("C:/Windows/System32/drivers/etc/hosts")
```

2. Review the output.
3. For a typical *NIX system, it will be similar to this:

```
PRIVATE KEY - Jenkins
-----BEGIN RSA PRIVATE KEY-----
MIIEpQIBAAKCAQEAwHV36wytKHYeZAaRZdgON5Bg80vurstlTLmDtMuYNJN8mU8O
Some Randomness
-----END RSA PRIVATE KEY-----
Users on System
root:x:0:0:root:/root:/bin/bash
sys:x:3:3:sys:/dev:/bin/sh
mysql:x:114:127:MySQL Server,,,:/var/lib/mysql:/bin/false
```

4. And for a Windows system, it will be:

```
/etc/passwd does not exist
/var/lib/jenkins/.ssh/id_rsa does not exist
Location C:/Windows/System32/drivers/etc/hosts
# Copyright (c) 1993-2006 Microsoft Corp.
#
# This is a sample HOSTS file used by Microsoft TCP/IP for
Windows.
```

How it works...

The script you have run is not as benign as it first seems. Groovy scripts can do anything the `jenkins` user has the power to do. A method is defined; it reads in a file. The file's location is passed in as a string input. The script then prints out the content. If the file does not exist, that is also mentioned. Three locations are tested. It is trivial for you to add a more detailed set of locations.

The existence of files clearly defines the type of OS being used and the structure of the disc partitioning.

The `/etc/passwd` file typically does not contain passwords. The passwords are hidden in a `shadow password` file, safely out of reach. However, the usernames and whether the username has a real login account (not `/bin/false`) suggest accounts to try and crack using **dictionary attacks**.

You can save the configuration effort if you generate a private and public key for Jenkins. This allows a script to run with a user's permission, without needing a password logon. It is typically used by Jenkins to control its slave nodes. Retrieving the keys through a Groovy script represents further dangers to the wider infrastructure.

If any plugin stores passwords, in plain or decipherable text, then you can capture and parse the plugin's XML configuration files.

Not only can you read files but also change permissions and write over binaries, making the attack harder to find and more aggressive.

There's more...

The best approach to limiting risk is to limit the number of logon accounts that have the power to run Groovy scripts in the **Script console** and to periodically review the audit log.

Limiting administrator accounts is made easier by using a matrix-based strategy, where you can decide the rights of each user or group. A refinement on this is a Project-based matrix strategy, where you can choose permissions per job. However, the Project-based matrix strategy costs you considerably more administration.



Warning: Since version 1.430 of Jenkins, there are extra permissions exposed to the matrix-based security strategy, to decide which group or user can run Groovy scripts. Expect more permission additions over time.

See also

- *Working with the Audit Trail plugin*
- *Reviewing Project-based Matrix tactics via a custom group script*

Working with the Audit Trail plugin

Jobs can fail. It speeds up debugging if you can see who the last person running the job was and what their changes were. This recipe ensures that you have auditing enabled and that a set of audit logs are created that contain a substantial history of events rather than the meager log size defined by default.

Getting ready

Install the Audit Trail plugin from the following location:

`https://wiki.jenkins-ci.org/display/JENKINS/Audit+Trail+Plugin`

How to do it...

1. Visit the Jenkins configuration screen (`http://localhost:8080/configure`).
2. Modify the default settings for the audit trail, to allow for a long observation. Change **Log File Size MB** to `128` and **Log File Count** to `40`.



How it works...

The audit plugin creates a log recorder named Audit Trail (`https://wiki.jenkins-ci.org/display/JENKINS/Logger+Configuration`). You can visit the log recorders pager under `http://localhost:8080/log/?`. The output from the log recorder is filtered via the URL patterns to log, as seen in the Jenkins configuration screen. You will find that the logfile format is more readable than most, with a date/time stamp at the beginning, a description of what is happening in the middle of the log, and the user who acted, at the end. For example:

```
Jul 18, 2011 3:18:51 PM job/Fulltests_1/ #3 Started by user Alan
Jul 18, 2011 3:19:22 PM /job/Fulltests_1/configSubmit by Alan
```

It is now clear who has done what and when.

There's more...

Here are a couple more things you should consider:

A complementary plugin—JobConfigHistory

A complementary plugin that keeps track of configuration changes and displays the information within the job itself is called the **JobConfigHistory** plugin (`https://wiki.jenkins-ci.org/display/JENKINS/JobConfigHistory+Plugin`). The advantage of this plugin is that you get to see who has made those crucial changes. The disadvantage is that it adds an icon to a potentially full GUI, leaving less room for other features.

Missing Audit Logs

For a security officer, it helps to be mildly paranoid. If your audit logs suddenly go missing, it may well be a sign that a cracker wishes to cover their trail. This is also true if one file goes missing, or there is a gap in time of the audit. Even if this is caused by issues with configuration or a damaged file system, you should investigate. Missing logs should trigger a wider review of the server in question. At the very least, the audit plugin(s) is not behaving as expected.

Consider adding a small reporting script for these highly valued logs. For example, consider modifying the recipe in *Chapter 3*, *Building Software*, to parse the logfile and make metrics that are then displayed graphically. This enables you to view, over time, the ebb and flow of your team's work. Of course, the data can be faked, but that would require effort.

Swatch

You can imagine a situation where you do not want Groovy scripts to be run by certain users and want to be e-mailed in case of their unwanted actions. If you want to react immediately to specific log patterns and do not already have infrastructure in place, consider using Swatch, an open source product that is freely available in most *NIX distributions (`http://sourceforge.net/projects/swatch/`, `http://www.jaxmag.com/itr/online_artikel/psecom,id,766,nodeid,147.html`).

Swatch is a Perl script that periodically reviews logs. If a pattern is found, it reacts with an e-mail or by executing commands.

See also

- *Improving security via small configuration changes*
- *Looking at the Jenkins user through Groovy*

Installing OpenLDAP with a test user and group

Lightweight Directory Access Protocol (**LDAP**) provides a highly popular Open Standards Directory Service. It is used in many organizations to display user information to the world. LDAP is also used as a central service to hold user passwords for authentication and can contain information necessary for routing mail, POSIX account administration, and various other pieces of information that external systems may require. Jenkins can directly connect to LDAP for authentication or indirectly through the CAS SSO server (`http://www.jasig.org/cas`), which then uses LDAP as its password container. Jenkins also has an **LDAP Email** plugin (`https://wiki.jenkins-ci.org/display/JENKINS/LDAP+Email+Plugin`) that pulls its routing information out of LDAP.

Because LDAP is a common Enterprise service, Jenkins may also encounter LDAP while running integration tests, as a part of the built-in applications testing infrastructure.

This recipe shows you how to quickly install an OpenLDAP (`http://www.openldap.org/`) server named **slapd** and then add organizations, users, and groups via **LDAP Data Interchange Format** (**LDIF**)—a simple text format for storing LDAP records (`http://en.wikipedia.org/wiki/LDAP_Data_Interchange_Format`).

Getting ready

This recipe assumes that you are running a modern Debian-based Linux operating system, such as Ubuntu.



For detailed instructions on installing OpenLDAP on Windows, please refer to `http://www.userbooster.de/en/support/feature-articles/openldap-for-windows-installation.aspx`.

Save the following LDIF entries to the file `basic_example.ldif`, and place it in your home directory:

```
dn: ou=mycompany,dc=nodomain
objectClass: organizationalUnit
ou: mycompany
dn: ou=people,ou=mycompany,dc=nodomain
objectClass: organizationalUnit
```

```
ou: people

dn: ou=groups,ou=mycompany,dc=nodomain
objectClass: organizationalUnit
ou: groups

dn: uid=tester1,ou=people,ou=mycompany,dc=nodomain
objectClass: inetOrgPerson
uid: tester1
sn: Tester
cn: I AM A Tester
displayName: tester1 Tester
userPassword: changeme
mail: tester1.tester@dev.null

dn: cn=dev,ou=groups,ou=mycompany,dc=nodomain
objectclass: groupofnames
cn: Development
description: Group for Development projects
member: uid=tester1,ou=people,dc=mycompany,dc=nodomain
```

How to do it...

1. Install the LDAP server `slapd`. Fill in the administrative password when asked.

   ```
   sudo apt-get install slapd ldap-utils
   ```

2. Add the LDIF records from the command line; you will then be asked for the administrator's password you filled in, in step 1.

   ```
   ldapadd -x -D cn=admin,dc=nodomain -W -f ./basic_example.ldif
   ```

How it works...

LDIF is a textual expression of the records inside LDAP.

Distinguished name (**dn**) is a unique identifier per record and is structured so objects reside in an organizational tree structure.

ObjectClasses such as organizational unit define a set of required and optional attributes. In the case of the organizational unit, the attribute `ou` is required. This is useful for bundling attributes that define a purpose, such as creating an organizational structure belonging to a group or having an e-mail account.

In the recipe, we have imported using the default dn of the admin account (dn: `cn=admin,dc=nodomain`). The account was created during package installation. To use another account, you will need to change the value of the `-D` option mentioned in step 2 of the recipe.

The default dn of the admin account may vary depending on which version of `slapd` you have installed.

The LDIF creates an organizational structure with three organizational units:

- dn: `ou=mycompany,dc=nodomain`
- dn: `ou=people,ou=mycompany,dc=nodomain`—Location to search for people
- dn: `ou=groups,ou=mycompany,dc=nodomain`—Location to search for groups

A **user**, dn: `uid=tester1,ou=people,ou=mycompany,dc=nodomain`, is created for testing. The list of attributes the record must have is defined by the objectClass `inetOrgPerson`.

A **group**, dn: `cn=dev,ou=groups,ou=mycompany,dc=nodomain`, is created via the objectClass `groupofnames`. The user is added to the group via adding the member attribute pointing to the dn of the user.

Jenkins looks for the username and to which groups the user belongs. In Jenkins, you can define which projects a user can configure, based on their group information. Therefore, you should consider adding groups that match your Jenkins Job structures such as development, acceptance, and also a group for those needing global powers.

There's more...

What is not covered by this LDIF example is the adding of objectClasses and **Access Control Lists** (**ACLs**).

- **ObjectClasses**: LDAP uses objectClasses as a sanity check on the incoming record creation requests. If the required attributes do not exist in a record or are of the wrong type, then LDAP will reject the data. Sometimes, it's necessary to add new objectClasses; you can do this with graphical tools. The recipe *Administering OpenLDAP* has an example of one such tool.
- **Access Control Lists**: ACLs define which user or which group can do what. For information on this complex subject area, please review `http://www.openldap.org/doc/admin24/access-control.html`. You can also review the `man` entry on your OpenLDAP server from the command line—`man slapd.access`

See also

- *Administering OpenLDAP*
- *Configuring the LDAP plugin*

Using Script Realm authentication for provisioning

For many Enterprise applications, provisioning occurs during the first login of the user. For example, a directory with content could be made, a user added to an e-mail distribution list, an Access Control List modified, or an e-mail sent to the marketing department.

This recipe will show you how to use two scripts—one to log in through LDAP and perform example provisioning, and the other to return the list of groups a user belongs to. Both scripts use Perl, which makes for compact code.

Getting ready

You need to have installed the Perl and the Net::LDAP modules. For a Debian distribution, you should install the **libnet-ldap-perl** package (`http://ldap.perl.org/FAQ.html`). You also need to have installed the **Script Realm** plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Script+Security+Realm`).

How to do it...

1. As a Jenkins user, place the following code in a file and save it under a directory that is controlled by a Jenkins user with executable permissions. Name the file `login.pl`. Verify that the `$home` variable is pointing to the correct workspace.

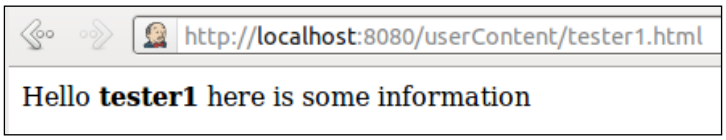
```
#!/usr/bin/perl
use Net::LDAP;
use Net::LDAP::Util qw(ldap_error_text);

my $dn_part="ou=people,ou=mycompany,dc=nodomain";
my $home="/var/lib/jenkins/userContent";
my $user=$ENV{'U'};
my $pass=$ENV{'P'};

my $ldap = Net::LDAP->new("localhost");
my $result =$ldap->bind
  ("uid=$user,$dn_part", password=>$pass);
```

```
if ($result->code){
  my $message=ldap_error_text($result->code);
  print "dn=$dn\nError Message: $message\n";
  exit(1);
}
# Do some provisioning
unless (-e "$home/$user.html"){
  open(HTML, ">$home/$user.html");
  print HTML
    "Hello <b>$user</b> here is some information";
  close(HTML);
}
exit(0);
```

2. As a Jenkins user, place the following code in a file and save it under a directory that is controlled by a Jenkins user with executable permissions. Name the file `group.pl`.

```
#!/usr/bin/perl
print "guest,all";
exit(0);
```

3. Configure the plugin via the Jenkins configuration screen, under the subsection **Security Realm**:
   - Check **Authenticate via custom script**
   - **Login Command**: `/var/lib/Jenkins/login.pl`
   - **Groups Command**: `/var/lib/Jenkins/group.pl`
   - **Groups Delimiter**: ,
4. Press the **Save** button.
5. Log in as `tester1`, with the password `changeme`.
6. Visit the provisioned content at `http://localhost:8080/userContent/tester1.html`.



How it works...

The file `login.pl` pulls in the username and password from the environment variables `U` and `P`. The script then tries to self bind the user to a calculated unique LDAP record. For example, the distinguished name of the user `tester1` is:

```
uid=tester1, ou=people,ou=mycompany,dc=nodomain
```



Self binding happens when you search for your own LDAP record and at the same time authenticate as yourself. This approach has the advantage of allowing your application to test a password's authenticity without using a global administration account.

If authentication fails, an exit code of `1` is returned. If authentication succeeds, the provisioning process takes place followed by an exit code of `0`.

If the file does not already exist, it is created. A simple HTML file is created during the provisioning process. This is just an example; you can do a lot more, from sending e-mail reminders to full account provisioning across the breadth of your organization.

The `group.pl` script simply returns two groups that include every user, guest, and many more. `guest` is a group intended for guests only. `all` is a group that all users belong to, including the guests. Later, if you want to send e-mails out about the maintenance of services, you can use an LDAP query to collect e-mail addresses via the `all` group.

There's more...

LDAP servers are used for many purposes, depending on the schemas used. You can route mail, create login accounts, and so on. These accounts are enforced by common authentication platforms, such as **Pluggable Authentication Modules** (**PAM**), in particular **PAM_LDAP** (`http://www.padl.com/OSS/pam_ldap.html` and `http://www.yolinux.com/TUTORIALS/LDAP_Authentication.html`).

At the University of Amsterdam, we use a custom schema so that user records have an attribute for the counting down of records. A scheduled task performs an LDAP search on the counter and then decreases the counter by one. The task notes when the counter reaches certain numbers and then performs actions, such as sending out e-mail warnings.

You can imagine using this method in conjunction with a custom login script. Once a consultant logs in to Jenkins for the first time, they are given a certain period of grace before their LDAP record is moved to a *to be ignored* branch.

See also

- *Reviewing Project-based Matrix tactics via a custom group script*

Reviewing project-based matrix tactics via a custom group script

Security best practices dictate that you should limit the rights of individual users to the level that they require.

This recipe explores the Project-based Matrix strategy. In this strategy, you can assign individual users or groups different permissions on a job-by-job basis.

The recipe uses custom realm scripts to allow you to log in with any name and a password whose length is greater than five characters and to place the test users in their own unique group. This will allow you to test out the Project-based Matrix strategy.

Getting ready

You will need to install the **Script Realm** plugin and also have Perl installed with the URI module (`http://search.cpan.org/dist/URI/URI/Escape.pm`). The URI module is included in modern Perl distributions, so in most situations, the script will work out of the box.

How to do it...

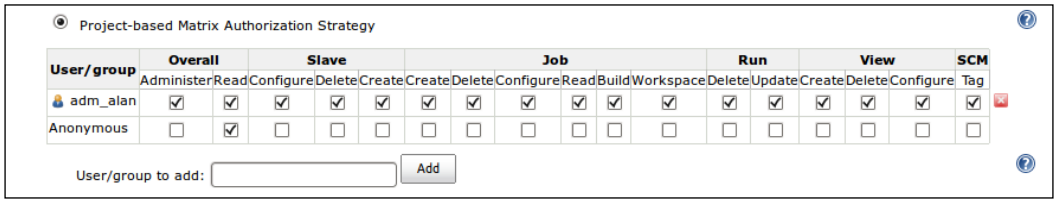
1. Copy the following script to the file `login2.pl`, in the Jenkins workspace, making sure that Jenkins can execute the script:

```
#!/usr/bin/perl
my $user=$ENV{'U'};
my $pass=$ENV{'P'};
my $min=5;

if (
  (length($user) < $min) || (length($pass) < $min)) {
    //Do something here for failed logins
    exit (-1);
}
exit(0);
```

2. Copy the following script to the file `group2.pl`, in the Jenkins workspace, making sure that Jenkins can execute the script:

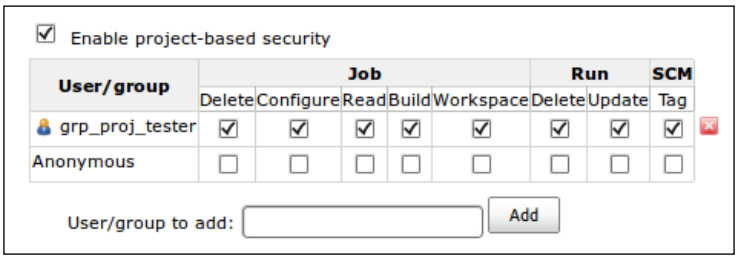
```
#!/usr/bin/perl
use URI;
use URI::Escape;
my $raw_user=$ENV{'U'};
my $group=uri_escape($raw_user);
print "grp_$group";
exit(0);
```

3. Configure the plugin via the Jenkins configuration screen under the subsection **Security Realm**.
4. Tick the **Authenticate via custom script** checkbox and add the following details:
   - **Login Command**: `/var/lib/Jenkins/login2.pl`
   - **Groups Command**: `/var/lib/Jenkins/group2.pl`
   - **Groups Delimiter**: ,
5. Check the **Project-based Matrix Authorization Strategy** checkbox.
6. Add a user, called `adm_alan`, with full rights.



7. Press the **Save** button.
8. Try to log in as `adm_alan` with a password less than five characters.
9. Log in as `adm_alan` with any password greater than five characters.
10. Create a new Job with the name `project_matrix_test`, with no configuration.
11. Check the **Enable project-based security** checkbox within the Job.

12. Add the group `grp_proj_tester`, with full rights (that is, all tickboxes checked).



13. Log in as user `I_cant_see_you`. Note that you cannot view the recently created job, `project_matrix_test`.
14. Log in as `proj_tester`. Note that you can now view and configure `project_matrix_test`.

How it works...

`login2.pl` allows any username/password combination to succeed, if it is at least the length defined in the `$min` variable.

`group2.pl` reads the username from the environment, and then escapes the name to make sure that no evil scripting is accidentally run later.

`group2.pl` places the user in the group `grp_username`. For example, if `proj_tester` logs in, they will belong to the group `grp_proj_tester`.

The group script allows us to log in as arbitrary users and view their permissions. In the Project-based Matrix strategy, the permissions per user or group are defined at two levels:

1. **Globally**, via the Jenkins configuration page. This is where you should define your global accounts for system-wide administration.
2. **Per project**, via the job configuration screen. The global accounts can gain extra permissions per project but cannot lose permissions.

In this recipe, you logged in with a global account, `adm_alan`, that behaved as a **root admin**. Then you logged in as `I_cant_see_you`—this has no extra permissions at all and can't even see the job from the front page. Finally, you logged in as `proj_tester`, who belonged to the group `grp_proj_tester`, which has full powers within the specific job.

Using the per-project permissions, you can not only limit the powers of individual users but also shape which projects they view. This feature is particularly useful for Jenkins masters that have a wealth of jobs.

There's more...

Here are a few more things you should consider:

My own custom security flaw

I expect you have already spotted this. The login script has a significant security flaw. The username input, as defined by the `U` variable, has not been checked for malicious content. For example, the username could be:

```
<script>alert('Do something');</script>
```

Later on, if an arbitrary plugin displays the username as part of a custom view, then if the plugin does not safely escape, the username is run in the end user's browser. This example shows how easy it is to get security wrong. You are better off using well-known and trusted libraries when you can. For example, the OWASP's Java specific **AntiSamy** library (`https://www.owasp.org/index.php/Category:OWASP_AntiSamy_Project`) does an excellent job of filtering input in the form of CSS or HTML fragments.

For Perl, there are numerous excellent articles on this subject, including the following one:

`http://www.perl.com/pub/2002/02/20/css.html`

Static code review, tainting, and untainting

Static code review is the name for tools that read code that is not running and review for known code defects of this. PMD and FindBugs are excellent examples (`http://fsmsh.com/2804.com`).

A number of these generic tools can review your code for security defects. One of the approaches taken is to consider input **tainted** if it comes from an external source, such as the Internet or directly from input from files. To **untaint**, the input has to first be passed through a regular expression and unwanted input safely escaped, removed, and/or reported.

See also

- *Using the Script Realm authentication for provisioning*

Administering OpenLDAP

This recipe is a quick start to LDAP administration. It details how you can add or delete user records from the command line and highlights the use of an example LDAP browser. These skills are useful for maintaining an LDAP server for use in integration tests or for Jenkins account administration.

Getting ready

To try this out, you will need Perl installed with the Net::LDAP modules. For example, for a Debian distribution, you should install the `libnet-ldap-perl` package (`http://ldap.perl.org`).

You will also need to install the LDAP browser—**JXplorer** (`http://jxplorer.org/`).

How to do it...

1. To add a user to LDAP, you will need to write the following LDIF record to a file named `basic_example.ldif`:

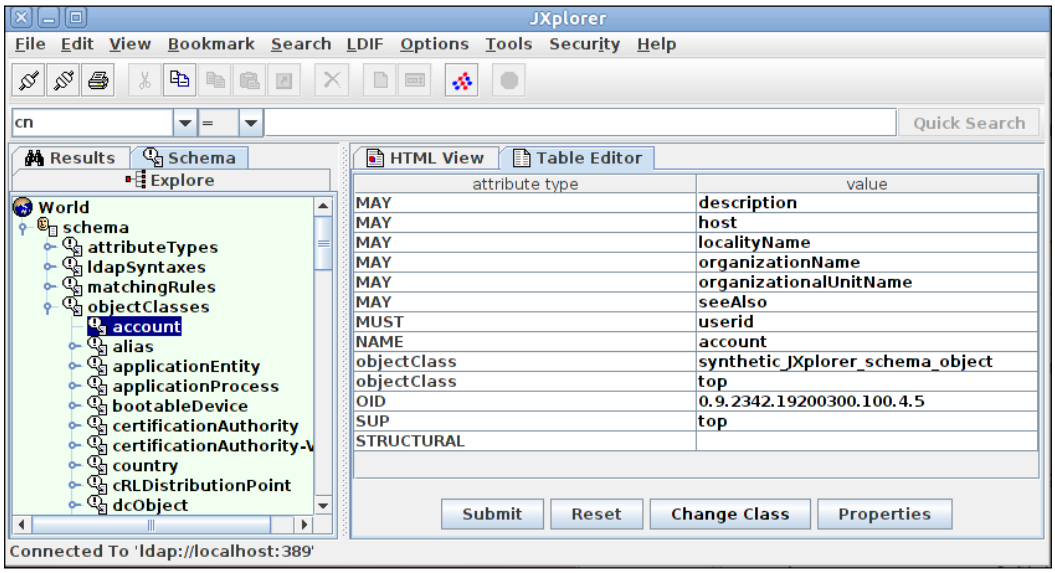
```
dn: uid=tester121,
  ou=people,ou=mycompany,dc=nodomain
objectClass: inetOrgPerson
uid: tester121
sn: Tester
givenName: Tester121 Tester
cn: Tester121 Tester
displayName: Tester121 Tester
userPassword: changeme
mail: 121.tester@dev.null
```

2. Add a new line at the end of the record and copy the aforementioned record into the textfile, replacing the number `121` with `122`, wherever it occurs in the second record.
3. Run the following `ldapadd` command, inputting the LDAP administrator's password when asked:

```
ldapadd -x -D cn=admin,dc=nodomain -W -f ./basic_example2.ldif
```

4. Run JXplorer, connecting with the following values:
   - **HOST**: `localhost`
   - **Level**: `Anonymous`
5. Select the **Schema** tab, and then select the objectClass **account**.

6. In **Table Editor**, you will see attributes mentioned with **MAY** or **MUST**:



7. Disconnect from the anonymous account by selecting **File | Disconnect**.
8. Reconnect as the admin account by selecting **File | Connect**:
   - **Host**: `Localhost`
   - **Level**: `User+Password`
   - **User DN**: `cn=admin,dc=nodomain`
   - **Password**: `your password`
9. Under the **Explore** tab, select **tester1**.
10. In the **Table Editor**, add the value `1021 XT` to the **postalCode**.
11. Click on **Submit**.
12. Select the **LDIF** menu option, at the top of the screen, and select **Export Subtree**.
13. Click on the **OK** button and write the name of the file that you are going to export to the LDIF.
14. Click on **Save**.
15. Create an executable script with the following code and run it:

```
#!/usr/bin/perl
use Net::LDAP;
use Net::LDAP::Util qw
  (ldap_error_text);
```

```
my $number_users=2;
my $counter=0;
my $start=100;

my $ldap = Net::LDAP->
  new("localhost");
$ldap->bind("cn=admin,dc=nodomain",
  password=>"your_password");

while ($counter < $number_users){
  $counter++;
  $total=$counter+$start;
  my $dn="uid=tester$total,ou=people,
    ou=mycompany,dc=nodomain";
  my $result = $ldap->delete($dn);
  if ($result->code){
    my $message=
      ldap_error_text($result->code);
    print "dn=$dn\nError Message: $message\n";
  }
}
```

How it works...

In the recipe, you have performed a range of tasks. First, you have used an LDIF file to add two users. This is a typical event for an LDAP administrator in a small organization. You can keep the LDIF file and then make minor modifications to add or delete users, groups, and so on.

Next, you have viewed the directory structure anonymously through an LDAP browser, in this case, JXplorer. JXplorer runs on a wide range of operating systems and is open source. Your actions indicate that LDAP is an Enterprise directory service, where things are supposed to be found even by anonymous users. The fact that pages render fast in JXplorer indicates that LDAP is a read-optimized database that returns search results efficiently.

Using an LDAP browser generally gets more frustrating as the number of objects to render increases. For example, at the University of Amsterdam, there are more than 60,000 student records under one branch. Under these circumstances, you are forced to use command-line tools or be very careful with search filters.

Being able to view objectClasses, knowing which attributes are optional and that you **may** use and which attributes are required and that you **must** use, helps you to optimize your records.

Next, you **bind** (perform some action) as an admin user and manipulate the `tester1` record. For small organizations, this is an efficient means of administration. Exporting the record to LDIF allows you to use the record as a template for further importing of records.

The deletion script is an example of programmatic control. This gives you a lot of flexibility for large-scale generation, modification, and deletion of records, by changing just a few variables. Perl was chosen because of its lack of verbosity. The use of these types of scripts is typical for the provisioning of integration tests.

Within the deletion script, you will see that the number of users to delete is set to `2` and the starting value of the tester accounts is `100`. This implies that the two records you had previously generated are going to be deleted, for example, `tester101` and `tester102`.

The script binds once, as the admin account, and then loops through a number of records using `$counter` as a part of the calculation of the distinguished name of each record. The `delete()` method is called for each record, and any errors generated will be printed out.

There's more...

You should consider deleting users' Perl script as an example of how to provision or clean up an LDAP server that is needed for integration tests, efficiently. To create an add rather than a delete script, you could write a similar script replacing `my $result = $ldap->delete($dn);` with something similar to:

```
my $result = $ldap->add($dn, attrs => [ @$whatToCreate ]);
```

(Where `@$whatTOCreate` is a **hash** containing attributes and objectClasses).

For more examples, refer to `http://search.cpan.org/~gbarr/perl-ldap/lib/Net/LDAP/Examples.pod#OPERATION_-_Adding_a_new_Record`.

See also

- *Installing OpenLDAP with a test user and group*
- *Configuring the LDAP plugin*

Configuring the LDAP plugin

LDAP is the standard for Enterprise directory services. This recipe explains how to attach Jenkins to your test LDAP server.

Getting ready

This recipe assumes that you have performed the *Installing OpenLDAP with a test user and group* recipe.

How to do it...

1. Enter the Jenkins configuration screen. Select **Enable security**.
2. Tick the **LDAP** checkbox.
3. Add the Server value `127.0.0.1`
4. Press the **Advance** button.
5. Add the following details:
   - **User Search Base**: `ou=people,ou=mycompany,dc=nodomain`
   - **User Search filter**: `uid={0}`
   - **Group Search base**: `ou=groups,ou=mycompany,dc=nodomain`

How it works...

The test LDAP server supports anonymous binding—you can search the server without authenticating. Most LDAP servers allow this approach. However, some servers are configured to enforce specific information security policies. For example, your policy might enforce being able anonymously to verify that a user's record exists, but you may not be able to retrieve specific attributes, such as their e-mail or postal address.

Anonymous binding simplifies configuration, otherwise you will need to add account details for a user in LDAP with the rights to perform the searches. This account, having great LDAP powers, should never be shared and can present a chink in your security armor.

The **User Search filter**, `uid={0}`, searches for users whose `uid` equals their username. Many organizations prefer to use `cn` instead of `uid`; the choice of attribute is a matter of taste. You can even imagine an e-mail attribute being used to uniquely identify a person, as long as that attribute cannot be changed by the user.



The security realm

When you log in to an instance of the Java class, `hudson.security.LDAPSecurityRealm` is called. The code is defined in a Groovy script, which you can find under `WEB-INF/security/LDAPBindSecurityRealm.groovy`, within the `Jenkins.war` file.

For further information visit `http://wiki.hudson-ci.org/display/HUDSON/Standard+Security+Setup`.

There's more...

Here are a few more things for you to think about:

The difference between misconfiguration and bad credentials

While configuring the LDAP plugin for the first time, your authentication process might fail due to misconfiguration. Luckily, Jenkins produces error messages. For the Debian Jenkins package, you can find the logfile at `/var/log/jenkins/jenkins.log`; for the Windows version running as a service, you can find the relevant logs through the **Events Viewer** by filtering on Jenkins source.

The two main errors consistently made are:

1. **Misconfigured DN**: A misconfigured DN for either **User Search Base** or **Group Search Base** will have the relevant log entry similar to the following:

```
org.acegisecurity.AuthenticationServiceException:
  LdapCallback;[LDAP: error code 32 - No Such Object];
  nested exception is javax.naming.NameNotFoundException:
  [LDAP: error code 32 - No Such Object];
  remaining name 'ou=people,dc=mycompany ,dc=nodomain'
```

2. **Bad Credentials**: If the user does not exist in LDAP, you have either typed in the wrong password or you have accidently searched the wrong part of the LDAP tree; the log error will start with the following text:

```
org.acegisecurity.BadCredentialsException: Bad credentials
```

Searching

Applications retrieve information from LDAP in a number of ways:

- **Anonymously** for generic information. This approach works only for information that is exposed to the world. However, the LDAP server can limit the search queries to specific IP addresses as well. The application will then be dependent on the attributes that your organization is prepared to disclose. If the information security policy changes, the risk is that your application might break accidently.
- **Self-bind**: The application binds as a user and then searches with the user's rights. This approach is the cleanest. However, it is not always clear in the logging whether the application is behind these actions.
- Using an **application-specific admin account** with many rights: The account gets all the information that your application requires, but if disclosed to the wrong people, can cause significant issues quickly.



If the LDAP server has an account-locking policy, it is simple for a cracker to lock out the application.

In reality, the approach chosen is defined by the already defined Access Control policy of your Enterprise directory service.



Review plugin configuration

Currently, there are over 300 plugins for Jenkins. It is possible, though unlikely, that occasionally passwords are being stored in plain text in the XML configuration files in the workspace directory or plugins directory. Every time you install a new plugin that requires a power user's account, you should double check the related configuration file. If you see a plain textfile, you should write a bug report attaching a patch to the report.

See also

- *Installing OpenLDAP with a test user and group*
- *Administering OpenLDAP*

Installing a CAS server

Yale CAS (`http://www.jasig.org/cas`) is a Single Sign-on Server. It is designed as a campus-wide solution and, as such, is easy to install and relatively simple to configure to meet your specific infrastructural requirements. CAS allows you to sign in once, and then automatically use lots of different applications without logging in again. This is made for a much more pleasant user interaction across the range of applications used by a typical Jenkins user during their day.

Yale CAS has helper libraries in Java and PHP that make integration of third-party applications straightforward.

Yale CAS also has the significant advantage of having a pluggable set of handlers that authenticate across a range of backend servers, such as LDAP, OpenID (`http://openid.net/`), and RADIUS (`http://en.wikipedia.org/wiki/RADIUS`).

In this recipe, you will install the complete version of a CAS server running from within a Tomcat 7 server. This recipe is more detailed than the rest in this chapter, and it is quite easy to misconfigure. To help, the modified configuration files mentioned in this recipe will be downloadable from the Packt website.

Getting ready

Download Yale CAS (`http://www.jasig.org/cas/download`) and unpack it.
This recipe was written with version 3.4.8, but it should work for earlier or later versions, with little modification.

Install Tomcat 7 (`http://tomcat.apache.org/download-70.cgi`). The recipe assumes that the installed Tomcat server is initially turned off.

How to do it...

1. In the unpacked Tomcat directory, edit `conf/server.xml`, and comment out the port 8080 configuration information:

```
<!--
  <Connector port="8080" protocol="HTTP/1.1"
-->
```

2. Add the following code underneath the text needed to enable port 9443 with SSL:

```
<Connector port="9443"
  protocol="org.apache.coyote.http11.Http11Protocol"
  SSLEnabled="true" maxThreads="150"
  scheme="https" secure="true"
  keystoreFile="${user.home}/.keystore"
  keystorePass="changeit"
  clientAuth="false" sslProtocol="TLS" />
```

3. As the user that Tomcat will run under, create a **self-signed** certificate via:

```
keytool -genkey -alias tomcat -keyalg RSA
```



Note: If `keytool` is not found on your `PATH`, you might have to fill in the full location to the `bin` directory of your Java folder.

4. From underneath the root directory of the unpacked CAS server, copy the file `modules/cas-server-uber-webapp-3.x.x` (where `x.x` is the specific version number) to the Tomcat `webapps` directory, making sure the file is renamed to `cas.war`.
5. Start Tomcat.
6. Log in via `https://localhost:9443/login/cas`, with the username equal to the password, for example, `smile/smile`.
7. Stop Tomcat.

8. Either modify the `webapps/cas/Web-INF/deployerConfigContext.xml` file or replace with the example file previously downloaded from the Packt website. To modify, you will need to comment out `SimpleTestUsernamePasswordAuthenticationHandler`:

```
<!--
  <bean class=
    "org.jasig.cas.authentication.handler.support.
    SimpleTestUsernamePasswordAuthenticationHandler" />
-->
```

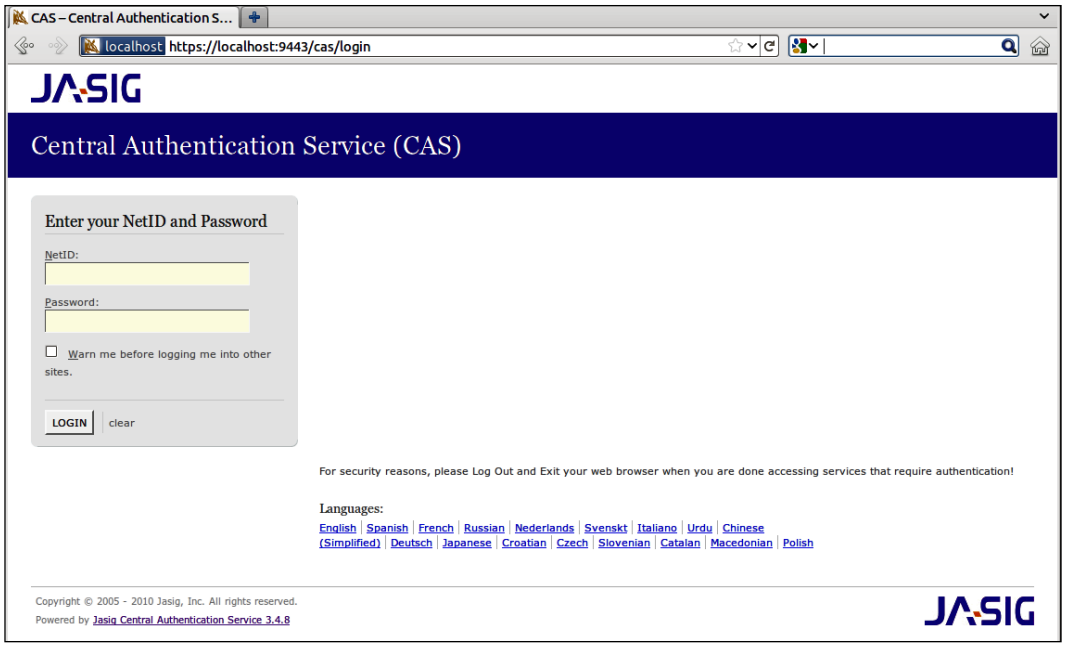
9. Underneath the commented-out code, add the configuration information for LDAP:

```
<bean class=
  "org.jasig.cas.adaptors.ldap.
  BindLdapAuthenticationHandler">
  <property name="filter" value="uid=%u" />
  <property name="searchBase"
    value="ou=people,ou=mycompany,dc=nodomain" />
  <property name="contextSource" ref="contextSource" />
</bean>
</list>
</property>
</bean>
```

10. Beneath `</bean>`, add an extra bean configuration, replacing the password value with that of yours:

```
<bean id="contextSource" class=
  "org.springframework.ldap.core.support.LdapContextSource">
    <property name="pooled" value="false"/>
    <property name="urls">
      <list>
        <value>ldap://localhost/</value>
      </list>
    </property>
    <property name="userDn" value="cn=admin,dc=nodomain"/>
    <property name="password" value="adminpassword"/>
    <property name="baseEnvironmentProperties">
      <map>
        <entry>
          <key>
            <value>java.naming.security.
              authentication</value>
          </key>
            <value>simple</value>
        </entry>
      </map>
  </property>
</bean>
```

11. Restart Tomcat.
12. Log in via `https://localhost:9443/cas/login`, using the `tester1` account. If you see a page similar to the following, congratulations, you now have a running SSO!



How it works...

By default, Tomcat runs against port 8080, which happens to be the same port number as Jenkins. To change the port number to 9443 and turn on SSL, you must modify `conf/server.xml`. For SSL to work, Tomcat needs to have a keystore with a private certificate, using the variable `${user.home}` to point to the home directory of the Tomcat user, for example:

```
keystoreFile="${user.home}/.keystore" keystorePass="changeit"
```

The protocol you choose is TLS, which is a more modern and secure version of SSL. For further details, see `http://tomcat.apache.org/tomcat-7.0-doc/ssl-howto.html`.

Next, you generate a certificate and place it in the Tomcat user's certificate store, ready for Tomcat to use. Your certificate store might contain many certificates; the alias `tomcat` uniquely identifies the appropriate certificate.

Within the downloaded CAS package, there are two CAS WAR files. The larger WAR file contains the libraries for all the authentication **handlers**, including the required LDAP handler.

The default setup allows you to log in with the same values for username and password. This setup is for demonstration purposes. To replace or chain together handlers, you have to edit `webapps/cas/Web-INF/deployerConfigContext.xml`. For more in-depth details, read `https://wiki.jasig.org/display/CASUM/LDAP`.

If at any time you are having problems with configuration, the best place to check is in Tomcat's main log, the `logs/catalina.out`. For example, a bad username or password will generate an error similar to the following:

```
WHO: [username: test]
WHAT: error.authentication.credentials.bad
ACTION: TICKET_GRANTING_TICKET_NOT_CREATED
APPLICATION: CAS
WHEN: Mon Aug 08 21:14:22 CEST 2011
CLIENT IP ADDRESS: 127.0.0.1
SERVER IP ADDRESS: 127.0.0.1
```

There's more...

Here are a few more things you should consider:

Backend authentication

Yale CAS has a wide range of backend authentication handlers, and it is straightforward for a Java developer to write his own. The following table mentions the current handlers. Expect the list to expand.



Note: Using well-supported third-party frameworks, such as JAAS and JDBC implementations, you can connect to a much wider set of services than mentioned in the table.

Active Directory	Connect to your windows infrastructure.
JAAS	JAAS implements a Java version of the standard Pluggable Authentication Module (PAM) framework. This allows you to pull in other authentication mechanisms, such as Kerberos.
LDAP	Connect to your Enterprise directory services.
RADIUS	Connect to Radius.
Trusted	Is used to offload some of the authentication to an Apache server or another CAS server.
Generic	A set of small generic handlers, such as a handler to accept a user from a list or from a file.

JDBC	Connect to databases, and there are even drivers for spreadsheets and LDAP.
Legacy	Supports the CAS2 protocol.
SPNEGO	Simple and Protected GSSAPI Negotiation Mechanism allows the CAS server to negotiate between protocols with a backend service, possibly allowing transitioning between backend services.
X.509 Certificates	Require a trusted client certificate.

An alternative installation recipe using ESUP CAS

The ESUP consortium also provides a repackaged version of CAS, which includes additional ease-of-use features, including an out of the box demonstration version. However, the ESUP version of the CAS server lags behind the most current version. If you want to compare the two versions, you can find the ESUP installation documentation at `http://esup-casgeneric.sourceforge.net/install-esup-cas-quick-start.html`.



The ESUP package is easier to install and configure than this recipe, however, it includes an older version of CAS.

Trusting LDAP SSL

Having SSL enabled on your test LDAP server avoids sniffable passwords being sent over the wire, but you will need to get the CAS server to trust the certificate of the LDAP server. Here's a relevant quote from the **Jasig Wiki**:

> *Please note that, your JVM needs to trust the certificate of your SSL enabled LDAP server or CAS will refuse to connect to your LDAP server. You can add the LDAP server's certificate to the JVM trust store* (`$JAVA_HOME/jre/lib/security/cacerts`) *to solve that issue.*

A few useful resources

There are many useful resources on the Jasig Wiki (`https://wiki.jasig.org/`), including:

- Securing your CAS server: `https://wiki.jasig.org/display/CASUM/Securing+Your+New+CAS+Server`
- Connecting CAS to a database: `https://wiki.jasig.org/display/CAS/Examples+to+Configure+CAS`
- Creating a high availability infrastructure: `http://www.ja-sig.org/wiki/download/attachments/22940141/HA+CAS.pdf?version=1`

See also

- *Enabling SSO in Jenkins*

Enabling SSO in Jenkins

In this recipe, you will enable CAS in Jenkins through the use of the **Cas1** plugin. For the CAS protocol to work, you will also need to build a trust relationship between Jenkins and the CAS server. The Jenkins plugin trusts the certificate of the CAS server.

Getting ready

To try this out, you will need to have installed a CAS server, as described in the recipe *Installing a CAS server*.

How to do it...

1. You will need to export the public certificate of the CAS server. You do this from a Firefox 6 web browser by visiting `http://localhost:9443`.
2. In the address bar, you will see a Tomcat icon on the left-hand side. Click on the icon, and a security pop-up dialog will appear.
3. Click on the **More information** button.
4. Click on the **View Certificate** button.
5. Select the **Details** tab.
6. Click on the **Export** button.
7. Choose a location for your public certificate to be stored.
8. Press **Save**.
9. Import the certificate into the Java keystore by using the command-line option:

```
sudo keytool -import -alias myprivateroot -keystore
  ./cacerts -file location_of_exported certificate
```

10. To configure your CAS setting, visit the Jenkins **Config Screen**.
11. Under **Access Control**, check the **CAS protocol version 1** checkbox, and fill in the following details:

```
CAS Server URL: https://localhost:9443
Hudson Host Name: localhost:8080
```

12. Log out of Jenkins.
13. Log in to Jenkins. You will now be redirected to the CAS server.
14. Log in to the CAS server. You will now be redirected back to Jenkins.

How it works...

The CAS plugin cannot verify the client's credentials unless it trusts the CAS server certificate. If the certificate is generated by a well-known trusted authority, their **root** certificates are most likely already in the default keystore (`cacerts`). This comes pre-packaged with your Java installation. However, in the CAS installation recipe, you had created a self-signed certificate.

The configuration details for the CAS plugin are trivial. Note that you left the the **Roles Validation script** blank. This implies that your matrix-based strategies will have to rely on users being given specific permissions rather than groups defined by a customized CAS server.

Congratulations, you have a working SSO in which Jenkins can play its part seamlessly with a large array of other applications and authentication services.

See also

- *Installing a CAS server*

3
Building Software

In this chapter, we will cover the following recipes:

- Plotting alternative code metrics in Jenkins
- Running Groovy scripts through Maven
- Manipulating environmental variables
- Running AntBuilder through Groovy in Maven
- Failing Jenkins Jobs based on JSP syntax errors
- Configuring Jetty for integration tests
- Looking at license violations with RATs
- Reviewing license violations from within Maven
- Exposing information through build descriptions
- Reacting to the generated data with the Post-build Groovy plugin
- Remotely triggering Jobs through the Jenkins API
- Adaptive site generation

Introduction

This chapter reviews the relationship between Jenkins and Maven builds and also a small amount of scripting with Groovy and Ant.

Jenkins is the master of flexibility. It works well across multiple platforms and technologies. Jenkins has an intuitive interface with clear configuration settings. This is great for getting the job done. However, it is also important that you clearly define the boundaries between the Jenkins plugins and the Maven build files.

A lack of separation will make you unnecessarily dependent on Jenkins. If you know that you will always run your builds through Jenkins, then you can afford to place some of the core work in the Jenkins plugins, gaining interesting extra functionality. However, if you want to always be able to build, test, and deploy directly, then you will need to keep the details in the `pom.xml`. You will have to judge the balance in where you add the configuration. It is easy to have the feature creep as you use more of the Jenkins plugin's feature set. The UI is easier to configure than writing a long `pom.xml`. The improved readability translates into fewer configuration-related defects. It is also simpler for you to use Jenkins for most of the common tasks, such as transporting artifacts, communicating, and plotting the trends of tests.

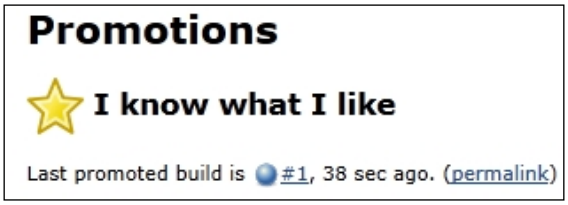
An example of the interplay between Jenkins and Maven is the use of Jenkins Publish Over SSH Plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Publish+Over+SSH+Plugin`). You can configure to transfer files or add a section of the `pom.xml`, as follows:

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-antrun-plugin</artifactId>
      <configuration>
        <tasks>
          <scp file="${user}:${pass}@${host}:${file.remote}"
            localTofile="${file.local}"/>
        </tasks>
      </configuration>
      <dependencies>
        <dependency>
          <groupId>ant</groupId>
          <artifactId>ant-jsch</artifactId>
          <version>1.6.5</version>
        </dependency>
        <dependency>
          <groupId>com.jcraft</groupId>
          <artifactId>jsch</artifactId>
          <version>0.1.42</version>
        </dependency>
      </dependencies>
    </plugin>
  </plugins>
</build>
```

Remembering the dependencies on specific JARs and versions which the Maven plugin uses at times feels like magic. The Jenkins plugins simplify details.

Later in the chapter, you will be given the chance to run Groovy scripts with **AntBuilder**. Each approach is viable, and the use depends more on your preferences rather than one choice.

Jenkins plugins work well together. For example, the promoted builds plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Promoted+Builds+Plugin`) signals when a build has met a certain criteria, placing an icon by a successful build. You can use this feature to signal, for example, to the QA team that they need to test the build or system administrators to pick up the artifacts and deploy.



Other plugins can also be triggered by promotion, including the SSH plugin. However, Maven is not aware of the promotion mechanism. As Jenkins evolves, expect more plugin interrelationships.

Jenkins is well-versed choreographing of actions. You should keep the running time of a Job to a minimum and offset heavier Jobs to nodes. Heavy Jobs tend to be clustered around document generation or testing. Jenkins allows you to chain Jobs together, and hence Jobs will be coupled to specific Maven goals, such as integration testing (`http://Maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference`). Under these circumstances, you are left with the choice of writing a number of build files, perhaps as a multi-module project (`http://Maven.apache.org/plugins/Maven-assembly-plugin/examples/multimodule/module-source-inclusion-simple.html`) or a thicker `pom.xml`, with different goals ready to be called across Jobs. **Keep It Simple Stupid** (**KISS**) biases the decision towards a larger, single file.

A template pom.xml

The recipes in this chapter will include `pom.xml` examples. To save page space, only the essential details will be shown. You can download the full examples from the Packt Publishing website.

The examples were tested against Maven 2.2.1. You will need to install this version on your Jenkins server, giving it the label `2.2.1`.

To generate a basic template for a Maven project, you have two choices. You can create a project through the archetype goal (`http://Maven.apache.org/guides/introduction/introduction-to-archetypes.html`). Or you can start off with a simple `pom.xml file`, as shown in the following code:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://Maven.apache.org/maven-v4_0_0.xsd">
```

```
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.berg</groupId>
    <artifactId>ch3.builds.xxx</artifactId>
    <version>1.0-SNAPSHOT</version>
    <name>Template</name>
</project>
```

The template looks simple, but is only part of a larger effective `pom.xml`. It is combined with the default values that reside hidden in Maven. To view the expanded version, you will need to run the following command:

```
mvn help:effective-pom
```

Unless otherwise stated, the fragments mentioned in the recipes should be inserted into the template, just before the `</project>` tag, updating your `groupID`, `artifactID`, and `version` values to your own taste.

Setting up a File System SCM

In the previous chapters, you used recipes that copied the files into the workspace. This is easy to explain, but is OS-specific. You can also do the file copying through the **File System SCM plugin** (`https://wiki.jenkins-ci.org/display/JENKINS/File+System+SCM`), which is OS-agnostic. You will need to install the plugin, ensuring that the files have the correct permissions, so that the Jenkins user can copy. In Linux, consider placing the files beneath the Jenkins home directory: `/var/lib/Jenkins`.

Plotting alternative code metrics in Jenkins

This recipe details how to plot custom data using the **plot** plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Plot+Plugin`). This allows you to expose numeric build data visually.

Jenkins has many plugins that create views of the test results generated by builds. The **Analysis collector plugin** pulls in the results from a number of these plugins to create an aggregated summary and history (`https://wiki.jenkins-ci.org/display/JENKINS/Analysis+Collector+Plugin`). This is great for plotting the history of the standard result types, such as JUnit, FindBugs, JMeter, and NCSS. There is also a plugin (`http://docs.codehaus.org/display/SONAR/Hudson+and+Jenkins+Plugin`) that supports pushing data to Sonar (`http://www.sonarsource.org/`). **Sonar** specializes in reporting a project's code quality. However, despite the wealth of options, there may come a time when you will need to plot custom results.

Scenario: You want to know the history of how many hits or misses are generated in your custom cache during integration testing. Plotting over builds will give you an indicator of whether the code change is improving or degrading performance.

In this recipe, a simple Perl script will generate random cache results.

Getting ready

In the **Plugin Manager** section of Jenkins, for example `http://localhost:8080/pluginManager/available`, install the Plot plugin.

Create the directory `ch3.building_software/plotting`.

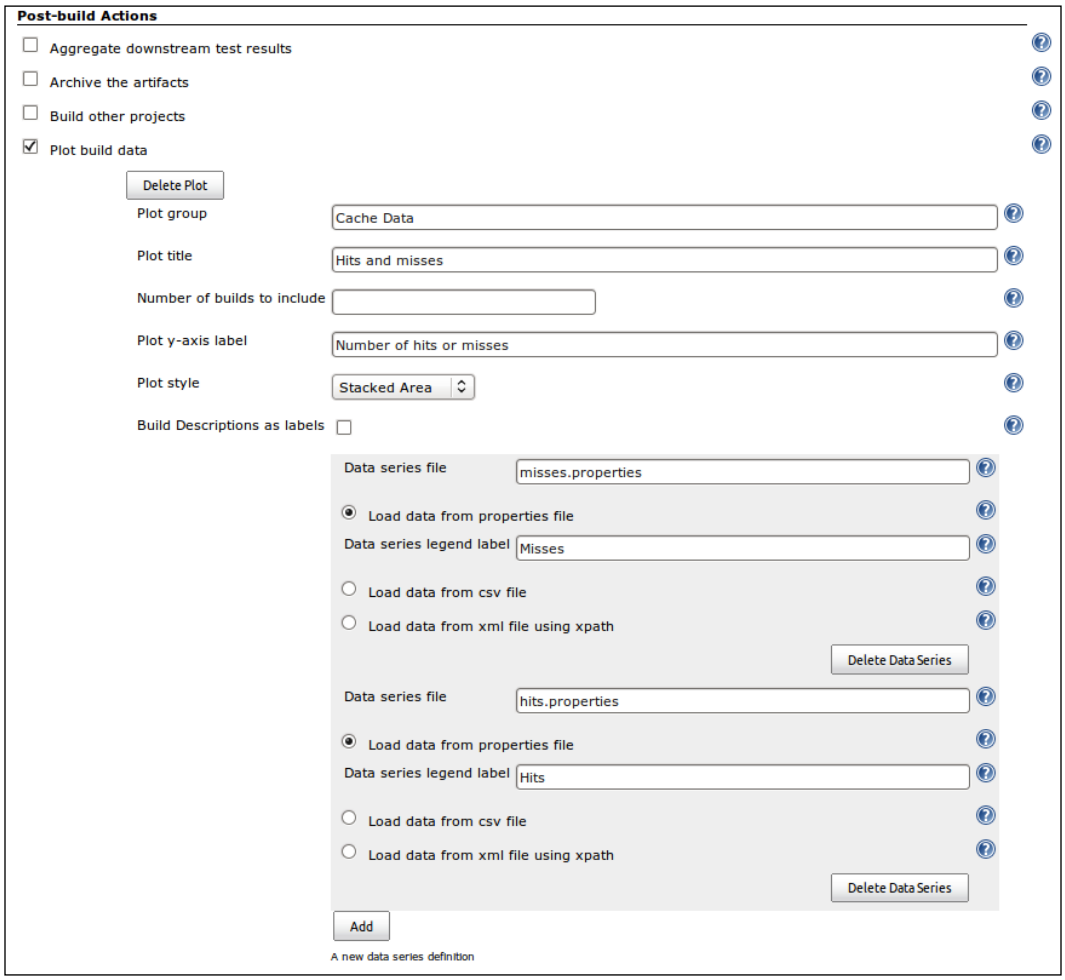
How to do it...

1. Create the file `ch3.building_software/plotting/hit_and_miss.pl`, adding to it the following contents:

```
#!/usr/bin/perl
my $workspace = $ENV{'WORKSPACE'};

open(P1, ">$workspace/hits.properties")|| die;
open(P2, ">$workspace/misses.properties")|| die;
print P1 "YVALUE=".rand(100);
print P2 "YVALUE=".rand(50);
```

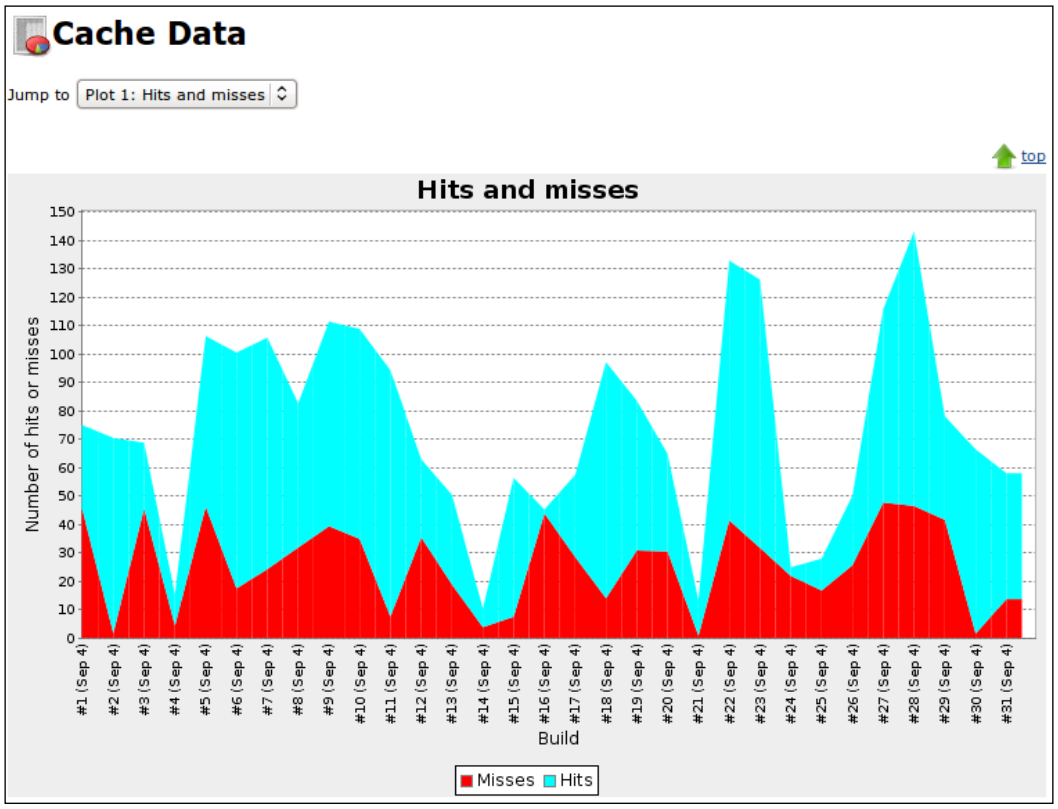
2. Create a freestyle Job with the Job named `ch3.plotting`.
3. In the **Source Code Management** section, check **File System**, adding for **Path** the fully qualified path to your plotting directory, for example `/var/lib/Jenkins/cookbook/ch3.building_software/plotting`.
4. In the **Build** section, add a build step for the **Execute** shell, or in the case of a Windows system, **Execute Windows** batch command.
5. For the command, add `perl hit_and_miss.pl`.
6. In the **Post-build Actions** section, select the **Plot build data** checkbox.
7. Add the following values to the newly expanded region:
   - **Plot group**: `Cache Data`
   - **Plot title**: `Hits and misses`
   - **Plot y-axis label**: `Number of hits or misses`
   - **Plot style**: `Stacked Area`
8. Add a **Data series file** `misses.properties` with **Data series legend label** `Misses`.
9. Add a **Data series file** `hits.properties` with **Data series legend label** `Hits`.

10. At the bottom of the **Configuration** page, click on the **Save** button.



11. Run the job multiple times.

12. Review the **Plot** link.



How it works...

The Perl script generates two property files: `hits` and `misses`. The `hits` file contains `YVALUE` between `0` and `100`, and the `misses` file contains `YVALUE` between `0` and `50`. The numbers are generated randomly. The plot plugin then reads values out of the property `YVALUE`.

The two property files are read by the plot plugin. The plugin keeps track of the history and their values displayed in a trend graph values. You will have to experiment with the different graphic types to find the optimum plot for your custom measurements.

There are currently two other data formats that you can use: **XML** and **CSV**. However, until the online help clearly explains the structures used, I would recommend staying with the properties format.

Perl was chosen for its coding brevity and because it is platform-agnostic. The script could have also been written in Groovy and run from within a Maven project. You can see a Groovy example in the *Running Groovy scripts through Maven* recipe.

There's more...

The plot plugin allows you to choose between a number of plot types, including Area, Bar, Bar 3D, Line, Line 3D, Stacked Area, Stacked Bar, Stacked Bar 3D, and Waterfall. If you choose the right graph type, you can generate beautiful plots.

If you want to add these custom graphs to your reports, you will need to save them. You can do so by right-clicking on the image in your browser.

You may also wish for a different graph size. You can generate an image by visiting the following URL: `http://host/job/JobName/plot/getPlot?index=n&width=x&height=y`.

The width and height define the size of the plot. `n` is an index number pointing to a specific plot. If you have only one plot, then `n` is equal to `0`. If you have two plots configured, then `n` could be either `0` or `1`. To discover the index, visit the plots link, examine the **Jump to** select box, and minus one from the highest Plot number.



To generate a graph in a PNG format having dimensions of `800x600` based on the Job in this recipe, you would use a URL similar to the following:

```
localhost:8080/job/ch3.plotting/plot/getPlot?index=0&width=800&height
=600
```



To download the image without logging in yourself, you can also use the scriptable authentication method mentioned in the recipe *Remotely triggering Jobs*.

See also

- *Running Groovy scripts through Maven*
- *Adaptive site generation*
- *Remotely triggering Jobs through the Jenkins API*

Running Groovy scripts through Maven

This recipe describes how to use the **gmaven** plugin (`http://docs.codehaus.org/display/GMAVEN/Home`) to run Groovy scripts.

The ability to run Groovy scripts in builds allows you to consistently use one scripting language in Maven and Jenkins. Groovy can be run in any Maven phase. Maven can execute the Groovy source code from within the build file, at another file location, or from a remote web server.



Maintainability of scripting

For later re-use, consider centralizing your Groovy code outside the build files.

Getting ready

Create the directory `ch3.building_software/running_groovy`.

How to do it...

1. Add the following fragment just before the `</project>` tag within your template file (mentioned in the introduction), making sure the `pom.xml` file is readable by Jenkins.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.gmaven</groupId>
      <artifactId>gmaven-plugin</artifactId>
      <version>1.3</version>
      <executions><execution>
        <id>run-myGroovy</id>
        <goals><goal>execute</goal></goals>
        <phase>verify</phase>
        <configuration>
          <classpath>
            <element>
              <groupId>commons-lang</groupId>
              <artifactId>commons-lang</artifactId>
              <version>2.6</version>
            </element>
          </classpath>
          <source>
            import org.apache.commons.lang.SystemUtils
            if(!SystemUtils.IS_OS_UNIX)
              { fail("Sorry, Not a UNIX box")}
            def command="ls -l".execute()
            println "OS Type ${SystemUtils.OS_NAME}"
```

```
            println "Output:\n ${command.text}"
          </source>
        </configuration>
      </execution></executions>
    </plugin>
  </plugins>
</build>
```

2. Create a free-style job with the Job name `ch3.groovy_verify`.
3. In the **Source Code Management** section, check **File System**, adding for **Path** the fully qualified path to your plotting directory, for example `/var/lib/jenkins/cookbook/ch3.building_software/running_groovy`.
4. In the **Build** section, add a build step for **Invoke top-level Maven targets**. In the newly expanded section, add:
   - **Maven Version**: `2.2.1`
   - **Goals**: `Verify`
5. Run the Job. If your system is on a `*NIX` box, then similar output will be seen to the following:

```
OS Type Linux
Output:
 total 12
-rwxrwxrwx 1 jenkins jenkins 1165 2011-09-02 14:26 pom.xml
drwxrwxrwx 1 jenkins jenkins 312 2011-09-02 14:53 target
```

   On a Windows system, with Jenkins properly configured, the script will fail with the following message:

```
Sorry, Not a UNIX box
```

How it works...

You can execute the `gmaven-plugin` multiple times during a build. In the example, the phase `verify` is the trigger point.

To enable the Groovy plugin to find the imported classes outside its core features, you will need to add an element in the `<classpath>`. The source code is contained within the `<source>` tag.

```
import org.apache.commons.lang.SystemUtils
if(!SystemUtils.IS_OS_UNIX) { fail("Sorry, Not a UNIX box")}
def command="ls -l".execute()
println "OS Type ${SystemUtils.OS_NAME}"
println "Output:\n ${command.text}"
```

The `import` statement works as the dependency is mentioned in the `<classpath>` tag.

The `systemutils` class (`http://commons.apache.org/lang/api-2.6/org/apache/commons/lang/SystemUtils.html`) provides helper methods, such as the ability to discern which OS you are running, the Java version, and the user's home directory.

The `fail` method allows the Groovy script to fail the build. In this case, when you are not running the build on a `*NIX` OS, then most times you will want your builds to be OS-agnostic. However, during integration testing, you may want to use a specific OS to perform functional tests with a specific web browser. The check will stop the build in case your tests find themselves on the wrong node.



Once you are satisfied with your Groovy code, consider compiling the code into the underlying Java byte code. You can find full instructions at the following URL:

`http://docs.codehaus.org/display/GMAVEN/Building+Groovy+Projects`

There's more...

Here are a number of tips that you might find useful.

Keeping track of warnings

It is important to review your log files, not only on failure but also for the warns. In this case, you will see the two warnings:

```
[WARNING] Using platform encoding (UTF-8 actually) to copy
[WARNING] JAR will be empty - no content was marked for inclusion!
```

The `platform encoding` warning states that the files will be copied using the default platform encoding. If you change the servers and the default encoding on the server is different, then the results of the copying may also be different. For consistency, it is better to enforce a specific coding by adding the lines:

```
<properties>
  <project.build.sourceEncoding>UTF8</project.build.sourceEncoding>
</properties>
```

Please update your template file to take this into account.

The JAR warning is because we are only running a script and have no content to make a JAR. If you had called the script in an earlier phase than the packaging of the JAR, you would not have triggered the warning.

Where's my source?

There are two other ways to point to Groovy scripts to be executed. The first is to point to the file system, for example:

```
<source>${script.dir}/scripts/do_some_good.Groovy</source>
```

The other approach is to connect to a web server through a URL similar to the following:

```
<source>http://localhost/scripts/test.Groovy</source>
```

Using a web server to store Groovy scripts adds an extra dependency to the infrastructure. However, it is also great for centralizing code in an SCM with web access.

Maven phases

Jenkins lumps work together in Jobs. It is coarsely grained for building with pre and post build support. Maven is much more refined, having 21 phases as trigger points. See `http://Maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html`.

Goals bundle phases, for example, for the site goal there are four phases: pre-site, site, post-site, and site-deploy, all of which will be called in order by `mvn site` or directly by using the syntax `mvn site:phase`.

The idea is to chain together a series of lightweight Jobs. You should farm out any heavy Jobs, such as integration tests or a large amount of JavaDoc generation, to a slave node. You should also separate by time to even load and aid in diagnosing issues.

For 2.2.1 and 3.0.3, you can find the XML wiring the lifecycle code at the following URLs:

- `http://svn.apache.org/viewvc/maven/maven-2/tags/maven-2.2.1/maven-core/src/main/resources/META-INF/plexus/components.xml?view=markup`
- `http://svn.apache.org/viewvc/maven/maven-3/tags/maven-3.0.3/maven-core/src/main/resources/META-INF/plexus/components.xml?view=markup`

You will find the Maven phases mentioned in `components.xml` under the line:

```
<!-- START SNIPPET: lifecycle -->
```

Maven plugins bind to particular phases. For site generation, the `<reporting>` tag surrounds the majority of the configuration. The plugins configured under reporting generate useful information, whose results are saved under the `target/site` directory. There are a number of plugins that pick up the generated results and then plot their history. Jenkins plugins, in general, do not perform the tests, but consume the results. There are exceptions, such as the **sloccount** plugin (`https://wiki.jenkins-ci.org/display/JENKINS/SLOCCount+Plugin`) and the **task scanner** plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Task+Scanner+Plugin`). These differences will be explored later in *Chapter 5*, *Using Metrics to Improve Quality*.

The Groovy plugin is useful in all the phases as it is not specialized to any specific task, such as packaging or deployment. It gives you a uniform approach to reacting to situations that are outside the common functionality of Maven.



The differences between Maven versions

There are differences between the behavior of Maven 2.2 and 3, especially around site generation. They are summarized at `https://cwiki.apache.org/MAVEN/Maven-3x-compatibility-notes.html`.

You will find the plugin compatibility list at `https://cwiki.apache.org/MAVEN/Maven-3x-plugin-compatibility-matrix.html`.

See also

- *Running AntBuilder through Groovy in Maven*
- *Reacting to the generated data with the the Post-build Groovy plugin*
- *Adaptive site generation*

Manipulating environmental variables

This recipe shows you how to pass variables from Jenkins to your build Job and how different variables are overwritten. It also describes one way of failing the build if crucial information has not been correctly passed.

Jenkins has a number of plugins for passing information to builds including the **Setenv** plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Setenv+Plugin`), **Envfile** plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Envfile+Plugin`), and the **EnvInject** plugin (`https://wiki.jenkins-ci.org/display/JENKINS/EnvInject+Plugin`). The Envinject plugin was chosen for this recipe as it is reported to work with nodes and offers a wide range of property injection options.

Getting ready

Install the Envinject plugin. Create the recipe directory named.

How to do it...

1. Create a `pom.xml` file that is readable by Jenkins, with the following contents:

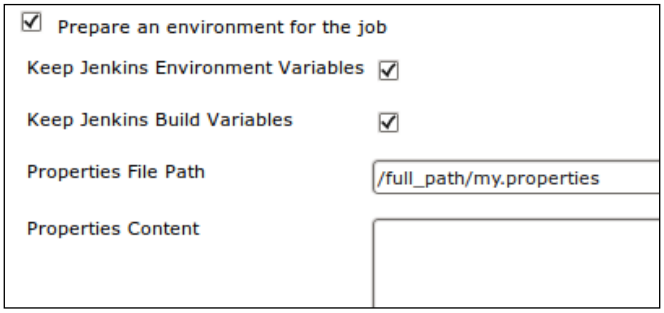
```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.berg</groupId>
  <artifactId>ch3.jenkins.builds.properties</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>${name.from.jenkins}</name>
  <properties>
    <project.build.sourceEncoding>
      UTF8
    </project.build.sourceEncoding>
  </properties>
  <build>
    <plugins><plugin>
      <groupId>org.codehaus.gmaven</groupId>
      <artifactId>gmaven-plugin</artifactId>
      <version>1.3</version>
      <executions><execution>
        <id>run-myGroovy</id>
        <goals><goal>execute</goal></goals>
        <phase>verify</phase>
        <configuration>
          <source>
            def environment = System.getenv()
            println "----Environment"
            environment.each{ println it }
            println "----Property"
            println(System.getProperty("longname"))
            println "----Project and session"
            println "Project: ${project.class}"
            println "Session: ${session.class}"
            println "longname: ${project.properties.longname}"
            println "Project name: ${project.name}"
            println "JENKINS_HOME:
              ${project.properties.JENKINS_HOME}"
```

```
          </source>
        </configuration>
      </execution></executions>
    </plugin></plugins>
  </build>
  </project>
```

2. Create a file named `my.properties` and place it in the same directory as the `pom.xml` file, adding the following contents:

```
project.type=prod
secrets.file=/etc/secrets
enable.email=true
JOB_URL=I AM REALLY NOT WHAT I SEEM
```

3. Create a blank free-style Job with the Job name `ch3.environment`.
4. In the **Source Code Management** section, check **File System**, adding for **Path** the fully qualified path to your directory, for example: `/var/lib/jenkins/cookbook/ch3.building_software/environment`
5. In the **Build** section, add a build step for **Invoke top-level Maven targets**. In the newly expanded section, add:
   - **Maven Version**: `2.2.1`
   - **Goals**: `Verify`
6. Click on the **Advanced** button, and add **Properties**: `longname=SuperGood`.
7. Inject the values in `my.properties`, by clicking on the **Prepare an environment for the job** (near the top of the **Job configuration** page).
8. For **Properties File Path**, add `/full_path/my.properties`, for example: `/home/var/lib/cookbook/ch3.building_software/environment/my.properties`.



9. Run the job. The build will fail with an output similar to the following:

```
----Project and session
Project: class org.apache.Maven.model.Model
Session: class org.apache.Maven.execution.MavenSession
longname: SuperGood
[INFO] ------------------------------------------------------------
[ERROR] BUILD ERROR
[INFO] ------------------------------------------------------------
[INFO] Groovy.lang.MissingPropertyException: No such property:
name for class: script1315151939046
```

10. In the **Build** section for **Invoke top-level Maven targets**, click on the **Advanced** button. In the newly expanded section, add an extra property `name.from.jenkins=The build with a name`
11. Run the Job. It should now succeed.

How it works...

The **Envinject** plugin is useful for injecting properties into builds.

In the recipe, Maven is run twice. The first time when it is run without the variable `name.from.jenkins` defined, the Jenkins Job fails. The second time when it is run with the variable defined, the Jenkins job succeeds.

Maven expects that the variable `name.from.jenkins` is defined, or the name of the project will also not be defined. Normally, this would not be enough to stop your job from succeeding. However, when running the Groovy code, the line `println "Project name: ${project.name}"` specifically the call `project.name` will fail the build. This is great for protecting against missing property values.

The Groovy code can see instances of the project `org.apache.Maven.model.Model` and session class `org.apache.Maven.execution.MavenSession`. The project instance is a model of the XML configuration that you can programmatically access. You can get the property `longname` by referencing it through `project.properties.longname`. Your Maven goal will fail if the property does not exist. You can also get at the property through the call `System.getProperty("longname"))`. However, you cannot get to the property by using the environment call `System.getenv()`.

It is well worth learning the various options, they are:

- **Keep Jenkins Environment Variables** and **Keep Jenkins Build Variables**: Both these options affect which Jenkins-related variables your Job sees. It is good to keep your environment as clean as possible, as it will aid you in debugging later.
- **Properties Content**: You can override specific values in the properties files.

- **Environment Script File Path**: It points to a script that will set up your environment. This is useful if you want to detect specific details of the running environment and configure your build accordingly.
- **Populate build cause**: You enable Jenkins to set the `BUILD_CAUSE` environment variable. This variable contains information about which event triggered the job.

There's more...

Maven has a plugin for reading properties (`http://mojo.codehaus.org/properties-maven-plugin/`). To choose between property files, you will need to set a variable in the plugin configuration, and call it as part of the Jenkins Job. For example:

```
<build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>properties-maven-plugin</artifactId>
        <version>1.0-alpha-2</version>
        <executions>
          <execution>
            <phase>initialize</phase>
            <goals>
              <goal>read-project-properties</goal>
            </goals>
            <configuration>
              <files>
                <file>${fullpath.to.properties}</file>
              </files>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
```

If you use a relative path to the properties file, then the file can reside in your Revision control system. If you use a full path, then the property file can be stored on the Jenkins server. The second option is preferable if sensitive passwords, such as for database connections, are included.

Jenkins has the ability to ask for variables when you run a Job manually. This is called a **Parameterized build** (`https://wiki.jenkins-ci.org/display/JENKINS/Parameterized+Build`). At the build time, you can choose your property files by selecting from a choice of property file locations.

See also

- *Running AntBuilder through Groovy in Maven*

Running AntBuilder through Groovy in Maven

Jenkins interacts with an audience with a wide technological background. There are many developers who became proficient in Ant scripting before moving on to using Maven. Developers may be happier with writing an Ant task than editing a `pom.xml` file. There are mission-critical Ant scripts that still run in a significant proportion of organizations.

In Maven, you can run Ant tasks directly with the `ant-run` plugin (`http://maven.apache.org/plugins/maven-antrun-plugin/`) or through Groovy (`http://docs.codehaus.org/display/GROOVY/Using+Ant+from+Groovy`). `Antrun` represents a natural migration path. This is the path of least initial work.

The Groovy approach makes sense for the Jenkins administrators who use Groovy as part of their tasks. Groovy, being a first-class programming language, has a wide range of control structures that are hard to replicate in Ant. You can partially do this by using the `Ant-contrib` library (`http://ant-contrib.sourceforge.net`). However, Groovy, being a mature programming language, is much more expressive.

This recipe details how you can run two Maven pom's involving Groovy and Ant. The first pom shows you how to run the simplest of the Ant tasks within Groovy, and the second performs an Ant contrib task to secure copy files from a large number of computers.

Getting ready

Create the directory named `ch3.building_software/antbuilder`.

How to do it...

1. Create a template file, and call it `pom_ant_simple.xml`.
2. Change the values of `groupId`, `artifactId`, `version`, and `name` to suit your preferences.
3. Add the following XML fragment just before the `</project>` tag:

```
<build>
  <plugins><plugin>
    <groupId>org.codehaus.gmaven</groupId>
    <artifactId>gmaven-plugin</artifactId>
    <version>1.3</version>
```

```
    <executions>
      <execution>
        <id>run-myGroovy-test</id>
        <goals><goal>execute</goal></goals>
        <phase>test</phase>
        <configuration>
          <source>
            def ant = new AntBuilder()
            ant.echo("\n\nTested ----> With Groovy")
          </source>
        </configuration>
      </execution>
      <execution>
        <id>run-myGroovy-verify</id>
        <goals><goal>execute</goal></goals>
        <phase>verify</phase>
        <configuration>
          <source>
            def ant = new AntBuilder()
            ant.echo("\n\nVerified at ${new Date()}")
          </source>
        </configuration>
      </execution>
    </executions>
  </plugin></plugins>
</build>
```

4. Run `mvn test`. Review the output. Notice that there are no warnings about the empty JAR files.

5. Run `mvn` verify. Review the output. It should look similar to the following:

```
[INFO] [surefire:test {execution: default-test}]
[INFO] No tests to run.
[INFO] [Groovy:execute {execution: run-myGroovy-test}]
     [echo]
     [echo]
     [echo] Tested ----> With Groovy
[INFO] [jar:jar {execution: default-jar}]
[WARNING] JAR will be empty - no content was marked for inclusion!
[INFO] Building jar:ch3.jenkins.builds-1.0-SNAPSHOT.jar
[INFO] [Groovy:execute {execution: run-myGroovy-verify}]
     [echo]
     [echo]
     [echo] Verified at Fri Sep 16 11:25:53 CEST 2011
[INFO] [install:install {execution: default-install}]
```

```
[INFO] Installing /target/ch3.jenkins.builds-1.0-SNAPSHOT.jar to
ch3.jenkins.builds/1.0-SNAPSHOT/ch3.jenkins.builds-1.0-SNAPSHOT.
jar
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------
[INFO] Total time: 4 seconds
```

6. Create a second template file named `pom_ant_contrib.xml`.
7. Change the values of `groupId`, `artifactId`, `version`, and `name` to suit your preferences.
8. Add the following XML fragment just before the `</project>` tag:

```
<build>
  <plugins><plugin>
    <groupId>org.codehaus.gmaven</groupId>
    <artifactId>gmaven-plugin</artifactId>
    <version>1.3</version>
    <executions><execution>
      <id>run-myGroovy</id>
      <goals><goal>execute</goal></goals>
      <phase>verify</phase>
      <configuration>
        <source>
          def ant = new AntBuilder()
          host="Myhost_series"
          print "user: "
          user = new String(System.console().readPassword())
          print "password: "
          pw = new String(System.console().readPassword())

          for ( i in 1..920) {
            counterStr=String.format('%02d',i)

            ant.scp(trust:'true',file:"${user}:
              ${pw}${host}${counterStr}:
              /${full_path_to_location}",

            localTofile:"${myfile}-${counterStr}", verbose:"true")
          }
        </source>
      </configuration>
    </execution></executions>
    <dependencies>
      <dependency>
```

```
            <groupId>ant</groupId>
            <artifactId>ant</artifactId>
            <version>1.6.5</version>
          </dependency>
          <dependency>
            <groupId>ant</groupId>
            <artifactId>ant-launcher</artifactId>
            <version>1.6.5</version>
          </dependency>
          <dependency>
            <groupId>ant</groupId>
            <artifactId>ant-jsch</artifactId>
            <version>1.6.5</version>
          </dependency>
          <dependency>
            <groupId>com.jcraft</groupId>
            <artifactId>jsch</artifactId>
            <version>0.1.42</version>
          </dependency>
        </dependencies>
      </plugin></plugins>
    </build>
```



The code is only representative. To make it work, you will have to configure it to point to the files on real servers.

How it works...

Groovy runs basic Ant tasks without the need of extra dependencies. An AntBuilder instance (`http://Groovy.codehaus.org/Using+Ant+Libraries+with+AntBuilder`) is created, and then the Ant `echo` task is called. Under the bonnet, Groovy calls the Java classes that Ant uses to perform the `echo` command. Within the `echo` command, a date is printed by directly creating an anonymous object, for example:

```
ant.echo("\n\nVerified at ${new Date()}").
```

You configured the `pom.xml` to fire off the Groovy scripts in two phases: the **test phase** and then later in the **verify phase**. The test phase occurs before the generation of a JAR file, and thus avoids creating a warning about an empty JAR. As the name suggests, this phase is useful for testing before packaging.

The second example script highlights the strength of combining Groovy with Ant. The SCP task (`http://Ant.apache.org/manual/Tasks/scp.html`) is run a large number of times across many servers. The script first asks for the USERNAME and password, avoiding storage on your file system or your revision control system. The Groovy script expects you to inject the variables host: `full_path_to_location` and `myfile`.

Notice the similarity between the Ant SCP task and the way it is expressed in the `pom_ant_contrib.xml` file.

There's more...

Creating custom property files on the fly allows you to pass on information from one Jenkins Job to another.

You can create property files through AntBuilder using the `echo` task. The following code creates a file named `value.properties` with two lines: `x=1` and `y=2`.

```
def ant = new AntBuilder()
ant.echo(message: "x=1\n", append: "false", file:
  "values.properties")
ant.echo(message: "y=2\n", append: "true", file: "values.properties")
```

The first `echo` command sets `append` to `false` so that every time a build occurs, a new `properties` file is created. The second `echo` appends its message.



You can remove the second append `attribute` as the default value is set to `true`.

See also

- *Running Groovy scripts through Maven*

Failing Jenkins Jobs based on JSP syntax errors

Java Server Pages (`http://www.oracle.com/technetwork/java/overview-138580.html`) is a standard that makes the creation of simple web applications straightforward. You write HTML, such as pages, with extra tags interspersed with Java coding into a text file. If you do this in a running web application, then the code recompiles on the next page call. This process supports **Rapid Application Development** (**RAD**), but the risk is that developers make messy and hard-to-read JSP code that is difficult to maintain. It would be nice if Jenkins could display metrics about the code to defend the quality.

JSP pages are compiled on the fly for the first time when a user request for the page is received. The user will perceive this as a slow loading of the page, and this may deter them from future visits. To avoid this situation, you can compile the JSP page during the build process, and place the compiled code in the `WEB-INF/classes` directory or packaged in the `WEB-INF/lib` directory of your web app. This approach has the advantage of a faster first page load.

A secondary advantage of having a compiled source code is that you can run a number of statistic code review tools over the code base and obtain testability metrics. This generates the testing data that is ready for Jenkins plugins to display.

This recipe describes how to compile JSP pages based on the `maven-jetty-jspc-plugin` plugin (`http://jetty.codehaus.org/jetty/jspc-maven-plugin/`). The compiled code will work with the Jetty server, which is often used for integration tests.



Warning: The JSP mentioned in this recipe is deliberately insecure, ready for testing later in this book.



Tomcat specific precompiling and deploying

A complementary plugin specifically for Tomcat deployment is the `tomcat-maven-plugin` plugin (`http://tomcat.apache.org/Maven-plugin.html`).

Getting ready

Create the directory named `ch3.building_software/jsp_example`.

How to do it...

1. Create a war project from a Maven archetype by typing the following command:
   ```
   mvn archetype:generate
   ```
2. Choose `Maven-archetype-webapp (142)`:
3. Enter the values:
   - **groupId**: `ch3.packt.builds`
   - **artifactId**: `jsp_example`
   - **version**: `1.0-SNAPSHOT`
   - **package**: `ch3.packt.builds`
4. Press *Enter* to confirm the values.

5. Edit the `jsp_example/pom.xml` file, adding the following build section:

```
<build>
  <finalName>jsp_example</finalName>
  <plugins>
    <plugin>
      <groupId>org.mortbay.jetty</groupId>
      <artifactId>maven-jetty-jspc-plugin</artifactId>
      <version>6.1.14</version>
      <executions>
        <execution>
          <id>jspc</id>
          <goals><goal>jspc</goal></goals>
          <configuration></configuration>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-war-plugin</artifactId>
      <configuration>
        <webXml>${basedir}/target/web.xml</webXml>
      </configuration>
    </plugin>
  </plugins>
</build>
```

6. Replace the `src/main/webapp/index.jsp` file with the following code:

```
<html>
  <head>

    <meta http-equiv="Content-Type" content="text/html;
      charset=UTF-8">

    <title>Hello World Example</title>
  </head>
  <body>
    <%
      String evilInput= null;
      evilInput = request.getParameter("someUnfilteredInput");
      if (evilInput==null){evilInput="Hello Kind Person";}
    %>
    <form action="index.jsp">
      The big head says: <%=evilInput%><p>
```

```
        Please add input:<input type='text'
        name='someUnfilteredInput'>
        <input type="submit">
      </form>
    </body>
  </html>
```

7. Create a WAR file by using the command `mvn package`.
8. Modify `./src/main/webapp/index.jsp` so that it is no longer a valid JSP file, by adding the line `if;` underneath the line starting with `if (evilInput==null)`.
9. Run the command `mvn package`. The build will now fail with an understandable error message similar to the following:

```
[INFO] : org.apache.jasper.JasperException: PWC6033: Unable to
compile class for JSP
PWC6197: An error occurred at line: 4 in the jsp file: /index.jsp
PWC6199: Generated servlet error:
Syntax error on token "if", delete this token


Failure processing jsps
```

How it works...

The Maven plugin seeing the `index.jsp` page compiles it into a class with the name `jsp.index_jsp`, placing the compiled class under `WEB-INF/classes`. The plugin then defines the class as a servlet in `WEB-INF/web.xml` with a mapping to `/index.jsp`. For example:

```
<servlet>
  <servlet-name>jsp.index_jsp</servlet-name>
  <servlet-class>jsp.index_jsp</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>jsp.index_jsp</servlet-name>
  <url-pattern>/index.jsp</url-pattern>
</servlet-mapping>
```

There's more...

Here are a few things that you should consider.

Different server types

By default, the Jetty Maven plugin (version 6.1.14) loads JSP2.1 libraries with JDK1.5. This will not work for all server types. For example, if you deploy the `.war` file generated by this recipe to a Tomcat 7 server, it will fail to deploy properly. If you look in the `logs/catalina.out`, you will see the following error:

```
javax.servlet.ServletException: Error instantiating servlet class jsp.
index_jsp
Root Cause
java.lang.NoClassDefFoundError: Lorg/apache/jasper/runtime/
ResourceInjector;
```

This is because different servers have different assumptions about how the JSP code is compiled and which libraries they depend on to run. For Tomcat, you will need to tweak the compiler used and the Maven plugin dependencies. For more details, review the following link: `http://docs.codehaus.org/display/JETTY/Maven+Jetty+Plugin`.

Eclipse templates for JSP pages

Eclipse is a popular open source IDE for Java developers (`http://www.eclipse.org/`). If you are using Eclipse with its default template for the JSP pages, then your pages may fail to compile. This is because the default compiler does not like the `meta` information mentioned before the `<html>` tag. For example:

```
<%@ page language="java" contentType="text/html;charset=UTF-8"
  pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
```

Simply remove the lines before compiling, or change the JSP compiler that you use.

See also

- *Configuring Jetty for integration tests*

Configuring Jetty for integration tests

Jenkins plugins that keep a history of tests are normally consumers of the data generated within Maven builds. For Maven to automatically run integration, performance, or functional tests, it will need to hit a live test server. You have two main choices:

1. Deploy your artifacts, such as `.war` files, to a live server. You can do this using the `Maven-wagon` plugin (`http://mojo.codehaus.org/wagon-maven-plugin/`), or through a Jenkins plugin, such as an aptly named deploy plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Deploy+Plugin`).
2. Run the lightweight Jetty server within the build. This simplifies your infrastructure. However, the server will be run as part of a Jenkins Job, consuming the potentially scarce resources. This will limit the number of parallel executors that Jenkins can run, decreasing the maximum throughput of Jobs.

This recipe runs the web application developed in the recipe named *Failing Jenkins Jobs based on JSP syntax errors*, tying Jetty into integration testing by bringing the server up just before tests are run, and then shutting down afterwards. The build creates a self-signed certificate. Two Jetty connectors are defined for HTTP and for the secure TLS traffic. To create a port to telnet, the `shutdown` command is also defined.

Getting ready

Follow the recipe *Failing Jenkins Jobs based on JSP syntax errors*, generating a `.war` file. Copy the project to the directory named `ch3.building_software/jsp_jetty`.

How to do it...

1. Add the following XML fragment just before `</plugins>` tag within the `pom.xml` file.

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>keytool-maven-plugin</artifactId>
  <executions>
    <execution>
      <phase>generate-resources</phase>
      <id>clean</id>
      <goals>
        <goal>clean</goal>
      </goals>
    </execution>
    <execution>
      <phase>generate-resources</phase>
```

```
            <id>genkey</id>
            <goals>
              <goal>genkey</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <keystore>
            ${project.build.directory}/jetty-ssl.keystore
          </keystore>
          <dname>cn=HOSTNAME</dname>
          <keypass>jetty8</keypass>
          <storepass>jetty8</storepass>
          <alias>jetty8</alias>
          <keyalg>RSA</keyalg>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>jetty-maven-plugin</artifactId>
        <version>8.0.0.M0</version>
        <configuration>
          <webApp>${basedir}/target/jsp_example.war</webApp>
          <stopPort>8083</stopPort>
          <stopKey>stopmeplease</stopKey>
          <connectors>
            <connector implementation=
               "org.eclipse.jetty.server.nio.SelectChannelConnector">
             <port>8082</port>
            </connector>
            <connector implementation=
              "org.eclipse.jetty.server.ssl.SslSocketConnector">
              <port>9443</port>
              <keystore>
                ${project.build.directory}/jetty-ssl.keystore
              </keystore>
              <password>jetty8</password>
              <keyPassword>jetty8</keyPassword>
            </connector>
          </connectors>
        </configuration>
        <executions>
          <execution>
            <id>start-jetty</id>
```

```
        <phase>pre-integration-test</phase>
        <goals>
          <goal>run</goal>
        </goals>
        <configuration>
          <daemon>true</daemon>
        </configuration>
      </execution>
      <execution>
        <id>stop-jetty</id>
        <phase>post-integration-test</phase>
        <goals>
          <goal>stop</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
```

2. Run the command `mvn jetty:run`. You will now see the console output from the Jetty server starting up.
3. Using a web browser, visit the location `https://localhost:9443`. After passing through the warnings about the self-signed certificate, you will see the web application working.
4. Press *Ctrl+C* to stop the server.
5. Run `mvn verify`. You will now see the server starting up, and then stopping.

How it works...

Within the `<executions>` tag, Jetty is run in the Maven **pre-integration-test** phase and later stopped in the Maven **post-integration-test** phase. In the **generate-resources** phase, Maven uses the `keytool` plugin to create a self-signed certificate. The certificate is stored in a Java key store with a known password and alias. The key encryption is set to `RSA` (`http://en.wikipedia.org/wiki/RSA_%28algorithm%29`). If the **Common Name** (**CN**) is not correctly set in your certificate, then your web browser will complain about the certificate. To change the **Distinguished Name** (**DN**) of the certificate to the name of your host, modify `<dname>cn=HOSTNAME</dname>`.

Jetty is configured with two connector types: port `8082` for HTTP and port `9443` for secure connections. These ports are chosen as they are above port `1023`, so that you do not need administrative rights to run the build. The port numbers also avoid the ports used by Jenkins. Both the Jetty and Keytool plugin use the `keystore` tag to define the location of the key store.

Using self-signed certificates causes extra work for functional testers. Every time they encounter a new version of the certificate, they will need to accept, in their web browser, the certificate as a security exception. It is better to use certificates from well-known authorities. You can achieve it with this recipe by removing the key generation and pointing the `keystore` tag to a known file location.

The generated `.war` file is pointed to by the `webapp` tag, and Jetty runs the application.

There's more...

Maven 3 is fussier about defining plugin versions than Maven 2.2.1. There are good reasons for this. If you know that your build works well with a specific version of a Maven, then this defends against unwanted changes. For example, the Jetty plugin used in this recipe is held at version 8.0.0.M0. As you can see from the bug report (`http://jira.codehaus.org/browse/JETTY-1071`), configuration details have changed over versions.

Another advantage is that if the plugin version is too old, then the plugin will be pulled out of the central plugin repository. When you next clean up your local repository, this will break your build. This is what you want, as this clearly signals the need to review and then upgrade.

See also

- *Failing Jenkins Jobs based on JSP syntax errors*

Looking at license violations with RATs

This recipe describes how to search any Job in Jenkins for license violations. It is based on the Apache RATs project (`http://incubator.apache.org/rat`). You can search for license violations by running a RAT JAR file directly, with a contributed ANT task or through Maven. In this recipe, you will be running directly through a `.jar` file. The report output goes to the console, ready for Jenkins plugins like the log parser plugin to process the information.

Getting ready

Log in to Jenkins.

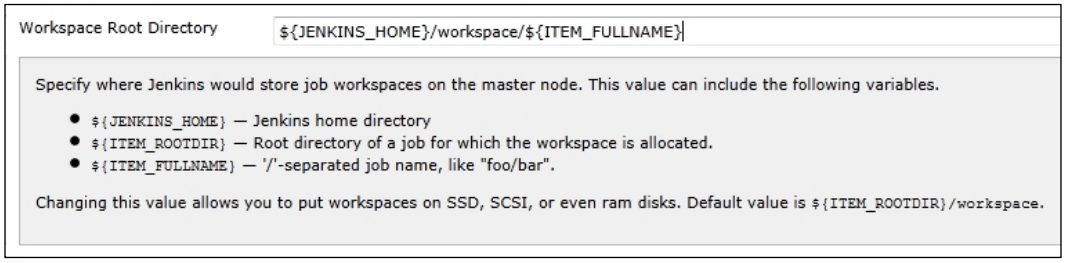
How to do it...

1. Create a free-style Job named `License_Check`.
2. Under the **Source Code Management**, check **Subversion**.

3. Fill in `http://svn.apache.org/repos/asf/incubator/rat/main/trunk` for the URL.
4. Set **Check-out Strategy** to **Use 'svn update' as much as possible**.
5. Under the Post steps section, check Run only if build succeeds.
6. Add a **Post-build** step for **Execute Shell** (we assume that you are running a NIX system). Add the following text to the **Execute Shell** text area, replacing the `jar` version with the correct value.

```
java -jar ./apache-rat/target/apache-rat-0.8-SNAPSHOT.jar --help

java -jar ./apache-rat/target/apache-rat-0.8-SNAPSHOT.jar -d
${JENKINS_HOME}/jobs/License_Check/workspace  -e '*.js'  -e
'*target*'
```

7. Press the **Save** button.
8. Run the Job.
9. Review the path to the workspace of your Jobs. Visit the **Configure Jenkins** screen, for example `http://localhost:8080/configure`. Just under the **Home Directory**, press the **Advance** button. The **Workspace Root Directory** values become visible.



How it works...

The RATs source code is compiled and run twice: the first time to print the help out, and the second time to check the license headers.

The code base is changing, and over time it expects the number of options to increase. You will find the most up-to-date information by running `help`.

The `-d` option tells the application in which directory is your source code. In this example, you have used the variable `${JENKINS_HOME}` to define the top level of the path. Next, we assume that the job is found under the `./job/jobname/workspace` directory. You checked that this assumption is true in *step 9* of the recipe. If incorrect, you will need to adjust the option. To generate a report for another project, simply change the path by replacing the Job name.

The `-e` option excludes certain file name patterns from review. You have excluded JavaScript files `*.js` and `*target*` for all the generated files under the target directory. In a complex project, expect a long list of exclusions.



Warning: Even If the directory to check does not exist, the build will still succeed with an error reported similar to:

ERROR: /var/lib/Jenkins/jobs/License_Check/workspace

Finished: Success

You will have to use a log parsing plugin to force failure.

There's more...

A complimentary Maven plugin for updating licenses in source code is the `Maven-license-Plugin` plugin (`http://code.google.com/p/maven-license-plugin`). You can use it to keep your source code license headers up to date. To add/update the source code with the license `src/etc/header.txt`, add the following XML fragment to your build section.

```
<plugin>
     <groupId>com.mycila.maven-license-plugin</groupId>
     <artifactId>maven-license-plugin</artifactId>
     <configuration>
           <header>src/etc/header.txt</header>
     </configuration>
</plugin>
```

You will then need to add your own license file: `src/etc/header.txt`.

A powerful feature is that you can add variables to expand. In the following example, `${year}` will get expanded.

```
Copyright (C) ${year} Licensed under this open source License
```

To format your source code, you would then run the following command:

```
mvn license:format -Dyear=2012
```

See also

- *Reviewing license violations from within Maven*
- *Reacting to the generated data with the Post-build Groovy plugin*

Reviewing license violations from within Maven

Getting ready

Create the directory named `ch3.building_software/license_maven`.

How to do it...

1. Create a template `pom.xml` file.
2. Change the values of `groupId`, `artifactId`, `version`, and `name` to suit your preferences.
3. Add the following XML fragment just before the `</project>` tag:

```
<pluginRepositories>
  <pluginRepository>
    <id>apache.snapshots</id>
    <url>http://repository.apache.org/snapshots/</url>
    </pluginRepository>
  </pluginRepositories>
<build>
<plugins><plugin>
  <groupId>org.apache.rat</groupId>
  <artifactId>apache-rat-plugin</artifactId>
  <version>0.8-SNAPSHOT</version>
  <executions><execution>
    <phase>verify</phase>
    <goals><goal>check</goal></goals>
    </execution></executions><configuration>
      <excludeSubProjects>false</excludeSubProjects>
      <numUnapprovedLicenses>97</numUnapprovedLicenses>
      <excludes>
        <exclude>**/.*/**</exclude>
        <exclude>**/target/**/*</exclude>
      </excludes>
      <includes>
        <include>**/src/**/*.css</include>
        <include>**/src/**/*.html</include>
        <include>**/src/**/*.java</include>
        <include>**/src/**/*.js</include>
        <include>**/src/**/*.jsp</include>
        <include>**/src/**/*.properties</include>
```

```
            <include>**/src/**/*.sh</include>
            <include>**/src/**/*.txt</include>
            <include>**/src/**/*.vm</include>
            <include>**/src/**/*.xml</include>
          </includes>
        </configuration>
      </plugin></plugins>

</build>
```

4. Create a Maven 2/3 project with project name `ch3.BasicLTI_license`.
5. Under the **Source Code Management** section, tick **Subversion** with the URL repository `https://source.sakaiproject.org/svn/basiclti/trunk`.



Warning: Please do not spam the **Subversion** repository. Double-check that there are no build triggers activated.

6. Under the `Build` section, set:
   - **Root POM**: `pom.xml`
   - **Goals and options**: `clean`
7. Under the **Post Steps** section, invoke the top-level Maven targets:
   - **Maven Version**: `2.2.1`
   - **Goals**: `verify`
8. Click on the **Advanced** button.
9. In the expanded section, set:
   - **POM**: full path to your RATs pom file, for example:
   - `/var/lib/cookbook/ch3.building_software/license_maven / pom.xml`
   - **Properties**: `rat.basedir=${WORKSPACE}`
10. Under the **Post Steps** section, add to the `Execute` shell a copy command to move the report into your workspace, for example:

```
cp /var/lib/cookbook/ch3.building_software/license_/target/rat.txt
${WORKSPACE}
```

11. Run the Job. You can now visit the Workspace and view `./target/rat.txt`. The file should look similar to the following:

```
Summary
-------
Notes: 0
Binaries: 0
Archives: 0
Standards: 128

Apache Licensed: 32
Generated Documents: 0

JavaDocs are generated and so license header is optional
Generated files do not required license headers

96 Unknown Licenses
```

How it works...

You have pulled the source code from an open source project; in this case, from the subversion repository of the **Sakai Foundation** (`www.sakaiproject.org`).

Background information: Sakai is a **Learning Management System** (**LMS**) that is used daily by millions of students. The Sakai Foundation represents over 80 organizations, mostly universities.

The source code includes different licenses, which are checked by the RATs Maven plugin. The plugin is called during the verify phase and checks the workspace location of your Job as defined by the `${WORKSPACE}` variable that Jenkins injected.

`excludeSubProjects` set to `false` tells RATs to visit any subproject as well as the master project. `numUnapprovedLicenses` is the number of unapproved licenses that are acceptable before your Job fails.

The `excludes` exclude the target directory and any other directory. The `includes` override specific file types under the `src` directory. Depending on the type of frameworks used in your projects, the range of includes will change.



For information on customizing RATs for specific license types, visit:
`http://incubator.apache.org/rat/apache-rat-plugin/examples/custom-license.html`

There's more...

Here are a few more useful tips to review:

Multiple approaches and anti-patterns

There were multiple approaches to configuring the Jenkins Job. You could avoid copying the RATs report file by fixing its location in the Maven plugins configuration. This has the advantage of avoiding a copying action. You could also use the **Multiple SCM** plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Multiple+SCMs+Plugin`) to first copy the source code into the workspace. You should also consider splitting into two Jobs, and then pointing the RATs Job at the source codes workspace. The last approach is a best practice, as it cleanly separates the testing.

Snapshots

Unlike fixed versions of artifacts, snapshots have no guarantee that their details will not vary over time. **Snapshots** are useful if you want to test the latest and greatest features. However, for the most maintainable code, it is much better to use fixed versions.

To defend the base-level stability, consider writing a Job that triggers a small Groovy script inside a `pom.xml` to visit all your projects. The script needs to search for the word `SNAPSHOT` in the `version` tag, and then write a recognizable warning for the Post-build Groovy plugin to pick up, and if necessary, fail the Job. Using this approach, you can incrementally tighten the boundaries, giving developers time to improve their builds.

See also

- *Looking at license violations with RATs*
- *Reacting to the generated data with the Post-build Groovy plugin*

Exposing information through build descriptions

The **setter** plugin allows you to gather information out of the build log and add it as a description to a build's history. This is useful as it allows you later to quickly assess the historic cause of the issue, without drilling down into the console output. This saves many mouse clicks. You can now see the details immediately in the **trend** report without needing to review all the build results separately.

The setter plugin uses the Regex expressions to scrape the descriptions. This recipe shows you how.

Getting ready

Install the Description Setter plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Description+Setter+Plugin`). Create a directory for the recipe files named `ch3.building_software/descriptions`.

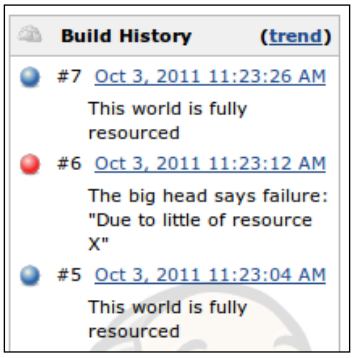
How to do it...

1. Create a template `pom.xml` file.
2. Change the values of `groupId`, `artifactId`, `version`, and `name` to suit your preferences.
3. Add the following XML fragment just before the `</project>` tag:

```
<build>
  <plugins><plugin>
    <groupId>org.codehaus.gmaven</groupId>
    <artifactId>gmaven-plugin</artifactId>
    <version>1.3</version>
    <executions><execution>
      <id>run-myGroovy</id>
      <goals><goal>execute</goal></goals>
      <phase>verify</phase>
      <configuration>
        <source>
          if ( new Random().nextInt(50) > 25){
            fail "MySevere issue:  Due to little of resource X"
          } else {
            println "Great stuff happens because: This world is
              fully resourced"
          }
        </source>
```

```
        </configuration>
      </execution></executions>
    </plugin></plugins>
  </build>
```

4. Create a Maven 2/3 project with a Job named `ch3.descriptions`.
5. In the **Source Code Management** section, check **File System**, adding for **Path** the fully qualified path to your directory, for example: `/var/lib/Jenkins/cookbook/ch3.building_software/description`.
6. Tick **Set build Description**, and add the following values to the expanded options.
   - **Regular expression**: `Great stuff happens because: (.*)`
   - **Regular expression for failed builds**: `MySevere issue: (.*)`
   - **Description for failed builds**: `The big head says failure: "\1"`
7. Run the Job a number of times, and review the build history. You will see that the description of each build varies.



How it works...

The Groovy code is called as part of the install goal. The code either fails the Job with the pattern `MySevere Issue` or prints the output to the build with the pattern `Great stuff happens because`.

```
if ( new Random().nextInt(50) > 25){
    fail "MySevere issue:  Due to little of resource X"
} else {
    println "Great stuff happens because: This world is fully
resourced"
```

As a Post-build action, the **description setter** plugin is triggered. On success of the build, it looks for the pattern `Great stuff happens because: (.*)`.

`(.*)` pulls in any text after the first part of the pattern into the variable `"\1"`, which is later expanded in the setting of the description of the specific build.

The same is true for the failed build, apart from some extra text that is added before the expansion of `"\1"`. You defined this in the configuration of Description for failed builds.

It is possible to have more variables than just `\1` by expanding the **regex expressions**. For example, if the console output was `fred is happy`, then the pattern `(.*) is (.*)` generates `"\1" equal to fred and "\2" equal to happy`.

There's more...

The plugin gets its ability to parse the text from the **Token Macro** plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Token+Macro+Plugin`). The token macro plugin allows the macros to be defined in the text, which are then expanded by calling a **utility** method. This approach of using utility plugins simplifies the plugin creation and supports consistency.

See also

- *Reacting to the generated data with the Post-build Groovy plugin*

Reacting to the generated data with the Post-build Groovy plugin

Build information is sometimes left obscure in log files or reports that are difficult for Jenkins to expose. This recipe will show you one approach of pulling those details into Jenkins.

The Post-build Groovy plugin allows you to run Groovy scripts after the build has run. Since the plugin runs within Jenkins, it has a programmatic access to services, such as being able to read console input or change a builds summary page.

This recipe uses a Groovy script within a Maven `pom.xml` file to output a file to the console. The console input is then picked up by the Groovy code from the plugin, and vital statistics is displayed in the build history. The build summary details are also modified.

Getting ready

Follow the recipe *Reviewing license violations from within Maven*. Add the Groovy Post-build plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Groovy+Postbuild+Plugin`).

How to do it...

1. Update the `pom.xml` file by adding the following XML fragment just before the `</plugins>` tag:

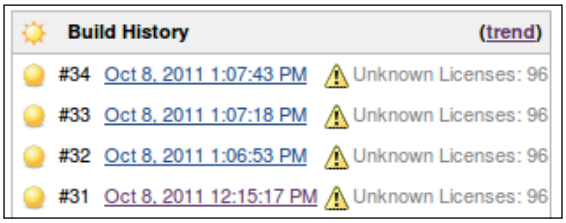
```
<plugin>
  <groupId>org.codehaus.gmaven</groupId>
  <artifactId>gmaven-plugin</artifactId>
  <version>1.3</version>
  <executions><execution>
    <id>run-myGroovy</id>
    <goals><goal>execute</goal></goals>
    <phase>verify</phase>
    <configuration>
      <source>
        new File("${basedir}/target/rat.txt").eachLine{
          line-> println line}
      </source>
    </configuration>
  </execution></executions>
</plugin>
```

2. Update the configuration of the `ch3.BasicLTI_license` Job under the **Post-build Actions** section. Check Groovy Postbuild. Add the following script to the Groovy script text input.

```
def matcher = manager.getMatcher(manager.build.logFile, "^(.*)
  Unknown Licenses\$")
if(matcher?.matches()) {
  title="Unknown Licenses: ${matcher.group(1)}"
  manager.addWarningBadge(title)
  manager.addShortText(title, "grey", "white", "0px", "white")
  manager.createSummary("error.gif")
    .appendText("<h2>${title}</h2>", false, false, false, "grey")
  manager.buildUnstable()
}
```

3. Make sure that the select box **If the script fails:** is set to **Do Nothing**.
4. Click on **Save**.

5. Run the Job a number of times. In the **Build History**, you will see results similar to the following screenshot:



The results are also summarized.



How it works...

The RATs licensing report is saved to the location `target/rat.txt`. The Groovy code then reads the RATs file and prints it out to the console, ready to be picked up by the Post-build plugin. You could have done all the work in the Post-build plugin, but you might later want to re-use the build.

After the build is finished, the Post-build Groovy plugin runs. A number of Jenkins services are visible to the plugin:

- `manager.build.logFile` gets the log file, which now includes the licensing information.

- `manager.getMatcher` checks the log file for patterns matching `"^(.*) Unknown Licenses\$"`. The symbol `^` checks for the beginning of the line, and `\$` checks for the end of the line. Any line with the pattern `Unknown Licenses` at the end of the line will be matched with anything before that stored in `matcher.group(1)`. It sets the title string to the number of `Unknown licenses`.
- `manager.addWarningBadge(title)` adds a warning badge to the build history box, and the title is used as text that is displayed as the mouse hovers over the icon.
- `manager.addShortText` adds visible text next to the icon.
- A summary is created through the `manager.createSummary` method. An image that already exists in Jenkins is added with the title.



You can add HTML tags to the summary. Consider this a security issue.

There's more...

Pulling information into a report by searching for a regular pattern is called **scraping**. The stability of scraping relies on a consistent pattern being generated in the RATs report. If you change the version of the RAT's plugin, the pattern might change and break your report. When possible, it is more maintainable for you to use the stable data sources, such as XML files, which have a well-defined syntax.

See also

- *Exposing information through build descriptions*

Chapter 2, *Enhancing Security*:

- *Improving security through small configuration changes*

Remotely triggering Jobs through the Jenkins API

Jenkins has a remote API, which allows you to enable, disable, run, delete Jobs, and change configuration. The API is increasing with the Jenkins version. To get the most up-to-date details, you will need to review `http://yourhost/job/Name_of_Job/api/`. Where `yourhost` is the location of your Jenkins server, and the `Name_of_Job` is the name of a Job that exists on your server.

This recipe details how you can trigger build remotely by using security tokens. This will allow you to run other Jobs from within your Maven.

Getting ready

This recipe expects Jenkins security to be turned on so that you have to log in as a user. It also assumes that you have a modern version of **wget** (`http://www.gnu.org/s/wget/`) installed.

How to do it...

1. Create a free-style project with Project named `ch3.RunMe`.
2. Check **This Build is parameterized** and select **String Parameter**:
   - **Name**: `myvariable`
   - **Default Value**: `Default`
   - **Description**: `This is my example variable`
3. Under the **Build Triggers** section, check **Trigger builds remotely** `(e.g., from scripts)`.
4. In the **Authentication Token** textbox, add `changeme`.
5. Click on the **Save** button.
6. Run the Job.
7. You will be asked for the variable **myvariable**. Click on **Build**.
8. Visit your personal configuration page, for example: `http://localhost:8080/user/your_user/configure`, where you replace `your_user` with your Jenkins username.
9. In the **API Token** section, click on the **Show API Token...** button.
10. Modify the token to `apiToken`.
11. Click on the **Change API Token** button.
12. From a terminal console, run `wget` to log in, and run the Job remotely. For example:

```
wget --auth-no-challenge --http-user=Alan --http-password=apiToken
http://localhost:8080/job/RunMe/build?token=changeme
```

13. Check the Jenkins Job to verify that it has run.
14. From a terminal console, run `wget` to log in, and run the Job. For example:

```
wget --auth-no-challenge --http-user=Alan --http-password=apiToken
http://localhost:8080/job/RunMe/buildWithParameters?token=changeme
\&Myvariable='Hello World'
```



Warning: There are two obvious security issues in this recipe:

Short tokens are easy to guess. You should make your tokens large and random.

The HTTP protocol can be packet sniffed by a third party. Use HTTPS when transporting passwords.

How it works...

To run a Job, you need to authenticate as a user and then obtain permission to run the specific Job. This is achieved through `apiTokens`, which you should consider as passwords.

There are two remote method calls. The first is `build`, which runs the build without passing parameters. The second is `buildWithParameters`, which expects you to pass at least one parameter to Jenkins. The parameters are separated by `\&`.

The `wget` tool does the heavy lifting, otherwise you would have had to write some tricky Groovy code. We have chosen simplicity and OS dependence for the sake of a short recipe running an executable risk, making your build OS-specific. The executable will depend on how the underlying environment has been set up. However, sometimes you will need to make compromises to avoid complexity. You can find the equivalent Java code at the following URL:

`https://wiki.jenkins-ci.org/display/JENKINS/Remote+access+API`

There are some excellent comments from the community at the end of the page.

There's more...

Here are a few things that you should consider:

Running Jobs from within Maven

With little fuss, you can run `wget` through the `maven-antrun-plugin` plugin. The following is the equivalent pom XML fragment.

```
<build>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>Maven-antrun-plugin</artifactId>
    <executions><execution>
    <phase>compile</phase>
    <configuration>
      <tasks>
        <exec executable="wget">
```

```
            <arg line="--auth-no-challenge --http-user=Alan --http-
              password=apiToken
              http://localhost:8080/job/RunMe/build?token=changeme" />
          </exec>
        </tasks>
      </configuration>
      <goals><goal>run</goal></goals>
      </execution></executions>
    </plugin>
  </build>
```

Remotely generating Jobs

There is also a project that allows you to remotely create Jenkins Jobs through Maven (`http://evgeny-goldin.com/wiki/maven-jenkins-plugin`). The advantage of this approach is its ability to enforce consistency and re-use between Jobs. You can use one parameter to choose Jenkins server and populate. This is useful for generating a large set of consistently-structured Jobs.

See also

- *Running AntBuilder through Groovy in Maven*

Adaptive site generation

Jenkins is a great communicator. It can consume the results of the tests generated by builds. Maven has a goal for site generation, where within the `pom.xml` file, many of the Maven testing plugins are configured. The configuration is bounded by the `reporting` tag.

When run, a Jenkins Maven 2/3 software project Job does a number of things. It notes when a site is generated and creates a shortcut icon in the Jobs home page. This is a highly-visible icon that you can link with content.



You can gain fine-grained control of Maven site generation by triggering the Groovy scripts that structure sites in different Maven phases.

In this recipe, you will use Groovy to generate a dynamic site menu that has different menu links, depending on a random choice made in the script. A second script then generates a fresh results page, per site generation. These actions are useful if you want to expose your own custom test results. The recipe *Reporting alternative code metrics* in *Jenkins* describes how you can plot custom results in Jenkins, enhancing the user's experience further.



Warning: This recipe works in version 2.2.1 of Maven or earlier. Maven 3 has a slightly different approach to site generation.

To enforce the Maven version from within your `pom.xml` file, you would need to add `<prerequisites><maven>2.2.1</maven></prerequisites>`.

Getting ready

Create the directory named `ch3.building_software/site` for the recipe. Install the **copy data to workspace** plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Copy+Data+To+Workspace+Plugin`). This will give you practice with another useful plugin. You will use this plugin to copy the files mentioned in this recipe into the Jenkins workspace.

How to do it...

1. Add the following XML fragment just before `</project>` within your template `pom.xml` file (mentioned in the introduction), making sure that the `pom.xml` file is readable by Jenkins.

```
<url>My_host/my_dir</url>
<description>This is the meaningful DESCRIPTION</description>
<build>
  <plugins><plugin>
    <groupId>org.codehaus.gmaven</groupId>
    <artifactId>gmaven-plugin</artifactId>
    <version>1.3</version>
    <executions>
      <execution>
        <id>run-myGroovy-add-site-xml</id>
        <goals><goal>execute</goal></goals>
        <phase>pre-site</phase>
        <configuration>
          <source>
            site_xml.Groovy
          </source>
        </configuration>
      </execution>
```

```
      <execution>
        <id>run-myGroovy-add-results-to-site</id>
        <goals><goal>execute</goal></goals>
        <phase>site</phase>
        <configuration>
          <source>
            site.Groovy
          </source>
        </configuration>
      </execution>
    </executions>
  </plugin></plugins>
</build>
```

2. Create the file named `site_xml.Groovy` within the same directory as your `pom.xml` file, with the following contents:

```
def site= new File('./src/site')
site.mkdirs()
def sxml=new File('./src/site/site.xml')
if (sxml.exists()){sxml.delete()}

sxml << '<?xml version="1.0" encoding="ISO-8859-1"?>'
sxml << '<project name="Super Project">'
sxml << '<body>'
def random = new Random()
if (random.nextInt(10) > 5){
    sxml << '    <menu name="My super project">'
    sxml << '     <item name="Key Performance Indicators" href="/
our_results.html"/>'
    sxml << '   </menu>'
    print "Data Found menu item created\n"
}
sxml << '   <menu ref="reports" />'
sxml << '  </body>'
sxml << '</project>'

print "FINISHED - site.xml creation\n"
```

3. Add the file named `site.Groovy` within the same directory as your `pom.xml` file, with the following contents:

```
def site= new File('./target/site')
site.mkdirs()
def index = new File('./target/site/our_results.html')
if (index.exists()){index.delete()}
```

```
index << '<h3>ImportAnt results</h3>'
index << "${new Date()}\n"
index << '<ol>'

def random = new Random()
for ( i in 1..40 ) {
    index << "<li>Result[${i}]=${random.nextInt(50)}\n"
}
index << '</ol>'
```

4. Create a Maven 2/3 project with the name `ch3.site`.
5. Under the **Build** section, fill in the following details:
   - **Maven Version**: `2.2.1`
   - **Root POM**: `pom.xml`
   - **Goals and options**: `site`
6. Under the **Build Environment** section, tick **Copy data to workspace**.
7. Add to **Path to folder:** the path to the directory where you have placed the files.
8. Run the job a number of times, reviewing the generated site. On the right-hand side, you should see a menu section named **My super project**. For half of the runs, there will be a sub-menu link named **Key Performance Indicators**.



How it works...

Two Groovy scripts are run in two different phases of the site goal. The first generates the `site.xml` file. Maven uses this to create an additional menu structure on the left-hand side of the index page. The second Groovy script generates a page of random results.

`site_xml.Groovy` runs in the pre-site phase. `site.Groovy` executes during site generation. `site_xml.Groovy` generates the directory `src/site`, and then the file `src/site/site.xml`. This is the file that the Maven site generation plugin uses to define the left-hand side of a `sites` menu. For more details of the process, review `http://Maven.apache.org/guides/mini/guide-site.html`.

The Groovy script then randomly decides in the line `if (random.nextInt(10) > 5)`, when to show an extra menu item for the results page.

`site.Groovy` generates a random results page of 40 entries. If an older results page exists, the Groovy script deletes it. The script cheats a little by creating the `target/site` directory first. If you want a much longer or shorter page, modify the number `40` in the line `for ( i in 1..40 ) {`.

After the build script is run, Jenkins sees that a site sits in the conventional place and adds an icon to the Job.



Warning: At the time of writing, only Maven 2/3 project jobs sense the existence of generated sites and publish the site icon. Free-style Jobs do not.

There's more...

Here is some more useful information:

Searching for example site generation configuration

Sometimes, there can be arbitrary XML magic in configuring site generation. One of the ways to learn quickly is to use a software code search engine. For example, try searching for the term `<reporting>`, using the **Koders** search engine (`http://www.koders.com`).

Maven 2 and Maven 3 – differences

Maven 3 is mostly backwardly-compatible with Maven 2. However, it does have some minor differences that you can review at `https://cwiki.apache.org/MAVEN/maven-3x-compatibility-notes.html`, and for the compatibility list of plugins: `https://cwiki.apache.org/MAVEN/maven-3x-plugin-compatibility-matrix.html`

Under the bonnet, Maven 3 is a rewrite of Maven 2, with improved architecture and performance. Emphasis has been placed on compatibility with Maven 2. You don't want to break the legacy configuration, as that would cause unnecessary maintenance work. Maven 3 is a little bit fussier about the syntax than Maven 2 is. For example, it will complain if you forget to add a version number for any of your dependencies or plugins.

The most visible change is the use of the `maven-site-plugin` plugin in Maven 3 reflected in the way the `<reporting>` section is configured. For more details, you can review `http://Maven.apache.org/plugins/Maven-site-plugin-3.0-beta-3/`.

See also

- *Running Groovy scripts through Maven*
- *Plotting alternative code metrics in Jenkins*

4

Communicating through Jenkins

In this chapter, we will cover the following recipes:

- Skinning Jenkins with the Simple Theme plugin
- Skinning and provisioning Jenkins using a WAR overlay
- Generating a home page
- Creating HTML reports
- Efficient use of views
- Saving screen space with the Dashboard plugin
- Making noise with HTML5 browsers
- An eXtreme view for reception areas
- Mobile presentation using Google Calendar
- Tweeting the world
- Mobile apps for Android and IOS
- Getting to know your audience with Google Analytics

Introduction

This chapter explores communication through Jenkins, recognizing that there are different target audiences.

Jenkins is a talented communicator. Its home page displays the status of all the jobs, allowing you to make quick decisions. You can easily set up multiple views, prioritizing information naturally. Jenkins, with its hoard of plugins, notifies you by e-mail, Twitter, and Google services. It shouts at you through mobile devices, radiates information as you walk past big screens, and fires at you with USB sponge missile launchers.

The primary audience is developers, but don't forget the wider audience that wants to use the software being developed. This chapter includes recipes to help you reach this wider audience.

When creating a coherent communication strategy, there are many Jenkins-specific details to configure. Here are a few that will be considered in this chapter:

- **Notifications**: Developers need to know quickly when something is broken. Jenkins has many plugins; you should select a few that suit the team's ethos.
- **Mobile-friendly plugins**: By firing notifications at well-known social media such as Twitter, it ensures access to a global audience without the need for extra installation tasks on the mobile device.
- **Page decoration**: A page decorator is a plugin that adds content to each page. You can cheaply generate a corporate look and feel by adding your own stylesheets and JavaScript.
- **Overlaying Jenkins**: Using the **Maven WAR** plugin, you can overlay your own content over Jenkins. You can use this to add custom content and provision resources such as home pages, which will enhance the corporate look and feel.
- **Optimize the views**: Front-page views are the lists of jobs that are displayed in a tab. The front page is used by the audience to quickly decide which job to select for review. Plugins expand the choice of view types and optimize information digestion. This potentially avoids the need to look further, thereby saving precious time.
- **Drive by notification**: Extreme views that radiate information visually look great on large monitors. If you place a monitor by watering holes such as receptions or coffee machines, passersby will absorb the ebb and the flow of job status changes. The view sublimely hints at the professionalism of your company and the stability of your product's roadmap.
- **Keeping track of your audience**: If you are openly communicating, you should track usage patterns so that you can improve services. Consider connecting your Jenkins pages to Google Analytics or Piwik, an open source analytics application.



Subversion repository

From this chapter onwards, you will need a Subversion repository. This will allow you to use Jenkins in the most natural way possible. If you do not already have a repository, there are a number of free or semi-free services you can sign up for on the Internet, for example, `http://www.straw-dogs.co.uk/09/20/6-free-svn-project-hosting-services/`.

Alternatively, you can consider setting up Subversion locally, for example, `https://help.ubuntu.com/community/Subversion`.

Skinning Jenkins with the Simple Theme plugin

This recipe modifies the Jenkins look and feel through the themes plugin.

The Simple Theme plugin is a page decorator; it decorates each page with extra HTML tags. The plugin allows you to upload a stylesheet and JavaScript file. The files are then reachable through a local URL. Each Jenkins page is then decorated with HTML tags that use the URLs to pull in your uploaded files. Although straightforward, when properly crafted, the visual effects are powerful.

Getting ready

Install the themes plugin available at `https://wiki.jenkins-ci.org/display/JENKINS/Simple+Theme+Plugin`.

How to do it...

1. Under the Jenkins `userContent` directory, create a file named `my.js`, with the following content:

```
document.write("<h1 id='test'>Example Location</h1>")
```

2. Create a `my.css` file in the Jenkins `userContent` directory, with the following content:

```
@charset "utf-8";
#test {
  background-image: url
    (/userContent/camera.png);
}
#main-table{
  background-image:
    url(/userContent/camera.png) !important;
}
```

3. Browse through `http://openiconlibrary.sourceforge.net/gallery2/` and review the freely available icons. Download an icon and add it to the `userContent` directory, renaming the icon to `camera.png`. For example, consider downloading `http://openiconlibrary.sourceforge.net/gallery2/open_icon_library-full/icons/png/128x128/emblems/emblem-camera.png`
4. Visit the Jenkins main configuration page `/configure`. Under the **Theme** section, fill in the location of the CSS and JavaScript files:
   - **URL of theme CSS**: `/userContent/my.css`
   - **URL of theme JS**: `/userContent/my.js`
5. Press **Save**.
6. Return to the Jenkins home page, and review your work.



How it works...

The Simple Theme plugin is a page decorator. It adds the following information to every page:

```
</script>
<link rel="stylesheet" type="text/css"
  href="/userContent/my.css" />
<script src="/userContent/my.js"
  type="text/javascript">
```

The JavaScript writes a heading near the top of the generated pages with `id='test'`. The Cascading Style Sheet, having a rule triggered through the CSS locator `#test`, adds the camera icon to the background.

The picture's dimensions are not properly tailored for the top of the screen; they are trimmed by the browser. This is a problem you can solve later by experimenting.

The second CSS rule is triggered for `main-table`, which is a part of the standard front page generated by Jenkins. The full camera icon is displayed there.

On visiting other parts of Jenkins, you will notice that the camera icon looks out of context and is oversized. You will need time to modify the CSS and JavaScript to generate better effects. With care and custom code, you can skin Jenkins to fit your corporate image.



CSS 3 Quirks

There are quirks in the support for the various CSS standards between browser types and versions. For an overview, please visit the following page:

`http://www.quirksmode.org/css/contents.html`

There's more...

Here are a few more things for you to consider:

CSS 3

CSS 3 has a number of new features; to draw a button around the header generated by the JavaScript, change the `#test` section of the CSS file to:

```
test {
  width: 180px; height: 60px;
  background: red; color: yellow;
  text-align: center;
  -moz-border-radius: 40px; -webkit-border-radius: 40px;
}
```

Using Firefox, the CSS rule generated the following button:





For the impatient, you can download a CSS 3 cheat sheet at the Smashing Magazine website: `http://coding.smashingmagazine.com/wp-content/uploads/images/css3-cheat-sheet/css3-cheat-sheet.pdf`

Included JavaScript library frameworks

Jenkins uses the **YUI** library (`http://developer.yahoo.com/yui/`). Decorated in each HTML page, the core YUI library—`yahoo-min.js`—is pulled in ready for reuse. However, many web developers are used to jQuery. You can also include the library by installing the jQuery plugin (`https://wiki.jenkins-ci.org/display/JENKINS/jQuery+Plugin`). You can also consider adding your favorite JavaScript library to the Jenkins `/scripts` directory through a WAR overlay (see the next recipe).

Trust, but verify

With great power comes great responsibility. If your Jenkins deployment is maintained by only a few administrators, you can most likely trust everyone to add JavaScript that has no harmful side effects. However, if you have a large set of administrators, who use a wide range of Java libraries, your maintenance and security risks increase rapidly. Please consider your security policy, at least adding an audit plugin to keep track of actions.

See also

- *Skinning and provisioning Jenkins using a WAR overlay*
- *Generating a home page*

Skinning and provisioning Jenkins using a WAR overlay

This recipe describes how to overlay the content onto the Jenkins WAR file. With a WAR overlay, you can change the Jenkins look and feel ready for corporate branding and content provisioning of home pages, among others. The basic example of adding your own custom `favicon.ico` (the icon in your web browser's address bar) is used. It requires a nominal effort to include more content.

Jenkins keeps its versions as dependencies in a Maven repository. You can use Maven to pull in the WAR file, expand it, add content, and then repackage. This enables you to provision resources, such as images, home pages, the icon in the address bar called a favicon, `robots.txt` (which affects how search engines look through your content), and so on.

Be careful—using a WAR overlay will work cheaply if the structure and the graphical content of Jenkins do not radically change over time. However, if the overlay does break the structure, you might not spot this until you perform detailed functional tests.

You can also consider minimal changes through a WAR overlay, perhaps only changing `favicon.ico`, adding images and `userContent`, and then using the Simple Theme plugin (see the previous recipe) to do the styling.

Getting ready

Create a directory named `ch4.communicating/war_overlay` for the files in this recipe.

How to do it...

1. Browse to the Maven repository, `https://maven.glassfish.org/content/groups/public/`, and review the Jenkins dependencies.
2. Create the following `pom.xml` file. Feel free to update to a newer Jenkins version.

```
<project xmlns=
  "http://maven.apache.org/POM/4.0.0" xmlns:xsi=
  "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>nl.uva.berg</groupId>
  <artifactId>overlay</artifactId>
  <packaging>war</packaging>
<!-- Keep version the same as Jenkins as a hint -->
  <version>1.437</version>
  <name>overlay Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>org.jenkins-ci.main</groupId>
      <artifactId>jenkins-war</artifactId>
      <version>1.437</version>
      <type>war</type>
      <scope>runtime</scope>
    </dependency>
  </dependencies>
  <repositories>
    <repository>
      <id>m.g.o-public</id>
      <url>
        http://maven.glassfish.org/content/groups/public/
      </url>
    </repository>
  </repositories>
</project>
```

3. Visit a `favicon.ico` generation website, such as `http://www.favicon.cc/`. Following their instructions, create your own `favicon.ico`. Alternatively, use the example provided.

4. Add `favicon.ico` to the location `src/main/webapp`.
5. Create the directory `src/main/webapp/META-INF` and add a file named `context.xml`, with the following content:

   ```
   <Context logEffectiveWebXml="true"path="/"></Context>
   ```

6. In your top-level directory, run the following command:

   ```
   mvn package
   ```

7. In the newly generated target directory, you will see the `overlay-1.437.war` file. Review the content, verifying that you have modified `favicon.ico`.
8. (Optional) Deploy the WAR file to a local Tomcat server, and verify and browse the updated Jenkins server.



How it works...

Jenkins has its WAR files exposed through a central Maven repository. This allows you to pull in specific versions of Jenkins through standard Maven dependency management.

Maven uses conventions. It expects to find the content to overlay at either `src/main/webapp` or `src/main/resources`.

The `context.xml` file defines certain behaviors for a web application, such as database settings. In this example, the setting `logEffectiveWebXML` is asking Tomcat to log specific information on startup of the application (`http://tomcat.apache.org/tomcat-7.0-doc/config/context.html`). The setting was recommended in the Jenkins Wiki (`https://wiki.jenkins-ci.org/display/JENKINS/Installation+via+Maven+WAR+Overlay`). The file is placed in the `META-INF` directory, because Tomcat picks up the settings here without the need of a server restart.

The `<packaging>war</packaging>` tag tells Maven to use the WAR plugin for packaging.

You used the same version number in the name of the final overlayed WAR as the original Jenkins WAR version. It makes it easier to spot if the Jenkins version changes. This again highlights that using conventions aids readability and decreases the opportunity for mistakes. When deploying from your acceptance environment to production, you should remove the version number.

In the `pom.xml` file, you defined `https://maven.java.net/content/groups/public/` as the repository in which to find Jenkins.

The Jenkins WAR file is pulled in as a dependency of type—`war` and scope—`runtime`. The `runtime` scope indicates that the dependency is not required for compilation but is for execution. For more detailed information on scoping, refer to `http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#Dependency_Scope`.

For further details about the WAR overlays, refer to `http://maven.apache.org/plugins/maven-war-plugin/`.



Avoiding work

To limit maintenance efforts, it is better to install extra content rather than replace content that might be used elsewhere or by third-party plugins.

There's more...

There are a lot of details that you need to cover if you wish to fully modify the look and feel of Jenkins. The following sections mention some of the details:

Which types of content can you replace?

The Jenkins server deploys into two main locations:

- The first location is for the core application
- The second location is the workspace, which stores information that changes

To gain a fuller understanding of the content, review the directory structure. A useful command in Linux is the `tree` command—it displays the directory structure. To install under Ubuntu, use the following command:

```
apt-get install tree
```

For the Jenkins Ubuntu workspace, using the following command generates a tree view of the workspace:

```
tree –d –L 1 /var/lib/Jenkins
```

- **fingerprints**: This is a directory to store checksums to uniquely identify files
- **jobs**: This directory stores Job configuration and build results
- **plugins**: This is the directory where plugins are deployed and mostly configured
- **tools**: This is the directory where tools such as Maven and Ant are deployed
- **updates**: This directory contains information about plugin updates

- **userContent**: The contents of this directory are made available under `/userContent]`
- **users** – This directory contains the user information displayed under `/me`

The default Ubuntu location of the webapp is `/var/run/Jenkins/war`. If you are running Jenkins from the command line, then the option for placing the webapp is `webroot`.

- **css** – This is the location of Jenkins stylesheets
- **executable** – This is used for running Jenkins from the command line
- **favicon.ico** – This is the icon we replaced in this recipe
- **help** – This directory contains the help content
- **images** – This directory stores graphics in different sizes
- **META-INF** – This is the location for the manifest file and the `pom.xml` file that generated the WAR
- **robots.txt** – This file is used to tell search engines where they are allowed to crawl
- **scripts** – This is the JavaScript library location
- **WEB-INF** – This is the main location for the servlet part of the web application
- **winstone.jar** – This is the servlet container; for more information, refer to `http://winstone.sourceforge.net/`

Search engines and robots.txt

If you are adding your own custom content, such as user home pages, company contact information, or product details, consider modifying the top-level file—`robots.txt`. At present, it excludes search engines from all content.

```
# we don't want robots to click "build" links
User-agent: *
Disallow: /
```

You can find the full details of the structure of the `robots.txt` file at `http://www.w3.org/TR/html4/appendix/notes.html#h-B.4.1.1`

Google uses richer structures that **allow** as well as **disallow** search engines from discovering content; see `https://developers.google.com/webmasters/control-crawl-index/docs/robots_txt`.

The following `robots.txt` file allows access by the Google crawler to the directory `/userContent/corporate/`. It is an open question if all web crawlers will honor the intent.

```
User-agent: *
Disallow: /
User-agent: Googlebot
Allow: /userContent/corporate/
```

See also

- *Skinning Jenkins with the Simple Theme plugin*
- *Generating a home page*

Generating a home page

The user's home page is a great place to express your organization's identity. You can create a consistent look and feel that expresses your team's spirit.

This recipe will explore the manipulation of home pages found under the `/user/userid` directory and configured by the user through the Jenkins `/me` folder path.

Getting ready

Install the Avatar plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Avatar+Plugin`). Create a Jenkins account for the user `fakeuser`. You can configure Jenkins with a number of authentication strategies; the choice will affect how you create a user. One example is to use Project-based Matrix tactics, which were detailed in the *Reviewing Project-based Matrix tactics via a custom script* recipe in *Chapter 2*, *Enhancing Security*.

How to do it...

1. Browse to the location `http://en.wikipedia.org/wiki/Wikipedia:Public_domain_image_resources` for a list of public domain sources of images.
2. Search for open source images at `http://commons.wikimedia.org/wiki/Main_Page`.

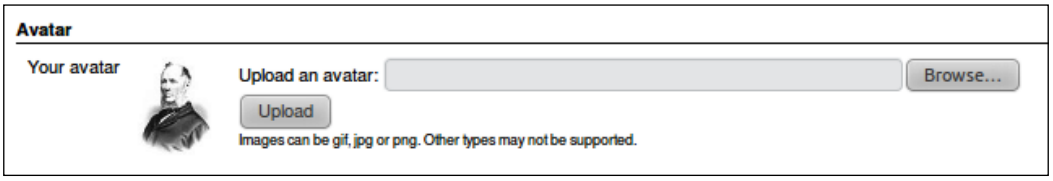
3. Download the image from `http://commons.wikimedia.org/wiki/File%3ACharles_Richardson_(W_H_Gibbs_1888).jpg`, by clicking the link **Download Image File: 75 px**.



Note: If the image is no longer available, choose another.



4. Log in to your sacrificial Jenkins server as `fakeuser`, and visit its configuration page at `http://localhost:8080/user/fakeuser/configure`.
5. Upload the image under the **Avatar** section.



6. Review the URL `http://localhost:8080/user/fakeuser/avatar/image`.



Note: You will now be able to use this known URL whenever you want to display your Avatar.

7. Add the following text to the job's description:

```
<script type="text/JavaScript">
  function changedivview()
  {
    var elem=document.getElementById("divid");
```

```
    elem.style.display=
      (elem.style.display=='none')?
      'block':'none';
  }
</script>
<h2>OFFICIAL PAGE</h2>
<div id="divid">
  <table border=5 bgcolor=gold>
    <tr><td>HELLO WORLD </td> </tr>
  </table>
</div>
<a href="javascript:;"
  onClick="changedivview();">Switch</a><p>
```

8. Visit the `/user/fakeuser` page. You will have a link in the description, named **Switch**. If you click on the link, the **HELLO WORLD** content will appear or disappear.
9. Copy the user directory for `fakeuser` to a directory `fakeuser2`, for example, `/var/lib/Jenkins/user/fakeuser2`.
10. In the `config.xml` file found in the `fakeuser2` directory, change the value of the tag `<fullName>` from `fakeuser` to `fakeuser2`. Change the `<emailAddress>` value to `fakeuser2@dev.null`.
11. Log in as `fakeuser2` with the same password as `fakeuser`.
12. Visit the home page `/user/fakeuser2`. Note the update to the e-mail address.

How it works...

The Avatar plugin allows you to upload an image to Jenkins. The image's URL is in a fixed location. You can reuse the image with the Simple Theme plugin to add content without using a WAR overlay.

There is a vast number of public domain and open source images freely available. Before generating your own content, it is worth reviewing resources on the Internet. If you create content, consider donating to an open source archive such as `http://archive.org`.

Unless you filter the description (see the recipe *Exposing information through build descriptions* in *Chapter 3*, *Building Software*) for HTML tags and JavaScript, you can use custom JavaScript or CSS animations to add eye candy to your personalized Jenkins.

Your `fakeuser` information is stored in `/user/fakeuser/config.xml`. By copying to another directory and slightly modifying the `config.xml` file, you have created a new user account. The format is readable and easy to structure into a template for the creation of yet more accounts. You created the `fakeuser2` account to demonstrate this point.

By using the WAR overlay recipe and adding extra `/user/username` directories containing customized `config.xml` files, you can control Jenkins user populations, for example, from a central provisioning script or at the first login attempt, using a custom authorization script (see *Using the Script realm authentication for provisioning* in *Chapter 2*, *Enhancing Security*).

There's more...

You can enforce consistency by using a template `config.xml`. This will enforce a wider uniform structure. You can set the initial password to a known value or keep it blank. An empty password only makes sense if the time from creation of the user to the first login is very short. You should consider this a bad practice; a problem waiting to happen.

The description is stored under the description tag. The content is stored as a URL escaped text. For example, `<h1>Description</h1>` is stored as:

```
<description>&lt;h1&gt;DESCRIPTION&lt;/h1&gt;</description>
```

A number of plugins also store their configuration in the same `config.xml`. As you increase the number of plugins in your Jenkins server, which is natural as you get to know the product, you will need to occasionally review the completeness of your template.

See also

- *Skinning Jenkins with the Simple Themes plugin*
- *Skinning and provisioning Jenkins using a WAR overlay*
- *Reviewing Project-based Matrix tactics via a custom script*, In *Chapter 2*, *Enhancing Security*

Creating HTML reports

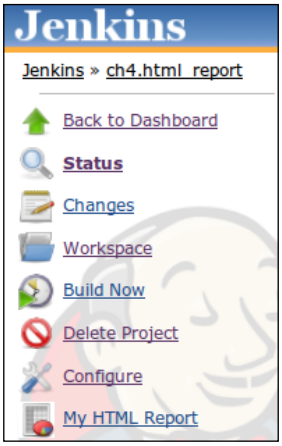
The left-hand menu of jobs on the front page is valuable real estate. The developer's eyes naturally scan this area. This recipe describes how you can add a link from a custom HTML report to the menu, getting the report more quickly noticed.

Getting ready

Install the HTML publisher plugin (`https://wiki.jenkins-ci.org/display/JENKINS/HTML+Publisher+Plugin`). We assume that you have a Subversion repository with the Packt code committed.

How to do it...

1. Create a free-style software project, naming it `ch4.html_report`.
2. Under the **Source Code Management** section, click on **Subversion**.
3. Under the **Modules** section, add `Repo/ch4.communicating/html_report` to **Repository URL**, where `Repo` is the URL to your Subversion repository.
4. Under the **Post-build Actions** section, check **Publish HTML reports**, adding the following details:
   - **HTML directory to archive**: `target/custom_report`
   - **Index page[s]:** `index.html`
   - **Report title**: `My HTML Report`
   - Tick the checkbox **Keep past HTML reports**
5. Press **Save**.
6. Run the job and review the left-hand menu. You will now see a link to your report:



How it works...

Your Subversion repo contains an `index.html` file, which is pulled into the workspace of the job. The plugin works as advertised and adds a link pointing to the HTML report. This allows your audience to efficiently find your custom-generated information.

There's more...

The example report is shown next:

```
<html>
<head>
  <title>Example Report</title>
  <link rel="stylesheet" type="text/css"
    href="/css/style.css" />
</head>
<body>
  <h2>Generated Report</h2>
  Example icon:
    <img title="A Jenkins Icon" alt="Schedule a build"
    src="/images/24x24/clock.png" />
</body>
</html>
```

It pulls in the main Jenkins stylesheet—`/css/style.css`.

It is possible that, when you update a stylesheet in an application, you do not see the changes in your browser until you have cleaned your browser cache. Jenkins gets around this latency issue in a clever way. It uses a URL with a unique number that changes with each Jenkins version. For example, for the `css` directory you have two URLs:

- `/css`
- `/static/uniquenumber/css`

Most Jenkins URLS use the latter form. Consider doing so for your stylesheets.



Note: The unique number changes per version, so you will need to update the URL with each upgrade.

When running the site goal in a Maven build, a local website is generated (`http://maven.apache.org/plugins/maven-site-plugin`). This website has a fixed URL inside the Jenkins job that you can point at with the **My HTML Report** link. This brings within easy reach documentation such as test results.

See also

- *Efficient use of views*
- *Saving screen space with the Dashboard plugin*

Efficient use of views

Jenkins' addictive ease lends itself to creating a large number of jobs. This increases the volume of information exposed to the developers. Jenkins needs to avoid chaos by utilizing the browser space efficiently. One approach is to define minimal views. In this recipe, you will use the **DropDown toolbar** plugin. This plugin removes the tab views and replaces them with one select-box. This aids quicker navigation. You will also be shown how to provision lots of jobs quickly, using a simple HTML form generated by a script.



Warning: In this recipe, you will be creating a large number of views, which you may want to delete later. If you are using a VirtualBox image, consider cloning the image and deleting after you have finished.

Getting ready

Install the DropDown ViewsTabBar plugin from `https://wiki.jenkins-ci.org/display/JENKINS/DropDown+ViewsTabBar+Plugin`.

How to do it...

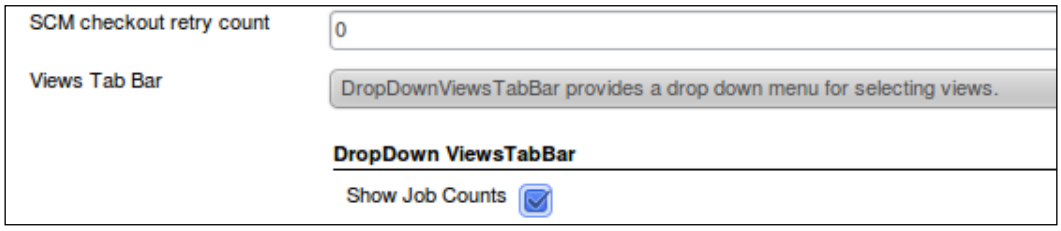
1. Cut and paste the following Perl Script into an executable file named `create.pl`:

```
#!/usr/bin/perl
$counter=0;
$end=20;
$host='http://localhost:8080';
while($end > $counter){
    $counter++;
    print "<form action=$host/
      createItem?mode=copy method=POST>\n";
    print "<input type=text
      name=name value=CH4.fake.$counter>\n";
    print "<input type=text
      name=from value=Template1 >\n";
    print "<input type=submit
      value='Create CH4.fake.$counter'>\n";
    print "</form><br>\n";
    print "<form action=$host/job/
      CH4.fake.$counter/doDelete method=POST>\n";
    print "<input type=submit
      value='Delete CH4.fake.$counter'>\n";
    print "</form><br>\n";
}
```

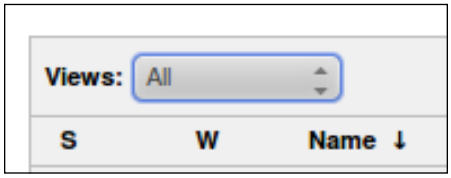
2. Create an HTML file from the output of the Perl Script, for example:

```
perl create.pl > form.html
```

3. In a web browser, as an administrator, log in to Jenkins.
4. Create the job `Template1`, adding any details you wish. This is your template job, which will be copied into many other jobs.
5. Load `form.html` into the same browser.
6. Click on all of the **Create CH4.fake** buttons.
7. Visit the front page of Jenkins, and verify that the jobs have been created and are based on the `Template1` job.
8. Create a large number of views with a random selection of jobs. Review the front page, noting the chaos.
9. Visit the configuration screen, `/configure`. From the **View Tab** Bar select-box, select the **DropDownViewsTabBar provides a drop down menu for selecting views** option.
10. In the subsection **DropDown ViewsTabBar**, check the **Show Job Counts** box.



11. Press the **Save** button.



How it works...

Once you have logged in to Jenkins as an administrator, you can create jobs. You can do this through the GUI or by sending POST information. In this recipe, we copied a job named `Template1` to new jobs starting with the name `CH4.fake`.

```
<form action=
  http://localhost:8080/createItem?mode=copy method=POST>
  <input type=text name=name value=CH4.fake.1>
  <input type=text name=from value=Template1 >
  <input type=submit value='Create CH4.fake.1'>
</form>
```

The `POST` variables you used were `name`, for the name of the new job, and `from`, with the name of your template job. The URL for the `POST` action is `/createItem?mode=copy`.

To change the hostname and port number, you will have to update the `$host` variable found in the Perl script.

To delete a job, the Perl script generated forms with actions pointing to `/job/Jobname/doDelete` (for example, `/job/CH4.fake.1/doDelete`). No extra variables were needed.

To increase the number of form entries, you can change the value of the `$end` variable.

There's more...

Jenkins uses Stapler (`http://stapler.java.net/what-is.html`) to bind services to URLs. Plugins also use Stapler. When you install plugins, the number of potential actions also increases. This means that you can activate a lot of actions through the HTML forms similar to this recipe. You will discover in *Chapter 7*, *Exploring Plugins* that writing binding code to Stapler requires minimal effort.

See also

- *Saving screen space with the Dashboard plugin*

Saving screen space with the Dashboard plugin

In the previous recipe, you discovered that you can save horizontal tab space using the DropDown ViewsTabBar plugin. In this recipe, you will use the Dashboard view plugin to condense the use of the horizontal space. Condensing the horizontal space aids in assimilating information efficiently.

The Dashboard view plugin allows you to configure areas of a view, to display specific functionality, for example, a grid view of the jobs or an area of the view that shows the subset of jobs failing. The user can drag-and-drop the areas around the screen.



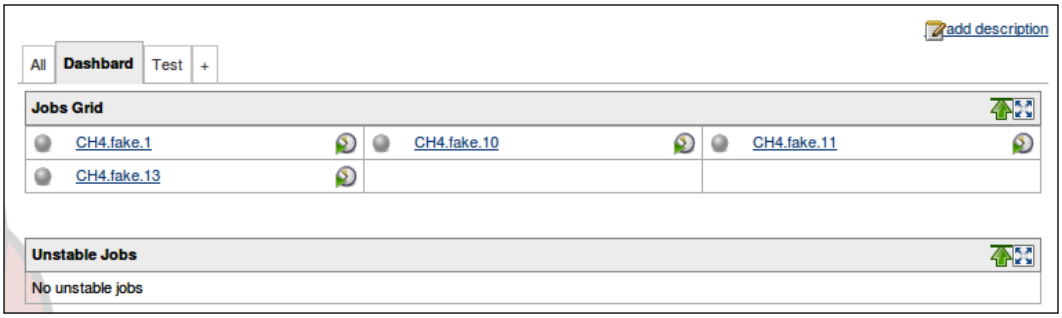
Note: The developers have made the dashboard easily extensible, so expect more choices later.

Getting ready

Install the Dashboard view plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Dashboard+View`). Either create a few jobs by hand, or use the HTML form that provisioned jobs in the previous recipe.

How to do it...

1. As a Jenkins administrator, log in to the home page of your Jenkins instance.
2. Create a new view by clicking on the **+** sign in the second tab, at the top of the screen.
3. Choose the **Dashboard View**.
4. Under the **Jobs** sections, select a few of your fake jobs.
5. Leave the **Dashboard Portlet** as default.
6. Click on **OK**. You will now see a blank view screen.
7. In the left-hand menu, click on the link **Edit View**.
8. In the **Dashboard Portlets** section of the view, select the following:
    - **Add Dashboard Portlet to the top of the view**: `Jobs Grid`
    - **Add Dashboard Portlet to bottom of the view**: `Unstable Jobs`
9. At the bottom of the configuration screen, press the **OK** button. You will now see the dashboard view.





Note: You can expand or contract the areas of functionality with the arrow icon.



How it works...

The Dashboard plugin divides the screen into areas. During the dashboard configuration, you choose the Jobs Grid and the unstable Jobs Portlets. Other dashboard Portlets include a jobs list, latest builds, slave statistics, test statistics (chart or grid), test trend chart, and so on. There will be more choices as the plugin matures.

The **Jobs Grid** portlet saves spaces compared to the other views, as the density of jobs displayed is high.



Warning: If you are also using the many views tab (see the previous recipe), there is a little glitch. When you click on the Dashboard tag, the original set of views is displayed, rather than the select-box.

There's more...

The Dashboard plugin provides a framework for other plugin developers to create dashboard views. One example of this type of usage is the Project Statistics Plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Project+Statistics+Plugin`).

See also

- *Creating HTML reports*
- *Efficient use of views*

Making noise with HTML5 browsers

This recipe describes how to send a custom sound to a Jenkins user's browser when an event, such as a successful build, occurs. You can also send sound messages at arbitrary times. Not only is this good for the developers who enjoy being shouted at, sang to by famous actors, and so on, but also for the system administrators who are looking for a computer in a large server farm.

Getting ready

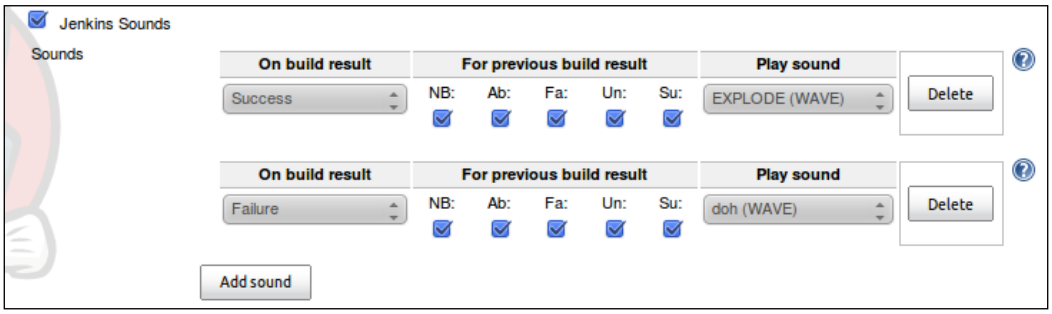
Install the Jenkins Sounds plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Jenkins+Sounds+plugin`). Make sure that you have a compliant web browser installed, say a current version of Firefox or Chrome.



For more details of HTML5 compliancy in browsers, consider reviewing `http://en.wikipedia.org/wiki/Comparison_of_layout_engines_%28HTML5%29`.

How to do it...

1. Log in as a Jenkins administrator, and visit the **Configure System** screen at `/configure`.
2. Under the **Jenkins Sound** section, check **Play through HTML5 Audio enabled Browser**.
3. Press the **Save** button.
4. Select the **Job creation** link, found on the Jenkins home page.
5. Create a **New Job** with the job name `ch4.sound`.
6. Select **Build a free-style software project**.
7. Press **OK**.
8. In the **Post-build Actions** section, check the **Jenkins Sounds** option.
9. Add two sounds: `EXPLODE` and `doh`.



10. Press **Save**.
11. Click on the **Build Now** link.
12. On success, your browser will play the sound in the `EXPLODE.wav` file.
13. Edit your job so that it fails, for example, by adding a non-existent source code repository.
14. Build the job again. On failure, your web browser will play the `doh.wav` file.

How it works...

You have successfully configured your job to play different sounds based on success or failure of the build.

You can refine how the plugin reacts further by configuring which event transitions will trigger a sound. For example, if the previous build result was a failure and the current build result is a success. This is defined in the **For previous build result** set of checkboxes.

The plugin works as a page decorator. It adds the following JavaScript that asynchronously polls for new sounds. Your browser is doing the majority of the work, freeing server resources.

```
<script src="/sounds/script" type="text/javascript">
</script>
<script type="text/javascript" defer="defer">
function _sounds_ajaxJsonFetcherFactory
  (onSuccess, onFailure)
{
     return function()
{
       new Ajax.Request("/sounds/getSounds", {
         parameters: { version: VERSION },
      onSuccess: function(rsp) {
        onSuccess(eval('x='+rsp.responseText))
      },
      onFailure: onFailure
    });
  }
   }
if (AUDIO_CAPABLE) {
     _sounds_pollForSounds
    (_sounds_ajaxJsonFetcherFactory);
   }
</script>
```

There's more...

The Sound plugin also allows you to stream arbitrary sounds to connected web browsers. Not only is this useful for practical jokes and motivational speeches directed at your distributed team, you can also perform useful actions such as a ten-minute warning alert before restarting a server.

You can find some decent sound collections at `http://www.archive.org/details/opensource_audio`.

For example, you can find a copy of the One Laptop per Child music library at `http://www.archive.org/details/OpenPathMusic44V2`. Within the collection, you will discover `shenai.wav`. First, add the sound somewhere on the Internet where it can be found. A good place is the Jenkins `userContent` directory. To play the sound on any connected web browser, you will need to visit the following address (replacing `localhost:8080` with your own address):

```
http://localhost:8080/sounds/playSound?src=http://localhost:8080/
userContent/shenai.wav
```

See also

- *Keeping in contact with Jenkins through Firefox*, *Chapter 1*, *Maintaining Jenkins*

An eXtreme view for reception areas

Agile projects emphasize the role of communication over the need to document. **Information radiators** aid in returning feedback quickly. Information radiators have two main characteristics: they change over time and the data presented is easy to digest.

The eXtreme Feedback plugin is one example of an information radiator. It is a highly visual Jenkins view. If the layout is formatted consistently and displayed on a large monitor, it is ideal for the task. Consider this also as a positive advertisement of your development process that you can display behind your reception desk, or in a well-frequented social area such as near the coffee machine or project room.

In this recipe, you will add the eXtreme Feedback plugin and modify its appearance, through the HTML tags in the description.

Getting ready

Install the eXtreme Feedback plugin (`https://wiki.jenkins-ci.org/display/JENKINS/eXtreme+Feedback+Panel+Plugin`).

How to do it...

1. Create a job with a descriptive name, such as **Blackboard Report PRD Access**, and add the following description:

```
<center>
  <p>Writes Blackboard sanity reports<br>
  and sends them to a list.
  <table border="1" class="myclass">
```

```
    <tr><td>More Details</td></tr>
  </table>
</center>
```

2. Create a new view (`/newView`) named `eXtreme`. Check **eXtreme FeedBack Panel**.
3. Pressing the **OK** button, select between 6 and 24 already created jobs, including the one previously created in this recipe.
4. Set the **Number of Columns** to `2`.
5. Refresh time in seconds to `20`.
6. Click on **Show Job descriptions**.
7. Press **OK**.
8. Experiment with the settings. Optimizing the view depends on the monitors used and the distance from the monitor that the audience will view.



How it works...

Setting up and running this information radiator was simple. The results deliver a beautifully rendered view of the dynamics of your software process.

Setting the refresh rate to 20 seconds is debatable. A long delay between updates dulls the viewer's interest.

You have written one description that is partially formatted. You can see that the information area is easier to digest than the other projects. This highlights the need to write consistent descriptions that follow in-house conventions, under a certain length to fit naturally on the screen. A longer, more descriptive name of a job helps the viewer understand the job's context better.

There's more...

Information radiators are fun and take a rich variety of shapes and forms. From different views displayed in large monitors, to USB sponge missile firing, and abuse from voices of famous actors (see the *Making noise with HTML5 browsers* recipe).

A number of example electronic projects in Jenkins that are worth exploring are:

- **Lava Lamps** – `https://wiki.jenkins-ci.org/display/JENKINS/Lava+Lamp+Notifier`
- **USB missile launcher** – `https://github.com/codedance/Retaliation`
- **Traffic lights** – `http://code.google.com/p/hudsontrafficlights/`

Remember, let's be careful out there.

See also

- *Saving screen space with the Dashboard plugin*
- *Making noise with HTML5 browsers*

Mobile presentation using Google Calendar

Jenkins plugins can push build history to different well-known social media services. Two of the main services are **Google Calendar** (used in agendas) and **Twitter**. Modern Android or IOS mobile devices have preinstalled applications for both these services, lowering the barrier to adoption. In this recipe, we will configure Jenkins to work with Google Calendar.

Getting ready

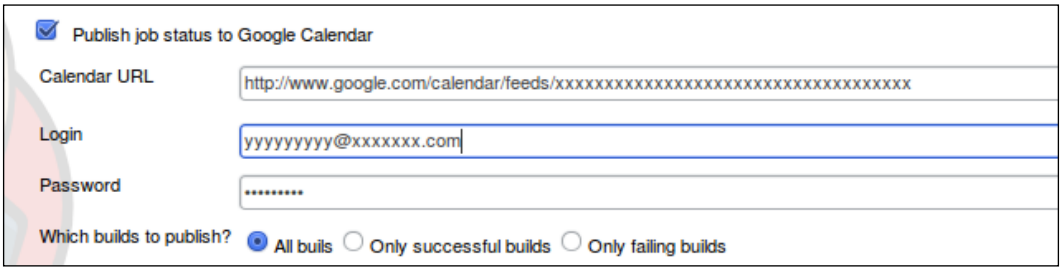
Download and install the Google Calendar plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Google+Calendar+Plugin`). Make sure you have a **test** user account for Gmail.

How to do it...

1. Log in to Gmail and visit the **Calendar** page.
2. Create a new calendar, under the **My Calendars** section.
3. Add the Calendar name `Test for Jenkins`.
4. Click on **Create Calendar**. By default, the new calendar is private. Keep it private for the time being.
5. Under the **My Calendars** section, click on the down-arrow icon next to **Test for Jenkins** and select the option **Calendar Settings**.
6. Right-click on the **XML** button; copy the link location.

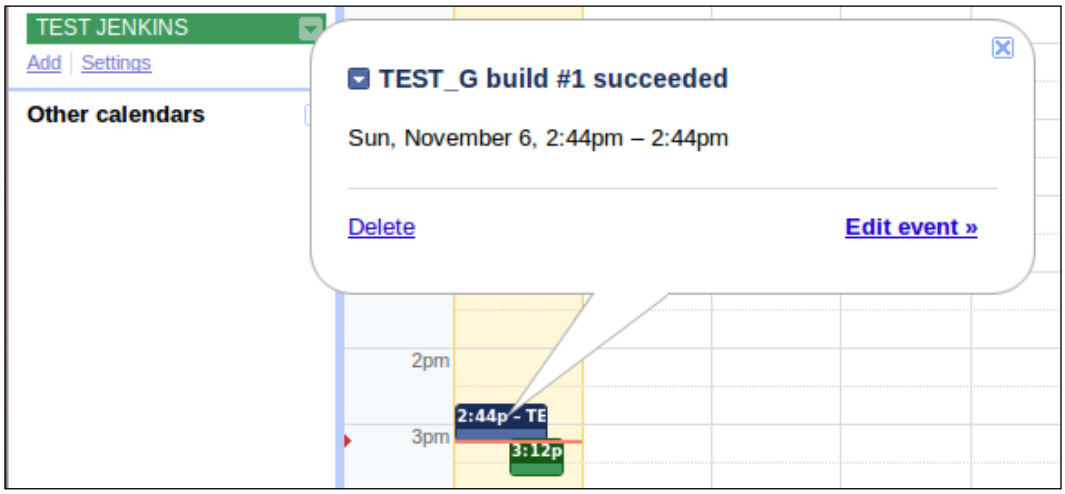


7. Review the section **Embed this calendar**. It describes how to add your calendar to a web page. Cut and paste the supplied code to an empty HTML page. Save and view it in a web browser.
8. Log in to Jenkins as an administrator.
9. Create a new job named `Test_G`.
10. In the **Post build** section, check **Publish job status to Google Calendar**.
11. Add the calendar details you copied from the XML button to the **Calendar URL** textbox.
12. Add your Gmail login name and password.



13. Press **Save**.
14. Build your job, making sure it succeeds.

15. Log in to Gmail and visit the **Calendar** page. You will now see the build's success has been published.



How it works...

By creating a calendar in Google and using just three configuration settings, you have exposed selected Jenkins jobs to Google Calendar. With the same amount of configuration, you can connect most modern smartphones and tablets to the calendar.

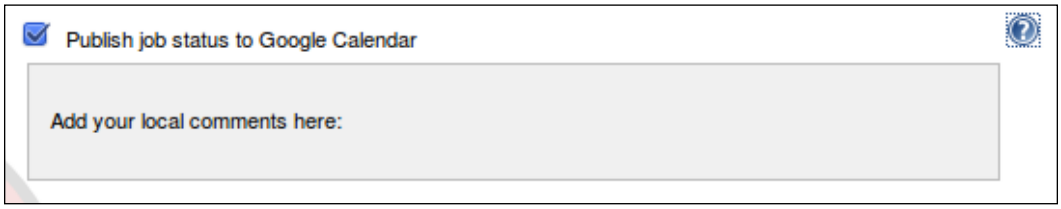
There's more...

In the `plugins` directory, under the Jenkins workspace, you will find an HTML file for help with the configuration of Google plugins, named `/plugins/gcal/help-projectConfig.html`.

Replace the contents with the following:

```
<div>
  <p>
    Add your local comments here:
  </p>
</div>
```

After restarting the Jenkins server, visit the plugin configuration `/configure`. You will now see the new content.



This example is an anti-pattern. If you need to change content for local needs, it is much better to work with the community, adding to the Jenkins SCM, so everyone can see and improve.

You will be told immediately that your content is not internationalized. It needs to be translated into the languages that Jenkins supports natively. Luckily, at the bottom of every Jenkins page, there is a link that volunteers can use to upload translations. The translation effort requires minimal start-up effort and is an easy way to start with an open source project.



For more development details on how to use property files for Internationalization in Jenkins, read `https://wiki.jenkins-ci.org/display/JENKINS/Internationalization`.

See also

- *Tweeting the world*
- *Mobile apps for Android and IOS*

Tweeting the world

Open source code is malleable; you can download, modify, commit, and review the code and form your own judgment on its quality. The Twitter channel is great for managers to see a history of success and failures, as they try and form an opinion about whether the roadmap of the product is realistic. Most modern social devices, such as Android or IOS mobile devices, have Twitter apps built in. The only configuration needed is your user account information. This recipe outlines how to make Jenkins tweet.



Sakai is an Open Source Learning Management System used by millions of students around the world, including the University of Amsterdam for whom I work. Sakai uses Jenkins to build its various sub-projects; see `@sakaibuilds`.

Getting ready

Install the Twitter plugin (`http://wiki.hudson-ci.org/display/HUDSON/Twitter+Plugin`) and download `auth.jar` from the same wiki page.

How to do it...

1. From the command line, run:

   ```
   java –jar auth.jar
   ```

2. Review the output, which will be similar to:

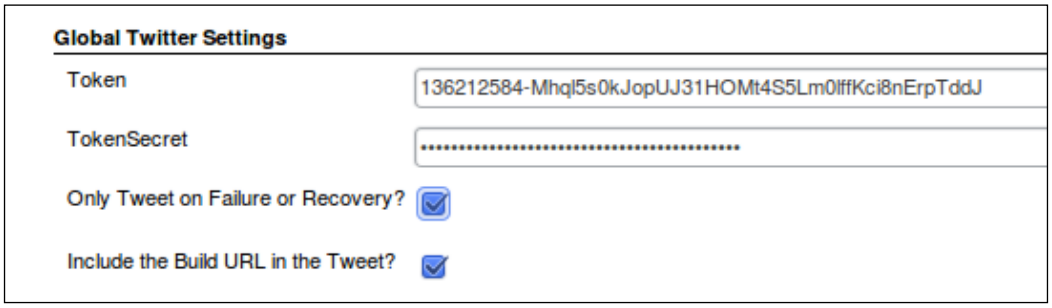
   ```
   [Thu Nov 10 15:54:49 CET 2011]
     Will use class twitter4j.internal.logging.StdOutLoggerFactory as
   logging factory.

   [Thu Nov 10 15:54:49 CET 2011]
     Will use twitter4j.internal.http.HttpClientImpl as HttpClient
   implementation.

   Open the following URL and grant access to your account:

   http://api.twitter.com/oauth/authorize?oauth_token=DlztJbtlOzjNL5F
   v7g5otrQEdBW3WvRxxNQWwmhLnk
   ```

3. Log in to `twitter.com` and follow the link mentioned in your output (starting with `http://api`), copying the PIN number.
4. Enter your PIN number at the command line. You will now see two tokens that you need as configuration in Jenkins, with command-line output similar to:

   ```
   access token:136212584-Mhql5s0kJopUJ31HOMt4S5Lm0lffKci8nErpTddJ

   access token secret:igfGaJaf5iSVJpCc5wK31hKZsyP6SFWMvWoWhOEZ80
   ```

5. Visit the Jenkins Configure System page (`/configure`).

6. Under **Global Twitter Settings**, update **Token** and **TokenSecret**. Check both **Only Tweet on Failure or Recovery** and **Include the Build URL in the Tweet**.



7. Press the **Save** button.
8. Create a new job named **Twitter**.
9. Under the **Post-build Actions** section, check **Twitter**.
10. Under the new **Advanced** section, set **Only Tweet on Failure or Recovery?** to `No`.
11. Click on **Save**.

How it works...

We used the **Twitter OAuth API** for authentication (`https://dev.twitter.com/docs/auth/oauth/faq`).



OAuth is an authentication protocol that allows users to approve applications to act on their behalf, without sharing their password.

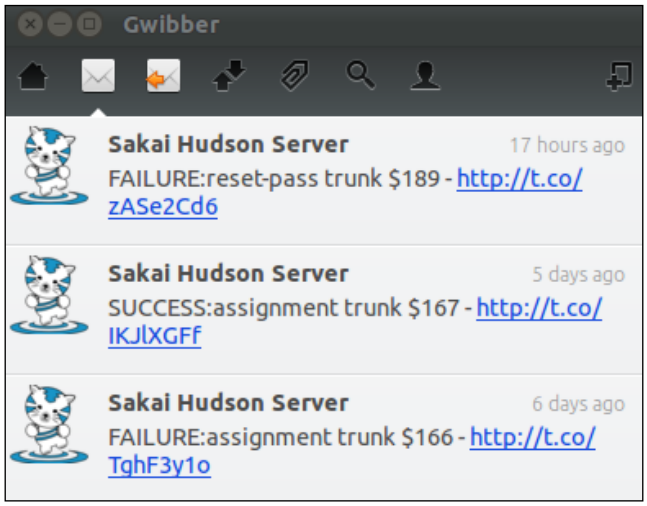
You need to be able to provide credentials so that your plugin can send tweets. The plugin uses **out of band agreements**, known as **oob**. You can find the exact details of how oob works on the Twitter development site, `https://dev.twitter.com/docs/auth#oob`.

In this recipe, you downloaded a JAR file that uses Twitter4J, a standard Twitter Java framework (`http://twitter4j.org`). To create the credentials, you needed to fill in a PIN number at the appropriate moment. You then added the returned token information to the plugin, enabling encryption so that tweeting could commence. You configured the plugin to supply as much information as possible in the tweet. The URL to the build information is a tiny URL, which saves space to make the tweet as mobile-friendly as possible.

There's more...

Almost all modern smartphones have an app for Twitter; however, if you want to compare with a freely available one, then **UberSocial** (`http://ubersocial.com/`) is a good alternative. The app runs on Blackberry, IOS, and Android.

You can also configure the Ubuntu desktop and most NIX desktops to receive tweets through Gwibber (`http://gwibber.com`).



See also

- *Mobile presentation using Google Calendar*
- *Mobile apps for Android and iOS*

Mobile apps for Android and iOS

There are a number of rich mobile apps for the notification of Jenkins job statuses. This recipe points you to their home pages so that you can select your favorite.

Getting ready

You will need a Jenkins instance reachable from the Internet, or you can use `http://ci.jenkins-ci.org/`, an excellent example of best practices. We also assume that you have a mobile device.

How to do it...

1. As an administrator, visit the **Configure System** (`/configure`) screen.
2. Review the Jenkins URL; if it is pointing to `localhost`, change it so that your server links can be reached from the Internet.
3. Visit the following app pages, and if compatible, install and use them:
   - **JenkinsMobi**: `http://www.jenkins-ci.mobi`
   - **Blamer**: `http://www.androidzoom.com/android_applications/tools/blamer_bavqz.html`, `https://github.com/mhussain/Blamer`
   - **Jenkins Mood widget**: `https://wiki.jenkins-ci.org/display/JENKINS/Jenkins+Mood+monitoring+widget+for+Android`
   - **Jenkins Mobile Monitor**: `http://www.androidzoom.com/android_applications/tools/jenkins-mobile-monitor_bmibm.html`
   - **Hudson Helper**: `http://wiki.hudson-ci.org/display/HUDSON/Hudson+Helper+iPhone+and+iPod+Touch+App`
   - **Hudson Mobi**: `http://www.hudson-mobi.com/`
   - **Hudson2Go Lite**: `http://www.androidzoom.com/android_applications/tools/hudson2go-lite_nane.html`
4. On your mobile device, search for Google Apps Marketplace or iTunes, and install any new Jenkins apps that are free and have positive user recommendations.

How it works...

Most of the apps pull in information using the RSS feeds from Jenkins, such as `/rssLatest` and `/rssFailed`, and then load the linked pages through a mobile web browser. Unless the Jenkins URL is properly configured, the links will break and 404-Page Not Found errors will be returned by your browser.

You will soon notice that there is a delicate balance between the refresh rate of your app potentially generating too many notifications, versus receiving timely information.

The JenkinsMobi application runs in both Android and IOS operating systems. It gathers its data using the remote API with XML (`http://www.slideshare.net/lucamilanesio/jenkinsmobi-jenkins-xml-api-for-mobile-applications`), rather than the more raw RSS feeds. This choice allowed the app writers to add a wide range of features, making it arguably the most compelling app in the collection.

There's more...

Here are a few more things for you to consider:

Android 1.6 and Hudson apps

Jenkins split up from the source code of Hudson relatively recently, due to an argument about trademarking the Hudson name (`http://en.wikipedia.org/wiki/Jenkins_%28software%29`). Most developers moved over to working with Jenkins. This left much of the third-party Hudson code either less supported or rebranded to Jenkins. However, Hudson and Jenkins have a large common base, including the content of the RSS feeds. This may well diverge in detail over time. For older Android versions, such as Android 1.6, you will not see any Jenkins apps in Google Apps Marketplace. Try looking for Hudson apps instead. They mostly work on Jenkins.

VirtualBox and the Android-x86 project

There are a number of options for running Android apps. The easiest is to download them through Google Apps Marketplace onto a mobile device. However, if you want to play with Android apps in a sandbox on your PC, consider downloading the Android SDK (`http://developer.android.com/sdk/index.html`), and use an emulator and a tool such as **adb** (`http://developer.android.com/guide/developing/tools/adb.html`) to upload and install apps.

You can also run a virtual machine through VirtualBox, VMware player, and so on, and install an x86 image (`http://www.android-x86.org`). A significant advantage of this approach is the raw speed of the Android OS and the ability to save the virtual machine in a specific state. However, you will not always get Google Apps Marketplace preinstalled. You will either have to find the `.apk` file for a particular app, yourself, or add other marketplaces, such as SlideME (`http://m.slideme.org`). Unfortunately, secondary marketplaces give you much less choice.



The Windows 7 Android emulator, `http://bluestacks.com/home.php`, shows great promise. Not only is it an emulator but it also provides a cloud service to move apps from your mobile device into and out of the emulation. This promises to be an efficient approach for development. However, if you do choose to use this emulator, please thoroughly review the license you agree to during installation. BlueStacks wishes to obtain detailed information about your system to help improve their product.

See also

- *Mobile presentation using Google Calendar*
- *Tweeting the world*

Getting to know your audience with Google Analytics

If you have a policy of pushing your build history or other information, such as home pages, to the public, then you will want to know viewer habits. One approach is to use Google Analytics. With Google Analytics, you can watch in real time as visitors arrive at your site. The detailed reporting mentions things such as overall volume of traffic, browser types, if mobile apps are hitting your site, entry points, country origins, and so on. This is particularly useful as your product reaches key points in its roadmap and you want to gain insight in customer interest.

In this recipe, you will create a Google Analytics account and configure tracking in Jenkins. You will then watch traffic live.

Getting ready

Install the Google Analytics plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Google+Analytics+Plugin`).



Warning: If you are not the owner of your Jenkins URL, please ask for permission first, before creating a Google Analytics profile.

How to do it...

1. Log in with your Gmail account to Google Analytics (`http://www.google.com/analytics/`).
2. Fill in the following details on the **Create New Account** page:
   - **Account Name**: `My Jenkins Server`
   - **Website's URL**: This will be same as the Jenkins URL on the `/configure` screen of Jenkins
   - **Time zone**: Select the correct value from the drop-down box
   - **Data Sharing Settings**: Select the **Do not share my Google Analytics data** option
   - Set the **User Agreement**—**Your country or territory** to the correct value
   - Check the **Terms and conditions** box—**Yes, I agree to the above terms and conditions**
3. Press **Create Account**.
4. You are now on the **Accounts** page for your newly created profile. Copy the **Web Property ID**, which will look something like `UA-121212121212121-1`.



The tracking status states the following:

Tracking Not Installed

The Google Analytics tracking code has not been detected on your website's home page. For Analytics to function, you or your web administrator must add the code to each page of your website.

5. Open a second browser and log in to Jenkins as an administrator.

6. On the Jenkins **Configure System** screen (`/configure`), add the `Profile ID` that you copied from the Google Analytics **Web Property ID** and set the **Domain Name** equal to your Jenkins URL.
7. Press the **Save** button.
8. Visit the home page of Jenkins so that tracking is triggered.
9. Return to Google Analytics; you should still be on the **Tracking code** tab. Press **Save** at the bottom of the page. You will now see that the warning about tracking not installed has disappeared.

How it works...

The plugin decorates each Jenkins page with a JavaScript page tracker, which includes `domain` and `Profile ID`. The JavaScript is kept fresh by being pulled in from the Google Analytics hosts.

```
<script type="text/javascript">
  var gaJsHost = (
    ("https:" == document.location.protocol) ?
    "https://ssl." : "http://www.");
    document.write(unescape("%3Cscript src='" + gaJsHost +
      "google-analytics.com/ga.js'
      type='text/javascript'%3E%3C/script%3E"));
</script>
<script type="text/javascript">
  var pageTracker = _gat._getTracker("TEST_ID");
    pageTracker._setDomainName("TEST_DOMAIN");
    pageTracker._trackPageview();
</script>
```

Google Analytics has the ability to drill into the details of your web usage thoroughly. Consider browsing Jenkins and reviewing the traffic generated through the **real-time reporting** feature.

There's more...

The open source version of Google Analytics is **Piwik** (`http://piwik.org/`). You can set up a server locally and use the equivalent Jenkins plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Piwik+Analytics+Plugin`) to generate statistics. This has the advantage of keeping your usage data local and under your control.

As you would expect, the Piwik plugin is a page decorator injecting similar JavaScript as the Google Analytics plugin.

See also

- *Generating a home page*

5
Using Metrics to Improve Quality

In this chapter, we will cover the following recipes:

- Estimating the value of your project through Sloccount
- Looking for "smelly" code through code coverage
- Activating more PMD rulesets
- Creating custom PMD rules
- Finding bugs with FindBugs
- Enabling extra FindBug rules
- Finding security defects with FindBugs
- Verifying HTML validity
- Reporting with JavaNCSS
- Checking style using an external pom.xml
- Faking checkstyle results
- Integrating Jenkins with SONAR



Some of the build files and code have deliberate mistakes, such as bad naming conventions, poor coding structures, or platform-specific encoding. These defects exist to give Jenkins a target to fire tests against.

Introduction

This chapter explores the use of Jenkins plugins to display code metrics and fail builds. Automation lowers costs and aids in consistency. The process does not get tired. If you decide the success and failure criteria before a project starts, then this will remove a degree of subjective debate from release meetings.

In 2002, NIST estimated that software defects were costing America around 60 billion dollars per year (`http://www.abeacha.com/NIST_press_release_bugs_cost.htm`), expecting the cost to have increased considerably since.

To save money and improve quality, you need to remove defects as early as possible in the software lifecycle. Jenkins test automation creates a safety net of measurements. Another key benefit is that once you have added tests, it is trivial to develop similar tests for other projects.

Jenkins works well with best practices such as **Test Driven Development** (**TDD**) or **Behavior Driven Development** (**BDD**). Using TDD, you write tests that fail first, and then build the functionality needed to pass the tests. With BDD, the project team writes the description of tests in terms of behavior. This makes the description understandable to a wider audience. The wider audience has more influence on the details of the implementation.

Regression tests increase confidence that you have not broken the code while refactoring the software. The more coverage of code by tests, the more confidence. The recipe *Looking for "smelly" code through code coverage* shows you how to measure coverage. You will also find recipes on static code review through PMD and FindBugs. **Static** means that you can look at the code without running it. PMD looks at the `.java` files for particular bug patterns. It is relatively easy to write new bug detection rules using the PMD rules designer. **FindBugs** scans the compiled `.class` files; you can review the application `.jar` files directly. FindBugs rules are accurate, mostly pointing at real defects. In this chapter, you will use FindBugs to search for security defects and PMD to search for design rule violations.

Also mentioned in this chapter is the use of Java classes with known defects. We will use the classes to check the value of the testing tools. This is a similar approach to benchmarks for virus checkers, where virus checkers parse files with known virus signatures. The advantage of injecting known defects is that you get to understand the rules that are violated. This is a great way to not only collect real defects found in your projects but also to characterize and reuse real defects. Consider adding your own classes to projects to see if the QA process picks up the defects.

Good documentation and source code structure aids in the maintainability and readability of your code. Sun coding conventions enforce a consistent standard across projects. In this chapter, you will use **Checkstyle** and **JavaNCSS** to measure your source code against Sun coding conventions (`http://www.oracle.com/technetwork/java/codeconv-138413.html`).

The results generated by the Jenkins plugins can be aggregated into one report through the **violations plugin** (`https://wiki.jenkins-ci.org/display/JENKINS/Violations`). There are other plugins, specific to a given tool, for example, PMD or FindBugs plugins. The plugins are supported by the **Analysis Collector plugin** (`https://wiki.jenkins-ci.org/display/JENKINS/Analysis+Collector+Plugin`), which aggregates the other reports into a consistent whole. The individual plugin reports can be displayed through the **Portlets dashboard plugin**, which was discussed in the *Saving screen space with the Dashboard plugin* recipe, *Chapter 4*, *Communicating Through Jenkins*.

Jenkins is not limited to testing Java; a number of the plugins, such as sloccount or the DRY plugin (spots duplication of code), are language-agnostic. There is even specific support for NUnit testing in .NET or compilation to other languages. If you are missing specific functionality, then you can always build your own Jenkins plugin as detailed in *Chapter 7*, *Exploring Plugins*.

There are a number of good introductions to software metrics; these include a wikibook on the details of the metrics (`http://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Quality/Metrics`). A well written book is by *Diomidis Spinellis Code Quality*: *The Open Source Perspective*, **ISBN 0-321-16607-8**.

In the last recipe of this chapter, you will link Jenkins projects to Sonar reports. **Sonar** is a specialized tool that collects software metrics and breaks them down into an understandable report. Sonar details the quality of a project. It uses a wide range of metrics, including the results of tools such as FindBugs and PMD mentioned in this chapter. The project itself is evolving rapidly. Consider using Jenkins for an early warning and to spot obvious defects, such as a bad commit. You can then use Sonar for a deeper review.



When dealing with multi-module Maven projects, the Maven plugins generate a series of results. The Maven 2/3 project type rigidly assumes that the results are stored in conventional locations, but this does not always happen consistently. With freestyle projects, you can explicitly tell the Jenkins plugins where to find the results using regular expressions that are consistent with Ant filesets (`http://ant.apache.org/manual/Types/fileset.html`).

Estimating the value of your project through Sloccount

One way to gain insight into the value of a project is to count the number of lines of code in the project, and divide the count between code languages. The sloccount (`http://www.dwheeler.com/sloccount/`) command-line tool by *Dr David Wheeler* does just that.

Getting ready

Install the sloccount plugin (`https://wiki.jenkins-ci.org/display/JENKINS/SLOCCount+Plugin`). Create a new directory for this recipes code. Install sloccount on the Jenkins instance as mentioned at `http://www.dwheeler.com/sloccount`. If you are running on a Debian OS, the following installation command will work:

```
sudo apt-get install sloccount
```

For details on how to install sloccount on other systems, please review `http://www.dwheeler.com/sloccount/sloccount.html`.

How to do it...

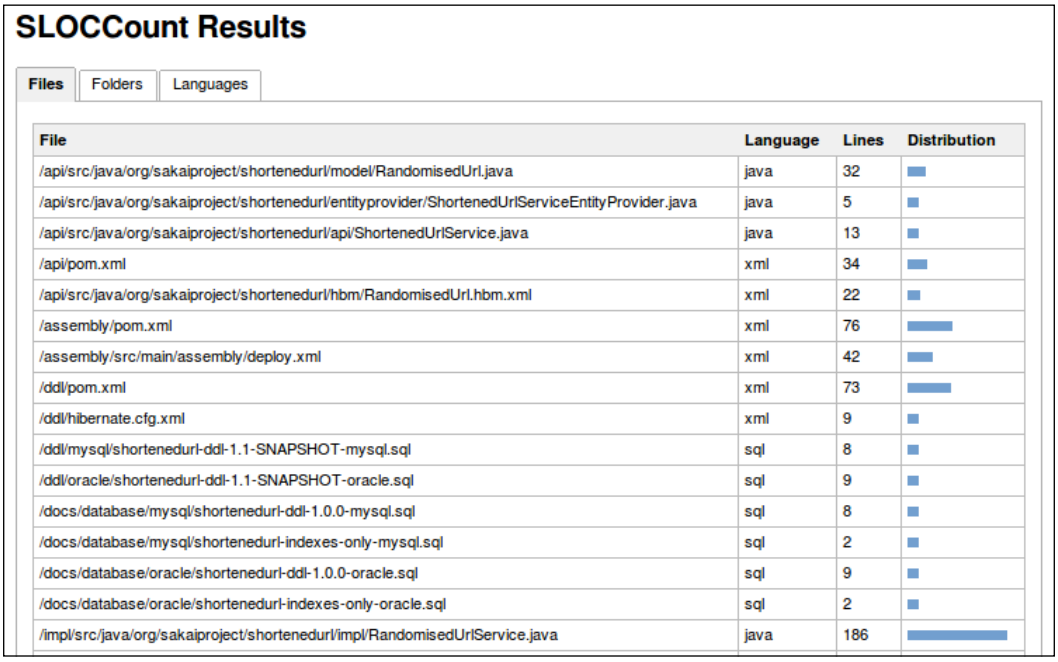
1. Create a freestyle project naming it `ch5.quality.sloccount`, and add the following code to it:

```
<h2>SLOCCOUNT REPORT Project</h2>
<h3>Compared to wider Sakai project</h3>
<script type="text/javascript" src
  =http://www.ohloh.net/p/3551/widgets/project_languages.js>
</script>
```

2. Under the **Source Code Management** section, check **Subversion**, adding for the **Repository URL**: `https://source.sakaiproject.org/svn/shortenedurl/trunk`.
3. Within the **Build** section, select **Execute shell** from **Add build step**. Add the following command to it:

```
/usr/bin/sloccount –duplicates –wide –details . >./sloccount.sc
```

4. In the **Post-build Actions** section, check **Publish SLOCCount analysis results**, adding `sloccount.sc` to the text input **SLOCCount reports**.
5. Click on **Save**.
6. Run the job, and review the details.



SLOCCount Results

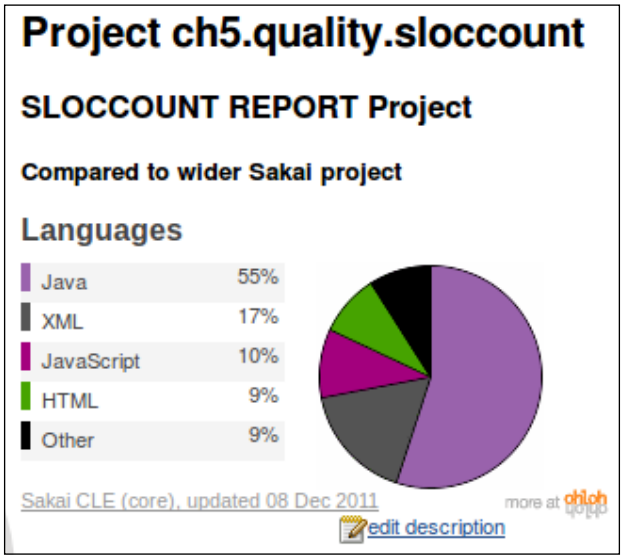
Files | Folders | Languages

File	Language	Lines	Distribution
/api/src/java/org/sakaiproject/shortenedurl/model/RandomisedUrl.java	java	32	
/api/src/java/org/sakaiproject/shortenedurl/entityprovider/ShortenedUrlServiceEntityProvider.java	java	5	
/api/src/java/org/sakaiproject/shortenedurl/api/ShortenedUrlService.java	java	13	
/api/pom.xml	xml	34	
/api/src/java/org/sakaiproject/shortenedurl/hbm/RandomisedUrl.hbm.xml	xml	22	
/assembly/pom.xml	xml	76	
/assembly/src/main/assembly/deploy.xml	xml	42	
/ddl/pom.xml	xml	73	
/ddl/hibernate.cfg.xml	xml	9	
/ddl/mysql/shortenedurl-ddl-1.1-SNAPSHOT-mysql.sql	sql	8	
/ddl/oracle/shortenedurl-ddl-1.1-SNAPSHOT-oracle.sql	sql	9	
/docs/database/mysql/shortenedurl-ddl-1.0.0-mysql.sql	sql	8	
/docs/database/mysql/shortenedurl-indexes-only-mysql.sql	sql	2	
/docs/database/oracle/shortenedurl-ddl-1.0.0-oracle.sql	sql	9	
/docs/database/oracle/shortenedurl-indexes-only-oracle.sql	sql	2	
/impl/src/java/org/sakaiproject/shortenedurl/impl/RandomisedUrlService.java	java	186	

How it works...

The recipe pulls in realistic code, a Java-based service that makes shortened URLs (`https://confluence.sakaiproject.org/display/SHRTURL`). The Jenkins plugin converts the results generated by sloccount into detailed information. The report is divided into a three-tabbed table summed and sorted by files, folders, and languages. From this information, you can estimate the degree of effort it would take to recreate the project from scratch.

The description of the Job includes a small amount of JavaScript pointing to Ohloh.net, a trusted third-party service. **Ohloh** allows you to add widgets to your web page with statistics. Ohloh is a well-known service with well-described privacy rules (`http://www.ohloh.net/about/privacy`). However, if you do not have complete trust in the reputation of a third-party service, then don't link in through a Jenkins description.



Information about the Sakai Learning Management System can be found by visiting `http://www.ohloh.net/p/3551`. The `shortenedURL` service is one small part of this whole. The combined statistics allows visitors to gain a better understanding of the wider context.

There's more...

Here are a few more details to consider.

Software cost estimation

Sloccount uses the **COCOMO model** (`http://en.wikipedia.org/wiki/COCOMO`) to estimate the cost of projects. You will not see this in the Jenkins report, but you can generate the estimated costs if you run sloccount from the command line.

Cost is estimated as `effort * personcost * overhead`.

The element that changes the most over time is `personcost` (in dollars). You can change the value with the command-line argument `personcost`.

Goodbye Google code search, hello Koders.com

Google has announced that it has closed its source code search engine. Luckily, koders.com, another viable search engine, announced that it will provide coverage of the code bases described at ohloh.net. Between koders.com and ohloh.net, you will be able to review a significant selection of open source projects.

See also

- *Knowing your audience with Google Analytic*, *Chapter 4*, *Communicating Through Jenkins*

Looking for "smelly" code through code coverage

This recipe uses **Cobertura** (`http://cobertura.sourceforge.net/`) to find the code that is not covered by unit tests.

With consistent practice, writing unit tests will become as difficult as writing debugging information to `stdout`. Most popular Java-specific IDE's have built-in support for running unit tests. Maven runs them as part of the test goal. If your code does not have regression tests, the code is more likely to break during refactoring. Measuring code coverage can be used to search for hotspots of non-tested code.

For more information, you can review `http://onjava.com/onjava/2007/03/02/statement-branch-and-path-coverage-testing-in-java.html`.

Getting ready

Install the Cobertura code coverage plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Cobertura+Plugin`).

How to do it...

1. Generate a template project by using the following command:

```
mvn archetype:generate -DgroupId=nl.berg.packt.coverage
-DartifactId=coverage -DarchetypeArtifactId=maven-archetype-
quickstart -Dversion=1.0-SNAPSHOT
```

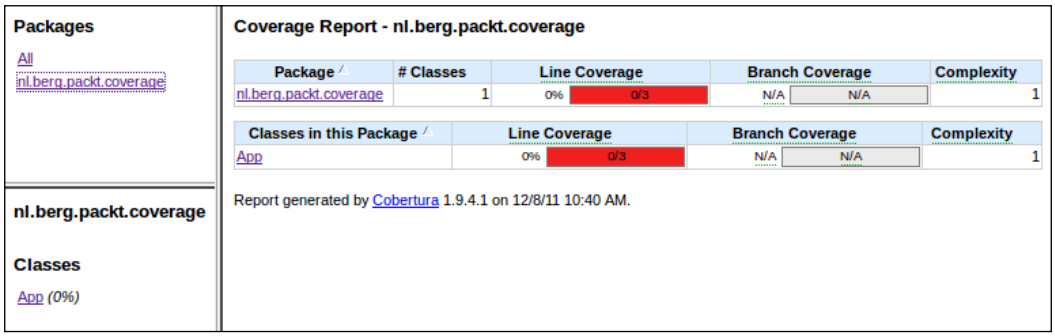
2. Test the code coverage of the unmodified project with the following command:

```
mvn clean cobertura:cobertura
```

3. Review the output from Maven, and it will look similar to the following:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running nl.berg.packt.coverage.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.036 sec

Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] [cobertura:cobertura {execution: default-cli}]
[INFO] Cobertura 1.9.4.1 - GNU GPL License (NO WARRANTY) –
Cobertura: Loaded information on 1 classes.
Report time: 107ms

[INFO] Cobertura Report generation was successful.
```

4. In a web browser, view `/target/site/cobertura/index.html`. Notice that there is no code coverage.



5. Add the following content to `src/main/java/nl/berg/packt/coverage/Dicey.java`:

```
package nl.berg.packt.coverage;
import java.util.Random;
public class Dicey {
  private Random generator;
  public Dicey(){
    this.generator = new Random();
    throwDice();
  }
```

```
  private int throwDice() {
    int value = generator.nextInt(6) + 1;
    if (value > 3){
      System.out.println("Dice > 3");
    }else{
      System.out.println("Dice < 4");
    }
    return value;
  }
}
```

6. Modify `src/test/java/nl/berg/packt/coverage/AppTest.java` to instantiate a new Dicey object, by changing the `testApp()` method to the following:

```
Public void testApp(){
  new Dicey();
  assertTrue( true );
}
```

7. Test the code coverage of the JUnit tests with the following command:

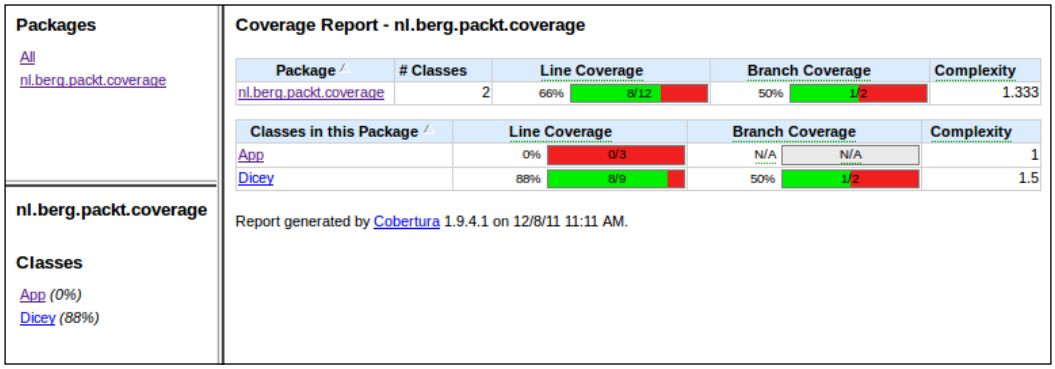
```
mvn clean cobertura:cobertura
```

8. Review the Maven output, noticing that `println` from within the Dicey constructor is also included.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
```

Running nl.berg.packt.coverage.AppTest
Dice < 4
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.033 sec

9. In a web browser, view `/target/site/cobertura/index.html`. Your project now has the code coverage, and you can see which lines of code have not yet been called.



10. Add the following build section to your `pom.xml`:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>cobertura-maven-plugin</artifactId>
      <version>2.5.1</version>
      <configuration>
        <formats>
          <format>xml</format>
          <format>html</format>
        </formats>
      </configuration>
    </plugin>
  </plugins>
</build>
```

11. Test the code coverage of the JUnit tests with the following command:

```
mvn clean cobertura:cobertura
```

12. Visit the location `target/site/cobertura`, noting that results are now also being stored in `coverage.xml`.
13. Run `mvn clean` to remove the target directory.
14. Add the Maven project to your subversion repository.
15. Create a new freestyle Jenkins Job named `ch5.quality.coverage`.
16. Under the **Source Code Management** section, check **Subversion**, adding your subversion repository location for the **Repository URL**.
17. *Under the* **build** section for **Goals and Options**, set the value to clean cobertura: cobertura.
18. Under the **Post-Build actions** section, check **Pubish Cobertura Coverage Report**, adding `**/target/site/cobertura/coverage.xml` for the **Cobertura xml report pattern** input.
19. Click on **Save**.
20. Build the Job twice; this will generate a trend. Review the results.

How it works...

Cobertura instruments the Java bytecode during compilation. The Maven plugin generates both HTML and XML reports. The **HTML report** allows you to quickly review the code status from the command line. The **XML report** is needed for parsing by the Jenkins plugin.

You had placed the plugin configuration in the build section rather than the reporting section to avoid having to run the site goal with its extra phases.

The free-style project was used so that the cobertura plugin picks up multiple XML reports. This was defined by the pattern `fileset **/target/site/cobertura/coverage.xml`, which states that any report that is called `coverage.xml` under any `target/site/cobertura` directory underneath the workspace will be processed.

Maven ran `clean cobertura:cobertura`. The `clean` goal removes all target directories including any previously compiled and instrumented code. The `cobertura:cobertura` goal compiles and instruments the code, runs unit tests, and generates a report.

The `testApp` unit test called the constructor for the Dicey class. The constructor randomly generates a number from `1` to `6`. This mimics a dice and chooses between two branches of an `if` statement. The cobertura report allows you to zoom in to the source code and discover which choice was made. The report is good for identifying missed tests. If you refactor the code, you will not have unit tests in these areas to spot when the code accidentally changes behavior. The report is also good at spotting the code of greater complexity than its surroundings. The more complex the code, the harder it is to understand, and the easier it is to introduce mistakes (`http://www.ibm.com/developerworks/java/library/j-cq01316/index.html?ca=drs`).

There's more...

An alternative open source tool to Cobertura is **emma**: `http://emma.sourceforge.net`. Emma also has an associated Jenkins plugin: `https://wiki.jenkins-ci.org/display/JENKINS/Emma+Plugin`. In Maven, you do not have to add any configuration to `pom.xml`. You simply need to run the goals `clean emma:emma package`, and point the Jenkins plugin at the results.

Activating more PMD rulesets

PMD has rules for capturing particular bugs. It bundles those rules into **rulesets**. For example, there is a ruleset with a theme about Android programming another for code size or design. By default, three non-controversial PMD rulesets are measured:

- **Basic**: Obvious practices that every developer should follow, such as don't ignore the Exceptions that are caught
- **Unusedcoded**: Finds code that is never used lines that can be eliminated, avoiding waste and aiding readability
- **Imports**: Spots unnecessary imports

This recipe shows you how to enable more rules. The main risk is that the extra rules generate a lot of false positives, making it difficult to see real defects. The benefit is that you will capture a wider range of defects, some of which are costly if they get to production.

Getting ready

Install the Jenkins PMD plugin (`https://wiki.jenkins-ci.org/display/JENKINS/PMD+Plugin`).

How to do it...

1. Generate a template project by using the following command:

   ```
   mvn archetype:generate -DgroupId=nl.berg.packt.pmd
   -DartifactId=pmd -DarchetypeArtifactId=maven-archetype-quickstart
   -Dversion=1.0-SNAPSHOT
   ```

2. Add the Java Class `src/main/java/nl/berg/packt/pmd/PMDCandle.java` with the following content:

   ```
   package nl.berg.packt.pmd;
   import java.util.Date;

   public class PMDCandle {
     private String MyIP = "123.123.123.123";

     public void dontDontDoThisInYoourCode(){
       System.out.println("Logging Framework please");
       try {
         int x =5;
       }catch(Exception e){}
       String myString=null;
       if (myString.contentEquals("NPE here"));

     }

   }
   ```

3. Test your unmodified project with the command:

   ```
   mvn clean pmd:pmd
   Review the directory target, and you will notice the results
   basic.xml, imports.xml, unusedcode.xml, and the aggregated results
   pmd.xml.
   ```

4. View the `target/site/pmd.html` file in a web browser.

5. Add the following reporting section to your `pom.xml`:

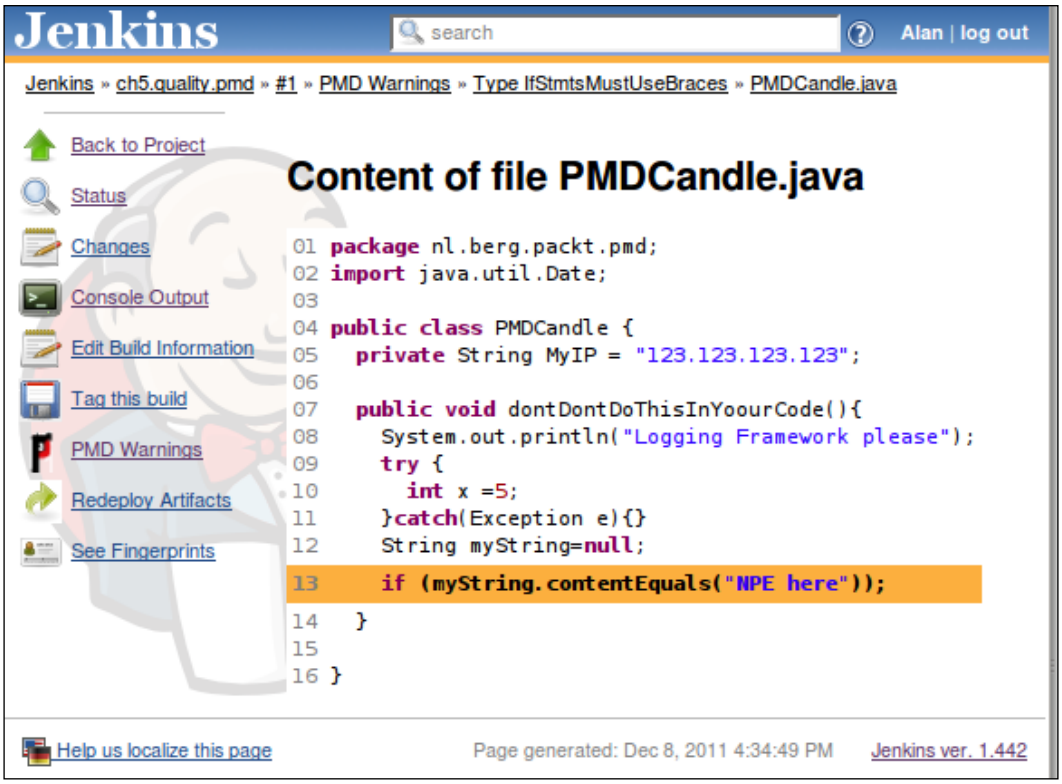
```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jxr-plugin</artifactId>
      <version>2.1</version>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-pmd-plugin</artifactId>
      <version>2.6</version>
      <configuration>
        <targetJdk>1.5</targetJdk>
        <format>xml</format>
        <linkXref>true</linkXref>
        <minimumTokens>100</minimumTokens>
        <rulesets>
          <ruleset>/rulesets/basic.xml</ruleset>
          <ruleset>/rulesets/braces.xml</ruleset>
          <ruleset>/rulesets/imports.xml</ruleset>
          <ruleset>/rulesets/logging-java.xml</ruleset>
          <ruleset>/rulesets/naming.xml</ruleset>
          <ruleset>/rulesets/optimizations.xml</ruleset>
          <ruleset>/rulesets/strings.xml</ruleset>
          <ruleset>/rulesets/sunsecure.xml</ruleset>
          <ruleset>/rulesets/unusedcode.xml</ruleset>
        </rulesets>
      </configuration>
    </plugin>
  </plugins>
</reporting>
```

6. Test your project with the following command:

```
mvn clean site
```

7. View the `target/site/pmd.htm` file in a web browser, and notice that extra violations have now been found. This is due to the extra rules added to `pom.xml`.
8. Run `mvn clean` to remove the target directory.
9. Add the source code to your subversion repository.
10. Create a new Maven 2/3 Jenkins Job named `ch5.quality.pmd`
11. Under the **Source Code Management** section, check **Subversion**, adding your subversion repository location for the **Repository URL**.

12. Within the **build** section for **Goals and options**, set the value to `clean site`.
13. Under the **Build Settings** section, check **Publish PMD analysis results**.
14. Click on **Save**.
15. To generate a trend, you will need to run the Job twice. Afterwards, review the results.



How it works...

The Maven PMD plugin tested a wide range of rulesets. By downloading the binary package from the PMD website, you can find the paths of the rulesets by listing the contents of the `pmd.jar` file. Under a `*NIX` system, the command to do this is:

```
unzip -l pmd-version.jar | grep rulesets
```

You had added a standard candle, a Java class with known defects that trigger PMD warnings. For example, there are multiple defects in the following two lines of code:

```
String myString=null;
if (myString.contentEquals("NPE here"));
```

The most significant defect is that a Java programmer needs to place the literal first, to avoid a `NullPointerException`, for example:

```
"NPE here".contentEquals(myString)
```

Placing the literal first returns `false` when `myString` is `null`. There is an issue with the lack of braces around the `if` statement. The same counts for the lack of a command to run when the `if` statement is triggered.

Another trivial example is hardcoding infrastructural details into your source; for example, passwords, IP addresses, and usernames. It is far better to move the details out into property files that reside only on the deployment server. The following line tests PMD for its ability to find this type of defect:

```
private String MyIP = "123.123.123.123";
```

Both FindBugs and PMD have their own set of bug pattern detectors. Neither will capture the full range of defects. It is, therefore, worth running both the tools to capture the widest range of defects. For a review of both products, visit `http://www.freesoftwaremagazine.com/articles/destroy_annoying_bugs_part_1`.

A couple of other static code review tools you may be interested in are **QJPro** (`http://qjpro.sourceforge.net/`) and **Jlint** (`http://jlint.sourceforge.net/`).

There's more...

Out of the box, PMD tests for a sensible set of bug defects, however, each project is different, and you will need to tweak.

Throttling down PMD rulesets

You can find the current PMD rulesets at `http://pmd.sourceforge.net/rules/index.html`.

It is important to understand the meaning of the rulesets, and shape the Maven configuration to include only the useful ones. If you do not do this, then for a medium-sized project, the report will include thousands of violations, hiding the real defects. The report will then take time to render in your web browser. Consider enabling a long list of rules only if you want to use the volume as an indicator of project maturity.

To throttle down, exclude parts of your code, and systematically clean up the areas reported.

The "don't repeat yourself" principle

Cut-and-paste programming, cloning, and then modifying code makes for a refactoring nightmare. If the code is not properly encapsulated, it is easy to have slightly different pieces scattered across your code base. If you then want to remove known defects, it will require extra effort.

PMD supports the **Don't Repeat Yourself** (**DRY**) principle by finding duplicate code. The trigger point is configured through the `minimumTokens` tag. However, the PMD plugin does not pick up the results (stored in `cpd.xml`). You will need to either install and configure the DRY plugin (`https://wiki.jenkins-ci.org/display/JENKINS/DRY+Plugin`) or the Violations Jenkins plugin.



If you have downloaded the PMD binary from its website (`http://sourceforge.net/projects/pmd/files/pmd/`), then in the `bin` directory is `cpdgui`. It is a Java swing application that allows you to explore your source code for duplications.

See also

- *Creating custom PMD rules*

Creating custom PMD rules

PMD has two extra features when compared to other static code review tools. The first is the `cpdgui` tool, which allows you to look for the code that has been cut-and-pasted from part of the code base to another. The second, and the one that we will explore in this recipe, is the ability to design custom bug discovery rules for Java source code using Xpath.

Getting ready

Make sure that you have installed the Jenkins PMD plugin (`https://wiki.jenkins-ci.org/display/JENKINS/PMD+Plugin`). Download and unpack the PMD distribution from `http://pmd.sourceforge.net`. Visit the PMD bin directory, and verify that you have the startup scripts `designer.sh` and `designer.bat`.

How to do it...

1. Create a Maven project from the command line using:

```
mvn archetype:generate -DgroupId=nl.berg.packt.pmdrule
-DartifactId=pmd_design -DarchetypeArtifactId=maven-archetype-
quickstart -Dversion=1.0-SNAPSHOT
```

2. In the `pom.xml` just before the `</project>` tag, add a reporting section with the following content:

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jxr-plugin</artifactId>
```

```
      <version>2.1</version>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-pmd-plugin</artifactId>
      <version>2.6</version>
      <configuration>
        <targetJdk>1.6</targetJdk>
        <format>xml</format>
        <rulesets>
          <ruleset>password_ruleset.xml</ruleset>
        </rulesets>
      </configuration>
    </plugin>
  </plugins>
</reporting>
```

3. In the top-level directory, create the file `password_ruleset.xml` with the content:

```
<?xml version="1.0"?>
<ruleset name="STUPID PASSWORDS ruleset"
  xmlns="http://pmd.sf.net/ruleset/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://pmd.sf.net/ruleset/1.0.0
  http://pmd.sf.net/ruleset_xml_schema.xsd"
  xsi:noNamespaceSchemaLocation=
  "http://pmd.sf.net/ruleset_xml_schema.xsd">
  <description>
    Lets find stupid password examples
  </description>
</ruleset>
```

4. Edit `src/main/java/nl/berg/packt/pmdrule/App.java`, so that the main method is:

```
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
        String PASSWORD="secret";
     }
```

5. Depending on your operating system, run `pmd designer` using either the startup script `bin/designer.sh` or `bin/designer.bat`.
6. Click on the **JDK** option at the top left of the screen, selecting **JDK 1.6** as the **Java version**.

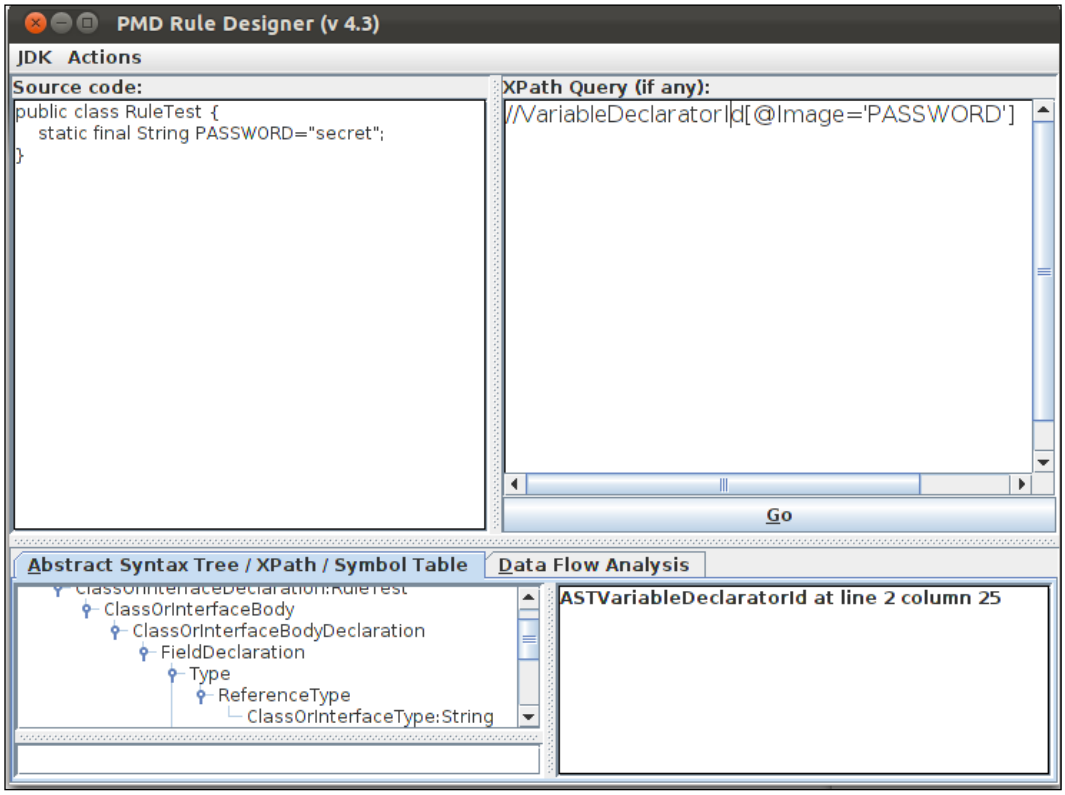
7. In the **Source Code** text area, add the example code you want to test against. In this example:

```
public class RuleTest {
  static final String PASSWORD="secret";
}
```

8. For the **Query** `(if any)` text area add:

```
//VariableDeclaratorId[@Image='PASSWORD']
```

9. Click on **Go**. You will now see the result **ASTVariableDeclarorID** at *line 2 column 20*.



10. Under the **Actions** menu option at the top of the screen, select **Create rule XML**, and add the following values:
    - **Rule name**: `No Password`
    - **Rule msg**: `If we see a PASSWORD we should flag`
    - **Rule desc**: `Let's find stupid password examples`

11. Click on **Create rule XML**. The generated XML should have a fragment similar to the following:

```
<rule  name="NO_PASSWORD"
  message="If we see a PASSWORD we should flag"
  class="net.sourceforge.pmd.rules.XPathRule">
  <description>
  If we see a PASSWORD we should flag
  </description>
  <properties>
    <property name="xpath">
    <value>
<![CDATA[
//VariableDeclaratorId[@Image='PASSWORD']

]]>
    </value>
    </property>
  </properties>
  <priority>3</priority>
  <example>
<![CDATA[
public class RuleTest {
    static final String PASSWORD="secret";
}
]]>
  </example>
</rule>
```



12. Copy-and-paste the generated code into `password_ruleset.xml` just before `</ruleset>`.
13. Commit the project to your subversion repository.
14. In Jenkins, create a Maven 2/3 Job named `ch5.quality.pmdrule`.
15. Under the **Source Code Management** section, check **Subversion**, adding your subversion repository location for the **Repository URL**.
16. Within the **build** section for **Goals and Options**, set the value to `clean pmd:pmd`.
17. In the **Build Settings** section, check **Publish PMD analysis results**.
18. Click on **Save**.
19. Run the Job.
20. Review the **PMD Warnings** link.

How it works...

PMD analyzes the source code and breaks it down into meta-data known as an **Abstract Syntax Tree** (**AST**) - `http://onjava.com/pub/a/onjava/2003/02/12/static_analysis.html`. PMD has the ability to use Xpath rules to search for patterns in the AST. **w3schools** provides a gentle introduction to Xpath (`http://www.w3schools.com/xpath/`). The **designer tool** enables you to write Xpath rules and tests your rules against a source code example. For readability, it is important that the source code you test against contains only the essential details. The rules are then stored in XML.

To bundle the XML rules together, you have to add the rules as part of a `<ruleset>` tag.

The Maven PMD plugin has the ability to read the rulesets from within its classpath, on the local file system or through the `http` protocol from a remote server. You added your ruleset by adding the configuration option.

```
<ruleset>password_ruleset.xml</ruleset>
```

If you build up a set of rules, then you should pull all the rules into one project for ease of management.

You can also create your own custom ruleset based on already existing rules, pulling out your favorite bug detection patterns. This is achieved by the `<rule>` tag with a `ref` pointing to the known rule; for example, the following pulls out the `DuplicateImports` rule from the `imports.xml ruleset`:

```
<rule ref="rulesets/imports.xml/DuplicateImports"/>
```

The rule generated in this recipe tested for variables with the name `PASSWORD`. I have seen the rule trigger a number of times in real projects.

The PMD home page is a great place to learn about what is possible with the Xpath rules. It contains descriptions and details of the rulesets; for example, for the logging rules, review `http://pmd.sourceforge.net/rules/logging-java.html`.

There's more...

It would be efficient if static code review tools could make recommendations about how to fix the code. However, that is a little dangerous as the detectors are not always accurate. As an experiment, I have written a small Perl script to first repair the literals and also discard some wasting of resources. The code is a "Proof Of Concept", and thus is not guaranteed to work correctly. It has the benefit of being succinct, see

`https://source.sakaiproject.org/contrib/qa/trunk/static/cleanup/easy_wins_find_java.pl`.

See also

- *Activating more PMD rulesets*

Finding bugs with FindBugs

It is easy to get lost in the volume of defects found by static code review tools. Another Quality Assurance attack pattern is to clean up the defects package by package, concentrating developer time on the most used features.

This recipe will show you how to generate and report defects found by FindBugs for specific packages.

Getting ready

Install the Jenkins FindBugs plugin (`https://wiki.jenkins-ci.org/display/JENKINS/FindBugs+Plugin`).

How to do it...

1. From the command line, create a Maven project:

```
mvn archetype:generate -DgroupId=nl.berg.packt.FindBugs_all
-DartifactId=FindBugs_all -DarchetypeArtifactId=maven-archetype-
quickstart -Dversion=1.0-SNAPSHOT
```

2. In `pom.xml`, add a build section just before the `</project>` tag, with the following content:

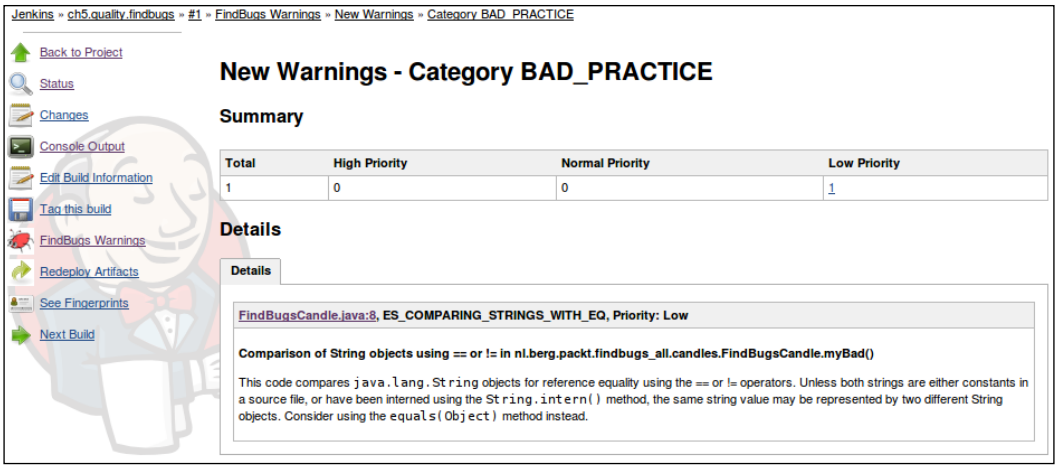
```
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>FindBugs-maven-plugin</artifactId>
      <version>2.3.3</version>
      <configuration>
        <FindBugsXmlOutput>true</FindBugsXmlOutput>
        <FindBugsXmlWithMessages>true</FindBugsXmlWithMessages>
        <onlyAnalyze>
          nl.berg.packt.FindBugs_all.candles.*
        </onlyAnalyze>
        <effort>Max</effort>
      </configuration>
    </plugin>
  </plugins>
</build>
```

3. Create the directory `src/main/java/nl/berg/packt/FindBugs_all/candles`.
4. In the candles directory, include `FindBugsCandle.java` with the following content:

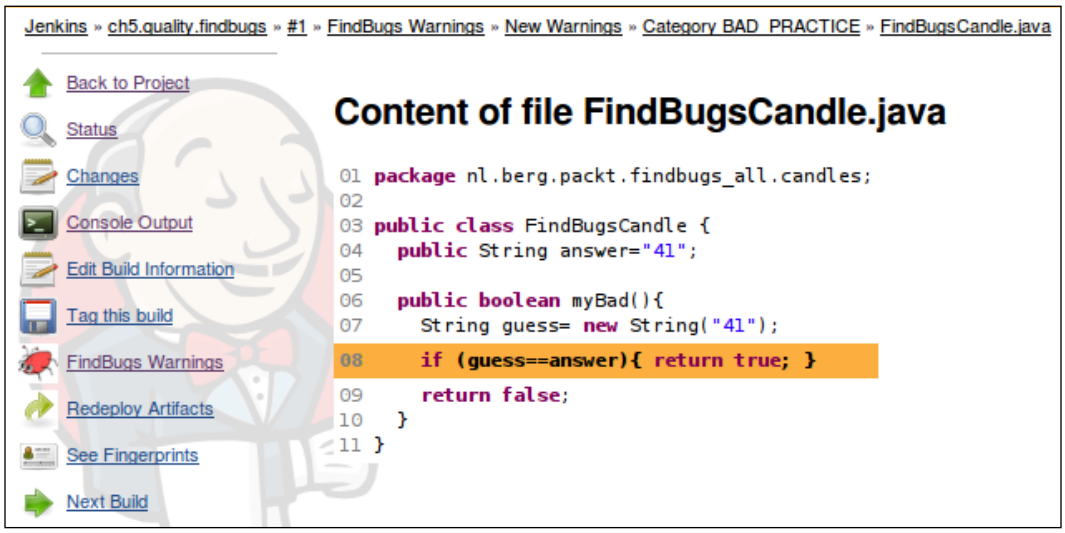
```
package nl.berg.packt.FindBugs_all.candles;

public class FindBugsCandle {
  public String answer="41";

  public boolean myBad(){
    String guess= new String("41");
    if (guess==answer){ return true; }
    return false;
  }
}
```

5. Create a free-style project with the name `ch5.quality.FindBugs`.
6. Under the **Source Code Management** section, check the **Subversion** radio box adding `Your Repository URL` to the **Repository URL**.
7. Within the **build** section for **Goals and options**, set the value to `clean compile findBugs:findBugs`.
8. Under the **Build settings** section, check **Publish FindBugs analysis results**.
9. Click on **Save**.

10. Run the Job.
11. Review the results.



A handy feature of the FindBugs report is that for each warning you can view the offending code.



How it works...

In this recipe, you have created a standard Maven project and added a Java file with known defects.

The `pom.xml` configuration forces `FindBugs` to report defects from the classes in the `nl.berg.packt.FindBugs_all.candles` package only.

The line in the standard candle with `guess==answer` is a typical programming defect. Two references to objects are being compared rather than the values of their Strings. As the guess object was created on the previous line, the result will always be `false`. These sorts of defects can appear as subtle problems in programs. The JVM caches Strings, and occasionally two apparently different objects are actually the same object.

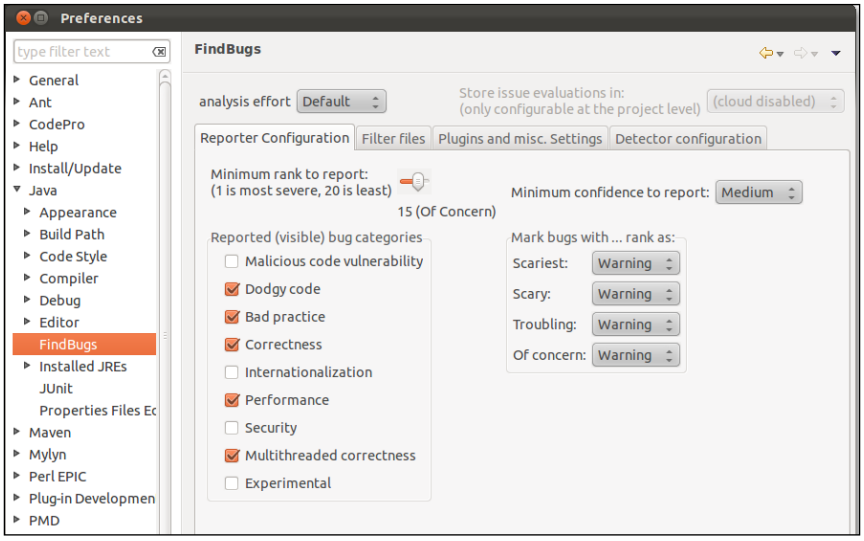
There's more...

FindBugs is popular among developers and has plugins for a number of popular IDEs. Its results are often used as part of wider reporting by other tools.

The FindBugs Eclipse plugin

The automatic install location for the Eclipse plugin is `http://findbugs.cs.umd.edu/eclipse`.

By default, the FindBugs Eclipse plugin has a limited number of rules enabled. To increase the set tested, you will need to go to the **Preferences** menu option under **Window**, selecting **FindBugs** from the left-hand menu. On the right-hand side, you will see the **Reported (Visible) bug categories** under **Reporter Configuration**. You can now tweak the visible categories.



The Xradar and Maven dashboards

There are alternative Maven plugin dashboards for the accumulation of generated software metrics. The Maven dashboard is one example (`http://mojo.codehaus.org/dashboard-maven-plugin/`). You will need to connect it to its own database. There is a recipe for this in *Apache Maven 3 Cookbook*, *Srirangan, Packt Publishing* (`http://www.packtpub.com/apache-maven-3-0-cookbook`), *Setting up the Maven dashboard*, *Chapter 4*, *Reporting and Documentation*.

Xradar and **QALab** are other, arguably less popular, examples of dashboards. (`http://xradar.sourceforge.net/usage/maven-plugin/howto.html`, `http://qalab.sourceforge.net/multiproject/maven2-qalab-plugin/index.html`).

See also

- *Enabling extra FindBugs rules*
- *Finding security defects with FindBugs*
- *Activating more PMD rulesets*

Enabling extra FindBugs rules

FindBugs has a wide range of auxiliary bug pattern detectors. These detectors are bundled into one contributor project hosted at **SourceForge** (`http://sourceforge.net/projects/fb-contrib/`).

This recipe details how to add the extra bug detectors to FindBugs from the `fb-contrib` project and use the detectors to capture known defects.

Getting ready

It is assumed that you have followed the previous recipe, *Finding bugs with FindBugs*. You will be using the recipe's Maven project as a starting point.

How to do it...

1. Copy the top-level `pom.xml` configuration to `pom_fb.xml`.
2. Replace the FindBugs `<plugin>` section of `pom_fb.xml` with the following content:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>FindBugs-maven-plugin</artifactId>
  <version>2.3.3</version>
  <configuration>
```

```
    <FindBugsXmlOutput>false</FindBugsXmlOutput>
    <FindBugsXmlWithMessages>true</FindBugsXmlWithMessages>
    <onlyAnalyze>
      nl.berg.packt.FindBugs_all.candles.*
    </onlyAnalyze>
    <pluginList>
      http://downloads.sourceforge.net/project/
        fb-contrib/Current/fb-contrib-4.6.1.jar
    </pluginList>
    <effort>Max</effort>
  </configuration>
</plugin>
```

3. In the `src/main/java/nl/berg/packt/fingbugs_all/candles` directory, add the Java class `FindBugsFBCandle.java` with the following content:

```
package nl.berg.packt.FindBugs_all.candles;

public class FindBugsFBCandle {
  public String FBexample(){
    String answer="This is the answer";
    return answer;
  }
}
```

4. Commit the updates to your subversion repository.
5. Create a Jenkins Maven 2/3 Job with the name `ch5.quality.FindBugs.fb`.
6. Under the **Source Code Management** section, check the **Subversion** radio box, adding the URL to your code for the **Repository URL**.
7. In the **Build** section, set the following:
   - **Root POM**: `pom_fb.xml`
   - **Goals and options**: `clean compile findbugs:findbugs`
8. Under the **Build Settings** section, check **Publish FindBugs analysis results**.
9. Click on **Save**.
10. Run the Job.
11. When the Job is finished building, review the **FindBugs Warnings** link. You will now see a new warning:

    USBR_UNNECESSARY_STORE_BEFORE_RETURN

There's more...

The Java language has a number of subtle boundary cases that are difficult to understand until explained by real examples. An excellent way to capture knowledge is to write examples yourself, when you see issues in your code. Injecting standard candles is a natural way of testing your team's knowledge and makes for target practice during the QA process.

The `FindBugs` project generated some of their detectors, based on the content of *Java puzzlers*, *Joshua Bloch* and *Neal Gafter* (`http://www.javapuzzlers.com/`).

How it works...

To include external detectors, you added an extra line to FindBugs' Maven configuration:

```
<pluginList>
  http://downloads.sourceforge.net/project/fb-contrib/Current/
    fb-contrib-4.6.1.jar
</pluginList>
```

It is worth visiting SourceForge to check for the most up-to-date version of the detectors.

Currently, it is not possible to use Maven's dependency management to pull in the detectors through from a repository, though this might change.

In this recipe, you have added a Java class to trigger the new bug detection rules. The anti-pattern is the unnecessary line with the creation of the answer object before the return. It is more succinct to return the object anonymously, for example:

```
Return "This is the answer";
```

The ant-pattern triggers the `USBR_UNNECESSARY_STORE_BEFORE_RETURN` pattern, which is described on the home page of the `fb-contrib` project.

See also

- *Finding bugs with FindBugs*
- *Finding security defects with FindBugs*
- *Activating more PMD rulesets*

Finding security defects with FindBugs

In this recipe, you will use FindBugs to discover a security flaw in a Java Server Page and some more security defects in a defective Java class.

Getting ready

Either follow the recipe *Failing Jenkins Jobs based on JSP syntax errors*, *Chapter 3*, *Building Software*, or use the provided project downloadable from the Packt website.

How to do it...

1. Edit `pom.xml` by just swapping the `<plugins>` within `<build>` to include the `FindBugs` plugin, by adding the following content:

```
<plugins>
  <plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>findBugs-maven-plugin</artifactId>
    <version>2.3.3</version>
    <configuration>
      <FindBugsXmlOutput>true</FindBugsXmlOutput>
      <FindBugsXmlWithMessages>true</FindBugsXmlWithMessages>
      <effort>Max</effort>
    </configuration>
  </plugin>
</plugins>
```

2. Create the directory structure `src/main/java/nl/berg/packt/finbugs_all/candles`.
3. Add the Java file `FindBugsSecurity.java` with the following content:

```
package nl.berg.packt.FindBugs_all.candles;

public class FindBugsSecurityCandle {
  private final String[] permissions={"Read", "SEARCH"};
  private void infiniteLoop(int loops){
    infiniteLoop(99);
  }

  public String[] exposure(){
    return permissions;
  }

  public static void main(String[] args) {
    String[] myPermissions = new
      FindBugsSecurityCandle().exposure();
    myPermissions[0]="READ/WRITE";
    System.out.println(myPermissions[0]);
  }
}
```

4. Commit the updates to your subversion repository.
5. Create a Maven 2/3 Jenkins Job with the name `ch5.quality.FindBugs.security`.
6. Under the **Source Code Management** section, check the **Subversion** radio box, adding your subversion repository location in the **Repository URL** text input.
7. Beneath the **Build** section for **Goals and options**, set the value to `clean package findbugs:findbugs`.
8. Click on **Save**.
9. Run the Job.
10. When the Job has completed, review the link **FindBugs Warning**. Notice that the JSP package exists with one warning for `XSS_REQUEST_PARAMETER_TO_JSP_WRITER`. However, the link fails to find the location of the source code.
11. Copy `src/main/webapp/index.jsp` to `jsp/jsp.index_jsp`.
12. Commit to your subversion repository.
13. Run the Job again.
14. View the results under the **FindBugs Warning** link. You will now be able to view the JSP source code.

How it works...

JSP's are first translated from text into Java source code and then compiled. FindBugs works by parsing the compiled Java bytecode.

The original JSP project has a massive security flaw—it trusts input from the Internet. This leads to a number of attack vectors, including **XSS attacks** (`http://en.wikipedia.org/wiki/Cross-site_scripting`). Parsing the input with white lists of allowed tokens is one approach to diminishing the risk. FindBugs discovers the defect and warns with `XSS_REQUEST_PARAMETER_TO_JSP_WRITER`. The Jenkins FindBugs plugin details the bug type. The messages are displayed because you had turned them on in the configuration with the following option:

```
<FindBugsXmlWithMessages>true</FindBugsXmlWithMessages>
```

The FindBugs plugin has not been implemented to understand the location of JSP files. When clicking on a link to the source code, the plugin will look in the wrong place. A temporary solution is to copy the JSP file to the location the Jenkins plugin expects.

The line number location reported by FindBugs also does not make sense. It is pointing to the line in the `.java` file that is generated from the `.jsp` file, and not directly the JSP file. Despite these limitations, FindBugs discovers useful information about JSP defects.



An alternative for JSP bug detection is PMD. From the command line, you can configure it to scan JSP files only with the option `-jsponly`; see `http://pmd.sourceforge.net/jspsupport.html`.

There's more...

Although FindBugs has rules that sit under the category security, there are other bug detectors that find security-related defects. The standard `candle` class includes two such defects. The first is a recursive loop that will keep calling the same method from within itself:

```
private void infiniteLoop(int loops){
  infiniteLoop(99);
}
```

Perhaps the programmer intended to use a counter to force an exit after 99 loops, but the code to do this does not exist. The end result, if this method is called, is that it will keep calling itself until the memory reserved for the stack is consumed and the application fails. This is also a security issue. If an attacker knows how to reach this code, they can bring down the related application, a **Denial Of Service** attack.

The other attack captured in the standard candle is the ability to change the content within an array that appears to be immutable. It is true that the reference to the array cannot be changed, but the internal references to the array elements can. In the example, a motivated cracker, having access to the internal objects, is able to change the `READ` permissions to `READ/WRITE` permissions. To prevent this situation, consider making a defensive copy of the original array passing the copy to the calling method.



The `OWASP` project provides a wealth of information on the subject of testing security; see:

`https://www.owasp.org/index.php/Category:OWASP_Java_Project`.

See also

- *Finding bugs with FindBugs*
- *Enabling extra FindBugs rules*
- *Activating more PMD rulesets*
- *Configuring Jetty for integration tests Chapter 3, Building Software*

Verifying HTML validity

This recipe tells you how to use Jenkins to test HTML pages for validity against HTML and CSS standards.

You can upload and verify your HTML files against the W3C's unified validator (`http://code.w3.org/unicorn`). The unified validator will check your web pages for correctness against a number of aggregated services. The Jenkins plugin does this for you automatically.

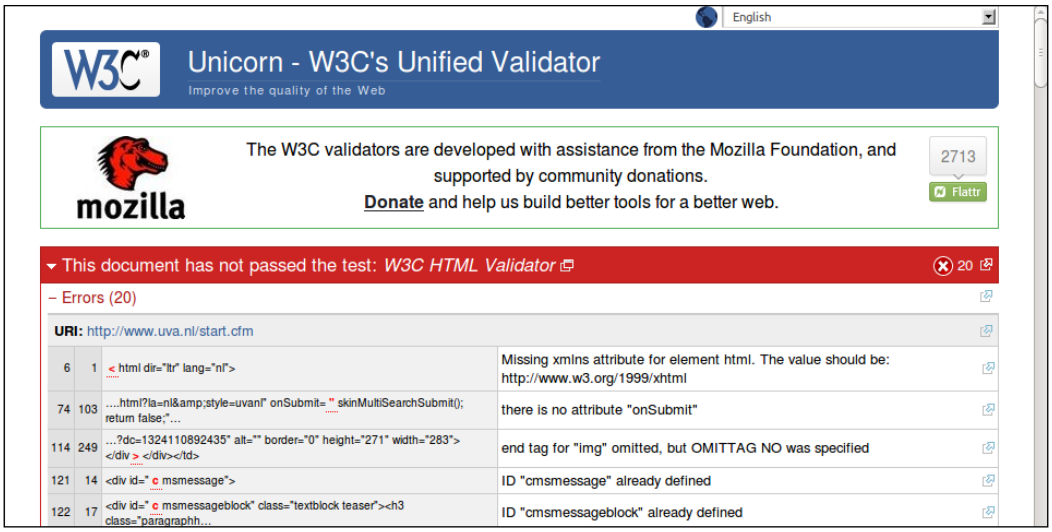
Getting ready

Install the **Unicon Validation plugin** (`https://wiki.jenkins-ci.org/display/JENKINS/Unicorn+Validation+Plugin`). If you have not already done so, also install the **plot plugin** (`https://wiki.jenkins-ci.org/display/JENKINS/Plot+Plugin`).

How to do it...

1. Create a free-style Job with the name `ch5.quality.html`.
2. Within the **build** section, add a build step by selecting **Unicorn Validator**.
3. For the **Site to validate** input, add a URL to a site that you are allowed to test.
4. Click on **Save**.
5. Run the Job.
6. Review the **Workspace** by clicking on the `unicorn_output.html` and `markup-validator_errors.properties`. For the properties file content, you will see content similar to `YVALUE=2`.
7. Configure the project. In the **Post-build Actions** section, check **Plot build data**, and add the following details:
    - **Plot group**: `Validation Errors`
    - **Plot title**: `Markup Validation errors`
    - **Number of builds to Include**: `40`
    - **Plot y-axis label**: `Errors`
    - **Plot style**: `Area`
    - **Data series file**: `markup-validator_errors.properties`
    - Verify that **Load data** from **properties file** radio box is checked
    - **Data series legend label**: `Feed errors`
8. Click on **Save**.

9. Run the Job.
10. Review the **Plot** link.



How it works...

The **Unicon validation** plugin uses the validation service at W3C to generate a report on the URL you configure. The returned report is processed by the plugin, and absolute counts of the defects are taken. The summation is then placed in the property file, where the values are then picked up by the plotting plugin (see the recipe *Plotting alternative code metrics in Jenkins* in *Chapter 3*, *Building Software*). If you see a sudden surge in the warnings, then review the HTML pages for repetitive defects.

There's more...

It is quite difficult to obtain a decent code coverage from unit testing. This is especially true for larger projects where there are a number of teams with varying practices. You can increase your automatic testing coverage of web applications considerably by using tools that visit as many links in your application as possible. This includes HTML validators, link checkers, search engine crawlers, and security tools. Consider setting up a range of tools to hit your applications during integration testing, remembering to parse the logfiles for unexpected errors. You can automate log parsing using the recipe *Deliberately failing builds through log parsing*, *Chapter 1*, *Maintaining Jenkins*.

Reporting with JavaNCSS

JavaNCSS (`http://javancss.codehaus.org/`) is a software metrics tool that calculates two types of information. The first are totals for number of source code lines in a package that are active, commented, or JavaDoc related. The second type calculates the complexity of code, based on how many different decision branches exist.

The Jenkins JavaNCSS plugin ignores the complexity calculation and focuses on the more understandable line counts.

Getting ready

Install the JavaNCSS plugin (`https://wiki.jenkins-ci.org/display/JENKINS/JavaNCSS+Plugin`).

How to do it...

1. Create a Maven 2/3 project named `ch5.quality.ncss`.
2. Under the **Source Code Management** section, check the **Subversion** radio box.
3. Add the **Repository URL** `https://source.sakaiproject.org/contrib/learninglog/tags/1.0`.
4. Review **Build Triggers**, making sure none are activated.
5. Under the **build** section for the **Goals and options** type, clean `javancss:report`.
6. Under the **Build Setting** section, check **Publish Java NCSS Report**.
7. Click on **Save**.
8. Run the Job.
9. Review the **Java NCSS Report**.
10. Review the top-level `pom.xml` configuration in the `Workspace`; for example: `http://localhost:8080job/ch5.quality.ncss/ws/pom.xml`.

Java NCSS Report

Results

Package	Classes	Functions	Javadocs	NCSS	JLC	SLCLC	MLCLC
org.sakaiproject.learninglog.tool	1	2	1	53	3	7	0
org.sakaiproject.learninglog.impl	7	131	8	1161	26	34	56
org.sakaiproject.learninglog.impl.entity	4	45	4	357	13	5	0
org.sakaiproject.learninglog.impl.sql	4	23	0	342	0	7	79
org.sakaiproject.learninglog.api	6	75	0	193	0	4	32
org.sakaiproject.learninglog.api.datamodel	4	53	8	230	33	0	32
org.sakaiproject.learninglog.api.sql	1	15	0	47	0	6	0
org.sakaiproject.learninglog.cover	1	5	1	21	5	0	16
Totals	28	349	22	2404	80	63	215

How it works...

The Job pulled in source code from Sakai's subversion repository. The project is a multi-module with the API separated from the implementation.

JavaNCSS needs no compiled classes or modifications to the Maven `pom.xml`. This makes for a simple cycle. The Job ran a Maven goal, publishing the report through the JavaNCSS Jenkins plugin.

Reviewing the report, it's observed that the implementation has many more numbers of lines of active code relative to other packages. Documentation of APIs is vital for the reuse of the code by other developers. Significantly, there are no JavaDoc lines in the API.

The abbreviations in the summary table have the following meanings:

- **Classes**: Number of classes in the package.
- **Functions**: Number of functions in the package.
- **JavaDocs**: The number of different JavaDoc blocks in the package. This is not fully indicative, as most modern IDEs generate classes using boilerplate templates. Therefore, you can have a lot of JavaDoc generated of poor quality, creating misleading results.
- **NCSS**: Number of non-commented lines of source code.
- **JLC**: Number of lines of JavaDoc.
- **SLCLC**: Number of lines that include only a single comment.
- **MLCLC**: Number of lines of source code that are part of multi-line comments.

The build summary displays information about changes (deltas) between the current and the last Jobs; for example:

```
classes (+28)
functions (+349)
ncss (+2404)
javadocs (+22)
javadoc lines (+80)
single line comments (+63)
multi-line comments (+215)
```

Within the summary, the + character signals that code has been added and the – character that the code has been deleted. If you see a large influx of code, but a lower than usual influx of JavaDoc, then either the code is auto-generated or is more likely being rushed to market.

There's more...

When you get used to the implications of the relatively simple summary of JavaNCSS, consider adding **JDepend** to your safety net of code metrics. JDepend generates a wider range of quality-related metrics (`http://clarkware.com/software/JDepend.html`, `http://mojo.codehaus.org/jdepend-maven-plugin/plugin-info.html` or `https://wiki.jenkins-ci.org/display/JENKINS/JDepend+Plugin`).

One of the most important metrics that JDepend generates is that of cyclic dependency. If class A is dependent on class B, and in turn, class B is dependent on class A, then that is a cyclic dependency. When there is such a dependency, it indicates that there is an increased risk of something going wrong, such as a fight for a resource, an infinite loop, or synchronization issues. Refactoring may be needed to eliminate the lack of clear responsibilities.

Checking style using an external pom.xml

If you just want to check the code style for the quality of its documentation without changing its source, then inject your own `pom.xml` file. This recipe shows you how to do this for checkstyle. **Checkstyle** is a tool that mostly checks documentation against well-defined standards, such as the Sun coding conventions.

Getting ready

Install the checkstyle plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Checkstyle+Plugin`). Create a new directory in your subversion repository for this recipes code.

How to do it...

1. Create a directory named `OVERRIDE`.
2. Create the file `OVERRIDE/pom_checkstyle.xml` with the following content:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>nl.berg.packt.checkstyle</groupId>
  <artifactId>checkstyle</artifactId>
  <packaging>pom</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>checkstyle</name>
```

```
    <url>http://maven.apache.org</url>

    <modules>
      <module>api</module>
      <module>help</module>
      <module>impl</module>
      <module>util</module>
      <module>pack</module>
      <module>tool</module>
      <module>assembly</module>
      <module>deploy</module>
      <module>bundle</module>
    </modules>
    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-checkstyle-plugin</artifactId>
          <version>2.8</version>
        </plugin>
      </plugins>
    </build>
    <properties>
      <project.build.sourceEncoding>
        UTF-8
      </project.build.sourceEncoding>
    </properties>
  </project>
```

3. Commit to the source code to your subversion repository.
4. Create a Maven 2/3 job with the name `ch5.quality.checkstyle.override`.
5. Under the **Source Code Management** section, check **Subversion**, add your subversion repository URL to **Repository URL**, and the value `./OVERRIDE` to **Local module directory** `(optional)`.
6. Click on the **Add more locations...** button.
7. For **new Repository URL**, add `https://source.sakaiproject.org/svn/profile2/tags/profile2-1.4.5`.
8. In the **Pre-Steps** section, for the **Add pre-build** step, select the option **Execute shell**.

9. In the command text area, add `cp OVERRIDE/pom_checkstyle.xml`.
10. Under the **Build** section, add the following details:
    - **Root POM**: `pom_checkstyle.xml`
    - **Goals and options**: `clean checkstyle:checkstyle`
11. Under the **Build Settings** section, check **Publish Checkstyle analysis results**.
12. Click on **Save**.
13. Run the Job a number of times, and review the output.

How it works...

The **profile2 tool** is used by many millions of users around the world within the **Sakai Learning Management System** (`http://sakaiproject.org`). It's a realistic piece of industrial-quality coding. It is a social hub for managing what others can see of your account details. The project divides the code between implementation, API, and model.

In this recipe, you had created a replacement `pom.xml` file. You did not need to copy any of the dependencies from the original `pom.xml`, as checkstyle does not need compiled code to do its calculations. However, the location of the modules was needed for it to know where to look for the code.

The injected `pom.xml` file is pulled into its own repository in the Jenkins workspace by configuring the **Local module directory** `(optional)` option. This avoids the injected code being overwritten when Jenkins pulls in the profile2 source code. The job then copies the injected `pom.xml` file to the main workspace directory so that it can find the modules in the correct relative location.

Checkstyle was not configured in detail in `pom.xml`, because we were only interested in the overall trend. However, if you want to zoom in to the details, checkstyle can be configured to generate results based on specific metric, such as the complexity of Boolean expressions or the **Non Commenting Source Statements** (**NCSS**)—`http://checkstyle.sourceforge.net/config_metrics.html`.

There's more...

You can view the statistics from most quality-measuring tools remotely by using the Jenkins XML API. The syntax for checkstyle, PMD, FindBugs, and so on is `Jenkins_HOST/job/[Job-Name]/[Build-Number]/[Plugin-URL]Result/api/xml`.

For example, a URL similar to the following will work in the case of this recipe:

```
localhost:8080/job/ch5.quality.checkstyle.override/11/
checkstyleResult/api/xml
```

The returned results for this recipe look similar to the following:

```
<checkStyleReporterResult>
  <newSuccessfulHighScore>true</newSuccessfulHighScore>
  <warningsDelta>38234</warningsDelta>
  <zeroWarningsHighScore>1026944</zeroWarningsHighScore>
  <zeroWarningsSinceBuild>0</zeroWarningsSinceBuild>
  <zeroWarningsSinceDate>0</zeroWarningsSinceDate>
</checkStyleReporterResult>
```

To obtain the data remotely, you will need to authenticate. For information on how to perform remote authentication, review the *Remotely triggering Jobs* recipe, *Chapter 3*, *Building Software*.

See also

- *Faking checkstyle results*

Faking checkstyle results

This recipe details how you can forge checkstyle reports. This will allow you to hook in your custom data to the **checkstyle Jenkins plugin** (`https://wiki.jenkins-ci.org/display/JENKINS/Checkstyle+Plugin`), exposing your custom test results without writing a new Jenkins plugin. The benefit of this method, when compared to custom plotting, is the more logical test results location in Jenkins. A further benefit is that you can also aggregate the fake results with other metrics summaries using the **Analysis Collector Plugin** (`https://wiki.jenkins-ci.org/display/JENKINS/Analysis+Collector+Plugin`).

Getting ready

If you have not already done so, install checkstyle and create a new directory in your subversion repository for the code.

How to do it...

1. Create a Perl script file named `generate_data.pl` with the following content:

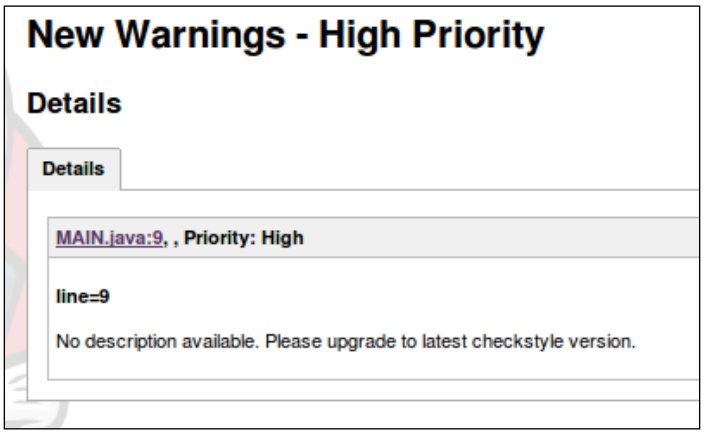
```
#!/usr/bin/perl
$rand=int(rand(9)+1);

print <<MYXML;
<?xml version="1.0" encoding="UTF-8"?>
<checkstyle version="5.4">
  <file name="src/main/java/MAIN.java">
    <error line="$rand" column="1" severity="error"
      message="line=$rand" source="MyCheck"/>
  </file>
</checkstyle>
MYXML
#Need this extra line for the print statement to work
```

2. Make the directory `src/main/java`.
3. Add the Java file `src/main/java/MAIN.java` with the following content:

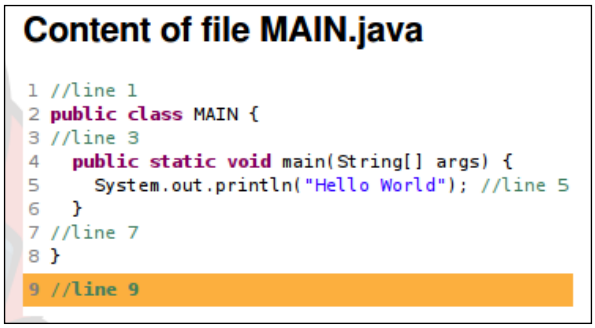
```
//line 1
public class MAIN {
  //line 3
  public static void main(String[] args) {
    System.out.println("Hello World"); //line 5
  }
//line 7
}
//line 9
```

4. Commit the files to your subversion repository.
5. Create a Jenkins free-style Job `ch5.quality.checkstyle.generation`.
6. Within the **Source Code Management** section, check **Subversion** adding `Your repo URL` to the **Repository URL**.
7. Within the **Build** section, select **build Step**, **Execute Shell**. In the **Command** input, add the command `perl generate_data.pl > my-results.xml`.
8. In the **Post-build Actions** section, check **Publish Checkstyle analysis results**. In the **Checkstyle results** text input, add `my-results.xml`.
9. Click on **Save**.

10. Run the Job a number of times, and review the results and trend.



The details report is linked to the correct line number in an HTML version of the source code.



How it works...

The plugins used in this chapter store their information in XML files. We chose the simplest XML structure to fake the results. This happened to be from the checkstyle tool.

The Perl code creates a simple XML results file, which chooses a line between 1 and 9, which then fails. The format outputted is similar to the following:

```
<checkstyle version="5.4">
<file name="src/main/java/MAIN.java">
  <error line="9" column="1" severity="error" message="line=9"
    source="MyCheck"/>
</file>
```

The file location is relative to the Jenkins workspace. The Jenkins plugin opens the file found at this location so that it can display it as source code.

For each error found, an `<error>` tag is created. The plugin maps the severity level error to high.

There's more...

You may not have to force your results into a fake format. First, consider the **Xunit plugin** (`https://wiki.jenkins-ci.org/display/JENKINS/xUnit+Plugin`). It is a utility plugin that supports the conversion of the results from different regression test frameworks. The plugin translates the different result types into a standardized JUnit format. You can find the JUnit results schema at `http://windyroad.org/dl/Open%20Source/JUnit.xsd`.

See also

- *Plotting alternative code metrics in Jenkins*, *Chapter 2*, *Building Software*
- *Checking style using an external pom.xml*

Integrating Jenkins with Sonar

Sonar is a rapidly evolving application for reporting quality metrics and finding code hotspots. This recipe details how, through a Jenkins plugin, to generate code metrics and push them directly to a Sonar database.

Getting ready

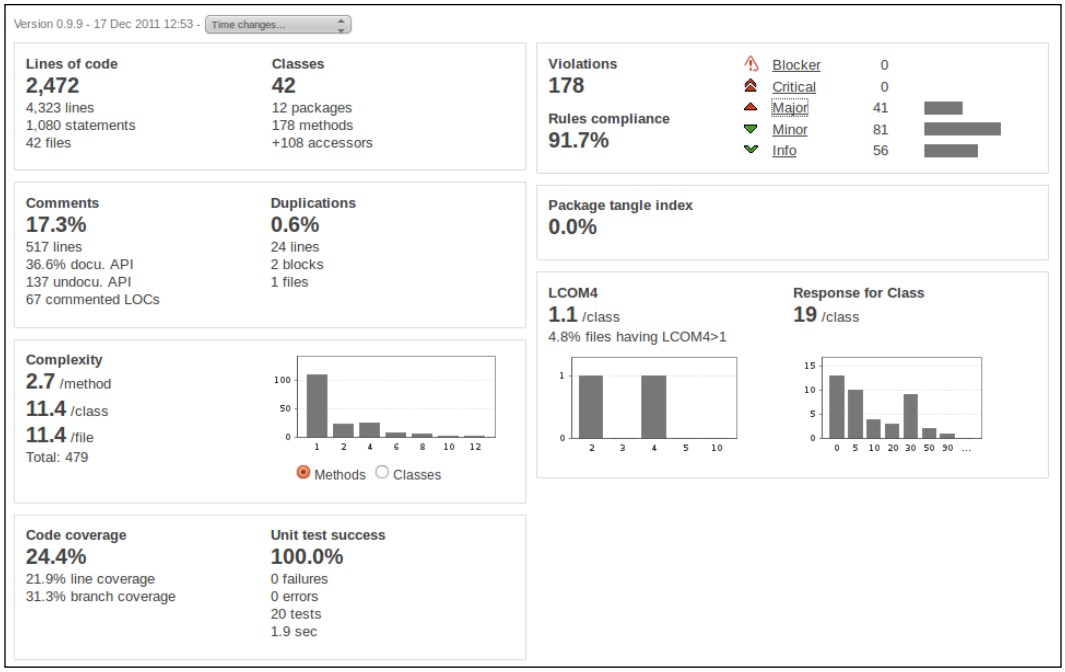
Install the **Sonar plugin** (`http://docs.codehaus.org/display/SONAR/Hudson+and+Jenkins+Plugin`).

Download and unpack Sonar. You can run directly from within the `bin` directory, by selecting the OS directory underneath. For example, the Desktop Ubuntu startup script is `bin/linux-x86-32/sonar.sh`. You now have an insecure default instance running on port `9000`. For fuller installation instructions, review `http://docs.codehaus.org/display/SONAR/Install+Sonar`.

How to do it...

1. Within the main Jenkins configuration (`/configure`), in the **Sonar** section, add `localhost` for **Name**.
2. Click on **Save**.
3. Create a Maven 2/3 Job named `ch5.quality.sonar`.

4. Under the **Source Code Management** section for the **Repository URL**, add `https://source.sakaiproject.org/contrib/programmerscafe/blogwow/tags/0.9.9`.
5. Within the **Build Triggers** section, verify that no build triggers are selected.
6. Under the **Build** section for **Goals and options**, add `clean install`.
7. For the **Post-build Actions** section, check **Sonar**.
8. Click on **Save**.
9. Run the Job.
10. Click on the **Sonar** link, and review the newly generated report.



How it works...

The source code is that of a blogging tool used in the programmer's café by the Sakai Foundation. The **blogging project** is a multi-module project with some relatively complex details. The **programmer's cafés** are events where new Sakai programmers get a chance to learn programming Sakai tools (`https://confluence.sakaiproject.org/display/BOOT/Programmer%27s+Cafe`).

The default Sonar instance is pre-configured with an in-memory database. The Jenkins plugin already knows the default configuration and requires little extra configuration. The Jenkins Sonar plugin does not need you to reconfigure your `pom.xml`. The Jenkins plugin handles all the details itself for generating results.

The Job first ran Maven to clean out the old compiled code from the workspace, and then ran the install goal, which compiles the code as part of one of its phases.

The Jenkins Sonar plugin then makes direct contact with the Sonar database and adds the previously generated results. You can now see the results in the Sonar application.

There's more...

Sonar is a dedicated application for measuring software quality metrics. Like Jenkins, it has a dedicated and active community. You can expect an aggressive roadmap of improvements. Features, such as its ability to point out hotspots of suspicious code, a visually appealing report dashboard, ease of configuration, and detailed control of inspection rules to view, currently differentiate it from Jenkins.

Sonar plugins

It is easy to expand the features of Sonar by adding extra plugins. You can find the official set mentioned at the following URL:

`http://docs.codehaus.org/display/SONAR/Sonar+Plugin+Library`

The plugins include a number of equivalent features to the ones you can find in Jenkins. Where Sonar is noticeably different are its governance plugins. This is where code coverage is used to defend the quality of a project.

Aggregating results using the Violations plugin

The Jenkins **Violations plugin** accepts the results from a range of quality metrics tools and combines them into a unified report. This plugin is the nearest equivalent to Sonar within Jenkins. Before deciding if you need an extra application in your infrastructure, it is worth reviewing to see if it fulfills your quality metrics needs.

See also

- *Looking for "smelly" code through code coverage*
- *Activating more PMD rulesets*
- *Interpreting JavaNCSS*

6
Testing Remotely

In this chapter, we will cover the following recipes:

- Deploying a WAR file from Jenkins to Tomcat
- Creating multiple Jenkins nodes
- Testing with Fitnesse
- Activating Fitnesse HtmlUnit Fixtures
- Running Selenium IDE tests
- Triggering failsafe integration tests with Selenium Webdriver
- Creating JMeter test plans
- Reporting JMeter performance metrics
- Functional testing using JMeter assertions
- Enabling Sakai web services
- Writing test plans with SoapUI
- Reporting SoapUI test results

Introduction

By the end of this chapter, you will have ran performance and functional tests against web applications 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.

Remote testing through Jenkins considerably increases the number of dependencies in your infrastructure, and thus the maintenance effort. Remote testing is problem domain-specific, decreasing the size of the audience that can write tests.

This chapter emphasizes the need to make test writing accessible to a large audience. Embracing the largest possible audience improves the chances that the tests defend the intent of the application.

The technologies highlighted include:

- **Fitnesse**: It is a server that runs different types of tests. The tests are written in a wiki format. Having a wiki-like language to express and change tests on the fly gives functional administrators, consultants, and the end users a place to express their needs. You will be shown how to run Fitnesse tests through Jenkins. Fitnesse is also a framework where you can extend Java interfaces to create new testing types. The testing types are called **fixtures**. There are a number of fixtures available, including ones for database testing, running tools from the command line, and functional testing of web applications.
- **JMeter**: It is a popular open source tool for stress testing. It can also be used to functionally test through the use of assertions. JMeter has a GUI that allows you to build test plans. The test plans are then stored in an XML format. JMeter is runnable through Maven or Ant scripts. JMeter is very efficient, and one instance is normally enough to hit your infrastructure hard. However, for super high load scenarios, JMeter can trigger an array of JMeter instances.
- **Selenium**: It is the de facto industrial standard for functional testing of web applications. With Selenium IDE, you can record your actions within Firefox, saving them in an HTML format to replay later. The tests can be re-run through Maven using Selenium **Remote Control** (**RC**). It is common to use Jenkins slaves with different OSs and browser types to run the tests. The alternative is to use **Selenium Grid** (`http://selenium-grid.seleniumhq.org/`).
- **Selenium Webdriver and TestNG unit tests**: A programmer-specific approach to functional testing is to write unit tests using the TestNG framework. The unit tests apply the Selenium Webdriver framework. Selenium RC is a proxy that controls the web browser. In contrast, the Webdriver framework uses native API calls to control the web browser. You can even run the **HtmlUnit** framework, removing the dependency of a real web browser. This enables OS-independent testing, but removes the ability to test for browser-specific dependencies. Webdriver supports many different browser types.
- **SoapUI**: It simplifies the creation of functional tests for web services. The tool can read **Web Service Definition Language** (**WSDL**) files publicized by web services, using the information to generate the skeleton for functional tests. The GUI makes it easy to understand the process.

Deploying a WAR file from Jenkins to Tomcat

The three main approaches to deploying web applications for integration tests are as follows:

- Run the web app locally in a container such as Jetty, brought to life during a Jenkins Job. The applications database is normally in-memory, and the data stored is not persisted past the end of the Job. This saves cleaning up and eliminates unnecessary dependency on the infrastructure.
- A **nightly build** is created where the application is rebuilt regularly through a scheduler. This normally happens at night when no one is using the infrastructure, hence the name. No polling of the SCM is needed. The advantage of this approach are a distributed team that knows exactly when and at which fixed web address a new build exists. This information simplifies writing deployment scripts.
- Deploy to an application server. First, package the web application in Jenkins, and then deploy it to an application server. It is now ready for testing by a second Jenkins Job. The disadvantage of this approach is that you are replacing an application on the fly, and the host server might not always respond stably.

In this recipe, you will be using the **Deploy** plugin to deploy a `war` file to a remote Tomcat 7 server. This plugin can deploy across a range of server types and version ranges including Tomcat, GlassFish, and JBoss.

Getting ready

Install the deploy plugin for Jenkins Deploy plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Deploy+Plugin`). Download the newest version of Tomcat 7 and unpack it (`http://tomcat.apache.org/download-70.cgi`).

How to do it...

1. Create a Maven project for a simple WAR file using the following command line:

```
mvn archetype:generate -DgroupId=nl.berg.packt.simplewar
-DartifactId=simplewar -DarchetypeArtifactId=maven-archetype-
webapp  -Dversion=1.0-SNAPSHOT
```

2. Commit the newly created project to your subversion repository.
3. Under the Tomcat root directory, edit `conf/server.xml`, changing the default connector port number to `38887`.

```
<Connector port="38887" protocol="HTTP/1.1"
  connectionTimeout="20000"
  redirectPort="8443" />
```

4. From the command line, start Tomcat.

   ```
   bin/startup.sh
   ```

5. Log in to Jenkins.
6. Create a Maven 2/3 project named `ch6.remote.deploy`.
7. Under the **Source Code Management** section, check the **subversion** radio box, adding your own subversion repository URL to **Repository URL**.
8. In the **Build** section, add `clean package` for **Goals and options**.
9. In the **Post-build Actions** section, check **Deploy war/ear to a container**, adding the following configuration:
   - **WAR/EAR files**: `target/simplewar.war`
   - **Container**: `Tomcat 7.x`
   - **Manager user name**: `jenkins_build`
   - **Manager password**: `mylongpassword`
   - **Tomcat URL**: `http://localhost:38887`
10. Click on **Save**.
11. Run the build.
12. The build will fail with an output similar to the following:

    ```
    java.io.IOException: Server returned HTTP response code: 401 for
    URL: http://localhost:38887/manager/text/list
    ```

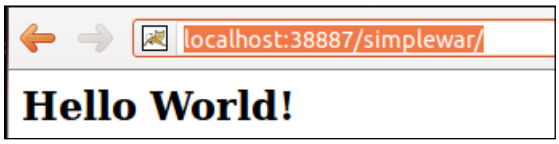
13. Edit `conf/tomcat-users.xml`, and add the following code before `</tomcat-users>`:

    ```
    <role rolename="manager-gui"/>
    <role rolename="manager-script"/>
    <role rolename="manager-jmx"/>
    <role rolename="manager-status"/>
    <user username="jenkins_build" password="mylongpassword"
      roles="manager-gui,manager-script,manager-jmx,manager-status"/>
    ```

14. Restart Tomcat.
15. In Jenkins, build the Job again. The build will now succeed. Reviewing the Tomcat log, `logs/catalina.out`, will reveal an output similar to the following:

    ```
    Jan 06, 2012 1:31:39 PM org.apache.catalina.startup.HostConfig
    deployWAR
    INFO: Deploying web application archive /xxxxx/apache-
    tomcat-7.0.23/webapps/simplewar.war
    ```

16. With a web browser, visit `http://localhost:38887/simplewar/`.



How it works...

At the time of writing, the deploy plugin deploys to the following server types and versions:

- Tomcat 4.x/5.x/6.x/7.x
- JBoss 3.x/4.x
- GlassFish 2.x/3.x

In this recipe, Jenkins packages a simple WAR file and deploys to a Tomcat 7 instance. By default, Tomcat listens on port `8080`, as does Jenkins. By editing `conf/server.xml`, the port was moved to `38887`, avoiding conflict.

The Jenkins plugin calls the Tomcat Manager. After failing to deploy with a `401 not authorized` error, (`http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`), you created a Tomcat user in with the required roles. In fact, the new user has more powers than is needed for deployment. The user has the power to review the JMX data for monitoring. This helps you with debugging later.

When deploying in production, use an SSL connection to avoid sending passwords unencrypted over the wire.

There's more...

On startup, the Tomcat logs mention that the Apache Tomcat Native library is missing.

```
INFO: The APR based Apache Tomcat Native library which allows optimal
performance in production environments was not found on the java.library.
path: /usr/java/packages/lib/i386:/usr/lib/i386-linux-gnu/jni:/lib/i386-
linux-gnu:/usr/lib/i386-linux-gnu:/usr/lib/jni:/lib:/usr/lib
```

The library improves the performance, and it is based on **Apache Portable Runtime Projects** effort (`http://apr.apache.org/`).

You can find the source code in `bin/tomcat-native.tar.gz`. The build instructions can be found at `http://tomcat.apache.org/native-doc/`.

See also

- *Configuring Jetty for integration tests*, *Chapter 3*, *Building Software*

Creating multiple Jenkins nodes

Testing is a heavy weight process. If you want to scale your services, then you will need to plan to offset most of the work to other nodes.

One evolutionary path for Jenkins in an organization is to start off with one Jenkins master. As the number of Jobs increases, we need to push off the heavier Jobs, such as testing, to slaves. This leaves the master the lighter and more specialized work of aggregating the results.

This recipe uses the **Multi slave config plugin** (`https://wiki.jenkins-ci.org/display/JENKINS/Multi+slave+config+plugin`) to install an extra Jenkins node locally. It is Ubuntu-specific, allowing Jenkins to install, configure, and command the slave through SSH.

Getting ready

In Jenkins, install the multi slave config plugin. You will also need to have a test instance of Ubuntu as described in the recipe *Using a sacrificial Jenkins instance*, *Chapter 1*, *Maintaining Jenkins*.

How to do it...

1. From the command line of the sacrificial Jenkins instance, create the user `jenkins-unix-nodex`.

```
sudo adduser jenkins-unix-nodex
```

2. Generate a private key and a public certificate for Jenkins with an empty passphrase:

```
sudo -u jenkins ssh-keygen -t rsa


Generating public/private rsa key pair.
Enter file in which to save the key (/var/lib/jenkins/.ssh/id_
rsa):
Created directory '/var/lib/jenkins/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /var/lib/jenkins/.ssh/id_
rsa.
Your public key has been saved in /var/lib/jenkins/.ssh/id_rsa.pub
```

3. Create the `.ssh` directory, copying the Jenkins public certificate to `.ssh/authorized_keys`.

```
sudo -u jenkins-unix-nodex mkdir /home/jenkins-unix-nodex/.ssh
sudo cp  /var/lib/jenkins/.ssh/id_rsa.pub /home/jenkins-unix-nodex/.ssh/authorized_keys
```

4. Change the ownership and group `of authorized_keys` to `Jenkins-unix-nodex:jenkins-unix-nodex`:

```
sudo chown jenkins-unix-nodex:jenkins-unix-nodex
.ssh/authorized_keys
```

5. Test that you can log in without a password as `jenkins` to `Jenkins-unix-nodex.jenkins-unix-nodex`.

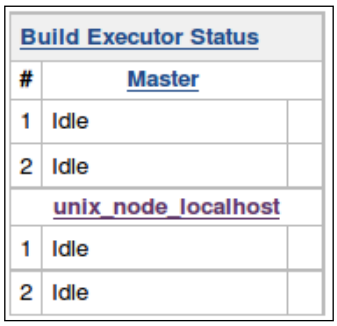


You will need to accept the host's certificate.

```
sudo su jenkins
ssh jenkins-unix-nodex@localhost
The authenticity of host 'localhost (127.0.0.1)' can't be established.
ECDSA key fingerprint is xx:yy:21:zz:46:dd:02:fa:1w:15:27:20:e6:74:3e:a2.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost' (ECDSA) to the list of known hosts.
```

6. Log in through the Jenkins web interface.
7. Visit the **MultiSlave Config Plugin** under **Manage Jenkins** (`localhost:8080/multi-slave-config-plugin/?`).
8. Click on **Add Slaves**.
9. Add `unix-node01` to **Create slaves by names separated with space**.
10. Click on **Proceed**.
11. In the **Multi Slave Config Plugin – Add slaves** screen, add the following details:
    - **Description**: `I am a dumb Ubuntu node`
    - **# of executors**: `2`
    - **Remote FS root**: `/home/Jenkins-unix-nodex/home/jenkins-unix-nodex`
    - **Set labels**: `unix dumb functional`

12. Select for **Launch** method **Launch slave agents on Unix machines via SSH**, adding the following details:
    - **Host**: `localhost`
    - **Username**: `Jenkins-unix-nodex`
    - **Private key File**: `/var/lib/Jenkins/.ssh/id_rsa`
13. Click on **Save**.
14. Return to the main page. You will now see **Build Executor Status** include the **Master** and **unix-node1**.



How it works...

In this recipe, you have deployed one node locally to a `*NIX` box. A second user account is used. The account is provisioned with the public key of the Jenkins user for easier administration. Jenkins can now use `ssh` and `scp` without a password.

The Multi slave config plugin takes the drudgery out of deploying slave nodes. It allows you to copy from one template slave and deploys a number of nodes.

Jenkins can run nodes in a number of different ways. Using SSH, the master runs a custom script or through Windows services (`https://wiki.jenkins-ci.org/display/JENKINS/Distributed+builds`). The most reliable approach is through the SSH protocol. The strength of this approach is multifold.

- The use of SSH is popular, implying a small learning curve for a large audience.
- SSH is a reliable technology that has been battle-hardened over many generations.
- There are SSH daemons for most Operating Systems and not just for `*NIX`. One alternative is to install **Cygwin** (`http://www.cygwin.com/`) with SSH daemon on Windows.



If you want to have your UNIX scripts running in Windows under Cygwin, consider installing the Cygpath plugin. The plugin converts UNIX style paths to Windows style. For more information, visit:

`https://wiki.jenkins-ci.org/display/JENKINS/Cygpath+Plugin`

The configured node has three labels assigned: `unix`, `dumb`, and `functional`. When creating a new Job, checking the setting **Restrict where this project can be run** and adding one of the labels will ensure that the Job is run on a node with that label.

The Master calculates which node to run a Job based on a priority list. Unless otherwise configured, Jobs created when there was only a master will still run on the master. Newer Jobs will run by default on the slaves.

Consistency breeds reliability: When deploying more than one Jenkins node, it saves effort if you are consistent with the structure of their environments. Consider using a virtual environment starting from the same basic set of images. **CloudBees** (`http://www.cloudbees.com`) is one example of a commercial service centered on deployment of virtual instances.

There's more...

Since version 1.446 (`http://jenkins-ci.org/changelog`), Jenkins has a built-in SSH daemon. This will decrease the amount of effort in writing the client-side code. The command-line interface is accessible through the SSH protocol. You can set the port number of the daemon through the Jenkins management web page, or leave the port number to float.

Jenkins publishes the port number using header information for `X-SSH-Endpoint`. To see for yourself, telnet into Jenkins and Get the login page. Jenkins returns the port numbers with other header information. For example, for `*NIX` systems from the command line, try the following:

```
telnet localhost 8080
GET /login
```

Jenkins' response will be similar to the following:

```
HTTP/0.9 200 OK
Server: Winstone Servlet Engine v0.9.10
Expires: 0
X-Hudson-Theme: default
Content-Type: text/html;charset=UTF-8
X-Hudson: 1.395
X-Jenkins: 1.447
```

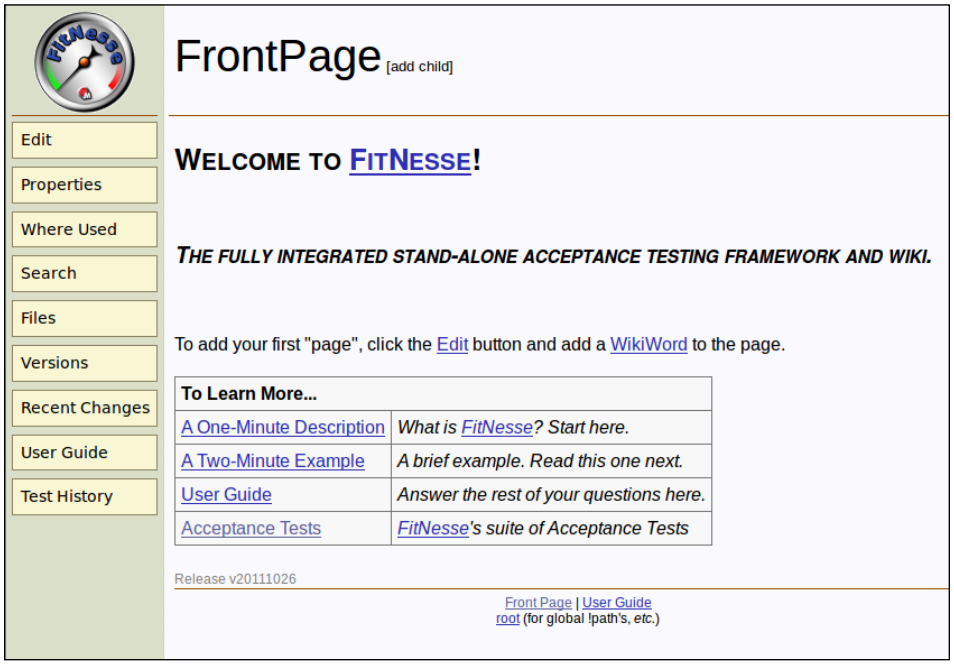
```
X-Hudson-CLI-Port: 51485
X-Jenkins-CLI-Port: 51485
X-Instance-Identity: MIIBIjANBgkqhkiG9w ….
X-SSH-Endpoint: localhost:48781
```

See also

- *Using a sacrificial Jenkins instance*, *Chapter 1*, *Maintaining Jenkins*

Testing with Fitnesse

Fitnesse (`http://fitnesse.org`) is a fully-integrated standalone wiki, and acceptance-testing framework. You can write tests in tables and run them. Writing tests in a wiki language widens the audience of potential test writers and decreases the initial efforts in learning a new framework.



If a test passes, the table row is displayed in green. If it fails, it is displayed in red. The tests can be surrounded by wiki content delivering context information, such as user stories, at the same location as the tests. You can also consider creating mock-ups of your web applications in Fitnesse next to the tests, and point the tests at those mock-ups.

This recipe describes how to run Fitnesse remotely and displays the results within Jenkins.

Getting ready

Download the latest stable Fitnesse JAR from `http://fitnesse.org/FrontPage.FitNesseDevelopment.DownLoad`. Install the Fitnesse plugins for Jenkins from `https://wiki.jenkins-ci.org/display/JENKINS/Fitnesse+Plugin`.



The release number used to test this recipe was `20111026` (`http://fitnesse.org/.FrontPage.FitNesseDevelopment.FitNesseRelease20111026`).

How to do it...

1. Create the directories `fit/logs`, and place in the fit directory `fitnesse.jar`.
2. Run the Fitnesse `help` from the command line, and review the options.

```
java -jar fitnesse.jar –help

Usage: java -jar fitnesse.jar [-pdrleoa]
    -p <port number> {80}
    -d <working directory> {.}
    -r <page root directory> {FitNesseRoot}
    -l <log directory> {no logging}
    -e <days> {14} Number of days before page versions expire
    -o omit updates
    -a {user:pwd | user-file-name} enable authentication.
    -i Install only, then quit.
    -c <command> execute single command.
```

3. Run Fitnesse from the command line, and review the startup output.

```
java -jar fitnesse.jar -p 39996 -l logs -a tester:test

FitNesse (v20111026) Started...
    port:             39996
    root page:        fitnesse.wiki.FileSystemPage at ./
FitNesseRoot
    logger:           /xxxxx/fit/logs
    authenticator:    fitnesse.authentication.OneUserAuthenticator
    html page factory: fitnesse.html.HtmlPageFactory
    page version expiration set to 14 days.
```

4. Using a web browser, visit `http://localhost:39996`.
5. Click on the **Acceptance Test** link.
6. Click on the **Suite** link. This will activate a set of tests. Depending on your computer, the tests may take a few minutes to complete. The direct link is `http://localhost:39996/FitNesse.SuiteAcceptanceTests?suite`.

7. Click on the **Test History** link. You will need to log on as user tester with the password as `test`. Review the log in the `fit/logs` directory. After running the suite again, you will now see an entry similar to the following:

```
127.0.0.1 - tester [06/Jan/2012:09:44:53 +0100] "GET /FitNesse.
SuiteAcceptanceTests?suite HTTP/1.1" 200 6086667
```

8. Log in to Jenkins, and create a freestyle software project named `ch6.remote.fitnesse`.
9. In the **Build** section, select the **Execute fitnesse tests** option from the **Add Build** step.
10. Check the option **Fitnesse instance is already running**, and add the following details:
    - **Fitnesses Host**: `localhost`
    - **Fitnesses Port**: `39996`
    - **Target Page**: `FitNesse.SuiteAcceptanceTests`
    - Check the **Is target a suite?** option
    - **HTTP Timeout (ms)**: `180000`
    - **Path to fitnesse xml results file**: `fitnesse-results.xml`
11. In the **Post-build Actions** section, check the **Publish Fitnesse results report** option.
12. Add the value `fitnesse-results.xml` to the input **Path to fitnesse xml results file**.
13. Click on **Save**.
14. Run the Job.
15. Review the latest job by clicking on the link **FitNesse Results**.



How it works...

Fitnesse has a built-in set of acceptance tests, which it uses to check itself for regressions. The Jenkins plugin calls the test and asks for the results to be returned in an XML format using an **HTTP GET** request with the following URL: `http://localhost:39996/FitNesse.SuiteAcceptanceTests?suite&format=xml`. The results look similar to the following:

```
<testResults>
<FitNesseVersion>v20111026</FitNesseVersion>
<rootPath>SuiteAcceptanceTests</rootPath>
<result>
  <counts>
    <right>103</right>
    <wrong>0</wrong>
    <ignores>0</ignores>
    <exceptions>0</exceptions>
  </counts>
  <runTimeInMillis>94</runTimeInMillis>
  <relativePageName>CopyAndAppendLastRow</relativePageName>
  <pageHistoryLink>
    FitNesse.SuiteAcceptanceTests.SuiteFitDecoratorTests
      .CopyAndAppendLastRow?pageHistory&resultDate=20120106102754
      &format=xml
  </pageHistoryLink>
</result>
```

The Jenkins plugin then parses the XML and generates a report.

By default, there is no security enabled on Fitnesse pages. In this recipe, a username and password were defined during startup. However, we did not take this further, and defined the security permissions on the page. To activate, you will need to go to the **properties** link on the left-hand side of a page, and check the security permission for **secure-test**.

You can also authenticate through a list of users in a text file or **Kerberos/Active Directory**. For more details, review `http://fitnesse.org/FitNesse.UserGuide.SecurityDescription`.

There is also a contributed plugin for LDAP authentication: `https://github.com/timander/fitnesse-ldap-authenticator`



Consider applying security in depth: Adding IP restrictions through a firewall on the Fitnesse server creates an extra layer of defense. For example, you can place an Apache server in front of the wiki, and enabling SSL/TLS ensures encrypted passwords. A thinner alternative to Apache is **nginx**: `http://wiki.nginx.org`.

There's more...

Fitnesse is not the only open source storyteller. An alternative for stories is **xPlanner** (`http://xplanner.codehaus.org/`, `http://www.projectmagazine.com/reviews/76-software/98-thinking-in-extremes-with-xplanner`), which is a web-based project planner based around iterating stories.

Unfortunately, at the time of writing, the last release was in 2006, so do not expect any software updates soon.

See also

- *Activating Fitnesse HtmlUnit Fixtures*

Activating Fitnesse HtmlUnit Fixtures

Fitnesse is an extendable testing framework. It is possible to write your own testing types known as **fixtures**, and call the new test types through Fitnesse tables. This allows Jenkins to run alternative tests than the ones available.

This recipe shows you how to integrate Functional tests using an **HtmlUnit** fixture. The same approach can be used for other fixtures as well.

Getting ready

This recipe assumes that you have already performed testing with the Fitnesse recipe.

How to do it...

1. Visit `http://chrispederick.com`, and download and unpack `HtmlFixture-2.5.1`.
2. Move the `HtmlFixture-2.5.1/lib` directory to the `FitNesseRoot` directory.
3. Copy `HtmlFixture-2.5.1/log4j.properties` to `FitNesseRoot/log4j.properties`.
4. Start Fitnesse.

   ```
   java -jar fitnesse.jar -p 39996 -l logs -a tester:test
   ```

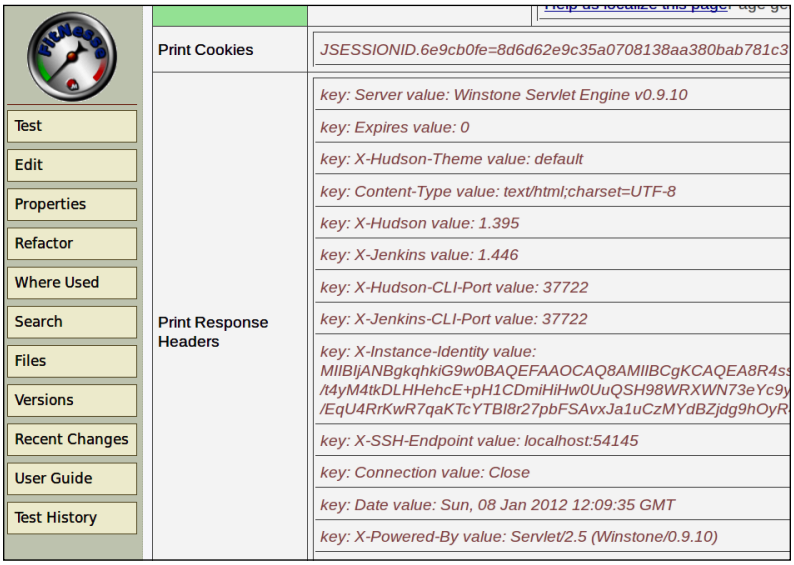
5. In a web browser, visit `http://localhost:39996/root?edit`, and add the following content, replacing `FitHome` with the fully qualified path to the home of your `fitnesse serverome`:

   ```
   !path /FitHome/FitNesseRoot/lib/*

   !fixture com.jbergin.HtmlFixture
   ```

6. Visit `http://localhost:39996`. In the left-hand menu, click on **Edit**.
7. At the bottom of the page, add the text `ThisIsMyPageTest`.
8. Click on **Save**.
9. Click on the new **ThisIsMyPageTest** link.
10. Click on the **Edit** button on the left-hand menu.
11. Add the following content after the line starting with `!contents`:

```
|Import|
|com.jbergin|

'''STORY'''
This is an example of using HtmlUnit:
http://htmlunit.sourceforge.net/

'''TESTS'''

!|HtmlFixture|
|http://localhost:8080/login| Login||
|Print Cookies||
|Print Response Headers||
|Has Text|log in|
|Element Focus|search-box|input|
|Set Value|ch5||
|Focus Parent Type|form|/search/||
```

12. Click on **Save**.
13. Click on **Test**.



14. In Jenkins, under **New Job**, copy the existing Job / copy from `ch6.remote.fitness` to the Job named `ch6.remote.fitness_fixture`.
15. In the **Build** section, under the **Target** sub-section, replace the **Target Page** text `FitNesse.SuiteAcceptanceTests` with `ThisIsMyPageTest`.
16. Uncheck **Is target a suite?**
17. Click on **Save**.
18. Run the Job. It fails because of the extra debugging information sent with the results, confusing the Jenkins plugin parser.
19. Visit the test page `http://localhost:39996/ThisIsMyPageTest?edit`, and replace the contents of the test table with the following:

```
!|HtmlFixture|
|http://localhost:8080/login| Login|
|Has Text|log in|
|Element Focus|search-box|input|
|Set Value|ch5|
|Focus Parent Type|form|/search/|
```

20. Run the Jenkins Job again; the results will now be parsed.

How it works...

Fixtures are written in Java. By placing the downloaded libraries in the Fitnesse `lib` directory, you are making them accessible. You then defined the classpath and location of the fixture in the root page, allowing the fixture to be loaded at the startup. For complete details, review the readme `HtmlFixture-2.5.1/README`.

Next, you created the link using wiki camelcase notation to the non-existent **ThisIsMyPageTest** page. An HtmlUnit fixture test was then added.

First, you needed to import the fixture whose library path was defined in the root page.

```
|Import|
|com.jbergin|
```

Next, some example descriptive wiki content was added to show that you can create a story without affecting the tests. Finally, the tests were added.

The first row of the table, `!|HtmlFixture|`, defines which fixture to use. The second row stores the location to test.

Print commands, such as `Print Cookies` or `Print Response Headers`, return information that is useful for building tests.

If you are not sure of a list of acceptable commands, then deliberately make a syntax error and the commands will be returned as results. For example:

```
|Print something||
```

The `Has Text` command is an assertion and will fail if the login is not found in the text of the returned page.

By focusing on a specific element and then `Set Value`, you can add input to a form.

During testing, if you want to display the returned content for a particular request, then you need three columns instead; for example, the first row with three columns displays the returned page, and the second row with two columns does not.

```
|http://localhost:8080/login| Login||
|http://localhost:8080/login| Login|
```

Returning HTML pages as part of the results adds extra information to the results that the Jenkins plugin needs to parse. This is prone to failure. Therefore, in *step 19*, you removed the extra columns, ensuring reliable parsing.

Full documentation for this fixture can be found at `http://htmlfixtureim.sourceforge.net/documentation.shtml`.

There's more...

Fitnesse has the potential to increase the vocabulary of remote tests that Jenkins can perform. A few interesting fixtures to review are listed as follows:

- **RestFixture for REST services**: `https://github.com/smartrics/RestFixture/wiki`
- **Webtestfixtures using Selenium for web-based functional testing**: `http://sourceforge.net/projects/webtestfixtures/`
- **DBfit**, **which allows you to test databases**: `http://gojko.net/fitnesse/dbfit/`

See also

- *Testing with Fitnesse*

Running Selenium IDE tests

Selenium IDE allows you to record your clicks within web pages and replay them in Firefox. This is good for functional testing. The test plans are saved in an HTML format.

This recipe shows you how to replay the tests automatically using Maven and then Jenkins. It uses an in-memory X server **Xvfb** (`http://en.wikipedia.org/wiki/Xvfb`) so that Firefox can be run on an otherwise headless server. Maven runs the tests using Selenium RC, which then acts as a proxy between the tests and the browser. Although we record with Firefox, you can run the tests with the other browser types as well.

It is beyond the scope of this chapter to discuss **Selenium Grid** (`http://selenium-grid.seleniumhq.org/`), other than to note that Selenium Grid allows you to run Selenium tests in parallel across a number of OSs.

Getting ready

Install the **Selenium HTML report plugin** (`https://wiki.jenkins-ci.org/display/JENKINS/seleniumhtmlreport+Plugin`) and **EnvInject plugin** (`https://wiki.jenkins-ci.org/display/JENKINS/EnvInject+Plugin`). Both Xvfb and Firefox are also required. To install Xvfb in a Debian Linux environment, run `sudo apt-get install xvfb`.

How to do it...

1. From the command line, create a simple Maven project:

```
mvn archetype:generate -DgroupId=nl.berg.packt.selenium
-DartifactId=selenium_html -DarchetypeArtifactId=maven-archetype-
quickstart -Dversion=1.0-SNAPSHOT
```

2. In the newly created `pom.xml` file, add the following `build` section just before the `</project>` tag:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>selenium-maven-plugin</artifactId>
      <version>2.1</version>
      <executions>
        <execution>
          <id>xvfb</id>
          <phase>pre-integration-test</phase>
```

```
            <goals>
              <goal>xvfb</goal>
            </goals>
          </execution>
          <execution>
            <id>start-selenium</id>
            <phase>integration-test</phase>
            <goals>
              <goal>selenese</goal>
           </goals>
           <configuration>
             <suite>
               src/test/resources/selenium/TestSuite.xhtml
             </suite>
             <browser>*firefox</browser>
             <multiWindow>true</multiWindow>
             <background>true</background>
             <results>./target/results/selenium.html</results>
             <startURL>http://localhost:8080/login/</startURL>
           </configuration>
         </execution>
       </executions>
     </plugin>
   </plugins>
</build>
```

3. Create the file `src/test/resources/log4j.properties` with the following content:

```
log4j.rootLogger=INFO, A1
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %c %x -
%m%n
```

4. Make the directory `src/test/resources/selenium`.
5. Create the file `src/test/resources/selenium/TestSuite.xhtml` with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
  lang="en">
  <head>
```

```
    <meta content="text/html; charset=UTF-8" http-equiv="content-
      type" />
    <title>My Test Suite</title>
  </head>
  <body>
    <table id="suiteTable" cellpadding="1" cellspacing="1"
      border="1" class="selenium">
      <tbody>
        <tr><td><b>Test Suite</b></td></tr>
        <tr><td><a href="MyTest.xhtml">
          Just pinging Jenkins Login Page
        </a></td></tr>
      </tbody>
    </table>
  </body>
</html>
```

The HTML will render the following output:



6. Create the test file `src/test/resources/selenium/MyTest.xhtml` with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
  lang="en">
  <head profile="http://selenium-ide.openqa.org/profiles/test-
    case">
    <meta http-equiv="Content-Type" content="text/html;
      charset=UTF-8" />
    <title>MyTest</title>
  </head>
  <body>
    <table cellpadding="1" cellspacing="1" border="1">
      <thead>
        <tr><td rowspan="1" colspan="3">MyTest</td></tr>
      </thead>
      <tbody>
```

```
        <tr><td>open</td><td>/login?from=%2F</td><td></td></tr>
        <tr>
          <td>verifyTextPresent</td>
          <td>log in</td><td></td>
        </tr>
      </tbody>

    </table>
  </body>
</html>
```

The HTML will render the following output:

MyTest	
open	/login?from=%2F
verifyTextPresent	log in

7. Run the Maven project from the command line, verifying that the build succeeds.

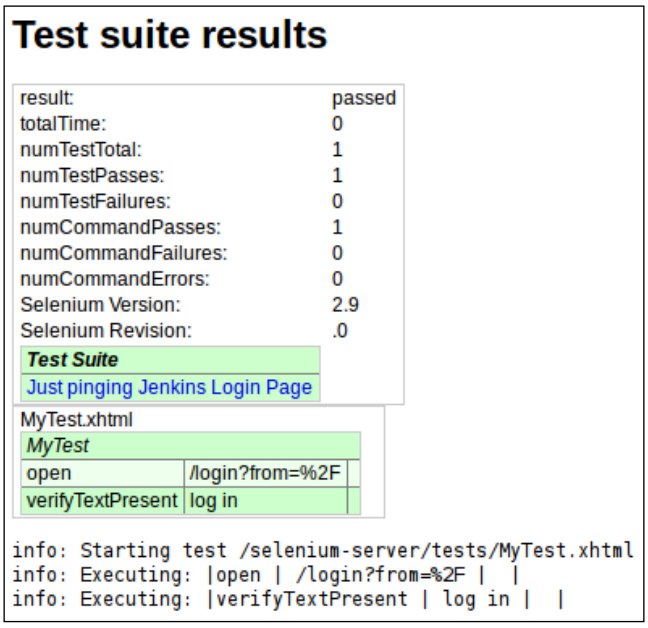
   ```
   mvn clean integration-test -Dlog4j.configuration=file./src/test/
   resources/log4j.properties
   ```

8. Run `mvn clean`, and then commit the project to your subversion repository.
9. Log in to Jenkins, and create a Maven 2/3 Job named `ch6.remote.selenium_html`.
10. In the **global** section (at the top of the configuration page), check **Prepare an environment for the job**, adding **DISPLAY=:20** for **Properties Content**.
11. In the **Source Code Management** section, check **Subversion**, adding your subversion URL to **Repository URL**.
12. In the **Build** section, on one line, add to **Goals and options**:

    ```
    clean integration-test –
      Dlog4j.configuration=file./src/test/resources/log4j.properties
    ```

13. In the **Post-build Actions** section, check **Publish Selenium HTML Report**.
14. Add the text `target/results` to the input for **Selenium test results location**.
15. Check **Set build result state to failure if an exception occurred while parsing results file.**
16. Click on **Save**.

17. Run the Job, and review the results.



How it works...

A primitive Selenium IDE test suite was created comprising two HTML pages. The first `TestSuite.xhtml` defines the suite having HTML links to the tests. We have only one test defined in `MyTest.xhtml`.

The test hits the login page for your local Jenkins and verifies that the login text is present.

`pom.xml` defines phases for bringing up and tearing down the Xvfb server. The default configuration is for Xvfb to accept input on **DISPLAY 20**:

Maven assumes that the Xvfb binary is installed and does not try to download it as a dependency. The same is true for the Firefox browser. This makes for fragile OS-specific configuration. In a complex Jenkins environment, it is this type of dependency that is the most likely to fail. There has to be a significant advantage to automatic functional testing to offset the increased maintenance effort.

The option **Multiwindow** is set to **true** as the tests run in their own Firefox window. The option **Background** is set to **true** so that Maven runs the tests in the background. The results are stored in the relative location, `./target/results/selenium.html` ready for the Jenkins plugin to parse. For more information on the selenium-maven-plugin, visit `http://mojo.codehaus.org/selenium-maven-plugin/`.

The Jenkins Job sets the **DISPLAY** variable to `20` so that Firefox renders within Xvfb. It then runs the Maven Job, generating the results page. The results are then parsed by the Jenkins plugin.

Two ways to increase the reliability of your automatic functional tests are:

- Use HtmlUnit, which does not need OS-specific configuration. However, you will then lose the ability to perform cross-browser checks.
- Run Webdriver instead of Selenium RC. Webdriver using native API calls that function more reliably. Similar to Selenium RC, Webdriver can be run against a number of different browser types.

The next recipe will showcase using unit testing with Webdriver and HtmlUnit.

There's more...

On my development Jenkins Ubuntu server, the Job running this recipe broke. The reason was that the dependencies in the Maven plugin for Selenium did not like the newer version of Firefox that was installed by an auto-update script. The resolution to the problem was to install the binary for Firefox 3.63 under the Jenkins `home` directory, and point directly at the binary in `pom.xml`, replacing:

```
<browser>*firefox</browser>
```

With:

```
<browser>*firefox Path</browser>
```

Where the `Path` is similar to `/var/lib/Jenkins/firefox/firefox-bin`.

Another cause of issues is the need to create a custom profile for Firefox that includes helper plugins to stop pop ups or the rejection of self-signed certificates. For more complete information, review `http://seleniumhq.org/docs/`.



An alternative to using Firefox as a browser is Chrome. There is a Jenkins plugin that helps provision Chrome across Jenkins nodes (`https://wiki.jenkins-ci.org/display/JENKINS/ChromeDriver+plugin`).

In the Maven `pom.xml` file, you will have to change the browser to `*chrome`.

See also

- *Triggering Failsafe integration tests with Selenium Webdriver*

Triggering Failsafe integration tests with Selenium Webdriver

Unit tests are a natural way for programmers to defend their code against regressions. Unit tests are lightweight and easy to run. Writing unit tests should be as easy as writing print statements. **JUnit** (`http://www.junit.org/`) is a popular unit test framework for Java; **TestNG** (`http://testng.org/doc/index.html`) is another.

This recipe uses Webdriver and HtmlUnit in combination with TestNG to write simple and automated functional tests. Using HtmlUnit instead of a real browser makes for stable OS agnostic tests, which, although does not test browser compatibility, can spot the majority of functional failures.

Getting ready

Create a project directory.

How to do it...

1. Create `pom.xml` with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>nl.uva.berg</groupId>
  <artifactId>integrationtest</artifactId>
  <version>1.0-SNAPSHOT</version>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.3.2</version>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-failsafe-plugin</artifactId>
        <version>2.10</version>
      </plugin>
    </plugins>
  </build>
```

```
  <dependencies>
    <dependency>
      <groupId>org.testng</groupId>
      <artifactId>testng</artifactId>
      <version>6.1.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.seleniumhq.selenium</groupId>
      <artifactId>selenium-htmlunit-driver</artifactId>
      <version>2.15.0</version>
    </dependency>
  </dependencies>
</project>
```

2. Create the directory named `src/test/nl/berg/packt/webdriver` by adding the file `TestIT.java` with the following contents:

```
package nl.berg.packt.webdriver;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.htmlunit.HtmlUnitDriver;
import org.testng.Assert;
import org.testng.annotations.*;
import java.io.File;
import java.io.IOException;

public class TestIT {
    private static final String WEBPAGE = "http://www.google.com";
    private static final String TITLE = "Google";
    private WebDriver driver;

    @BeforeSuite
    public void creatDriver(){
        this.driver= new HtmlUnitDriver(true);
    }

    @Test
    public void getLoginPageWithHTMLUNIT() throws IOException,
      InterruptedException {
        driver.get(WEBPAGE);
        System.out.println("TITLE IS
          ==>\""+driver.getTitle()+"\"");
        Assert.assertEquals(driver.getTitle(), TITLE);
```

```
    }

    @AfterSuite
    public void closeDriver(){
        driver.close();
    }
}
```

3. In the top-level project directory, run `mvn clean verify`. The build should succeed with an output similar to the following:

```
TITLE IS ==>"Google"
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
4.31 sec

Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

4. Commit the code to your subversion repository.
5. Log in to Jenkins and create a new Maven 2/3 project named `ch6.remote.driver`.
6. In the **Source Code Management** section, check **Subversion**.
7. Under **Modules/Repository URL**, add the location of your local subversion repository.
8. In the **Build** section for **Goals and options**, add `clean verify`.
9. Click on **Save**.
10. Run the Job. After a successful build, you will see a link to **Latest Test Results**, which details the functional tests.

How it works...

Maven uses the **failsafe plugin** (`http://maven.apache.org/plugins/maven-failsafe-plugin`) to run integration tests. The plugin does not fail a build if its integration-test phase contains failures. Rather, it allows the post-integration-test phase to run, allowing teardown duties to occur.

`pom.xml` has two dependencies mentioned: one for TestNG and the other for HtmlUnit driver. If you are going to use a real browser, then you will need to define their Maven dependencies.

For further details on how the failsafe plugin works with the TestNG framework, see `http://maven.apache.org/plugins/maven-failsafe-plugin/examples/testng.html`.

The Java class uses annotations to define in which part of the unit testing cycle the code will be called. `@BeforeSuite` calls the creation of the Webdriver instance at the start of the suite of tests. `@AfterSuite` closes down the driver after the tests have run. `@test` defines a method as a test.

The test visits the Google page and verifies the existence of the title. HtmlUnit notices some errors in the stylesheet and JavaScript of the returned Google page and resources; however, the assertion succeeds.

The main weakness of the example tests is the failure to separate the assertions from the navigation of web pages. Consider creating Java classes per webpage (`http://code.google.com/p/selenium/wiki/PageObjects`). Page objects return other page objects. The test assertions are then run in separate classes, comparing the members of the Page objects returned with expected values. This design pattern supports a greater degree of reusability.



An excellent framework in Groovy that supports the Page Object architecture is **Geb** (`http://www.gebish.org/`).

There's more...

80 percent of all sensory information processed by the brain is delivered through the eyes. A picture can save a thousand words of descriptive text. Webdriver has the ability to capture screenshots. For example, the following code for the Firefox driver saves a screenshot to `loginpage_firefox.png`:

```
public void getLoginPageWithFirefox() throws IOException,
InterruptedException {
  FirefoxDriver driver = new FirefoxDriver();
  driver.get("http://localhost:8080/login);
  FileUtils.copyFile(driver.getScreenshotAs(OutputType.FILE), new
    File("loginpage_firefox.png"));
  driver.close();
}
```

The most significant limitation is that screenshooting does not work with the HtmlUnit driver: `http://code.google.com/p/selenium/issues/detail?id=1361`.

See also

- *Running Selenium IDE tests*
- *Activating Fitnesse HtmlUnit fixtures*

Creating JMeter test plans

JMeter (`http://jmeter.apache.org`) is an open source tool for stress testing. It allows you to visually create a test plan, and hammer systems based on that plan.

JMeter can make many types of requests known as **samplers**. It can sample HTTP, LDAP, databases, use scripts, and much more. It can report back visually with listeners.



A beginner's book on JMeter is *Apache JMeter*, *Emily H. Halili*, Packt Publishing (`http://www.packtpub.com/beginning-apache-jmeter`).

In this recipe, you will write a test plan for hitting web pages whose URLs are defined in a textfile. In the next recipe, *Reporting JMeter test plans*, you will configure Jenkins to run JMeter test plans.

Getting ready

Download and unpack a modern version of JMeter (`http://jmeter.apache.org/download_jmeter.cgi`). JMeter is a Java application, so will run on any system that has Java correctly installed.

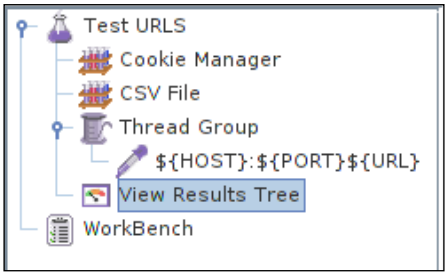
How to do it...

1. Create the subdirectory plans and example.
2. Create a CSV file `./data/URLS.csv` with the following content:

   ```
   localhost,80808,/login
   localhost,9080,/blah
   ```

3. Run the JMeter GUI; for example, `./bin/jmeter.sh` or `jmeter.bat`, depending upon the OS. The GUI will start up with a new test plan.
4. Right-click on **Test Plan**, then select **Add/Threads (Users)/Thread Group**.
5. Change the **Number of Threads (users):** to `2`.
6. Right-click on **Test Plan**, then select **Add | Config Element | CSV Data Set Config**. Add the following details:
   - **Filename**: Full path to CSV file.
   - **Variable Names (comma-delimited)**: `HOST,PORT,URL`
   - **Delimiter (use '\t'for tab)**: ,
7. Right-click on **Test Plan**, then select **Add | Config Element | HTTP cookie Manager**.

8. Right-click on **Test Plan**, then select **Add | Listener | View Tree Results**.
9. Right-click on **Thread Group**, then select **Add | Sampler | HTTP request**. Add the following details:
   - **Name**: `${HOST}:${PORT}${URL}`
   - **Server Name or IP**: `${HOST}`
   - **Port Number**: `${PORT}`
   - Under **Optional Tasks**, check **Retrieve All Embedded Resources from HTML Files**.



10. Click on **Test Plan** and **File | Save**. Save the test plan to `examples/jmeter_example.jmx`.
11. Run the test plan by pressing *CTRL+R*.
12. Click on **View Results Tree**, and explore the responses.
13. Commit this project to your subversion repository.

How it works...

JMeter uses threads to run requests in parallel. Each thread is supposed to approximately simulate one user.

The test plan uses a number of elements:

- The Thread group defines the number of thread that runs.
- The Cookie manager keeps tracks of cookies per thread. This is important if you want to keep a track through cookies between requests. For example, if a thread logs in to a Tomcat server, then the unique Jsessionid needs to be stored for each thread.
- The CSV Data Set Config element parses the content of a CSV file, putting values in the HOST, PORT, and URL variables. A new line of the CSV file is read for each thread, once per iteration. The variables are expanded in the elements by using the `${variable_name}` notation.

- The View Results Tree listener displays the results in the GUI as a tree of requests and responses. This is great for debugging, but should be removed later.

A common mistake is to assume that a thread is equivalent to a user. The main difference is that threads can respond faster than an average user. If you do not add delay factors in the request, then you can really hammer your applications with a few threads. For example, a delay of 25 seconds per click is typical for the online systems at the University of Amsterdam.



If you are looking to coax out multi-threading issues in your applications, then use a random delay element rather than a constant delay. This is also a better simulation of a typical user interaction.

There's more...

Consider storing User-Agents and other browser headers in a textfile, and then picking the values up for HTTP requests through the CSV Data Set Config element. This is useful if resources returned to your web browser, such as JavaScript or images, depend on the User-Agents. JMeter can then loop through the User-Agents, asserting that the resources exist.

See also

- *Reporting JMeter performance metrics*
- *Functional testing using JMeter assertions*

Reporting JMeter performance metrics

In this recipe, you will be shown how to configure Jenkins to run a JMeter test plan, and then collect and report the results. The passing of variables from an **Ant script** to JMeter will also be explained.

Getting ready

It is assumed that you have run through the last recipe, *Creating JMeter test plans*. You will also need to install the Jenkins performance plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Performance+Plugin`).

How to do it...

1. Open `./examples/jmeter_example.jmx` in JMeter, and save it as `./plans/URL_ping.jmx`.
2. Select **CSV Data Set Config**, changing **Filename** to `${__property(csv)}`.

3. Under the **File** menu, click on **Save**.
4. Create a `build.xml` file at the top level of your project with the following content:

```
<project default="jmeter.tests">
  <property name="jmeter" location="/var/lib/jenkins/jmeter" />
  <property name="target" location="${basedir}/target" />
  <echo message="Running... Expecting variables [jvarg,desc]" />
  <echo message="For help please read ${basedir}/README"/>
  <echo message="[DESCRIPTION] ${desc}" />

  <taskdef  name="jmeter" classname=
    "org.programmerplanet.ant.taskdefs.jmeter.JMeterTask"
    classpath="${jmeter}/extras/ant-jmeter-1.0.9.jar" />

  <target name="jmeter.init">
    <mkdir  dir="${basedir}/jmeter_results"/>
    <delete includeemptydirs="true">
      <fileset dir="${basedir}/jmeter_results" includes="**/*" />
    </delete>
  </target>

  <target name="jmeter.tests" depends="jmeter.init"
    description="launch jmeter load tests">

    <echo message="[Running] jmeter tests..." />
    <jmeter jmeterhome="${jmeter}" resultlog="${basedir}
      /jmeter_results/LoadTestResults.jtl">

      <testplans dir="${basedir}/plans" includes="*.jmx"/>
      <jvmarg value="${jvarg}" />
      <property name="csv" value="${basedir}/data/URLS.csv" />
    </jmeter>
  </target>
</project>
```

5. Commit the updates to your subversion project.
6. Log in to Jenkins.
7. Create a new free-style job with the name `ch6.remote.jmeter`.
8. Under **Source Code Management**, check **Subversion**, and add your subversion repository URL to **Repository URL**.
9. Within the **Build** section, add the build step **Invoke Ant**.

10. Press **Advanced** in the new **Invoke Ant** sub-section, adding the following for properties:

```
Jvarg=-Xmx512m
desc= This is the first iteration in a performance test
environment – Driven by Jenkins
```

11. In the **Post-build Actions** section, check **Publish Performance test result report**. Add `jmeter_results/*.jtl` to the **Report Files** input.
12. Click on **Save**.
13. Run the Job a couple of times, and review the results found under the **Performance trend** link.

How it works...

The `build.xml` file is an Ant script that sets up the environment, and then calls the JMeter Ant tasks defined in the library `/extras/ant-jmeter-1.0.9.jar`. The JAR file is installed as part of the standard JMeter distribution.

Any JMeter test plan found under the `plans` directory will be run. Moving the test plan from the `examples` directory to the `plans` directory activates it. The results are aggregated in `jmeter_results/LoadTestResults.jtl`.

The Ant script passes the `csv` variable to the JMeter test plan, with the location of the `csv` file `${basedir}/data/URLS.csv`. `${basedir}` automatically defined by Ant. As the name suggests, it is the base directory of the Ant project.

You can call JMeter functions within its elements using the structure `${__functioncall(parameters)}`. You had added the function call `${__property(csv)}` to the test plan CSV Data Set Config element. The function pulls in the value of CSV that was defined in the Ant script.

The Jenkins Job runs the Ant script, which in turn runs the JMeter test plans and aggregates the results. The Jenkins performance plugin then parses the results, creating a report.

There's more...

To build complex test plans speedily, consider using the transparent proxy (`http://jmeter.apache.org/usermanual/component_reference.html#HTTP_Proxy_Server`) built into JMeter. You can run it on a given port on your local machine, setting the proxy preferences in your web browser to match. The recorded JMeter elements will then give you a good idea of the parameters sent in the captured requests.

An alternative is **BadBoy** (`http://www.badboysoftware.biz/docs/jmeter.htm`), which has its own built-in web browser. It allows you to record your actions in a similar way to Selenium IDE, and then save to a JMeter plan.

See also

- *Creating JMeter test plans*
- *Functional testing using JMeter assertions*

Functional testing using JMeter assertions

This recipe will show you how to use JMeter assertions in combination with a Jenkins Job. JMeter can test the responses to its HTTP requests and other samplers with assertions. This allows JMeter to fail Jenkins builds based on a range of JMeter tests. This approach is especially important when starting from an HTML mockup of a web application, whose underlying code is changing rapidly.

The test plan logs in and out of your local instance of Jenkins, checking size, duration, and text found in the login response.

Getting ready

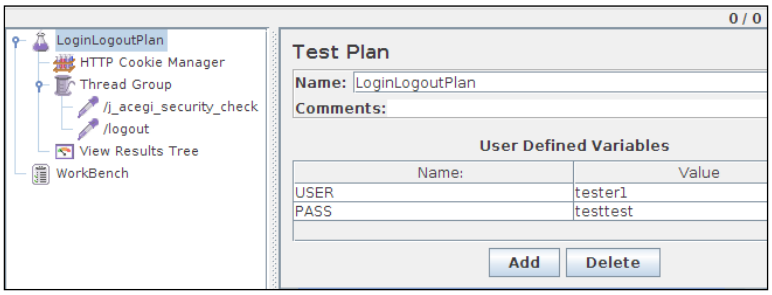
We assume that you have already performed the *Creating JMeter test plans* and *Reporting JMeter performance metrics* recipes.



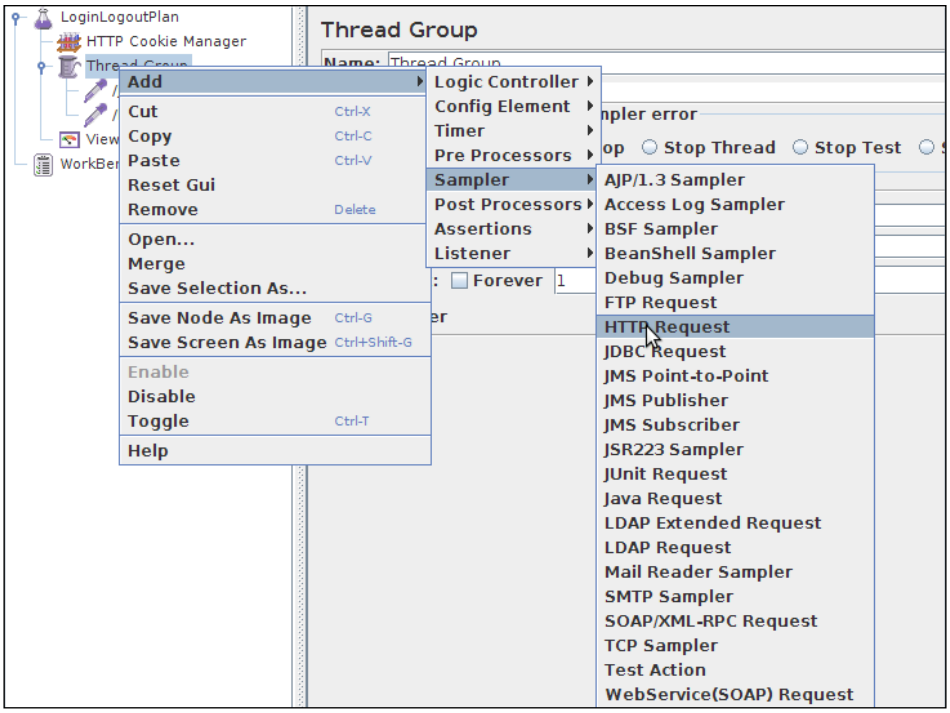
The recipe requires the creation of a user `tester1` in Jenkins. Feel free to change the username and password. Remember to delete the test user once it is no longer needed.

How to do it...

1. Create a user in Jenkins named `tester1` with password `testtest`.
2. Run JMeter. In the **Test Plan** element, change **Name** to `LoginLogoutPlan`, and add the following details for **User Defined Variables**:
    - **Name**: `USER`; **Value**: `tester1`
    - **Name**: `PASS`; **Value**: `testtest`

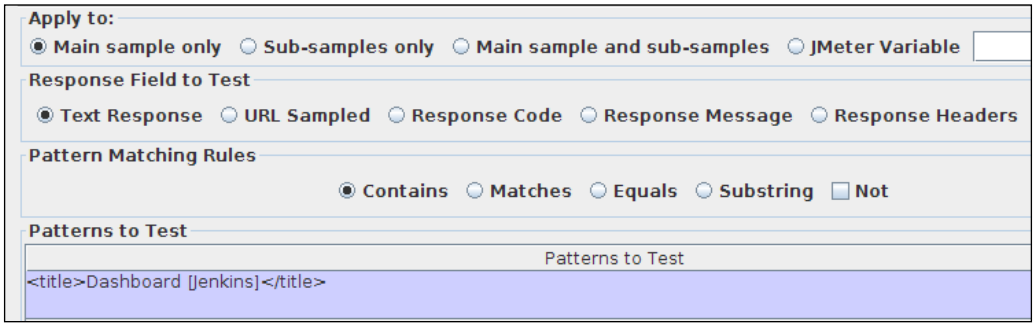


3. Right-click on **Test Plan**, then select **Add | Config Element | HTTP cookie Manager**.
4. Right-click on **Test Plan**, then select **Add | Listener | View Tree Results**.
5. Right-click on **Test Plan**, then select **Add | Threads (Users) | Thread Group**.
6. Right-click on **Thread Group**, then select **Add | Sampler | HTTP Request**.



7. Add the following details to **HTTP Request Sampler**:
   - **Name**: `/j_aceqi_security_check`
   - **Server Name or IP**: `localhost`
   - **Port Number**: `8080`
   - **Path**: `/j_aceqi_security_check`
8. Under the section **Send Parameters With the Request**, add the following details:
   - **Name**: `j_username`; **Value**: `${USER}`
   - **Name**: `j_password`; **Value**: `${PASS}`
9. Right-click on **Thread Group**, then select **Add | Sampler | HTTP Request**.
10. Add the following details to **HTTP Request Sampler**. If necessary, drag-and-drop the newly created element so that it is placed after the `/j_acegi_security_check`.

11. Add the following details to **HTTP Request Sampler**:
    - **Name**: `/logout`
    - **Server Name or IP**: `localhost`
    - **Port Number**: `8080`
    - **Path**: `/logout`
12. Save the testplan to the location `./plans/LoginLogoutPlan_without_assertions.jmx`.
13. Commit the changes to your local subversion repository.
14. In Jenkins, run the previously created Job `ch6.remote.jmeter`. Notice that at the **Performance Report** link, the `/j_acegi_security_check` HTTP request sampler succeeds.
15. Copy `./plans/LoginLogoutPlan_without_assertions.jmx` to `./plans/LoginLogoutPlan.jmx`.
16. In JMeter, edit `./plans/LoginLogoutPlan.jmx`.
17. Right-click on the JMeter element `j_acegi_security_check`, selecting **Add | Assertion | Duration Assertion**.
18. In the newly created assertion, set **Duration in milliseconds** to `1000`.
19. Right-click on the JMeter element `j_acegi_security_check`, selecting **Add | Assertion | Size Assertion**.
20. In the newly created assertion, set **Size in bytes:** to `40000`, checking **Type of Comparison** to `<`.
21. Right-click on the JMeter element `j_acegi_security_check`, selecting **Add | Assertion | Response Assertion** with the following details:
    - In the **Apply to** section, check **Main Sample only**
    - In the **Response Field to Test** section, check **Text Response**
    - In the **Pattern Matching Rules** section, check **Contains**
    - For **Patterns to Test**, add `<title>Dashboard [Jenkins]</title>`



22. Save the Test plan and commit to your local subversion repository.
23. Run from in JMeter (*Ctrl+R*), and review the **View Results Tree**. Notice that the **Size** and **Response** assertions fail.
24. In Jenkins, run the previously created Job `ch6.remote.jmeter`. Notice that within the **Performance Report** link, `/j_acegi_security_check` also fails.

How it works...

The scaffolding from the previous recipe has not changed. Any JMeter test plan found under the `plans` directory is called during the running of the Jenkins Job.

You created a new test plan with two HTTP request samplers. The first sampler posts to the login URL `/j_acegi_security_check` with the variables `j_username` and `j_password`. The response contains a cookie with a valid session ID, which is stored in the cookie manager. Three assertion elements were also added as children under the HTTP request login sampler. If any of the assertions fail, then the HTTP request result fails. In Jenkins, you can configure the Job to fail or to warn, based on definable thresholds.

The three assertions are typical for a test plan. These are:

- An assertion on the size of the result returned. The size should not be greater than 40,000 bytes.
- An assertion for duration. If the response takes too long, then you have a performance regression that you want to check further.
- The most powerful assertion is for checking for text patterns. In this case, reviewing details about the returned title. The JMeter element can also parse text against regular patterns.

There's more...

JMeter has the power to hammer away with requests. 200 threads, each firing one request per second, is roughly equivalent to 5,000 users simultaneously logged in to an application, clicking once every 25 seconds. A rough rule of thumb is that approximately 10 percent of the membership of a site is logged to an application in the busiest hour of the year. Therefore, 200 threads hitting once a second is good for a total membership of 50,000 users.

The understanding of usage patterns is also important; the less you know about how your system is going to be used, the larger a safety margin you will have to build in. It is not uncommon to plan for a 100 percent extra capacity. The extra capacity may well be the difference between you going on a holiday or not.



To expand its load creation capabilities, JMeter has the ability to run a number of JMeter slave nodes. For an official tutorial on this subject, review `http://jmeter.apache.org/usermanual/jmeter_distributed_testing_step_by_step.pdf`.

See also

- *Creating JMeter test plans*
- *Reporting JMeter performance metrics*

Enabling Sakai web services

Sakai CLE is an application used by many hundreds of universities around the world. Based on more than a million lines of Java code, Sakai CLE allows students to interact with online course and project sites. It empowers instructors to make those sites easily.

In this recipe, you will enable web services and write your own simple ping service. In the next recipe, you will write tests for these services.

Getting ready

You can find links to the newest downloads under `http://sakaiporject.org.http://sakaiproject.org`. Download and unpack Sakai CLE version 2.8.1 from `http://source.sakaiproject.org/release/2.8.1`.

How to do it...

1. Edit `sakai/sakai.properties` to include the following content:

```
webservices.allowlogin=true
webservices.allow=.*
webservices.log-denied=true
```

2. Run Sakai from the root folder `./start-sakai.sh` for *NIX systems or `./start-sakai.bat` for Windows. If Jenkins or another service is running on port `8080`, Sakai will fail with the following error:

```
2012-01-14 14:09:16,845 ERROR main org.apache.coyote.http11.
Http11BaseProtocol - Error starting endpoint
java.net.BindException: Address already in use:8080
```

3. Stop Sakai using `./stop-sakai.sh` or `./stop-sakai.bat`.

4. Modify `conf/server.xml` to move the port number to `39955`; for example:

```
<Connector port="39955" maxHttpHeaderSize="8192"
URIEncoding="UTF-8" maxThreads="150" minSpareThreads="25"
maxSpareThreads="75" enableLookups="false" redirectPort="8443"
acceptCount="100" connectionTimeout="20000"
disableUploadTimeout="true" />
```

5. Run Sakai from the root folder `./start-sakai.sh` for NIX systems or `./start-sakai.bat` for Windows.
6. In a web browser, visit `http://localhost:39955/portal`.
7. Log in as user admin with the password as `admin`.
8. Log out.
9. Visit `http://localhost:39955/sakai-axis/SakaiScript.jws?wsdl`.
10. Create a simple unauthenticated web service by adding the following content to `./webapps/sakai-axis/PingTest.jws`:

```
public class PingTest {
    public String ping(String ignore){
        return "Insecure answer =>"+ignore;
    }
    public String pong(String ignoreMeAsWell){
        return youCantSeeMe();
    }
    private String  youCantSeeMe(){
        return "PONG";
}
}
```

11. To verify that the service is available, visit `http://localhost:39955/sakai-axis/PingTest.jws?wsdl`.
12. To verify that the REST services are available, visit `http://localhost:39955/direct`.

How it works...

The Sakai package is self-contained with its own database and Tomcat server. Its main configuration file is `sakai/sakai.properties`. You updated it to allow the use of web services from anywhere. In real-world deployments, the IP address is more restricted.

To avoid port conflict with your local Jenkins server, the Tomcat `conf/server.xml` file was modified.

Sakai has both REST and SOAP web services. You will find the REST services underneath the `/direct` URL. The `many` services are described at `/direct/describe`. Services are supplied one level down. For example, to create or delete users, you would need to use the user service described at `/direct/user/describe`.

The REST services use the Sakai framework to register with **Entitybroker** (`https://confluence.sakaiproject.org/display/SAKDEV/Entity+Provider+and+Broker`). Entitybroker ensures consistent handling between services, saving coding effort. Entitybroker takes care of supplying the services information in the right format. To view who Sakai thinks you currently are in an XML format, visit `http://localhost:399955/direct/user/current.xml`, and to view the JSON format, replace `current.xml` with `current.json`.

The SOAP services are based on the Apache **AXIS framework** (`http://axis.apache.org/axis/`). To create a new SOAP-based web service, you can drop a text file in the `webapps/sakai-axis` directory with the extension `.jws`. Apache AXIS compiles the code on the fly the first time it is called. This allows for rapid application development, as any modifications to the text files are seen immediately by the caller.

The **PingTest** includes a class without a package. The class name is the same as the filename with the `.jws` extension removed. Any `public` methods become web services. If you visit `http://localhost:39955/sakai-axis/SakaiScript.jws?wsdl`, you will notice that the `youCantSeeMe` method is not publicized; that is because it has a `private` scope.

Most of the interesting web services require logging in to Sakai through `/sakai-axis/SakaiLogin.jws` using the method `login`, passing the username and password as strings. The returned string is a **GUID** (a long random string of letters and numbers) that is needed to pass to other methods as evidence of authentication.

To log out at the end of the transaction, use the method `logout`, passing to it the GUID.

There's more...

Sakai CLE is not only a learning management system but also a framework that makes developing new tools straightforward.

The programmer's café for new Sakai developers can be found at the following URL:

`https://confluence.sakaiproject.org/display/BOOT/Programmer%27s+Cafe`

Boot camps based on the programmer's café occur periodically at Sakai conferences or through consultancy engagements. The boot camps walk developers through creating their first Sakai tools using Eclipse as the standard IDE of choice.

Another related product is Sakai **Open Academic Environment** (**OAE**), which is also mentioned at `http://sakaiproject.org`. Sakai OAE builds on the strengths of Sakai CLE and works well with Sakai CLE using the hybrid mode. Hybrid mode allows both systems to share courses.

You can find the description of the book *Sakai CLE Courseware Management: The Official Guide* at the following URL:

`http://www.packtpub.com/sakai-cle-courseware-management-for-elearning-research/book`

See also

- *Writing test plans with SoapUI*
- *Reporting SoapUI test results*

Writing test plans with SoapUI

SoapUI (`http://www.soapui.org/`) is a tool that allows the efficient writing of functional, performance, and security tests, mostly for web services.

In this recipe, you will be using SoapUI to create a basic functional test against the Sakai SOAP web service created in the last recipe.

Getting ready

As described in the previous recipe, we assume that you have Sakai CLE running on port `39955` with the `PingTest` service available.

To download and install SoapUI, visit `http://www.soapui.org/Getting-Started/`, following the installation instructions.

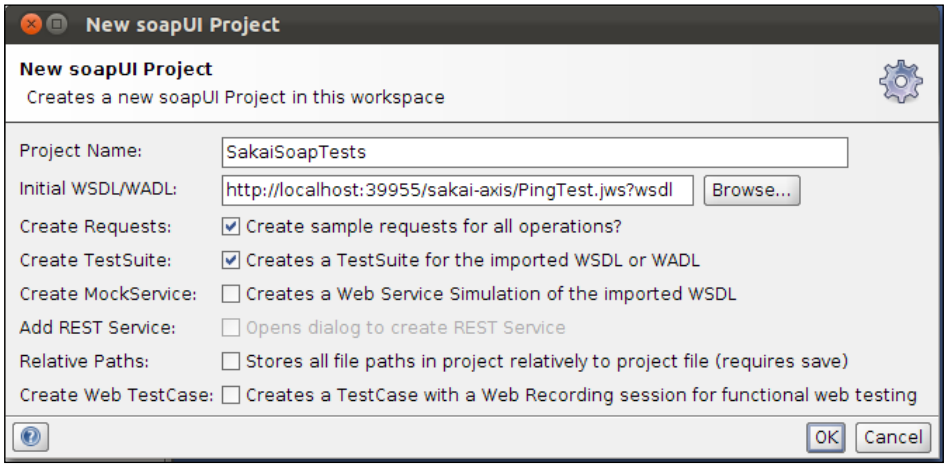
For the Linux package to work with Ubuntu 11.10, you may have to uncomment the following line in the SoapUI startup script:

```
JAVA_OPTS="$JAVA_OPTS -Dsoapui.jxbrowser.disable=true"
```

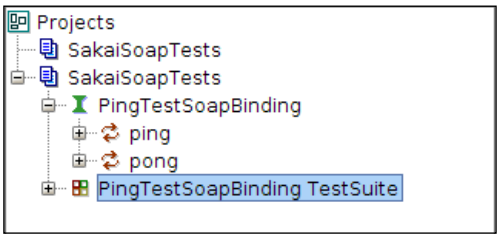
How to do it...

1. Start SoapUI.
2. Right-click on **Projects**, and select **New SoapUI Project**.
3. Fill in the dialog box with the following details:
    - **Project Name**: `SakaiSoapTests`

- **Initial WSDL/WADL**: `http://localhost:39955/sakai-axis/PingTest.jws?wsdl`



4. Check **Create TestSuite**.
5. Click on **OK**.
6. Click on **OK** for the **Generate TestSuite** dialog.
7. Click on **OK** for **TestSuite to create**.
8. In the left-hand side navigator, click on the **+** icon next to **PingTestSoapBinding TestSuite**.



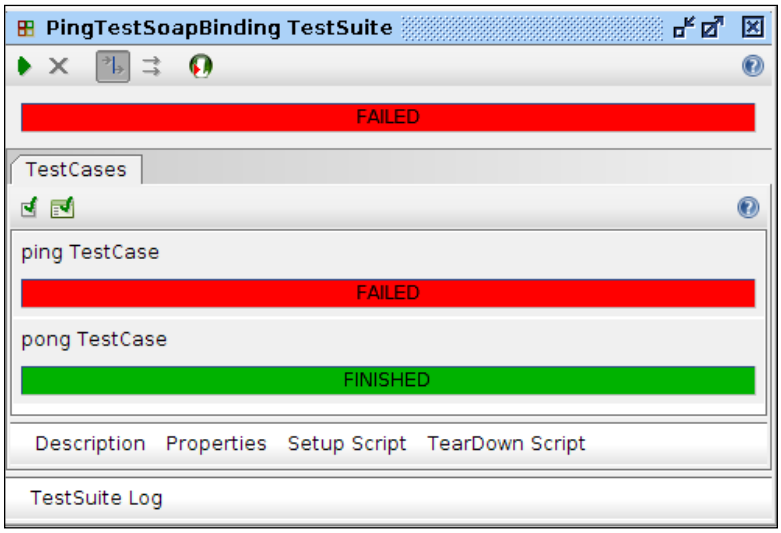
9. Click on the **+** icon next to **Ping TestCase**.
10. Click on the **+** icon next to **Test Steps (1)**.
11. Right-click on **Ping**, and select **Open Editor**.
12. At the top of the editor, click on the **Add assertion** icon.



13. Select **Assertion Not Contains**, and click on **OK**.
14. Add for content **Insecure answer =>?**, and click on **OK**.
15. In the left-hand side navigation, right-click on **PingTestSoapBinding TestSuite**, selecting **Show TestSuite Editor**.
16. In the **Editor**, click on the **Start tests** icon.



17. Review the results. The **ping TestCase** fails due to the assertion, and the **pong TestCase** succeeds.
18. Create the directory named `src/test/soapui`.
19. Right-click on **SakaiSoapTest**, then save the project as `SakaiSoapTests-soapui-project.xml` in the `src/test/soapui` directory.



How it works...

SoapUI takes the drudge work out of making test suites for SOAP services. SoapUI used the PingTest WSDL file to discover the details of the service. The file contains information on the location, and allowable parameters are used with the PingTest service.

From the WSDL file, SoapUI created a basic test for the `Ping` and `Pong` services. You added an assertion under the `Ping` service, checking that the text `Insecure answer =>?` does not exist in the SOAP response. As the text does exist, the assertion failed.

SoapUI has a wide range of assertions that it can enforce, including checking for Xpath or Xquery matches and checking for status codes or assertions tested by custom scripts.

Finally, the project was saved in XML format, ready for reuse in a Maven project in the next recipe.



WSDL stands for **Web Services Description Language** (`http://www.w3.org/TR/wsdl`). A WSDL file is an XML file that supports the discovery of services.

There's more...

SoapUI does a lot more than functional tests for web services. It performs security tests by checking the boundary input. It also has a load runner for stress testing.

Another important feature is its ability to build mock services from WSDL files. This allows the building of tests locally while the web services are still being developed. Early creation of tests reduces the number of defects that reach production-lowering costs. You can find an excellent introduction to mock services at `http://www.soapui.org/Service-Mocking/mocking-soap-services.html`.

See also

- *Enabling Sakai web services*
- *Reporting SoapUI test results*

Reporting SoapUI test results

In this recipe, you will be creating a Maven project that runs the SoapUI test created in the last recipe. A Jenkins project using the **xUnit plugin** (`https://wiki.jenkins-ci.org/display/JENKINS/xUnit+Plugin`) will then parse the results, generating a detailed report.

Getting ready

Install the Jenkins xUnit plugin. Run both the *Enabling Sakai web services* and *Writing test plans with SoapUI* recipes. You will now have Sakai CLE running and a SoapUI test plan ready to use.

How to do it...

1. Create a project directory. At the root of the project, add a `pom.xml` file with the following content:

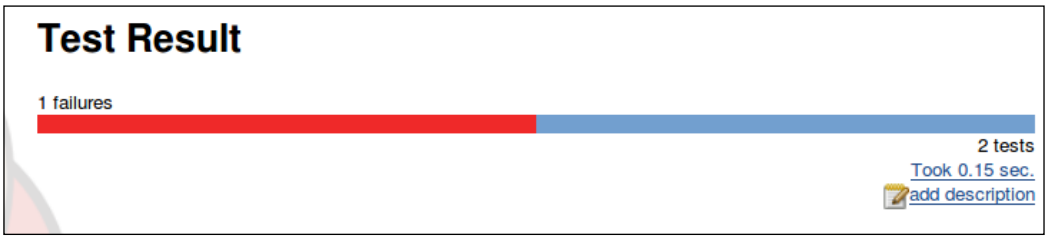
```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <name>Ping regression suite</name>
  <groupId>test.soapui</groupId>
  <artifactId>test.soapui</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <description>Sakai webservices test</description>
  <pluginRepositories>
    <pluginRepository>
      <id>eviwarePluginRepository</id>
      <url>http://www.eviware.com/repository/maven2/</url>
    </pluginRepository>
  </pluginRepositories>
  <build>
    <plugins>
      <plugin>
        <groupId>eviware</groupId>
        <artifactId>maven-soapui-plugin</artifactId>
        <version>4.0.1</version>
        <executions>
          <execution>
            <id>ubyregression</id>
            <goals>
              <goal>test</goal>
            </goals>
            <phase>test</phase>
          </execution>
        </executions>
        <configuration>
          <projectFile>
            src/test/soapui/SakaiSoapTests-soapui-project.xml
          </projectFile>
          <host>localhost:39955</host>
          <outputFolder>
            ${project.build.directory}/surefire-reports
          </outputFolder>
```

```
            <junitReport>true</junitReport>
            <exportwAll>true</exportwAll>
            <printReport>true</printReport>
          </configuration>
        </plugin>
      </plugins>
    </build>
</project>
```

2. Verify that you have correctly placed the SoapUI project at `src/test/soapui/SakaiSoapTests-soapui-project.xml`.
3. Run from the command line:

   ```
   mvn clean test
   ```

4. Log in to Jenkins.
5. Create a Maven 2/3 project named `ch6.remote.soapui`.
6. Under the **Source Code Management** section, check **Subversion**, adding your **Repository URL**.
7. In the **Build** section, under **Goals and options**, add `clean test`.
8. In the **Post-build Actions** section, check **Publish testing tools result report**.
9. Click on the **Add** button.
10. Select **Junit**.
11. Under the **JUNIT Pattern**, add `**/target/surefire-reports/TEST-PingTestSoapBinding_TestSuite.xml`.
12. Click on **Save**.
13. Run the Job.
14. Click on the **Latest Test Result** link. You will see one failed and one succeeded job.



15. You will find the complete details of the failure at `http://localhost:8080/job/ch6.remote.soapui/ws/target/surefire-reports/PingTestSoapBinding_TestSuite-ping_TestCase-ping-0-FAILED.txt`.

How it works...

The Maven project uses the **maven-soapui plugin** (`http://www.soapui.org/Test-Automation/maven-2x.html`). As the plugin is not available in one of the main Maven repositories, you had to configure it to use the `eviwarePluginRepository` repository.

The SoapUI plugin was configured to pick up its plan from the project file `src/test/soapui/SakaiSoapTests-soapui-project.xml` and save the results relative to `project.build.directory`, which is the root of the workspace.

The options set were:

```
<junitReport>true</junitReport>
<exportwAll>true</exportwAll>
<printReport>true</printReport>
```

`JunitReport` set to `true` tells the plugin to create a JUnit report. `exportwAll` set to `true` implies that the results of all tests are exported, not just the errors. This option is useful during the debugging phase and should be set on unless you have severe disk space constraints. `printReport` set to `true` ensures that Maven sends a small test report to the console with an output similar to the following:

```
SoapUI 4.0.1 TestCaseRunner Summary
------------------------------
Total TestSuites: 1
Total TestCases: 2 (1 failed)
Total Request Assertions: 1
Total Failed Assertions: 1
Total Exported Results: 1
[ERROR] java.lang.Exception: Not Contains in [ping] failed;
[Response contains token [Insecure answer =>?]]
```

The `ping` test case failed as the assertion failed. The `pong` test case succeeded as the service existed. Therefore, even without assertions, using the autogeneration feature of SoapUI allows you to quickly generate a scaffold that ensures that all services are running. You can always add assertions later as the project develops.

Creation of the Jenkins Job is straightforward. The xUnit plugin allows you to pull in many types of unit test including the JUnit ones created from the Maven project. The location is set in *step 10* as `**/target/surefire-reports/TEST-PingTestSoapBinding_TestSuite.xml`.



The **custom reports** option is yet another way of pulling in your own custom data and displaying their historic trends within Jenkins. It works by parsing the XML results found by the plugin with a custom stylesheet. This gives you a great deal of flexibility to add your own custom results.

There's more...

The `Ping` service is dangerous as it does not filter the input, and the input is reflected back through the output.

Many web applications use web services to load the content into a page, avoiding reloading the full page. A typical example is when you type in a search term and alternative suggestions are shown on the fly. With a little social engineering magic, a victim will end up sending a request including scripting to the web service. On returning the response, the script is run in the client browser. This bypasses the intent of the same origin policy (`http://en.wikipedia.org/wiki/Same_origin_policy`). This is known as a **non-persistent attack**, as the script is not persisted to storage.

Web services are more difficult to test than web pages for XSS attacks. Luckily, SoapUI simplifies the testing process to a manageable level. You can find an introductory tutorial on SoapUI security tests at `http://www.soapui.org/Security/working-with-security-tests.html`.

See also

- *Enabling Sakai web services*
- *Writing test plans with SoapUI*

7
Exploring Plugins

In this chapter, we will cover the following recipes:

- Personalizing Jenkins
- Testing and then promoting
- Fun with pinning JS Games
- Looking at the GUI Samples plugin
- Changing the help of the File system scm plugin
- Adding a banner to Job descriptions
- Creating a RootAction plugin
- Exporting data
- Triggering events on startup
- Triggering events when web content changes
- Reviewing three ListView plugins
- Creating my first ListView plugin

Introduction

This chapter has two purposes. The first is to show a number of interesting plugins. The second is to briefly review how plugins work. If you are not a programmer, then feel free to skip the how plugins work discussion.

When I started writing this book, there were over 300 Jenkins plugins available; at the time of writing this page, there are more than 400. It is likely that there are plugins already available that meet or nearly meet your needs. Jenkins is not only a Continuous Integration Server but also a platform to create extra functionality. Once a few concepts are learned, a programmer can adapt the available plugins to his/her organization's needs.

If you see a feature that is missing, it is normally easier to adapt an existing one than to write from scratch. If you are thinking of adapting, then the plugin tutorial (`https://wiki.jenkins-ci.org/display/JENKINS/Plugin+tutorial`) is a good starting point. The tutorial gives relevant background information on the infrastructure you use daily.

There is a large amount of information available on plugins. Here are some key points:

- There are many plugins, and more will be developed. To keep up with these changes, you will need to regularly review the available section of the Jenkins plugin manager.
- Work with the community: If you centrally commit your improvements, then they become visible to a wide audience. Under the careful watch of the community, the code is likely to be improved.
- Don't reinvent the wheel: With so many plugins, in the majority of situations, it is easier to adapt an already existing plugin than write from scratch.
- Pinning a plugin occurs when you cannot update the plugin to a new version through the Jenkins plugin manager. Pinning helps to maintain a stable Jenkins environment.
- Most plugin workflows are easy to understand. However, as the number of plugins you use expands, the likelihood of an inadvertent configuration error increases.
- The Jenkins Maven Plugin allows you to run a test Jenkins server from within a Maven build without any risk.
- Conventions save effort: The location of files in plugins matters. For example, you can find the description of a plugin displayed in Jenkins at the file location `/src/main/resources/index.jelly`.
- By keeping to Jenkins conventions, the amount of source code you write is minimized and the readability is improved.
- The three frameworks that are heavily used in Jenkins are:
  - **Jelly** for the creation of the GUI
  - **Stapler** for the binding of the Java classes to the URL space
  - **Xstream** for persistence of configuration into XML

Personalizing Jenkins

This recipe highlights two plugins that improve the user experience: the **green balls** plugin and the **favorites plugin**.

Jenkins has a wide international audience. At times, there can be subtle cultural differences expressed in the way Jenkins looks. One example is when a build succeeds, a blue ball is shown as the icon. However, many Jenkins users naturally associate the green from traffic lights as the signal to go further.

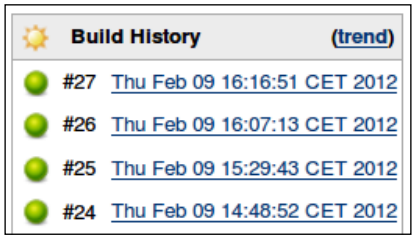
The favorites plugin allows you select your favorite projects and display an icon in a view to highlight your picks.

Getting ready

Install the green balls and favorite plugins (`https://wiki.jenkins-ci.org/display/JENKINS/Green+Balls`, `https://wiki.jenkins-ci.org/display/JENKINS/Favorite+Plugin`).

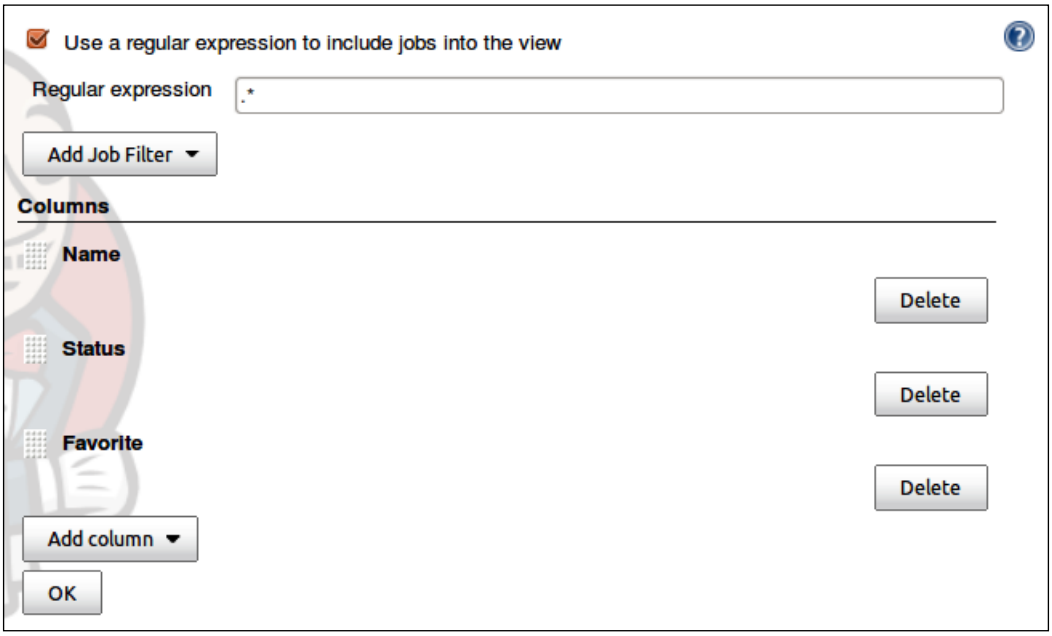
How to do it...

1. Create an empty new free-style job named `ch7.plugin.favourit`.
2. Build the Job a number of times, reviewing the build history. You will now see green balls instead of the usual blue.

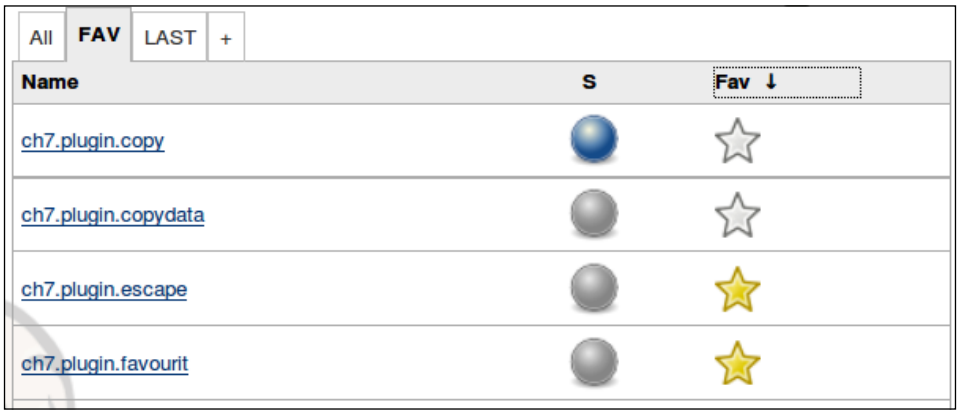


3. Return to the main page.
4. To create a new view, click on the **+** icon.
5. Fill in `FAV` for the **Name**.
6. Under the **Job Filters** section, check **Use a regular expression to include jobs into the view**. Add `.*` for **Regular expression**.

7. In the **Columns** section, make sure you have three columns: **Name**, **Status**, and **Favorite**.



8. Click on **OK**.
9. You will find yourself in the **FAV** view. By clicking on the star icon, you can select/deselect your favorite projects.



How it works...

The green balls plugin works as advertised. However, one limitation is that it does not currently affect the standard list view, which still displays blue balls.

The favorites plugin allows you to select which project interests you the most and displays that as a favorites icon. This reminds you that the project needs some immediate action.



If you are interested in working with the community, then these plugins are examples that you could add extra features to.

There's more...

The opposite of a favorite project, at least temporarily, is a project whose build has failed. The **claims plugin** (`https://wiki.jenkins-ci.org/display/JENKINS/Claim+plugin`) allows individual developers to claim a failed build. This enables the mapping of workflow to individual responsibilities.

Once the claims plugin is installed, you will be able to find a tickbox in the **Post-Build Actions** section of a Job for **Allow broken build claiming**. Once enabled, if a build fails, you can claim a specific build, adding a note about your motivation.



Within the Jenkins home page, there is now a link to a log that keeps a summary of all the claimed builds. A project manager can now read a quick overview of issues. The log is a direct link to the team members who deal with current issues.



The favorites plugin is elegant in its simplicity. The next recipe, *Testing and then promoting*, will signal further, allowing you to incorporate complex workflows.

See also

- *Testing and then promoting*
- *Fun with pinning JSGames*

Testing and then promoting

You do not want the QA team to review a packaged application until it has been automatically tested. To achieve this, you can use the **promotion plugin**.

Promotion is a visual signal in Jenkins. An icon is set next to a specific build to remind the team to perform an action.

The difference between the promotion and the favorites plugin mentioned in the last recipe is that the promotion plugin can be triggered automatically based on a variety of automated actions. Actions include the running of scripts or the verification of the status of other up or downstream jobs.

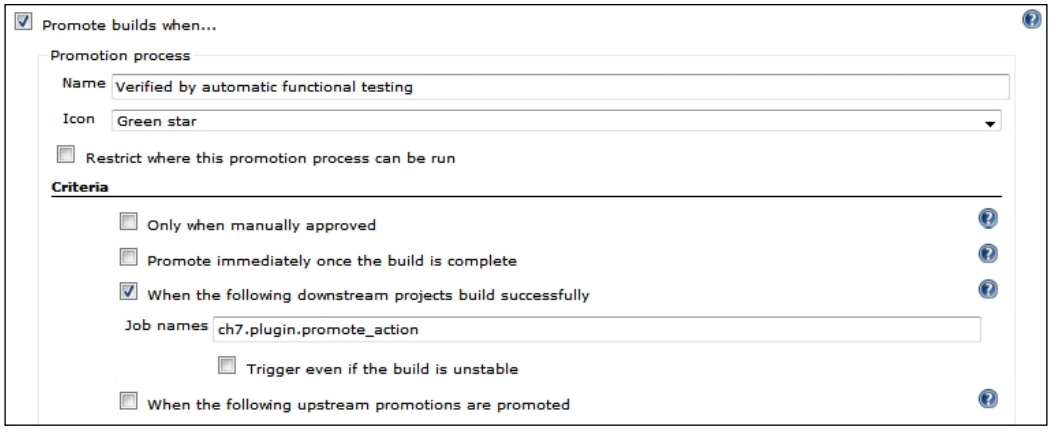
In this recipe, you will be writing two simple Jobs. The first Job will trigger the second Job, and if the second Job is successful, then the first Job will be promoted. This is the core of a realistic QA process – the testing Job promoting the packaging Job.

Getting ready

Install the promoted builds plugin `https://wiki.jenkins-ci.org/display/JENKINS/Promoted+Builds+Plugin`.

How to do it...

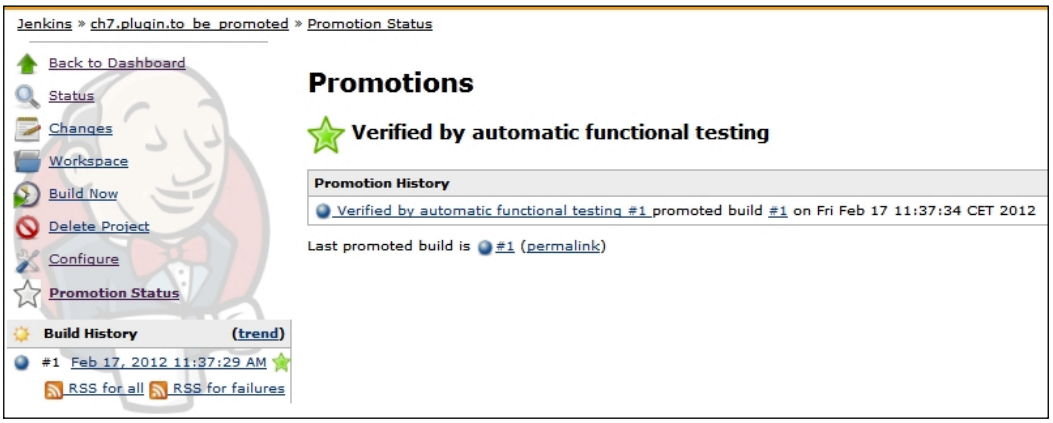
1. Create a free-style Job named `ch7.plugin.promote_action`.
2. Run this Job and verify that it succeeds.
3. Create a free-style Job named `ch7.plugin.to_be_promoted`.
4. Near the top of the configuration page, check **Promote builds when...**.
5. Fill in the following details:
   - **Name**: `Verified by automatic functional testing`
   - Select `Green star` for **Icon**
   - Check **When the following downstream projects build successfully**
   - **Job names**: `ch7.plugin.promote_action`



6. In the **Post-build Action** section, check **Build other projects**.
7. Fill in `ch7.plugin.promote_action` for **projects to build**.
8. Tick **Trigger only if build succeeds**.
9. Click on **Save**.
10. Build the Job.
11. Click on the **Promotion Status** link.



12. Review the build report.



How it works...

Promoted builds is similar to the favorites plugin, but with automation of workflow. You can promote depending on job(s) triggered by the creation of artifacts. This is a typical workflow when you want a Job tested for baseline quality before being picked up and reviewed.

The plugin has enough configuration options to make it malleable to most workflows. Another example, for a typical development, acceptance, or production infrastructure, is that you do not want an artifice to be deployed to production before development and acceptance have also been promoted. The way to configure this is to have a series of Jobs with the last promotion to production, depending on the promotion of upstream development and acceptance jobs.

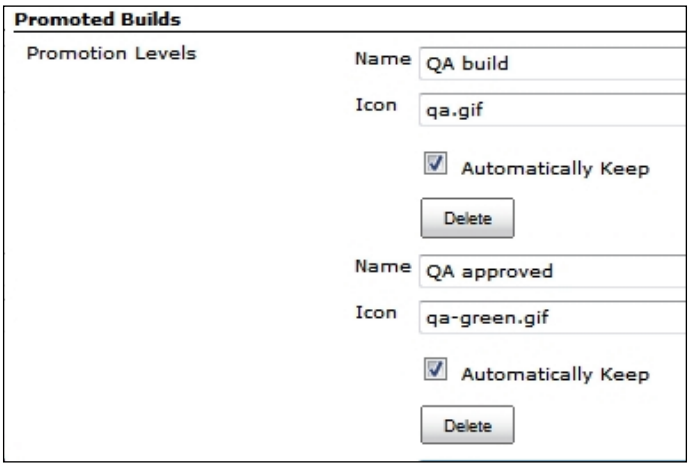


If you want to add human intervention, then check **Only when manually approved** in **Jobs configuration** and add a list of approvers.

There's more...

If you are relying on human intervention and have no automatic tests, consider using the simplified promoted builds plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Promoted+Builds+Simple+Plugin`). As its name suggests, the plugin simplifies the configuration and works well with a large subset of QA workflows. Simplifying the configuration eases the effort of explaining, allowing use by a wider audience.

You can configure the different types of promotion within the main Jenkins configuration page.



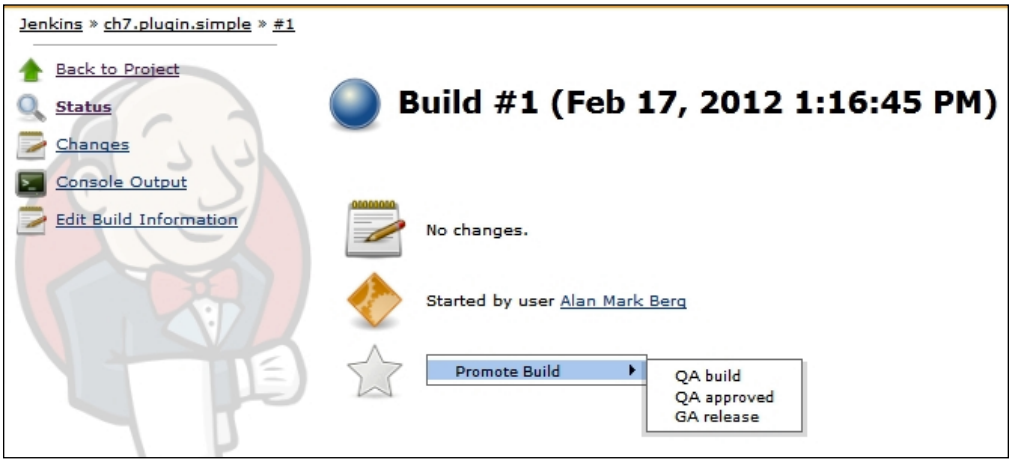


Warning: Use the **Automatically Keep** feature wisely. The option tells Jenkins to keep the artifacts from the build for all time. If used as part of an incremental build process, you will end up consuming a lot of disk space.

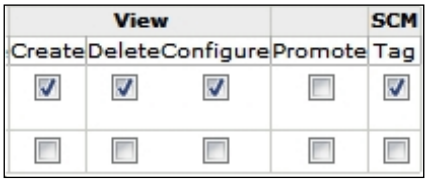
The plugin allows you to elevate promotions. There is a simple choice available through a link on the left-hand side of the build. This feature allows you to add a series of players into the promotion process.



Warning: When the final promotion occurs, for example, when you set the promotion to **Generally Available** (**GA**), the promotion is locked and can no longer be demoted.



The ability of a user to promote depends upon the permissions granted to them. For example, if you are using matrix-based security, then you will need to update its table before you can see an extra option in the configuration page of the Job.



See also

- *Personalizing Jenkins*

Fun with pinning JS Games

This recipe shows you how to pin a Jenkins plugin. Pinning a plugin stops you from being able to update its version within the Jenkins plugin manager.

Now that the boss has gone, life is not always about code quality. To let off pressure, consider allowing access to your team to relaxation with the JS Games plugin.

Getting ready

Install the JS Games plugin (`https://wiki.jenkins-ci.org/display/JENKINS/JSGames+Plugin`).

How to do it...

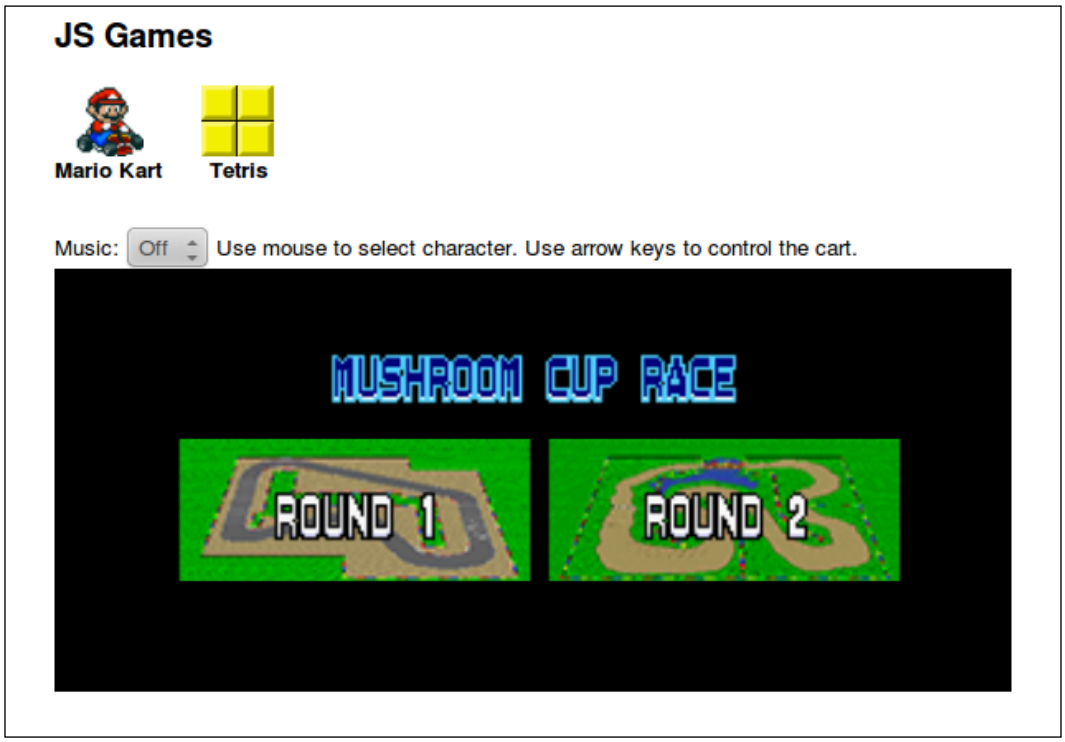
1. Checkout and review the tag jsgames-0.2 with the commands: :

```
git clone https://github.com/jenkinsci/jsgames-plugin
git tag
git checkout jsgames-0.2
```

2. Review the front page of Jenkins; you will see a link to **JS Games**.



3. Click on the link and you will have the choice of two games: **Mario Kart** and **Tetris**.



4. As a Jenkins administrator, visit the **Manage Plugins** section, and click on the **installed** tab (`http://localhost:8080/pluginManager/installed`). Notice that the JS Games plugin is not pinned.
5. From the command line, list the contents of the plugin directory (`JENKINS_HOME/plugin`), for example:

   ```
   ls /var/lib/jenkins/plugins
   ```

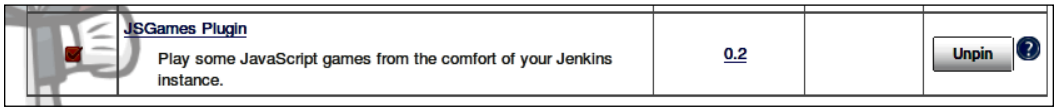
   The output will be similar to the following:

   ```
   ant                             ant.jpi
   jsgames                      jsgames.jpi
   maven-plugin              maven-plugin.jpi
   ```

6. In the `plugins` directory, create a file named `jsgames.jpi.pinned`. For example:

   ```
   sudo touch /var/lib/jenkins/plugins/jsgames.jpi.pinned
   sudo chown jenkins /var/lib/jenkins/plugins/jsgames.jpi.pinned
   ```

7. In your web browser, refresh the **Installed Plugin** page. You will now see that the **JSGames Plugin** is pinned.



How it works...

Pinning a plugin stops a Jenkins administrator from updating to a new version of a plugin. To pin a plugin, you need to create a file in the `plugins` directory with the same name as the plugin ending with the extension pinned. See `https://wiki.jenkins-ci.org/display/JENKINS/Pinned+Plugins`.

Roughly every week, a new version of Jenkins is released with bug fixes and feature updates. This leads to delivering improvements quickly to the market, but also leads to failures at times. Pinning a plugin implies that you can stop a plugin from being accidentally updated until you have had time to assess the stability and value of the newer version. Pinning is a tool to maintain the production server stability.

There's more...

The source code includes a top-level `pom.xml` to control the Maven build process. By convention, the four main source code areas are:

- `src/test`: This is the code that tests during the build. For `jsgames`, there are a bunch of JUnit tests.
- `src/main/java`: This is the location of the Java code. Jenkins uses **Stapler** (`https://wiki.jenkins-ci.org/display/JENKINS/Architecture`) to map the data between the Java Objects in this directory and the views that Jenkins finds in the directories below `src/main/resources`.
- `src/main/resources`: This is the location of the view for the plugin. The GUI is associated with the plugin you see when you interact in Jenkins; for example, the link to JS Games. The view is defined using **Jelly tags**.
- `src/main/webapp`: This is the location of resources, such as images, stylesheets, and JavaScript. The location maps to the URL space. `/src/main/webapp` maps to the URL `/plugin/name_of_plugin`. For example, the location `/src/main/webapp/tetris/resources/tetris.js` maps to the URL `/plugin/jsgames/tetris/resources/tetris.js`.

See also

- *Creating a RootAction plugin*

Looking at the GUI Samples plugin

This recipe describes how to run a Jenkins test server through Maven. In the test server, you will get to see the example GUI plugin. The GUI plugin demonstrates a number of tag elements that you can use later in your own plugins.

Getting ready

Create a directory to keep the results of this recipe.

How to do it...

1. In the recipe directory, add `pom.xml` with the following content:

```
<?xml version="1.0"?>
  <project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.jenkins-ci.plugins</groupId>
    <artifactId>plugin</artifactId>
    <version>1.449</version>
  </parent>
  <artifactId>Startup</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>hpi</packaging>
  <name>Startup</name>
  <repositories>
    <repository>
      <id>m.g.o-public</id>
      <url>http://maven.glassfish.org/content/groups/public/</url>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>m.g.o-public</id>
      <url>http://maven.glassfish.org/content/groups/public/</url>
    </pluginRepository>
  </pluginRepositories>
</project>
```

2. From the command line, run `mvn hpi:run`. If you have a default Jenkins running on port `8080`, then you will see an error message similar to the following:

```
2012-02-05 09:56:57.827::WARN:  failed SelectChannelConnector @
0.0.0.0:8080
java.net.BindException: Address already in use
        at sun.nio.ch.Net.bind0(Native Method)
```

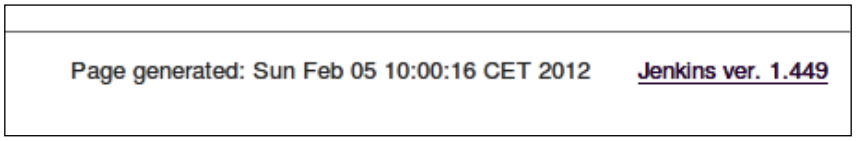
3. If the server is still running, press *Ctrl + C*.
4. To run on port `8090`, type the following command:

```
mvn hpi:run -Djetty.port=8090
```

5. The server will now run and generate a `SEVERE` error from the console.

```
SEVERE: Failed Inspecting plugin  /DRAFT/Exploring_plugins/hpi_
run/./work/plugins/Startup.hpl
java.io.IOException: No such file:  /DRAFT/Exploring_plugins/hpi_
run/target/classes
```

6. Visit `localhost:8090`. At the bottom of the page, review the version number of Jenkins.



7. Click on the **UI Samples** link.



8. Review the various types of examples mentioned, such as `AutoCompleteTextBox` (`http://localhost:8090/ui-samples/AutoCompleteTextBox/`).

How it works...

For development purposes, the ability to run a test server from Maven is great. You can change your code, compile, package, and then view it on a local instance of Jenkins, without worrying about configuring or damaging a real server. You do not have to worry too much about security, because the test server only runs as long as you are testing.

The goal `hpi:run` tries to package and then deploy a plugin called **Startup**. However, the package is not available, so it logs a complaint and then faithfully runs a Jenkins server. The version number of the Jenkins server is the same as the version number defined in the `pom.xml <version>` tag within the `<parent>` tag.

To avoid hitting the same port as your local instance of Jenkins, you set the `jetty.port` option.

Once Jenkins is running, visiting the GUI example plugin will show you many different GUI elements written in the Jelly language. These elements will later come in handy for programming your own plugins. The Jelly files used in plugins sit under the `/src/main/resources` directory. Jenkins uses Stapler to bind any relevant classes found in `src/main/java`.

You can find the Jenkins workspace in the `work` folder. Any configuration changes you make on the test server are persisted here. To have a fresh start, you will need to delete the directory by hand.

For all the recipes in this chapter, we will pin the Jenkins version at `1.449`. The reason for this is two-fold:

- The dependencies take a lot of space. The Jenkins WAR file and test WAR file take about 120 MB of your local Maven repository. Multiply this number by the number of versions of Jenkins used, and you can quickly fill up GBs of disk space.
- Holding at a specific Jenkins version stabilizes the recipes.

Feel free to update to the newest and greatest Jenkins version; the examples in this chapter should still work. In case of difficulty, you can always return to the known safe number.

There's more...

Behind the scenes, Maven does a lot of heavy lifting. The `pom.xml` file defines the repository `http://maven.glassfish.org/content/groups/public/` to pull in the dependencies. It calls version `1.449` of `org.jenkins-ci.plugins.plugin`. The version number is in sync with the version number of Jenkins that Maven runs.

To discover which version numbers are acceptable, visit the following URL:

```
http://maven.jenkins-ci.org/content/groups/artifacts/org/jenkins-ci/
plugins/plugin/
```

The details of the Jenkins server and any extra plugins can be found relative to this URL in `1.449/plugin-1.449.pom`. The `ui-samples-plugin` version is also pegged at version `1.449`.

See also

- *Changing the help of the file system scm plugin*

Changing the help of the file system scm plugin

This recipe reviews the inner workings of the file system scm plugin. This plugin allows you to place the code in a local directory that is then picked up in a build.

Getting ready

Create a directory named `ready` for the code in this recipe.

How to do it...

1. Download the source of the plugin.

   ```
   svn export -r 40275  https://svn.jenkins-ci.org/trunk/hudson/
   plugins/filesystem_scm
   ```

2. In the top-level directory, edit the `pom.xml` file by changing the version of `<parent>` to `1.449`.

   ```
   <parent>
     <groupId>org.jenkins-ci.plugins</groupId>
     <artifactId>plugin</artifactId>
     <version>1.449</version>
   </parent>
   ```

3. Replace the content of `src/main/webapp/help-clearWorkspace.html` with the following:

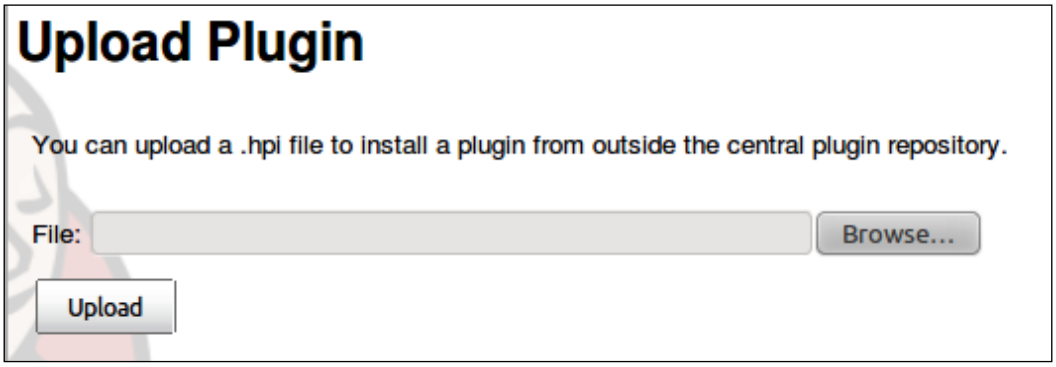
   ```
   <div>
     <p>
       <h3>HELLO WORLD</h3>
     </p>
   </div>
   ```

4. Run `mvn clean install`. The unit tests fail with the following output:

   ```
   Failed tests:   test1(hudson.plugins.filesystem_scm.
   SimpleAntWildcardFilterTest): expected:<2> but was:<0>

   Tests run: 27, Failures: 1, Errors: 0, Skipped: 0
   ```

5. Skip the failing tests by running `mvn clean package -Dmaven.test.skip=true`. The plugin is now packaged.

6. Upload the plugin `./target/filesystem_scm.hpi` in the **Advanced** section of your plugin manager (`http://localhost:8080/pluginManager/advanced`).



7. Restart the Jenkins server.
8. Log in to Jenkins, and visit the list of installed plugins (`http://localhost:8080/pluginManager/installed`).
9. Create a Maven 2/3 Job named `ch7.plugins.filesystem_scm`.
10. Under **Source Code Management**, you now have a section called **File system**.
11. Click on the **help** icon. You will see your custom message.



12. To delete the plugin, remove the JPI file and the expanded directory from under `JENKINS_HOME/plugins`.
13. Restart Jenkins.

How it works...

Congratulations! You have updated the scm plugin.

First, you modified the plugin's `pom.xml` file, updating the version of the test Jenkins server. Next, you modified its `help` file.

For each Java class, you can configure its GUI representation through an associated `config.jelly` file. The mapping is from `src/main/java/package_path/classname.java` to `src/main/resources/package_path/classname/config.jelly`.

For example, `src/main/resources/hudson/plugins/filestem_scm/FSSCM/config.jelly` configures the Jenkins GUI for `src/main/java/hudson/plugins/filesystem_scm/FSSCM.java`.

The location of the `help` files are defined in `config.jelly` with the attribute `help` in the `entry` Jelly tag:

```
<f:entry title="Clear Workspace" help="/plugin/filesystem_scm/
  help-clearWorkspace.html">
  <f:checkbox name="fs_scm.clearWorkspace"
    checked="${scm.clearWorkspace}"/>
</f:entry>
```

The `src/main/webapps` directory provides a stable Jenkins URL `/plugin/name_of_plugin` for static content, such as images, stylesheets, and JavaScript files. This is why the `help` files were stored here. Modifying `help-clearWorkspace.html` updated the help pointed to by the entry tab.

The variable `${scm.clearworkspace}` is a reference to the value of the `clearWorkspace` member in the `FSSCM` instance.

There's more...

Plugins generally ship with two types of Jelly files: `global.jelly` and `config.jelly`. `config.jelly` files generate the configuration elements seen when configuring Jobs. `global.jelly` files are rendered in the main Jenkins configuration page.

Data is persisted in XML files using the Xstream framework. You can find the data for Job configuration under the working area of Jenkins within `./jobs/job_name/plugin_name.xml`, and `./work/name_of_plugin.xml` for the global plugin configuration.

See also

- *Looking at the GUI Samples plugin*

Adding a banner to Job descriptions

Scenario: Your company has a public-facing Jenkins instance. The owner does not want the project owners to write unescaped tagging in the descriptions of projects. This poses too much of a security issue. However, the owner does want to put a company banner at the bottom of each description. You have 15 minutes to sort out the problem before the management starts buying in unnecessary advice. Within the first five minutes, you ascertain that the escape markup plugin (see *Finding 500 errors and XSS attacks in Jenkins Through Fuzzing*, *Chapter 2*, *Enhancing Security*) performs the escaping of the description.

This recipe shows you how to modify the markup plugin to add a banner to all the descriptions.

Getting ready

Create a directory for your project.

How to do it...

1. Check out the `escape-markup-plugin-0.1` tag of `escaped-markup-plugin`.

   ```
   git clone https://github.com/jenkinsci/escaped-markup-plugin
   cd escaped-markup-plugin
   git checkout escaped-markup-plugin-0.1
   ```

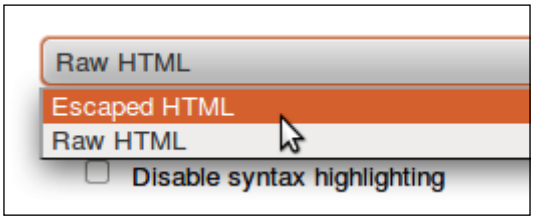
2. In the top-level directory of the project, try to create the plugin by using the command `mvn install`. The build fails.
3. Change the Jenkins plugin version in the pom.xml from 1.408 to 1.449:

   ```
   <parent>
     <groupId>org.jenkins-ci.plugins</groupId>
     <artifactId>plugin</artifactId>
     version>1.449</version>
   </parent>
   ```

4. Build the plugin with `mvn install`. The build and tests will succeed. You can now find the plugin at `target/escaped-markup-plugin.hpi`.
5. Install the plugin by visiting the **Advanced** tab under the plugin Manager (`http://localhost:8080/pluginManager/advanced`).
6. In the **Upload Plugin** section, upload the `escaped-markup-plugin.hpi` file.
7. Restart the server; for example:

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

8. Visit the Jenkins configuration page (`http://localhost:8080/configure`), and review the markup formatters.



9. Replace `src/main/resources/index.jelly` with the following:

```
<div>
  This plugin escapes the description of project , user, view ,
    and build  to prevent from XSS.
  Revision: Unescaped banner added at the end.
</div>
```

10. Replace the class definition of `src/main/java/org/jenkinsci/plugins/escapedmarkup/ EscapedMarkupFormatter.java` with:

```
public class EscapedMarkupFormatter extends MarkupFormatter {
  private final String BANNER= "\n<hr><h2>THINK BIG WITH xyz dot
    blah</h2><hr>\n";

    @DataBoundConstructor
    public EscapedMarkupFormatter() {
    }

    @Override
    public void translate(String markup, Writer output) throws
       IOException {
        output.write(Util.escape(markup)+BANNER);
    }

    @Extension
    public static class DescriptorImpl extends
      MarkupFormatterDescriptor {
        @Override
        public String getDisplayName() {
            return "Escaped HTML with BANNER";
        }
    }
}
```

11. Build with `mvn install`. The build fails due to failed tests (which is a good thing).
12. Build again using the following command, this time skipping the tests:

```
mvn -Dmaven.test.skip=true -DskipTests=true clean install
```

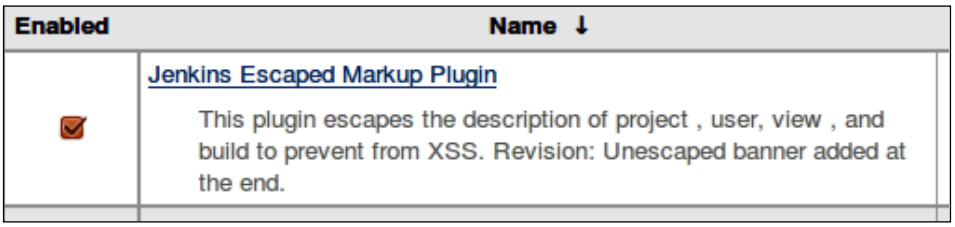
13. Stop Jenkins; for example:

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

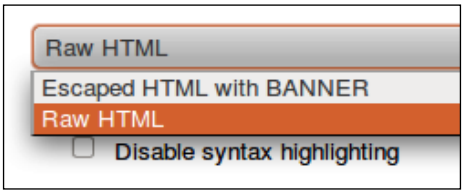
14. Delete the escaped markup plugin from the Jenkins plugin directory and the expanded version in the same directory. For example:

```
sudo rm /var/lib/jenkins/plugins/escaped-markup-plugin.jpi
sudo rm -rf /var/lib/jenkins/plugins/escaped-markup-plugin
```

15. Copy the `target/escaped-markup-plugin.hpi` plugin to the Jenkins plugin directory.
16. Restart Jenkins.
17. Visit the **Installed plugins** page at `http://localhost:8080/pluginManager/installed`. You will now see an updated description of the plugin.

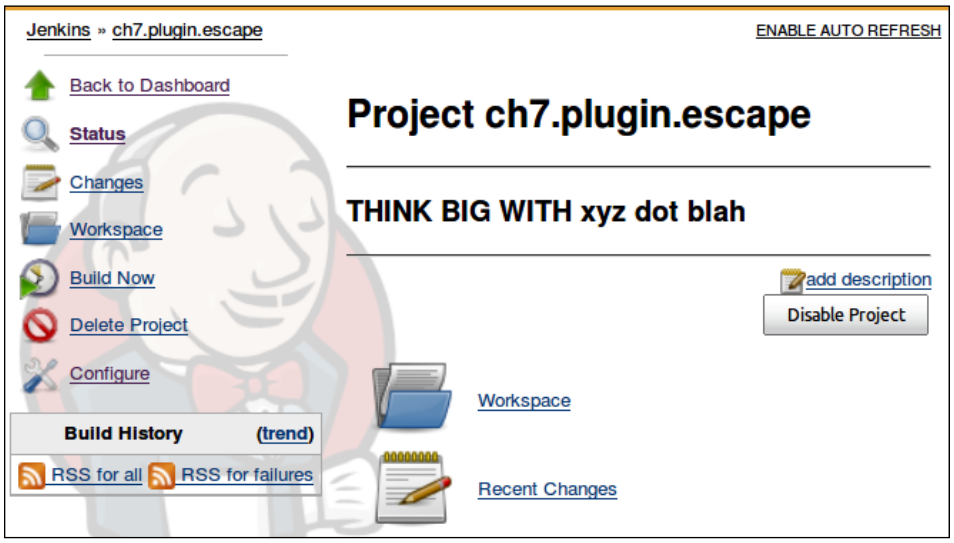


18. In Jenkins, as an administrator, visit the configure page at `http://localhost:8080/configure`.
19. For **Markup Formatter**, choose **Escape HTML with Banner**:



20. Click on **Save**.
21. Create a new Job named `ch7.plugin.escape`.

22. Within the Jobs main page, you will now see the banner.



How it works...

The markup plugin escapes the tags in descriptions so that arbitrary scripting actions cannot be injected. The use of the plugin was explained in the recipe *Finding 500 errors and XSS attacks in Jenkins Through Fuzzing*, *Chapter 2*, *Security*.

In this recipe, we adapted the plugin to escape a project's description, and then added a banner. The banner contains arbitrary HTML.

First, you compiled and uploaded the markup plugin. Then, you modified the source to include a banner at the end of a Job's description. The plugin was redeployed to a sacrificial test instance that was ready for review. You could have also used the `mvn hpi:run` goal to run Jenkins through Maven. There are multiple ways to deploy, including dumping the plugin directly into the Jenkins plugin directory. Which of the deployment methods you decide to use is a matter of taste.

The description of the plugin rendered is defined in `src/main/resources/index.jelly`. You updated the file to accurately describe the new banner feature.

In Jenkins, extension points are Java interfaces or abstract classes that model a part of the Jenkins functionality. Jenkins has a wealth of extension points (`https://wiki.jenkins-ci.org/display/JENKINS/Extension+points`). You can even make your own extension point (`https://wiki.jenkins-ci.org/display/JENKINS/Defining+a+new+extension+point`).

The markup plugin had minimal changes made to it to suite our purposes. We extended the `MarkupFormatter` extension point.

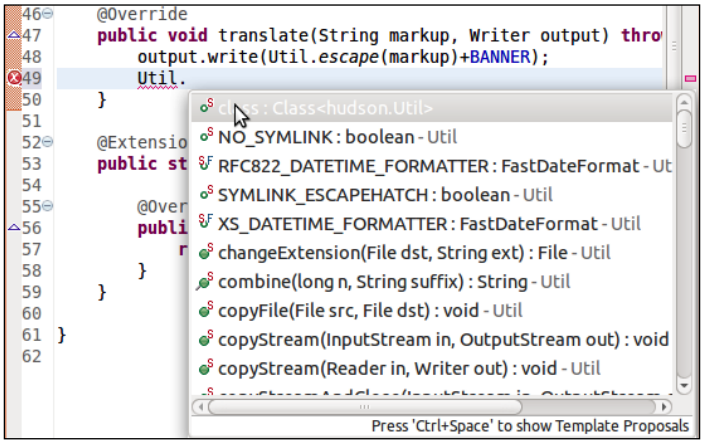
Jenkins uses annotations. The `@Override` annotation tells the compiler to override the method. In this case, we overrode the `translate` method and used a utility class to filter the markup string using a Jenkins `utility` method. At the end of the resulting string, the banner string was added and passed to the Java writer. The writer is then passed back to the calling method.

The text inside the `selectbox` (see *step 19*) of the plugin is defined in the `getDisplayName()` method of the `DescriptorImpl` class.

Conclusion: Writing a new plugin, and understanding the Jenkins object model takes more effort than copying a plugin that works and then tweaking it. The amount of code change needed to add the banner feature to an already existing plugin was minimal.

There's more...

There is a lot of documentation available for Jenkins. However, for the hardcore programmer, the best source of details is reviewing the code. Examples include JavaDoc (`http://javadoc.jenkins-ci.org/`) and the built-in code completion facilities of IDEs, such as Eclipse. If you import the Jenkins plugin project into Eclipse as a Maven project, then the newest versions of Eclipse will sort out the dependencies for you, enabling code completion during the editing of files. In a rapidly-moving project such as Jenkins, there is sometimes a lag between when a feature is added and when it is documented. In this situation, the code needs to be self-documenting. Code completion in combination with well-written JavaDoc eases a developer's learning curve.



See also

- *Finding 500 errors and XSS attacks in Jenkins Through Fuzzing*, *Chapter 2*, *Enhancing security*
- *Changing the help of the file system scm plugin*
- *Creating a RootAction plugin*

Creating a RootAction plugin

Before building your own plugin, it is worth seeing if you can adapt another's. In the **Fun with pinning JS Games** recipe, the plugin created a link on the front page.



In this recipe, we shall use the elements of the plugin to create a link in the Jenkins home page.

Getting ready

Create a directory to store your source code.

How to do it...

1. Create a copy of the `pom.xml` file from the *Looking at the GUI Samples plugin and hpi:run* recipe, and replace:

```
<artifactId>Startup</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>hpi</packaging>
 <name>Startup</name>
```

   with:

```
<artifactId>rootaction</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>hpi</packaging>
 <name>Jenkins Root Action Plugin</name>
```

2. Create the directories `src/main/java/Jenkins/plugins/rootaction`, `src/main/resources` and `src/main/webapp`.
3. In `src/main/java/Jenkins/plugins/rootaction`, add the file `MyRootAction.java` with the following content:

```
package jenkins.plugins.rootaction;
import hudson.Extension;
import hudson.model.RootAction;

@Extension
public class MyRootAction implements RootAction {

    public final String getDisplayName() {
        return "Root Action Example";
    }

    public final String getIconFileName() {
        return "/plugin/rootaction/myicon.png";
    }

    //Feel free to modify the URL
    public final String getUrlName() {
        return "http://www.uva.nl";
    }
}
```

4. In the `src/main/webapp` directory, add a PNG file named `myicon.png`. For an example image, see `http://www.iconfinder.com/icondetails/46509/32/youtube_icon`.
5. Add the `src/main/resources/index.jelly` file with the following content:

```
<div>
  This plugin adds a root link.
</div>
```

6. In the top-level directory, run the following command:

```
mvn -Dmaven.test.skip=true -DskipTests=true clean install hpi:run
-Djetty.port=8090
```

7. Visit the main page at `http://localhost:8090`.



8. Click on the **Root Action Example** link; your browser is now sent to the main website of the University of Amsterdam (`http://www.uva.nl`).
9. Review the Jenkins installed plugin page (`http://localhost:8090/pluginManager/installed`).

How it works...

You implemented the `RootAction` extension point. It is used to add links to the main menu in Jenkins.

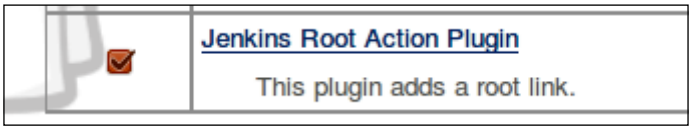
The extension point is easy to extend. The link name is defined in the `getDisplayName` method, the location of an icon in the `getIconFileName` method, and the URL to link to in `getUrlName`.

There's more...

Conventions save programming effort. By convention, the description of the plugin is defined in `src/main/resources/index.jelly`, and the link name in the `pom.xml` file under the `<name>` tag next to the `<packaging>` tag. For example:

```
<artifactId>rootaction</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>hpi</packaging>
<name>Jenkins Root Action Plugin</name>
```

The location of the details in the Jenkins Wiki are calculated as a fixed URL (`http://wiki.jenkins-ci.org/display/JENKINS/`) with the plugin name, after that with the spaces in the name replaced with `+` symbols. This is true for this plugin as well, which has the link generated `http://wiki.jenkins-ci.org/display/JENKINS/Jenkins+Root+Action+Plugin`.



See also

- *Fun with pinning JS Games*

Exporting data

The job exporter plugin creates a property file with a list of project-related properties. This is a handy glue for when you want Jenkins to pass the information from one job to another.

Getting ready

Install the Job exporter plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Job+Exporter+Plugin`).

How to do it...

1. Download the source code at a known version number.

   ```
   svn export -r 40275 https://svn.jenkins-ci.org/trunk/hudson/
   plugins/job-exporter
   ```

2. Create a free-style job named `ch7.plugins.job_export`.
3. In the `build` section, add a build step `export runtime parameters`.
4. Click on `Save`.
5. Run the Job.
6. In the build History for the Job within the console output, you will see output similar to the following:

   ```
   Started by user Alan Mark Berg
   Building in workspace /var/lib/jenkins/workspace/ch7.plugins.job_
   export
   ```

```
#######################################################################
#
job-exporter plugin  started
    hudson.version: 1.450
    host:
    id: 2012-02-02_15-58-51
    duration: 2 ms
    slave:
    started: 2012-02-02T15:58:51
    result: SUCCESS
    summary: Executor #0 for master : executing ch7.plugins.job_
export #1
    executor: 0
    elapsedTime: 3
    number: 1
    jobName: ch7.plugins.job_export
  we have 1 build cause:
      Cause.UserIdCause  Started by user Alan Mark Berg
    user.id: Alan
    user.name: Alan Mark Berg
    user.fullName: Alan Mark Berg
    user.emailAddress: xxx@yyy.nl
  new file written: /var/lib/jenkins/workspace/ch7.plugins.job_
export/hudsonBuild.properties
job-exporter plugin  finished.  That's All Folks!
#######################################################################
#
Finished: SUCCESS
```

7. Reviewing the newly created properties file; you will see a text similar to the following:

```
#created by com.meyling.hudson.plugin.job_exporter.ExporterBuilder
#Thu Feb 02 15:58:51 CET 2012
build.user.id=Alan
build.result=SUCCESS
```

How it works...

The Job exporter plugin gives Jenkins the ability to export Job-related information into a properties file that can later be picked up for re-use by other Jobs.

Reviewing the code `src/main/java/com/meyling/hudson/plugin/job_exporter/ExporterBuilder.java` extends `hudson.tasks.Builder`, whose `perform` method is invoked when a build is run. The `perform` method receives the `hudson.model.Build` object when it is called. The `Build` instance contains information about the build itself. Calling the `build.getBuiltOnStr()` method returns a string, which contains the name of the node that the build is running on. The plugin uses a number of these methods to discover the information that is later outputted to a properties file.

There's more...

While reviewing the plugin code, you can find interesting tricks ready for re-use in your own plugin. The plugin discovered the environment variables by using the following method:

```
final EnvVars env = build.getEnvironment(new
  LogTaskListener(Logger.getLogger(
  this.getClass().getName()), Level.INFO));
```

Here, `EnvVars` is of the class `hudson.EnvVars` (`http://javadoc.jenkins-ci.org/hudson/EnvVars.html`). `EnvVars` even has a method to get the environment variables from remote Jenkins nodes.

You can also find a list of all environment variables defined for Jenkins in the **Jenkins Management** area under system info (`http://localhost:8080/systemInfo`).

See also

- *My first ListView plugin*

Triggering events on startup

Often when a server starts up, you will want to have a clean-up action performed. For example, running a Job that sends an e-mail to all of the Jenkins admins warning them of the startup event. You can achieve this with the **startup trigger plugin**.

Getting ready

Install the startup trigger plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Startup+Trigger`).

How to do it...

1. Download the source code.

   ```
   svn export -r 40275 https://svn.jenkins-ci.org/trunk/hudson/
   plugins/startup-trigger-plugin
   ```

2. Create a free-style Job named `ch7.plugin.startup`.
3. In the section **Build Triggers**, check **Build when Jenkins first starts**.
4. Click on **Save**.
5. Restart Jenkins.
6. Return to the project page; you will notice that a Job has been triggered.
7. Review the builds history console output. You will see an output similar to the following:

   ```
   Started due to Jenkins startup.
   Building in workspace /var/lib/jenkins/workspace/ch7.plugins.
   startup
   Finished: SUCCESS
   ```

How it works...

The startup trigger plugin runs a Job at startup. This is useful for administrative tasks, such as reviewing the file system. It is also concise in its design.

The startup trigger plugin extends `hudson.triggers.Trigger` in the `/src/main/java/org/jvnet/hudson/plugins/triggers/startup/HudsonStartupTrigger` Java class, and overrides the method `start`, which is later called during the startup of Jenkins.

The `start` method calls the parents `start` method, and if it is not a new instance, it will call the `project.scheduleBuild` method that then starts the build.

```
@Override
  public void start( BuildableItem project, boolean newInstance )
  {
    super.start( project, newInstance );

    // do not schedule build when trigger was just added to the job
    if ( !newInstance )
    {
      project.scheduleBuild( new HudsonStartupCause() );
    }
  }
```

The cause of the startup is defined in `HudsonStartupCause`, which itself extends `hudson.model.Cause`. The plugin overrides the `getShortDescription()` method, returning the string `Started due to Hudson startup`. The string is outputted to the console as part of the logging.

```
@Override
  public String  getShortDescription()
  {
    return "Started due to Hudson startup.";
  }
```

See also

- *Triggering events when web content changes*

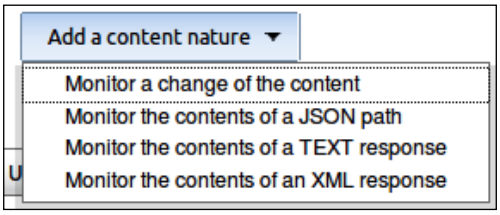
Triggering events when web content changes

In this recipe, the URL trigger plugin will trigger a build if a web page changes its content.

Jenkins is deployed in varied infrastructures. There will be times when standard plugins cannot be triggered by your system of choice. Web servers are well-understood technologies. In most situations, the system to which you want to connect to has its own web interface. If the application does not, then you can still set up a web page, which changes when the application needs a reaction from Jenkins.

How to do it...

1. Create a new free-style Job named `ch7.plugin.url`.
2. In the **Build Triggers** section, check the **[URLTrigger] - Poll with a URL** checkbox.
3. Click on **Add URL to monitor**.
4. For **URL**, add `http://www.google.com`.
5. Check **Inspect URL content**.
6. Select **Monitor a change of the content** from **Add a content nature**.



7. For the **schedule** input, add the text `*****`. This sets the **schedule** to **once a minute**.
8. Click on **Save**.
9. On the right-hand side, there is a link to **URLTrigger Log**. Click on this link.



10. You will now see the log information update once a minute with content similar to the following:

```
Polling for the job ch7.plugin.url
Polling on master.
Polling started on Feb 9, 2012 4:55:45 PM
Invoking the url:
 http://www.google.com
Inspecting the content
The content of the URL has changed.
Polling complete. Took 0.21 sec.
Changes found. Scheduling a build.
```

11. Delete the Job, as we don't want to poll Google every minute.

How it works...

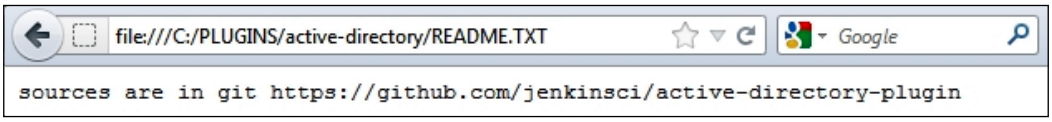
You configured the plugin to visit google.com once a minute, and downloaded and compared the Google page for changes. A schedule of once a minute is aggressive; consider using similar time intervals as for your SCM repositories, for example, once every five minutes.

As there are subtle differences in each Google page returned, the trigger is activated. This was verified by looking in the **URLTrigger Log**.

The **URLTrigger plugin** can also be used for JSON and text or XML responses. In the future, expect more options.

There's more...

Part of the URI schema is for pointing to your local file system (`http://en.wikipedia.org/wiki/File_URI_scheme`). You get to see examples of this when you load a local file into your web browser.



Changes in the local file system cannot be monitored by this plugin. If you reconfigure the Job to point at the location `file:///`, you will get the following error message:

```
java.lang.ClassCastException: sun.net.www.protocol.file.FileURLConnection
cannot be cast to java.net.HttpURLConnection
```

You will have to use the file system SCM plugin instead.

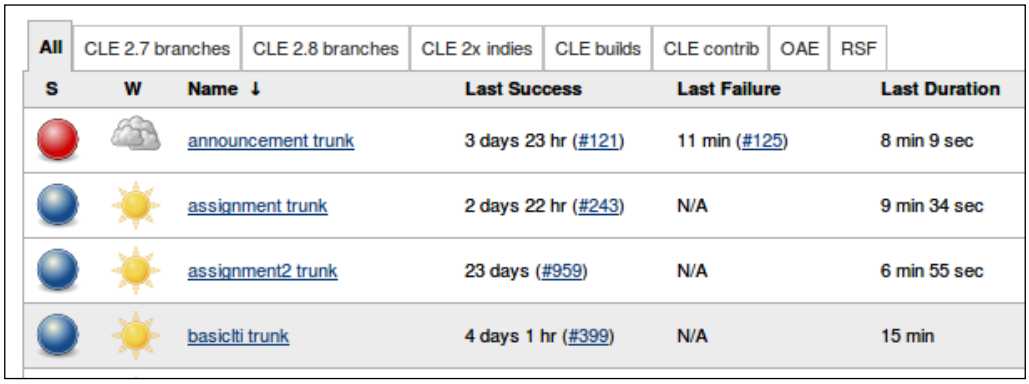
See also

- *Triggering events on startup*

Reviewing three ListView plugins

The information radiated out by the front page of Jenkins is important. The initial perception of the quality of your projects is likely to be judged by this initial encounter.

In this recipe, we will review the **Last Success**, **Last Failure**, and **Last Duration** columns that you can add to the list view.



In the next recipe, you will be shown how to write a plugin for your own column in the list view.

Getting ready

Install the List View Columns plugin, Last Failure Version Column plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Last+Failure+Version+Column+Plugin`), Last Success Description Column plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Last+Success+Description+Column+Plugin`), and the Last Success Version Column plugin (`https://wiki.jenkins-ci.org/display/JENKINS/Last+Success+Version+Column+Plugin`).

How to do it...

1. Install the source code locally in a directory of choice.

```
git clone https://github.com/jenkinsci/lastfailureversioncolumn-plugin
git clone https://github.com/jenkinsci/lastsuccessversioncolumn-plugin
svn export -r 40277
https://svn.jenkins-ci.org/trunk/hudson/plugins/lastsuccessdescriptioncolumn
```

2. Check out the correct tag in the git source code:

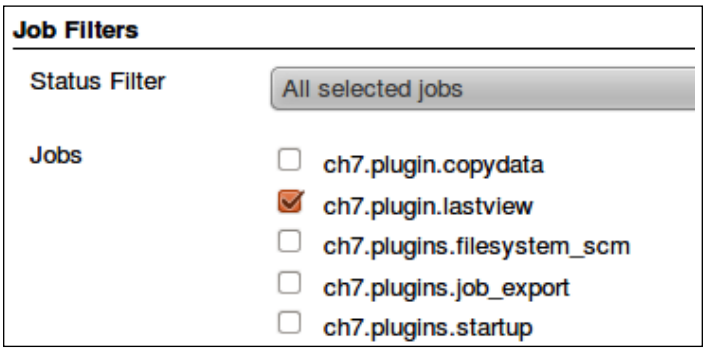
```
cd lastfailureversioncolumn-plugin
git checkout lastfailureversioncolumn-1.1
cd ../lastsuccessversioncolumn-plugin
git checkout lastsuccessversioncolumn-1.1
```

3. In Jenkins, create a new free-style Job named `ch7.plugin.lastview`. No further configuration is needed.
4. On the **Main Page**, press the + tab next to the **All** tab.

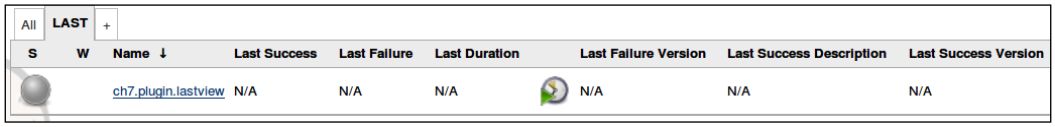


5. Create a **List View** named `LAST`.

6. Under **Job Filters | Jobs**, check the **ch7.plugin.lastview** checkbox.



7. Click on **OK**. You will be returned to the main page with the **LAST** list view showing.



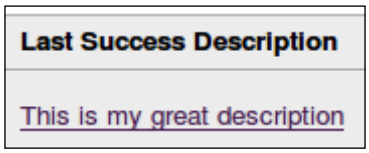
8. Click on the build icon to run the `ch7.plugin.lastview` Job.



9. Refresh your page. The **Last Success Version** column now has data with a link to the build's history.
10. In the **Last Success Description** column, click on the **N/A** link.
11. On the right-hand side, click on **add description**.
12. Add the description for the build `"This is my great description"`.
13. Click on **Submit**.
14. Return to the **LAST** list view by clicking on **LAST** in the breadcrumb displayed at the top of the page.



15. The **Last Success Description** column is now populated.



How it works...

The three plugins perform similar functions; the only difference is a slight variation in the details of the columns. The details are useful for making quick decision about projects. You can give the casual viewer an oversight into the last significant action in the project without them diving down into the source code. When a build succeeds, add a meaningful description to the build, such as "Updated core libraries to work with modern browsers". This exposure of information through the correct use of descriptions saves a significant amount of clicking.



There's more...

There is a healthy market of **ListView** plugins. These include:

- **Extra Columns plugin**: It adds options for counting the number of successful and failed builds, a shortcut to the configure page of the project, an enable/disable project button, and a project description button. Each one of these new columns allows you to better understand the state of the project or perform actions efficiently.
- **Cron Column plugin**: It displays the scheduled triggers in the project and shows whether they are enabled or disabled. This is useful if you want to compare the system-monitoring information with the melody plugin.
- **Emma Coverage plugin**: It displays the code coverage results reported by the emma plugin. This is especially useful if your organization has an in-house style guide where the code needs to reach a specific level of code coverage.
- **Progress Bar plugin**: It displays a progress bar for running Jobs. This adds a feeling of activity to the front page.

See also

- *Creating my first ListView plugin*
- *Efficient use of views*, *Chapter 4*, *Communicating Through Jenkins*
- *Saving screen space with the Dashboard plugin*, *Chapter 4*, *Communicating Through Jenkins*
- *Monitoring through JavaMelody*, *Chapter 1*, *Maintaining Jenkins*

Creating my first ListView plugin

In this final recipe, you will create your first custom ListView plugin. This allows you to add an extra column to the standard list view with a column with comments. The code for the content of the column is a placeholder, just waiting for you to replace with your own brilliant experiments.

Getting ready

Create a directory that is ready for the code.

How to do it...

1. Create a top-level `pom.xml` file with the content of `pom.xml` from the *Creating my first RootAction plugin* recipe. Change the `<parent>` section from:

```
<artifactId>Startup</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>hpi</packaging>
 <name>Startup</name>
```

   to the content:

```
<artifactId>commentscolumn</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>hpi</packaging>
   <name>Jenkins Fake Comments Plugin</name>
```

2. Create the `src/main/java/jenkins/plugins/comments` directories.

3. In the `comments` directory, add `CommentsColumn.java` with the following content:

```
package jenkins.plugins.comments;
import org.kohsuke.stapler.StaplerRequest;
import hudson.views.ListViewColumn;
import net.sf.json.JSONObject;
import hudson.Extension;
import hudson.model.Descriptor;
import hudson.model.Job;

public class CommentsColumn extends ListViewColumn {
  public String getFakeComment(Job job) {
    return "Comments for <em>"+job.getName()+"</em>"+
      "Short URL: <em>"+job.getShortUrl()+"</em>";
  }

  @Extension
  public static final Descriptor<ListViewColumn> DESCRIPTOR = new
    DescriptorImpl();

  public Descriptor<ListViewColumn> getDescriptor(){
    return DESCRIPTOR;
  }

  private static class DescriptorImpl extends
    Descriptor<ListViewColumn> {
    @Override
    public ListViewColumn newInstance(StaplerRequest req,
      JSONObject formData) throws FormException {
      return new CommentsColumn();
    }

    @Override
    public String getDisplayName() {
      return "FakeCommentsColumn";
    }
  }
}
```

4. Create the `src/main/resources/jenkins/plugins/comments/CommentsColumn` directory.

5. In the `CommentsColum` directory, add `column.jelly` with the following content:

```
<j:jelly xmlns:j="jelly:core">
  <j:set var="comment" value="${it.getFakeComment(job)}"/>
  <td data="${comment}">${comment}</td>
</j:jelly>
```

6. In the `CommentsColum` directory, `add columnHeader.jelly` with the following content:

```
<j:jelly xmlns:j="jelly:core">
    <th>${%Fake Comment}</th>
</j:jelly>
```

7. In the `CommentsColumn` directory, add `columnHeader.properties` with the following content:

```
Fake\ Comment=My Fake Column [Default]
```

8. In the `CommentsColumn` directory, add `columnHeader_an.properties` with the following content:

```
Fake\ Comment=My Fake Column [an]
```

9. In the `src/main/resources` directory, add the plugin description `index.jelly` file with the following content:

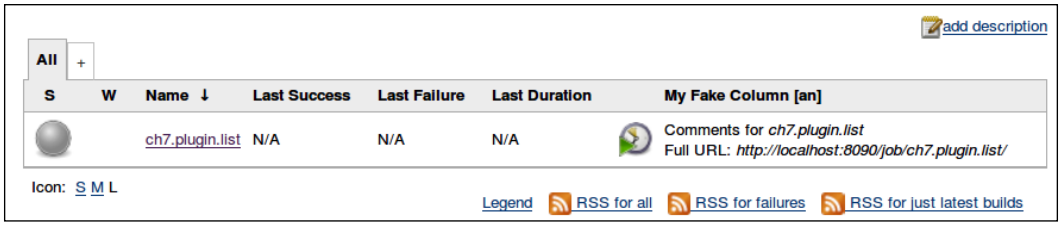
```
<div>
  This plugin adds a comment to the sections mentioned in list
view.
</div>
```

10. In the top-level directory, run the following command:

```
mvn -Dmaven.test.skip=true clean install hpi:run -Djetty.port=8090
```

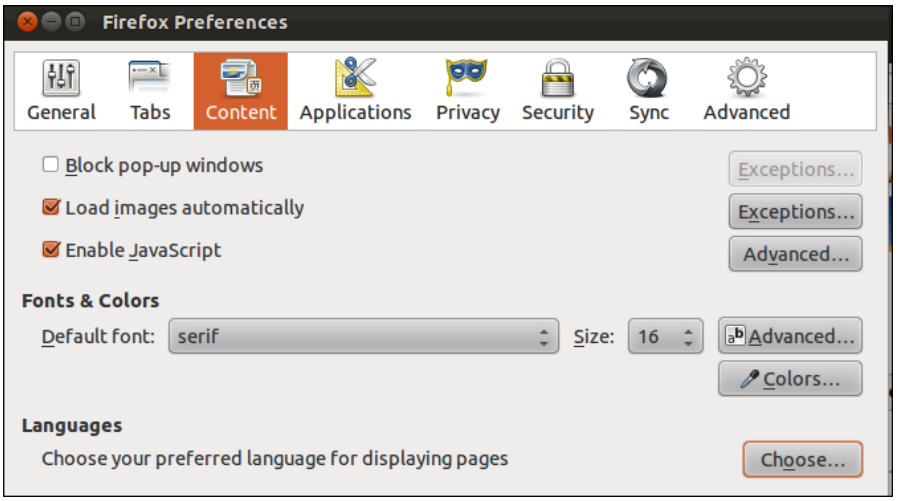
11. Visit the Jenkins Job creation page at `http://localhost:8090/view/All/newJob`. Create a new free-style Job named `ch7.plugin.list`.

    On the main Jenkins page, `http://localhost:8090`, you will now have a view with the column called **My Fake Column [Default]**. If you change the preferred language of your web browser to `Aragonese [an]`, then the column will now be called **My Fake Column [an]**.





In the default Firefox browser for Ubuntu, you can change the preferred language under the **Edit | Preferences | Content** tab, in the section **Languages**.



How it works...

In this recipe, a basic ListView plugin was created with the following structure:

```
├── pom.xml
└── src
    └── main
        ├── java
        │   └── jenkins
        │       └── plugins
        │           └── comments
        │               └── CommentsColumn.java
        └── resources
            ├── index.jelly
            └── jenkins
```

```
└── plugins
    └── comments
        └── CommentsColumn
            ├── columnHeader_an.properties
            ├── columnHeader.jelly
            ├── columnHeader.properties
            └── column.jelly
```

The one Java file included in the plugin is `CommentsColumn.java` under `/src/main/java/Jenkins/plugins/comments`. The class extends the `ListViewColumn` extension point.

The method `getFakeComment` expects an input of type Job and returns a String. This method is used to populate the entries in the column.

The GUI in the ListView is defined under `/src/main/resources/packagename/Classname/`. You find the GUI for `/src/main/java/ Jenkins/plugins/comments/CommentsColumn.java` mapped to the `/src/main/resources/Jenkins/plugins/comments/CommentsColumn` directory. In this directory, there are two Jelly files: `columnHeader.jelly` and `column.jelly`.

As the name suggests, `columnHeader.jelly` renders the header of the column in the ListView. Its contents are as follows:

```
<j:jelly xmlns:j="jelly:core">
    <th>${%Fake Comment}</th>
</j:jelly>
```

`FAKE Comment` is defined in `columnHeader.properties`. The `%` sign tells Jelly to look in different properties files depending on the value on the **Language** settings returned by the web browser. In this recipe, we set the web browser's language value to **an**, and this translates to looking for the `columnHeader_an.properties` file first. If the web browser returns a language that does not have its own properties file, then Jelly defaults to `columnHeader.properties`.

`columns.jelly` has the following content:

```
<j:jelly xmlns:j="jelly:core">
  <j:set var="comment" value="${it.getFakeComment(job)}"/>
  <td data="${comment}">${comment}</td>
</j:jelly>
```

`it.getFakeComment` calls the method `getFakeComment` on an instance of the `CommentsColumn` class. It is the default name for the instance of the object. The type of object returned is defined by convention by the file structure `/src/main/resources/Jenkins/plugins/comments/CommentsColumn`.

The returned string is placed in the variable `comment` and then displayed inside a `<td>` tag.



If you are curious about the Jelly tags available in Jenkins, then review `https://wiki.jenkins-ci.org/display/JENKINS/Understanding+Jelly+Tags`.

There's more...

If you want to participate in the community, then the Governance page is a necessary read (`https://wiki.jenkins-ci.org/display/JENKINS/Governance+Document`). On the subject of licensing, the page states:

> *The core is entirely in the MIT license, so are the most infrastructure code (that runs the project itself), and many plugins. We encourage hosted plugins to use the same MIT license, to simplify the story for users, but plugins are free to choose their own licenses, so long as it's OSI-approved open-source license.*

You can find the list of approved OSI licenses at `http://opensource.org/licenses/alphabetical`.

The majority of plugins have a `License.txt` file in their top-level directory with an **MIT license** (`http://en.wikipedia.org/wiki/MIT_License`). For an example, review `https://github.com/jenkinsci/lastfailureversioncolumn-plugin/blob/master/LICENSE.txt`. It has a structure, which is similar to the following:

```
The MIT License

Copyright (c) 20xx, Name x, Name y…

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.
```

```
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

See also

- *Reviewing three ListView plugins*

Processes that Improve Quality

Quality Assurance requires the expert to pay attention to a wide range of details. Rather than being purely technical, many of these details relate to human behavior. Here are a few hard-learned observations.

Avoiding group think

It is easy to be perfect on paper, defining the importance of a solid set of JavaDocs and unit tests. However, the real world on its best days is chaotic. Project momentum, motivated by the need to deliver, is an elusive force to push back against.

Related to project momentum is the potential of group think (`http://en.wikipedia.org/wiki/Groupthink`) by the project team or resource owners. If the team has the wrong collective attitude, then as a Quality Assurance professional, it is much harder to inject the hard-learnt realism. Quality Assurance is not only about finding and capturing defects as early as possible but also about injecting the objective criteria for success or failure into the different phases of a project's cycle.

Consider adding measurable criteria into the Jenkins build. Obviously, if the code fails to compile, then the product should not go to acceptance and then production. Less obviously, are the rules around code coverage of unit tests worth defending in release-management meetings?

Try getting the whole team involved at the start of the project before any coding has taken place, and agree on the metrics that fail a build. One approach is to compare a small successful project to a small failed project. If there is a disagreement later, then the debate is about the processes and numbers rather than personality.

See the *Looking for smelly code through Code coverage* recipe in *Chapter 5*, *Using Metrics to Improve Quality*.

Considering test automation as a software project

If you see automated testing as a software project and apply well-known principles, then you will save on maintenance costs and increase the reliability of tests.

The **Don't Repeat Yourself** (**DRY**) principle is a great example. Under time pressure, it is tempting to cut-and-paste similar tests from one area of the code base to another—DON'T. Projects evolve bending the shape of the code base, and the tests need to be re-usable to adapt to that change. Fragile tests push up maintenance costs. One concrete example discussed briefly in *Chapter 6*, *Testing Remotely* is the use of page objects with Selenium Webdriver. If you separate the code into pages, then when the workflow between pages changes, most of the testing code remains intact.

See the *Activating more PMD rulesets* recipe in *Chapter 5*, *Using Metrics to Improve Quality*.

The *Keep It Simple Stupid* (*KISS*) principle implies keeping every aspect of the project as simple as possible. For example, it is possible to use real browsers for automated functional tests or use the **HtmlUnit** framework to simulate a browser. The second choice avoids the need to set up an in-memory X server (or VNC – `http://en.wikipedia.org/wiki/Virtual_Network_Computing`) and will also keep a track of browser versioning. These extra chores decrease the reliability of running a Jenkins Job, but do increase the value of the tests. Therefore, for small projects, consider starting with HtmlUnit. For larger projects, the extra effort is worth the cost.

See the *Triggering failsafe integration tests with Selenium Webdriver* recipe in *Chapter 3*, *Building Software*.

Consider if you need a standalone integration server or if you can get away with using a Jetty server called during the integration goal in Maven. For an example recipe, see the *Configuring Jetty for integration tests recipe in Chapter 3*, *Building Software*.

Offsetting work to Jenkins nodes

Jenkins usage can grow virally in an organization. Testing and JavaDoc generation takes up a lot of system resources. A master Jenkins is best used to report back quickly on Jobs distributed across a range of Jenkins nodes. This approach makes it easier to analyze where the failure lies in the infrastructure.

See the *Monitoring through JavaMelody* recipe in *Chapter 1*, *Maintaining Jenkins*.

See the *Running multiple Jenkins nodes* recipe in *Chapter 6*, *Testing Remotely*.

Learning from history

Teams tend to have their own coding habits. If a project fails because of the quality of the code, try and work out which code metrics would have stopped the code from reaching production or which mistakes are seen repeatedly; a few examples include the following:

- **Friday afternoon code failure**: We are all human and have secondary agendas. By the end of the week, a programmer may have their minds focused elsewhere than the code. A small subset of programmers have their code quality affected, consistently injecting more defects towards the tail end of their roster. Consider scheduling a weekly Jenkins Job that has harsher thresholds for quality metrics pushing back near the time of least attention.
- **Code churn**: A warning for experienced quality assurers is the sudden surge in code commits just before a product is moved from an acceptance environment to production. This indicates that there is a last-minute rush. For some teams with a strong sense of code quality, this is also a sign of extra vigilance. For other less-disciplined teams, this could be a naive push towards destruction. If a project fails and QA is overwhelmed due to a surge of code changes, then look at setting up a warning Jenkins Job based on the commit velocity. If necessary, you can display your own custom metrics.

  See the *Plotting alternative code metrics in Jenkins* recipe in *Chapter 3*, *Building Software*.
- **A rogue coder**: Not all code bashers create code of the same uniform and high quality. It is possible that there is consistent underachievement within a project. Rogue coders are caught by human code review. However, for a secondary defense, consider setting thresholds on static code review reports from FindBugs and PMD. If a particular developer is not following the accepted practice, then builds will fail with great regularity.

  See the *Finding bugs with Findbugs* recipe in *Chapter 5*, *Using Metrics to Improve Quality*.

- **The GUI does not make sense**: Isn't it painful when you build a web application only to be told at the last moment that the GUI does not quite interact in the way the product owner expected? One solution is to write a mockup in Fitnesse, and surround it with automatic-functional tests using fixtures. When the GUI diverges from the planned workflow, Jenkins will start shouting.

  See the *Activating the Fitnesse HTMLUnit Fixtures* recipe in *Chapter 6*, *Testing Remotely*.

- **Tracking responsibility**: Mistakes are made and lessons need to be learned. However, if there is no clear chain of documented responsibility, then it is difficult to pin down who needs the learning opportunity. One approach is to structure the workflow in Jenkins through a series of connected jobs, and use the promoted builds plugin to make sure the right group verifies at the right point. This methodology is also good for reminding the team of the short-term tasks.

  See the *Testing and then promoting* recipe in *Chapter 7*, *Exploring Plugins*.

Test frameworks are emerging

In the past few years, there has been a lot of improvement in test automation. Static code review is being used more thoroughly for security. Sonar is an all-encompassing reporter of project quality, and new frameworks are emerging to improve on the old. Here are a few implications:

- **Sonar measures project quality**: Its community is active. Sonar will evolve faster than the full range of Jenkins quality metrics plugins. Consider using Jenkins plugins for early warnings of negative quality changes and Sonar for the in-depth reporting.

  See the *Integrating Jenkins with SONAR* recipe in *Chapter 5*, *Using Metrics to Improve Quality*.

- **Static code review tools are improving**: FindBugs has moved comment making into the cloud. More bug pattern detectors are being developed. Static code review tools are getting better at finding security defects. Expect significantly improved tools over time, possibly just by updating the version of your current tools.

  See the *Finding security defects with Findbugs* recipe in *Chapter 5*, *Using Metrics to Improve Quality*.

- **Code searching**: Wouldn't it be great if code search engines, such as koders, ranked the position in their search results of a particular piece of code, based on the defect density or coding practice? You could then search a wide range of open source products for best practices. You could search for defects to remove and then send patches back to the codes communities.

- **The cloud**: Cloudbee allows you to create on-demand slave nodes in the cloud. Expect more kinds of cloud, such as integrations, around Jenkins.

Starve QA/ integration servers

A few hundred years ago, coal miners would die because of the build-up of methane in the mines. To give an early warning of this situation, canary birds were brought into the mines. Being more sensitive, the birds would faint first, giving the miners enough time to escape. Consider doing the same for your integration servers; deliberately starve them of resources. If they fall over, you will have enough time to review before watching the explosion in production.

And there's always more

There are always more points to consider. Here are a few of the cherry-picked ones:

- **Blurring the team boundary**: Tools such as Fitnesse and Selenium IDE make it easier for non-Java programmers to write tests. The easier it is to write tests, the more likely the tests reflect user behavior. Look for new Jenkins plugins that support tools that lower the learning curve.

  See the *Running Selenium IDE tests* recipe in *Chapter 6*, *Testing Remotely*.

- **Deliberately adding defects**: By rotating through Jenkins builds and then deliberately adding code that fails, you can test the alertness and response time of the team.

  Warning: Before adding defects, make sure that the team has agreed to the process or you might be getting angry e-mails late in the night.

- **Increasing code coverage with link crawlers and security scanners**: A fuzzer discovers the inputs of the application it is attacking, and then fires off an unexpected input. Not only is this good for security testing, but also boundary testing. If your server returns an unexpected error, then use this to trigger a more thorough review. Fuzzers and link crawlers are a cheap way to increase the code coverage of your tests.

  See the *Finding 500 errors and XSS attacks in Jenkins through Fuzzing* recipe in *Chapter 2*, *Enhancing Security*.

  You can cover more testing surface if you use a data-driven testing approach. For example, when writing JMeter test plans, you can use the CSV configuration element to read in variables from textfiles. This allows JMeter to pull out parameters, such as hostname and loop, through a series of hostnames. This enables one test plan to attack many servers.

  See the *Creating JMeter test plans* recipe in *Chapter 6*, *Testing Remotely*.

Final comments

The combination of Jenkins with aggressive automated testing acts as a solid safety net around coding projects. The recipes in this book support best practices.

Producing quality requires a great attention to details. Jenkins can pay attention to many of the details and shout loudly when violations occur.

Each project is different, and there are many ways to structure the workflow. Luckily, with over 400 plugins, Jenkins is flexible enough to adapt to even the most obscure infrastructures.

If you do not have the exact plugin that you want, then it is straightforward for a Java programmer to adapt or create their own.



Without a thriving open source Jenkins community, none of this would be possible. Jenkins is yet another positive example of the open source mentality working in practice. Well done to you, the Jenkins community.

Index

Symbols

@AfterSuite 242
@BeforeSuite 242
<classpath> tag 95
<executions> tag 113
<html> tag 110
***NIX installation package 48**
@Override annotation 287
<reporting> tag 97
<source> tag 94

A

AbstractItem 42
Abstract Syntax Tree (AST)
URL 192
Access Control Lists. (ACLs) 63
ACLs 63
Active Directory 81
adaptive site generation, Jenkins 129-132
adb 168
alternative code metrics
plotting, in Jenkins 88-91
Analysis Collector plugin
about 88
URL 175, 210
Android 1.6
and Hudson apps 168
Android-x86 project
and VirtualBox 168, 169
Ant 85
AntBuilder
about 86
running, through Groovy in Maven 102-106
Ant-contrib library 102
AntiSamy library 70
Ant script 246
Apache Archiva
URL 23
Apache AXIS framework
URL 255
applications retrieve information, from LDAP
anonymously 76
application-specific admin account 76
self-bind 76
archiving, Jenkins
need for 42
Artifactory
URL 23
Audit Logs
missing 60
Audit Trail plugin
about 58
installing 59
URL 59
working with 58, 59
Automatically Keep feature 273
automatic testing, of Jenkins
w3af, used 48, 49
Avatar plugin
installing 145
used, for generating home page 145, 146
working 147

B

backup, Jenkins
about 14
performing 14, 15
working 16
BadBoy
about 248
URL 248

bad credentials 76
banner
adding, to job description 283, 285
Behavior Driven Development (BDD) 174
Blamer
URL 167
blogging project 214
BlueStacks 169
bugs
finding, FindBugs used 193-195
build descriptions
information, exposing through 121-123

C

CAS 46
Cas1 plugin
about 83
working 84
CAS server. *See* **also Yale CAS**
installing 77-80
working 80, 81
CAS SSO server 61
Central Authentication Server. *See* **CAS**
cheat sheets 47
Checkstyle
about 175, 207
installing 207
URL 207, 210
checkstyle results
faking 210, 212
claims plugin 269, 270
CloudBees
about 225
URL 225
Cobertura code coverage plugin
about 179
installing 179
URL 179
used, for finding code 179-182
working 182
COCOMO model 178
code
finding, Cobertura code coverage plugin used 179-182
command, Jenkins Ubuntu workspace 143
Command Line Interface (CLI) 18
common log patterns
Failure to start up custom integration services 26
MD5 check sums 26
Common Name (CN) 113
community bug reports
adding 9
complementary plugin 60
configuring
Jetty, for integration tests 111-113
consistency breeds reliability 225
Continuous Integration (CI) servers 9
Cron Column plugin 300
Cross Site Request Forgery
about 56
URL 56
css 144
CSS 3 139
CSS 3 cheat sheet 139
CSV 91
custom data
plotting, plot plugin used 88-91
custom group script
used, for reviewing Project-based Matrix tactics 67, 68
working 69
custom ListView plugin
creating 301-303
working 304, 305
custom PMD rules
creating 188-192
working 192
custom security flaw 70
custom sounds
sending, HTML5 browsers used 155, 156
Cygpath plugin
about 225
URL 225
Cygwin
URL 224

D

Dashboard view plugin
about 153
installing 154
used, for saving screen space 153, 154

working 155
data
exporting, Job exporter plugin used 291, 292
DBfit
URL 233
Debian OS image 10
Denial Of Service attack 202
deploy plugin
installing 219
used, for deploying war file 219
Description Column plugin
URL 298
Description Setter plugin
about 122
installing 121
DescriptorImpl class 287
designer tool 192
dictionary attacks 58
disc monitoring
strengthening 27
disc usage
reporting 21, 22
disc usage plugin
about 21
installing 21
working 22
disc usage violations
warning, through log parsing 27, 28
Distinguished name (dn) 62, 113
DKMS 13
Don't Repeat Yourself (DRY) principle 187, 188, 310
DropDown toolbar plugin 151
DropDown ViewsTabBar plugin
installing 151
using 151, 152
working 152, 153
Dynamic Kernel Module Support (DKMS) 13

E

Eclipse 110
Eclipse templates
for JSP pages 110
emma 183
Emma Coverage plugin 300
Entitybroker
about 255
URL 255
Envfile plugin 97
Envinject plugin
about 97
environmental variables, manipulating 99-101
installing 98, 234
URL 234
environmental variables
manipulating 97-101
EPIC Perl 17
EPIC plugin
about 17
URL 17
Escaped Markup plugin
installing 53
URL 53
working 55
ESUP CAS
used, for installing CAS server 82
ESUP consortium 82
ESUP package 82
events
triggering on startup, startup trigger plugin used 293, 294
triggering when web content changes, URL trigger plugin used 295
evil URL 51
evolution charts 32
exclude patterns
testing 17
exec module 52
executable 144
Extra Columns plugin 300
extra FindBugs rules
enabling 197, 198
working 199
eXtreme Feedback plugin
about 158
adding 158
installing 158
modifying 159
working 159

F

fail method 95
Failsafe integration tests
triggering, Selenium Webdriver used 240-243
working 242, 243
failsafe plugin
about 242
URL 242
using 242
favicon.ico 144
favorites plugin
about 267
installing 267, 268
working 269
file system scm plugin
about 88, 280
installing 280
setting up 88
uploading 281
working 282
FindBugs
about 88, 174
used, for finding bugs 193, 194
used, for finding security defects 199-201
working 196
FindBugs Eclipse Plugin
about 196
installing 196
URL 196
Firefox add-on
used, for pulling Jenkins RSS feeds 30, 31
Fitnesse
about 218, 226
running remotely 226, 227
URL 226
working 229
Fitnesse HtmlUnit fixtures
activating 230, 232
working 232
Fitnesse jar
downloading 227
Fitnesse plugins
downloading 227
fixtures 218, 230
functional testing
Jmeter assertions, using 249-252
working 252
fuzzer
about 50
used, for finding server-side errors and XSS attacks 50, 51
working 52

G

Geb
URL 243
Generally Available (GA) 273
generated data
Postbuild Groovy plugin, used for reacting to 124-126
generate-resources phase 113
generic 81
getDisplayName() method 287
getShortDescription() method 295
gmaven plugin 93
Google Analytics plugin
about 169
installing 170
used, for tracking 170, 171
warning 170
working 171
Google Calendar plugin
about 160
installing 160
working 162, 163
green balls plugin
about 267
installing 267, 268
working 269
Groovy
about 85
AntBuilder, running through 102-106
Groovy plugin 97
Groovy scripts
about 37, 56
running, through Maven 93, 94
source, locating 96
tips 95
used, for looking at Jenkin user 56
working 57
group 63
group2.pl 69

group think
avoiding 309
grp_proj_tester 69
grp_username 69
guest additions 13
GUID 255
GUI Samples plugin
about 277
running 277, 278
working 278, 279

H

hard-learned observations, quality improvement
final comments 314
group think, avoiding 309
history, learning from 311
key points 313
offsetting work, to Jenkins nodes 311
QA/ integration servers, starving 313
test automation, as software project 310
test frameworks, emerging 312
hash 74
help 144
history
learning from 311
hits file 91
home page
generating, Avatar plugin used 145-147
Host server 9
HTML5 browsers
used, for sending custom sounds 155
HtmlFixture-2.5.1
downloading 230
HTML publisher plugin
installing 148
used, for creating HTML reports 148-150
HTML reports
about 182
creating, with HTML publisher plugin 148-150
HtmlUnit fixture. *See* **Fitnesse HtmlUnit fixture**
HtmlUnit framework 218, 310
HTML validity
verifying 203
Hudson2Go Lite
URL 167
Hudson apps
and Android 1.6 168
Hudson Helper
URL 167
Hudson Mobi
URL 167
HudsonStartupCause 295

I

images 144
import statement 95
information
exposing, through build descriptions 121-123
information radiators 158, 160
installing
Description Setter plugin 121
Envinject plugin 98
plot plugin 89
integration tests
Jetty, configuring for 111-113

J

JAAS 81
Jasig Wiki 82
JavaDocs 206
JavaMelody
about 32
installing 32
memory, troubleshooting 34
troubleshooting 35
URL 32
used, for monitoring 34
working 34
JavaNCSS
about 175, 205
reporting 205
URL 205
JavaScript library frameworks 140
Java Server Pages. *See* **JSP pages**
JDBC 82
Jdepend
about 207
URL 207

Jelly
about 266
URL 46
Jelly framework
URL 19
Jelly tags 276
Jenkins
about 8, 85
adaptive site generation 129-132
alternative code metrics, plotting 88-91
archiving, need for 42-44
backing up 13-15
built-in SSH daemon 225
configuring, for Google Calendar 160, 161
Groovy scripts, running through Maven 93, 94
home page, generating 145, 147
integrating, with Sonar 213
JavaMelody, installing 32
Jetty, configuring for integration tests 111, 113
Job exporter plugin 291
Jobs 8
jobs, global modifications with Groovy 40-42
long-term support release 9
maintaining 8
mobile apps 166
overall disc usage, reporting 21-23
overview 86, 87
personalizing 267, 268
provisioning, WAR overlay used 140, 141
restoring 15
sacrificial instance 9
Scriptler plugin, installing 36
scripts, managing using Scriptler plugin 36, 37
server-side errors and XSS attacks, finding using fuzzer 50, 51
skinning, Simple Theme plugin used 137
skinning, WAR overlay used 140, 141
SSO, enabling 83
testing remotely 217
tracking, Google Analytics plugin used 170, 171
tweeting 163-165
Jenkins API
jobs, triggering through 126-128
Jenkins command line interface
about 37
scripting 37, 39
working 39
Jenkins configuration
JavaDoc, finding for custom plugin extensions 20
modifying, from command line 18, 19
rubbish configuration 20
security, turning off 19
Jenkins FindBugs plugin
installing 193
URL 193
Jenkins JavaNCSS plugin
installing 205
URL 205
Jenkins jobs
anti-patterns 120
failing, based on JSP syntax errors 107, 109
multiple approaches 120
JenkinsMobi
URL 167
JenkinsMobi application 168
Jenkins Mobile Monitor
URL 167
Jenkins Mood widget
URL 167
Jenkins nodes
offsetting work to 311
Jenkins performance plugin
installing 246
Jenkins plugins
about 174
Analysis Collector Plugin 210
Audit Trail plugin 59
Avatar plugin 145
checkstyle 207
Cobertura code coverage 179
custom ListView plugin, creating 301
Dashboard view plugin 154
deploy plugin 219
DropDown ViewsTabBar plugin 151
Escaped Markup plugin 53
eXtreme Feedback plugin 158
favorites plugin 267
file system scm plugin 280

FindBugs 193
Google Analytics plugin 170
Google Calendar plugin 160
green balls plugin 267
GUI Samples plugin 277
HTML publisher plugin 148
JavaNCSS 205
JobConfigHistory plugin 60
JS games plugin 274
LDAP Email plugin 61
ListView plugins, reviewing 297
markup plugin 283
Mask Passwords plugin 53
multi slave config plugin 222
plot plugin 203
PMD plugin 184
promoted builds plugin 270
RootAction plugin 288
sloccount 176
Sonar plugin 213
Sounds plugin 155
startup trigger plugin 293
themes plugin 137
Twitter plugin 164
Unicon Validation plugin 203
URL trigger plugin 295
Jenkins PMD plugin
installing 184
URL 188
working 186
Jenkins RSS feeds
pulling, Firefox add-on used 30, 31
Jenkins server deployment 143
Jenkins-Sonar integration
about 213
results, aggregating 215
Jenkins Sounds plugin
installing 155
working 157
Jenkins Ubuntu workspace
command 143
fingerprints 143
jobs 143
plugins 143
tools 143
updates 143
userContent 144
users 144
Jenkins user
looking, through Groovy scripts 56, 57
Jenkins Violations plugin 215
Jenkins xUnit plugin
installing 259
Jetty
configuring, for integration tests 111-113
JLC 206
Jlint
URL 187
Jmeter
about 88, 218, 244, 252
samplers 244
URL 244
Jmeter assertions
used, for functional testing 249-252
Jmeter performance metrics
reporting 246, 247
working 248
Jmeter test plans
creating 244, 245
working 245, 246
JobConfigHistory plugin
about 60
URL 60
Job exporter plugin
about 291
installing 291
working 293
job.getLastSuccessfulBuild() 44
jobs
generating, remotely 129
running, from within Maven 128
triggering, through Jenkins API 126-128
jobs, global modifications
Groovy used 40, 41
Jobs Grid portlet 155
Jquery plugin 140
JS games plugin
installing 274, 275
pinning 274
working 276
JSP pages
about 107
Eclipse templates 110

JSP syntax errors
Jenkins jobs, failing 107-109
Junit
about 88, 240
URL 240
Junit results schema
URL 213
JXplorer
about 71
URL 71
working 73

K

Keep It Simple Stupid (KISS) principle 87, 310
keytool 78
keytool plugin 113
Koders
about 133
URL 133
Koders.com 179

L

Last Failure Version Column plugin
URL 298
lastsuccessversioncolumn plugin 44
Lava Lamps
URL 160
LDAP 46, 61, 81
LDAP administration
about 70
working 73
LDAP Data Interchange Format. *See* **LDIF**
LDAP Email plugin 61
LDAP plugin
configuring 74
working 75
LDAP SSL 82
LDIF 61, 62
Learning Management System (LMS) 119
legacy 82
libnet-ldap-perl package 64
license violations
reviwing, from within Maven 117-119
viewing, with Rats 114-116
Lightweight Directory Access Protocol. *See* **LDAP**
List View Columns plugin
URL 298
ListView plugins
Cron Column plugin 300
Emma Coverage plugin 300
Extra Columns plugin 300
Last Failure Version Column plugin 298
Last Success Description Column plugin 298
List View Columns plugin 298
Progress Bar plugin 300
reviewing 297-299
Success Version Column plugin 298
login2.pl 69
log parsing plugin
configuring 24, 25
installing 24
working 26
log_rules directory 24
long-term support release, Jenkins 9

M

main source code, JS games plugin
src/main/java 276
src/main/resources 276
src/main/webapp 276
src/test 276
maintenance, Jenkins
disks overflowing with artifacts 8
general lack of consistency 8
new plugins causing exceptions 8
resource depletion 8
script spaghetti 8
markup plugin
about 283
installing 283-286
used, for adding banner to job descriptions 283
working 286
Mask Passwords plugin
installing 53
URL 53
working 55
Master instance 32

Maven
about 85
AntBuilder, running through Groovy 102-106
Groovy scripts, running through 93, 94
jobs, running from within 128
license violations, reviewing from within 117-119
phases 96
URL, for plugin compatibility list 97
URL, for version differences 97
Maven 2
about 133
versus Maven 3 133
Maven 2.2.1 114
Maven 3
about 114, 133
versus Maven 2 133
Maven dashboard
about 197
URL 197
Maven plugin 182
Maven PMD plugin 186
Maven repository 23
maven-soapui plugin
URL 262
Maven WAR plugin 136
META-INF 144
misconfiguration and bad credentials
differences 76
misconfigured DN 76
misses file 91
MIT license 306
MLCLC 206
mobile apps
about 166
Blamer 167
Hudson2Go Lite 167
Hudson Helper 167
Hudson Mobi 167
JenkinsMobi 167
Jenkins Mobile Monitor 167
Jenkins Mood widget 167
working 167
monitoring 34
multiple Jenkins nodes
creating 222-225
Multiple SCM plugin 120
multi slave config plugin
installing 222
working 224
Multi slave config plugin
using 222
MyTest.xhtml 238

N

NCSS 88, 206
Nexus
URL 23
nginx
about 229
URL 229
nightly build 219
Nikto
about 50
URL 50
nodes 32
nonce feature 56
non-persistent attack 263

O

ObjectClasses 62, 63
Ohloh
about 178
URL 178
oob 165
OpenLDAP
administering 70-72
installing, with test user and group 61-63
Open Web Application Security Project. *See* **OWASP**
OWASP
about 47
URL 47
OWASP Store front 47
OWASP_TOP10 profile 49
OWASP top-ten list of insecurities
A2-Cross Site Scripting (XSS) 47
A6-Security Misconfiguration 47
A7-Insecure Cryptographic Storage 47
A9-Insufficient Transport Layer Protection 47

P

PAM_LDAP
URL 66
Parameterized build 101
penetration tests 48
Perl 91
Perl script
about 29, 91
using 27
permission errors
checking 16
permissions, Project-based Matrix strategy
globally 69
per project 69
Ping service 263
PingTest 255
Piwik
URL 171
platform encoding warning 95
plot plugin
about 88
installing 89, 203
URL 203
used, for plotting custom data 88-91
working 92
Pluggable Authentication Modules (PAM) 66
plugins
about 14
key points 266
PMD 174
PMD rulesets
activating 183-185
basic 183
Don't Repeat Yourself (DRY) principle 187, 188
imports 183
throttling down 187
unusedcoded 183
URL 187
pom.xml 87, 88, 238
port 1023 113
port 8082 113
port 9443 113
Portlets dashboard plugin
URL 175
POSIX account administration 61
Post build Groovy Plugin
about 123
used, for reacting to generated data 124-126
post-integration-test phase, Maven 113
pre-integration-test phase, Maven 113
profile2 tool 209
programmer's cafés 214
Progress Bar plugin 300
Project-based Matrix strategy
about 67
permissions 69
Project-based Matrix tactics
reviewing, custom group script used 67-69
project value
estimating, sloccount used 176, 178
promoted builds plugin
installing 270, 271
working 272
promotion plugin
about 270
using 270
Publish Over SSH Plugin 86
Python programming language 51

Q

QA/ integration servers
starving 313
QALab
URL 197
QJPro
URL 187
QualityA 309

R

RADIUS 81
Rapid Application Development (RAD) 106
Rats
used, for looking at license violations 114-116
real-time reporting feature 171
regex expressions 121, 123
remote testing, through Jenkins 217
replay attacks 56
reporting, with JavaNCSS 205, 206
repository managers
advantages 23

Apache Archiva 23
Artifactory 23
Nexus 23
resources, CAS server
URLs 82
RestFixture
URL 233
restore, Jenkins
performing 15
working 16
robots.txt 144
Roles Validation script 84
RootAction plugin
creating 288, 289
working 290
root admin 69
Roster tool 24
rulesets 183

S

sacrificial instance
about 9
setting up 9
VirtualBox, downloading 10
VirtualBox, installing 10-12
sacrificial Jenkins instance
advantages 9
using 9
Sakai
about 119
URL 26
Sakai CLE 253, 255
Sakai Foundation 119
Sakai Learning Management System 209
Sakai Open Academic Environment (OAE) 256
Sakai package 254
Sakai Web services
enabling 253, 254
working 254
samplers 244
scraping 126
screen space
saving, Dashboard view plugin used 153-155
Scriptler plugin
about 36
installing 36
used, for managing scripts 36
working 37
Script Realm authentication
used, for provisioning 64, 65
Script Realm plugin
about 64
installing 64
URL 64
working 66
scripts 144
search engines and robots.txt 144
security
improving, via small configuration changes 53-55
security defects
finding, FindBugs used 199-201
Selenium 218
Selenium Grid
URL 218, 234
Selenium HTML report plugin
installing 234
URL 234
Selenium IDE 234
Selenium IDE tests
running 234-237
working 238
selenium-maven-plugin
URL 238
Selenium Remote Control (RC) 218
Selenium Webdriver
about 218
used, for triggering Failsafe integration tests 240-242
Self binding 66
server-side errors
finding, fuzzer used 51
server types 110
Setenv plugin 97
setter plugin 121
Simple Theme plugin
about 137
used, for skinning Jenkins 137
working 138
Single Sign On (SSO) 46
size_summary method 29
Skipfish
about 50

URL 50
slapd 61
SLCLC 206
SlideME
URL 168
sloccount plugin 97
about 176
installing 176
URL 176
used, for estimating project value 176
working 177
snapshots 120
SoapUI
about 218, 256, 259
installing 256
URL 256
used, for writing SoapUI 256, 257
working 258
SoapUI test results
exportwAll 262
JunitReport 262
printReport 262
reporting 259-262
software cost estimation 178
Sonar
about 88, 175, 213
installing 213
Jenkins, integrating with 213
URL 213
Sonar plugins 215
SourceForge 48
URL 197
SPNEGO 82
SSO
enabling, in Jenkins 83
Stapler 153, 266, 276
start method 294
Startup 279
startup trigger plugin
about 293
installing 293
working 294
static code review 70
style
checking, external pom.xml used 207, 208
Success Version Column plugin
URL 298
Swatch 60
systemutils class 95

T

tainted 70
task scanner plugin 97
testApp unit test 183
test automation
considering, as software project 310
Test Driven Development (TDD) 174
test frameworks
cloud 312
code searching 312
emerging 312
Sonar measures project quality 312
static code review tools are improving 312
TestNG
about 240
URL 240
TestNG unit tests 218
test phase 105
test plans
writing, SoapUI used 256, 257
TestSuite.xhtml 238
themes plugin
installing 137
used, for modifying Jenkins look 137, 138
thinBackup plugin
about 14
installing 14
URL 14
Token Macro plugin 123
Tomcat 7
installing 78
Traffic lights
URL 160
translate method 287
troubleshooting
JavaMelody 34, 35
trusted 81
Twitter Java framework
URL 165
Twitter OAuth API
URL 165
Twitter plugin
installing 164

working 165

U

UberSocial
URL 166
Ubuntu
about 9, 61
installing 143
Ubuntu virtual image
URL 10
Unicon Validation plugin
installing 203
URL 203
working 204
unified validator 203
unit tests 240
untaint 70
URL trigger plugin
about 295
working 296
USB missile launcher
URL 160
user 63
utility method 123

V

verify phase 105
violations plugin
URL 175
VirtualBox
and Android-x86 project 168, 169
downloading 10
installing 10-12
URL 10
working 12
virtual images
sources 13

W

w3af 47
used, for automatic testing of Jenkins 48, 49
warning 48
working 49
w3schools
about 192
URL 192
Wapiti
about 50
URL 50
war file
deploying, from Jenkins to Tomcat 219, 221
warning, w3af 48
WAR overlay
about 140
used, for provisioning Jenkins 140-142
used, for skinning Jenkins 140-142
working 142
web applications deployment, for Integration tests
approaches 219
webapp tag 114
Webgoat
about 49
URL 49
WEB-INF 144
Web Service Definition Language (WSDL) files 218
Web services. *See* **Sakai Web services**
Webtestfixtures
URL 233
wget tool 128
WIKI pages
modifying 9
Windows 7 Android emulator
URL 169
winstone.jar 144
workspace plugin 130
WSDL
URL 259

X

x86 image
installing 168
X.509 Certificates 82
XML 91
XML report 182
xPlanner
URL 230

Xradar
URL 197
XSS attacks
about 50
finding, fuzzer used 51
URL 201
X-SSH-Endpoint 225
Xstream
about 266
URL 19, 46
xUnit plugin. *See* **also Jenkins xUnit plugin**
URL 259
Xunit plugin
URL 213
Xvfb
URL 234

Y

Yale CAS. *See* **also CAS server**
about 77
advantages 77
backend authentication 81
downloading 78
installing, ESUP CAS used 82
LDAP SSL 82
resources 82
URL 77
YUI library 140



Thank you for buying
Jenkins Continuous Integration Cookbook

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

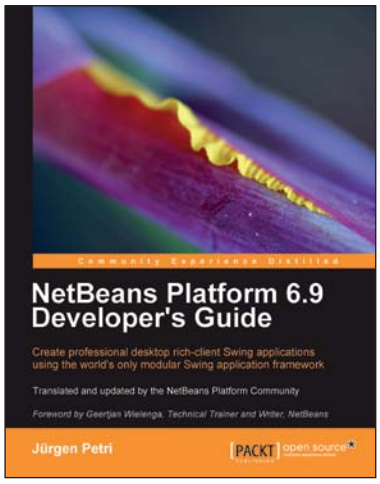
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING



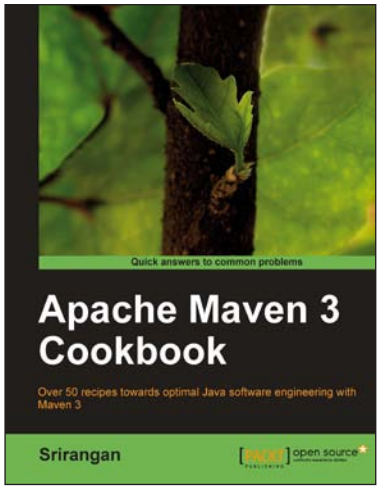
NetBeans Platform 6.9 Developer's Guide

ISBN: 978-1-84951-176-6 Paperback: 288 pages

Create professional desktop rich-client Swing applications using the world's only modular Swing application framework

1. Create large, scalable, modular Swing applications from scratch
2. Master a broad range of topics essential to have in your desktop application development toolkit, right from conceptualization to distribution
3. Pursue an easy-to-follow sequential and tutorial approach that builds to a complete Swing application

---



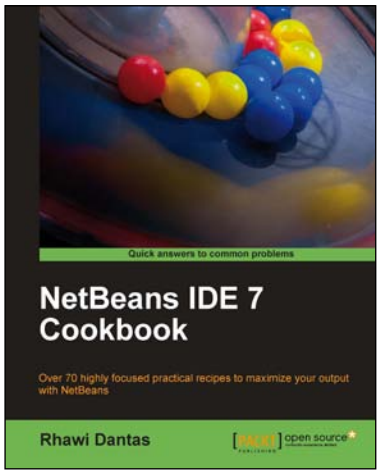
Apache Maven 3 Cookbook

ISBN: 978-1-84951-244-2 Paperback: 224 pages

Over 50 recipes towards optimal Java software engineering with Maven 3

1. Grasp the fundamentals and extend Apache Maven 3 to meet your needs
2. Implement engineering practices in your application development process with Apache Maven
3. Collaboration techniques for Agile teams with Apache Maven
4. Use Apache Maven with Java, Enterprise Frameworks, and various other cutting-edge technologies

[PACKT] open source
community experience distilled
PUBLISHING



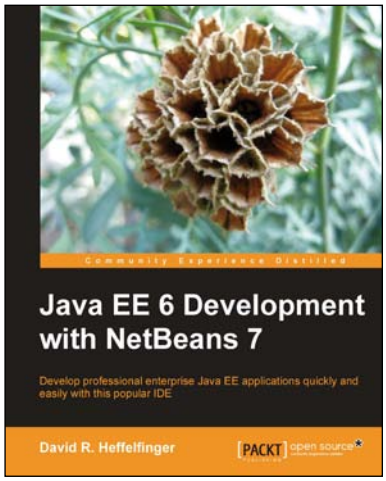
NetBeans IDE 7 Cookbook

ISBN: 978-1-84951-250-3 Paperback: 308 pages

Over 70 highly focused practical recipes to maximize your output with NetBeans

1. Covers the full spectrum of features offered by the NetBeans IDE
2. Discover ready-to-implement solutions for developing desktop and web applications
3. Learn how to deploy, debug, and test your software using NetBeans IDE
4. Another title in Packt's Cookbook series giving clear, real-world solutions to common practical problems

---



Java EE 6 Development with NetBeans 7

ISBN: 978-1-84951-270-1 Paperback: 392 pages

Develop professional enterprise Java EE applications quickly and easily with this popular IDE

1. Use features of the popular NetBeans IDE to accelerate development of Java EE applications
2. Develop JavaServer Pages (JSPs) to display both static and dynamic content in a web browser
3. Covers the latest versions of major Java EE APIs such as JSF 2.0, EJB 3.1, and JPA 2.0, and new additions to Java EE such as CDI and JAX-RS
4. Learn development with the popular PrimeFaces JSF 2.0 component library

Please check **www.PacktPub.com** for information on our titles