

Quick answers to common problems

Jenkins Continuous Integration Cookbook

Second Edition

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

Alan Mark Berg

[PACKT] open source*
PUBLISHING

Table of Contents

Jenkins Continuous Integration Cookbook Second Edition
Credits
About the Author
Acknowledgments
About the Reviewers
www.PacktPub.com
    Support files, eBooks, discount offers, and more
        Why Subscribe?
        Free Access for Packt account holders
Preface
    What this book covers
    What you need for this book
    Who this book is for
    Sections
        Getting ready
        How to do it…
        How it works…
        There's more…
        See also
    Conventions
    Reader feedback
    Customer support
        Downloading the example code
        Errata
        Piracy
        Questions
1. Maintaining Jenkins
    Introduction
    Using a test Jenkins instance
        Getting ready
        How to do it...
        How it works...
        See also
    Backing up and restoring
        Getting ready
        How to do it...
        How it works...
        There's more...
            Checking for permission errors
            Testing exclude patterns
        See also
    Modifying the Jenkins configuration from the command line
        Getting ready
        How to do it...

- How it works...
- There's more...
  - Turning off security
  - Finding JavaDoc for custom plugin extensions
  - The effects of adding garbage
- See also

Installing Nginx
- Getting ready
- How to do it...
- How it works...
- There's more...
  - Naming logfiles
  - Backing up configuration
- See also

Configuring Nginx as a reverse proxy
- Getting ready
- How to do it...
- How it works...
- There's more...
  - Testing complex configuration
  - Offloading SSL
- See also

Reporting overall storage use
- Getting ready
- How to do it...
- How it works...
- There's more...
- See also

Deliberately failing builds through log parsing
- Getting ready
- How to do it...
- How it works...
- There's more...
- See also

Adding a job to warn of storage use violations through log parsing
- Getting ready
- How to do it...
- How it works...
- There's more...
- See also

Keeping in contact with Jenkins through Firefox
- Getting ready
- How to do it...
- How it works...
- There's more...
  - RSS credentials
  - Alternatives to Firefox
- See also

- Monitoring via JavaMelody
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
    - Troubleshooting with JavaMelody – memory
    - Troubleshooting with JavaMelody – painful jobs
  - See also
- Keeping track of script glue
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
  - See also
- Scripting the Jenkins CLI
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
  - See also
- Global modifications of jobs with Groovy
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
  - See also
- Signaling the need to archive
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
  - See also

2. Enhancing Security
- Introduction
- Testing for OWASP's top 10 security issues
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
    - Target practice with Webgoat
    - More tools of the trade
  - See also
- Finding 500 errors and XSS attacks in Jenkins through fuzzing
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
  - See also

- Improving security via small configuration changes
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
  - See also
- Avoiding sign-up bots with JCaptcha
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
    - Defense in depth
    - More information on bots
  - See also
- Looking at the Jenkins user through Groovy
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
  - See also
- Working with the Audit Trail plugin
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
    - A complementary plugin – JobConfigHistory
    - Missing audit logs
    - Swatch
  - See also
- Installing OpenLDAP
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
  - See also
- Using Script Realm authentication for provisioning
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
  - See also
- Reviewing project-based matrix tactics via a custom group script
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
    - My own custom security flaw
    - Static code review, tainting, and untainting

See also
Administering OpenLDAP
Getting ready
How to do it...
How it works...
There's more...
See also
Configuring the LDAP plugin
Getting ready
How to do it...
How it works...
There's more...
The difference between misconfiguration and bad credentials
Searching
See also
Installing a CAS server
Getting ready
How to do it...
How it works...
There's more...
Backend authentication
An alternative installation recipe using ESUP CAS
Trusting LDAP SSL
A few useful resources
See also
Enabling SSO in Jenkins
Getting ready
How to do it...
How it works...
See also
Exploring the OWASP Dependency-Check plugin
Getting ready
How to do it...
How it works...
There's more...
See also
3. Building Software
Introduction
Jenkins is an enterprise-friendly agnostic
A pom.xml template
Maven changes
Setting up a File System SCM
Plotting alternative code metrics in Jenkins
Getting ready
How to do it...
How it works...
There's more...
See also

- Running Groovy scripts through Maven
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
    - Keeping track of warnings
    - Where's my source?
    - Maven phases
  - See also
- Manipulating environmental variables
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
  - See also
- Running Ant through Groovy in Maven
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
  - See also
- Failing Jenkins jobs based on JSP syntax errors
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
    - Different server types
    - Eclipse templates for JSP pages
  - See also
- Configuring Jetty for integration tests
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
  - See also
- Looking at license violations with Rat
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
  - See also
- Reviewing license violations from within Maven
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
    - Multiple approaches and antipatterns
    - Snapshots

- See also
- Exposing information through build descriptions
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
  - See also
- Reacting to generated data with the groovy-postbuild plugin
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
  - See also
- Remotely triggering jobs through the Jenkins API
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
    - Running jobs from within Maven
    - Remotely generating jobs
  - See also
- Adaptive site generation
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
    - Searching for example site generation configurations
    - Maven 2 and 3 pitfalls
  - See also

4. Communicating Through Jenkins
- Introduction
- Skinning Jenkins with the simple themes plugin
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
    - CSS 3
    - Included JavaScript library frameworks
    - Trust but verify
  - See also
- Skinning and provisioning Jenkins using a WAR overlay
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
    - Which types of content can you replace?
    - Search engines and robots.txt
  - See also

Generating a home page
- Getting ready
- How to do it...
- How it works...
- There's more...
- See also

Creating HTML reports
- Getting ready
- How to do it...
- How it works...
- There's more...
- See also

Efficient use of views
- Getting ready
- How to do it...
- How it works...
- There's more...
- See also

Saving screen space with the Dashboard View plugin
- Getting ready
- How to do it...
- How it works...
- There's more...
- See also

Making noise with HTML5 browsers
- Getting ready
- How to do it...
- How it works...
- There's more...
- See also

An extreme view for reception areas
- Getting ready
- How to do it...
- How it works...
- There's more...
- See also

Mobile presentation using Google Calendar
- Getting ready
- How to do it...
- How it works...
- There's more...
- See also

Mobile apps for Android and iOS
- Getting ready
- How to do it...
- How it works...
- There's more...
  - Android 1.6 and Hudson apps

Virtualbox and the Android x86 project
See also
Knowing your audience with Google Analytics
Getting ready
How to do it...
How it works...
There's more...
See also
Simplifying powerful visualizations using the R plugin
Getting ready
How to do it...
How it works...
There's more...
RStudio or StatET
Quickly getting help
For more information
See also
5. Using Metrics to Improve Quality
Introduction
Estimating the value of your project through sloccount
Getting ready
How to do it...
How it works...
There's more...
Software cost estimation
Goodbye Google code search; hello code.ohoh.net
See also
Looking for "smelly" code through code coverage
Getting ready
How to do it...
How it works...
There's more...
Activating more PMD rulesets
Getting ready
How to do it...
How it works...
There's more...
Throttling down PMD rulesets
The Don't Repeat Yourself principle
See also
Creating custom PMD rules
Getting ready
How to do it...
How it works...
There's more...
See also
Finding bugs with FindBugs
Getting ready

- How to do it...
- How it works...
- There's more...
  - The FindBugs Eclipse plugin
  - The Xradar and Maven dashboards
- See also

Enabling extra FindBug rules
- Getting ready
- How to do it...
- How it works...
- There's more...
- See also

Finding security defects with FindBugs
- Getting ready
- How to do it...
- How it works...
- There's more...
- See also

Verifying HTML validity
- Getting ready
- How to do it...
- How it works...
- There's more...

Reporting with JavaNCSS
- Getting ready
- How to do it...
- How it works...
- There's more...

Checking code style using an external pom.xml file
- Getting ready
- How to do it...
- How it works...
- There's more...
- See also

Faking Checkstyle results
- Getting ready
- How to do it...
- How it works...
- There's more...
- See also

Integrating Jenkins with SonarQube
- Getting ready
- How to do it...
- How it works...
- There's more...
  - SonarQube plugins
  - Alternative aggregator – the violations plugin
- See also

- Analyzing project data with the R plugin
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
  - See also
- 6. Testing Remotely
  - Introduction
  - Deploying a WAR file from Jenkins to Tomcat
    - Getting ready
    - How to do it...
    - How it works...
    - There's more...
    - See also
  - Creating multiple Jenkins nodes
    - Getting ready
    - How to do it...
    - How it works...
    - There's more...
    - See also
  - Custom setup scripts for slave nodes
    - Getting ready
    - How to do it...
    - How it works...
    - There's more...
    - See also
  - Testing with FitNesse
    - Getting ready
    - How to do it...
    - How it works...
    - There's more...
    - See also
  - Activating FitNesse HtmlUnit fixtures
    - Getting ready
    - How to do it...
    - How it works...
    - There's more...
    - See also
  - Running Selenium IDE tests
    - Getting ready
    - How to do it...
    - How it works...
    - There's more...
    - See also
  - Triggering failsafe integration tests with Selenium WebDriver
    - Getting ready
    - How to do it...
    - How it works...

There's more...
See also
Creating JMeter test plans
Getting ready
How to do it...
How it works...
There's more...
See also
Reporting JMeter performance metrics
Getting ready
How to do it...
How it works...
There's more...
See also
Functional testing using JMeter assertions
Getting ready
How to do it...
How it works...
There's more...
See also
Enabling Sakai web services
Getting ready
How to do it...
How it works...
There's more...
See also
Writing test plans with SoapUI
Getting ready
How to do it...
How it works...
There's more...
See also
Reporting SoapUI test results
Getting ready
How to do it...
How it works...
There's more...
See also
7. Exploring Plugins
Introduction
Personalizing Jenkins
Getting ready
How to do it...
How it works...
There's more...
See also
Testing and then promoting builds
Getting ready

- How to do it...
- How it works...
- There's more...
- See also

Fun with pinning JSGames
- Getting ready
- How to do it...
- How it works...
- There's more...
- See also

Looking at the GUI samples plugin
- Getting ready
- How to do it...
- How it works...
- There's more...
- See also

Changing the help of the FileSystem SCM plugin
- Getting ready
- How to do it...
- How it works...
- There's more...
- See also

Adding a banner to job descriptions
- Getting ready
- How to do it...
- How it works...
- There's more...
- See also

Creating a RootAction plugin
- Getting ready
- How to do it...
- How it works...
- There's more...
- See also

Exporting data
- Getting ready
- How to do it...
- How it works...
- There's more...
- See also

Triggering events on startup
- Getting ready
- How to do it...
- How it works...
- See also

Groovy hook scripts and triggering events on startup
- Getting ready
- How to do it...

How it works...
There's more...
See also
Triggering events when web content changes
Getting ready
How to do it...
How it works...
There's more...
See also
Reviewing three ListView plugins
Getting ready
How to do it...
How it works...
There's more...
See also
Creating my first ListView plugin
Getting ready
How to do it...
How it works...
There's more...
See also
A. Processes that Improve Quality
Fail early
Data-driven testing
Learning from history
Considering test automation as a software project
Visualize, visualize, visualize
Conventions are good
Test frameworks and commercial choices are increasing
Offsetting work to Jenkins nodes
Starving QA/integration servers
Reading the change log of Jenkins
Avoiding human bottlenecks
Avoiding groupthink
Training and community
Visibly rewarding successful developers
Stability and code maintenance
Resources on quality assurance
And there's always more
Final comments
Index

Jenkins Continuous Integration Cookbook Second Edition

Jenkins Continuous Integration Cookbook Second Edition

Copyright © 2015 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

Second edition: January 2015

Production reference: 1240115

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78439-008-2

www.packtpub.com

Cover image by Hester van der Heijden
(<hestervanderheijden@gmail.com>)

Credits

Author

Alan Mark Berg

Reviewers

Dr. Alex Blewitt

Manuel Doninger

Takafumi Ikeda

Michael Peacock

Donald Simpson

Commissioning Editor

Amarabha Banerjee

Acquisition Editor

Neha Nagwekar

Content Development Editor

Samantha Gonsalves

Technical Editors

Manan Badani

Ankita Thakur

Copy Editors

Safis Editing

Ameesha Green

Project Coordinator

Sanchita Mandal

Proofreaders

Stephen Copestake

Ameesha Green

Indexer

Hemangini Bari

Production Coordinator

Manu Joseph

Cover Work

Manu Joseph

About the Author

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

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

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

Acknowledgments

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

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

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

About the Reviewers

Dr. Alex Blewitt has been working with developer tooling and toolchains for over 15 years. Having worked with Java since its early roots with VisualAge for Java through Eclipse and IntelliJ, developing plugins to optimize the developer experience and productivity has been his passion. Alex has been a strong advocate of Git for over 5 years and designed and rolled out a highly available distributed Git review system with integrated Jenkins builds to form a secure but highly performant software build chain.

Alex currently works in the finance industry in London, as well as has his own company, Bandlem Limited. Alex also writes for InfoQ, an online news magazine, and has authored two books *Swift Essentials* and *Eclipse 4 Plug-in Development by Example Beginner's Guide* by Packt Publishing. He blogs regularly at http://alblue.bandlem.com and `tweets @alblue`.

Manuel Doninger has been working as a Java developer since 2009. Since 2014, he has worked as a freelance developer and consultant based in Germany. His recent projects include enterprise applications on the Java EE platform and Spring applications. At his former employers, he managed the conversion from CVS/subversion to Git, and conducted and supported the installation of the build infrastructure with Jenkins, SonarQube, and other products for Continuous Integration and Continuous Delivery.

In his spare time, he occasionally contributes to open source projects, such as EGit, Spring Boot, and several Apache Maven plugins.

Takafumi Ikeda is the maintainer of Play Framework1 and now works as a senior engineer/release manager at a mobile Internet company in Japan. He is the author *Introduction to Building Continuous Delivery Pipeline* (http://www.amazon.co.jp/dp/4774164283/). He wrote *Appendix B*, *Plugin Development*, in *Jenkins definitive guide Japanese version* (http://www.oreilly.co.jp/books/9784873115344/). He has also reviewed *Instant Play Framework Starter*, *Packt Publishing* (https://www.packtpub.com/web-development/instant-play-framework-starter-instant).

He is also the representative of Japan Play Framework Users Group and has experience as a speaker of several technical matters at tech conference/seminars in Japan.

Takafumi's Twitter ID is `@ikeike443` (in Japanese) or `@Takafumi_Ikeda` (in English). His current employer is DeNA (http://dena.com/intl).

Michael Peacock is an experienced software developer and team lead from Newcastle, UK, with a degree in software engineering from Durham University.

After spending a number of years running his own web agency and subsequently working directly for a number of software start-ups, Michael now runs his own software development agency, working on a range of projects for an array of different clients.

He is the author of *Creating Development Environments with Vagrant*, *PHP 5 Social Networking*, *PHP 5 E-commerce Development*, *Drupal 7 Social Networking*, *Selling Online with Drupal e-Commerce*, and *Building Websites with TYPO3*, all by Packt Publishing. Other books Michael has been involved in include *Mobile Web Development*, *Jenkins Continuous Integration Cookbook*, and *Drupal for Education and E-Learning*, for which he acted as a technical reviewer.

Michael has also presented at a number of user groups and technical conferences, including PHP UK Conference, Dutch PHP Conference, ConFoo, PHPNE, PHPNW, and Cloud Connect Santa Clara.

You can follow Michael on Twitter (`@michaelpeacock`) or find out more about him through his website (www.michaelpeacock.co.uk).

Donald Simpson is an information technology consultant based in Scotland, UK. He specializes in helping organizations improve the quality and reduce the cost of software development through Software Build Automation. He has also designed and implemented Continuous Integration solutions for a broad range of companies and Agile projects. He can be reached at www.donaldsimpson.co.uk.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

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.



https://www2.packtpub.com/books/subscription/packtlib

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read 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 a 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 9 entirely free books. Simply use your login credentials for immediate access.

Preface

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

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

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

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

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

What this book covers

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

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

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

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

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

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

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

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

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 3 (http://maven.apache.org)
- Jenkins (http://jenkins-ci.org/)
- Java Version 1.8 (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 or Git repository
- OS of preference: Linux (Ubuntu)

> **Note**
>
> Note that from the Jenkins GUI (`http://localhost:8080/configure`), you can install different versions of Maven, Ant, and Java. You do not need to install these as part of the OS.

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 it directly from a WAR file. It is up to you to choose the approach that you feel is most comfortable. However, you can 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 `JENKINS_HOME` environment variable to the directory you wish Jenkins to run under. Next, run a command similar to the following command:

```
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 by executing 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 test 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 a familiarity with basic application server concepts are expected. A basic understanding of Jenkins is also assumed.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it, How it works, There's more, and See also).

To give clear instructions on how to complete a recipe, we use these sections as follows.

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it…

This section contains the steps required to follow the recipe.

How it works…

This section usually consists of a detailed explanation of what happened in the previous section.

There's more…

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The job-specific configuration is then stored in `config.xml` within the subdirectory."

A block of code is set as follows:

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

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

```
server {
  listen   80;
  server_name  localhost;
  access_log  /var/log/nginx/jenkins
_8080_proxypass_access.log;
  error_log
 /var/log/nginx/jenkins_8080_proxypass_access.log;
  location / {
    proxy_pass      http://127.0.0.1:7070/;
    include         /etc/nginx/proxy.conf;
  }
}
```

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

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

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Click on **Save**."

> ## Note
>
> Warnings or important notes appear in a box like this.

> ## Tip
>
> 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 disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail <feedback@packtpub.com>, and mention the book's title in 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 at 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 from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

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 could 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/submit-errata, 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.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted 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

If you have a problem with any aspect of this book, you can contact us at <questions@packtpub.com>, and we will do our best to address the problem.

Chapter 1. Maintaining Jenkins

In this chapter, we will cover the following recipes:

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

Introduction

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

- **New plugins causing exceptions**: There are a lot of good plugins being written with rapid version changes. In this situation, it is easy for you to accidentally add new versions of plugins with new defects. There have been a number of occasions during upgrades when suddenly the plugin does not work. To combat the risk of plugin exceptions, consider using a test Jenkins instance before releasing to a critical system.
- **Storage over-flowing with artifacts**: If you keep a build history that includes artifacts such as war files, large sets of JAR files, or other types of binaries and source code, then your storage space is consumed at a surprising rate. Storage costs have decreased tremendously, but storage usage equates to longer backup times and

more communication from slave to master. To minimize the risk of disk overflowing, you will need to consider your backup and restore policy and the associated build retention policy expressed in the advanced options of jobs.

- **Script spaghetti**: As jobs are written by various development teams, the location and style of the included scripts vary. This makes it difficult for you to keep track. Consider using well-defined locations for your scripts and a scripts repository managed through a plugin.
- **Resource depletion**: As memory is consumed or the number of intense jobs increases, then Jenkins slows down. Proper monitoring and quick reactions reduce impact.
- **A general lack of consistency between jobs due to organic growth**: Jenkins is easy to install and use. The ability to seamlessly turn on plugins is addictive. The pace of adoption of Jenkins within an organization can be breathtaking. Without a consistent policy, your teams will introduce lots of plugins and also lots of ways of performing the same work. Conventions improve consistency and readability of jobs and thus decrease maintenance.

> ## Note
>
> The recipes in this chapter are designed to address the risks mentioned. They represent only one set of approaches. If you have comments or improvements, feel free to contact me at <bergsmooth@gmail.com> or better still add tutorials to the Jenkins community wiki.

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

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

Using a test Jenkins instance

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

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

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

- **Saving state**: At any moment, you can save the state of the running virtual image and return to that running state later. This is excellent for short-term experiments that have a high risk of failure.
- **Ability to share images**: You can run your virtual image anywhere that a player can run. This may include your home desktop or a hardcore server.
- **Use a number of different operating systems**: This is good for node machines running integration tests or functional tests with multiple browser types.
- **Swap workspaces**: By having the workspace outside the virtual image on the host of the virtual server, you can test different version levels of OSes against one workspace. You can also test one version of Jenkins against different host workspaces with different plugin combinations.

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

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

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

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

Getting ready

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

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

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

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

> **Tip**
>
> **Security Considerations**

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

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. Set **Name** to `Jenkins_Ubuntu_11.04`. The OS type will be automatically updated. Click on the **Next** button.
3. Set **Memory** to **2048 MB** and then click on **Next**.

   Note that the host machine requires 1 GB more RAM than the total allocated to its guest images. In this example, your host machine requires 3 GB of RAM. For more details, visit http://www.oracle.com/us/technologies/virtualization/oraclevm/oracle-vm-virtualbox-ds-1655169.pdf.

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



5. Press the **Create** button.
6. Start the virtual image by clicking on the **Start** icon:



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

```
sudo passwd
```

9. Install the Jenkins repository as explained at http://pkg.jenkins-ci.org/debian/.
10. Update the OS with regard to security patches (this may take some time depending on bandwidth):

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

11. Install the kernel's `dkms` module :

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

Note that the `dkms` module supports installing other kernel modules such as the modules needed by VirtualBox. For more details, visit https://help.ubuntu.com/community/DKMS.

12. Install Jenkins:

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

13. Install the kernel modules for VirtualBox:

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

14. Install guest additions using the **Devices** menu option in the VirtualBox window:



15. Add the `jenkins` user to the `vboxsf` group, as follows:

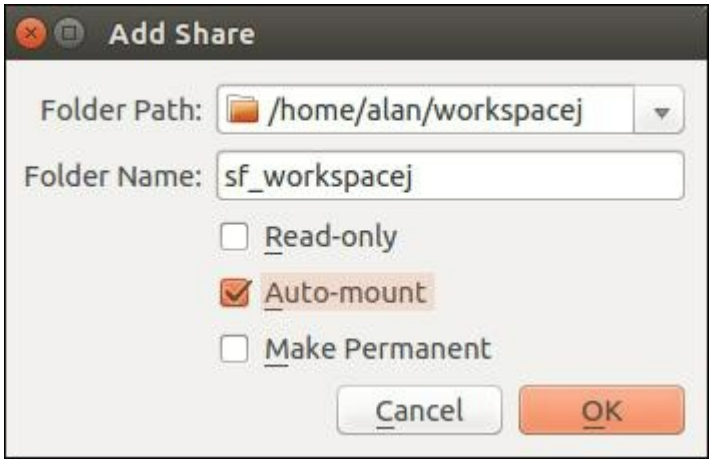
```
sudo gedit /etc/group
vboxsf:x:1001:Jenkins
```

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

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

17. On the host OS, create the directory named `workspacej`.
18. Within VirtualBox, right-click on the Ubuntu image and select **Settings**.

19. Update the **Folder Path** field to point to the directory that you previously created. In the following screenshot, you can see that the folder was created under my `home` directory:



20. Restart VirtualBox and then start the Ubuntu guest OS.
21. On the guest OS, run Firefox and browse to `http://localhost:8080`. You will see a locally running instance of Jenkins ready for your experiments.

How it works...

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

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

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

Note

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

The default Jenkins instance now needs a little reconfiguration:

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

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

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

See also

- The *Monitoring via JavaMelody* recipe
- Two excellent sources of virtual images at http://virtualboximages.com/ and http://virtualboxes.org/images/

Backing up and restoring

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

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

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

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

Getting ready

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

> **Tip**

Murphy as a friend

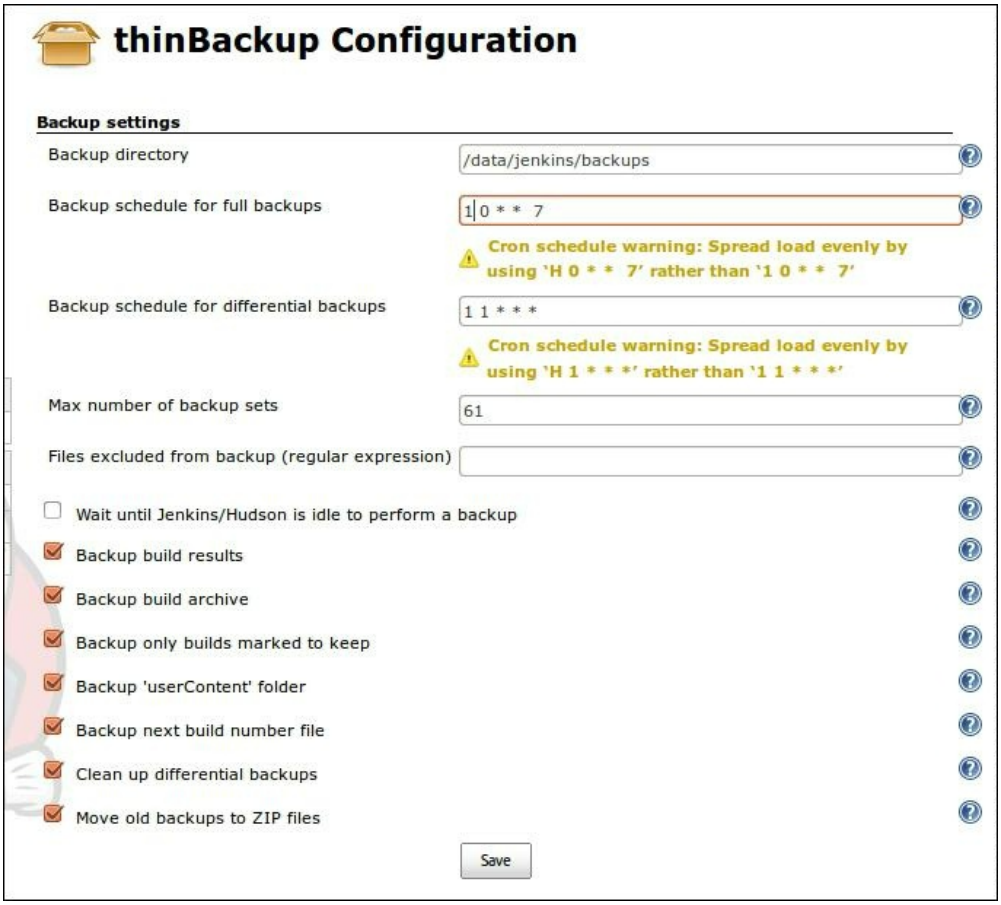
You should assume the worst for all of the recipes in this book: aliens attacking, coffee on motherboard, cat eats cable, cable eats cat, and so on. Make sure that you are using a test Jenkins instance.

How to do it...

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

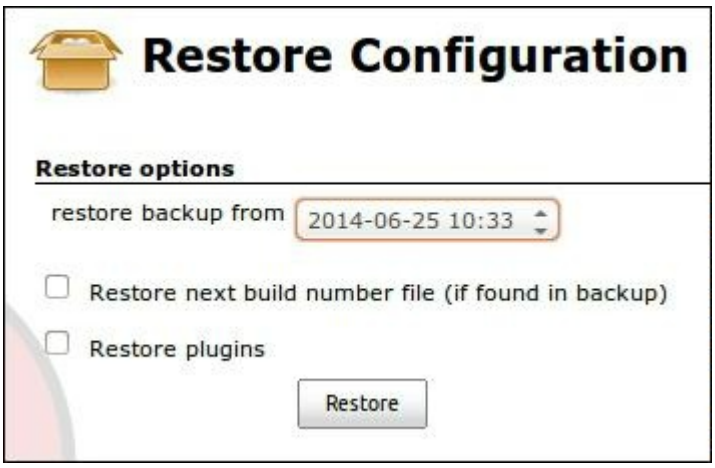


2. Click on the link to **Settings** by the **Toolset** icon.
3. Add the details as shown in the following screenshot where `/data/jenkins/backups` is a placeholder for the directory you have previously created. Notice the warning about using the `H` syntax; this will be explained later.



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

6. From the command line, visit your backup directory. You should now see an extra subdirectory named `FULL-{timestamp}` where `{timestamp}` is the time in seconds needed to create a full backup.
7. Click on the **Restore** icon.
8. A drop-down menu named **restore backup from** will be shown with the dates of the backups. Select the backup just created and click on the **Restore** button:



9. To guarantee consistency, restart the Jenkins server.

How it works...

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

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

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

Force Jenkins to quiet mode after specified minutes ensures that no jobs are running while backing up. This option forces quiet mode after waiting for Jenkins to be quiet for a specific amount of time. This avoids problems with backups waiting on Jenkins naturally reaching a quiet moment.

Differential backups contain only files that have been modified since the last full backup. The plugin looks at the last modified date to work out which files need to be backed up. The process can sometimes go wrong if another

process changes the last modified date without actually changing the content of the files.

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

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

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

> **Note**
>
> For safety's sake, regularly copy the archives off your system.

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

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

> **Note**
>
> I cannot repeat this enough; you should practice a restore occasionally to avoid embarrassment.
>
> Full backups are the safest as they restore to a known state. Therefore, don't generate too many differential backups between full backups.

There's more...

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

Checking for permission errors

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

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

Testing exclude patterns

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

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

Tip

Downloading the example code

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

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

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

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

Tip

EPIC Perl

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

See also

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

Modifying the Jenkins configuration from the command line

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

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

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

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

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

Getting ready

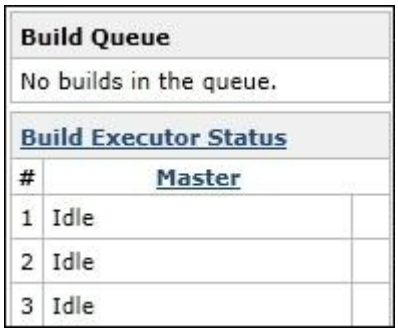
You will need a Jenkins server with security enabled and the ability to edit files either by logging in and working from the command line or through editing with a text editor.

How to do it...

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

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

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



3. Plugins persist their configuration through XML files. To prove this point, look for the `thinBackup.xml` file. You will not find it unless you have installed the thinBackup plugin.
4. Look at the *Backing up and restoring* recipe again. You will now find the following XML file:

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

How it works...

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

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

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

There's more...

Here are a few things for you to consider.

Turning off security

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

Finding JavaDoc for custom plugin extensions

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

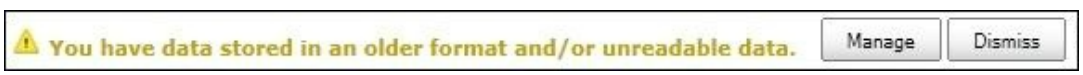
```
<org.jvnet.hudson.plugins.thinbackup.ThinBackupPluginImpl>
```

The effects of adding garbage

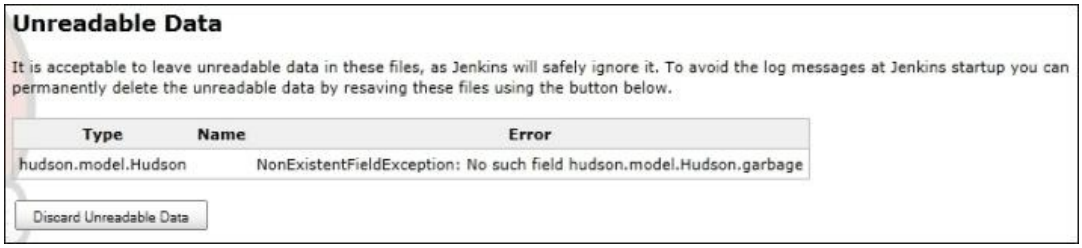
Jenkins is great at recognizing invalid configurations as long as they are recognizable as a valid XML fragment. For example, add the following line of code to `config.xml`:

```
<garbage>yeuchbllllllaaaaaa</garbage>
```

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



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



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

See also

- The *Using a test Jenkins instance* recipe

Installing Nginx

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

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

> **Note**
>
> You can find the wiki site for the Nginx community at http://wiki.nginx.org/Main.

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

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

Getting ready

Read the official installation instructions at http://wiki.nginx.org/Install.

How to do it...

1. From a terminal, type:

```
sudo apt-get install nginx
```

2. Browse to the localhost location. You will now see the Nginx welcome page:



3. From a terminal, type `sudo /etc/init.d/nginx` and you'll get the following output:

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

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

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

```
* nginx is running
```

5. Edit the welcome page within gedit:

```
sudo gedit /usr/share/nginx/html/index.html.
```

6. After the `<body>` tag, add `<h1>Welcome to nginx working with Jenkins</h1>`.
7. Save the file.
8. Browse to the localhost location. You will see a modified welcome page:



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

```
include /etc/nginx/conf.d/*.conf;
include /etc/nginx/sites-enabled/*;
access_log /var/log/nginx/access.log;
error_log /var/log/nginx/error.log;
```

10. Edit and save `/etc/nginx/sites-available/default`. For the two listen stanzas, change the number `80` to `8000`:

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

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

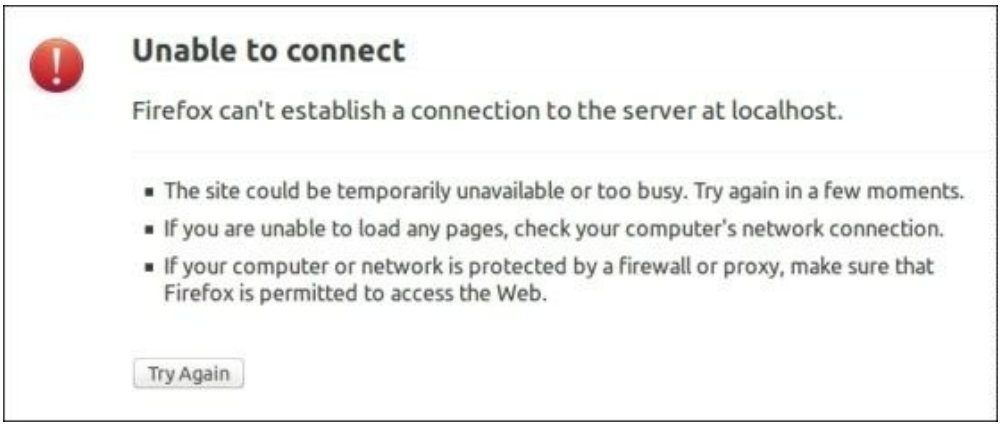
11. Test the configuration by running the following command from the terminal:

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

12. From the terminal, restart the server:

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

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



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

How it works...

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

The master configuration file is`/etc/nginx/nginx.conf`. The `include /etc/nginx/conf.d/*.conf;` line collects configuration settings from any file in the `/etc/nginx/conf.d` directory that has the `conf` extension. It also collects any configuration file from the `/etc/nginx/sites-enabled` directory through the `include /etc/nginx/sites-enabled/*;` command.

You changed the port number that the Nginx server was listening to through the `listen` directives in the default configuration file named `/etc/nginx/sites-available/default`. To avoid embarrassment, we tested the configuration before deploying the changes. You did this through the terminal with the `/etc/init.d/nginx configtest` command.

Tip

Support information

The *Nginx HTTP Server* book by *Packt Publishing* details many aspects of Nginx. You can find this book at https://www.packtpub.com/nginx-http-server-for-web-applications/book.

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

There's more...

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

Naming logfiles

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

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

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

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

Backing up configuration

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

See also

- The *Configuring Nginx as a reverse proxy* recipe

Configuring Nginx as a reverse proxy

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

Getting ready

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

How to do it...

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

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

2. Create the `/etc/nginx/sites-enabled/jenkins_8080_proxypass` file with the following lines of code:

```
server {
listen   80;
server_name  localhost;
access_log  /var/log/nginx/jenkins
_8080_proxypass_access.log;
error_log
 /var/log/nginx/jenkins_8080_proxypass_access.log;

location / {
proxy_pass      http://127.0.0.1:7070/;
include         /etc/nginx/proxy.conf;
        }
}
```

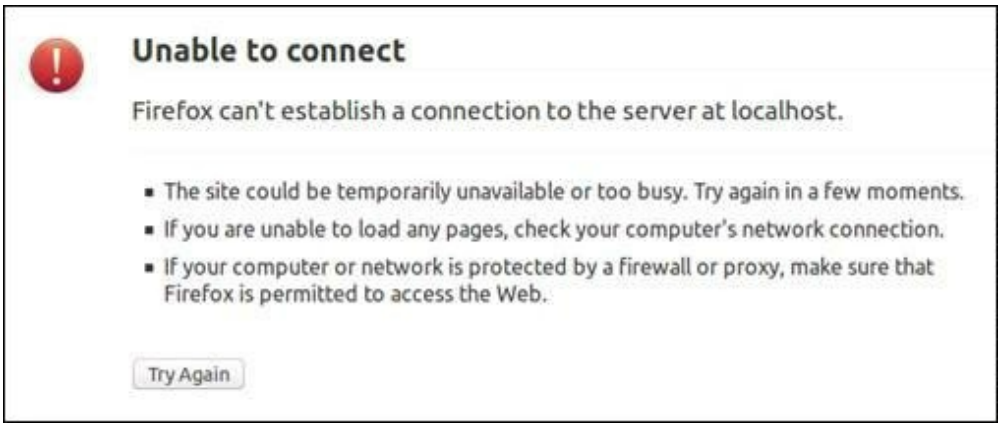
3. From the terminal, run `sudo /etc/init.d/nginx configtest`. You will see the following output:

```
* Testing nginx configuration    [ OK ]
```

4. In a terminal, restart the server by running the following command:

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

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



6. Review the access log `/var/log/nginx/jenkins_8080_proxypass_access.log`. You will see a line similar to the following line (note that `499` is the status code):

```
127.0.0.1 - - [25/Jun/2014:17:50:50 +0200]
"GET / HTTP/1.1" 499 0 "-" "Mozilla/5.0 (X11;
Ubuntu; Linux i686; rv:30.0) Gecko/20100101
Firefox/30.0"
```

7. Edit `/etc/nginx/sites-enabled/jenkins_8080_proxypass` by changing `7070` to `8080`:

```
location / {
proxy_pass      http://127.0.0.1:8080/;
include         /etc/nginx/proxy.conf;
        }
```

8. Test the configuration changes:

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

9. From a terminal, restart the Nginx server by running the following command:

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

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

How it works...

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

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

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

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

You also defined the location of the logfiles, as follows:

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

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

```
location / {
```

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

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

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

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

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

```
include         /etc/nginx/proxy.conf;
```

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

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

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

> **Note**
>
> For more information about `X-Forwarded-For`, visit http://en.wikipedia.org/wiki/X-Forwarded-For.

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

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

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

There's more...

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

Testing complex configuration

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

As mentioned in the preface, you can run Jenkins from the command line with commands similar to:

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

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

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

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

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

Offloading SSL

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

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

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

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

```
server {
listen 443;
server_name localhost;

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

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

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

Tip

Despite its simplicity, there are well-known configuration pitfalls, some of which are mentioned at http://wiki.nginx.org/Pitfalls.

See also

- The *Installing Nginx* recipe

Reporting overall storage use

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

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

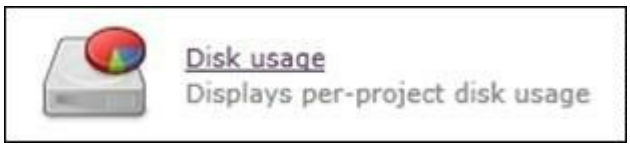
The disk usage plugin is strongest in combination with an early warning system that notifies you when soft or hard disk limits are reached. The *Adding a job to warn of storage use violations through log parsing* recipe details a solution. This recipe show that configuring Jenkins requires little effort.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

Getting ready

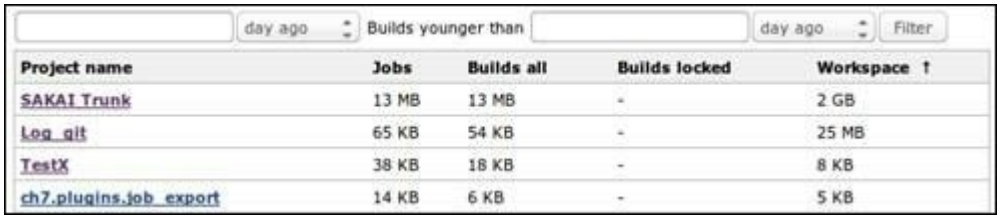
You will need to install the disk usage plugin.

How to do it...

1. Click on the **Disk usage** link under the **Manage Jenkins** page:



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

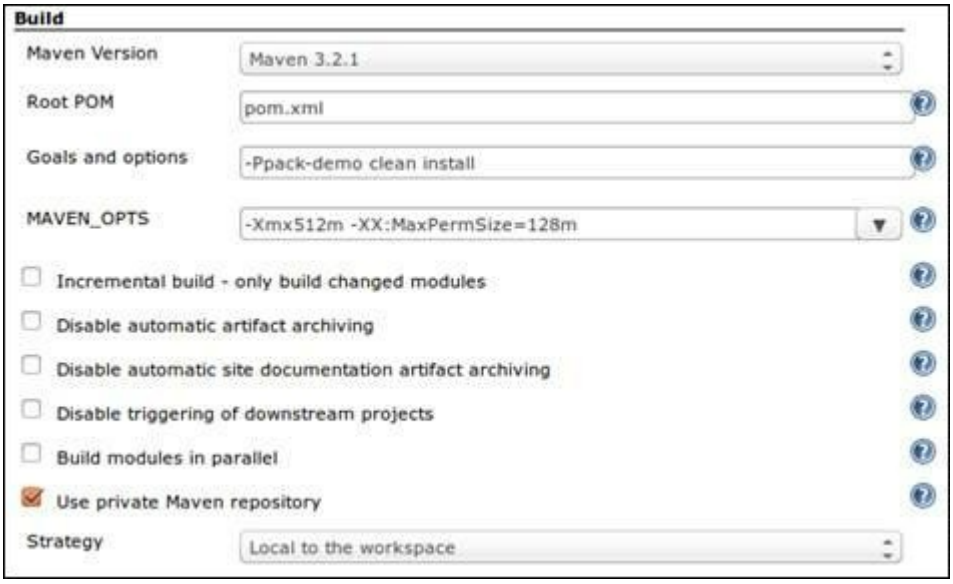


How it works...

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

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

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



Reviewing the advanced options of a project after looking at the disc usage of the project helps you to find unnecessary private repositories.

There's more...

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

Apache Archiva (http://archiva.apache.org/), or Nexus (http://nexus.sonatype.org/). These repository managers have significant advantages over dumping to disk, such as:

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

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

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

Note

Retention policy

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

See also

- The *Backing up and restoring* recipe

Deliberately failing builds through log parsing

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

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

Getting ready

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

How to do it...

1. Create the `log_rules` directory owned by Jenkins under the Jenkins workspace.
2. Add the `no_tests.rule` file to the `log_rules` directory with one line:

```
error /no tests/
```

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

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

4. Run the build. It should not fail.
5. As shown in the following screenshot, visit the **Manage Configuration** page for Jenkins and add a description and the location of parsing rules file in the **Console Output Parsing** section:



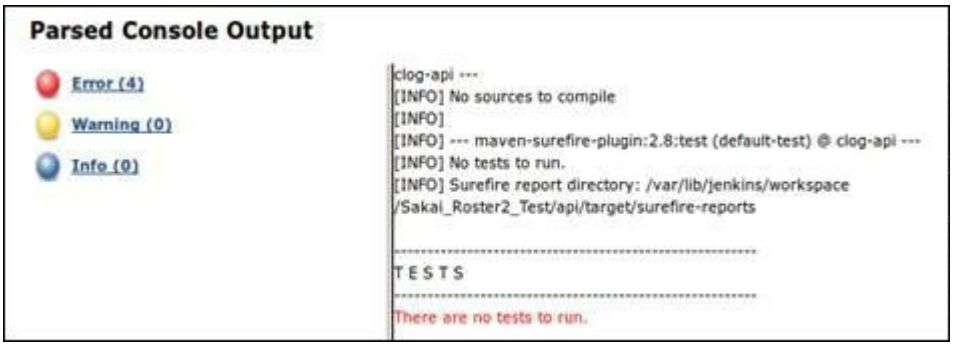
6. Check the **Console output (build log) parsing** box in the **Post-build Actions** section of your `Sakai_CLOG_Test` job.
7. Check the **Mark build Failed on Error** checkbox:



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

   Build the job and it should now fail.

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



How it works...

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

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

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

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

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

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

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

There's more...

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

- **MD5 checksums**: If a Maven repository has an artifact but not its associated MD5 checksum file, then the build will download the artifact even if it already has a copy. Luckily, the process will leave a `warn` message in the console output.
- **Failure to start up custom integration services**: These failures might be logged at the `warn` or `info` level when you really want them to fail the build.

See also

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

Adding a job to warn of storage use violations through log parsing

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

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

> **Note**
>
> 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 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 named `disk.rule` to the `log_rules` directory with the following two lines:

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

2. Visit the **Manage Configuration** page for Jenkins and add the description `DISC_USAGE` to the **Console Output** section. Point to the location of the parsing rules file.
3. Add the following Perl script named `disk_limits.pl` to a location of choice, making sure that the Jenkins user can read the file:

```
use File::Find;
my $content = "/var/lib/jenkins";
```

```
if ($#ARGV != 1 ) {
  print "[MISCONFIG ERROR] usage: hard soft
(in Bytes)\n";
  exit(-1);
}
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>=
$hard_limit (Bytes)\n";
}elseif ($total_bytes>= $soft_limit){
print "[SOFT_LIMIT WARN] $total_bytes>=
$soft_limit (Bytes)\n";
}else{
print "[SUCCESS] total bytes =
$total_bytes\n";
}

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

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



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



How it works...

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

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

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

There's more...

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

See also

- The *Backing up and restoring* recipe

Keeping in contact with Jenkins through Firefox

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

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

Getting ready

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

> **Note**
>
> The following URL will explain what happened to the Firefox status bar since the last edition of this book https://support.mozilla.org/en-US/kb/what-happened-status-bar.

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

How to do it...

1. Select the open menu icon at the top right-hand side of the browser:



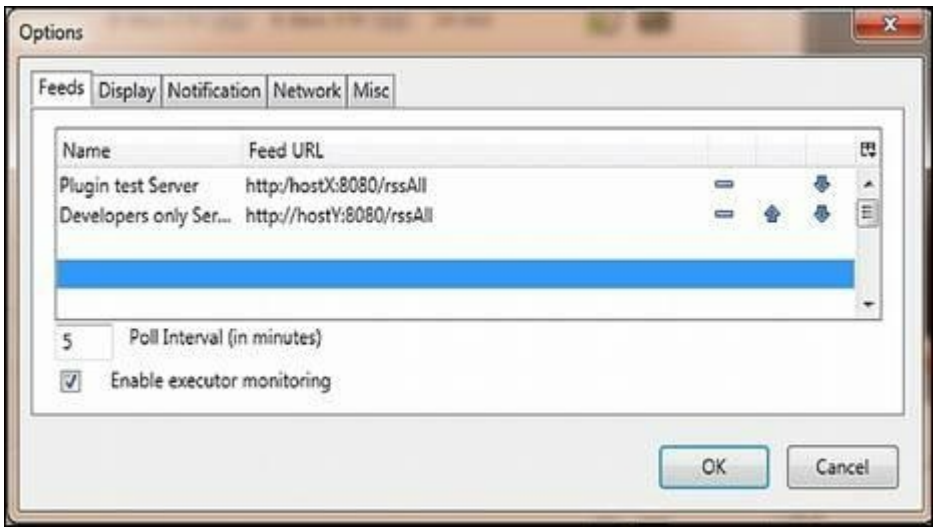
2. Click on the add-ons button:



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



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



11. Check **Enable executor monitoring**.
12. Click on the **OK** button. An area in the add-on toolbar will appear with the **Plugin test Server** name of the feed URL(s) displayed and a

health icon. If you hover your mouse over the name, more detailed status information will be displayed:



How it works...

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

To configure for a specific crucial job, you will need to use the following structure: `http://host:port/job/job name/rssAll`

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

There's more...

Here are a few more things to consider.

RSS credentials

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

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

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

Alternatives to Firefox

Firefox runs on a number of operating systems. This allows you to use one plugin for notifications across those operating systems. However, the disadvantage is that you have to keep a Firefox browser running in the background. An alternative is OS-specific notification software that pops up in the system tray. Examples of this type of software include CCMenu for

Mac OSX (http://ccmenu.org) or CCTray for Windows (http://en.sourceforge.jp/projects/sfnet_ccnet/releases/).

See also

- The *Mobile presentations using Google Calendar* recipe in Chapter 4, *Communicating Through Jenkins*

Monitoring via JavaMelody

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

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

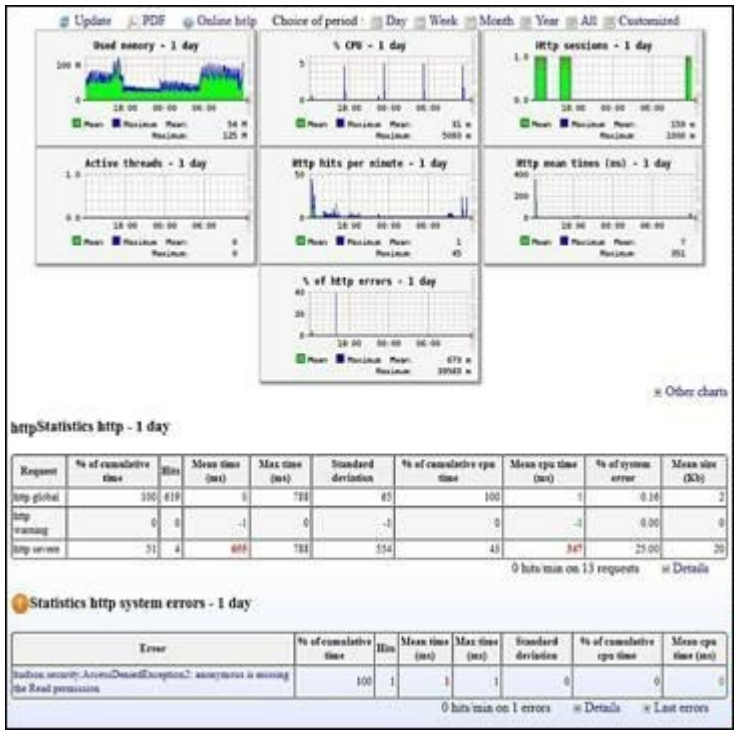
> **Tip**
>
> **Community partnership**
>
> If you find this plugin useful, consider contributing back to either the plugin or the core JavaMelody project.

Getting ready

You will need to have installed the Monitoring plugin.

How to do it...

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



2. Read the online help at `http://host:port/monitoring?resource=help/help.html` where 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 solution.

There's more...

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

Troubleshooting with JavaMelody – memory

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

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

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

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

Troubleshooting with JavaMelody – painful jobs

You should consider the following points:

- **Offload work**: For a stable infrastructure, offload as much work from the master instance as possible. If you have scheduled tasks, keep the heaviest ones separate in time. Time separation not only evens out load, but also makes finding the problematic build easier through the observation of the evolution charts of JavaMelody. Also consider spatial separation; if a given node or a labeled set of nodes shows problematic issues, then start switching around machine location of jobs and view their individual performance characteristics through `http://host:port/monitoring/nodes`.
- **Hardware is cheap**: Compared to paying for human hours, buying an extra 8 GB is roughly equivalent to one man's hour effort.

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

- **Review the build scripts**: Javadoc generation and custom Ant scripts can fork JVMs and reserve memory defined within their own configuration. Programming errors can also be the cause of the frustration. Don't forget to review JavaMelody's report on **Statistic system error log** and **Statistic http system errors**.
- **Don't forget external factors**: Factors include backups, cron jobs, updating the locate database, and network maintenance. These will show up as periodic patterns in the evolution charts.
- **Strength in numbers**: Use JavaMelody in combination with the disk usage plugin and others to keep a comprehensive overview of the vital statistics. Each plugin is simple to configure, but their usefulness to you will grow quicker than the maintenance costs of adding extra plugins.

See also

- The *Using Groovy hook scripts and triggering events on startup* recipe in Chapter 7, *Exploring Plugins*

Keeping track of script glue

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

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

Getting ready

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

How to do it...

1. Click on the **Scriptler** link under the **Manage Jenkins** page. You will notice the text in bold. Currently you do not have any scripts available; you can import scripts from a remote catalog or create your own.
2. Click on the **Remote Script catalogs** link on the left-hand side.
3. Click on the **ScriptierWeb** tab.
4. Click on the icon of the floppy disk for **getThreadDump**. If the script is not available, then choose another script of your choice.
5. Click on the **Submit** button.
6. You have now returned to the **Scriptler** main page. You will see three icons. Choose the furthest right to execute the script:



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

> **Note**

> If the script fails with a message `startup failed`, then please add a new line between `entry.key` and `for` and the script will then function correctly.

8. To write a new Groovy script or upload the one that you have on your local system, click on the **Add a new Script** link on the left-hand side.

How it works...

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

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

All local scripts are contained in the subdirectory scripts.

There's more...

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

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

See also

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

Scripting the Jenkins CLI

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

In this recipe, you will log on to a Jenkins instance, run a Groovy script that looks for files greater than a certain size, and log off. The script represents a typical maintenance task. After reviewing the output, you can run a second script to remove a list of files you want deleted.

> **Note**
>
> At the time of writing this chapter, the interactive Groovy shell was not working from the CLI. This is mentioned in the bug report at https://issues.jenkins-ci.org/browse/JENKINS-5930.

Getting ready

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 following 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 output will now mention all the oversized files:

```
java -jar jenkins-cli.jar -s http://host
groovy large_files.groovy
```

5. Logout by running the following command:

```
java -jar jenkins-cli.jar -s http://host
logout.
```

How it works...

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

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

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

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

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

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

There's more...

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

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

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

See also

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

Global modifications of jobs with Groovy

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

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

Getting ready

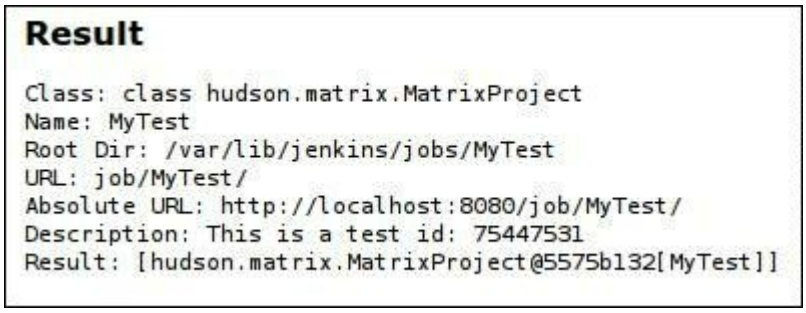
Log in to Jenkins with an administrative account.

How to do it...

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

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

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

if ("MyTest".equals(job.name)){
  println ("Description: ${job.description}")
job.setDescription("This is a test id:
${random.nextInt(99999999)}")
}
}
```

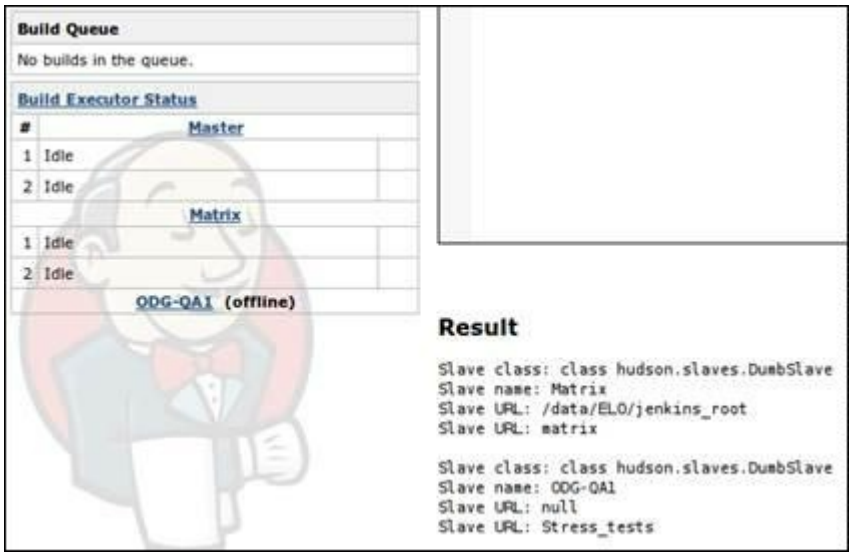
5. Click on the **Run** button. The results should be similar to the following screenshot:



6. Run the script again; you will notice that the random number in the description has now changed.
7. Copy and run the following script:

```
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"
}
```

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 that is exposed to the script console. The first script iterates through jobs whose parent is `AbstractItem` (http://javadoc.jenkins-ci.org/hudson/model/AbstractItem.html). The second

script iterates through instances of `slave` objects (http://javadoc.jenkins-ci.org/hudson/slaves/SlaveComputer.html).

There's more...

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

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

See also

- The *Scripting the Jenkins CLI* recipe
- The *Scripting global build reports* recipe

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, making a full backup of the entire filesystem.

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

Getting ready

Log in to Jenkins with an administrative account.

How to do it...

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

```
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, as shown in the following screenshot:



How it works...

Reviewing the code listing:

- A warning string is defined and the current date is stored in now. Each job in Jenkins is programmatically iterated via the `for` statement.
- Jenkins has a class to store build run information. The runtime information is retrieved via `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 via `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.

Tip

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 1,000 plugins and expanding, Jenkins has a large, freely available, and openly licensed example code base. Although in this case, the standard

API was used, it is well worth reviewing the plugin code base. In this example, you will find part of the code reused from the `lastsuccessversioncolumn` plugin. (http://tinyurl.com/pack-jenkins-1).

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

See also

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

Chapter 2. Enhancing Security

In this chapter, we will cover the following recipes:

- Testing for OWASP's top 10 security issues
- Finding 500 errors and XSS attacks in Jenkins through fuzzing
- Improving security via small configuration changes
- Avoiding sign-up bots with JCaptcha
- Looking at the Jenkins user through Groovy
- Working with the Audit Trail plugin
- Installing OpenLDAP
- Using 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
- Exploring the OWASP Dependency-Check plugin

Introduction

In this chapter, we'll discuss the security of Jenkins, taking into account that Jenkins can live in a rich variety of infrastructures. We will also look at how to scan for known security issues in the libraries used by Java code that Jenkins compiles.

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 open to attack. Jenkins' primary surfaces are its web-based graphical user interface and its trust relationships with its slave nodes and the native OS. Online services need vigorous attention to their security surface. For Jenkins, there are three main reasons why:

- Jenkins has the ability to communicate to a wide range of infrastructure through either 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
- You need to harden the default install that is open to the world

A counter-balance is that 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 vigil eye. It is unlikely that anyone contributing code would deliberately add defects or unexpected license headers. However, you should trust but verify.

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

Lightweight Directory Access (**LDAP**) is widely available and the de facto standard for Enterprise directory services. We will use LDAP for Jenkins authentication and authorization and later **single sign-on** (**SSO**) by JASIG's **Central Authentication Service** (**CAS**). To know more, visit 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, text files, and an increasing number of other methods. This allows Jenkins indirectly to use many logon protocols on top of the ones its plugins already provide.

Tip

Security Advisories

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

Testing for OWASP's top 10 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**). For more information, visit http://w3af.sourceforge.net. OWASP's purpose is to make application security visible. The OWASP's top 10 lists of insecurities for 2010 include the following:

- **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 600 plugins for Jenkins, each storing its 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 over HTTP by default. It can be a hassle and involve extra costs to obtain a trusted certificate. You might be tempted not to implement TLS, leaving your packets open.

You will find the OWASP's top 10 lists of insecurities for 2013 has some changes compared with the 2010 version. The most significant change is the inclusion of A9-Using Known Vulnerable Components. If your software depends on older libraries, then there is a window of opportunity for manipulation of known weaknesses.

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 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.

In the recipe named *Exploring the OWASP Dependency-Check plugin*, you will configure Jenkins to give you warning of known attack vectors, based on automatically reviewing your code dependencies.

> **Tip**
>
> **OWASP storefront**
>
> OWASP publish each year a list of the top 10 most common security attack vectors for web applications. They publish this document and a wide range of books through http://lulu.com. At Lulu, you have free access to PDF versions of OWASP's documents, or you can buy cheap-on-demand printed versions. You can find the official storefront at: http://stores.lulu.com/owasp.

Getting ready

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

Download the newest version of w3af from SourceForge (http://w3af.org/download/) and also download and read the OWASP's top 10 list of well-known attacks from https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

w3af has both Windows and *NIX installation packages; use the OS install of your choice. However, the Windows installer is no longer supported and the installation process without the installer is complex. Therefore, it's better to use a *NIX version of the tool.

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

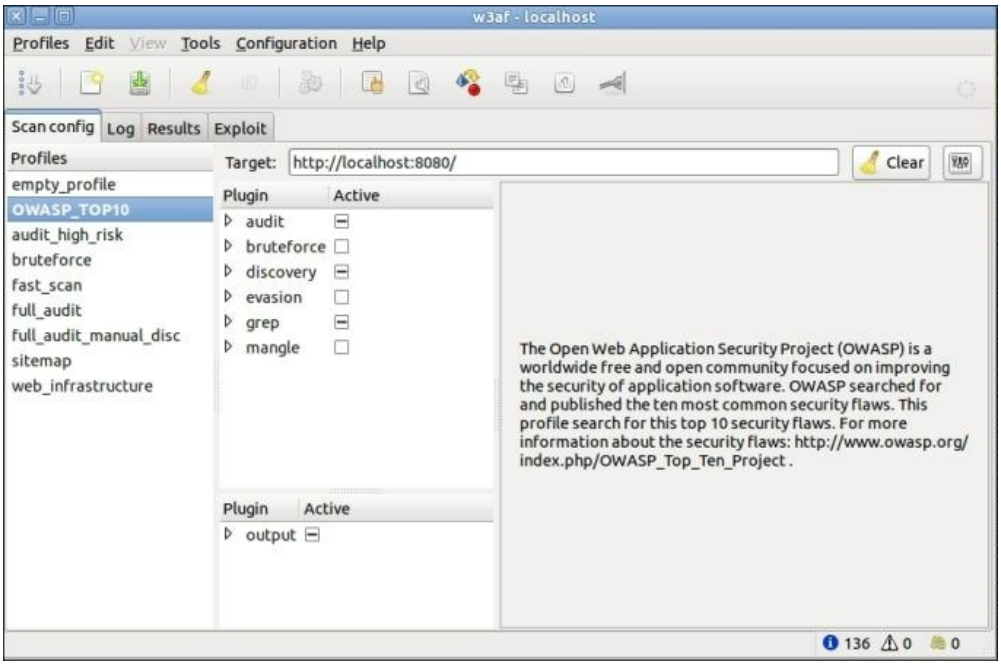
How to do it...

1. To install w3af, follow the instructions given on the developer site (http://w3af.org/download/). If there are any unsolvable dependency

issues for Ubuntu, fall back to the `apt-get` installation approach and install an older version of the tool as follows:

```
sudo apt-get install w3af
```

2. Run w3af.
3. Under the **Profiles** tab, select **OWASP_TOP10**.
4. Under the **Target** address window, fill in `http://localhost:8080/`, changing the hostname to suit your environment.
5. 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**:



6. View the attack history by selecting the **Log** tab.
7. Review the results by clicking on the **Results** tab.
8. After the first scan, select **full_audit** under **Profiles**.
9. Click on the **Clear** button.
10. Type `http://localhost:8080/` in the **Target** address window.
11. Click on the **Start** button.
12. Wait until the scan has finished and review 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 configuration 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 are occasionally flagged that do not exist. 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 like to note in 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 10 list of security defects can at times seem difficult to understand. If you have some spare time and like to practice 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 the following tools:

- **Nmap** (http://nmap.org/): A simple to use, highly popular, award-winning network scanner.
- **Nikto** (http://cirt.net/nikto2): A Perl script that quickly summarizes system details and looks for the most obvious of defects.
- **Skipfish** (https://code.google.com/p/skipfish/downloads/list): 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 that discovers attackable URLs and then cycles through a list of evil parameters.

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

> **Note**
>
> There are a number of great resources for securing native OSes including the Debian security how-to (https://www.debian.org/doc/manuals/securing-debian-howto/); for Windows, articles found underneath the MSDN security center (http://msdn.microsoft.com/en-us/security/); and for the Mac, Apple's official security guides (https://www.apple.com/support/security/guides/). Online services need vigorous attention to their security surface.

See also

- The *Finding 500 errors and XSS attacks in Jenkins through fuzzing* recipe
- The *Improving security via small configuration changes* recipe
- The *Exploring the OWASP Dependency-Check plugin* recipe

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 server's response. The inputted parameters are variations on 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, the fuzzer knows it has found an **evil URL**.

To fully integrate with the build process of a web-based application, you will need to build the application, deploy and run the application, run the fuzzer from a script, and finally use log parsing to fail the build if evil URLs are mentioned in the output. This process will be similar for other command-line tools you wish to integrate. For more information about log parsing, refer to *Deliberately failing builds through log parsing* recipe in Chapter 1, *Maintaining Jenkins*.

Getting ready

Backup 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://wapiti.sourceforge.net.

Note

If you're 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.

In this recipe, the methodology and command-line options are correct. However, at the time of reading, the results mentioned may not exist. Jenkins goes through a rapid lifecycle where developers remove bugs rapidly.

How to do it...

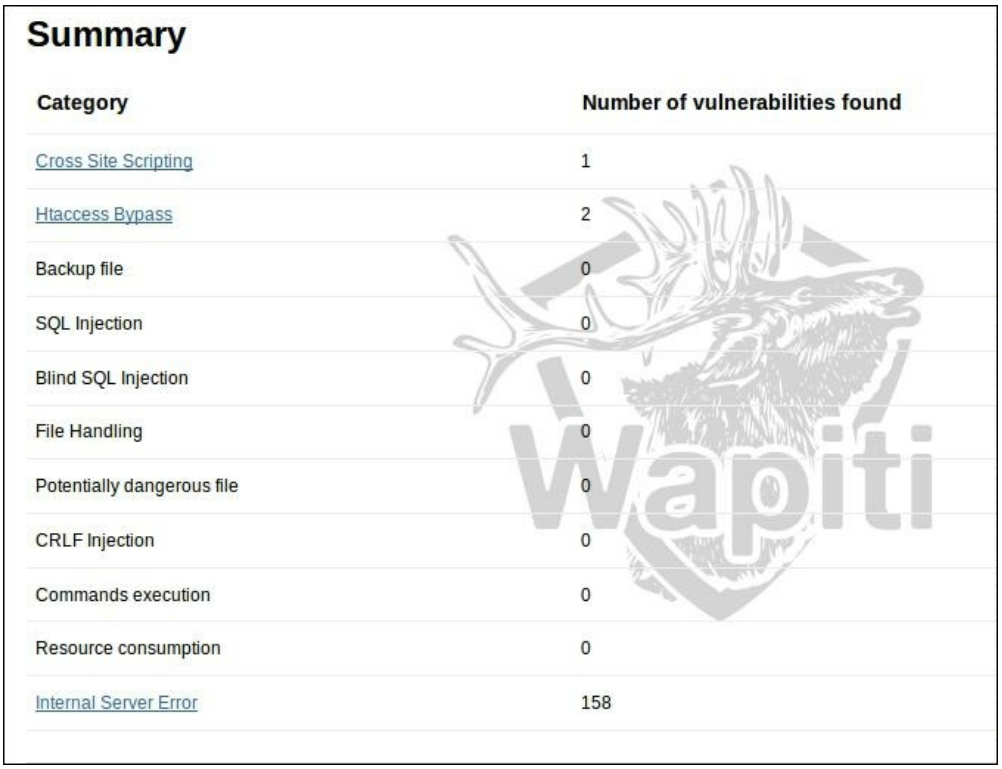
1. Within the `wapiti` bin directory, run the following command:

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

2. When the command has finished running, you will see the location of the final report on the console output:

```
Report
------
A report has been generated in the file
~/.wapiti/generated_report
~/.wapiti/generated_report/index.html with a
browser to see this report.
```

3. Open the report in a web browser and review:



Summary

Category	Number of vulnerabilities found
Cross Site Scripting	1
Htaccess Bypass	2
Backup file	0
SQL Injection	0
Blind SQL Injection	0
File Handling	0
Potentially dangerous file	0
CRLF Injection	0
Commands execution	0
Resource consumption	0
Internal Server Error	158

4. Click on the **Internal Server Error** link.
5. For one of the items named **Anomaly found in /iconSize**, copy the URL from the **cURL command line** tab:



6. Open the URL in a web browser. You will now see a newly generated Jenkins bug report page, as shown in the following screenshot:



7. Run the following command:

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

8. View the output to verify that the `permanentxss` module was run:

```
[*] 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
```

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, this causes Wapiti to crash or timeout.

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

The `-m "-all,xss,exec"` statement 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, then 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 cause work for the plugin manager. If we do, it will then generate a lot of requests to an innocent external service.

Wapiti crawls websites. If you are not careful, the tool might follow a link to locations that you do not want testing. To avoid embarrassment, carefully use the exclude URL's option `-x`.

The `-v2` option 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 at times finds bonafide XSS attacks, depending on the race between developers building features and cleaning bugs.

> **Note**
>
> **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 time to search out. Consider automating through a Jenkins job as part of a project's QA process.

There's more...

The reports you generated in this recipe mention many more server errors than XSS attacks. This is because many of the errors generated are due to unexpected input causing failures that are only caught by the final layer of error handling, in this case the bug report page. If you consider the error worth reporting, then follow the instructions found on the bug report page.

Here are some guidelines for the meaning behind the output of the stack traces:

- `java.lang.SecurityException`: If a Jenkins user is doing something that the programmer considers insecure, such as hitting a URL, this should only be reachable once you have logged in.
- `java.lang.IllegalArgumentException`: Jenkins checked for a valid range for your parameter and the parameter value was outside that range. This is a deliberately thrown exception.
- `java.lang.NumberFormatException`: Jenkins did not check for a valid string and then tried to parse a nonconformant string to a number.
- `java.lang.NullPointerException`: This normally happens when you hit a URL without all the parameters set that Jenkins expects. In programmer's language: the code was expecting an object that does not exist and then tries to call a method of the nonexistent object without checking that the object exists. The programmer needs to add more error-checking. Write a bug report.

See also

- The *Testing for OWASP's top 10 security issues* recipe
- The *Improving security via small configuration changes* recipe

Improving security via small configuration changes

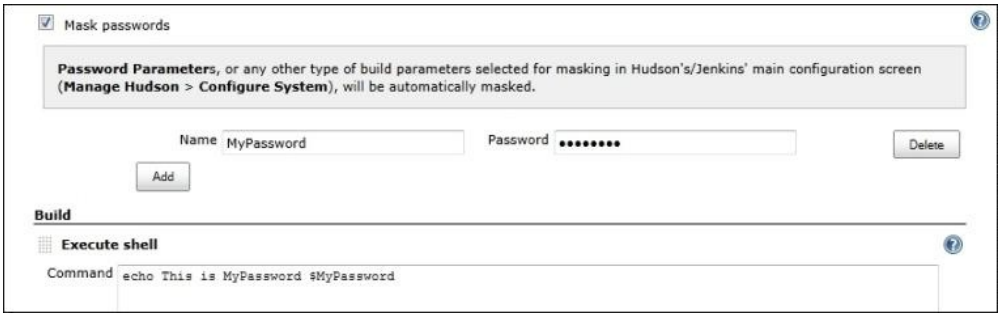
This recipe describes modest configuration changes that strengthen the default security settings of Jenkins. The reconfiguration includes masking passwords in console output and adding a one-time random number, which makes it more difficult for form input to be forged. The combination of tweaks strengthens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

Getting ready

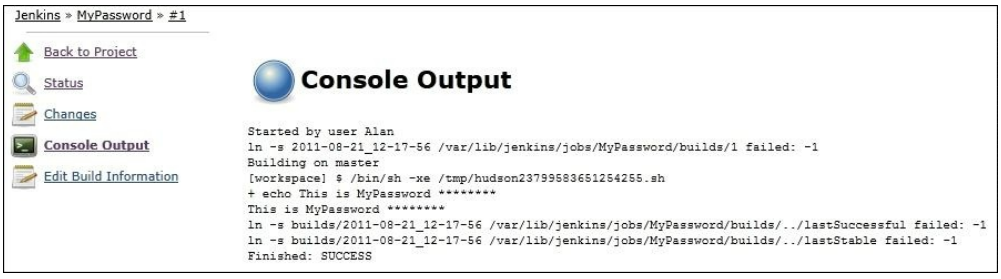
You will need to install the Mask Passwords plugin (https://wiki.jenkins-ci.org/display/JENKINS/Mask+Passwords+Plugin).

How to do it...

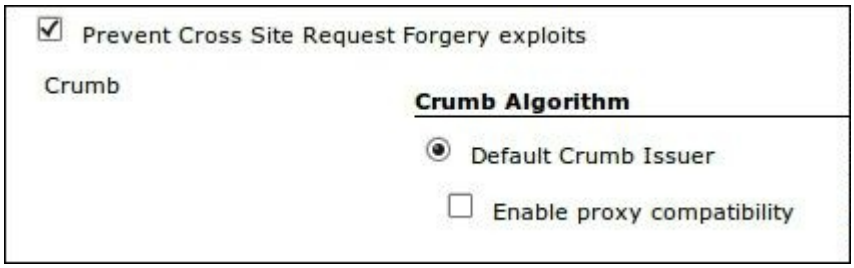
1. Create a job.
2. Click on the **Mask passwords** checkbox and add a variable.
3. Type `MyPassword` in the **Name** field and `changeme` in the **Password** field, as shown in the following screenshot:



4. Type in `echo This is MyPassword $MyPassword` in **Execute shell**.
5. Run the job.
6. Review **Console Output**:



7. Return to the **Configure Global Security** page and click on **Prevent Cross Site Request Forgery exploits**, making sure the **Default Crumb Issuer** option is selected:



How it works...

The Mask Passwords plugin removes the password from the screen or the console, replacing it with **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.

> **Note**
>
> You can globally set parameters to mask within the **Configure System** under the **Mask passwords – Parameters to automatically mask** section.

Cross-site request forgery (http://en.wikipedia.org/wiki/Cross-site_request_forgery) occurs, for example, if you accidentally visit a third party's location; a script at that location tries to make your browser perform an action (such as deleting a job) by making your web browser visit a known URL within Jenkins. Jenkins, thinking 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**, which is returned as part of the request. The number is not easily known and is also invalidated after a short period 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 HTML escaping early before the group of administrators gets used to the flexibility of being able to add arbitrary tags.

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

See also

- The *Testing for OWASP's top 10 security issues* recipe
- The *Finding 500 errors and XSS attacks in Jenkins through fuzzing* recipe

Avoiding sign-up bots with JCaptcha

CAPTCHA stands for **Completely Automated Public Turing Test to tell Computers and Humans Apart**. The most commonly viewed CAPTCHAs are sequential letters and numbers displayed as graphics that you have to correctly feed into a text input.

If you let anyone sign up for an account on your Jenkins server, then the last thing you want are bots (automated scripts) creating accounts and then using them for less-than-polite purposes. Bots have an economy of scale, being able to scan the Internet rapidly and never getting bored. CAPTCHAs are a necessary defense against these dumb attacks.

The negative purposes of bots are as follows:

- Performing a **Denial Of Service** (**DOS**) attack on your server, for example, by automatically creating numerous heavyweight jobs
- **Distributed Denial Of Service attack** (**DDOS**) on other servers by harvesting many Jenkins servers to fire off large numbers of requests
- By injecting unwanted advertisements or content that then points to malicious sites
- By adding scripts that are stored permanently and run when a user accidentally browses Jenkins site

> **Note**
>
> There are commercial motivations for criminals to circumvent CAPTCHAs that led to well documented law cases. You can find one such law case at http://www.wired.com/2010/10/hacking-captcha/.

Getting ready

Make sure you have backed up your sacrificial Jenkins server. You are going to modify its security settings. It is easy to make a service changing mistake.

> **Tip**

The JCaptcha plugin is based on Java implementation that you can find at https://jcaptcha.atlassian.net/wiki/display/general/Home.

How to do it...

1. Log in as an administrator.
2. Click on the **Configure Global Security** link.
3. Select Jenkins' own user database under **Security Realm**.
4. Select **Allow users to sign up**, as shown in the following screenshot:



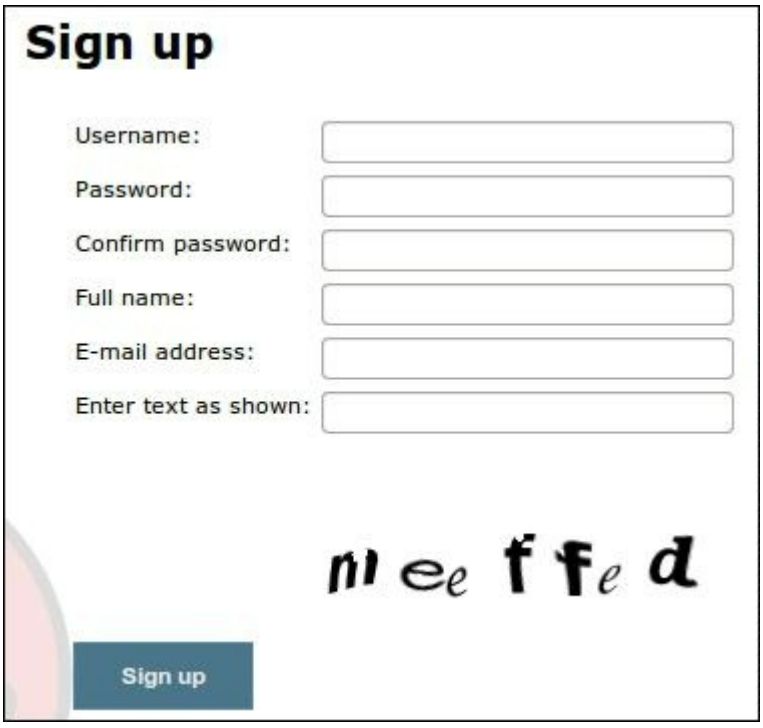
5. Press **Save**.
6. Browse the signup location `http://localhost:8080/signup`. You will see something similar to the following screenshot:



7. Click on the **Manage Plugins** link in the **Manage Jenkins** page.
8. Select the **Available** tab.
9. Install the JCaptcha plugin.
10. Click on the **Configure Global Security** link under the **Manage Jenkins** page.
11. Select Jenkins' own user database under **Security Realm**.
12. Select **Enable captcha on sign up**, as shown in the following screenshot:



13. Press **Save** and then click on the **Log Out** link.
14. Browse the signup location `http://localhost:8080/signup`. The page is now defended by a CAPTCHA, as shown in the following screenshot:



How it works...

Installing the plugin adds the CAPTCHA image to the sign-up process. The image needs pattern recognition to decipher. Humans are very good at this; automated processes are a lot worse, but improving.

There's more...

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

Defense in depth

There is an arms race between defensive methods such as CAPTCHAs and offensive methods such as increasingly intelligent bots. No one solution will cut down the risk to zero. It is best practice to consider a layered approach. Depending on your requirements, consider adding authentication, keeping access down to known IP addresses, backing up your configuration, reviewing your logfiles, vulnerability testing, and working on the general security hygiene of your site.

Tip

The SANS institute has written a paper on the Defense in Depth strategy http://www.sans.org/reading-room/whitepapers/basics/defense-in-depth-525.

More information on bots

The security arms race continues. Bots are getting cleverer and script kiddies more numerous. Here are a few background articles to this arms race:

- For more information about the script kiddie, visit http://en.wikipedia.org/wiki/Script_kiddie
- A report by Imperva explaining why CAPTCHAs are getting easier to crack at http://www.imperva.com/docs/HII_a_CAPTCHA_in_the_Rye.pdf
- Google are improving the difficulty of mimicking CAPTCHAs (http://www.cnet.com/news/whats-up-bot-google-tries-new-captcha-method/)

See also

- The *Testing for OWASP's top 10 security issues* recipe

Looking at the Jenkins user through Groovy

Groovy scripts run as the Jenkins user on the host server. This recipe highlights the power and danger to the Jenkins application and the host server.

Getting ready

Log in to your test Jenkins instance as an administrator.

How to do it...

1. Run the following script from **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.

   For a typical *NIX system, it will be similar to the following screenshot:

Result

```
Location /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
```

For a Windows system, it will be similar to the following screenshot:

Result

```
/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.
#
# This file contains the mappings of IP addresses to host names. Each
# entry should be kept on an individual line. The IP address should
# be placed in the first column followed by the corresponding host name.
# The IP address and the host name should be separated by at least one
# space.
#
# Additionally, comments (such as these) may be inserted on individual
# lines or following the machine name denoted by a '#' symbol.
#
# For example:
#
#      102.54.94.97     rhino.acme.com          # source server
#       38.25.63.10     x.acme.com              # x client host

127.0.0.1       localhost
::1             localhost
```

How it works...

The script you have run is less benign than it first seems. Groovy scripts can do anything the Jenkins user has the power to do on the host server, as well as within the test Jenkins server. A method is defined that reads in a file whose location is passed as a string. The script then prints the content. If the file does not exist, then 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 username has a real login account (not `/bin/false`) and whether they have shell scripts suggest accounts to try and crack by focusing 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 the plugin's XML configuration files and parse.

Not only can you read files, but you can 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 **Script console** and to periodically review the audit log.

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

> **Note**
>
> 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

- The *Working with the Audit Trail plugin* recipe
- The *Reviewing project-based matrix tactics via a custom group script* recipe

Working with the Audit Trail plugin

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

Getting ready

Install the Audit Trail plugin (https://wiki.jenkins-ci.org/display/JENKINS/Audit+Trail+Plugin).

How to do it...

1. Visit the **Configure Jenkins** screen (`http://localhost:8080/configure`).
2. Under the **Audit Trail** section, click the **Add Logger** button.
3. Modify the default settings for **Audit Trail** to allow for a long observation. Change **Log File Size MB** to `128` and **Log File Count** to `40`.
4. Click on the **Advanced…** button to review all the settings.



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's **Recorders** page at `http://localhost:8080/log/?` to see which recorders are logging.

The output from the log recorder is filtered via **URL Patterns to log** as seen in Jenkins's configure screen. You will find that the logfile format is more readable than most, with a date timestamp at the beginning, a description of what is happening in the middle of the log, and the user who acted at the end. Take a look at the following 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.

> **Note**
>
> Consider placing the `audit.log` file itself under a version control system. It is a good idea to do so for three main reasons. The first is in case of storage failure. The second is to make it more difficult to alter the audit logs without leaving evidence. Finally, this is a central location where you can gather small logfiles from your whole enterprise.

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, then this 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 filesystem, you should investigate. Missing logs should trigger a wider review of the server in question. At the 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 *Reporting alternative code metrics in Jenkins* recipe in Chapter 3, *Building Software*, to parse the logfile and make metrics that are then displayed graphically. This enables you to view the ebb and flow of your team's work over time. Of course, the data can be faked, but that would require a lot of extra effort.

> **Note**
>
> One method to minimize the risk of logfile tampering is also to send your log events to a central remote syslog server. You can configure the Audit Trail plugin to work with syslog within the **Configure System** page.

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 the case of 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 for most *NIX distributions. (http://sourceforge.net/projects/swatch/ and 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 e-mail or by executing commands.

See also

- The *Improving security via small configuration changes* recipe
- The *Looking at Jenkins user through Groovy* recipe
- The *Reporting alternative code metrics in Jenkins* recipe in Chapter 3, *Building Software*

Installing OpenLDAP

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 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 part of the build application's 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).

> **Note**
>
> Active Directory is also popular in corporate environments. Jenkins has a plugin for Active Directory (https://wiki.jenkins-ci.org/display/JENKINS/Active+Directory+plugin).

Getting ready

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

> **Note**
>
> For a detailed description of installing OpenLDAP on the Windows, refer to http://www.userbooster.de/en/support/feature-articles/openldap-for-windows-installation.aspx.

Save the following LDIF entries to the `basic_example.ldif` file 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` by executing the following command:

```
sudo apt-get install slapdldap-utils
```

2. When asked, fill in the administration password.
3. Add the LDIF records from the command line; you will then be asked for the administrator password you used in step 2. Execute the following command:

```
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**): This is a unique identifier per record and is structured so that objects reside in an organizational tree structure.
- `objectClass`: `objectClass`, such as `organizationalUnit`, defines a set of required and optional attributes. In the case of the `organizationalUnit`, the `ou` attribute 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, after installing an LDAP server, we imported via the admin account (default `dn:cn=admin,dc=nodomain`) created during the package installation; if this is the case, you will need to change the value of the `-D` option 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 `inetOrgPerson` objectClass.

A group (`dn: cn=dev,ou=groups,ou=mycompany,dc=nodomain`) is created via the `groupOfNames` objectClass. 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 which group the user belongs to. 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 `objectClass` and **Access Control Lists** (**ACLs**):

- `objectClass`: LDAP uses `objectClass` as a sanity check on 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 a new `objectClass`; you can do this with graphical tools. The *Administering OpenLDAP* recipe shows one such tool.
- **Access Control Lists**: These define which user or which group can do what. For information on this complex subject area, visit http://www.openldap.org/doc/admin24/access-control.html. You can also review the main entry on your OpenLDAP server from the `man slapd.access` command line.

See also

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

Using Script Realm authentication for provisioning

For many enterprise applications, provisioning occurs during the user's first login. For example, a directory with content can 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 the Perl and `Net::LDAP` modules installed. For a Debian distribution, you should install the `libnet-ldap-perl` package via the following command:

```
sudo apt-get install libnet-ldap-perl
```

You also need to have the Script Realm plugin installed (https://wiki.jenkins-ci.org/display/JENKINS/Script+Security+Realm).

How to do it...

1. As the Jenkins user, place the following file under a directory that is controlled by Jenkins, 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::Utilqw(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 the Jenkins user, place the following file under a directory that is controlled by Jenkins, with executable permissions. Name the file `group.pl`:

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

3. Configure the plugin via the **ConfigureGlobalSecurity** screen under the **Security Realm** subsection and then add the following details:
    - 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`. You will see the following screenshot:



How it works...

The `login.pl` script 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`.

Note

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

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

If the file does not already exist, then 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 includes every user, guests and all. 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, then 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 counting down records. A scheduled task performs an LDAP search on the counter and then decrements the counter by one. The task notices when the counter reaches certain numbers and 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 grace period before their LDAP record is moved to a "to be ignored" branch.

See also

- The *Reviewing project-based matrix tactics via a custom group script* recipe

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 different permissions to individual users or groups on a job-by-job basis.

The recipe uses custom realm scripts enabled through the Script Security plugin to allow you to log in with any name and password whose length is greater than five characters, and places the test users in their own unique group. This will allow you to test out the project-based matrix strategy.

Using custom scripts to authenticate users and define groups allows your test Jenkins server to connect to a wide variety of nonstandard authentication services.

Getting ready

You will need to install the Script Security Realm plugin (https://wiki.jenkins-ci.org/display/JENKINS/Script+Security+Realm) 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 `login2.pl` file in the Jenkins workspace:

```
#!/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. Change the owner and group of the script to `jenkins`, as follows:

```
sudo chown
jenkins:jenkins /var/lib/jenkins/login2.pl
```

3. Copy the following script to the `group2.pl` file in the Jenkins workspace:

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

4. Change the owner and group of the script to `jenkins`, as follows:

```
sudo chown
jenkins:jenkins /var/lib/jenkins/group2.pl
```

5. Configure the plugin via the **Configure Global Security** screen under the **Security Realm** subsection.
6. Select the **Authenticate via custom script** radio button and add the following details:

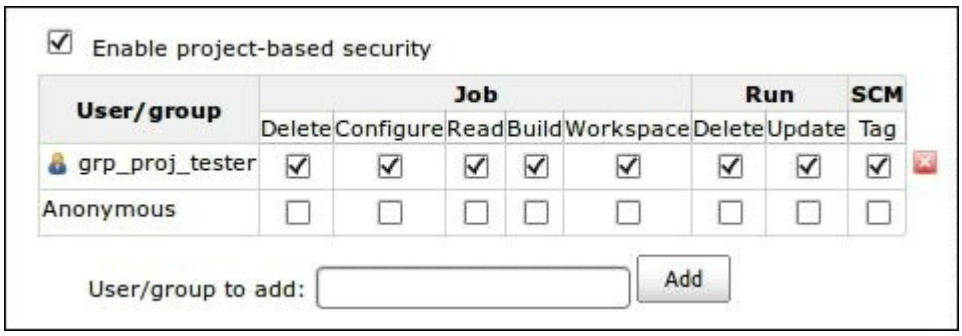


7. Check the **Project-based Matrix Authorization Strategy** checkbox.
8. Add a user called `adm_alan` with full rights, as shown in the following screenshot:



9. Press the **Save** button.
10. Try to log in as `adm_alan` with a password less than five characters.
11. Log in as `adm_alan` with any password greater than five characters.

12. Create a new job with the name `project_ matrix_test` and no configuration.
13. Check the **Enable project-based security** checkbox within the job.
14. Add the `grp_proj_tester` group with full rights (for example, check all the checkboxes):



15. Log in as user `I_cant_see_you`. Notice that you cannot view the recently created job `project_matrix_test`.
16. Login as `proj_tester`. Notice that you can now view and configure `project_matrix_test`.

How it works...

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

The `group2.pl` script reads the username from the environment and then escapes the name to make sure that no evil scripting is accidentally run later. The `group2.pl` script places the user in the `grp_username` group. For example, if `proj_tester` logs in, they will belong to the `grp_proj_tester` group.

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

- Globally via the Jenkins configuration page. This is where you should define your global accounts for system-wide administration.
- 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 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 `grp_proj_tester` group, which has full powers within the specific job.

Using per-project permissions, you not only limit the powers of individual users but you can also shape which projects they can 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 can be as follows:

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

Later, 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 such as 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. PMD and FindBugs are excellent examples (http://fsmsh.com/2804.com). A number of these generic of 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 it, the input has to be first passed through a regular expression and unwanted input safely escaped, removed, or reported.

See also

- The *Using Script Realm authentication for provisioning* recipe

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 JExplorer (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
```

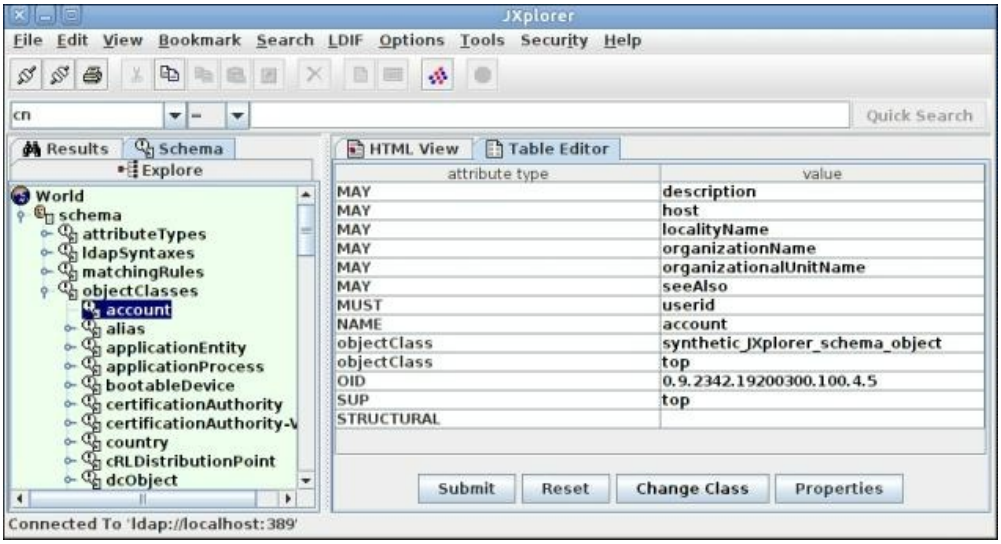
2. Add a new line at the end of the record and copy the preceding record into the text file, replacing the number `121` with `122` wherever it occurs in the second record.
3. Run the following `ldapadd` command, and when asked, input the LDAP administrator's password.

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

4. Run Jxplorer connecting with the following values:

    - **HOST**: `localhost`
    - **Level**: `Anonymous`
    - Select the **Schema** tab and then select **account** under **objectClasses**.

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



5. Disconnect from the `Anonymous` account by selecting **File** and then **Disconnect**.
6. Reconnect as the `admin` account by selecting **File** and then **Connect**. Add the following details:

    - **Host**: **Localhost**
    - **Level**: User + Password
    - **User DN**: `cn=admin,dc=nodomain`
    - **Password**: your password

7. Under the **Explore** tab, select **tester1**.
8. In **Table Editor**, add the **1021 XT** value to **postalCode** and click on **Submit**.
9. Select the **LDIF** menu option at the top of the screen and then click on **Export Subtree**.
10. Click on the **OK** button and write the name of the file that you are going to export the LDIF to and then click on **Save**.
11. Create an executable script with the following lines of code and run it:

```
#!/usr/bin/perl
use Net::LDAP;
use Net::LDAP::Utilqw(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 OS and is open source. Your actions highlight 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 highlights 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 the command-line tools or be very careful with search filters.

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

Next, you bind (perform some action) as an admin user and manipulate the tester1's 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 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 the user's Perl script as an example of how to provision or cleanup an LDAP server that is needed for integration tests efficiently. To create an add script instead of a delete script, you can write a similar script replacing my `$result = $ldap->delete($dn)` with the following line of code:

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

Here, `@$whatTOCreate` is a hash containing attributes and `objectClass`. For more examples, visit http://search.cpan.org/~gbarr/perl-ldap/lib/Net/LDAP/Examples.pod#OPERATION_-_Adding_a_new_Record.

See also

- The *Installing OpenLDAP* recipe
- The *Configuring the LDAP plugin* recipe

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

To try this recipe, you should first perform the steps mentioned in the *Installing OpenLDAP* recipe.

How to do it...

1. Go to the **Configure Global Security** screen and select **Enable security**.
2. Check the **LDAP** checkbox.
3. Add the **Server** value as `127.0.0.1`.
4. Click on the **Advance** button and 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 to anonymously 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 that has the rights to perform the searches. This account has 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 `uids` 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.

Tip

The Security Realm

When you log in, an instance of the `hudson.security.LDAPSecurityRealm` class is called. The code is defined in a Groovy script that 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 as follows:

- **A misconfigured DN for either userSearchBase or GroupSearch base**: The relevant log entry will look as follows:

```
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'
```

- **Bad credentials**: If the user does not exist in LDAP, you have either typed in the wrong password or you have accidentally searched the wrong part of the LDAP tree. So 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 accidentally.
- **Self-bind**: The application binds as a user and then searches with that user's rights. This approach is the cleanest. However, it is not always clear in the logging that the application is behind the 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 it can cause significant issues quickly.

> ### Note
>
> If the LDAP server has an account locking policy, then it is trivial for a cracker to lock out the application.

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

> ### Tip
>
> **Reviewing plugin configuration**
>
> Currently, there are over 600 plugins for Jenkins. It is possible, though unlikely, that occasionally passwords are being stored in plaintext 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 text, you should write a bug report attaching a patch to the report.

See also

- The *Installing OpenLDAP* recipe

- The *Administering OpenLDAP* recipe

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 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 makes integration of third-party applications straight-forward.

Yale CAS has also 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. The modified configuration files mentioned in this recipe are downloadable from the book's website.

Getting ready

Download Yale CAS (https://www.apereo.org/cas/download) from the 3.4 line and unpack. This recipe was written with version 3.4.12.1, but it should work with little modification with earlier or later versions of the 3.4 line.

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

> **Note**
>
> Since June 2014, the CAS 4 documentation has moved from the JASIG website to http://jasig.github.io/cas/4.0.x/index.html.

How to do it...

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

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

```
-->
```

2. Add the following underneath the text needed to enable port `9443` with a SSL enabled:

```
<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. The user that Tomcat will run under creates a self-signed certificate via the following command:

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

> **Note**
>
> If `keytool` is not found on your `PATH` environmental variable, then you might have to fill in the full location to the `bin` directory of your installed Java.

4. From underneath the root directory of the unpacked CAS server, copy the `modules/cas-server-uber-webapp-3.x.x` file. (where `x.x` is the specific version number) to the Tomcat web app's directory, making sure the file is renamed to `cas.war`.
5. Start Tomcat.
6. Log in via `https://localhost:9443/cas/login` 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 book's website. To modify, you will need to comment out the `SimpleTestUsernamePasswordAuthenticationHandler` line, as follows:

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

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

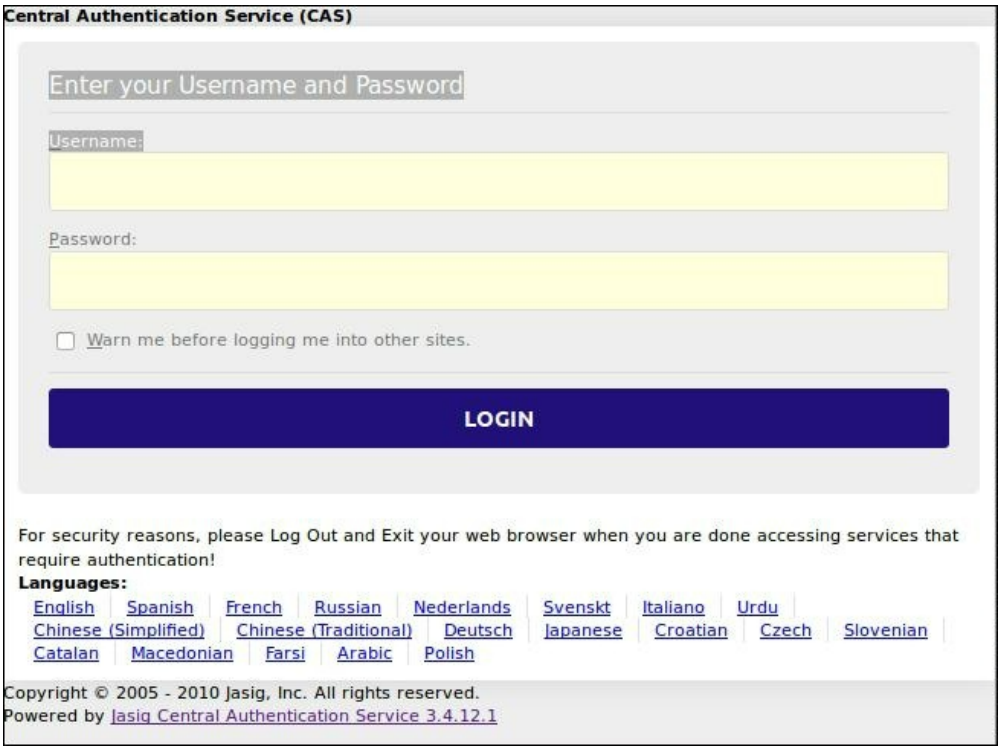
```
<bean
class="org.jasig.cas.adaptors.ldap.BindLdapAuthenticationHandler">
<property name="filter" value="uid=%u" />
<property name="searchBase"
value="ou=people,ou=mycompany,dc=nodomain" />
<property name="contextSource"
ref="contextSource" />
</bean>
</list>
</property>
</bean>
```

10. After `</bean>`, add an extra bean configuration, replacing the `password value` with 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>
```

Restart Tomcat.

11. Log in via `https://localhost:9443/cas/login` using the `tester1` account. If you see a page similar to the following screenshot, congratulations; you now have 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 the SSL to work, Tomcat needs to have a keystore with a private certificate. Use the `${user.home}` variable to point to the home directory of the Tomcat user, for example, `keystoreFile="${user.home}/.keystore" keystorePass="changeit"`.

The protocol you chose was TLS, which is a more modern and secure version of SSL. For further details, visit 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, so the `tomcat` alias uniquely identifies the 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 a password equal to username. This setup is for demonstration purposes. To replace or chain together handlers, you have to edit `webapps/cas/Web-`

`INF/deployerConfigContext.xml`. For more detail, refer to 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, `logs/catalina.out`. For example, a bad username or password will generate the following error:

```
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 that, by 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 following table:

Active Directory	This connects to your Windows infrastructure.
JAAS	This 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	This connects to your Enterprise directory services.
RADIUS	This connects to RADIUS.

Trusted	This 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	This connects to databases and there are even drivers for spreadsheets and LDAP.
Legacy	This supports the CAS2 protocol.
SPNEGO	Simple and Protected GSSAPI Negotiation Mechanism allows the CAS server to negotiate between protocols with a backend service. It potentially allows transitioning between backend services.
X.509 Certificates	This requires a trusted client certificate.

An alternative installation recipe using ESUP CAS

The ESUP consortium also provides a repackaged version of CAS that 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.

> **Note**
>
> 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. The relevant quote from the JASIG WIKI is:

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 for the CAS 3.4 line on the JASIG WIKI (https://wiki.jasig.org/):

- 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

- The *Enabling SSO in Jenkins* recipe

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 *Installing a CAS server* recipe and the Cas1 plugin (https://wiki.jenkins-ci.org/display/JENKINS/CAS1+Plugin).

> **Note**
>
> The Cas1 plugin has performed stably in the environments where the author has tried it. However, there is a second CAS plugin (https://wiki.jenkins-ci.org/display/JENKINS/CAS+Plugin) that is meant to replace the CAS1 plugin by providing new features along with the existing ones, for example, support for the CAS 2.0 protocol.
>
> Once comfortable with this recipe consider experimenting with the CAS plugin.

How to do it...

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

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

9. To configure your CAS setting, visit the Jenkins **Config Global Security** screen in the **Security Realm** section. Under **Access Control**, check the **CAS protocol version 1** checkbox and add the following details:

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

10. Log out of Jenkins.
11. Log in to Jenkins. You will now be redirected to the CAS server.
12. 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, then their **ROOT** certificates are most likely already in the default keystore (**cacerts**). This comes prepackaged with your Java installation. However, in the CAS installation recipe you created a self-signed certificate.

The configuration details for the CAS plugin are trivial. Note that you left the **Roles Validation Script** field 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

- The *Installing a CAS server* recipe

Exploring the OWASP Dependency-Check plugin

The OWASP Dependency-Check tool compares Java programs and JavaScript libraries to known threats in the CVE database (https://cve.mitre.org/). CVE is a dictionary of around 69,000 publicly known information security vulnerabilities and exposures. This process is a natural defense against the OWASP top 10 A9 - Using Known Vulnerable Components.

The CVE database is used as a standard for reporting issues by vulnerability scanners, allowing for a common language that tool users can use to compare the susceptibility of their software. The CVE reports include descriptions, where the issue was first reported, and the estimated danger level.

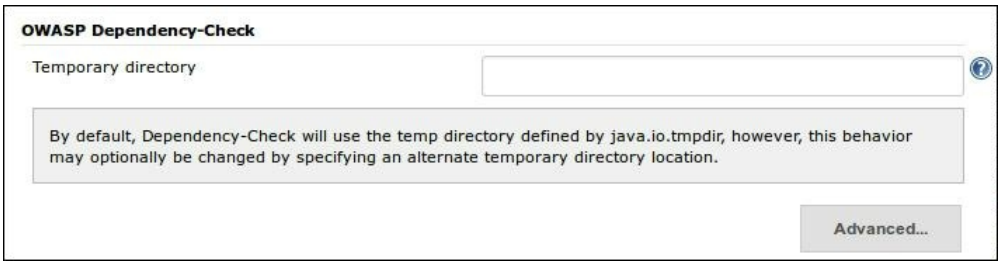
> **Note**
>
> The Dependency-Check tool is not always accurate as it needs to link the libraries with the vulnerabilities and sometimes it hard to for it to be accurate with library signatures. Therefore, you will need to review and filter actions based on the output.

Getting ready

Install the OWASP Dependency-Check plugin.

How to do it...

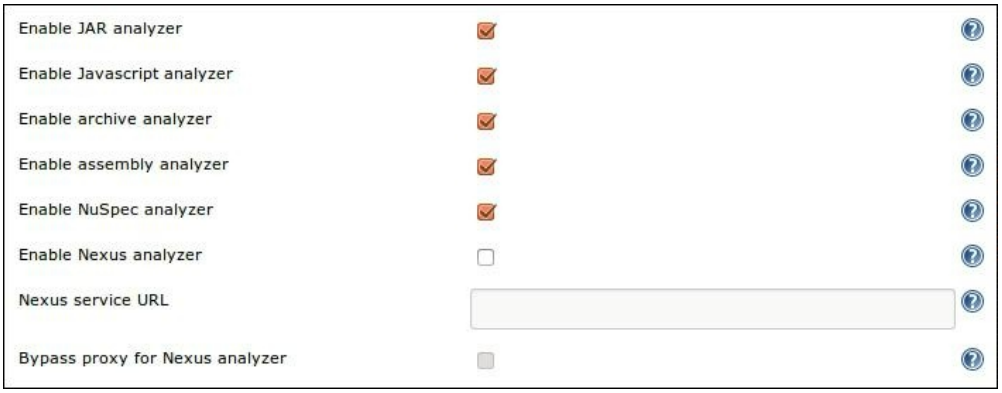
1. Click on the **Configure System** link in the **Manage Jenkins** page.
2. Review the **OWASP Dependency-Check** section, as shown in the following screenshot:



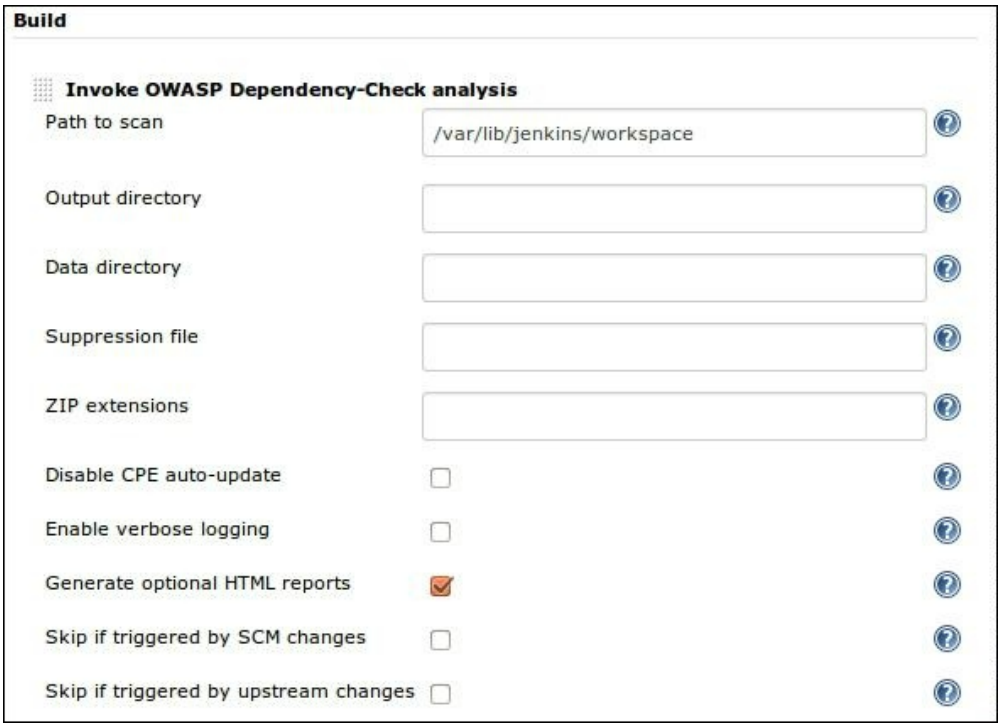
3. Press the **Advanced...** button and you'll get something similar to the following screenshot:



4. Press the **Analyzers...** button and you'll get something similar to the following screenshot:



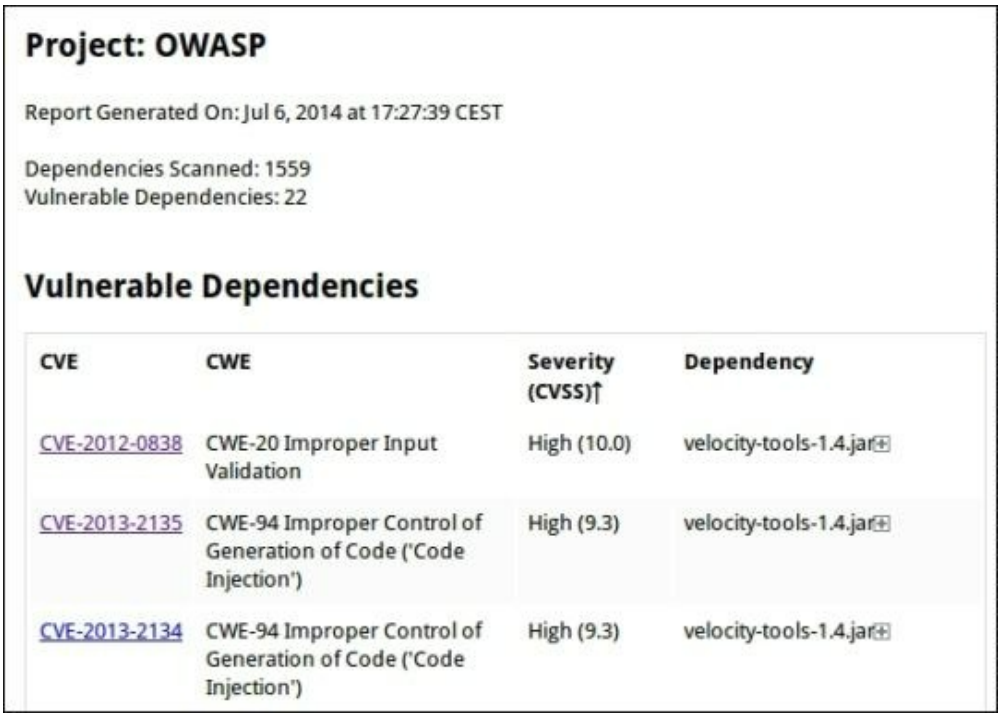
5. Visit `http://localhost:8080/view/All/newJob`.
6. Create a free-style job named `OWASP`.
7. Add a **Build** step for **Invoke OWASP Dependency-Check analysis**.
8. In the **Path to scan** field, type `/var/lib/jenkins/workspace` or the path to a project of your choice.
9. Make sure **Generate optional HTML reports** is the only checkbox ticked. Notice that you have not selected the **Disable CPE auto-update** checkbox:



10. Press **Save**.
11. Press the **Build Now** icon.
12. Once the job is finished, press the following workspace icon:



13. Click on the **dependency-check-vulnerabilty.html** link. Depending on the jobs run within the Jenkins workspace, you will see a report similar to the one shown in the following screenshot:



Project: OWASP

Report Generated On: Jul 6, 2014 at 17:27:39 CEST

Dependencies Scanned: 1559
Vulnerable Dependencies: 22

Vulnerable Dependencies

CVE	CWE	Severity (CVSS)↑	Dependency
CVE-2012-0838	CWE-20 Improper Input Validation	High (10.0)	velocity-tools-1.4.jar
CVE-2013-2135	CWE-94 Improper Control of Generation of Code ('Code Injection')	High (9.3)	velocity-tools-1.4.jar
CVE-2013-2134	CWE-94 Improper Control of Generation of Code ('Code Injection')	High (9.3)	velocity-tools-1.4.jar

How it works...

Installing the plugin automatically installs the OWASP Dependency-Check tool. You can find the tool's home page at https://www.owasp.org/index.php/OWASP_Dependency_Check:



Through the Jenkins interface, you configured the tool to look at every Java library `.jar` file underneath the Jenkins home directory. The scan will take time if your Jenkins server has many jobs configured.

The **Disable CPE auto-update** option was not selected. This was necessary, as the first time the tool runs it runs needs to download security information from an external CVE database. If you do not allow this to happen, then the report will contain no information. Although downloading the newest threat information takes time it is the safest approach to finding new issues in time.

There's more...

At the time of writing, the dependency plugin options in Jenkins lag behind the options available for the command-line tool. To give you an idea of what changes you can expect in the plugin, download the command-line tool from https://www.owasp.org/index.php/OWASP_Dependency_Check.

Once the tool is downloaded and extracted, run the advanced help as follows:

```
sh dependency-check.sh –advancedHelp
```

Your output will be similar to the following screenshot:

```
usage: Dependency-Check Core [-a <name>] [--advancedHelp] [-c <timeout>]
       [--connectionString <connStr>] [-d <path>] [--dbDriverName
       <driver>] [--dbDriverPath <path>] [--dbPassword <password>]
       [--dbUser <user>] [--disableArchive] [--disableAssembly]
       [--disableJar] [--disableNexus] [--disableNuspec] [--exclude
       <pattern>] [-f <format>] [-h] [-l <file>] [--mono <path>] [-n]
       [--nexus <url>] [--nexusUsesProxy <true/false>] [-o <path>] [-P
       <file>] [--proxypass <pass>] [--proxyport <port>] [--proxyserver
       <server>] [--proxyuser <user>] [-s <path>] [--suppression <file>]
       [-v] [--zipExtensions <extensions>]

Dependency-Check Core can be used to identify if there are any known CVE
vulnerabilities in libraries utilized by an application. Dependency-Check
Core will automatically update required data from the Internet, such as
the CVE and CPE data files from nvd.nist.gov.
 -a,--app <name>                    The name of the application being
                                    scanned. This is a required argument.
    --advancedHelp                  Print the advanced help message.
```

The text is followed by a brief set of descriptions for all the options, such as:

```
-n,--
noupdate              Disables the automatic
updating of the CPE data.
```

The options are or will be mirrored in the Jenkins GUI configuration.

> **Note**
>
> You can find the most up-to-date code source for the tool at GitHub (https://github.com/jeremylong/DependencyCheck).

See also

- The *Reporting overall storage usage* recipe in Chapter 1, *Maintaining Jenkins*
- The *Adding a job to warn of storage use violations through log parsing* recipe in Chapter 1, *Maintaining Jenkins*

Chapter 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 Ant through Groovy in Maven
- Failing Jenkins jobs based on JSP syntax errors
- Configuring Jetty for integration tests
- Looking at license violations with Rat
- Reviewing license violations from within Maven
- Exposing information through build descriptions
- Reacting to generated data with the groovy-postbuild plugin
- Remotely triggering jobs through the Jenkins API
- Adaptive site generation

Introduction

This chapter reviews the relationship between Jenkins and Maven builds, and there is 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 Jenkins plugins and 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 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 `pom.xml`. You will have to judge the balance; it is easy to have "feature creep". The UI is easier to configure than writing a long `pom.xml` file. 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 the Jenkins Publish Over SSH plugin (https://wiki.jenkins-ci.org/display/JENKINS/Publish+Over+SSH+Plugin). You can configure transferring files or add a section to 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 Maven plugin to use at times feels like magic. Jenkins plugins simplify details.

Maven uses profiles so that you can use different configurations in your projects—for example, server names for development, acceptance, or production. This also allows you to update version numbers for plugins, easing maintenance effort. For more information, visit http://maven.apache.org/guides/introduction/introduction-to-profiles.html.

Later in the chapter, you will be given the chance to run Groovy scripts with AntBuilder. Each approach is viable; use depends more on your preferences than one clear 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 certain criteria, placing an icon by a successful build as shown in the following screenshot:



You can use this feature to signal, for example, to the QA team that they need to test the build, or for system administrators to pick up artifacts and deploy. Other plugins can also be triggered by promotion (for example, when a developer signs off on a build using the promotions plugin), including the SSH plugin. However, Maven is not aware of the promotion mechanism. As Jenkins evolves, expect more plugin interrelationships.

Jenkins is well versed in the 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/guides/mini/guide-multiple-modules.html) or a thicker `pom.xml` file with different goals ready to be called across jobs. **Keep It Simple Stupid** (**KISS**) biases the decision towards a larger single file.

Jenkins is an enterprise-friendly agnostic

Jenkins is technology-agnostic and can glue together project technologies across the organization, development teams, and software position in their lifecycle. Jenkins lets you run scripting languages of choice, makes it easy to pull in source code using Git, subversion, CVS, and a number of other version control systems. If Jenkins is not compatible, developers with a little practice can write their own integration.

In this book, you will see both subversion and GIT projects mentioned. This represents a realistic mix. Many consider Git more versatile than subversion. Feel free to use Git as your repository of choice for the examples in this book. Designed in from the start, Jenkins makes it easy for you to choose between the different revision control systems.

Note

If you look at the relative use of Git and subversion for a representative collection from Ohoh in early 2014, for Git there were 247,103 repositories (37 percent of the total) and subversion had 324,895 repositories (48 percent of the total).

Typical enterprises lag behind when using the most modern offerings because of their resistance to changing their working processes. Therefore, expect this category of businesses to have a higher

percentage of subversion repositories compared to smaller organizations.

A pom.xml template

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 book's website.

The examples were tested against Maven 3.2.1, though the examples should work with the latest version of Maven.

From the main Jenkins configuration screen (`http://localhost:8080/configure`) under the **Maven** section, you will need to install this version, giving it the label `3.2.1`.

To generate a basic template for a Maven project, you have two choices. You can create a project via 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 here:

```
<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 default values that are 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 based on convention. For more detail, visit http://maven.apache.org/guides/mini/guide-naming-conventions.html.

Maven changes

Maven 2 has reached the end of its life (http://maven.apache.org/maven-2.x-eol.html) and the development team has stopped supporting it. You cannot expect prompt removal of newly discovered bugs. At the time of writing this book, Maven 4 is in planning and has not been released.

If you have Maven 2 installed as a package and wish to upgrade to Maven 3, then you will need to install the Maven package. To swap between the alternative Maven versions, you will need to run the following Ubuntu command:

```
sudo update-alternatives --config mvn
```

Setting up a File System SCM

In previous chapters, you used recipes that copied files into the workspace. This is easy to explain, but 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), as this 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 them. 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 standard result types such as JUnit, JMeter, FindBugs, and NCSS. There is also a SonarQube plugin (http://docs.codehaus.org/display/SONAR/Jenkins+Plugin) that supports pushing data to SonarQube (http://www.sonarsource.org/). SonarQube 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.

Let's assume 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 changes in the code are improving or degrading the performance. The data is faked: a simple Perl script will generate random results.

Getting ready

In the plugin **Manager** section of Jenkins (`http://localhost:8080/pluginManager/available`), install the plot plugin. Create a directory named `ch3.building_software/plotting`.

How to do it...

1. Create the `ch3.building_software/plotting/hit_and_miss.pl` file and add the following lines of code:

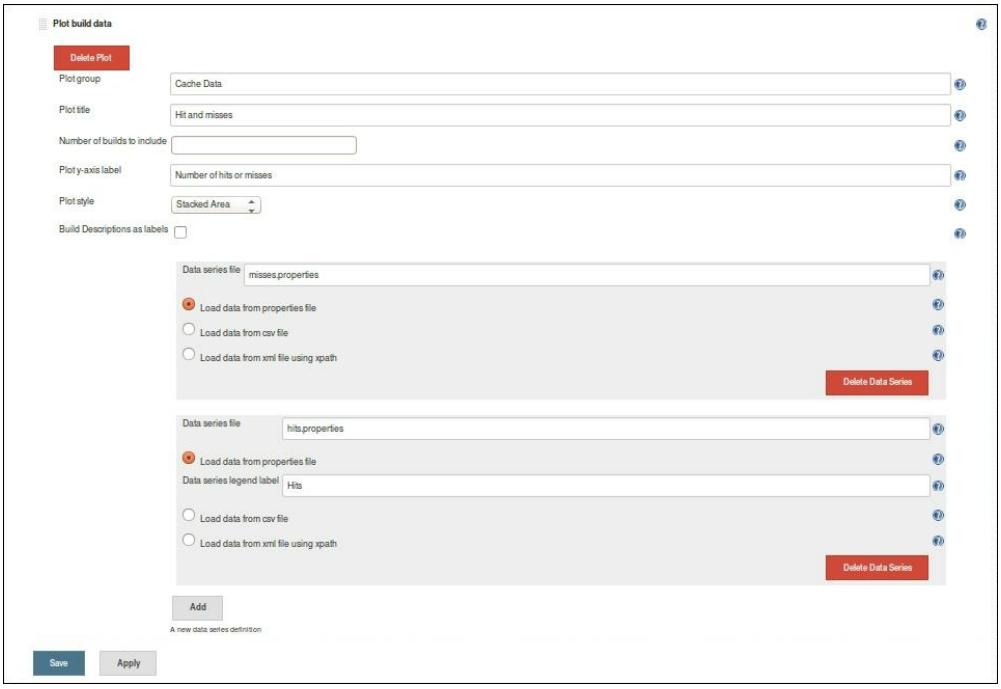
```
#!/usr/bin/perl
my $workspace = $ENV{'WORKSPACE'};

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 free-style job with **Job name** as `ch3.plotting`.

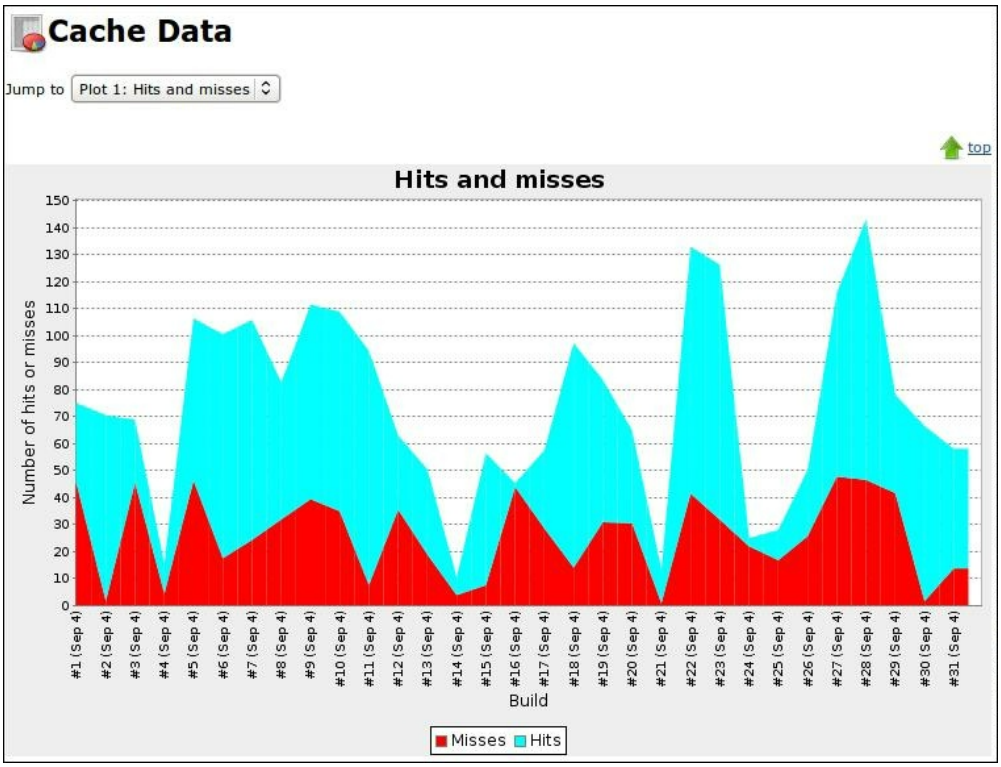
3. In the **Source Code Management** section, check **File System**, and add a fully qualified path of your plotting directory, such as `/var/lib/jenkins/cookbook/ch3.building_software/plotting`, in the **Path** field.
4. In the **Build** section, select **Add a build step** for **Execute Shell** or in the case of a Windows system, select the **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**: `Hit and misses`
    - **Plot y-axis label**: `Number of hits or misses`
    - **Plot style**: **Stacked Area**

8. Type `misses.properties` in **Data series file** and type `Misses` in the **Data series legend** label.
9. Type `hits.properties` in **Data series file** and type `Hits` in the **Data series legend** label.
10. At the bottom of the configuration page, click on the **Save** button, as shown in the following screenshot:



11. Run the job multiple times.
12. Review the **Plot** link and you will see something similar to the following screenshot:



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 `YVALUE` property.

The two property files are read by the plot plugin. The plugin keeps track of the history, their values displayed in a trend graph. You will have to experiment with the different graph 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 can choose from 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 `http://host/job/JobName/plot/getPlot?index=n&width=x&height=y`.

The `[Width]` and `[height]` parameters define the size of the plot. `n` is an index number pointing to a specific plot. If you have only one plot, then `n=0`. If you have two plots configured, then `n` could be either 0 or 1. To discover the index, visit the plot's link and examine the **Jump to** drop-down menu and take one from the highest **Plot** number, as shown in the following screenshot:



To generate a graph in PNG format of dimensions 800 x 600 based on the job in this recipe, you would use a URL similar to `localhost:8080/job/ch3.plotting/plot/getPlot?index=0&width=800&height=600`.

Tip

To download the image without logging in yourself, you can use the scriptable authentication method mentioned in the *Remotely triggering jobs through the Jenkins API* recipe.

See also

- The *Running Groovy scripts through Maven* recipe
- The *Adaptive site generation* recipe
- The *Remotely triggering jobs through the Jenkins API* recipe

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. For more detail, refer to the *Maven phases* section in this recipe.

Maven can execute the Groovy source code from within the build file, at another file location, or from a remote web server.

> **Note**
>
> An alternative plugin is GMavenPlus. For a comparison between the GMaven and GMavenPlus plugins, visit http://docs.codehaus.org/display/GMAVENPLUS/Choosing+Your+Build+Tool.
>
> You will find instructions on how to configure the plugin at http://groovy.github.io/GMavenPlus/index.html.

Getting ready

Create a directory named `ch3.building_software/running_groovy`.

> **Tip**
>
> **Maintainability of scripting**
>
> For later reuse, consider centralizing your Groovy code outside the build files.

How to do it...

1. Add the following lines of code just before the `</project>` tag within your template file (mentioned in the introduction). Make 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 **Job name** as `ch3.groovy_verify`.
3. In the **Source Code Management** section, check **File System** and a fully qualified path of your plotting directory, such as `/var/lib/jenkins/cookbook/ch3.building_software/running_groovy`, in the **Path** field.
4. In the **Build** section, select **Add a build step** for **Invoke top-level Maven targets**. In the newly expanded section, add the following details:

    - **Maven Version**: `3.2.1`
    - **Goals**: `verify`

5. Run the job. If your system is on a *NIX box, you'll get the following output:

```
OS Type Linux
Output:
total 12
```

```
-
rwxrwxrwx 1 jenkins jenkins 1165 2011-09-02
11:03 pom.xml
drwxrwxrwx 1 jenkins jenkins 3120 2014-09-02
11:03 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 `verify` phase is the trigger point.

To enable the Groovy plugin to find imported classes outside its core features, you will need to add an element in the `<classpath>` tag. 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 (https://commons.apache.org/proper/commons-lang/javadocs/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. Most of the time, 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 if your tests find themselves on the wrong node.

Tip

Once you are satisfied with your Groovy code, consider compiling the code into the underlying Java byte code. You can find full instructions at http://docs.codehaus.org/display/GMAVEN/Building+Groovy+Projects.

There's more...

Here are a number of tips you might find useful.

Keeping track of warnings

It is important to review your logfiles, not only on failure, but also for the warnings. 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 servers and the default encoding on the server is different, the results of the copying may also be different. For consistency, it is better to enforce a specific coding in the file by adding the following lines just before the `<build>` tag:

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

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 way is to point to the filesystem, as follows:

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

The other approach is to connect to a web server through a URL as follows:

```
<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. For more information, visit http://Maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html.

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

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 the load and aid in diagnosing issues.

You can find the XML wiring the lifecycle code at https://git-wip-us.apache.org/repos/asf?p=maven.git;a=blob;f=maven-core/src/main/resources/META-INF/plexus/components.xml.

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

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

Maven plugins bind to particular phases. For site generation, the `<reporting>` tag surrounds the majority of 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. In general, Jenkins plugins do not perform the tests; they consume the results. There are exceptions such as the Sloccount plugin (https://wiki.jenkins-ci.org/display/JENKINS/SLOCCount+Plugin) and 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*.

Note

To install the sloccount plugin, you will need first to install the static analysis utilities plugin.

The Groovy plugin is useful in all 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.

Tip

The differences between Maven versions

To upgrade between Maven 2 and Maven 3 projects, you need to know the differences and incompatibilities. There are a number of differences, especially around site generation. They are summarized at https://cwiki.apache.org/confluence/display/MAVEN/Maven+3.x+Compatibility+Notes.

You will find the plugin compatibility list at https://cwiki.apache.org/confluence/display/MAVEN/Maven+3.x+Plugin+Compatibility+Matrix.

See also

- The *Running Ant through Groovy in Maven* recipe
- The *Reacting to generated data with the groovy-postbuild plugin* recipe
- The *Adaptive site generation* recipe

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.

In a typical development/acceptance/production environment, you will want to keep the same `pom.xml` files, but pass different configuration. One example is the extension names of property files such as `.dev`, `.acc`, and `.prd`. You would want to fail the build if critical configuration values are missing due to human error.

Jenkins has a number of plugins for passing information to builds, including the 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 (https://wiki.jenkins-ci.org/display/JENKINS/EnvInject+Plugin). Create the recipe directory named `ch3.building_software/environment`.

How to do it...

1. Create a `pom.xml` file that is readable by Jenkins with the following lines of 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.jenkins.builds.properties</artifactId>
<version>1.0-SNAPSHOT</version>
<name>${name.from.jenkins}</name>
<properties><project.build.sourceEncoding>UTF8</project.build.sourcel
</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. Then, add the following lines of code in the `my.properties` file:

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

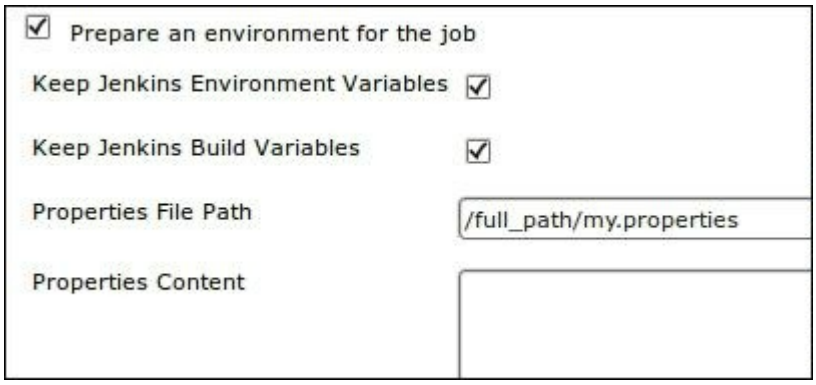
3. Create a blank free-style job with the **Job name** as `ch3.environment`.
4. In the **Source Code Management** section, check **File System** and add a fully qualified path of your directory, such as `/var/lib/jenkins/cookbook/ch3.building_software/environment`, in the **Path** field.
5. In the **Build** section, select **Add a build step** for **Invoke top-level Maven targets**. In the newly expanded section, add the following details:

    - **Maven Version**: `3.2.1`
    - **Goals**: `verify`

6. Click on the **Advanced** button and type `longname=SuperGood` in **Properties**.

7. Inject the values in `my.properties` by selecting the **Prepare an environment for the job** checkbox (near the top of the job configuration page).
8. For the **Properties File Path**, add `/full_path/my.properties`; for example `/home/var/lib/cookbook/ch3.building_software/environment/my.properties`.

   The preceding option is depicted in the following screenshot:



9. Run the job. The build will fail:

```
----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 a build.

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

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

The Groovy code can see instances of the `org.apache.Maven.model.Model` project and the `org.apache.Maven.execution.MavenSession` class. The project instance is a model of the XML configuration that you can programmatically access. You can get the `longname` property 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 `System.getProperty("longname")` call. However, you cannot get to the property by using the `System.getenv()` environment call.

It is well worth learning the various options:

- **Keep Jenkins Environment Variables** and **Keep Jenkins Build Variables**: Both these options affect the Jenkins-related variables that 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**: This option 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. The variable contains information about the event that 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, as follows:

```
<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 source code. 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 those 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 build time, you can choose your property files by selecting from a choice of property file locations.

See also

- The *Running Ant through Groovy in Maven* recipe

Running Ant 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 who might 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 AntRun 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 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, as a feature-rich programming language, is much more expressive.

This recipe details how you can run two Maven POMs involving Groovy and Ant. The first POM shows you how to run the simplest of Ant tasks within Groovy and the second performs an Ant-contrib task to securely copy files from a large number of computers.

Getting ready

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

How to do it...

1. Create a template file and name 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 -f pom_ant_simple.xml`. Review the output (note that there are no warnings about empty JAR files):



5. Run `mvn verify -f pom_ant_simple.xml`. Review the output; it should look similar to the following screenshot:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------

Results :

Tests run: 0, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] --- gmaven-plugin:1.3:execute (run-myGroovy-test) @ ch3.jenkins.builds ---
     [echo]
     [echo]
     [echo] Tested ----> With Groovy
[INFO]
[INFO] --- maven-jar-plugin:2.2:jar (default-jar) @ ch3.jenkins.builds ---
[WARNING] JAR will be empty - no content was marked for inclusion!
[INFO] Building jar: /home/user/antbuilder/target/ch3.jenkins.builds-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- gmaven-plugin:1.3:execute (run-myGroovy-verify) @ ch3.jenkins.builds ---
     [echo]
     [echo]
     [echo] Verified at Fri Nov 21 17:04:59 CET 2014
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3.601s
[INFO] Finished at: Fri Nov 21 17:04:59 CET 2014
[INFO] Final Memory: 14M/177M
[INFO] ------------------------------------------------------------------------
```

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>
```

This is only representative code, unless you have set it up to point to real files on real servers:

```
mvn verify -f pom_ant_simple.xml will fail
```

How it works...

Groovy runs basic Ant tasks without the need for 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:

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

You configured the `pom.xml` file 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 file. 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 many times across many servers. The script first asks for the username and password, avoiding storage on your filesystem or your revision control system. The Groovy script expects you to inject the `host`, `full_path_to_location`, and `myfile` variables.

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

There's more...

Another example of running Ant through Groovy is the creation of custom property files on the fly. This 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 lines of code creates a `value.properties` file 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.

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

See also

- The *Running Groovy scripts through Maven* recipe

Failing Jenkins jobs based on JSP syntax errors

JavaServer Pages (**JSP**) (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 agile programming practices, but the risk is that developers make messy, hard-to-read JSP code that is difficult to maintain. It would be nice if Jenkins could display metrics about the code to defend quality.

JSP pages are compiled on the fly the first time 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 compiled source code is that you can run a number of statistical code review tools over the code base and obtain testability metrics. This generates testing data ready for Jenkins plugins to display.

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

> **Note**
>
> The JSP mentioned in this recipe is deliberately insecure and hence ready for testing later in this book.

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

Getting ready

Create a 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
-DarchetypeArtifactId=maven-archetype-webapp
```

2. Enter the following values:

   - **groupId**: `ch3.packt.builds`
   - **artifactId**: `jsp_example`
   - **version**: `1.0-SNAPSHOT`
   - **package**: `ch3.packt.builds`

3. Click on **Enter to confirm the values**.
4. Edit the `jsp_example/pom.xml` file by 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>
<version>2.4</version>
<configuration>
<webXml>${basedir}/target/web.xml</webXml>
</configuration>
</plugin>
</plugins>
</build>
```

5. Replace the code snippet in the `src/main/webapp/index.jsp` file with the following lines of 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>
```

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

```
[ERROR] Failed to execute goal
org.mortbay.jetty:maven-jetty-jspc-
plugin:6.1.14:jspc (jspc) on project
jsp_example: Failure processing jsps -> [Help
1]
```

How it works...

You created a template project using an archetype.

The Maven plugin, upon seeing the `index.jsp` page, compiles it into a class with the `jsp.index_jsp` name, 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`. Let's take a look at the following 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>
```

> **Tip**
>
> The list of archetypes is increasing over time. You can find the full list at http://maven-repository.com/archetypes. If you are running Ubuntu, you will find a local XML catalog listing all the archetypes named `archetype-catalog.xml` in the `~/.m2` directory.

There's more...

Here are a few things you should consider.

Different server types

By default, the Jetty Maven plugin (Version 6.1.14) loads JSP 2.1 libraries with JDK 15. 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 at `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 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, visit http://wiki.eclipse.org/Jetty/Feature/Jetty_Maven_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 JSP pages, then your pages may fail to compile. This is because, at the time of writing, the default compiler does not like the meta-information mentioned before the `<html>` tag as follows:

```
<
%@ 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">
```

As the meta-information follows the JSP specification, it is likely that later the JSP compiler will accept the information. Until that day, simply remove the lines before compiling or change the JSP compiler that you use.

See also

- The *Configuring Jetty for integration tests* recipe

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:

- **Deploy your artifacts such as WAR files to a live server**: This can be done using the Maven Wagon plugin (http://mojo.codehaus.org/wagon-maven-plugin/) or through a Jenkins plugin such as the aptly-named Deploy plugin (https://wiki.jenkins-ci.org/display/JENKINS/Deploy+Plugin).
- **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 potentially scarce resources. This will limit the number of parallel executors Jenkins can run, decreasing the maximum throughput of jobs. This should be delegated to dedicated slave nodes set up for this purpose.

This recipe runs the web application developed in the *Failing Jenkins jobs based on JSP syntax errors* recipe, 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 *Failing Jenkins jobs based on JSP syntax errors* recipe 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>
<version>1.5</version>
<executions>
<execution>
<phase>generate-resources</phase>
<id>clean</id>
```

```
<goals>
<goal>clean</goal>
</goals>
</execution>
<execution>
<phase>generate-resources</phase>
<id>generateKeyPair</id>
<goals>
<goal>generateKeyPair</goal>
</goals>
</execution>
</executions>
<configuration>
<keystore>${project.build.directory}/jetty-
ssl.keystore</keystore>
<dname>cn=HOSTNAME</dname>
<keypass>jetty8</keypass>
<storepass>jetty8</storepass>
<alias>jetty8</alias>
<keyalg>RSA</keyalg>
</configuration>
</plugin>
<plugin>
<groupId>org.mortbay.jetty</groupId>
<artifactId>jetty-maven-plugin</artifactId>
<version>8.1.16.v20140903</version>
<configuration>
<war>${basedir}/target/jsp_example.war</war>
<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 `mvn jetty:run` command. You will now see console output from the Jetty server starting up.
3. Using a web browser, visit the `https://localhost:9443` location. 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's `pre-integration-test` phase and later stopped in the Maven's `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 Java `keystore` with a known password and alias. The key encryption is set to RSA. 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 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 keystore.

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

> ## Note
>
> 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 the certificate as a security exception in their web browser. It is better to use certificates from well-known authorities. You

can achieve this with this recipe by removing the key generation and pointing the `keystore` tag to a known file location.

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 Maven, this defends against unwanted changes. For example, at the time of writing this book, the Jetty plugin used in this recipe is held at Version 8.1.16.v20140903. As you can see from the bug report at http://jira.codehaus.org/browse/JETTY-1071, configuration details have changed over versions.

Another advantage is that if the plugin version is too old, 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

- The *Failing Jenkins jobs based on JSP syntax errors* recipe
- The *Adaptive site generation* recipe

Looking at license violations with Rat

This recipe describes how to search any job in Jenkins for license violations. It is based on the Apache Rat project (http://creadur.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 such as the log-parser plugin to process the information.

Getting ready

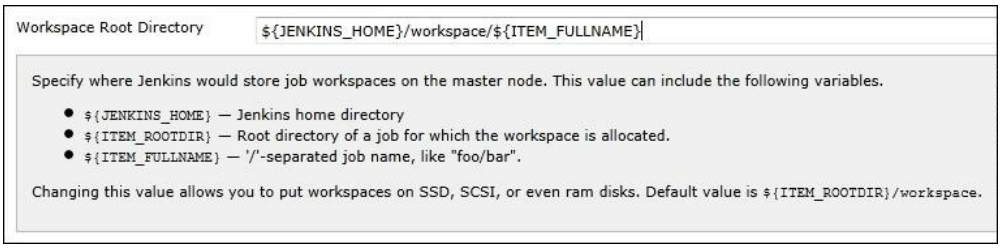
Create the `License_Check` directory underneath the Jenkins home directory (`/var/lib/jenkins`). Log in to Jenkins.

How to do it...

1. Create a Maven job named `License_Check`.
2. Under the **Source Code Management** section, check **Subversion**.
3. Fill in `http://svn.apache.org/repos/asf/creadur/rat/trunk/` for **Modules, Repository URL**.
4. Set **Check-out Strategy** to **Use 'svn update' as much as possible**.
5. Under the **Build** section, add a clean package to **Goals and options**.
6. Under the **Post steps** section, check **Run only if build succeeds**.
7. Add **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 if necessary, replacing the JAR version number:

```
java
-jar ./apache-rat/target/apache-rat-0.12-
SNAPSHOT.jar --help
java
-jar ./apache-rat/target/apache-rat-0.12-
SNAPSHOT.jar -d
${JENKINS_HOME}/workspace/License_Check/  -e
'*.js'  -e '*target*'
```

8. Click on the **Save** button and 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 **Home Directory**, click on the **Advanced** button. The **Workspace Root Directory** values become visible, as shown in the following screenshot:



How it works...

The Rat source code is compiled and then run twice—the first time to print the help out and the second time to check license headers.

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

The `-d` option tells the application in which directory your source code can be found. In this example, you have used the `${JENKINS_HOME}` variable 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 generated files under the target directory. In a complex project, expect a long list of exclusions.

> **Note**
>
> Even if the directory to check does not exist, then the build will still succeed with an error reported as follows:
>
> ```
> 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 (http://code.mycila.com/license-maven-plugin/). You

can use it to keep your source code license headers up to date. To add/update the source code with the `src/etc/header.txt` license, add the following XML fragment to your build section:

```
<plugin>
<groupId>com.mycila.maven-license-plugin</groupId>
<artifactId>maven-license-plugin</artifactId>
<version>2.6</version>
<configuration>
<header>src/etc/header.txt</header>
</configuration>
</plugin>
```

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

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

```
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

- The *Reviewing license violations from within Maven* recipe
- The *Reacting to generated data with the groovy-postbuild plugin* recipe

Reviewing license violations from within Maven

In this recipe, you will run Rat through Maven. It will then check for license violations in your source code.

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>0.11-SNAPSHOT</version>
<executions><execution>
<phase>verify</phase>
<goals><goal>check</goal></goals>
</execution></executions><configuration>
<excludeSubProjects>false</excludeSubProjects><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 project with the **Project name** as `ch3.BasicLTI_license`.
5. Under the **Source Code Management** section, tick **Subversion** with **URL Repository** as `https://source.sakaiproject.org/svn/basiclti/trunk`.

> **Note**
>
> Do not spam the subversion repository. Double-check that there are no build triggers activated.

6. Under the **Build** section set, add the following details:
   - **Root POM**: `pom.xml`
   - **Goals and options**: `clean`
7. Under the **Pre Steps** section, invoke Inject environment variables and add the following to the property's context:

```
rat.basedir=/var/lib/Jenkins/workspace/ch3.BasicLTI_license
```

8. Under the **Post Steps** section, invoke the top-level Maven targets:
   - **Maven Version**: `3.2.1`
   - **Goals**: `verify`
9. Click on the **Advanced** button.
10. In the expanded section, set the **POM** section to the full path to your Rat's POM file, for example, `/var/lib/cookbook/ch3.building_software/license_maven /pom.xml`.
11. Under the **Post Steps** section, add a copy command to move the report into your workspace (such as `cp /var/lib/cookbook/ch3.building_software/license_maven/target/rat.txt ${WORKSPACE}`) and **Execute Shell**.
12. Run the job. You can now visit the workspace and view `./target/rat.txt`. The file should look similar to the following screenshot:

```
*****************************************************
Summary
-------
Generated at: 2014-11-22T11:56:57+01:00
Notes: 0
Binaries: 0
Archives: 0
Standards: 251

Apache Licensed: 35
Generated Documents: 0

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

216 Unknown Licenses

*******************************

Unapproved licenses:
```

How it works...

You have pulled source code from an open source project; in this case, from one of the subversion and Git repositories of the Apereo Foundation (https://source.sakaiproject.org/svn/).

> **Note**
>
> In 2013, the Sakai Foundation (www.sakaiproject.org) merged with JASIG (www.jasig.org) to become the Apereo Foundation (www.apereo.org).
>
> Sakai is a **Learning Management System** (**LMS**) used by many millions of students daily. The Apereo Foundation represents over 100 organizations, mostly universities.

The source code includes different licenses that are checked by the Rat 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.

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

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

> **Note**
>
> For information on customizing Rat for specific license types, visit:
>
> http://creadur.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 antipatterns

There were multiple approaches to configuring the Jenkins job. You can avoid copying the Rat report file by fixing its location in the Maven plugins configuration. This has the advantage of avoiding a copying action. You can also use the multiple-scms 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 Rat job at the source code's 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. However, for the most maintainable code, it is much better to use fixed versions artifact.

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

See also

- The *Looking at license violations with Rat* recipe

- The *Reacting to generated data with the groovy-postbuild plugin* recipe

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 builds 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 details immediately in the trend report without needing to review all the build results separately.

The setter plugin uses regex expressions to scrape the descriptions. This recipe shows you how to do this.

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 project with the **Job name** as `ch3.descriptions`.
5. In the **Source Code Management** section, check **File System** and add a fully qualified path of your directory, such as `/var/lib/Jenkins/cookbook/ch3.building_software/description`, in the **Path** field.
6. Tick **Set build description** and add the values shown in the following screenshot:



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 `MySever issue` pattern or prints the output to the build with the `Great stuff happens because` pattern:

```
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 build success, it looks for the `Great stuff happens because: (.*)` pattern.

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

> **Tip**
>
> 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 generates `"\1"` equal to `fred` and `"\2"` equal to `happy`.

There's more...

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

See also

- The *Reacting to generated data with the groovy-postbuild plugin* recipe

Reacting to generated data with the groovy-postbuild plugin

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

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

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

Getting ready

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

How to do it...

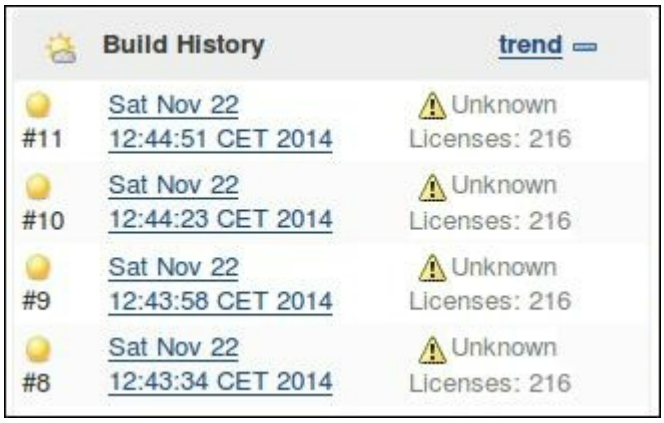
1. Update the `pom.xml` file by adding the following XML fragment just before the `</plugins>` tag:

```
<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 **If the script fails** select box is set to **Do Nothing**.
4. Click on **Save**.
5. Run the job a number of times. In **Build History**, you will see results similar to the following screenshot:



6. Clicking on the newest build link displays the build page with summary information about unknown licenses, as shown in the following screenshot:



How it works...

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

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

- `manager.build.logFile`: This gets the logfile, which now includes the licensing information.
- `manager.getMatcher`: This checks the logfile 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 `Unknown Licenses` pattern 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)`: This adds a warning badge to the build history box and `title` is used as the text that is displayed as the mouse hovers over the icon.
- `manager.addShortText`: This 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.

There's more...

Pulling information into a report by searching for regular patterns is called scraping. The stability of scraping relies on a consistent pattern being generated in the Rat report. If you change the version of the Rat plugin, the

pattern might change and break your report. When possible, it is more maintainable for you to use stable data sources, such as XML files, that have a well-defined syntax.

See also

- The *Exposing information through build descriptions* recipe
- The *Improving security via small configuration changes* recipe in Chapter 2, *Enhancing Security*

Remotely triggering jobs through the Jenkins API

Jenkins has a remote API which allows you to enable, disable, run, and delete jobs; it also lets you change the configuration. The API is increasing with each 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 `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 can log in as a user. It also assumes you have a modern version of `wget` (http://www.gnu.org/software/wget/) installed.

How to do it...

1. Create a free-style project with **Project name** as `ch3.RunMe`.
2. Check **This Build is parameterized**, select **String Parameter**, and add the following details:
    - **Name**: `myvariable`
    - **Default Value**: `Default`
    - **Description**: `This is my example variable`
3. Under the **Build Triggers** section, check **Trigger builds remotely** (For example, from scripts).
4. In the **Authentication Token** textbox, add `changeme`.
5. Click on the **Save** button.
6. Click on the **Build with Parameters** link.
7. You will be asked for the variable named `myvariable`. Click on **Build**.
8. Visit your personal configuration page, such as `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. Copy the token to `apiToken`.
11. From a terminal console, run `wget` to log in and run the job remotely:

```
wget --auth-no-challenge --http-user=username
--http-
password=apiToken
```

```
http://localhost:8080/job/ch3.RunMe/build?
token=changeme
```

12. Check the Jenkins job to verify that it has not run and returns a `405` HTTP status code:

```
Resolving localhost (localhost)...
127.0.0.1Connecting to localhost
(localhost)|127.0.0.1|:8080... connected.
HTTP request sent, awaiting response... 405
Method Not Allowed
2014-08-
14 15:08:43 ERROR 405: Method Not Allowed.
```

13. From a terminal console, run `wget` to log in and run the job returning a `201` HTTP status code:

```
wget --auth-no-challenge --http-user=username
--http-
password=apiToken
http://localhost:8080/job/ch3.RunMe/buildWithParameters?
token=changeme\&myvariable='Hello World'
Connecting to localhost
(localhost)|127.0.0.1|:8080... connected.
HTTP request sent, awaiting response... 201
Created
```

> **Note**
>
> HTTP 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 to be the same as passwords.

There were two remote methods calls. The first is build, which used to run the build without passing parameters. The method is currently not accepted. The second working method 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 to write some tricky Groovy code. We have chosen simplicity and OS dependence

for the sake of a short recipe. Running an executable risks 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.

For more details, visit https://wiki.jenkins-ci.org/display/JENKINS/Authenticating+scripted+clients.

> **Note**
>
> You can find the equivalent Java code at the following URL:
>
> https://wiki.jenkins-ci.org/display/JENKINS/Remote+access+API.

There's more...

Here are a few things you should consider.

Running jobs from within Maven

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

```
<build>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-antrun-plugin</artifactId>
<version>1.7</version>
<executions><execution>
<phase>compile</phase>
<configuration>
<tasks>
<exec executable="wget">
<arg line="--auth-no-challenge
--http-
user=username --http-password=apiToken
http://localhost:8080/job/ch3.RunMe/build?
token=changeme" />
</exec>
</tasks>
</configuration>
<goals><goal>run</goal></goals>
</execution></executions>
</plugin>
</build>
```

You can use the exec-maven plugin for the same purpose as the maven-ant plugin. For more details, visit http://mojo.codehaus.org/exec-maven-plugin/.

Remotely generating jobs

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

See also

- The *Running Ant through Groovy in Maven* recipe

Adaptive site generation

Jenkins is a great communicator. It can consume the results of 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.

A Jenkins Maven software project job run notes when a site is generated, and creates a shortcut icon in the jobs home page. This is a very visible icon that you can link with content:



You can gain fine-grained control of Maven site generation through triggering 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 *Reporting alternative code metrics in Jenkins* recipe describes how you can plot custom results in Jenkins, enhancing the user experience further.

> **Note**
>
> This recipe works in Version 2.2.1 of Maven or earlier. Maven 3 has a slightly different approach to site generation.
>
> To enforce a minimum Maven version from within your `pom.xml` file, you will need to add `<prerequisites><maven>2.2.1</maven></prerequisites>`.

Getting ready

Create a directory named `ch3.building_software/site`. 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, as mentioned in this recipe, into the Jenkins workspace. This is used to copy sensitive configuration files with passwords into a project, which you do not want in your Revision Control System.

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 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 `site_xml.Groovy` file within the same directory as your `pom.xml` file with the following lines of code:

```
def site= new File('./src/site')
site.mkdirs()
defs xml=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 `site.Groovy` file within the same directory as your `pom.xml` file with the following lines of code:

```
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 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, select **Copy data to workspace**.
7. Add whichever directory you have placed the files in (mentioned in this recipe) to the **Path to folder** field.
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 submenu 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.

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

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

The `site.Groovy` file 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 `for ( i in 1..40 ) {` line.

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

> ### Note
>
> At the time of writing this book, only **Maven** 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 configurations

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 Black Duck code search engine (http://code.ohloh.net/).

Maven 2 and 3 pitfalls

Maven 3 is mostly backwards-compatible with Maven 2. However, it does have some differences that you can review at https://cwiki.apache.org/confluence/display/MAVEN/Maven+3.x+Compatibility+Notes. For the list of compatible plugins, visit https://cwiki.apache.org/confluence/display/MAVEN/Maven+3.x+Plugin+Compatibility+Matrix.

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 legacy configuration as that would cause unnecessary maintenance work. Maven 3 is fussier about syntax than Maven 2. It will complain if you forget to add a version number for any of your dependencies or plugins. For example, in the first edition of this book, the *Failing Jenkins jobs based on JSP syntax errors* recipe included a `pom.xml` file that had its `keytool-maven-plugin` float without a defined version:

```
<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>
```

When running with Maven 3, the recipe fails with the following output.

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 0.450s
[INFO] Finished at: Sat Nov 22 12:56:02 CET 2014
[INFO] Final Memory: 5M/109M
[INFO] ------------------------------------------------------------------------
[ERROR] Could not find goal 'genkey' in plugin org.codehaus.mojo:keytool-maven-plugin:1.5 among available
als clean, exportCertificate, changeKeyPassword, generateSecretKey, importKeystore, changeStorePassword,
ortCertificate, generateCertificateRequest, list, printCertificateRequest, printCertificate, deleteAlias,
lp, generateKeyPair, changeAlias, printCRLFile, generateCertificate -> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read the following articles:
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MojoNotFoundException
```

The `genkey` goal no longer existed because Maven 3 was scanning using the newest version of the plugin, Version 1.5. Reviewing the plugin's website at http://mojo.codehaus.org/keytool/keytool-maven-plugin/, it was clear that we needed to update the version number and goal:

Deprecated goals

In version 1.2, a lots of new goals were introduced (MKEYTOOL-19) to perform all possible task available in the keytool command.

Note also that new goals are not attached to any phase, so you will have to set explicitly the phase on which attach executions of goals.

As the new goals can set all parameters than older one, we prefer to deprecate the old one... Please use the new mojo (old one will be at next major release removed).

- **Deprecated since 1.2** keytool:genkey generates a keystore **replaced by generateKeyPair**.
- **Deprecated since 1.2** keytool:export reads a certificate from a keystore and stores it in a file **replaced by exportCertificate**
- **Deprecated since 1.2** keytool:import reads a certificate from a file and stores it in a keystore **replaced by importCertificate**.

The changes are reflected in the updated `pom.xml` file:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>keytool-maven-plugin</artifactId>
     <version>1.5</version>
  <executions>
    <execution>
      <phase>generate-resources</phase>
      <id>clean</id>
      <goals>
        <goal>clean</goal>
      </goals>
    </execution>
    <execution>
      <phase>generate-resources</phase>
        <id>generateKeyPair</id>
        <goals>
          <goal>generateKeyPair</goal>
        </goals>
    </execution>
  </executions>
```

```
</plugin>
```

Another pitfall is the use of the Maven site plugin in Maven 3 reflected in the way the `<reporting>` section is configured.

An efficient method of upgrading site generation from Maven 2 is to start from a working archetype generated in Maven 3 and incrementally transfer and test the features from the Maven 2 project. Once you have a fully featured Maven 3 project, you can turn it into its own archetype later to act as a template for further projects.

> **Note**
>
> You will find information on building your own archetypes at http://maven.apache.org/guides/mini/guide-creating-archetypes.html.

When upgrading from Maven 2 to 3, you will mostly find that JAR dependencies and versions are explicitly mentioned. Let's take a look at the following example:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
      <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

An upgrade is an ideal time to look to see if you can find newer versions with bugs and known security issues removed. The Maven repository search engine (http://search.maven.org/) is a logical place to look for new versions. You can also consider browsing the repository at http://search.maven.org/#browse for JUnit and click on the link:

../	
3.7/	07-Dec-2010
3.8.1/	07-Dec-2010
3.8.2/	07-Dec-2010
3.8/	07-Dec-2010
4.0/	07-Dec-2010
4.1/	07-Dec-2010
4.10/	29-Sep-2011
4.11-beta-1/	15-Oct-2012
4.11/	14-Nov-2012
4.12-beta-1/	27-Jul-2014
4.2/	07-Dec-2010

You can now see the different version numbers and dates uploaded. In the case of JUnit, I would upgrade to the latest version; if the build does not work because of API incompatibilities, then fall back to the last stable point version, Version 3.8.2.

See also

- The *Running Groovy scripts through Maven* recipe
- The *Plotting alternative code metrics in Jenkins* recipe
- The *Failing Jenkins jobs based on JSP syntax errors* recipe

Chapter 4. Communicating Through Jenkins

In this chapter, we will cover the following recipes:

- Skinning Jenkins with the simple themes 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 View plugin
- Making noise with HTML5 browsers
- An extreme view for reception areas
- Mobile presentation using Google Calendar
- Mobile apps for Android and iOS
- Knowing your audience with Google Analytics
- Simplifying powerful visualizations using the R plugin

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 jobs, allowing you to make quick decisions. You can easily set up multiple views, prioritizing information naturally. Jenkins, with its hordes of plugins, notifies you by e-mail, dashboards, 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.

Its primary audience is developers, but don't forget the wider audience that wants to use the software being developed. Seeing Jenkins regularly building with consistent views and a corporate look and feel builds confidence in the software's roadmap. 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.

- **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 style sheets and JavaScript.
- **Overlaying Jenkins**: Using the Maven WAR plugin, you can overlay your own content on top of 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 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, 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, then passers-by will absorb the ebb and 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 then 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.

Tip

Subversion repository

From this chapter onwards you will need either a Git or a subversion repository. This will allow you to use Jenkins in the most natural way possible. For the sake of brevity, we mention only subversion in the recipes, but it is easy to choose Git. If you do not already have a repository, there are a number of free or semi-free services you can sign up to on the Internet, for example, http://www.straw-dogs.co.uk/09/20/6-free-svn-project-hosting-services/ or an example of a Git repository https://bitbucket.org/.

Alternatively, you can consider setting up subversion or Git locally. For Ubuntu installation instructions, visit https://help.ubuntu.com/community/Subversion and https://help.ubuntu.com/lts/serverguide/git.html.

Skinning Jenkins with the simple themes plugin

This recipe modifies the Jenkins look and feel through the themes plugin.

The themes plugin is a page decorator: it decorates each page with extra HTML tags. The plugin allows you to upload a style sheet 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 (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 line of code:

```
document.write("<h1 id='test'>Example
Location</h1>")
```

2. Create a `mycss.css` file in the Jenkins `userContent` directory with the following lines of code:

```
@charset "utf-8";
#test {
  background-image:
url(/userContent/camera.png);
}
#main-table{
  background-image:
url(/userContent/camera.png) !important;
```

3. Download and unpack the icon archive http://sourceforge.net/projects/openiconlibrary/files/0.11/open_icon_library-standard-0.11.tar.bz2/download and review the available icons. Alternatively, you can use the icon included in the download from the book's website (www.packtpub.com/support). Add an icon to the `userContent` directory renaming to `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/myjavascript.css`

- **URL of theme JS**: `/userContent/mycss.js`

5. Click on **Save**.
6. Return to the Jenkins home page and review your work, as shown in the following screenshot:



How it works...

The simple themes plugin is a page decorator. It adds the following information to every page:

```
<script>
<link rel="stylesheet" type="text/css"
href="/userContent/mycss.css" /><script
src="/userContent/myjavascript.js"
type="text/javascript">
</script>
```

The JavaScript writes a heading near the top of the generated pages with `id="test"`. Triggering the cascading style sheet rule through the CSS locator `#test` adds the camera icon to the background.

The picture 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 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.

Tip

CSS 3 quirks

There are quirks in the support for the various CSS standards between browser types and versions. For an overview, please visit 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 features. To draw a button around the header generated by the JavaScript, change the `#test` section of the CSS file to the following code:

```
#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:



Note

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://yuilibrary.com/. Decorated in each HTML page, the core YUI library (`/scripts/yui/yahoo/yahoo-min.js`) is pulled in ready for reuse. However, many web developers are used to jQuery. You

can include this library as well 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 only a few administrators maintain your Jenkins deployment, then you can most likely trust everyone to add JavaScript with no harmful side effects. However, if you have a large set of administrators who use a wide range of Java libraries, then your maintenance and security risks increase rapidly. Please consider your security policy and at least add the audit trail plugin (https://wiki.jenkins-ci.org/display/JENKINS/Audit+Trail+Plugin) to keep track of actions.

See also

- The *Skinning and provisioning Jenkins using a WAR overlay* recipe
- The *Generating a home page* recipe

Skinning and provisioning Jenkins using a WAR overlay

This recipe describes how to overlay 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. The basic example of adding your own custom `favicon.ico` (the icon in your web browser's address bar) is used. It requires little 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 `fav` icon, and `robots.txt` that affects how search engines look through your content.

Be careful: using a WAR overlay will be cheap if the structure and the graphical content of Jenkins do not radically change over time. However, if the overlay does break the structure, then 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`, then using the simple theme plugin (see the preceding recipe) to do the styling.

Getting ready

Create the directory named `ch4.communicating/war_overlay` for the files in this recipe.

How to do it...

1. Browse to the Maven repository http://repo.jenkins-ci.org/releases/org/jenkins-ci/main/jenkins-war/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>Jenkins</id>

 <url>http://repo.jenkins-ci.org/releases</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 `src/main/webapp` location.
5. Create the directory `src/main/webapp/META-INF` and add a file named `context.xml` with the following line of code:

```
<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 WAR file `overlay-1.437.war`. Review the content, verifying that you have modified `favicon.ico`.
8. [Optional] Deploy the WAR file to a local Tomcat server, 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 `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 as Tomcat picks up the settings here without the need for 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 http://repo.jenkins-ci.org/releases 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, review http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#Dependency_Scope.

For further details about WAR overlays, review http://maven.apache.org/plugins/maven-war-plugin/index.html.

Tip

Avoiding work

To limit maintenance effort, it is better to install extra content rather than replacing 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 to two main locations. The first location is the core application and the second 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, which 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** (directory to store checksums to uniquely identify files)

├── **jobs** (stores job configuration and build results)

├── **plugins** (where plugins are deployed and usually configured)

├── **tools** (where tools such as Maven and Ant are deployed)

├── **updates** (updates)

├── **userContent** (content made available under the URL `/userContent`)

└── **users** (user information displayed under the `/me` URL)

The default Ubuntu location of the web app is `/var/run/jenkins/war`. If you are running Jenkins from the command line, then the option for placing the web app is:

```
-webroot
```

├── **css** (location of Jenkins style sheets)

├── **executable** (used for running Jenkins from the command line)

├── **favicon.ico** (the icon we replaced in this recipe)

├── **help** (directory with help content)

├── **images** (graphics in different sizes)

├── **META-INF** (location for manifest file and `pom.xml` file that generated the WAR)

├── **robots.txt** (used to tell search engines where they are allowed to crawl)

├── **scripts** (JavaScript library location)

├── **WEB-INF** (main location for the servlet part of the web application)

└── **winstone.jar** (The servlet container: 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, then consider modifying the top-level `robots.txt` file. 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` 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; see https://developers.google.com/webmasters/control-crawl-index/docs/robots_txt?csw=1

The following `robots.txt` allows access by the Google crawler to the directory `/userContent/corporate/`. It is an open question whether all web crawlers will honor the intent.

```
User-agent: *
Disallow: /
User-agent: Googlebot
Allow: /userContent/corporate/
```

> **Note**
>
> To help secure your Jenkins infrastructure, refer to recipes in Chapter 2, *Enhancing Security*.

See also

- The *Skinning Jenkins with the simple themes plugin* recipe
- The *Generating a home page* recipe

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 express 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` URL.

> **Note**
>
> A similar plugin worth reviewing is the Gravatar plugin. You can find the plugins home page at https://wiki.jenkins-ci.org/display/JENKINS/Gravatar+plugin

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 detailed in the *Reviewing project-based matrix tactics via a custom group script* recipe in Chapter 2, *Enhancing Security*.

> **Note**
>
> You will not get this recipe to work unless as a Jenkins administrator you configure the **Markup Formatter** under the **Configure Global Security** page as **Raw**. By doing so you allow anyone who can edit descriptions to inject their own script code. If you are a very trusting small team of developers, this may be an OK practice. However, in general consider this a security issue.

How to do it...

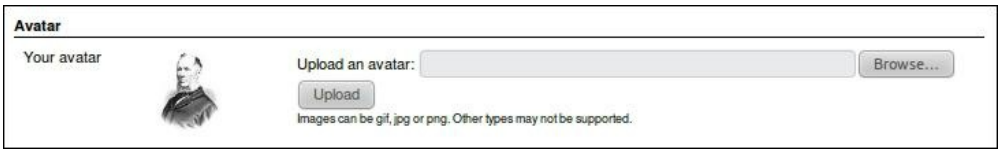
1. Browse to the location http://en.wikipedia.org/wiki/Wikipedia:Public_domain_image_resources for a list of public domain sources of images.

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 on the **Download Image File: 75 px** link, as shown in the following screenshot:



> **Note**
>
> If the image is no longer available then choose another.

4. Log in to your sacrificial Jenkins server as `fakeuser` and visit their 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 user profile description:

```
<script type="text/JavaScript">
functionchangedivview()
{
```

```
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>
```

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 the `fakeuser` to a directory `fakeuser2`, for example, `/var/lib/jenkins/user/fakeuser`. 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`.
10. Log in as `fakeuser2` with the same password as `fakeuser`.
11. Visit the home page `/user/fakeuser2`. Notice 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 it with the simple themes 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 archive.org.

Unless you filter the description (refer to the *Exposing information through build descriptions* recipe 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 it 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 (refer to the *Using Script Realm authentication for provisioning* recipe 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 empty. 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 bad practice, a problem waiting to happen.

The description is stored under the description tag. The content is stored as 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

- The *Skinning Jenkins with the simple themes plugin* recipe
- The *Skinning and provisioning Jenkins using a WAR overlay* recipe
- The *Reviewing project-based matrix tactics via a custom group script* recipe in Chapter 2, *Enhancing Security*

Creating HTML reports

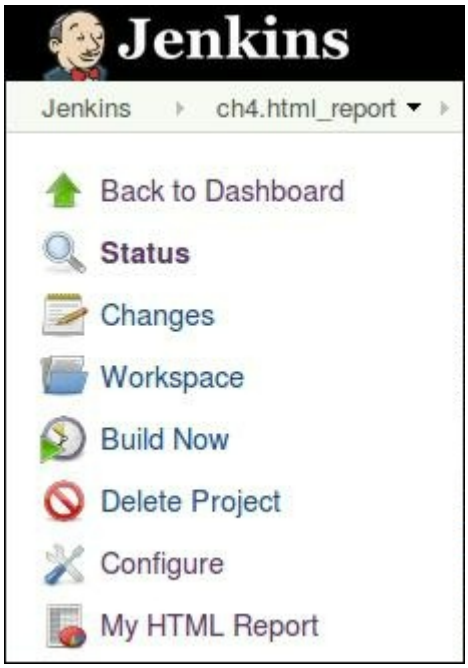
The left-hand side menu of a jobs dashboard is valuable real estate. The developer's eyes naturally scan this area. This recipe describes how you can add a link from a custom HTML report to the menu, getting the report noticed more quickly.

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 and name 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**. Add the following details:
    - **HTML directory to archive**: `target/custom_report`
    - **Index pages[s]**: `index.html`
    - **Report title**: `My HTML Report`
    - Tick the **Keep past HTML reports** checkbox
5. Click on **Save**.
6. Run the job and review the left-hand side menu. You will now see a link to your report, as shown in the following screenshot:



How it works...

Your subversion repo contains an `index.html` file that 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:

```
<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 style sheet `/css/style.css`.

It is possible that when you update a style sheet 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 later form. Consider doing so for your style sheets.

> **Note**
>
> The unique number changes per version, so you will need to update the URL for 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 documentation such as test results within easy reach.

See also

- The *Efficient use of views* recipe
- The *Saving screen space with the Dashboard View plugin* recipe

Efficient use of views

Jenkins' addictive ease of configuration lends itself to creating a large number of jobs. This increases the volume of information exposed to developers. Jenkins needs to avoid chaos by utilizing browser space efficiently. One approach is to define minimal views. In this recipe, you will use the DropDown ViewsTabBar plugin. It removes views as tabs and replaces the tabs with one select box. This aids in quicker navigation. You will also be shown how to provision lots of jobs quickly using a simple HTML form generated by a script.

> **Tip**
>
> In this recipe, you will be creating a large number of views that you may want to delete later. If you are using a virtual box image, consider cloning the image and deleting it after you have finished.

Getting ready

Install the DropDown ViewsTabBar plugin (https://wiki.jenkins-ci.org/display/JENKINS/DropDown+ViewsTabBar+Plugin).

How to do it...

1. Copy 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
```

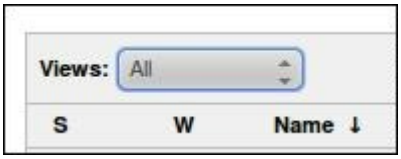
3. In a web browser, as an administrator, log in to Jenkins.
4. Create the job `Template1`, adding any details you wish. This is your template job that will be copied into many other jobs.
5. Load `form.html` into the same browser.
6. Click on one of the **Create CH4.fake** buttons. Jenkins returns an error message:

```
HTTP ERROR 403 Problem accessing /createItem.
Reason:
No valid crumb was included in the request
```

7. Visit **Configure Global Security** on `http://localhost:8080/configureSecurity` and uncheck the **Prevent Cross Site Request Forgery exploits** box.
8. Click on **Save**.
9. Click on all of the **Create CH4.fake** buttons.
10. Visit the front page of Jenkins and verify that the jobs have been created and are based on the `Template1` job.
11. Create a large number of views with a random selection of jobs. Review the front page, noting the chaos.
12. Visit the configuration screen `/configure`, selecting **DropDownViewsTabBar provides a drop down menu for selecting views** in the **View Tab Bar** select box**.** In the subsection **DropDownViewsTabBar**, check the **Show Job Counts** box, as shown in the following screenshot:



13. Click on the **Save** button:



14. In Jenkins, visit **Configure Global Security** `http://localhost:8080/configureSecurity` and check the **Prevent Cross Site Request Forgery exploits** box.
15. Click on **Save**.

How it works...

The form works as long as the bread crumbing security feature in Jenkins is turned off. The feature, when turned on, generates a random number that the form has to return when you submit. This allows Jenkins to know that the form is part of a valid conversation with the server. The HTTP status error generated is in the 4xx range which implies that the client input is invalid. If Jenkins returned a 5xx error then that would imply a server error. We therefore had to turn the feature off when submitting our own data. We do not recommend this in a production environment.

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`, as shown in the following code:

```
<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` for 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 variable `$end` from `20`.

There's more...

Jenkins uses a standard library, Stapler (http://stapler.kohsuke.org/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 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

- The *Saving screen space with the Dashboard View plugin* recipe

Saving screen space with the Dashboard View plugin

In the *Efficient use of views* recipe, you discovered that you can save horizontal tab space using the Views 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 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

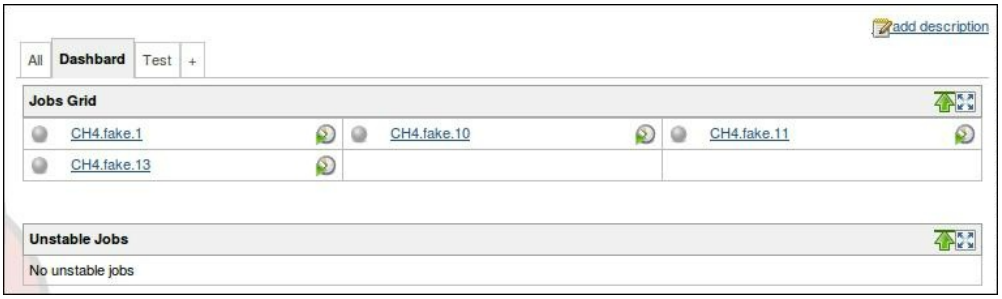
Install the Dashboard View plugin (https://wiki.jenkins-ci.org/display/JENKINS/Dashboard+View). Either create a few jobs by hand or use the HTML form that provisioned jobs in the last recipe.

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 the default.
6. Click on **OK**. You will now see a blank view screen.
7. In the left-hand side menu, click on the **Edit View** link.
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, click on the **OK** button. You will now see the **Dashboard** view:



You can expand or contract the areas of functionality with the arrow icon:



How it works...

The Dashboard plugin divides up the screen into areas. During 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, and the test trend chart. There will be more choices as the plugin matures.

The **Job grid** portlet saves space compared to other views, as the density of jobs displayed is high.

> ## Tip
>
> If you are also using the Many **Views** tab (see the preceding 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

- The *Creating HTML reports* recipe
- The *Efficient use of views* recipe

Making noise with HTML5 browsers

This recipe describes how to send a custom sound to a Jenkins user's browser when an event occurs, such as a successful build. You can also send sound messages at arbitrary times. Not only is this good for developers who enjoy being shouted at, sang at by famous actors, and so on, but also for 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, such as a current version of Firefox or Chrome.

> **Note**
>
> For more details of HTML5 compliancy in browsers, consider reviewing: http://en.wikipedia.org/wiki/Comparison_of_layout_engines_%28HTML5%29.

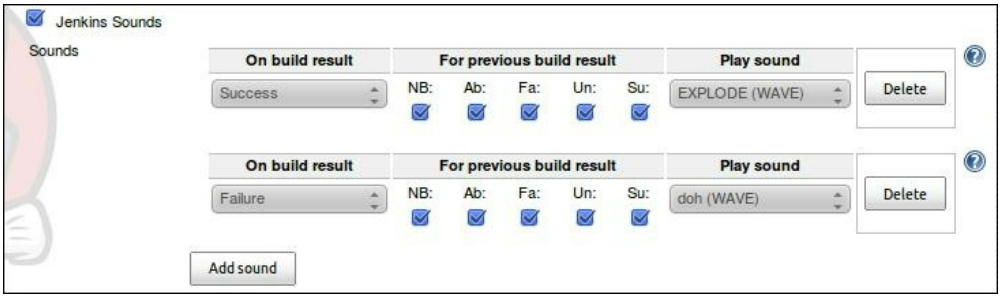
How to do it...

1. Log in as a Jenkins administrator and visit the **Configure System** screen `/configure`.
2. Under the **Jenkins Sound** section, check **Play through HTML5 Audio enabled Browser**.

> **Note**
>
> If Jenkins has problems finding the sound archive with an error message such as `File not found 'file:/C:/Users/Alan/.jenkins/jar:file:/C:/Users/Alan/.jenkins/plugins/sou INF/lib/classes.jar/sound-archive.zip`', then unzip the `classes.jar` file and move the `sounds-archive.zip` file to the same directory mentioned in the error message. Finally, point the configuration to the archive, for example,

```
file:/C:/Users/Alan/.jenkins/plugins/sounds/WEB-
INF/lib/sound-archive.zip.
```

3. Click on 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. Click on **OK**.
8. In the **Post-build Actions** section, check the **Jenkins Sounds** option.
9. Add two sounds: **EXPLODE** and **doh**:



10. Click on **Save**.
11. Click on the **Build Now** link.
12. On success, your browser will play 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 the 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() {
      newAjax.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 10-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 trigger address (replacing `localhost:8080` with your own address):

```
http://localhost:8080/sounds/playSound?
src=http://localhost:8080/userContent/shenai.wav
```

See also

- The *Keeping in contact with Jenkins through Firefox* recipe in 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 Panel 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. You can display behind your reception desk or in a well-visited social area such as near the coffee machine or project room.

In this recipe, you will add the eXtreme Feedback Panel plugin and modify its appearance through HTML tags in the description.

Getting ready

Install the eXtreme Feedback Panel plugin (https://wiki.jenkins-ci.org/display/JENKINS/eXtreme+Feedback+Panel+Plugin).

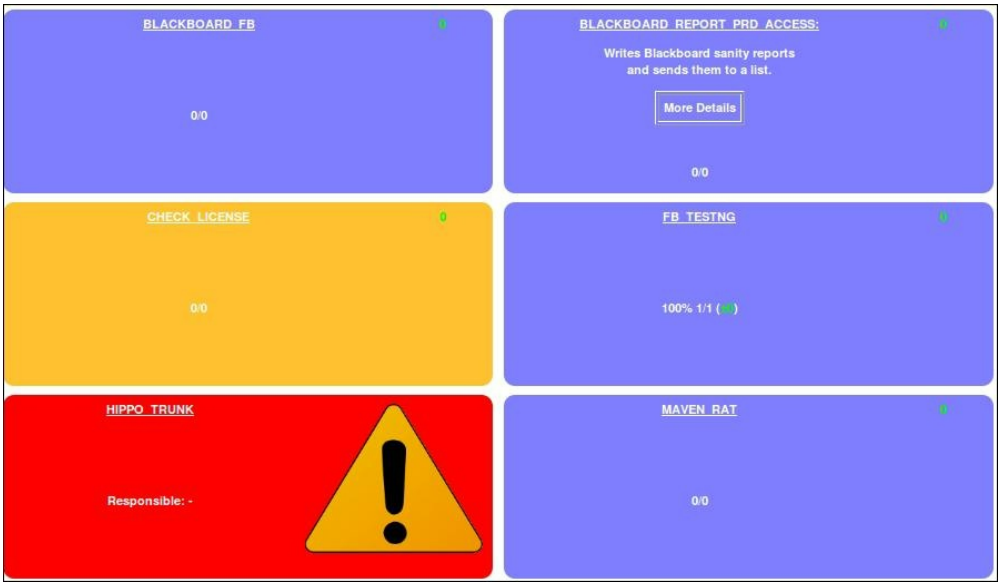
How to do it...

1. Create a job with a descriptive name such as `Blackboard Report Pro 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 the **eXtremeFeedBack Panel** and then click on **OK**.
3. Select 6-24 already created jobs including the one previously created in this recipe.
4. Set the number of columns as **2**.
5. Select the refresh time in seconds as **20**.
6. Click on **Show Job descriptions**.
7. Click on **OK**.
8. Experiment with the settings (especially the pixel size of the fonts). Optimizing the view depends on the monitors used and the distance

from the monitor that the audience view at, as shown in the following screenshot:



How it works...

Setting up and running this information radiator was easy. 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 in the extreme view, but HTML escaped in the jobs configuration page and elsewhere in Jenkins. 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, below a certain length to fit naturally on the screen. A longer, more descriptive name for a job helps the viewer understand the job's context better.

> **Note**
>
> A short cut to configuring views is through the URL `http://localhost:8080/view/Jobname/configure`, replacing any spaces in `Jobname` with `%20`.

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 the voices of famous actors (see the *Making noise with HTML5 browsers* recipe).

A number of example electronic projects worth exploring in Jenkins 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

- The *Saving screen space with the Dashboard View plugin* recipe
- The *Making noise with HTML5 browsers* recipe

Mobile presentation using Google Calendar

Jenkins plugins can push build history to different well-known social media services. Modern Android or iOS telephones 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...

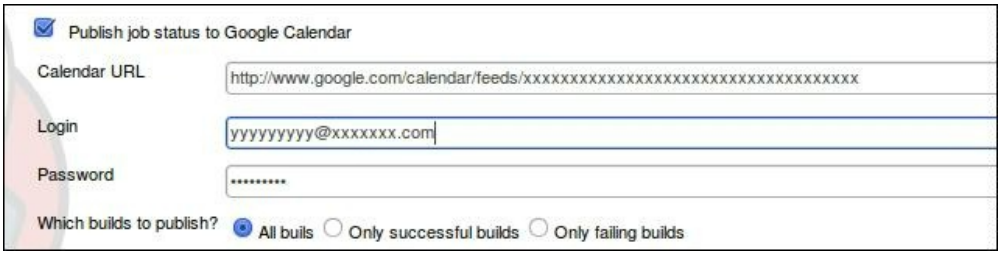
1. Log in to Gmail and visit the **Calendar** page.
2. Create a new calendar under the **My Calendars** section by clicking on the **Add** link.
3. Add the calendar name `Test for Jenkins`.
4. Click on **Create Calendar**. By default, the new calendar is private. Keep it private for the time being.
5. Under the **My Calendars** section, click on the down icon next to **Test for Jenkins**. Select the option **Calendar settings**.
6. Right-click on the XML button **Copy the link location**:



7. Review the **Embed this calendar** section. 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 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** text box.
12. Add your Gmail login name and password.

> **Tip**

Your Gmail credentials will be stored in the `server.xml` file in plain text. Unless your server is properly secured, this is not recommended.



13. Click on **Save**.
14. Build your job, making sure it succeeds.
15. Log in to Gmail. Visit the **Calendar** page. You will now see the build's success has been published, as shown in the following screenshot:



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.

Note

Jenkins has a credentials manager that you can find at `http://localhost:8080/credential-store/`. The credential manager works with a number of plugins; however, at the time of writing not with the Google Calendar plugin. For the most up-to-date compatibility

information visit: https://wiki.jenkins-ci.org/display/JENKINS/Credentials+Plugin

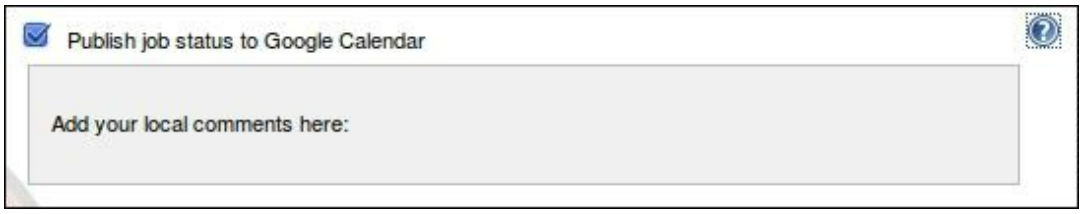
There's more...

Under the Jenkins workspace in the plugins directory, you will find an HTML file for the Google plugins configuration help `/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, as shown in the following screenshot:



This example is an anti-pattern. If you need to change content for local needs, then it is much better to work with the community, adding to the Jenkins SCM that 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 requires minimal startup effort and is an easy way to start with an open source project.

Note

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

- The *Mobile apps for Android and iOS* recipe

Mobile apps for Android and iOS

There are a number of rich mobile apps for 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 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:

    - **JenkinsMobi** (http://www.jenkins-ci.mobi)
    - **Blamer** (http://www.androidzoom.com/android_applications/tools/blamer_bavqz.html and 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)
    - **Hudson2Go Lite** (http://www.androidzoom.com/android_applications/tools/hudson2go-lite_nane.html)

4. On your mobile device, search Google 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 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 off from the source code of Hudson 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 the Google 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 through the Google Marketplace onto a mobile device. However, if you want to play with Android apps in a sand box 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, plus the ability to save the virtual machine in a specific state. However, you will not always get Google Marketplace preinstalled. You will either have to find particular apps' `.apk` file yourself or add other marketplaces such as **Slide**

me (http://m.slideme.org). Unfortunately, secondary marketplaces give you much less choice:



The Windows Android emulator http://bluestacks.com/home.html shows great promise. Not only is it an emulator, 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 on installation. BlueStacks wishes to obtain detailed information about your system to help improve their product.

See also

- The *Mobile presentation using Google Calendar* recipe

Knowing 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, 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, (for example, if mobile apps are hitting your site), entry points, and country origins. This is particularly useful as your product reaches key points in its roadmap and you want to gain insight into 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).

> **Tip**
>
> 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 details of the **Create New Account** page:
    - **Account Name**: My Jenkins Server
    - **Website Name**: Jenkins Server X
    - **Website URL**: This is the same as the Jenkins URL in the `/configure` screen of Jenkins.
    - **Reporting Time zone**: Enter the correct value
    - Select **Data Sharing Settings** | **Sharing Setting** | **Do not share my Google Analytics data**
    - Click on **Get Tracking ID**
    - Click on **I Accept** for the **Google Analytics Terms of Service Agreement**
3. Click on **Create Account**.

4. You are now in the **Accounts** page for your newly created profile. Copy the **TrackingID,** which will look something like `UA-121212121212121-1`.
5. Open a second browser and log in as an administrator to Jenkins.
6. In 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. Click on the **Save** button.
8. Visit the home page of Jenkins so that tracking is triggered.
9. Return to Google Analytics, and you should still be on the **Tracking code** tab. Click on **Save** at the bottom of the page. You will now see that the warning **Tracking not installed** has disappeared.

How it works...

The plugin decorates each Jenkins page with a JavaScript page tracker that includes the domain and profile ID. The JavaScript is kept fresh by being pulled in from the Google Analytics hosts, as shown in the following code:

```
<script type="text/javascript">
var _gaq = _gaq || [];
_gaq.push(['_setAccount', '
UA-121212121212121-1']);
_gaq.push(['_setDomainName', 'Domain Name']);
_gaq.push(['_trackPageview']);

(function() {
  varga = document.createElement('script');
  ga.type = 'text/javascript'; ga.async = true;
  ga.src = ('https:' ==
document.location.protocol ? 'https://ssl' :
'http://www') + '.google-analytics.com/ga.js';
  var s =
document.getElementsByTagName('script')[0];
  s.parentNode.insertBefore(ga, s);
})();
</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.

> **Note**
>
> Google regularly updates its analytics services. If you notice any changes then the help page for analytics will document them (https://support.google.com/analytics).

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 local data usage under your control.

As you would expect, the Piwik plugin is a page decorator injecting in a similar way to JavaScript as the Google Analytics plugin.

See also

- The *Generating a home page* recipe

Simplifying powerful visualizations using the R plugin

R is a popular programming language for statistics http://en.wikipedia.org/wiki/R_(programming_language). It has many hundreds of extensions and has a powerful set of graphical capabilities. In this recipe, we will show you how to use the graphical capabilities of R within your Jenkins jobs and then point you to some excellent starter resources.

> **Note**
>
> For a full list of plugins that improve the UI of Jenkins, including Jenkins graphical capabilities, visit https://wiki.jenkins-ci.org/display/JENKINS/Plugins#Plugins-UIplugins.

Getting ready

Install the R plugin (https://wiki.jenkins-ci.org/display/JENKINS/R+Plugin). Review the R installation documentation (http://cran.r-project.org/doc/manuals/r-release/R-admin.html).

How to do it...

1. From the command line install the R language:

```
sudo apt-get install r-base
```

2. Review the available R packages:

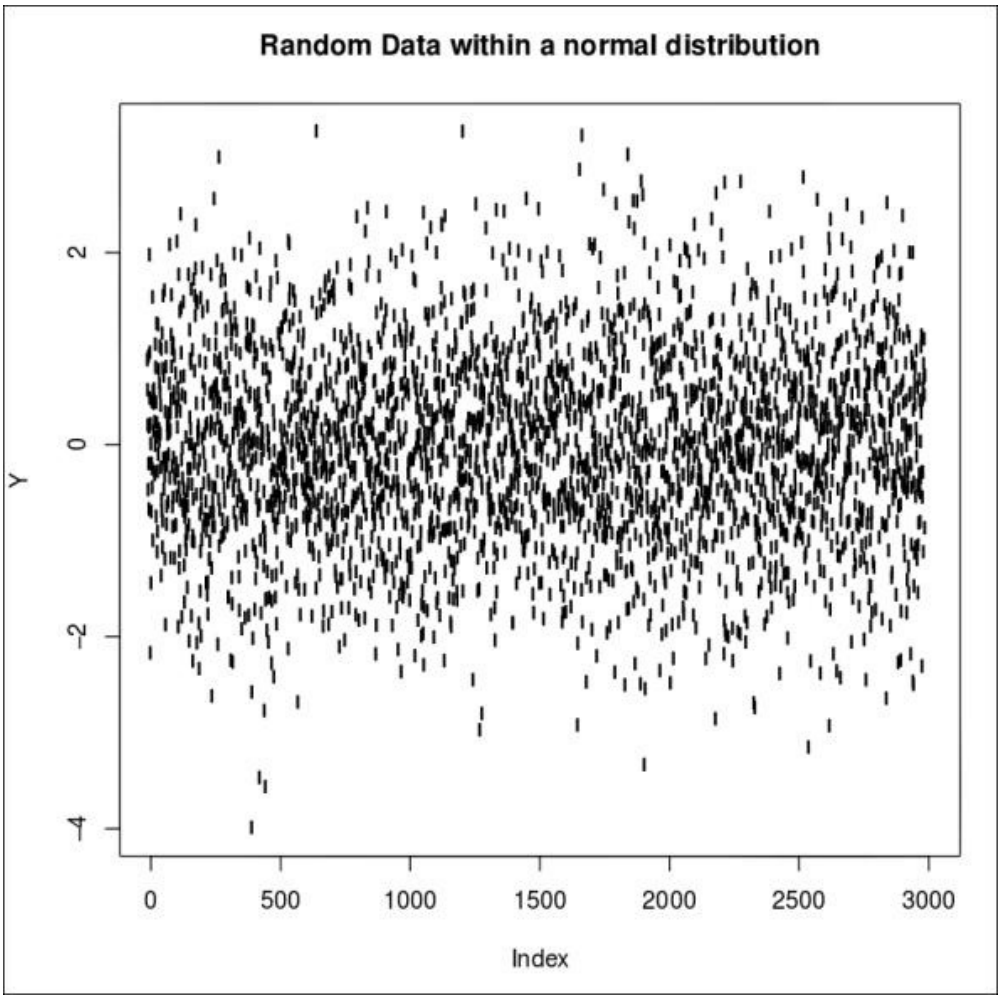
```
apt-cache search r-cran | less
```

3. Create a free-style job with the name `ch4.powerfull.visualizations`.
4. In the **Build** section, under **Add build step**, select **Execute R script**.
5. In the **Script** text area add the following code:

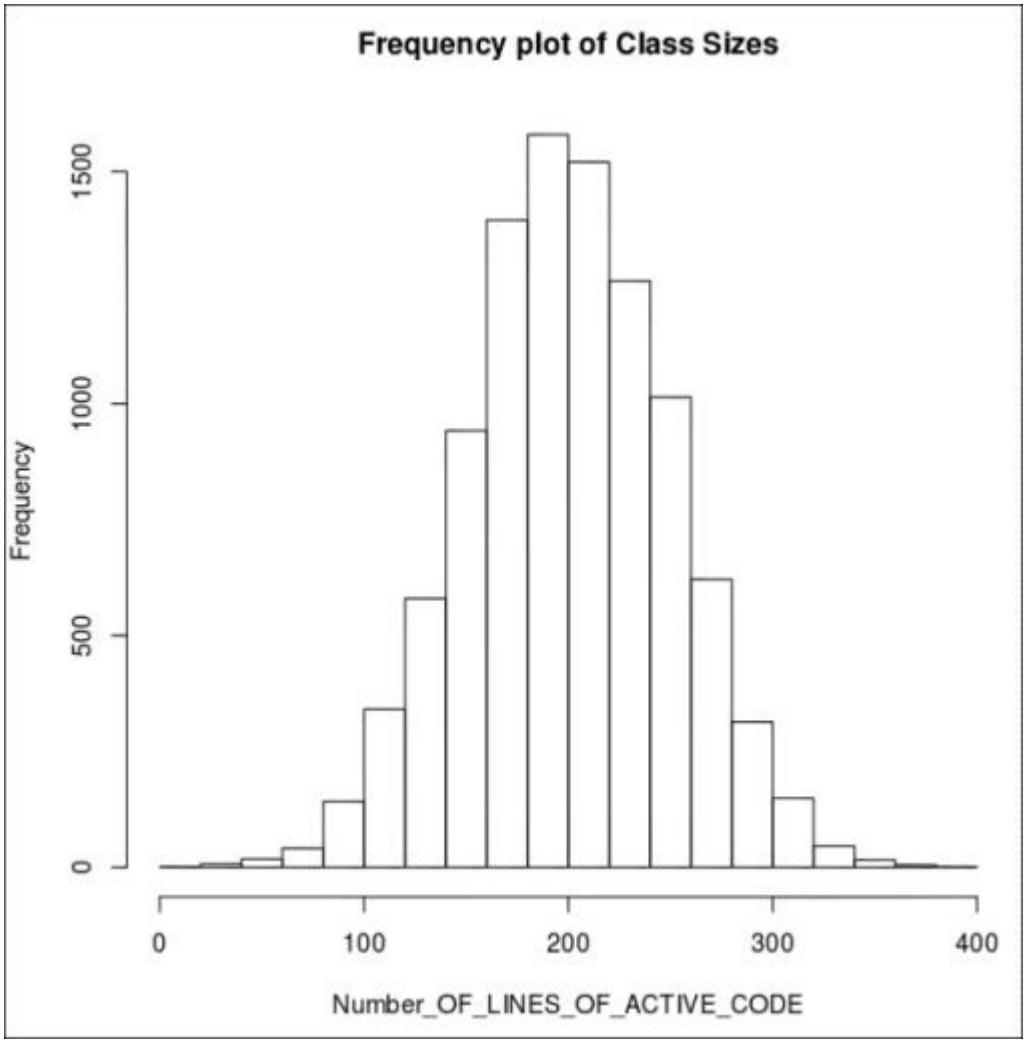
```
paste('=====================================');
paste('WORKSPACE: ', Sys.getenv('WORKSPACE'))
paste('BUILD_URL: ', Sys.getenv('BUILD_URL'))
print('ls /var/lib/jenkins/jobs/R-ME/builds/')
paste('BUILD_NUMBER: ',
Sys.getenv('BUILD_NUMBER'))
paste('JOB_NAME: ', Sys.getenv('JOB_NAME'))
```

```
paste('JENKINS_HOME: ',
Sys.getenv('JENKINS_HOME'))
paste( 'JOB LOCATION: ',
Sys.getenv('JENKINS_HOME'),'/jobs/',Sys.getenv('JOB_NAME'),'/builds/
Sys.getenv('BUILD_NUMBER'),"/test.pdf",sep="")
paste('======================================');

filename<-
paste('pie_',Sys.getenv('BUILD_NUMBER'),'.pdf',sep="")
pdf(file=filename)
slices<- c(1,2,3,3,6,2,2)
labels <- c("Monday", "Tuesday", "Wednesday",
"Thursday", "Friday","Saturday","Sunday")
pie(slices, labels = labels, main="Number of
failed jobs for each day of the week")

filename<-
paste('freq_',Sys.getenv('BUILD_NUMBER'),'.pdf',sep="")
pdf(file=filename)
Number_OF_LINES_OF_ACTIVE_CODE=rnorm(10000,
mean=200, sd=50)
hist(Number_OF_LINES_OF_ACTIVE_CODE,main="Frequency
plot of Class Sizes")

filename<-
paste('scatter_',Sys.getenv('BUILD_NUMBER'),'.pdf',sep="")
pdf(file=filename)
Y <- rnorm(3000)
plot(Y,main='Random Data within a normal
distribution')
```

6. Click on the **Save** button.
7. Click on the **Build Now** icon.
8. Beneath **Build History**, click on the **Workspace** button.
9. Review the generated graphics by clicking on the links **freq_1.pdf**, **pie_1.pdf**, and **scatter_1.pdf**, as shown in the following screenshot:



The following screenshot is a histogram of the values from the random data generated by the R script during the build process. The data simulates class sizes within a large project.



Another view is a pie chart. The fake data represents the number of failed jobs for each day of the week. If you plot this against your own values, you might see particularly bad days, such as the day before or after the weekend. This might have implications for how developers work, or how motivation is distributed through the week.



Perform the following steps:

1. Run the job and review the **Workspace**.
2. Click on **Console Output**. You will see output similar to:

```
Started by user anonymous
Building in
workspace /var/lib/jenkins/workspace/ch4.Powerfull.Visualizations
[ch4.Powerfull.Visualizations] $
Rscript /tmp/hudson6203634518082768146.R
[1] "========================================"
[1] "WORKSPACE:
 /var/lib/jenkins/workspace/ch4.Powerfull.Visualizations"
[1] "BUILD_URL:  "
[1] "ls /var/lib/jenkins/jobs/R-ME/builds/"
[1] "BUILD_NUMBER:  9"
[1] "JOB_NAME:  ch4.Powerfull.Visualizations"
[1] "JENKINS_HOME:  /var/lib/jenkins"
[1] "JOB
LOCATION: /var/lib/jenkins/jobs/ch4.Powerfull.Visualizations/builds/
[1] "========================================"
Finished: SUCCESS
```

3. Click on **Back to Proj**.
4. Click on **Workspace**.

How it works...

With a few lines of R code, you have generated three different well-presented PDF graphs.

The R plugin ran a script as part of the build. The script printed out the `WORKSPACE` and other Jenkins environment variables to the console:

```
paste ('WORKSPACE: ', Sys.getenv('WORKSPACE'))
```

Next, a filename is set with the build number appended to the string `pie_`. This allows the script to generate a different filename each time it is run, as shown:

```
filename
<-
paste('pie_',Sys.getenv('BUILD_NUMBER'),'.pdf',sep="")
```

The script now opens output to the location defined in the `filename` variable, through the command `pdf(file=filename)`. By default, the output directory is the job's workspace.

Next, we define fake data for the graph, representing the number of failed jobs on any given day of the week. Notice that in the simulated world Friday is a bad day:

```
slices <- c(1,2,3,3,6,2,2)
labels <- c("Monday", "Tuesday", "Wednesday",
"Thursday", "Friday","Saturday","Sunday")
```

Plot a pie graph:

```
pie(slices, labels = labels, main="Number of
failed jobs for each day of the week")
```

For the second graph, we generated 10,000 pieces of random data within a normal distribution. The fake data represents the number of lines of active code that ran for a given job, as shown:

```
Number_OF_LINES_OF_ACTIVE_CODE=rnorm(10000,
mean=200, sd=50)
```

The `hist` command generates a frequency plot:

```
hist(Number_OF_LINES_OF_ACTIVE_CODE,main="Frequency
plot of Class Sizes")
```

The third graph is a scatter plot with 3,000 data points generated at random within a normal distribution. This represents a typical sampling process, such as the number of potential defects found using Sonar or FindBugs, as shown:

```
Y <- rnorm(3000)
plot(Y,main='Random Data within a normal
distribution')
```

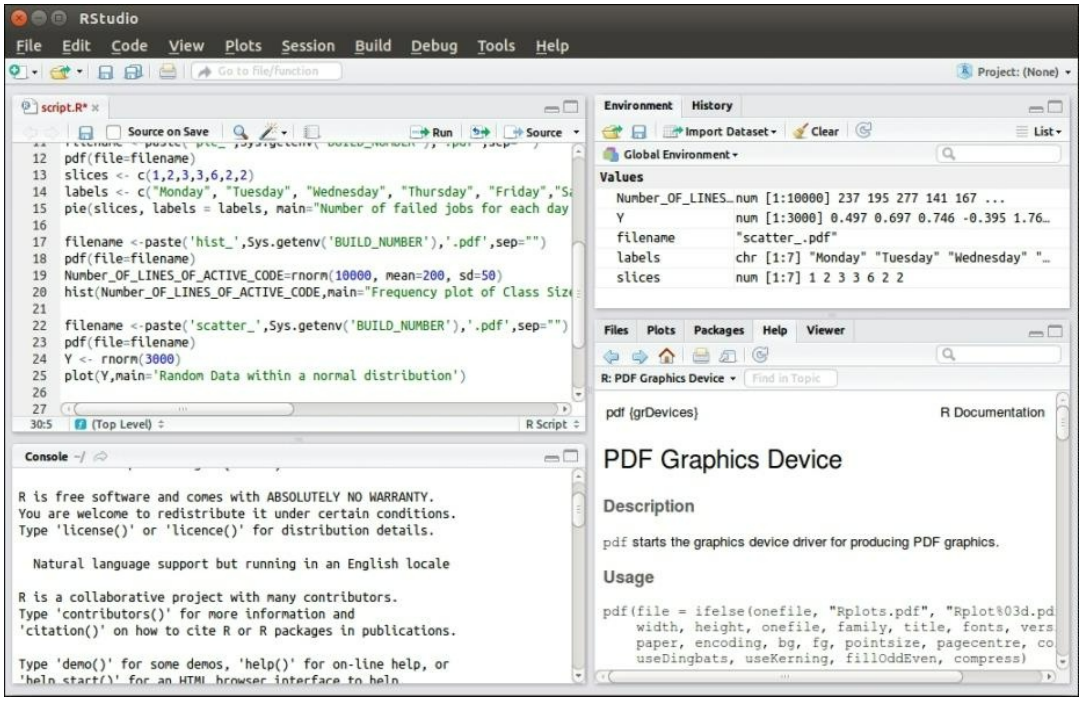
We will leave it as an exercise for the reader to link real data to the graphing capabilities of R.

There's more...

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

RStudio or StatET

A popular IDE for R is **RStudio** (http://www.rstudio.com/). The open source edition is free. The feature set includes a source code editor with code completion and syntax highlighting, integrated help, solid debugging features, and a slew of other features, as shown in the following screenshot:



An alternative for the Eclipse environment is the StatET plugin (http://www.walware.de/goto/statet).

Quickly getting help

The first place to start to learn R is by typing `help.start()` from the R console. The command launches a browser with an overview of the main documentation.

If you want descriptions of R commands then typing `?` before a command generates detailed help documentation. For example, in the recipe we looked at the use of the `rnorm` command. Typing `?rnorm` produces documentation similar to:

The Normal Distribution

Description

Density, distribution function, quantile function and random generation for the normal distribution with mean equal to mean and standard deviation equal to sd.

Usage

dnorm(x, mean = 0, sd = 1, log = FALSE)

pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)

qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)

rnorm(n, mean = 0, sd = 1)

For more information

The R language is well documented. Here are a number of useful resources:

- **Data Camp** (https://www.datacamp.com/courses): This is a great collection of free online courses including a basic introduction and a more detailed course on statistics.
- **R programming for those coming from other languages** (http://www.johndcook.com/blog/r_language_for_programmers/): This is a quick introduction to potential gotchas for new programmers with experience in other languages.
- **R style guide by Google** (https://google-styleguide.googlecode.com/svn/trunk/Rguide.xml): If you follow these guidelines, your code will be consistent and readable.

- **MOOCs**: There are a number of online courses available from Edx, Coursera, and so on. It is well worth reviewing their listings for relevance. For the most up-to-date listings of Coursera courses, visit https://www.coursera.org/courses.
- **Two-minute tutorials** (http://www.twotorials.com/): This contains many two-minute YouTube examples of what you can do in R.
- **Wiki books on R** (http://en.wikibooks.org/wiki/Category:R_Programming): This has great articles and examples assembled together in one place.

See also

- The *Analyzing project data with the R plugin* recipe in Chapter 5, *Using Metrics to Improve Quality*

Chapter 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 file
- Faking Checkstyle results
- Integrating Jenkins with SonarQube
- Analyzing project data with the R plugin

> **Note**
>
> 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 improves 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). Expect the cost to have increased considerably since.

To save money and improve quality, you need to remove defects as early in the software lifecycle as possible. The Jenkins test automation creates a

safety net of measurements. Another key benefit is that, once you have added tests, it is easy 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 over the details of the implementation.

Regression tests increase confidence that you have not broken code while refactoring 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 with Cobertura (https://cobertura.github.io/cobertura/). A similar framework is Emma (http://emma.sourceforge.net/). 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 collect real defects found in your projects, and 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 aid 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/codeconventions-150003.pdf).

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 for 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 View plugin* recipe in Chapter 4, *Communicating Through Jenkins*.

Jenkins is not limited to testing Java; a number of plugins such as SLOCCount or the DRY plugin (it spots duplication of code) are language-agnostic. There is even specific support for NUnit testing in .NET or compilation to other languages.

> ## Note
>
> NUnit, JUnit, RUnit, and several other unit testing frameworks follow the xUnit standard. For a detailed overview review the Wikipedia entry: http://en.wikipedia.org/wiki/XUnit

If you are missing specific functionality, 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) and a well written book by Diomidis Spinellis, *Code Quality: The Open Source Perspective*.

In the *Integrating Jenkins with SonarQube* 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 early warnings and to spot obvious defects such as a bad commit. You can then use Sonar for a deeper review.

Finally, you will run R code that parses all the files in a project and reports simple metrics. This custom process is easily adapted for complex analyses based on the wealth of statistical packages included in the R language.

> ## Note
>
> At the time of writing, both the FindBugs and PMD Jenkins plugins require a specific version of Maven. As an administrator, you can install the Maven version automatically through the main configuration screen (`http://hostname/configure`) under the **Maven** section by pressing the **Add Maven** button. Later when you create a job, Jenkins will give you a choice of Maven versions.

When dealing with multimodule Maven projects, the Maven plugins generate a series of results. The Maven project type rigidly assumes the results are stored in conventional locations, but this does not always happen consistently. With free-style projects, you can explicitly tell 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. SLOCCount pronounced "sloc-count" written by Dr. David Wheeler (http://www.dwheeler.com/sloccount/) is a command-line tool suite for counting physical source lines of code (SLOC) in potentially large software systems. From these metrics, you can estimate how many hours it would take to write the code and the estimated development costs.

Getting ready

Install the SLOCCount plugin (https://wiki.jenkins-ci.org/display/JENKINS/SLOCCount+Plugin). Create a new directory for this recipe's code. Install SLOCCount on the Jenkins instance as mentioned at http://www.dwheeler.com/sloccount. If you are running 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...

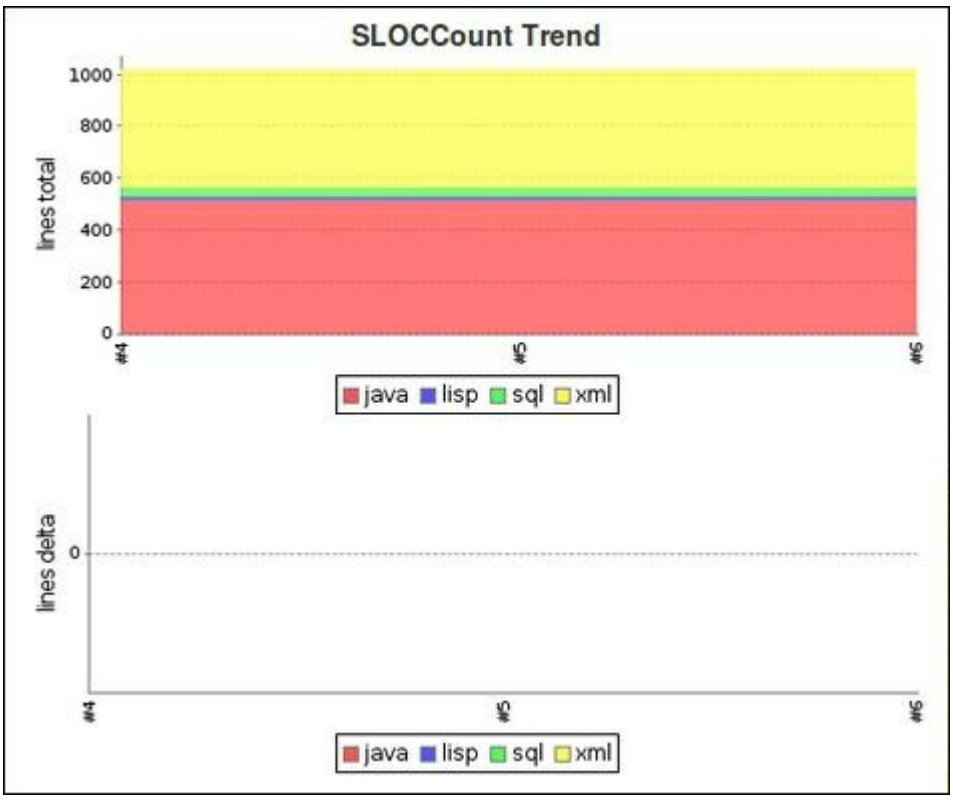
1. Create a free-style project and name it `ch5.quality.sloccount`. Add `SLOCCOUNT REPORT Project` as the description.
2. Under the **Source Code Management** section, check **Subversion**, adding for the **Repository URL**: https://source.sakaiproject.org/svn/shortenedurl/trunk.
3. Within the **Build** section, select **Execute shell** from the **Add build** step. Add the `/usr/bin/sloccount --duplicates --wide --details . >./sloccount.sc command`.
4. In the **Post-build Actions** section, check the **Publish SLOCCount analysis results** adding to the text input **SLOCCount reports**, `sloccount.sc`.
5. Click on **Save**.

Run the job and review the details. You will now see an overview in the relevant language, as shown in the following screenshot:

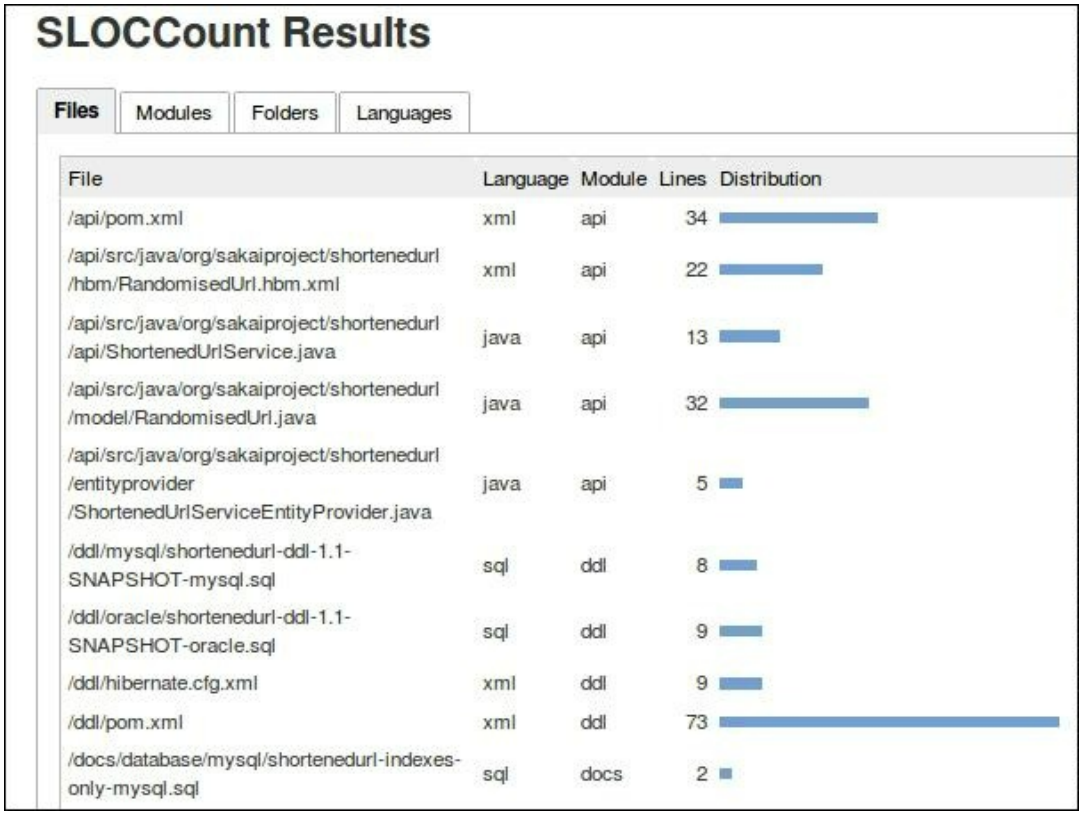
SLOCCount Results

Language	Lines	Lines Delta	Files	Files Delta
java	513		9	
xml	458		10	
sql	38		6	
lisp	11		1	
Total	1,020		26	

At the top level, you will also see a time series of how the lines of code per language evolve over time. It is useful for managers that need to estimate the resources needed to complete projects:



The report also allows you to drill down into specific files. The larger the file, the easier it is for a developer to lose track of the meaning of the code. If you see a particularly large file then it might be worth reviewing, as shown in the following screenshot:



Note

To compare the report you generated with the wider Sakai project, visit https://www.openhub.net/p/sakai#.

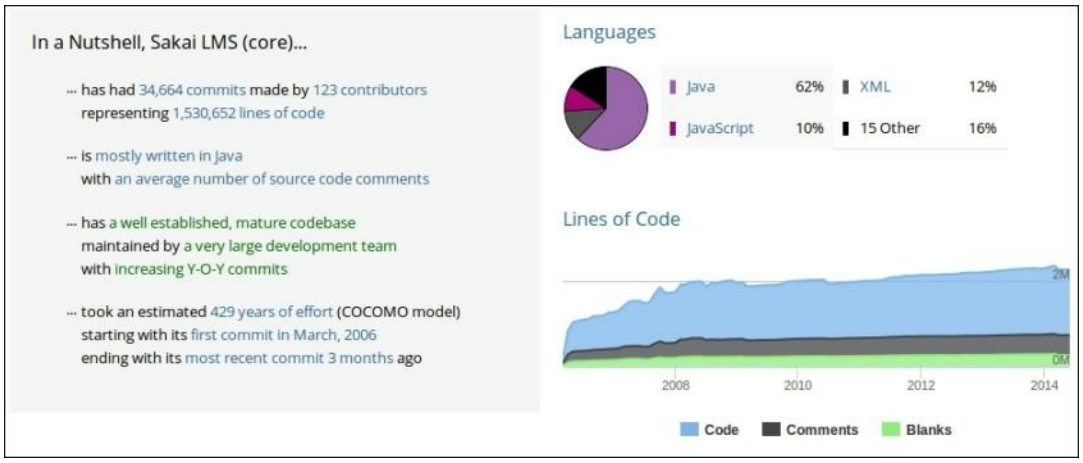
How it works...

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 four-tabbed table summed and sorted by files, modules, 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 URL pointing to open hub (http://blog.openhub.net/2014/07/black-duck-open-hub/), a trusted third-person service. Open hub is a well-known service with well-described privacy rules (http://blog.openhub.net/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 https://www.openhub.net/p/sakai#. The shortened URL service is

one small part of this whole. The combined statistics allow visitors to gain a better understanding of the wider context, as shown in the following screenshot:



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 most over time is person cost (in dollars). You can change the value with the command-line argument `-personcost`.

Goodbye Google code search; hello code.ohoh.net

Google has announced that it has closed its source code search engine. Luckily, code.ohloh.net (formerly koders.com), another viable search engine, announced that it will provide coverage of the code bases described at ohloh.net. With this search engine, you will be able to review a significant selection of open source projects. The search engine complements the code you can search for within your favorite online repositories such as GitHub and Bitbucket.

See also

- The *Knowing your audience with Google Analytics* recipe in Chapter 4, *Communicating Through Jenkins*

- The *Analyzing project data with the R plugin* recipe

Looking for "smelly" code through code coverage

This recipe uses **Cobertura** (http://cobertura.sourceforge.net/) to find code that is not covered by unit tests.

Without consistent practice, writing unit tests will become as difficult as writing debugging information to `stdout`. Most popular Java-specific IDEs have inbuilt 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.

> **Note**
>
> 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
```

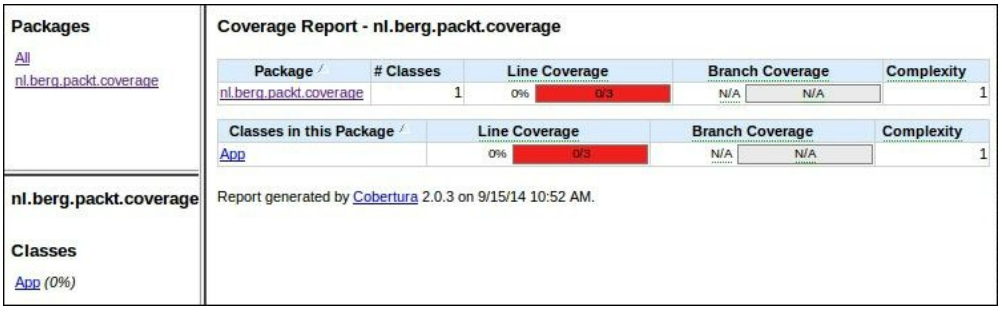
2. Test the code coverage of the unmodified project with the following command:

```
mvn clean cobertura:cobertura
```

3. Review the output from Maven. It will look similar to the following output:

```
-------------------------------------------------------
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 there is no code coverage, as shown in the following screenshot:



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:

```
Public void testApp(){
  new Dicey();
  assertTrue( true );
}
```

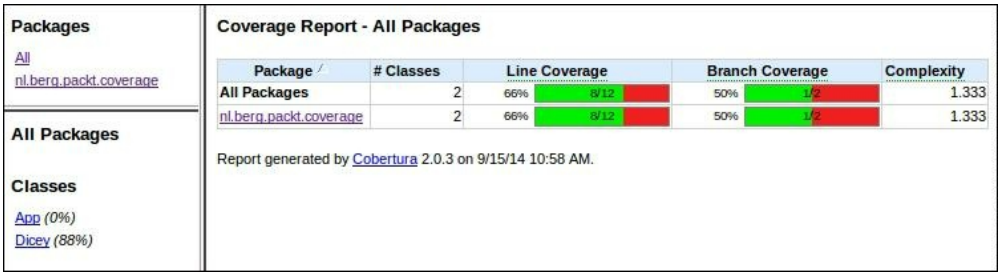
7. Test the code coverage of the JUnit tests with the command:

```
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.AppTestDice <
4 Tests run: 1, Failures: 0, Errors: 0,
Skipped: 0, Time elapsed: 0.033 sec
```

9. In a web browser, open `view /target/site/cobertura/index.html`. Your project now has code coverage and you can see which lines of code have not yet been called, as shown in the following screenshot:



10. Add the following **build** section to your `pom.xml`:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>

 <artifactId>cobertura-maven-
plugin</artifactId>
        <version>2.6</version>
        <configuration>
        <formats>
          <format>xml</format>
```

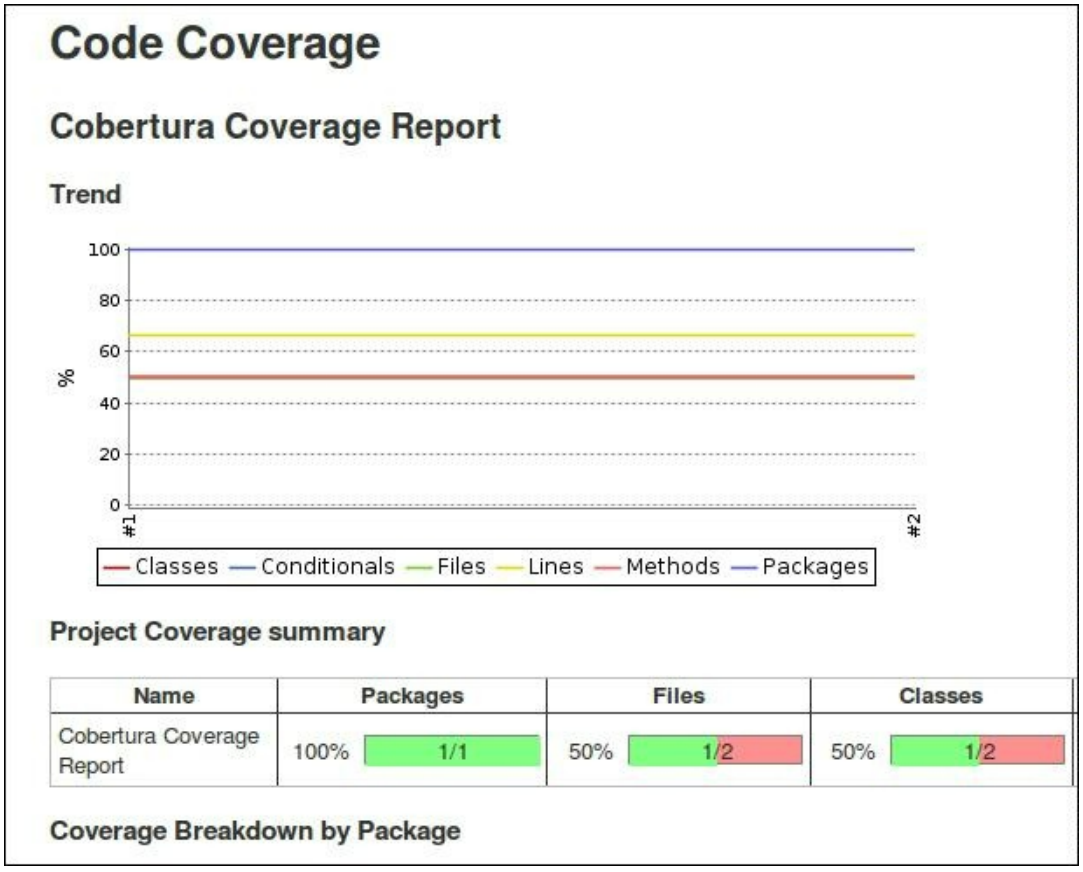
```
            <format>html</format>
          </formats>
        </configuration>
      </plugin>
    </plugins>
  </build>
```

11. Test the code coverage of the JUnit tests with the 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 **Maven** job named `ch5.quality.coverage`.
16. In the **Source Code Management** section, check **Subversion** and add your repository location.
17. In the **Build** section under **Goals and options** add `clean cobertura:cobertura`.
18. Within the **Post-Build actions** section check **Publish Cobertura Coverage Report**. For the Cobertura xml report pattern input add `**/target/site/cobertura/coverage.xml`.
19. Click on **Save**.
20. Click on **Build Now** twice for the job, to generate a trend and then review the results.

The trend graph is a line plot of the percentage of classes, conditions (such as branches of `if` statements), files, lines of code, methods, and packages covered. Jenkins displays each type with a different colored line, as shown in the following screenshot:



How it works...

Cobertura instruments Java bytecode during compilation. The Maven plugin generates both an HTML and XML report. 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 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 fileset `**/target/site/cobertura/coverage.xml`, which states that any report is called `coverage.xml` under any `target/site/cobertura` directory underneath the workspace.

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 then generates a report.

The `testApp` unit test called the constructor for the `Dicey` class. The constructor randomly generates a number from 1 to 6, which mimics a dice, and then 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 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.

The following article is a great example of how to use cobertura and the meaning behind the generated metrics: 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 the `pom.xml` file. You simply need to run the goals `clean emma:emma package` and point the Jenkins plugin at the results.

Note

Another alternative framework is Jacoco (http://www.eclemma.org/index.html). Jacoco is designed as a descendant of Emma. You can find a full description of its Jenkins plugin at: https://wiki.jenkins-ci.org/display/JENKINS/JaCoCo+Plugin.

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 and another for code size or design. By default, three non-controversial PMD rulesets are measured:

- **Basic**: This ruleset has obvious practices that every developer should follow, such as don't ignore the exceptions that are caught
- **Unused code**: This ruleset finds code that is never used, and lines that can be eliminated, avoiding waste and aiding readability
- **Imports**: This ruleset 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).

> ### Tip
>
> **Jenkins bug [Jenkins-22252]**
>
> https://issues.jenkins-ci.org/browse/JENKINS-22252
>
> At the time of writing, Maven 3.2.1 in combination with Jenkins does not work with the PMD plugin. The short-term solution is to use Maven 3.0.5 in your build. However, by the time you read this warning, I expect the issue to have been resolved.
>
> You can automatically install different versions of Java, Maven, or Ant from Jenkins' main configuration screen (`http://localhost:8080/configure`)

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 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
```

4. Review the directory target, and you will notice the results `java-basic.xml`, `java-imports.xml`, `java-unusedcode.xml`, and the aggregated results `pmd.xml`.
5. View the `target/site/pmd.html` file in a web browser.
6. Add the following reporting section to your `pom.xml file`:

```
<reporting>
  <plugins>
    <plugin>

 <groupId>org.apache.maven.plugins</groupId>

 <artifactId>maven-jxr-plugin</artifactId>
      <version>2.3</version>
    </plugin>
   <plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-pmd-plugin</artifactId>
  <version>3.2</version>
  <configuration>
  <targetJdk>1.6</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>
```

7. Test your project with the command:

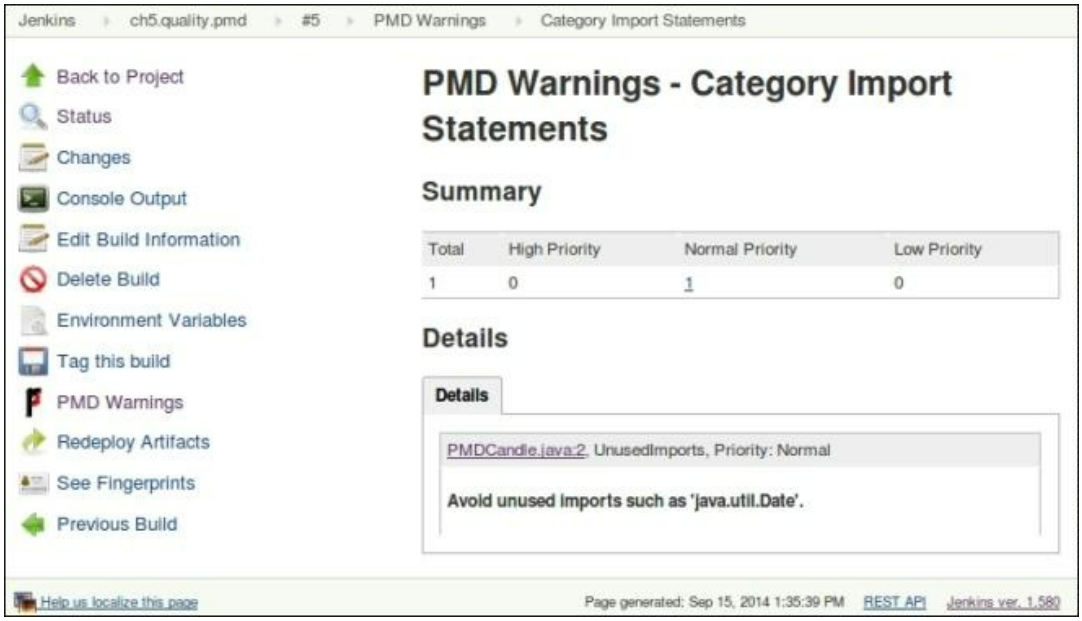
```
mvn clean pmd:pmd
```

8. View in a web browser the file `target/site/pmd.html`, noticing that extra violations have now been found. This is due to the extra rules added to the `pom.xml` file.
9. Run `mvn clean` to remove the `target` directory.
10. Add the source code to your subversion repository.
11. Create a new **Maven** Jenkins job named `ch5.quality.pmd` with the following details:

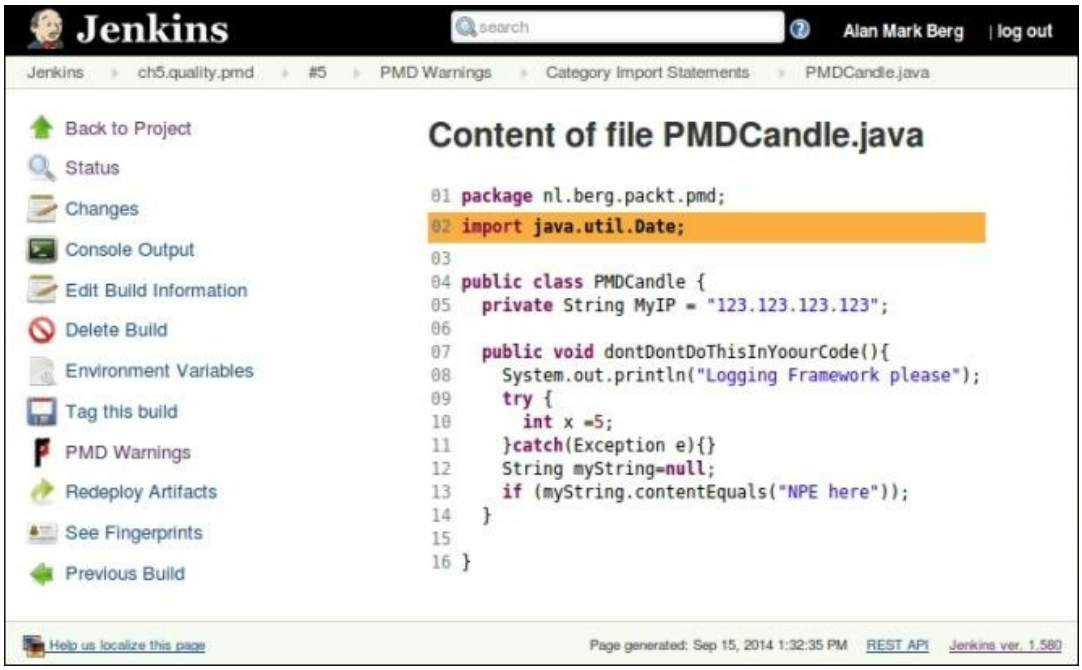
    - **Source Code Management** | **Subversion**: your repository
    - **Build** | **Goals and options**: `clean pmd:pmd`
    - **Build Settings**: **Publish PMD analysis results**

12. Click on **Save**.
13. Click on **Build Now** twice for the job to generate a trend. Review the results.

The top level report summarizes the number of defects and their priorities. It also mentions some of the details underneath, as shown in the following screenshot:



You can then zoom into the code and look at the highlighted areas for defects:



How it works...

The Maven PMD plugin tests a wide range of rulesets. When you download the binary package from the PMD website (http://pmd.sourceforge.net/), you can find the paths of the rulesets by listing the content of the `pmd.jar` file. Under a *NIX system the command to do this is:

```
unzip -l pmd-version.jar | grep rulesets
```

You 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 holds true for the lack of a command to run when the `if` statement is triggered.

Another trivial example is hard-coding 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 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

It is important to understand the importance of the rulesets and shape the Maven configuration to include only the useful ones. If you do not do this for a medium size 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.

> **Note**
>
> You can find the current PMD rulesets at:
> http://pmd.sourceforge.net/rules/index.html

The Don't Repeat Yourself principle

Cut and paste programming, cloning, and then modifying code makes for a refactoring nightmare. If 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 either need to install and configure the DRY plugin (https://wiki.jenkins-ci.org/display/JENKINS/DRY+Plugin) or the Jenkins violations plugin.

> **Note**
>
> If you have downloaded the PMD binary from its website (http://sourceforge.net/projects/pmd/files/pmd/), then in the bin directory you'll find `cpdgui`. It is a Java swing application that allows you to explore your source code for duplications.

See also

- The *Creating custom PMD rules* recipe
- The *Analyzing project data with the R plugin* recipe

Creating custom PMD rules

PMD has two extra features when compared to other static code review tools. The first is the `cpdgui` tool that allows you to look for code that has been cut and pasted from one part of the code base to another. The second, and the one 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 start-up scripts `run.sh designer` 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` file 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>
```

> **Note**
>
> This recipe will only work for version 2.6.

3. In the top level directory, create the file `password_ruleset.xml` with the following 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
  <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";
}
```

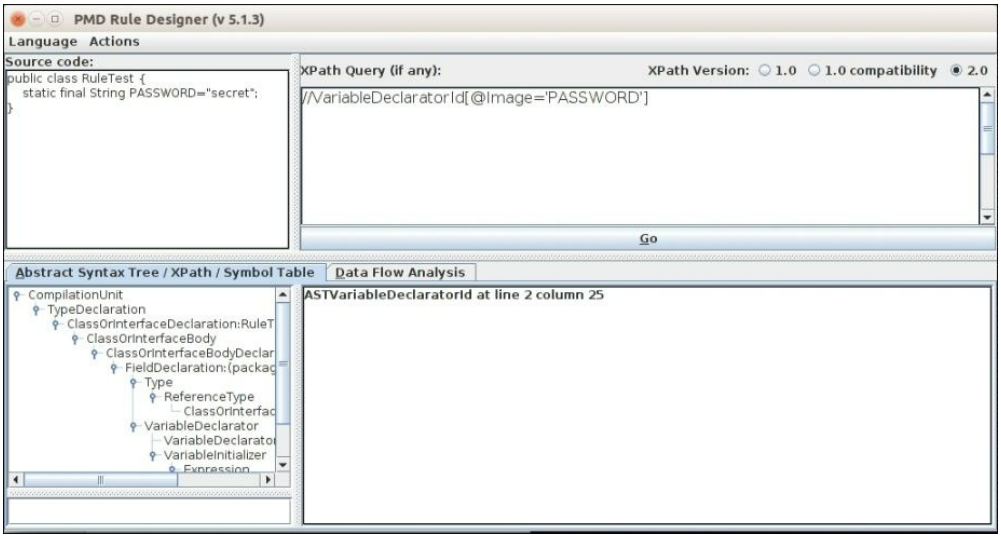
5. Depending on your operating system, run the pmd designer using either the startup script `bin/run.sh designer` or `bin/designer.bat`.
6. Click on the **JDK** option at the top-left of the screen, selecting **JDK 1.6** as the Java version.
7. In the **Source Code** text area, add the example code you 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**, as shown in the following screenshot:



10. Under the **Actions** menu option at the top of the screen, select **Create rule XML**. 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

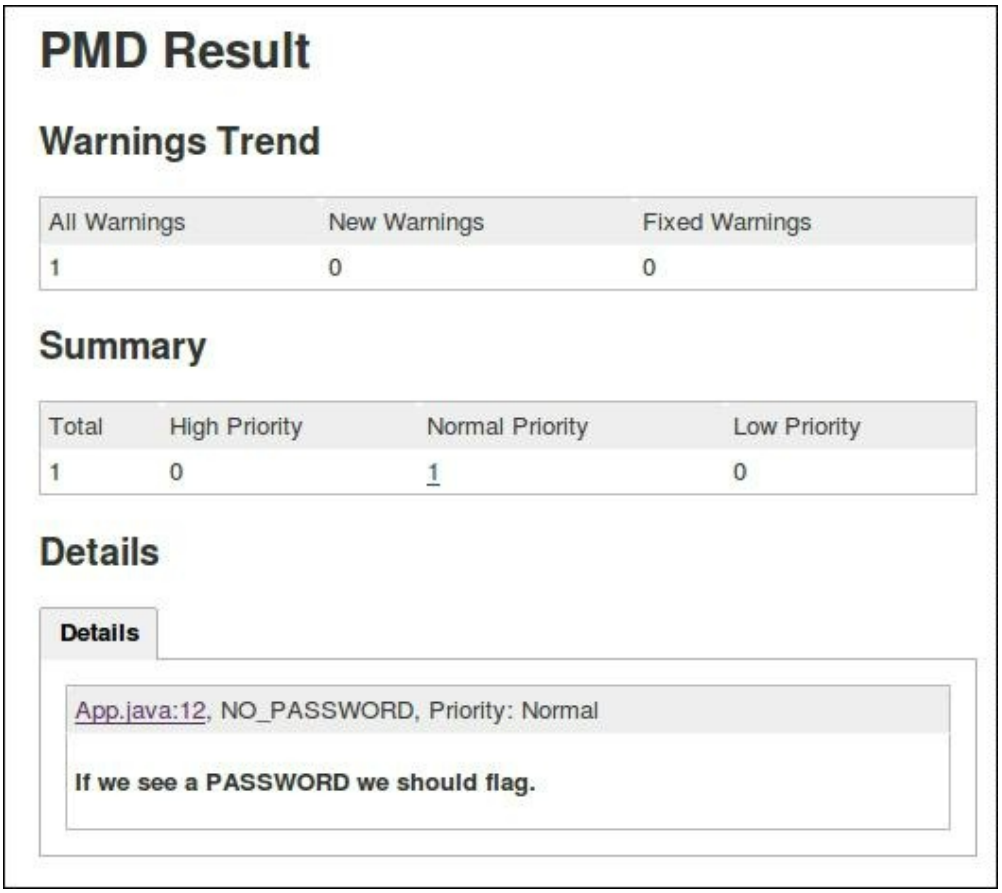
11. Click on **Create rule XML**. The generated XML should have a fragment similar to:

```
<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** job named `ch5.quality.pmdrule`.
15. Under the **Source Code Management** section, check **Subversion,** adding for the **Repository URL** your subversion repository location.
16. Within the **build** section for **Goals and Options** set the value to `clean site`.
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, as shown in the following screenshot:



PMD Result

Warnings Trend

All Warnings	New Warnings	Fixed Warnings
1	0	0

Summary

Total	High Priority	Normal Priority	Low Priority
1	0	1	0

Details

Details

App.java:12, NO_PASSWORD, Priority: Normal

If we see a PASSWORD we should flag.

How it works...

PMD analyzes 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 then test 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 filesystem, 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, 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`. We have seen the rule triggered a number of times in real projects.

We pegged the version of the Maven PMD plugin to version 2.6, so that we are sure the recipe still works after future releases of the plugin.

> **Note**
>
> The PMD homepage 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/pmd-4.3.0/rules/logging-java.html.

There's more...

It would be efficient if static code review tools could make recommendations on 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 repair literals first and also remove some wastage 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

- The *Activating more PMD rulesets* recipe

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 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).

> **Tip**
>
> **Java version**
>
> The FindBugs plugin version 3 requires Java 7 or above.

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 the `pom.xml` file, 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>3.0.0</version>
<configuration>
<FindBugsXmlOutput>true</FindBugsXmlOutput>
<FindBugsXmlWithMessages>true</FindBugsXmlWithMessages>
```

```
<onlyAnalyze>nl.berg.packt.FindBugs_all.candles.*</onlyAnal
yze>
<effort>Max</effort>
</configuration>
</plugin>
</plugins>
</build>
```

3. Create the directories `src/main/java/nl/berg/packt/FindBugs_all/candles`.
4. In the `candles` directory include the `FindBugsCandle.java` file 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 **Maven** project with the name `ch5.quality.FindBugs`.
6. Under the **Source Code Management** section, check the **Subversion** radio box, adding to **Repository URL** your repository URL.
7. Within the **build** section add `clean compile findBugs:findBugs` for **Goals and options**.
8. In the **post-build action** option, select **Publish FindBugs analysis results**.
9. Click on **Save**.
10. Run the job.
11. Review the results.

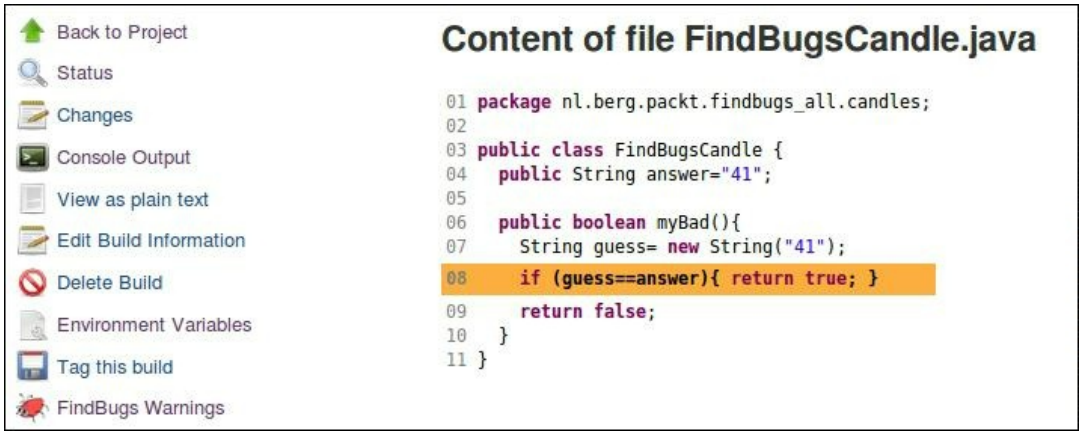
The first page is a summary page that allows you to efficiently zoom in on the details, as shown in the following screenshot:



Looking at a category such as **BAD_PRACTICE** allows you to review the descriptions of each error type triggered:



You can then review the associated code. The highlighted code is useful for focusing your attention, as shown in the following screenshot:



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 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.

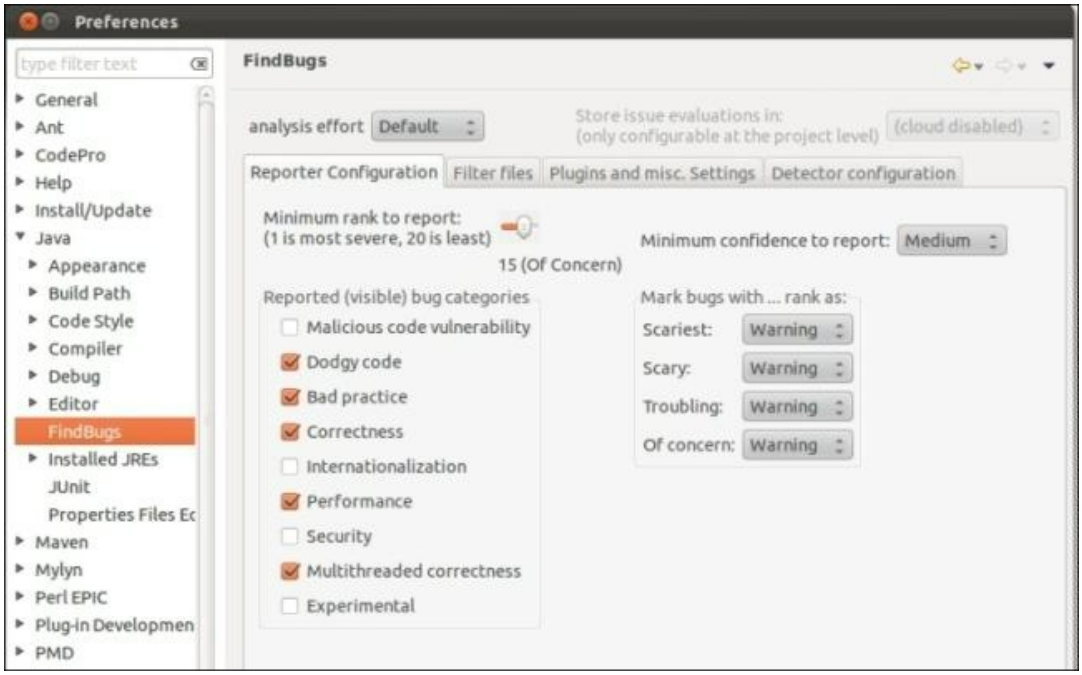
There's more...

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 side menu. On the right-hand side you will see the **Reported (Visible) bug categories** option under the **Reporter Configuration**. You can now tweak the visible categories, as shown in the following screenshot:



The Xradar and Maven dashboards

There are alternatives though to aging 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 named *Setting up the Maven dashboard* in Chapter 4, *Reporting and Documentation*, for this in the book *Apache Maven 3 Cookbook, Srirangan, Packt Publishing* (https://www.packtpub.com/application-development/apache-maven-3-cookbook).

Xradar is another example of a dashboard (http://xradar.sourceforge.net/usage/maven-plugin/howto.html) and QALab is a third (http://qalab.sourceforge.net/multiproject/maven2-qalab-plugin/index.html).

See also

- The *Enabling extra FindBug rules* recipe
- The *Finding security defects with FindBugs* recipe
- The *Activating more PMD rulesets* recipe

Enabling extra FindBug 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 then 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.

> **Tip**
>
> **fb-contrib version change**
>
> In the following recipe, Maven automatically downloads a library file (`.jar`). The build might fail because the developers have incremented the version number. In this case, to find the correct filename browse http://downloads.sourceforge.net/project/fb-contrib/Current/.

How to do it...

1. Copy the top-level `pom.xml` file 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>3.0.0</version>
<configuration>
<FindBugsXmlOutput>false</FindBugsXmlOutput>
<FindBugsXmlWithMessages>true</FindBugsXmlWithMessages>
<onlyAnalyze>nl.berg.packt.FindBugs_all.candles.*</onlyAnal
yze>
<pluginList>http://downloads.sourceforge.net/project/fb-
contrib/Current/fb-contrib-
6.0.0.jar</pluginList>
<effort>Max</effort>
</configuration>
</plugin>
```

3. In the `src/main/java/nl/berg/packt/fingbugs_all/candles` directory add the following code snippet to the `FindBugsFBCandle.java` Java class:

```
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** job with the name `ch5.quality.FindBugs.fb`.
6. Under the **Source Code Management** section, check the **Subversion** radio box adding for the **Repository URL** the URL to your code.
7. In the **build** section set:
    - **Root POM** to `pom_fb.xml`
    - **Goals and options** 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. When the job is finished building, review the **FindBugs Warnings** link. You will now see a new warning **USBR_UNNECESSARY_STORE_BEFORE_RETURN**.

How it works...

To include external detectors, you added an extra line to the FindBugs Maven configuration, as shown:

```
<pluginList>http://downloads.sourceforge.net/project/fb-contrib/Current/fb-contrib-6.0.0.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 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 anti-pattern triggers the **USBR_UNNECESSARY_STORE_BEFORE_RETURN** pattern that is described on the homepage of the `fb-contrib` project.

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 the book Java puzzlers by Joshua Bloch and Neal Gafter (http://www.javapuzzlers.com/).

See also

- The *Finding bugs with FindBugs* recipe
- The *Finding security defects with FindBugs* recipe
- The *Activating more PMD rulesets* recipe

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 *Failing Jenkins jobs based on JSP syntax errors* recipe in Chapter 3, *Building Software*, or use the provided project downloadable from the Packt Publishing website.

How to do it...

1. Edit the `pom.xml` file just under `<plugins>` within `<build>` to include the FindBugs plugin and add the following content:

```
<plugins>
<plugin>
<groupId>org.codehaus.mojo</groupId>
<artifactId>findBugs-maven-plugin</artifactId>
<version>3.0.0</version>
<configuration>
<FindBugsXmlOutput>true</FindBugsXmlOutput>
<FindBugsXmlWithMessages>true</FindBugsXmlWithMessages>
<effort>Max</effort>
</configuration>
</plugin>
```

2. Create the directory structure `src/main/java/nl/berg/packt/finbugs_all/candles`.
3. Add the Java file `FindBugsSecurity.java` with the 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** 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. Under the **Build Settings** section, check **Publish FindBugs analysis results**.
9. Click on **Save**.
10. Run the job.
11. 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.
12. Copy `src/main/webapp/index.jsp` to `jsp/jsp.index_jsp`.
13. Commit to your subversion repository.
14. Run the job again.
15. View the results under the **FindBugs Warning** link. You will now be able to view the JSP source code.

How it works...

JSPs are first translated from text into Java source code and then compiled. FindBugs works by parsing compiled Java bytecode.

The original JSP project has a massive security flaw. It trusts input from the Internet. This led 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 reducing the risk. FindBugs discovers the defect and warns with `XSS_REQUEST_PARAMETER_TO_JSP_WRITER`. The Jenkins FindBugs plugin details the bug type, as you had turned messages on in configuration with:

```
<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 to the JSP file. Despite these limitations, FindBugs discovers useful information about JSP defects.

> **Note**
>
> An alternative to JSP bug detection is PMD. From the command line, you can configure it to scan JSP files only with the option `-jsp`, see: http://pmd.sourceforge.net/jspsupport.html

There's more...

Although FindBugs has rules that sit under the category of 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, as shown in the following code:

```
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 in a **Denial Of Service** (**DOS**) attack.

The other attack captured in the standard candle is the ability to change 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 and pass the copy to the calling method.

> **Note**
>
> The OWASP project provides a wealth of information on the subject of testing security, find the following link:
>
> https://www.owasp.org/index.php/Category:OWASP_Java_Project

See also

- The *Finding bugs with FindBugs* recipe
- The *Enabling extra FindBug rules* recipe
- The *Activating more PMD rulesets* recipe
- The *Configuring Jetty for integration tests* recipe in 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.

Web browsers are not fussy. You can have broken templates in your applications that generate HTML that works on one browser, but are ugly on another. Validation improves consistency and captures non-trivial but difficult-to-find issues early.

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 build step,** 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**, clicking on the `unicorn_output.html` link and then `markup-validator_errors.properties`. For the property's file content, you will see content similar to `YVALUE=2`.
7. **Configure** the project. In the **Post-build Actions** section, check **Plot build data**, adding 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 the **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 property files where the values are then picked up by the plotting plugin (refer to the *Plotting alternative code metrics in Jenkins* recipe in Chapter 3, *Building Software*). If you see a sudden surge in warnings, then review the HTML pages for repetitive defects.

There's more...

It is quite difficult to obtain 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 log files for unexpected errors. You can automate log parsing using the *Deliberately failing builds through log parsing* recipe in Chapter 1, *Maintaining Jenkins*.

Note

For details on incrementally validating your online content, visit http://www.w3.org/QA/2002/09/Step-by-step.

Reporting with JavaNCSS

JavaNCSS (http://javancss.codehaus.org/) is a software metrics tool that calculates two types of information: the first is the total 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.

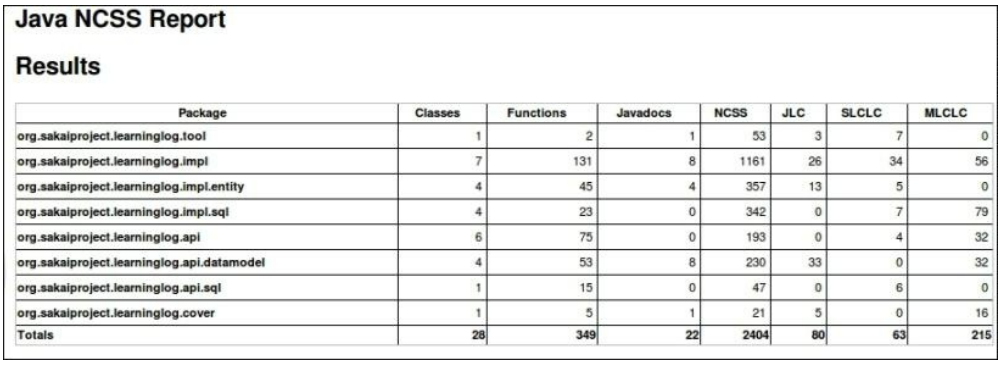
> **Note**
>
> **NCSS** stands for **Non Commenting Source Statements** and is the number of lines of code minus the comments and extra new lines.

Getting ready

Install the JavaNCSS plugin (https://wiki.jenkins-ci.org/display/JENKINS/JavaNCSS+Plugin).

How to do it...

1. Create a **Maven** 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 the **Build Triggers,** making sure none are activated.
5. Under the **build** section for **Goals and options,** type `clean javancss:report`.
6. Under the **Build Settings** section, check **Publish Java NCSS Report**.
7. Click on **Save**.
8. Run the job.
9. Review the **Java NCSS Report** link.
10. Review the top level `pom.xml` file 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 the Sakai project's learning log tool subversion repository. The project is a multimodule with the API separated from the implementation.

JavaNCSS needs no compiled classes or modifications to the Maven `pom.xml` file; this makes for a simple cycle. The job ran a Maven goal, publishing the report through the JavaNCSS Jenkins plugin.

Reviewing the report, the implementation has a much great number of lines of active code relative to other packages. Documentation of APIs is vital for reuse of code by other developers. Significantly, there are no JavaDoc lines in the API.

The abbreviations in the summary table have the following meanings:

- **Classes**: This is the number of classes in the package.
- **Functions**: This is the number of functions in the package.
- **JavaDocs**: This is 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**: This is the number of non-commented lines of source code.
- **JLC**: This is the number of lines of JavaDoc.
- **SLCLC**: This is the number of lines that include only a single comment.
- **MLCLC**: This is the number of lines of source code that are part of multiline comments.

The build summary displays information about changes (deltas) between the current job and the previous one, for example:

```
classes (+28)
functions (+349)
ncss (+2404)
```

```
javadocs (+22)
javadoc lines (+80)
single line comments (+63)
multi-line comments (+215)
```

The + symbol signals that the code is added, - signals deleted. If you see a large influx of code, but a lower than usual influx of JavaDoc, then either the code is autogenerated or more likely is being rushed to market.

There's more...

When you have gotten 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, and https://wiki.jenkins-ci.org/display/JENKINS/JDepend+Plugin).

One of the most important metrics JDepend generates is **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 code style using an external pom.xml file

If you just want to check 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 checks most 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).

> **Tip**
>
> If you have issues with this recipe due to an **illegalAccessError on AbstractMapBasedMultimap** error, then this probably is due to the bug reported in **Jenkins-22252** (https://issues.jenkins-ci.org/browse/JENKINS-22252). The current solution is to run with version 3.0.5 of Maven.

How to do it...

1. Create a directory named `/var/lib/jenkins/OVERRIDE`.
2. Make sure that the directory is owned by the Jenkins user and group `sudo chown jenkins:jenkins /var/lib/jenkins/OVERRIDE`.
3. Create the file `/var/lib/Jenkins/OVERRIDE/pom_checkstyle.xml` with the following content:

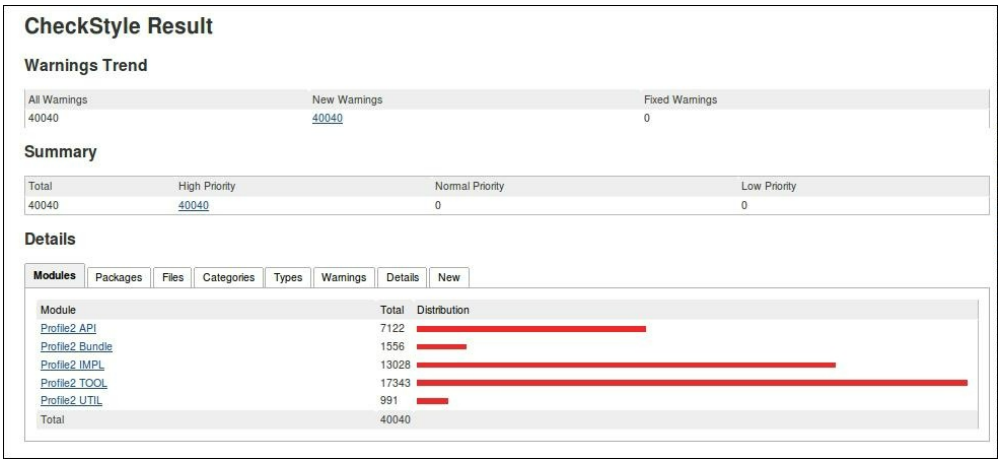
```
<project
xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"

 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>
```

4. Make sure the file is owned by the Jenkins user and group `sudo chown jenkins:jenkins /var/lib/jenkins/OVERRIDE/pom_checkstyle.xml`.
5. Create a **Maven** job with the name `ch5.quality.checkstyle.override`.
6. Under the **Source Code Management** section, check **Subversion** and add the subversion repository https://source.sakaiproject.org/svn/profile2/tags/profile2-1.4.5.
7. In the **Pre-steps** section for the **Add pre-build** step, select **Execute shell**.
8. In the command text area, add `cp /var/lib/Jenkins/OVERRIDE/pom_checkstyle.xml`.
9. Under the **build** section add:

    - **Root POM**: `pom_checkstyle.xml`
    - **Goals and options**: `clean checkstyle:checkstyle`

10. Under the **Build Settings** section, check **Publish Checkstyle analysis results**.
11. Click on **Save**.
12. Run the job a number of times, reviewing the output.



How it works...

The **profile2** tool is used by 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 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.

The job then copies the `pom_checkstyle.xml` file to the main workspace. Checkstyle was not configured in detail in the `pom_checkstyle.xml` file because we were only interested in the overall trend. However, if you want to zoom into the details, Checkstyle can be configured to generate results based on specific metrics 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 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/ap
```

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, refer to the *Remotely triggering jobs through the Jenkins API* recipe in Chapter 3, *Building Software*.

See also

- The *Faking Checkstyle results* recipe

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 compared to using the *Plotting alternative code metrics in Jenkins* recipe in Chapter 3, *Building Software*, is the location that the results are displayed. You can then 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
```

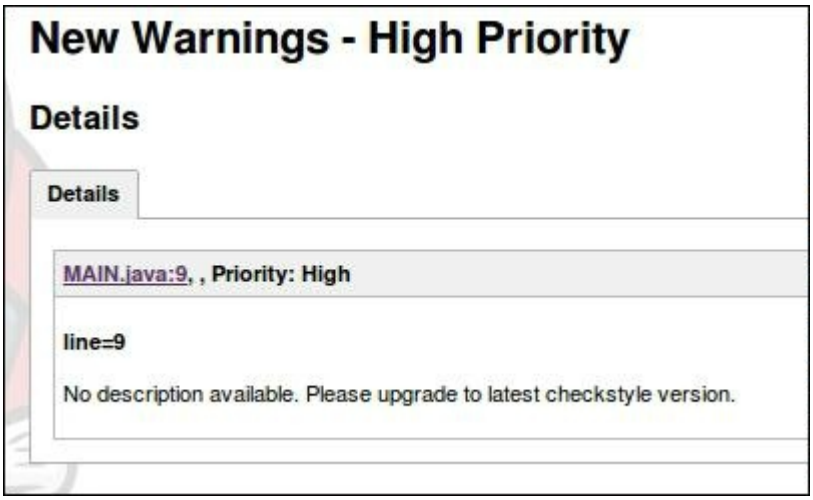
2. Make the directories `src/main/java`.
3. Add the Java file `src/main/java/MAIN.java` with the following content:

```
//line 1
public class MAIN {
//line 3
public static void main(String[] args) {
  System.out.println("Hello World"); //line 5
}
//line 7
```

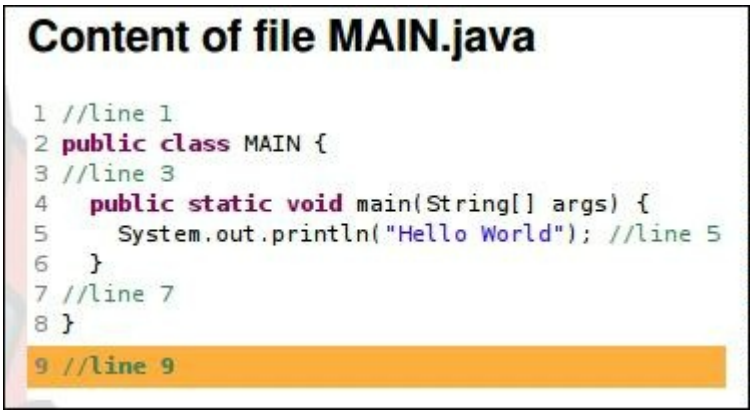
```
}
//line 9
```

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** and add the **Repository URL**: your repo URL.
7. Within the **build** section, select the **Build step** as **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, reviewing the results and trend.

The top level report mentions your new rule:



Clicking on the code link **MAIN.java** takes you to the code page and highlights the error line randomly selected by the Perl code, as shown in the following screenshot:



How it works...

The plugins used in this chapter store their information in XML files. The Checkstyle XML structure is the simplest of all the tools and hence the XML format chosen for our generated fake results.

The Perl code creates a simple XML results file that chooses a line between 1...9 to fail. The format outputted is similar to the following code:

```
<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 a 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

- The *Checking style using an external pom.xml* recipe

Integrating Jenkins with SonarQube

SonarQube, previously known as Sonar, is a rapidly evolving application for reporting quality metrics and finding code hot spots. This recipe details how to generate code metrics through a Jenkins plugin, and then push them directly to a Sonar database.

Getting ready

Install the Sonar plugin (http://docs.codehaus.org/display/SONAR/Jenkins+Plugin).

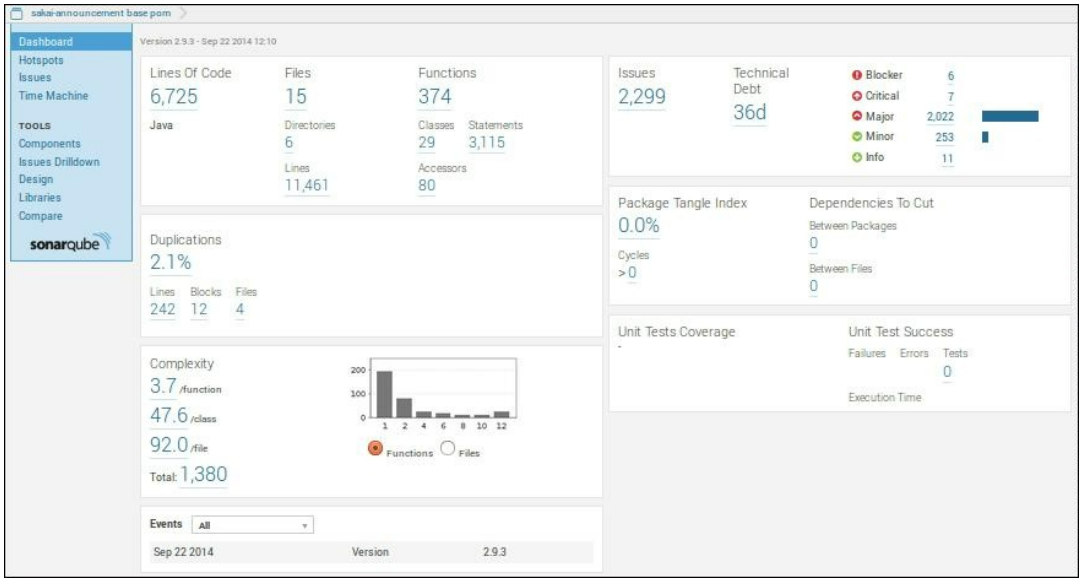
Download and unpack SonarQube. You can run it directly from within the bin directory, selecting the OS directory underneath. For example, the Desktop Ubuntu start-up script is `bin/linux-x86-32/sonar.sh console`. You now have an insecure default instance running on port `9000`. For more complete installation instructions, review:

http://docs.codehaus.org/display/SONAR/Setup+and+Upgrade and http://docs.sonarqube.org/display/SONAR/Installing

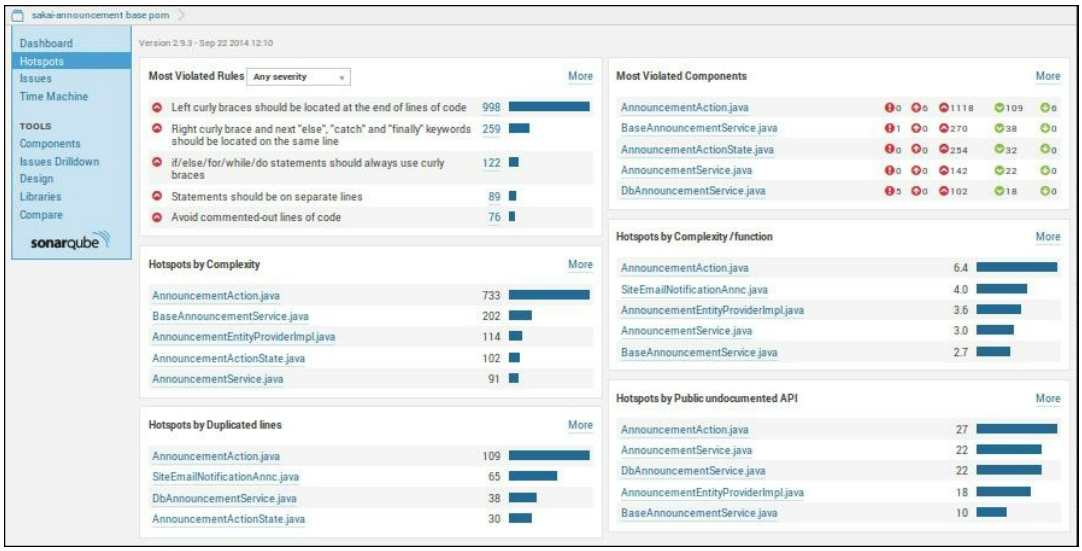
How to do it...

1. Within the main Jenkins configuration (`/configure`), in the **Sonar** section add `localhost` for **Name**.
2. Click on **Save**.
3. Create a **Maven** job named `ch5.quality.sonar`.
4. Under the **Source Code Management** section for the **Repository URL**, add https://source.sakaiproject.org/svn/announcement/tags/announcement-2.9.3.
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 reports.

The top level of the report delivers a quick summary of the key quality metrics, as shown in the following screenshot:



From the left-hand side menu, you can drill down into the details:



How it works...

The source code is that of an announcement tool used within Sakai. The project is a multimodule project with some relatively complex details.

The default SonarQube instance is preconfigured 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 for generating results itself.

The job ran Maven first to clean out the old compiled code from the workspace, and then ran the `install` goal that 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, all currently differentiate it from Jenkins.

SonarQube 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/Plugin+Library

The plugins include a number of features that are equivalent to the ones you can find in Jenkins. Where Sonar is noticeably different is the governance plugins, where code coverage moves to center stage in defending the quality of a project.

Alternative aggregator – 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 it to see if it fulfills your quality metric needs.

See also

- The *Looking for "smelly" code through code coverage* recipe
- The *Activating more PMD rulesets* recipe
- The *Reporting with JavaNCSS* recipe

Analyzing project data with the R plugin

This recipe describes how to use R to process metrics on each file in your project workspace. The recipe does this by traversing the workspace and collecting a list of files of a particular extension such as Java. The R script then analyzes each file individually and finally plots the results in a graphical format to a PDF file. The workflow is common to almost all quality-related analysis of software projects. This recipe is easily customized for tasks that are more complex.

In this example, we are looking at the size in words of the text files, printing to the console the names of large files and plotting the sizes of all files. From the visual representation, you can easily see which files are particularly large. If your property file is much larger than the other property files, it is probably corrupt. If a Java file is too large, it is difficult to read and understand.

Getting ready

It is assumed that you have followed the *Simplifying powerful visualizations using the R plugin* recipe in Chapter 4, *Communicating Through Jenkins*, and have already installed the R plugin (https://wiki.jenkins-ci.org/display/JENKINS/R+Plugin).

How to do it...

1. Create a free-style job with the name `ch5.R.project.data`.
2. In the **Source Code Management** section, select **Subversion**.
3. Add the **Repository URL** as https://source.sakaiproject.org/svn/profile2/trunk.
4. In the **build** section, under **Add build step**, select **Execute R script**.
5. In the **Script text** area add the following code:

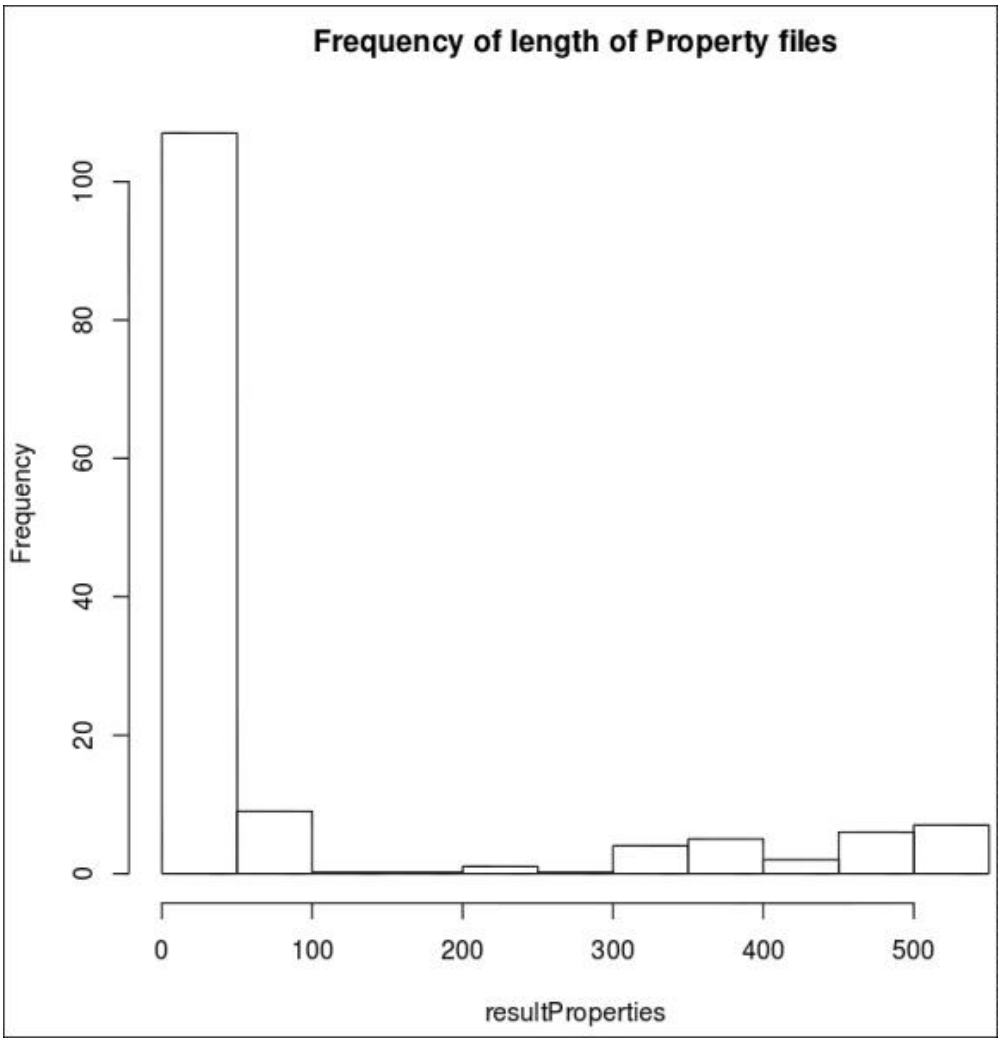
```
processFile <- function(file){
  text <- readLines(file,encoding="UTF-8")
  if (length(text)> 500) print(file)
  length(text)
}
javaFiles <-
list.files(Sys.getenv('WORKSPACE'), recursive
= TRUE, full.names = TRUE, pattern =
"\\.java$")
propertiesFiles <-
list.files(Sys.getenv('WORKSPACE'), recursive
= TRUE, full.names = TRUE, pattern =
"\\.properties$")
```

```
resultJava <- sapply(javaFiles, processFile)
resultProperties <-
sapply(propertiesFiles,processFile)
warnings()
filename
<-
paste('Lengths_JAVA_',Sys.getenv('BUILD_NUMBER'),'.pdf',sep
="")
pdf(file=filename)
hist(resultJava,main="Frequency of length of
JAVA files")

filename
<-
paste('Lengths_Properties_',Sys.getenv('BUILD_NUMBER'),'.pdf',sep=""
pdf(file=filename)
hist(resultProperties,main="Frequency of
length of Property files")
```

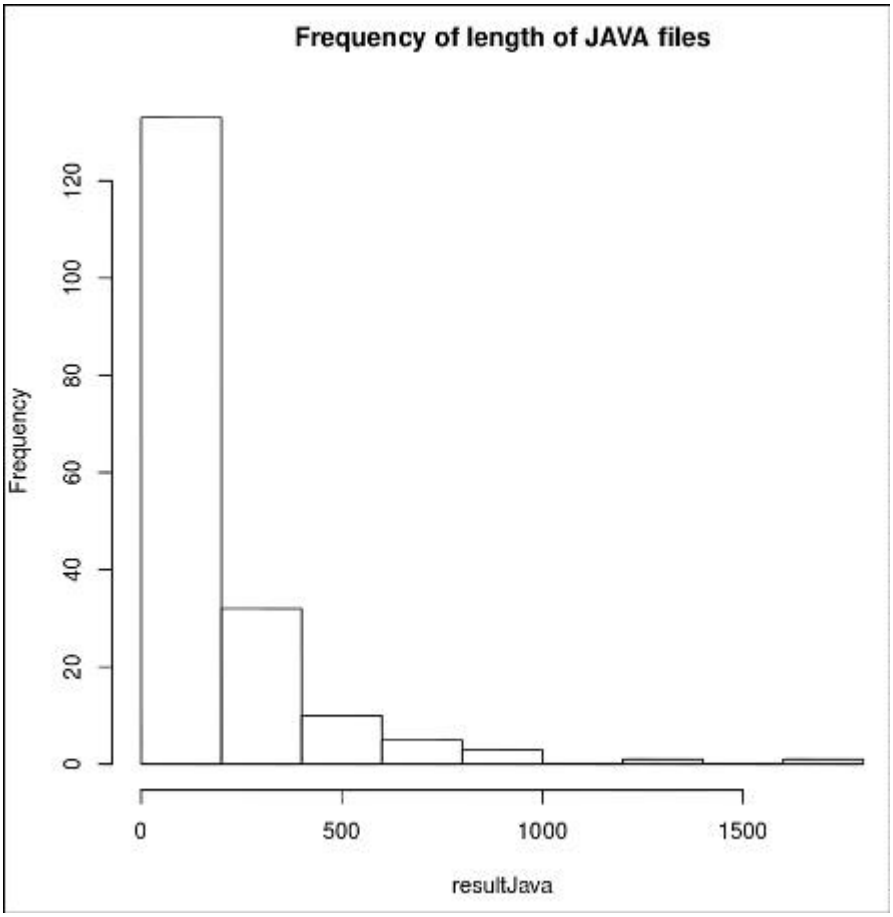
6. Click on the **Save** button.
7. Click on the **Build Now** icon.
8. Review the console output from the build. It should appear similar to the following:

```
At revision 313948
no change for
https://source.sakaiproject.org/svn/profile2/trunk
since the previous build
[ch5.R.project.data] $
Rscript /tmp/hudson7641363251840585368.R
[1]
"/var/lib/jenkins/workspace/ch5.project.data/api/src/java/org/sakaip
[1]
"/var/lib/jenkins/workspace/ch5.project.data/impl/src/java/org/sakai
[1]
"/var/lib/jenkins/workspace/ch5.project.data/impl/src/java/org/sakai
14: In readLines(file, encoding = "UTF-8") :
  incomplete final line found on
'/var/lib/jenkins/workspace/ch5.project.data/tool/src/java/org/apach
Finished: SUCCESS
```

9. Visit the workspace and review the files `Lengths_Properties_1.pdf`, `Lengths_JAVA_1.pdf`.



Notice the straggler files with a large number of lines. Property files should be of roughly similar length, as they contain, in this case, the international translations for the GUI.



This feels like a well-balanced project, as there are only a few files that have a large number of lines of code.

How it works...

You loaded in the profile2 tool from subversion https://source.sakaiproject.org/svn/profile2/trunk. This code is used by millions of students around the world and represents mature, realistic production code.

Within your R script, you defined a function that takes a filename as input and then reads the file into a text object. The function then checks to see whether the number of lines is greater than 500. If it is greater than 500 lines then the filename is printed to the console output. Finally, the function returns the number of lines in the text file.

```
processFile <- function(file){
  text <- readLines(file,encoding="UTF-8")
  if (length(text)> 500) print(file)
  length(text)
}
```

Next, the script discovers the property and Java files under the workspace. The file search is filtered by the value defined in the `pattern` argument. In this case, `.java`:

```
javaFiles <- list.files(Sys.getenv('WORKSPACE'),
recursive = TRUE, full.names = TRUE, pattern =
"\\.java$")
propertiesFiles <-
list.files(Sys.getenv('WORKSPACE'), recursive =
TRUE, full.names = TRUE, pattern =
"\\.properties$")
```

The list of filenames is passed one name at a time to the `processFile` function you have previously defined. The results are a list of file lengths that are stored in the `resultJava` and `resultProperties` objects:

```
resultJava <- sapply(javaFiles, processFile)
resultProperties <-
sapply(propertiesFiles,processFile)
```

The `warnings()` function produces a list of issues generated while running the `sapply` command:

```
14: In readLines(file, encoding = "UTF-8") :
  incomplete final line found on
'/var/lib/jenkins/workspace/ch5.project.data/tool/src/java/org/apache/wic
```

This is stating that a new line was expected at the end of the file. It is not a critical issue. Showing the warnings is a helpful approach to discovering corrupted files.

Finally, we generate two histograms of the results, one for the Java file and the other for the properties files. The filename is created from a constant string followed by the `BUILD_NUMBER` environment variable that is set uniquely for each build. The `pdf` function tells R that the output is to be stored in a PDF file and the `hist` function draws a histogram of the results:

```
filename
<-
paste('Lengths_JAVA_',Sys.getenv('BUILD_NUMBER'),'.pdf',sep="")
pdf(file=filename)
hist(resultJava,main="Frequency of length of JAVA
files")
```

There's more...

When writing R code for processing your files, don't reinvent the wheel. R has many libraries for manipulating text. The stringi library is one example (http://cran.r-project.org/web/packages/stringi/stringi.pdf). Here is some example code that counts the number of words in a text file:

```
library(stringi)
processFile <-function(file){
stri_stats_latex(readLines(file,encoding="UTF-8"))
}
results<-processFile(file.choose())
paste("Number of words in file:", results[4])
```

The script defines the function `processFile`. The function requires a filename. The file is read into the `stri_stats_latex` function. This function is included in the `stringi` library. It returns a summary of the file as a vector (a series of numbers).

The `file.choose()` function pops up a dialog that allows you to browse your file system and choose a file. The call returns the fully qualified path to the file. It passes the value to the `processFile` function call. The results are stored in the results vector. The script then prints out the fourth number that is the number of words in the file.

> **Note**
>
> Another interesting R package for text mining is `tm`: (http://cran.r-project.org/web/packages/tm/tm.pdf). The `tm` package has the ability to load a set of text files and analyze them in many different ways.

See also

- The *Simplifying powerful visualizations using the R plugin* recipe in Chapter 4, *Communicating Through Jenkins*
- The *Adding a job to warn of storage use violations through log parsing* recipe in Chapter 1, *Maintaining Jenkins*

Chapter 6. Testing Remotely

In this chapter, we will cover the following recipes:

- Deploying a WAR file from Jenkins to Tomcat
- Creating multiple Jenkins nodes
- Custom setup scripts for slave 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 run performance and functional tests against a web application and web services. Two typical setup recipes are included: the first is the deployment of a WAR file through Jenkins to an application server, the second is the creation of multiple slave nodes, ready to move the hard work of testing away from the master node.

Remote testing through Jenkins considerably increases the number of dependencies in your infrastructure and thus the maintenance effort. Remote testing is a problem that is 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**: This is a wiki with which you can write different types of tests. Having a wiki-like language to express and change tests on-the-fly gives functional administrators, consultants, and the end user 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**: This 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 XML format. JMeter is executable through a Maven or Ant script. 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**: This is the de-facto industrial standard for the functional testing of web applications. With Selenium IDE, you can record your actions within Firefox or Chrome, saving them in HTML format to replay later. The tests can be re-run through Maven using Selenium RC (Remote Control). It is common to use Jenkins slaves with different OSes and browser types to run the tests. The alternative is to use Selenium grid (https://code.google.com/p/selenium/wiki/Grid2).
- **Selenium 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**: This simplifies the creation of functional tests for web services. The tool can read **WSDL** (**Web Service Definition Language**) 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 through a scheduler. No polling of the SCM is needed. The advantages of this approach are a distributed team that knows exactly when and at which URL a new build exists, and that the deployment script is thin.
- Deploy to an application server. First, package the web application in Jenkins and then the deploy is 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 latest version of Tomcat 7 and unpack (http://tomcat.apache.org/download-70.cgi).

How to do it...

1. Create a Maven project for a simple WAR file from the command line:

```
mvn archetype:generate
-
DgroupId=nl.berg.packt.simplewar
-
DartifactId=simplewar -Dversion=1.0-SNAPSHOT
-DarchetypeArtifactId=maven-archetype-webapp
```

2. Commit the newly created project to your Git or subversion repository.

3. To avoid conflict with Jenkins listening on port `8080`, under the Tomcat root directory edit `conf/server.xml` change 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 project named `ch6.remote.deploy`.
7. Under the **Source Code Management** section, check the **Subversion** radio box, adding your own subversion repository URL to the **Repository URL**.
8. In the **build** section, for **Goals and options** add `clean package`.
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 output similar to: **java.io.IOException: Server returned HTTP response code: 401 for URL: http://localhost:38887/manager/text/list**.
13. Edit `conf/tomcat-users.xml` by adding the following 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 output similar to: **Oct 06, 2014 9:37:11 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/`, as shown in the following screenshot:



> **Note**
>
> A potential gotcha: if you misspell the name of the WAR file in your post build configuration then it will fail silently and the build will still succeed.

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 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 with the required roles. In fact, the new user has more power than is needed for deployment. The user has the power to review JMX data for monitoring. This helps you with debugging later.

When deploying in production, use an SSL connection to avoid sending unencrypted passwords 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 when running on a Linux platform, 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

- The *Configuring Jetty for integration tests* recipe in Chapter 3, *Building Software*

Creating multiple Jenkins nodes

Testing is a heavyweight 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. There are other reasons as well to farm out testing, for example for functional testing when you want to use different web browsers under different OS or run .NET applications natively.

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 Linux-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 *Using a test Jenkins instance* recipe, Chapter 1, *Maintaining Jenkins*.

How to do it...

1. From the command line of the slave node create the user `jenkins-unix-nodex`:

```
sudo adduser jenkins-unix-nodex
```

2. Generate a private key and a public certificate for the master Jenkins with an empty passphase:

```
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 and 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`:

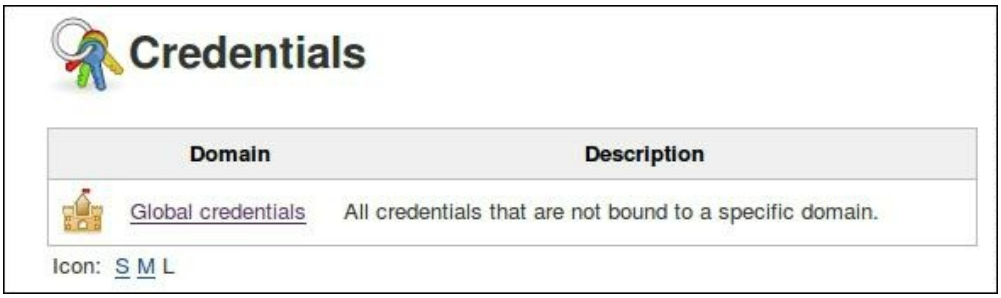
```
sudo chown
jenkins-unix-nodex:jenkins-unix-
nodex /home/jenkins-unix-
nodex/.ssh/authorized_keys
```

5. Test that you can log in without a password as `jenkins` to `jenkins-unix-nodex`.`jenkins-unix-nodex`:

```
sudo –u 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.
```

> **Note**
>
> You will need to accept the key fingerprint.

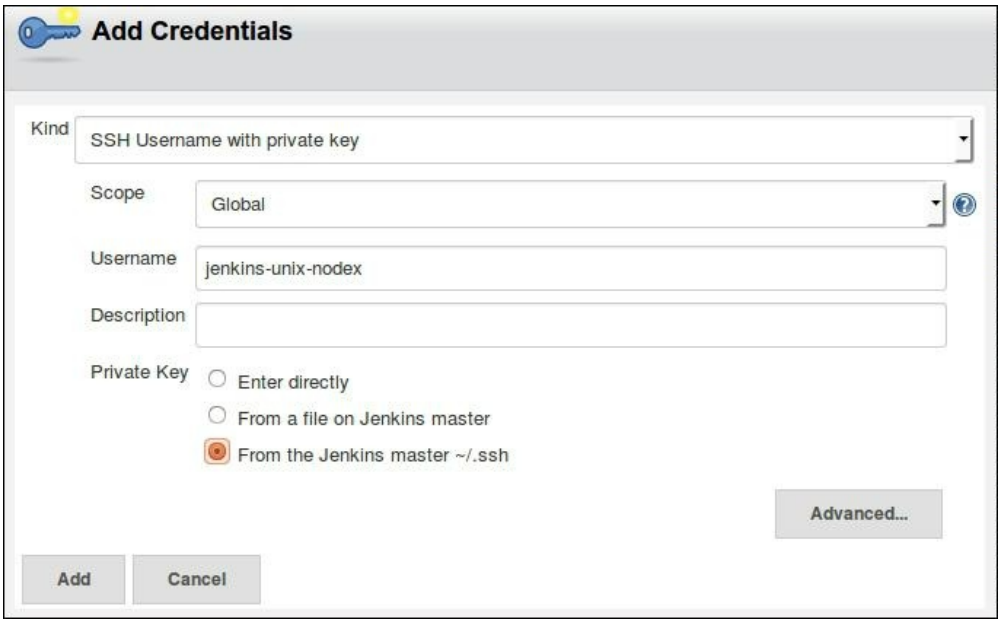
6. Log in to Jenkins.
7. Visit the credentials store (`localhost:8080/credential-store`).
8. Click on the **Global credentials** link, as shown in the following screenshot:



9. Click on **Add Credentials**.
10. Add the following details:

    - **Kind**: SSH username with private key
    - **Scope**: **Global**
    - **Username**: `jenkins-unix-nodex`
    - **Private Key**: **From the Jenkins master ~ /.ssh**

11. Click on **Save**.



12. Visit the **MultiSlave Config Plugin** under **Manage Jenkins** (`localhost:8080/multi-slave-config-plugin/?`).
13. Click on **Add Slaves**.
14. Add to **Create slaves by names separated with space**: **unix-node01** and then click on **Proceed**.
15. In the **Multi Slave Config Plugin – Add slaves** screen, add the following details:

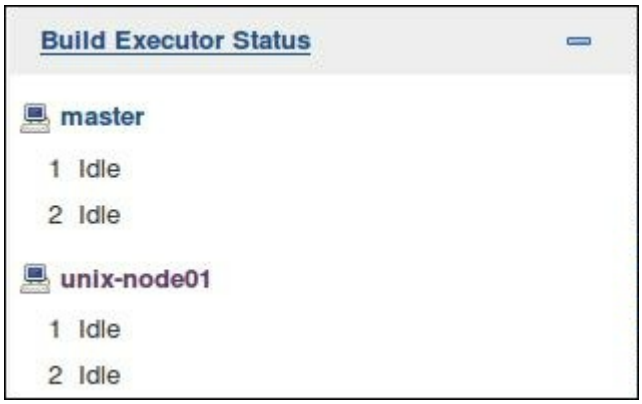
    - **Description**: **I am a dumb Ubuntu node**
    - **# of executors**: **2**
    - **Remote FS root**: **/home/jenkins-unix-nodex**

- **Set labels**: **unix dumb functional**

16. Select for launch method **Launch slave agents on Unix machines via SSH** and add the details:

    - **Host**: **localhost**
    - **Credentials**: **jenkins-unix-nodex**

17. Click on **Save**.
18. Return to the main page. You will now see that the **Build Executor Status** includes the **Master** and **unix-node01**, as shown in the following screenshot:



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 deploy 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 shallow 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, not just for *NIX. One alternative is to install **Cygwin** (http://www.cygwin.com/) with an SSH daemon on Windows.

> **Note**
>
> 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.

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 the deployment of virtual instances.

> **Note**
>
> You can find more information about installing a Windows service for a Jenkins slave at https://wiki.jenkins-ci.org/display/JENKINS/Step+by+step+guide+to+set+up+master+and+slave+machines.

There's more...

Since Version 1.446 (http://jenkins-ci.org/changelog), Jenkins has an in-built SSH daemon. This will decrease the amount of effort writing 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, use curl (http://curl.haxx.se/) to find out what headers are returned from Jenkins. For example, for *NIX systems from the command line try:

```
curl -s -D - localhost:8080 -o /dev/null
```

The header information is sent to `stdout` for you to view and the body is sent to `/dev/null` which is the system location that ignores all input.

The response from Jenkins will be similar to the following:

```
HTTP/1.1 200 OK
Cache-Control: no-cache,no-store,must-revalidate
X-Hudson-Theme: default
Content-Type: text/html;charset=UTF-8
Set-
Cookie:
JSESSIONID.bebd81dc=1mkx0f5m97ipsjqhygljrbqmo;Path=/;HttpOnly
Expires: Thu, 01 Jan 1970 00:00:00 GMT
X-Hudson: 1.395
X-Jenkins: 1.583
X-Jenkins-Session: 5c9958f6
X-Hudson-CLI-Port: 39269
X-Jenkins-CLI-Port: 39269
X-Jenkins-CLI2-Port: 39269
X-Frame-Options: sameorigin
X-SSH-Endpoint: localhost:57024
X-Instance-
Identity:
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAjOABhI
+cuNtKfu5b46FKGr/IXh9IgaTVgf16QgCmoAR41S00gXJezDRJ1i4tC0tB6Tqz5SuKqcDDxU1
+KzADEhTB/voiLLoEkogj22WtUd7TZWhzRnAW58wrzI6uAWHqOtHvlO7MxFo1AY4ZyXLw202D
+1tlKkECr5oy9dFyKy3U1lnpilg6snG70AYz
+/uFs52FeOl3qkCfDVCGMDHqLEvJJzWsZ5hAv37fEaj1QyMA69joBjesgt1n1CeJeD0cy5+BI
Length: 19857
Server: Jetty(8.y.z-SNAPSHOT)
```

See also

- The *Using a test Jenkins instance* recipe in Chapter 1, *Maintaining Jenkins*
- The *Custom setup scripts for slave nodes* recipe

Custom setup scripts for slave nodes

This recipe shows you how to run your own initialization scripts on slave nodes. This allows you to perform node system cleanups, check health, set up tests, and a multitude of other necessary tasks.

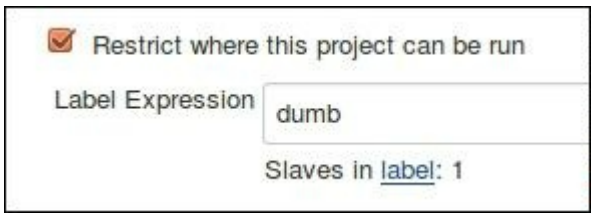
Getting ready

For this recipe to work, you will need to have installed a slave node as described by the *Creating multiple Jenkins nodes* recipe.

You will also have installed the Slave Setup plugin (https://wiki.jenkins-ci.org/display/JENKINS/Slave+Setup+Plugin).

How to do it...

1. Create a free-style job named `ch6.remote.slave.setup`.
2. Check **Restrict** where this project can run.
3. Add the text `dumb` to **Label Expression**, as shown in the following screenshot:



4. Click on **Save** and then build the job.
5. Click on **Back to Dashboard**.
6. Under **Build Executor Status** click on the **unix-node01**.
7. Click on **Script Console**.
8. Add the following text to the **Script Console** and then click on **Run**:

```
println "pwd".execute().text
println "ls ./workspace".execute().text
```

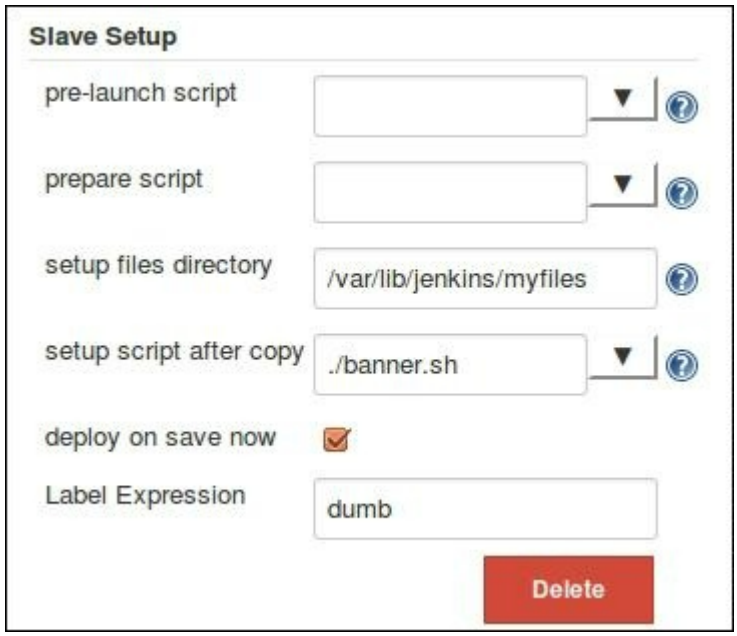
9. Create the directory `/var/lib/jenkins/myfiles`.
10. Create the file `/var/lib/jenkins/myfiles/banner.sh` with the following text:

```
#!/bin/sh
echo ------------------------- >
slave_banner.txt
echo THIS IS A SLAVE INIT BANNER  >>
slave_banner.txt
```

```
echo WORKING ON SLAVE: ${NODE_TO_SETUP_NAME}
>> slave_banner.txt
date >> slave_banner.txt
echo SCRIPT DOES SOME WORK HERE >>
slave_banner.txt
echo ------------------------- >>
slave_banner.txt
mv slave_banner.txt  /home/jenkins-unix-
nodex/workspace/ch6.remote.slave.setup/
```

11. Visit the **Configure system** page (`http://localhost:8080/configure`).
12. Under the **Slave Setup** section, click on the **Add** button for the **Slave Setups list**.
13. Add the following details:

    - **setup files directory**: `/var/lib/jenkins/myfiles`
    - **setup script after copy**: `./banner.sh`
    - **Label Expression**: `dumb`



14. Check **deploy on save now**.
15. Click on **Save**.
16. Run the job `ch6.remote.slave.setup`.
17. Review the workspace. You will now see a `banner.txt` file with content similar to the following:

```
-------------------------
THIS IS A SLAVE INIT BANNER
WORKING ON SLAVE: unix-node01
Tue Oct 14 13:39:09 CEST 2014
SCRIPT DOES SOME WORK HERE
```

```
------------------------
```

How it works...

You used the Slave Setup plugin to copy `banner.sh` from `/var/lib/jenkins` to the home directory of the slave, `banner.sh`. This action runs before each job run on the node.

Checking **deploy on save now** makes sure that the script on the slave is fresh.

You used the script console to discover the home location of the node. You also verified that the workspace contained a directory for the `ch6.remote.slave.setup` job with the same name as the job.

In the job, you restricted where it ran to the nodes with the **dumb** label. This way you are certain that the job runs on the node.

`banner.sh` uses the `sh` shell by default that is really pointing to `bash` the **Bourne Again Shell** (http://www.gnu.org/software/bash/), or `dash` the **Debian Almquist Shell** (http://gondor.apana.org.au/~herbert/dash/).

> **Note**
>
> For the reasoning behind the use of dash in Ubuntu visit https://wiki.ubuntu.com/DashAsBinSh.

To show that it has run, the script outputs to `banner.txt`, a small banner with a time stamp. The last command in the script moves `banner.txt` to the jobs directory under the nodes workspace.

Once the job has run, the slave copies the workspace back to the Jenkins master's workspace. Later you viewed the results.

There's more...

If your Jenkins node supports other languages, such as Perl, you can run them by adding the `#!` convention as the first line of your script, pointing to the full path of the binary of the scripting language. To discover the path to the binary, you can use the nodes script console and run the `which` command:

```
println "which perl".execute().text
```

This results in `/usr/bin/perl`.

A "Hello World" Perl script would then look like the following code:

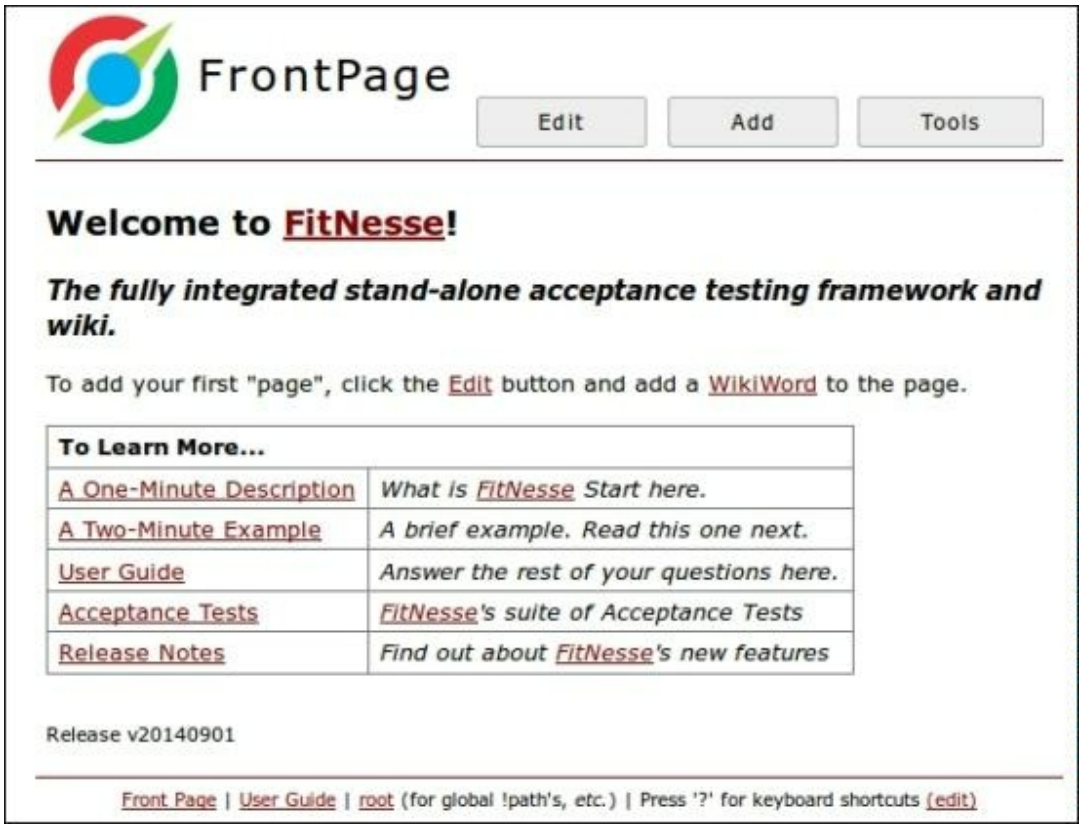
```
#!/usr/bin/perl
print "Hello World"
```

See also

- The *Creating multiple Jenkins nodes* recipe

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 required to 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 pointing the tests at those mock-ups.

This recipe describes how to run FitNesse remotely and display the results within Jenkins.

Getting ready

Download the latest stable FitNesse JAR from http://fitnesse.org/FitNesseDownload. Install the FitNesse plugins for Jenkins from https://wiki.jenkins-ci.org/display/JENKINS/FitNesse+Plugin.

Note

The release number used to test this recipe was `20140901`.

How to do it...

1. Create the directories `fit/logs` and place them in the fit directory `fitnesse-standalone.jar`.
2. Run the FitNesse help from the command line and review the options:

```
java -jar fitnesse-standalone.jar -help
Usage: java -jar fitnesse.jar [-vpdrleoab]
-p <port number> {80}
-d <working directory> {.}
-r <page root directory> {FitNesseRoot}
-l <log directory> {no logging}
-
f <config properties file>
{plugins.properties}
-
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.
-b <filename> redirect command output.
-v {off} Verbose logging
```

3. Run FitNesse from the command line and review the startup output:

```
java -jar fitnesse-standalone.jar -p 39996 -l
logs -a tester:test
Bootstrapping FitNesse, the fully integrated
standalone wiki and acceptance testing
framework.
root page: fitnesse.wiki.fs.FileSystemPage
at ./FitNesseRoot#latest
logger: /home/alan/Desktop/X/fitness/logs
authenticator:
fitnesse.authentication.OneUserAuthenticator
page factory:
fitnesse.html.template.PageFactory
page theme: fitnesse_straight
Starting FitNesse on port: 39996
```

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 password `test`.
8. Review the log in the `fit/logs` directory. After running the suite again, you will now see an entry similar to:

```
127.0.0.1 - tester [01/Oct/2014:11:14:59
+0100]
"GET /FitNesse.SuiteAcceptanceTests?
suite HTTP/1.1" 200 6086667
```

9. Log in to Jenkins and create a free-style software project named `ch6.remote.fitnesse`.
10. In the **build** section, select the **Execute fitnesse tests** option from **Add Build step**.
11. Check the option **FitNesse instance is already running**, adding:
    - **Fitnesses Host**: `localhost`
    - **Fitnesses Port**: `39996`
    - **Target Page**: `FitNesse.SuiteAcceptanceTests?suite`
    - Check the **Is target a suite?** option
    - **HTTP Timeout (ms)**: `180000`
    - **Path to fitnesse xml results file**: `fitnesse-results.xml`
12. In the **Post-build Actions** section, check the **Publish FitNesse results** report option.
13. Add the value `fitnesse-results.xml` to the input **Path to fitnesse xml results file**.
14. Click on **Save**.
15. Run the job.
16. Review the latest job by clicking on the link **FitNesse Results**.



How it works...

FitNesse has a built-in set of acceptance tests that it uses to check itself for regressions. The Jenkins plugin calls the test and asks for the results to be returned in XML format using an HTTP `GET` request with the URL: `http://localhost:39996/FitNesse.SuiteAcceptanceTests?suite&format=xml`. The results look similar to the following:

```
<testResults>
<FitNesseVersion>v20140901</FitNesseVersion>
<rootPath>FitNesse.SuiteAcceptanceTests</rootPath>
<result>
<counts><right>103</right>
<wrong>0</wrong>
<ignores>0</ignores>
<exceptions>0</exceptions>
</counts>
<runTimeInMillis>27</runTimeInMillis>
<relativePageName>CopyAndAppendLastRow</relativePageName>
<pageHistoryLink>
FitNesse.SuiteAcceptanceTests.SuiteFitDecoratorTests.CopyAndAppendLastRow
pageHistory&resultDate=20141101164526
</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 start-up. However, we did not take this further and define 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/ActiveDirectory. For more details review http://fitnesse.org/FitNesse.FullReferenceGuide.UserGuide.AdministeringFitNesse.SecurityDe

There is also a contributed plugin for LDAP authentication: https://github.com/timander/fitnesse-ldap-authenticator

Note

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...

You will find the source code with information on building the newest version of FitNesse at its GitHub home: https://github.com/unclebob/fitnesse

If you like FitNesse, why not involve yourself in the community discussions? You can subscribe to its Yahoo group at <fitnesse-subscribe@yahoogroups.com> and then post messages at <fitnesse@yahoogroups.com.> Yahoo's usage guidelines discusses the general etiquette: http://info.yahoo.com/guidelines/us/yahoo/groups/

See also

- The *Activating FitNesse HtmlUnit fixtures* recipe

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 from 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 the *Testing with FitNesse* recipe.

How to do it...

1. Visit http://sourceforge.net/projects/htmlfixtureim/ 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-standalone.jar -p 39996 -l
logs -a tester:test
```

5. In a web browser, visit `http://localhost:39996/root?edit`, adding the following content, replacing `FitHome` with the fully qualified path to the home of your Fitnesse server:

```
!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 **Tools** button.
11. Select **Properties**.
12. Click on **Page Type** test.
13. A pop-up appears asking for your **username** and **password**. Type `tester` and `test`.
14. You will be returned to the **ThisisMyPageTest** page; click on **Save**.

15. Click on the **Edit** button on the left-hand side menu.
16. 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/||
```

17. Click on **Save**.
18. Click on **Test**.



19. In Jenkins under **New Job**, copy **existing job** / **copy** from `ch6.remote.fitness` to **Job name** `ch6.remote.fitness_fixture`.
20. In the **build** section, under **Target** | **Target Page** replace `FitNesse.SuiteAcceptanceTests` with `ThisIsMyPageTest`.
21. Uncheck **Is target a suite?**.
22. Click on **Save**.
23. Run the job. It fails because the extra debugging information sent with the results confuses the Jenkins plugin parser.
24. Visit the test page `http://localhost:39996/ThisIsMyPageTest?edit`, replacing the contents of the test table with the following code:

```
!|HtmlFixture|
|http://localhost:8080/login| Login|
|Has Text|log in|
|Element Focus|search-box|input|
|Set Value|ch5|
|Focus Parent Type|form|/search/|
```

25. 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 start-up. For more details, review the file `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 are returned as results. For example:

```
|Print something||
```

The `Has Text` command is an assertion and will fail if `log in` 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; for example, the first row displays the returned page and the second 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:

- **RestFixture for REST services**:

  https://github.com/smartrics/RestFixture/wiki

- **Webtestfixtures using Selenium for web-based functional testing**:

  http://sourceforge.net/projects/webtestfixtures/

- **DBfit that allows you to test databases**:

  http://gojko.net/fitnesse/dbfit/

See also

- The *Testing with FitNesse* recipe

Running Selenium IDE tests

Selenium IDE allows you to record your clicks within web pages in Firefox and replay them. This is good for functional testing. The test plans are saved in 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 other browser types as well.

With the release of Selenium 2.0, the Selenium server now has built-in grid functionality (https://code.google.com/p/selenium/wiki/Grid2). It is beyond the scope of this chapter to discuss this other than to note that Selenium grid allows you to run Selenium tests in parallel across a number of OS.

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 an Ubuntu Linux environment run `sudo apt-get install xvfb`.

> **Note**
>
> In the Jenkins plugin manager, the plugin is called the Environment Injector plugin, whereas in the wiki it is called the EnvInject plugin. It can be confusing, but both names belong to the same plugin.

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>2.3</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 `src/test/resources/log4j.properties` file with the following content:

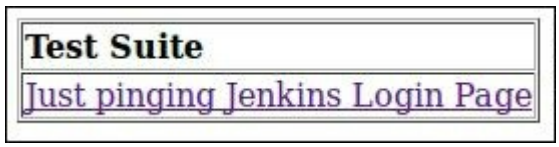
```
log4j.rootLogger=INFO, A1
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-
4r [%t] %-5p %c %x - %m%n
```

4. Make the directory `src/test/resources/selenium`.
5. Create the file `src/test/resources/selenium/TestSuite.xhtml` with the 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 into the following screenshot:



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>
```

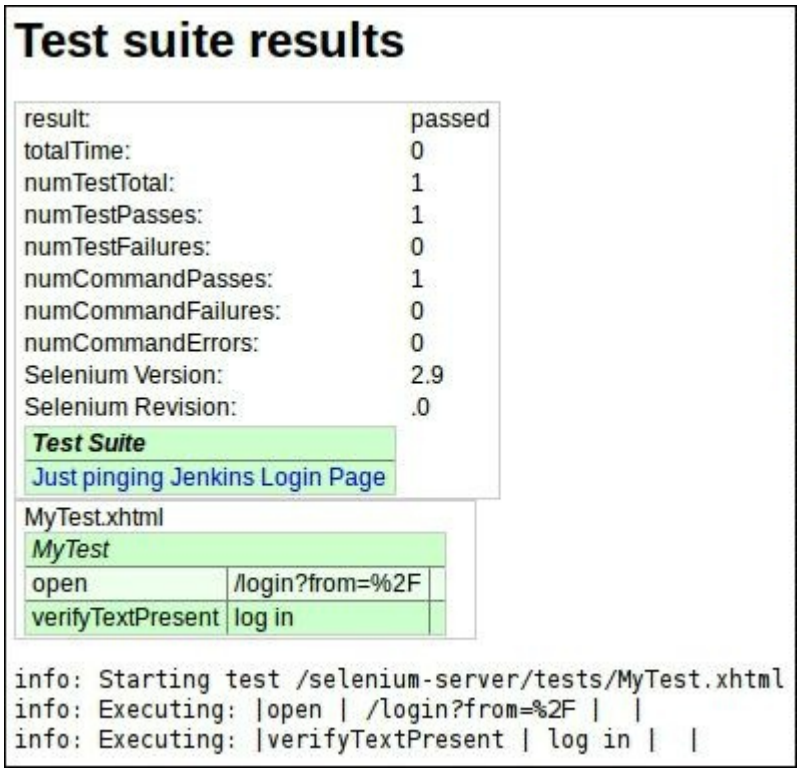
The HTML will render as per the following screenshot:

MyTest	
open	/login?from=%2F
verifyTextPresent	log in

7. Run the Maven project from the command line, verifying that the build succeeds, as shown:

```
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 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** and add your subversion URL to **Repository URL**.
12. In the **build** section, add `clean integration-test -Dlog4j.configuration=file./src/test/resources/log4j.properties` to **Goals and options**.
13. In the **Post-build Actions** section, check **Publish Selenium HTML Report**.
14. Add `target/results` to **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, reviewing the results, as shown in the following screenshot:



How it works...

A primitive Selenium IDE test suite was created comprising two HTML pages. The first `TestSuite.xhtml` defines the suite as 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 **log in** text is present.

The `pom.xml` file 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 a 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 in automating 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 and generates 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 uses native API calls that function more reliably. Like 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 a known working binary for Firefox under the Jenkins home directory and point directly at the binary in the `pom.xml` file, replacing `<browser>*firefox</browser>` with `<browser>*firefox Path</browser>`.

Here, 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://docs.seleniumhq.org/docs/

> **Note**
>
> An alternative to Firefox 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

- The *Triggering failsafe integration tests with Selenium WebDriver* recipe

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 automated functional tests. Using HtmlUnit instead of a real browser makes for stable OS-agnostic tests that, although they do not test browser compatibility, can spot the majority of functional failures.

Getting ready

Create a project directory. Review the Maven Compiler plugin documentation (http://maven.apache.org/plugins/maven-compiler-plugin/).

How to do it...

1. Create `pom.xml` with the 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/java/nl/berg/packt/webdriver` by adding the `TestIT.java` file with the following content:

```
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 output similar to:

```
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 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** that details the functional tests.

How it works...

Maven uses the Failsafe plugin (http://maven.apache.org/surefire/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.

The `pom.xml` file 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 style sheet 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 out the assertions from the navigation of web pages. Consider creating Java classes according to the web page (https://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.

Tip

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();
}
```

Note

Unfortunately, the HtmlUnit driver does not create screenshots: http://code.google.com/p/selenium/issues/detail?id=1361.

However, you can find an experimental update at https://groups.google.com/forum/#!msg/selenium-developers/PTR_j4xLVRM/k2yVq01Fa7oJ.

See also

- The *Running Selenium IDE tests* recipe
- The *Activating FitNesse HtmlUnit fixtures* recipe

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 then hammer systems based on that plan.

JMeter can make many types of requests known as **samplers**. It can sample HTTP, LDAP, and databases, use scripts, and much more. It can report back visually with **listeners**.

> **Note**
>
> A beginner's book on JMeter is: Apache JMeter by Emily H. Halili published by Packt Publishing, ISBN 1847192955 (http://www.packtpub.com/beginning-apache-jmeter).
>
> Two more advanced books from the same publisher are https://www.packtpub.com/application-development/performance-testing-jmeter-29 and https://www.packtpub.com/application-development/jmeter-cookbook-raw.

In this recipe, you will write a test plan for hitting web pages whose URLs are defined in a text file. 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 and so will run on any system that has Java correctly installed.

How to do it...

1. Create the subdirectories `plans` and `example`.
2. Create a CSV file `./data/URLS.csv` with the following content:

```
localhost,8080,/login
localhost,9080,/blah
```

3. Run the JMeter GUI—for example, `./bin/jmeter.sh` or `jmeter.bat`, depending on 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 the 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 then **File** | **Save**. Save the test plan to `example/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. In reality, a real user hits the system a lot less hard than a thread. Threads can hit the system many times a second, whereas typically a user clicks approximately once every twenty seconds.

The test plan uses a number of elements:

- **Thread Group**:This defines the number of threads that run.
- **Cookie manager**: This keeps track of cookies per thread. This is important if you want to keep track through cookies between requests.

For example, if a thread logs in to a Tomcat server the unique `Jsessionid` needs to be stored for each thread.

- **CSV Data Set Config**: This 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.
- **View Results Tree**: This 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.

Tip

If you are looking to coax out multithreading 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 text file 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 agent. JMeter can then loop through the user agents, asserting that the resources exist.

See also

- The *Reporting JMeter performance metrics* recipe
- The *Functional testing using JMeter assertions* recipe

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 the JMeter test plan. 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 as `./plans/URL_ping.jmx`.
2. Select **CSV Data Set Config** changing **Filename** to `${__property(csv)}`.
3. Under the **File** menu select the **Save** option.
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**, adding your subversion repository URL to **Repository URL**.
9. Within the **build** section, add the build step **Invoke Ant**.
10. Click on **Advanced** in the new **Invoke Ant** subsection, adding 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 the input `jmeter_results/*.jtl` to **Report Files**.
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; the location of the CSV file `${basedir}/data/URLS.csv`. `${basedir}`is 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 `${__functioncall(parameters)}`. You have added the function call `${__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

- The *Creating JMeter test plans* recipe
- The *Functional testing using JMeter assertions* recipe

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 a mockup of a web from an HTML 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

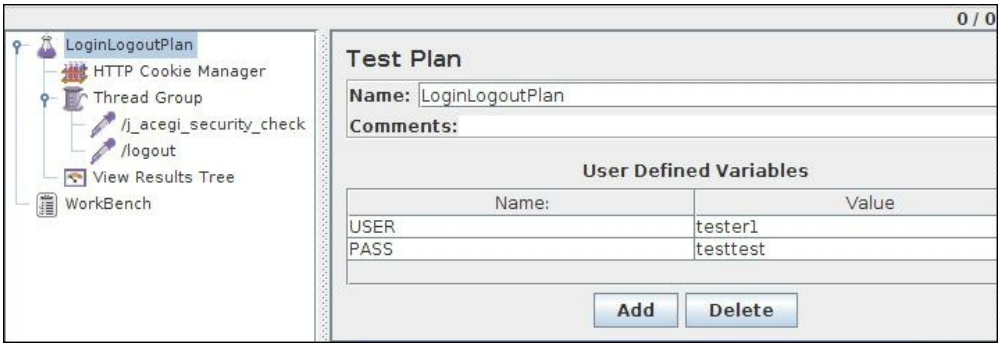
We assume that you have already performed the *Creating JMeter test plans* and *Reporting JMeter performance metrics* recipes.

> **Note**
>
> 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.

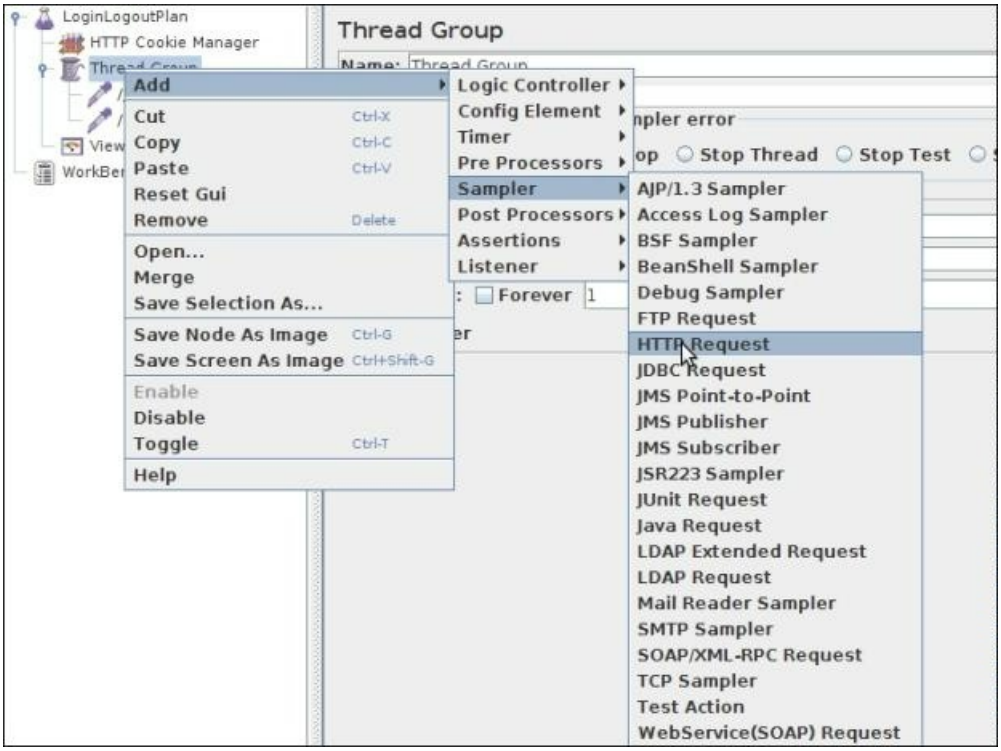
How to do it...

1. Create a user in Jenkins named `tester1` with password `testtest`.
2. Run JMeter. In the **Test Plan** element change the **Name** to `LoginLogoutPlan`, adding for **User defined Variables**:

    - **Name**: `USER` and **Value**: `tester1`
    - **Name**: `PASS` and **Value**: `testtest`



3. Right-click on **Test Plan** then select **Add** | **Config Element** | **HTTP cookie Manager**.
4. Right-click on **Test Plan** and then select **Add** | **Listener** | **View Tree Results**.
5. Right-click on **Test Plan** and then select **Add** | **Threads (Users)** | **Thread Group**.
6. Right-click on **Thread Group** and then select **Add** | **Sampler** | **HTTP Request**, as shown in the following screenshot:



7. Add the following details to the **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:

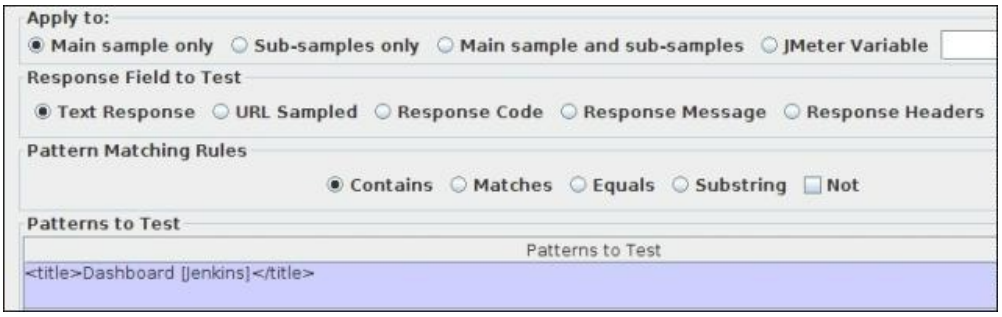
    - **Name**: `j_username` and **Value**: `${USER}`
    - **Name**: `j_password` and **Value**: `${PASS}`

9. Right-click on **Thread Group** and then select **Add** | **Sampler** | **HTTP Request**.
10. Add the following details to the **HTTP Request Sampler**. If necessary drag-and-drop the newly created element so that it is placed after `/j_acegi_security_check`.
11. Add the following details to the **HTTP Request Sampler**:

- **Name**: `/logout`
- **Server Name or IP**: `localhost`
- **Port Number**: `8080`
- **Path**: `/logout`

12. Save the test plan 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` and checking **Type of Comparison** to **<**.
21. Right-click on the JMeter element `j_acegi_security_check`, selecting **Add** | **Assertion** | **Response Assertion** with the 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 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 the `/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 that 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 in 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 wider a safety margin you will have to build in. It is not uncommon to plan for 100 percent extra capacity. The extra capacity may well be the difference between you going on holiday or not.

Note

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

- The *Creating JMeter test plans* recipe
- The *Reporting JMeter performance metrics* recipe

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://sakaiproject.org.

Download and unpack Sakai CLE version 2.8.1 from http://source.sakaiproject.org/release/2.8.1.

The newest release can be found at http://source.sakaiproject.org/release.

Please note that the server will take more time on its first startup than later startups. This is due to the initial creation of sample courses.

How to do it...

1. Edit `sakai/sakai.properties` to include:

```
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:

```
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 `./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.

> **Note**
>
> The first startup may take a long time as demonstration data is populated into a built in database.

6. In a web browser visit `http://localhost:39955/portal`.
7. Log in in as user `admin` with password `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 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 XML format visit `http://localhost:39955/direct/user/current.xml` and to view 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 dropped 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, it is also a framework that makes developing new tools straightforward.

The programmer's cafe 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 cafe 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.

You can find the book *Sakai CLE Courseware Management: The Official Guide*, *Packt Publishing*, at http://www.packtpub.com/sakai-cle-courseware-management-for-elearning-research/book.

> **Note**
>
> Another related product is the Apereo **Open Academic Environment** (**OAE**) http://www.oaeproject.org/.
>
> Apereo OAE, like Sakai, is community-sourced. It has unique capabilities such as the ability to run in multiple organizations at the same time, looking different for each, and being able to search documents between organizations or not, depending on how you configure groups.

See also

- The *Writing test plans with SoapUI* recipe
- The *Reporting SoapUI test results* recipe

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/installing-soapui/installing-on-windows.html and follow the installation instructions.

For the Linux package to work with older versions of SoapUI, you may have to uncomment the following line in the SoapUI startup script:

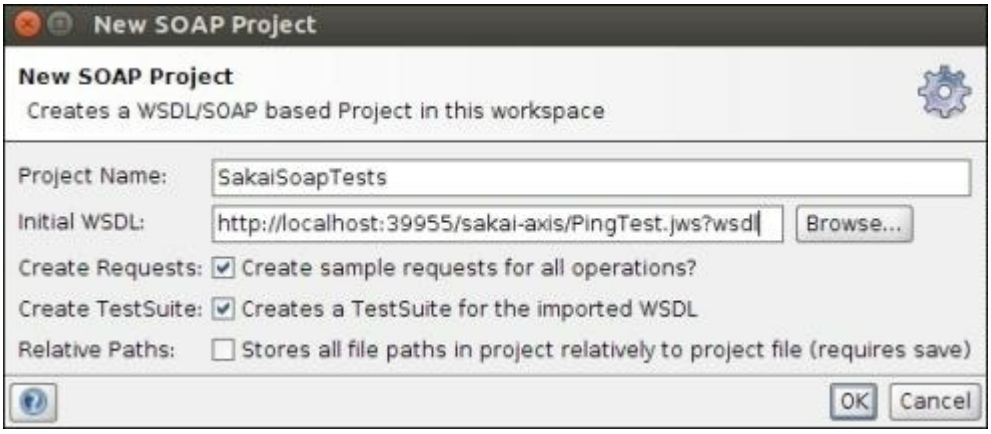
```
JAVA_OPTS="$JAVA_OPTS
-Dsoapui.jxbrowser.disable=true"
```

> **Note**
>
> This recipe was tested against version 5.0.0 of SoapUI.

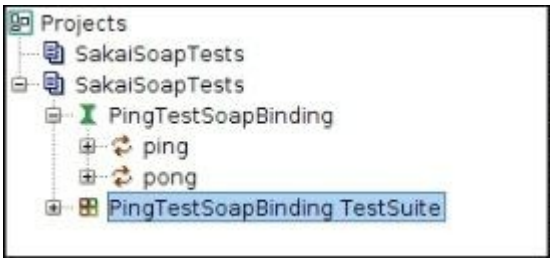
How to do it...

1. Start SoapUI.
2. Right-click on **Projects** and select **New Soap Project**.
3. Fill in the dialog box with the following details:

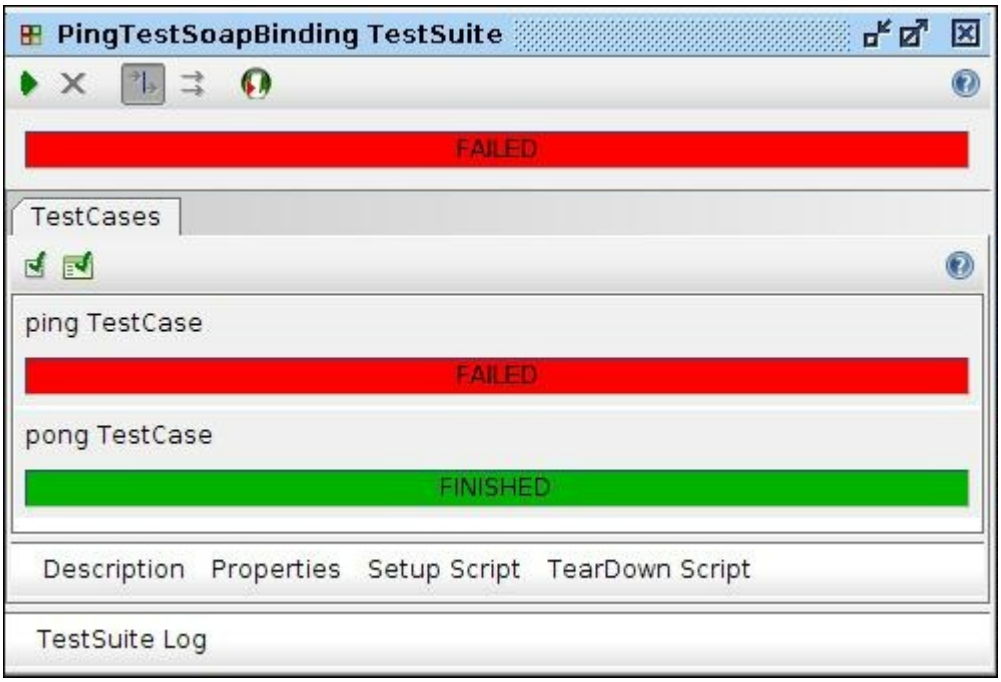
    - **Project Name**: `SakaiSoapTests`
    - **Initial WSDL/WADL**: `http://localhost:39955/sakai-axis/PingTest.jws?wsdl`



4. Check **Create TestSuite**.
5. Click on **OK**.
6. Click on **OK** for the **Generate TestSuite** dialog box.
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 then select **Open Editor**.
12. At the top of the editor, click on the **Add Assertion** icon .
13. Select **Assertion Contains** and click on **OK**.
14. For **Content** select `NOT IN TEXT` and click on **OK**.
15. In the left-hand side navigation, right-click on **PingTestSoapBinding TestSuite** and select **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** and then **Save Project as** `SakaiSoapTests-soapui-project.xml` in directory `src/test/soapui`.



How it works...

SoapUI takes the drudgery out of making test suites for Soap services. SoapUI used the `PingTest` WSDL file to discover the details of the service.

WSDL stands for **Web Services Description Language** (http://www.w3.org/TR/wsdl). An XML file with information on the location and use of the `PingTest` service is generated.

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 `NOT IN TEXT` exists 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, 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.

There's more...

SoapUI does a lot more than functional tests for web services. It performs security tests by checking 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

- The *Enabling Sakai web services* recipe
- The *Reporting SoapUI test results* recipe

Reporting SoapUI test results

In this recipe, you will be creating a Maven project that runs the SoapUI test created in the previous recipe. A Jenkins project using the xUnit plugin (https://wiki.jenkins-ci.org/display/JENKINS/xUnit+Plugin) will then parse the results and generate 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.

To experiment with the newest version of the Maven plugin visit http://www.soapui.org/Test-Automation/maven-2x.html

How to do it...

1. Create a project directory. At the root of the project, add a `pom.xml` file with the following content:

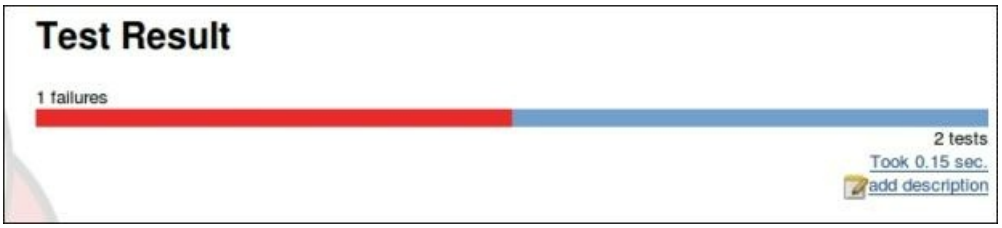
```
<project
xmlns="http://maven.apache.org/POM/4.0.0"

 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 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 successful job, as shown in the following screenshot:



15. You will find the full 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 the `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 to `on` unless you have severe disc space constraints. `printReport` set to `true` ensures Maven sends a small test report to the console with output similar to:

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 auto generation 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`.

Tip

The custom reports option is yet another way of pulling in your own custom data and displaying its historic trends within Jenkins. It works by parsing the XML results found by the plugin with a custom style sheet. This gives you a great deal of flexibility in adding your own custom results.

There's more...

The ping service is dangerous as it does not filter input, and the input is reflected back through the output.

Many web applications use web services to load content into a page, to avoid 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

- The *Enabling Sakai web services* recipe
- The *Writing test plans with SoapUI* recipe

Chapter 7. Exploring Plugins

In this chapter, we will cover the following recipes:

- Personalizing Jenkins
- Testing and then promoting builds
- Having fun with pinning JSGames
- Looking at the GUI samples plugin
- Changing the help of the FileSystem SCM plugin
- Adding a banner to job descriptions
- Creating a RootAction plugin
- Exporting data
- Triggering events on startup
- Using Groovy hook scripts and 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, 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 1,000. It is likely that there are plugins already available that meet your needs. Jenkins is not only a Continuous Integration server, it is also a platform to create extra functionality. Once a few concepts are learned, a programmer can adapt available plugins to an organization's needs.

If you see a feature that is missing, it is normally easier to adapt an existing one than to write one 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 has 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 already, 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 wider audience. Under

the careful watch of the community, the code is more likely to be reviewed and further 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 to 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 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 improved.
- The three frameworks that are regularly used in Jenkins are:
    - **Jelly** for the creation of the GUI
    - **Stapler** to bind of the Java classes to the URL space
    - **XStream** for the persistence of configurations into XML

> ## Note
>
> Source code for a number of plugins mentioned in this chapter is in subversion. After the time of writing, if any of the code is moved, it is likely to have moved to the Jenkins GitHub repository (https://github.com/jenkinsci). The convention for plugin developers is to leave a `README` file in the old subversion repository listing the new Git location.

Personalizing Jenkins

This recipe highlights two plugins that improve the user experience: the Green Balls plugin and the Favorite plugin.

Jenkins has a wide international audience. At times, there can be subtle cultural differences expressed in the way Jenkins looks. One example is that when a build succeeds, a blue ball is shown as the icon. However, many naturally associate the green from traffic lights with the signal to carry on.

The Favorite plugin allows you to select your favorite projects and display an icon in a view to highlight your picks.

Getting ready

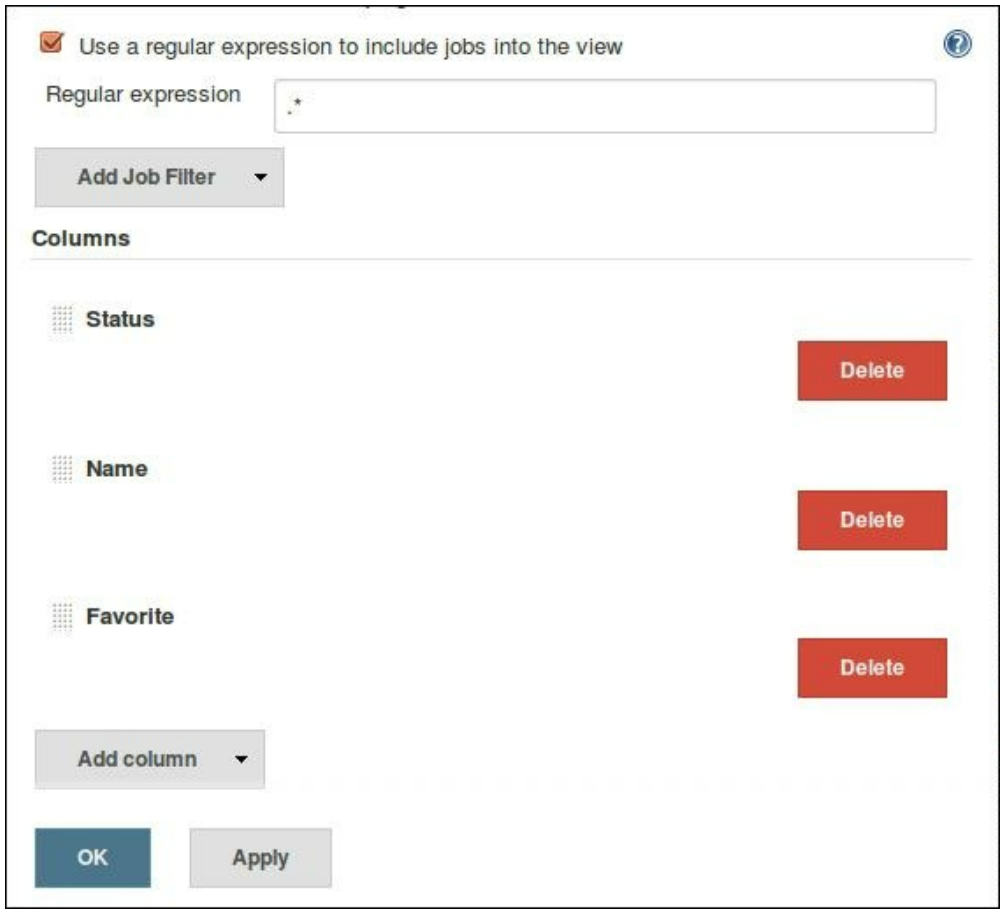
Install the Green Balls and Favorite plugins (https://wiki.jenkins-ci.org/display/JENKINS/Green+Balls, https://wiki.jenkins-ci.org/display/JENKINS/Favorite+Plugin).

How to do it...

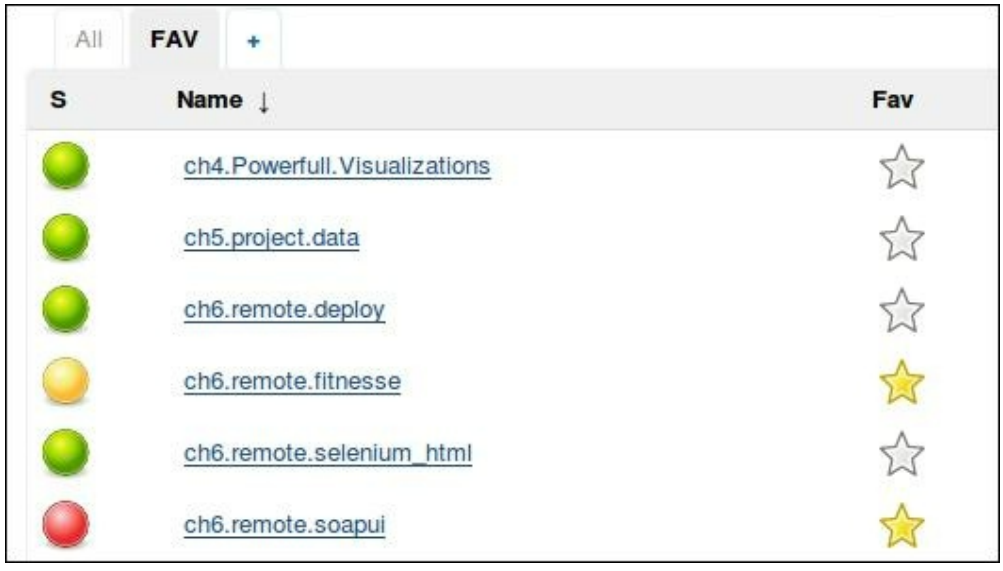
1. Create an empty new free-style job `named ch7.plugin.favourite`.
2. Build the job a number of times, reviewing the build history. You will now see green balls instead of the usual blue, as shown in the following screenshot:



3. Return to the main page.
4. To create a new view, click on the **+** icon.
5. Fill in **FAV** for the name.
6. Check **List View**.
7. Under the **Job Filters** section, check **Use a regular expression to include jobs into the view**. Add `.*` for the **Regular expression**.
8. In the **Columns** section, make sure you have three columns **Name**, **Status**, and **Favorite**, as shown in the following screenshot:



9. Click on **OK**.
10. You will find yourself in the **FAV** view. By clicking on the star icon, you can select/deselect your favorite projects, as shown in the following screenshot:



How it works...

The Green Balls plugin works as advertised. The Favorite plugin allows you to select which project interests you the most and display that as a favorites icon. This reminds you that the project needs some immediate action.

> **Note**
>
> If you are interested in working with the community, then these plugins are examples that you could add extra features to. You can find the official guide to contributing here: https://wiki.jenkins-ci.org/display/JENKINS/Beginners+Guide+to+Contributing

There's more...

The opposite of a favorite project, at least temporarily, is a project whose build has failed. The Claim 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 responsibility.

Once the Claims plugin is installed, you will be able to find in the **Post-Build Actions** section of a job a tick box for **Allow broken build claiming**. Once enabled, if a build fails you can claim a specific build, adding a note about your motivation.



On 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 that are dealing with current issues, as shown in the following screenshot:



The Favorite plugin is elegant in its simplicity. In the next recipe, testing and then promoting will signal that further incorporation of complex workflows is allowed.

See also

- The *Testing and then promoting builds* recipe
- The *Fun with pinning JSGames* recipe

Testing and then promoting builds

You do not want the QA team to review an 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 promotion and the Favorite plugin mentioned in the preceding recipe is that promotion can be triggered automatically, based on a variety of automated actions. These 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; 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:

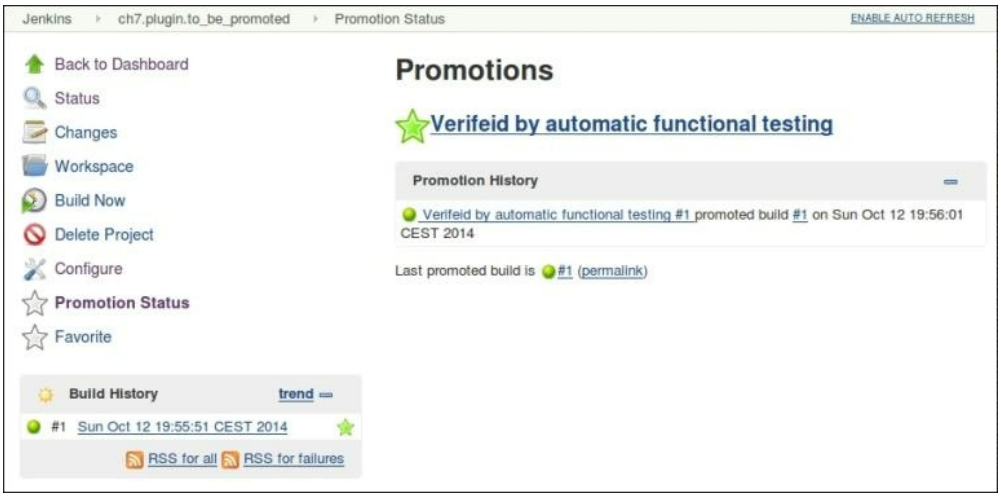
    - **Name**: Verified by automatic functional testing
    - Select **Green star** for the **Icon**
    - Check **When the following downstream projects build successfully**
    - **Job names**: `ch7.plugin.promote_action`



6. In the **Post-build Action** section check **Build other projects**.
7. Fill in for **projects to build** `ch7.plugin.promote_action`.
8. Tick **Trigger only if build is stable**.
9. Click on **Save**.
10. Build the job.
11. Click on the **Promotion Status** link, as shown in the following screenshot:



12. Review the build report.



How it works...

Promoted Builds is similar to the Favorite plugin, but with automation of workflow. You can promote depending on job(s) triggered by the creation of artifacts. This is typical workflow when you want a job tested for baseline quality before being picked up and reviewed.

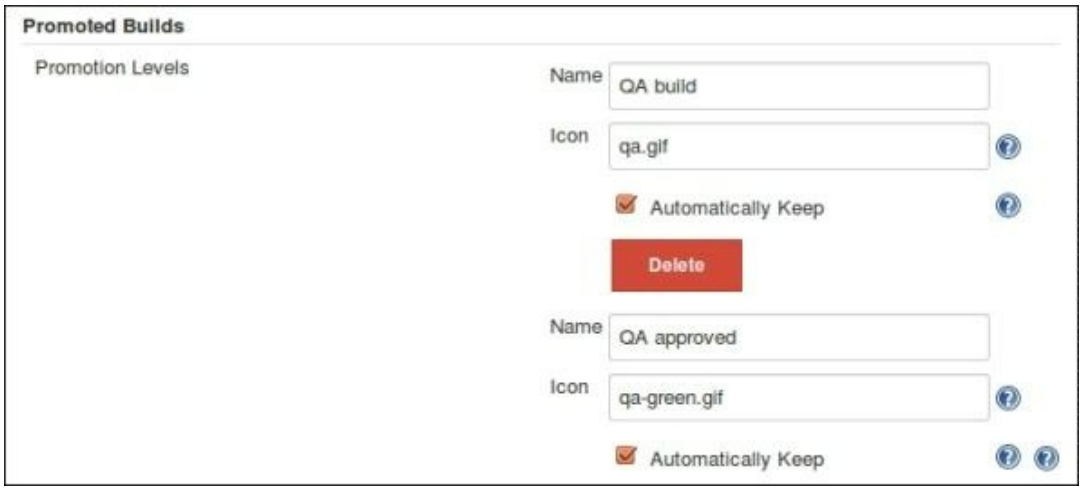
The plugin has enough configuration options to make it malleable to most workflows. Another example, for a typical development, acceptance, and production infrastructure, is that you do not want an artifact to be deployed to production before development and acceptance have also been promoted. The way 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.

> **Note**
>
> If you want to add human intervention, then check **Only when manually approved** in the 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 makes it easier to explain, allowing use by a wider audience.

You can configure the different types of promotion within the main Jenkins configuration page, as shown in the following screenshot:



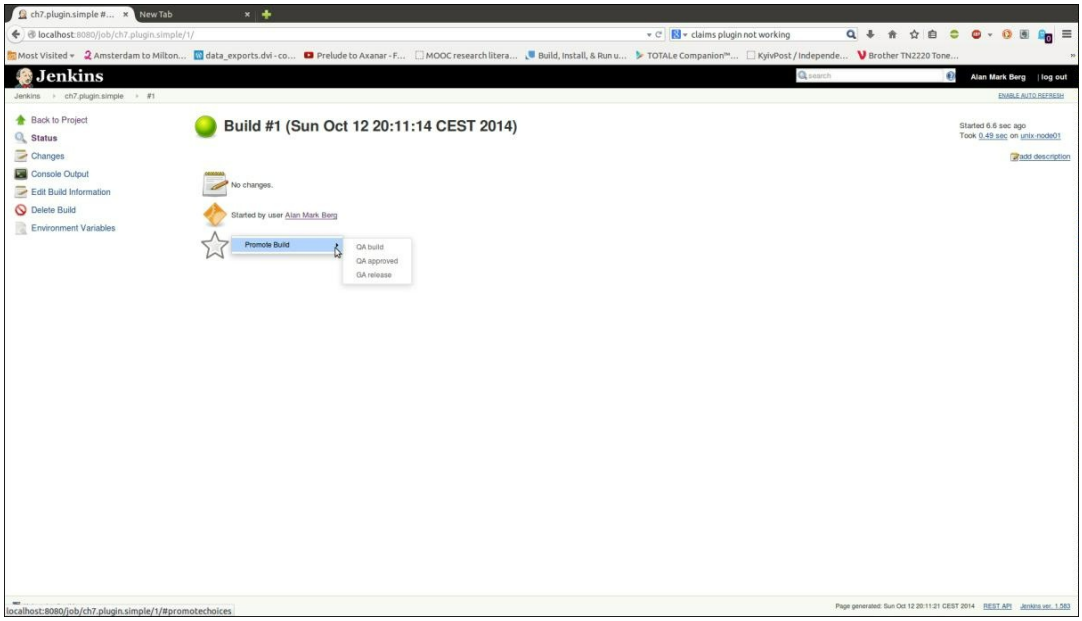
Note

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.

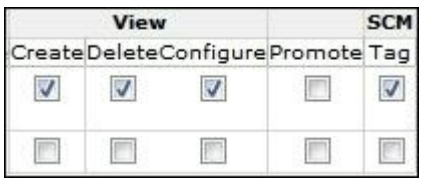
The plugin allows you to elevate promotions. There is a simple choice available through a link on the left-hand side of the build. This feature allows you to add a series of players into the promotion process.

Note

When the final promotion occurs, for example to **GA** (**Generally Available**), the promotion is locked and can no longer be demoted.



The ability of a user to promote depends on their permissions. 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, as shown in the following screenshot:



See also

- The *Personalizing Jenkins* recipe

Fun with pinning JSGames

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 reduce pressure, consider allowing your team access to relaxation with the JSGames plugin.

Getting ready

Install the JSGames plugin (https://wiki.jenkins-ci.org/display/JENKINS/JSGames+Plugin).

How to do it...

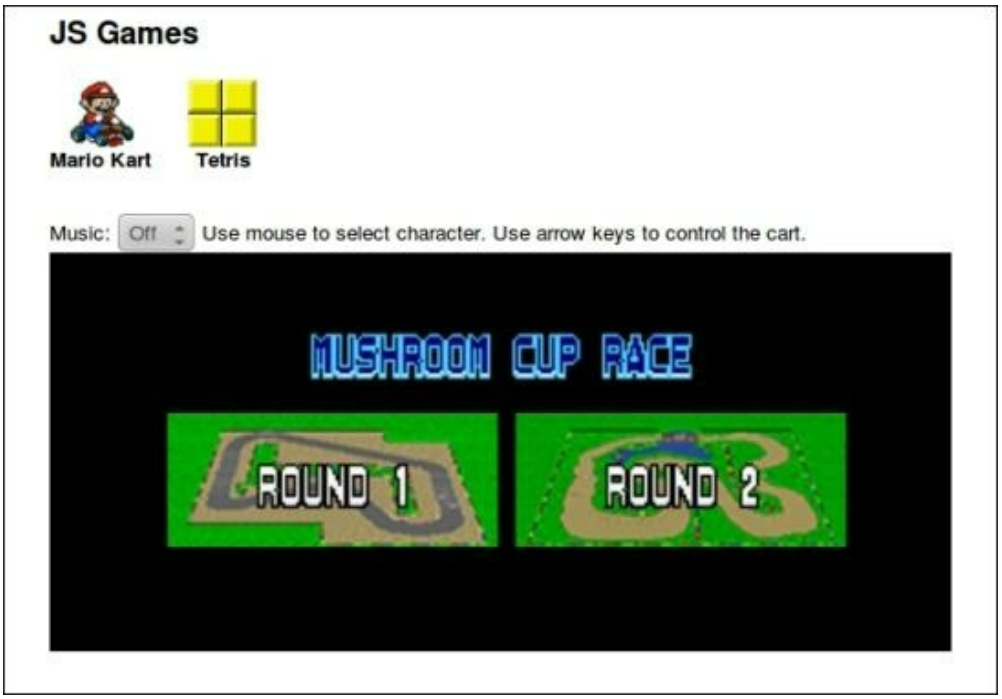
1. Check out and review the tag **jsgames-0.2** under a directory of your choice with the commands:

```
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, as shown:



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 then click on the installed tab (`http://localhost:8080/pluginManager/installed`). Notice that the JSGames 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
```

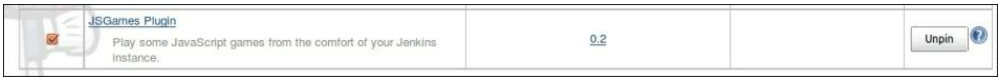
   The output will be similar to:

```
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.

A new version of Jenkins is released roughly every week with bug fixes and feature updates. This leads to delivering improvements quickly to market, but at times it also leads to failures. Pinning a plugin prevents a plugin from being accidentally updated until you have had time to access the stability and value of the newer version. Pinning is a tool to maintain production server stability.

There's more...

The source code includes a top-level `pom.xml` file to control the Maven build process. By convention, the four main source code areas are:

- `src/test`: This contains 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 data between the Java objects in this directory and the views Jenkins finds in the directories under `src/main/resources`.
- `src/main/resources`: This is the location of the view for the plugin. You use the GUI associated with the plugin 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, style sheets, and JavaScript. The location maps to 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

- The *Creating a RootAction plugin* recipe

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 the following content in the `pom.xml` file:

```
<?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>1.584</version>
</parent>
  <artifactId>Startup</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>hpi</packaging>
  <name>Startup</name>
  <repositories>
    <repository>
    <id>repo.jenkins-ci.org</id>

 <url>http://repo.jenkins-ci.org/public/</url>
    </repository>
    </repositories>
    <pluginRepositories>
    <pluginRepository>
    <id>repo.jenkins-ci.org</id>

 <url>http://repo.jenkins-ci.org/public/</url>
  </pluginRepository>
  </pluginRepositories>
  <properties>
    <project.build.sourceEncoding>UTF-8
    </project.build.sourceEncoding>
  </properties>
```

```
</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:

```
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 command:

```
mvn hpi:run -Djetty.port=8090
```

5. The server will now run and generates 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/jenkins`. At the bottom of the page, review the version number of Jenkins, as shown in the following screenshot:



7. Install the UI Samples plugin through the plugin manager (`http://localhost:8090/pluginManager/available`).
8. On the front page, click on the **UI Samples** link:



9. 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 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 running, visiting the GUI example plugin shows examples of creating various GUI elements in Jelly. These elements will come in handy later 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 Jenkins Version 1.584. The reason for this is twofold:

- The dependencies take a lot of space. The Jenkins war file and test war file take about up 120 MB of your local Maven repository. Multiply this number by the number of versions of Jenkins used and you can quickly fill up gigabytes of storage space.
- Holding at a specific Jenkins version stabilizes the recipes.

Feel free to update to the newest and greatest Jenkins version, as the examples in this chapter should still work. In case of difficulties, 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://repo.jenkins-ci.org/public/ to pull in the dependencies. It calls Version 1.584 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 http://repo.jenkins-ci.org/public/org/jenkins-ci/plugins/plugin/.

The details of the Jenkins server and any extra plugins included can be found relative to the preceding URL in `1.584/plugin-1.584.pom`. The UI Samples plugin version is also pegged at Version 1.584

The official page and the most up-to-date information on plugin building can be found at https://wiki.jenkins-ci.org/display/JENKINS/Plugin+tutorial.

See also

- The *Changing the help of the FileSystem SCM plugin* recipe

Changing the help of the FileSystem SCM plugin

This recipe reviews the inner workings of the FileSystem SCM plugin. The plugin allows you to place code in a local directory and have it picked up in a build. As an example, you will change the text in the plugins help file.

Getting ready

Create a directory ready for the code in this recipe. In the newly created directory, download the source of the plugin:

```
git clone
https://github.com/jenkinsci/filesystem_scm-
plugin cd filesystem_scm-plugin/
```

How to do it...

1. Review the tag information and then check out the newest stable tag:

```
git tag -l
filesystem_scm-0.1
filesystem_scm-1.10
filesystem_scm-1.20
filesystem_scm-1.5
filesystem_scm-1.6
filesystem_scm-1.7
filesystem_scm-1.8
filesystem_scm-1.9
git checkout -b filesystem_scm-1.20
```

2. In the top-level directory, edit the `pom.xml` file, changing the version under `<parent>` to `1.584`:

```
<parent>
  <groupId>org.jenkins-ci.plugins</groupId>
  <artifactId>plugin</artifactId>
  <version>1.584</version>
</parent>
```

3. Replace the repositories and `pluginRepositories` stanza with the following code:

```
<repositories>
  <repository>
    <id>repo.jenkins-ci.org</id>
```

```
<url>http://repo.jenkins-ci.org/public/</url>
   </repository>
 </repositories>
 <pluginRepositories>
   <pluginRepository>
     <id>repo.jenkins-ci.org</id>

<url>http://repo.jenkins-ci.org/public/</url>
   </pluginRepository>
 </pluginRepositories>
```

4. Replace the content of `src/main/webapp/help-clearWorkspace.html` with the following code:

```
<div>
  <p>
<h3>HELLO WORLD</h3>
  </p>
</div>
```

5. Run `mvn clean install`. The unit tests fail with the output:

```
Failed tests:
test1(hudson.plugins.filesystem_scm.SimpleAntWildcardFilterTest):
expected:<2> but was:<0>
Tests run: 27, Failures: 1, Errors: 0,
Skipped: 0
```

6. Skip the failing tests by running `mvn clean package -Dmaven.test.skip=true` The plugin is now packaged.

   Upload the plugin to `./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 job named `ch7.plugins.filesystem_scm`.

10. Under the **Source Code Management** section you now have a section called **File System**.
11. Click on the **Help** icon for **Clear Workspace**. You will see your custom message, as shown in the following screenshot:



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 and pointing at the right repository for Maven to download artifacts. 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 is 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, style sheets, and JavaScript files. This is why the help files are stored here. Modifying `help-clearWorkspace.html` updates 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 file, `global.jelly` and `config.jelly`. The `config.jelly` files generate the configuration elements seen when configuring jobs. The `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 for the global plugin configuration under `./work/name_of_plugin.xml`.

See also

- The *Looking at the GUI samples plugin* recipe

Adding a banner to job descriptions

Consider a scenario. Your company has a public-facing Jenkins instance. The owner does not want project owners to write un-escaped 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 management starts buying in unnecessary advice. Within the first five minutes, you ascertain that the escape markup plugin (see the *Finding 500 errors and XSS attacks in Jenkins through fuzzing* recipe in 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 descriptions.

Getting ready

Assuming you are testing locally, create a directory for your project. In your newly created directory, check out the `escape-markup-plugin-0.1` tag of the escaped-markup-plugin:

```
git clone
https://github.com/jenkinsci/escaped-markup-plugin
```

How to do it...

1. Visit the local copy of the plugin source code, list the possible tags, and check out a stable version of the plugin:

```
cd escaped-markup-plugin
git tag -l
git –b 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` file from `1.408` to `1.58`:

```
<parent>
  <groupId>org.jenkins-ci.plugins</groupId>
  <artifactId>plugin</artifactId>
  <version>1.584</version>
</parent>
```

4. Replace the `repositories` and `pluginRepositories` stanza with the following code:

```
 <repositories>
   <repository>
     <id>repo.jenkins-ci.org</id>

<url>http://repo.jenkins-ci.org/public/</url>
   </repository>
 </repositories>
 <pluginRepositories>
   <pluginRepository>
     <id>repo.jenkins-ci.org</id>

<url>http://repo.jenkins-ci.org/public/</url>
   </pluginRepository>
 </pluginRepositories>
```

5. Build the plugin with `mvn install`. The build will succeed. You can now find the plugin at `target/escaped-markup-plugin.hpi`.
6. Install the plugin by visiting the **Advanced** tab under the plugin manager (`http://localhost:8080/pluginManager/advanced`).
7. In the **Upload Plugin** section, upload the `escaped-markup-plugin.hpi` file.
8. Restart the server, for example:

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

9. Visit the Jenkins security configuration page (`http://localhost:8080/configureSecurity`) and review the **Markup Formatters**.
10. Replace `src/main/resources/index.jelly` with the following code:

```
<div>
  This plugin escapes the description of
project, use, view, and build  to prevent
from XSS.
  Revision: Unescaped banner added at the end.
</div>
```

11. Replace the class definition of `src/main/java/org/jenkinsci/plugins/escapedmarkup/EscapedMarkupFormatter.ja` with the following code:

```
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";
    }
  }
}
```

12. Build with `mvn install`. The build fails due to failed tests (which is a good thing).
13. Build again using the following command, this time skipping the tests:

```
mvn -Dmaven.test.skip=true -DskipTests=true
clean install
```

14. Stop Jenkins, as shown:

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

15. 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
```

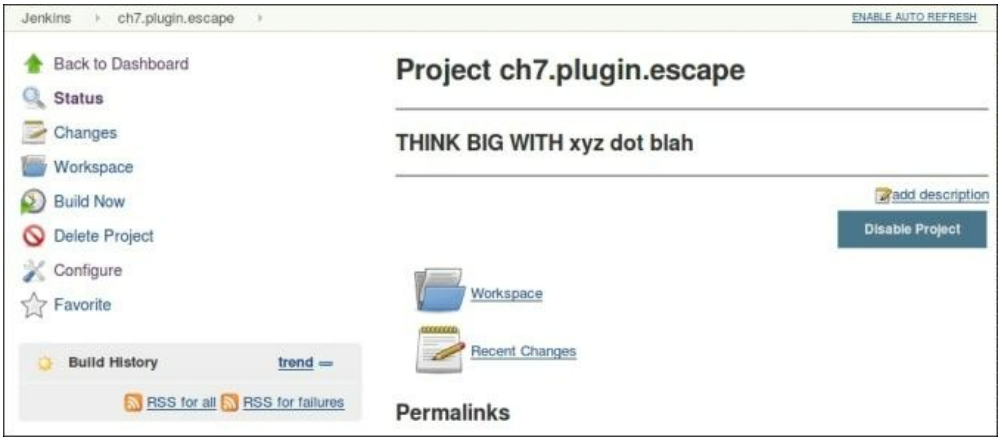
16. Copy the plugin `target/escaped-markup-plugin.hpi` to the Jenkins plugin directory.
17. Restart Jenkins.
18. Visit the installed plugins page: `http://localhost:8080/pluginManager/installed`. You will now see an updated description of the plugin, as shown in the following screenshot:



19. In Jenkins, as an administrator, visit the configure page: `http://localhost:8080/configureSecurity`.
20. For **Markup Formatter** choose **Escaped HTML with BANNER**, as shown in the following screenshot:



21. Click on **Save**.
22. Create a new job named `ch7.plugin.escape`.
23. Within the job's main page, you will now see the banner:



How it works...

The markup plugin escapes tags in descriptions so that arbitrary scripting actions cannot be injected. The use of the plugin was explained in the *Finding 500 errors and XSS attacks in Jenkins through fuzzing* recipe in Chapter 2, *Enhancing Security*.

In this recipe, we adapted the plugin to escape a projects description and then add 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 jobs description. The plugin was redeployed to a sacrificial test instance 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. Whichever 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 part of the functionality of Jenkins. Jenkins has a wealth of extension points (https://wiki.jenkins-ci.org/display/JENKINS/Extension+points). You can even make your own (https://wiki.jenkins-ci.org/display/JENKINS/Defining+a+new+extension+point).

The markup plugin had minimal changes made to it to suit 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 the resulting string plus the banner string was added and passed to the Java writer. The writer was 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.

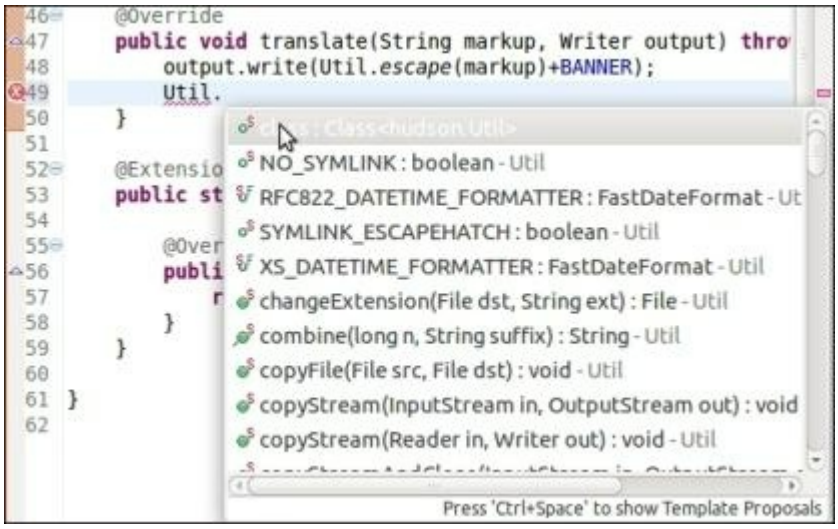
> **Note**
>
> Writing a new plugin and understanding the Jenkins object model takes more effort than copying a plugin that works and then tweaking it. The amount of code changes needed to add the banner feature to an already existing plugin was minimal.

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, the JavaDoc that you can find starting at: http://javadoc.jenkins-ci.org/, and the code completion facilities in 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, sometimes there is 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 a well-written JavaDoc eases a developer's learning curve.

The next screenshot shows code completion at work for Jenkins within the Eclipse IDE:



See also

- The *Finding 500 errors and XSS attacks in Jenkins through fuzzing* recipe in Chapter 2, *Enhancing Security*
- The *Changing the help of the FileSystem SCM plugin* recipe
- The *Creating a RootAction plugin* recipe

Creating a RootAction plugin

Before building your own plugin, it is worth seeing if you can adapt another. In the *Fun with pinning JSGames* recipe, the plugin created a link on the front page, as shown in the following screenshot:



In this recipe, we will use elements of the plugin to create a link on the Jenkins home page.

Getting ready

Create a directory locally 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* recipe, replacing:

```
<artifactId>Startup</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>hpi</packaging>
 <name>Startup</name>
```

   with the following:

```
<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 contents:

```
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 file `src/main/resources/index.jelly` with the following content:

```
<div>
  This plugin adds a root link.
</div>
```

6. In the top-level directory, run the command:

```
mvn -Dmaven.test.skip=true -DskipTests=true
clean install hpi:run -Djetty.port=8090
```

7. Visit the main page: `http://localhost:8090/jenkins`:



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 is calculated as a fixed URL (http://wiki.jenkins-ci.org/display/JENKINS/) with the plugin name after that URL, and 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

- The *Fun with pinning JSGames* recipe

Exporting data

The Job Exporter plugin creates a property file with a list of project-related properties. This is handy glue for when you want Jenkins to pass 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 of 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:

```
(Replace with email)
Started by user Alan Mark Berg
Building remotely on unix-node01 (dumb
functional unix) in workspace
 /home/jenkins-unix-
nodex/workspace/c7.plugins.job_export
[job_exporter] ###################################################################
job-exporter plugin started
     hudson.version: 1.584
     host: home.local
     id: 2014-10-13_16-58-18
     duration: 2.5 sec
     slave: unix-node01
     started: 2014-10-13T16:58:18
     result: SUCCESS
     summary: Executor #1 for unix-node01 :
executing c7.plugins.job_export #1
     executor: 1
     elapsedTime: 2602
     number: 1
     jobName: c7.plugins.job_export
```

```
   we have 1 build cause:
Cause.UserIdCause  Started by user Alan Mark
Berg
     user.id: aaaaaaa
     user.name: Alan Mark Berg
     user.fullName: Alan Mark Berg
     user.emailAddress: xxxxx@yyyy.zzz
   new file
written: /home/jenkins-unix-
nodex/workspace/c7.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 text similar to:

```
#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 be picked up later for reuse by other jobs.

Reviewing the code in `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 that contains the name of the node that the build is running on. The plugin uses a number of these methods to discover information that is later outputted to a properties file.

There's more...

While reviewing plugin code, you can find interesting tricks ready for reuse in your own plugin. The plugin discovered the environment variables by using the method:

```
final EnvVars env = build.getEnvironment(new
LogTaskListener(Logger.getLogger(
```

```
        this.getClass().getName()), Level.INFO));
```

In this method, `EnvVars` is of the class `hudson.EnvVars` (http://javadoc.jenkins-ci.org/hudson/EnvVars.html). `EnvVars` even has a method to get 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

- The *My first ListView plugin* recipe

Triggering events on startup

Often when a server starts up, you will want to have clean up actions performed, for example, running a job that sends an e-mail to all of the Jenkins admins warning them of the start-up 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 job nodes start**.
4. Click on **Save**.
5. Restart Jenkins.
6. Return to the project page, where you will notice that a job has been triggered.
7. Review the build history console output. You will see output similar to:

```
Started due to the start of a node.
Building on master 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 Java class `/src/main/java/org/jvnet/hudson/plugins/triggers/startup/HudsonStartupTrigger`

and overrides the method `start`, which is later called on during Jenkins startup.

The `start` method calls the parent start method, and if it is not a new instance, it will call the method `project.scheduleBuild` 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` that itself extends `hudson.model.Cause`. The plugin overrides the method `getShortDescription()`, returning the string `Started due to Hudson startup`. The string is output to the console as part of the logging:

```
@Override
   public String  getShortDescription()
  {
    return "Started due to Hudson startup.";
  }
```

See also

- The *Triggering events when web content changes* recipe
- The *Groovy hook scripts and triggering events on startup* recipe

Groovy hook scripts and triggering events on startup

In the preceding recipe, you saw that you could use a plugin to run arbitrary startup code. An alternative approach is to place the Groovy script `init.groovy` in the Jenkins home directory. Your Jenkins instance then runs the Groovy script on startup.

Getting ready

Visit the **Scriptler** website http://scriptlerweb.appspot.com/catalog/list and review the currently available Groovy scripts.

How to do it...

1. Visit http://scriptlerweb.appspot.com/script/show/256001 and review the installed plugin list script.
2. In your Jenkins home directory create the following `init.groovy` script, as shown in the following code:

```
import jenkins.model.Jenkins
import java.util.logging.LogManager
def logger =
LogManager.getLogManager().getLogger("")logger.info("RUNNING
init.groovy from
${Jenkins.instance.getRootDir().absolutePath}")logger.info("Here
are the Jenkins environment variables on
startup")
def env = System.getenv()
env.each{
 logger.info("${it}")
}
logger.info("<PLUGINS>")
count=0
for(plugin in
hudson.model.Hudson.instance.pluginManager.plugins.sort())
{
 logger.info( plugin.shortName)
 count = count+1
}
logger.info("<PLUGINS>
\nFound " + count + " plugins")
logger.info("End of init.groovy \n... Goodbye
Cruel world")
```

3. Restart your Jenkins instance:

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

4. Review the log file found at `/var/log/jenkins/jenkins.log`. The output will be similar to the following (note that for the sake of brevity, every second line of timestamps has been removed):

```
INFO: RUNNING init.groovy
from /var/lib/jenkins
INFO: Here are the Jenkins environment
variables on startup
INFO: TERM=xterm
INFO: JENKINS_HOME=/var/lib/jenkins
INFO: SHLVL=1
INFO:
XFILESEARCHPATH=/usr/dt/app-defaults/%L/Dt
INFO: COLORTERM=gnome-terminal
INFO: MAIL=/var/mail/jenkins
INFO:
XDG_SESSION_COOKIE=3e4cca1158bd9704ef8146c500000008-
1413227204.360563-1079526024
INFO: QT_QPA_PLATFORMTHEME=appmenu-qt5
INFO: PWD=/var/lib/jenkins
INFO: LOGNAME=jenkins
INFO: _=/usr/bin/daemon
INFO: NLSPATH=/usr/dt/lib/nls/msg/%L/%N.cat
INFO: SHELL=/bin/bash
INFO:
PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
INFO: DISPLAY=:0.0
INFO: USER=jenkins
INFO: HOME=/var/lib/jenkins
INFO: XAUTHORITY=/home/alan/.Xauthority
INFO: XDG_SEAT=seat0
INFO: XDG_SESSION_ID=c3
INFO: XDG_VTNR=7
INFO: LANG=en_US.UTF-8
INFO: <PLUGINS>
INFO: ant
INFO: antisamy-markup-formatter
INFO: buildresult-trigger
INFO: claim
INFO: credentials
INFO: cvs
INFO:
Found 46 plugins
INFO: End of init.groovy
... Goodbye Cruel world
```

5. Visit `http://localhost:8080/systemInfo` and compare the log file with the system info displayed in Jenkins, as shown in the following screenshot:



Environment Variables

Name ↓	Value
_	/usr/bin/daemon
COLORTERM	gnome-terminal
DISPLAY	:0
HOME	/var/lib/jenkins
JENKINS_HOME	/var/lib/jenkins
LANG	en_US.UTF-8
LOGNAME	jenkins
MAIL	/var/mail/jenkins
NLSPATH	/usr/dt/lib/nls/msg/%L/%N.cat
PATH	/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
PWD	/var/lib/jenkins
QT_QPA_PLATFORMTHEME	appmenu-qt5
SHELL	/bin/bash
SHLVL	1
TERM	xterm
USER	jenkins
XAUTHORITY	/home/alan/.Xauthority
XDG_SEAT	seat0
XDG_SESSION_COOKIE	3e4cca1158bd9704ef8146c500000008-1411995580.205482-1742950951
XDG_SESSION_ID	c3
XDG_VTNR	7
XFILESEARCHPATH	/usr/dt/app-defaults/%L/Dt

How it works...

Jenkins looks for Groovy scripts to run on startup. Acceptable locations are mentioned here (https://wiki.jenkins-ci.org/display/JENKINS/Groovy+Hook+Script) and include:

- `$JENKINS_HOME/init.groovy`
- `$JENKINS_HOME/init.groovy.d/*.groovy` (files run in their lexical order)

Jenkins runs the code from `init.groovy`. It uses the standard `java.util.logging` framework that we initialized with the line `def logger = LogManager.getLogManager().getLogger("")`.

It is advisable to use a logging framework rather than plain old `println` statements. In a logging framework, the configuration is separated out from

the reporting. With little effort, this allows you to change the location of the output (file system, syslog server, and so on.), format, filter, rotate, and so on. For more details of the logger framework, review http://docs.oracle.com/javase/8/docs/technotes/guides/logging/overview.html.

The logger can report at various levels; you can then filter the results. In this example, you used the `info` level as we are not shouting about problems.

First, the script iterated through the environment variables:

```
def env = System.getenv()
env.each{
  logger.info("${it}")
}
```

Next, you used a slightly altered version of the Scriptler example to list all the active plugins in Jenkins, as shown in the following code:

```
for(plugin in
hudson.model.Hudson.instance.pluginManager.plugins.sort())
{
  logger.info( plugin.shortName)
```

Finally, we visually compared with the **System Info** page, which also displays the same environmental details.

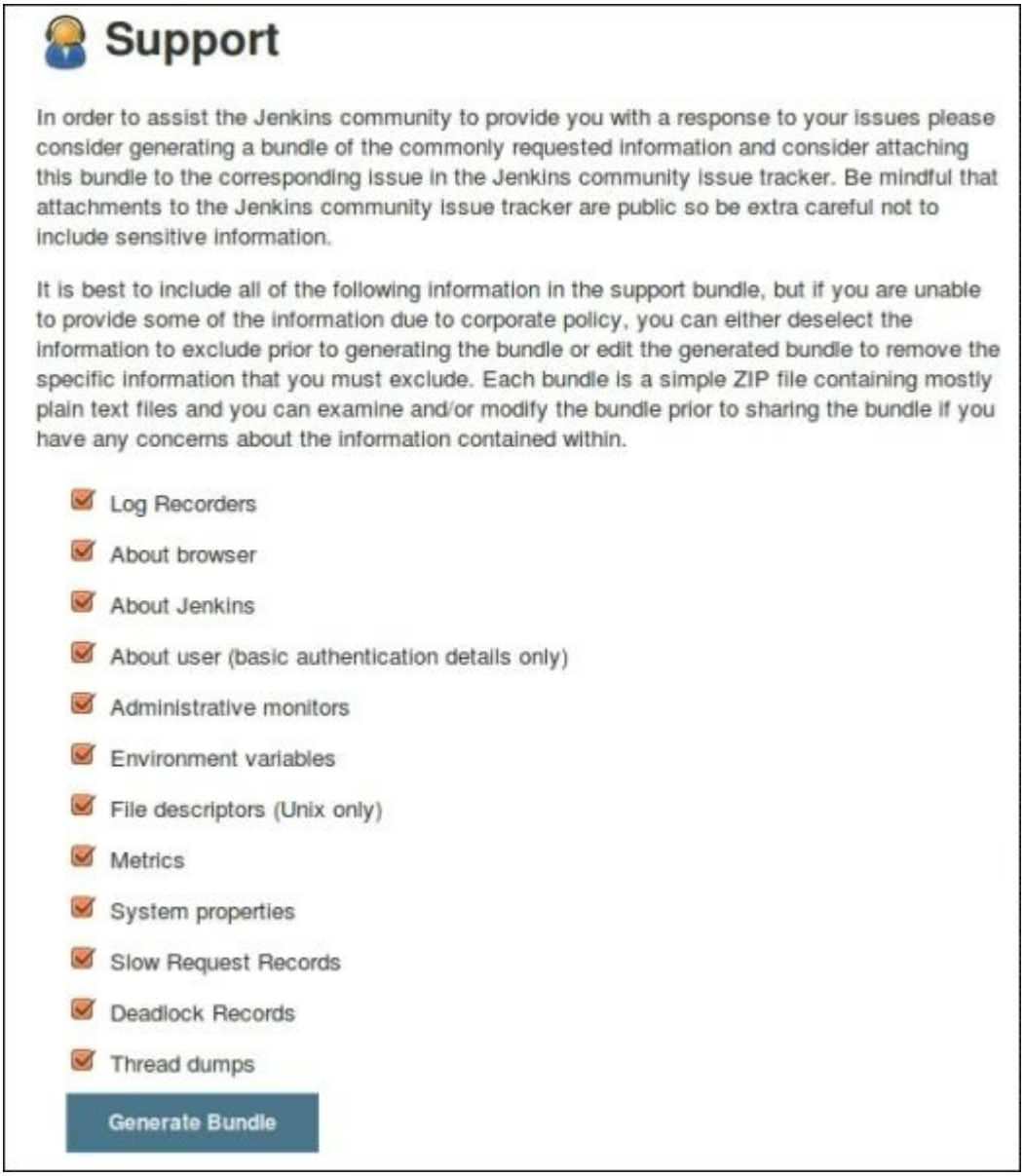
There's more...

If you wish to analyze system information to support debugging, install the Support Core plugin (https://wiki.jenkins-ci.org/display/JENKINS/Support+Core+Plugin). Once installed, you will be able to configure it at the URL `http://localhost:8080/support/`.

Clicking on the **Generate bundle** button will download the support data archive to your browser. Jenkins will also generate a similar archive once an hour and place the ZIP file under the directory `$JENKINS_HOME/support`. The information is stored in multiple text files. For example, the file `support/plugins/active` lists the current version of all active plugins:

```
ant:1.2
antisamy-markup-
formatter:1.2
backup:1.6.1
cas1:1.0.1
credentials:1.16.1
```

The following screenshots shows the **Support** page:



See also

- The *Triggering events on startup* recipe
- The *Triggering events when web content changes* recipe

Triggering events when web content changes

In this recipe, the URL trigger plugin will trigger a build if a web page changes its content.

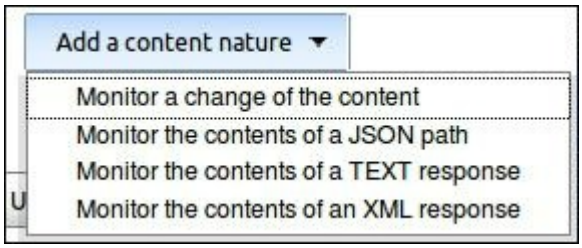
Jenkins is deployed to varied infrastructures. There will be times when standard plugins cannot be triggered by your system of choice. Web servers are well-understood technologies. In most situations, the system to which you want to connect has its own web interface. If the application does not, then you can still set up a web page that changes when the application needs a reaction from Jenkins.

Getting ready

Install the XTrigger plugin (https://wiki.jenkins-ci.org/display/JENKINS/XTrigger+Plugin). This will automatically install the URLTrigger plugin and a number of other plugins that monitor the changes between two checks, for example by running a local script, Groovy code, or monitoring a file or folder change.

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** tick box.
3. Click on **Add URL to monitor**.
4. For **URL** add `http://localhost:8080`.
5. Check **Inspect URL content**.
6. Select from **Add a content nature** the value **Monitor a change of the content**, as shown in the following screenshot:



7. Add a **schedule** for once a minute.
8. Click on **Save**.
9. On the right-hand side, click on the link **URLTrigger Log**:



You will now see the log information update once a minute, with content similar to the following e-mail:

```
Inspecting Monitor a change of the content
for URL http://localhost:8080
Polling started on Oct 13, 2014 5:49:00 PM
Polling for the job ch7.plugin.url
Looking nodes where the poll can be run.
Looking for a candidate node to run the poll.
Looking for a node with no predefined label.
Trying to poll with the last built on node.
Trying to poll on master node.

Polling on master.
Invoking the url:   http://localhost:8080
Inspecting the content
The content of the URL has changed.

Polling complete. Took 0.13 sec.
Changes found. Scheduling a build.
```

10. Delete the job as we don't want to poll `http://localhost:8080` every minute.

How it works...

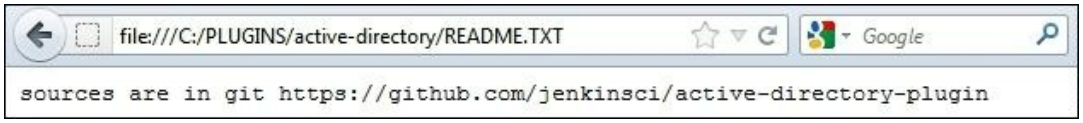
You have configured the plugin to visit our local Jenkins `http://localhost:8080` once a minute, download, and compare for changes. A schedule of once a minute is aggressive; consider using time intervals that are similar to those on your SCM repositories, such as once every 5 minutes.

As there were subtle differences in each page returned, the trigger was activated. This was verified by looking in the **URLTrigger Log**.

The URLTrigger plugin can also be used for JSON and text or XML responses.

There's more...

Part of the URI schema points 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, as shown in the following screenshot:



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

- The *Triggering events on startup* recipe

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, as shown in the following screenshot:



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 plugins; 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).

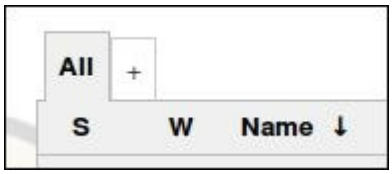
How to do it...

1. Install the source code locally in a directory of choice:

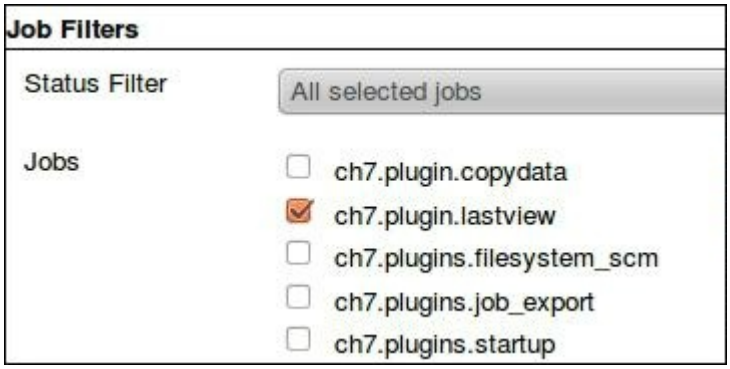
```
git clone
https://github.com/jenkinsci/lastfailureversioncolumn-
plugin
git clone
https://github.com/jenkinsci/lastsuccessversioncolumn-
plugin
```

```
git clone
https://github.com/jenkinsci/lastsuccessdescriptioncolumn-
plugin
```

2. In Jenkins, create a new free-style job named `ch7.plugin.lastview`. No further configuration is necessary.
3. On the main page, click on the **+** tab next to the **All** tab, as shown in the following screenshot:



4. Create a **List View** named `LAST`.
5. Under **Job Filters** | **Jobs** check the **ch7.plugin.lastview** tick box:



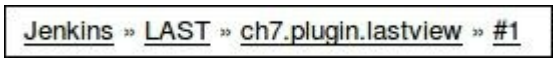
6. Click on **OK**. You are returned to the main page with the **LAST** list view shown:



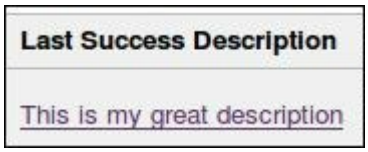
7. Press the **Build** icon to run the `ch7.plugin.lastview` job:



8. Refresh your page. The **Last Success Version** column now has data with a link to the builds history.
9. In the **Last Success Description** column, click on the **N/A** link.
10. On the right-hand side, click on **Add description**.
11. Add the description for the build: **This is my great description**.
12. Click on **Submit**.
13. Return to the **LAST** list view by clicking on **LAST** in the breadcrumb displayed at the top of the page, as shown in the following screenshot:



14. 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. Details are useful for making quick decisions about projects, for example, when a build succeeds, adding a description to the build such as **Updated core libraries to work with modern browsers** gives a casual viewer an overview of the last significant action in the project without delving down into the source code. This saves a significant amount of clicking about:



There's more...

There is a healthy supply of ListView plugins, these include:

- **The Extra Columns plugin**: This 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 more efficiently.
- **The Cron Column plugin**: This displays the scheduled triggers in the project and whether they are enabled or disabled. This is useful if you want to compare system monitoring information from the Melody plugin.
- **The Emma Coverage** plugin: This displays 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.
- **The Progress Bar plugin**: This displays a progress bar for running jobs. This adds a feeling of activity to the front page.

See also

- The *Creating my first ListView plugin* recipe
- The *Efficient use of views* recipe in Chapter 4, *Communicating Through Jenkins*
- The *Saving screen space with the Dashboard View plugin* recipe in Chapter 4, *Communicating Through Jenkins*
- The *Monitoring via JavaMelody* recipe in 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 comments. The code for the content of the column is a placeholder, just waiting for you to replace it with your own brilliant experiments.

Getting ready

Create a directory ready for the code.

How to do it...

1. Create a top-level `pom.xml` file with the content as follows:

```
<artifactId> rootaction</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>hpi</packaging>
   <name>Jenkins Root Action Plugin</name>
<?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>1.548</version>
</parent>
  <artifactId>rootaction</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>hpi</packaging>
   <name>Jenkins Root Action Plugin</name>
  <repositories>
    <repository>
      <id>repo.jenkins-ci.org</id>

 <url>http://repo.jenkins-ci.org/public/</url>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>repo.jenkins-ci.org</id>

 <url>http://repo.jenkins-ci.org/public/</url>
    </pluginRepository>
  </pluginRepositories>
 <properties>
```

```
<project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
  </properties>
</project>
```

2. Create the directory `src/main/java/jenkins/plugins/comments`.
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 directory `src/main/resources/jenkins/plugins/comments/CommentsColumn`.
5. In the `CommentsColumn` 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 `CommentsColumn` 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 file `index.jelly` 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: `http://localhost:8090/jenkins/view/All/newJob`. Create a new free-style job named `ch7.plugin.1`.
12. On the main Jenkins page, `http://localhost:8090/jenkins`, 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]**, as shown in the following screenshot:



In the default Firefox browser for Ubuntu, you can change the preferred language under the **Edit/Preferences** content tab, in the **Languages** section, as shown in the following screenshot:



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 the 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 can 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 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 property files, depending on the value of 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 property 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, 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.

> **Note**
>
> 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 is 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 a 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). As an example, review https://github.com/jenkinsci/lastfailureversioncolumn-plugin/blob/master/LICENSE.txt. Its structure 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

- The *Reviewing three ListView plugins* recipe

Appendix A. Processes that Improve Quality

Quality assurance requires that the expert pays attention to a wide range of details. Rather than being purely technical, many of these details relate to human behavior. This chapter mentions both the soft skills needed to run successful projects and the configuration skills codified in the recipes of this book. Here are a few hard-learned observations.

Fail early

The later in the software life cycle you correct a problem, the more costly it will become. Failing early is significantly cheaper than failing later. Continuous Integration allows you to automatically fail software early. Adding extra tests through plugins or connected cloud services gives greater opportunity to face your issues early, improving quality and decreasing costs. Embrace acknowledging issues because you are saving time and money.

The following are a few relevant resources:

- The *Deliberately failing builds through log parsing* recipe in Chapter 1, *Maintaining Jenkins*
- The *Triggering failsafe integration tests with Selenium Webdriver* recipe in Chapter 6, *Testing Remotely*
- Error Cost Escalation Through the Project Life Cycle at http://ntrs.nasa.gov/search.jsp?R=20100036670

Data-driven testing

You can efficiently cover your 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 text files. This allows JMeter to read out parameters from your CSV files, such as hostname, and transverse your infrastructure. This enables one test plan to attack many servers. The same is true for SoapUI; by adding an Excel data source and looping through the rows, you can test an application with many different test users who each have a range of roles. Data-driven testing has a tendency to be maintainable. During refactoring, instead of changing your test plan as the URLs in your application change, you can factor the URLs out into CSV files.

The following are a few relevant resources:

- The *Creating JMeter test plans* recipe in Chapter 6, *Testing Remotely*
- The *Writing test plans with SoapUI* recipe in Chapter 6, *Testing Remotely*
- The *Custom setup scripts for slave nodes* recipe in Chapter 6, *Testing Remotely*
- Test cloud-based applications with Apache JMeter at http://www.ibm.com/developerworks/cloud/library/cl-jmeter-restful/
- The data-driven approach with SoapUI at http://www.soapui.org/Data-Driven-Testing/functional-tests.html

Learning from history

Teams tend to have their own coding habits. If a project fails because of the quality of the code, try to work out which code metrics would have stopped the code reaching production. Which mistakes are seen repeatedly? Take a look at the following examples:

- **Friday afternoon code failure**: We all are human and have secondary agendas. By the end of the week, programmers may have their minds focused elsewhere other than on 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 near the time of least attention.
- **Code churn**: A sudden surge in code commits just before a product is being moved from an acceptance environment to product 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 naïve push towards destruction. If a project fails and QA is overwhelmed due to a surge of code changes, look at setting up a warning Jenkins job based on commit velocity. If necessary, you can display your own custom metrics. For more information, refer to the *Plotting alternative code metrics in Jenkins* recipe in Chapter 3, *Building Software*.
- **A rogue coder**: Not all developers create code of the same uniform 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 accepted practice, builds will fail with great regularity. For more information, refer to 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 that 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, then Jenkins will start shouting. For more information, refer to the *Activating 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, 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. For more information, refer to the *Testing and then promoting builds* recipe in Chapter 7, *Exploring Plugins*.

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; the tests need to be reusable 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.

For more information, refer to 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 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 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. For more detail, refer to 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 more detail, refer to the *Configuring Jetty for integration tests* recipe in Chapter 3, *Building Software*.

Visualize, visualize, visualize

When you have many projects scattered across multiple servers developed by different teams and individuals, it is difficult to understand the key metrics and emerging issues.

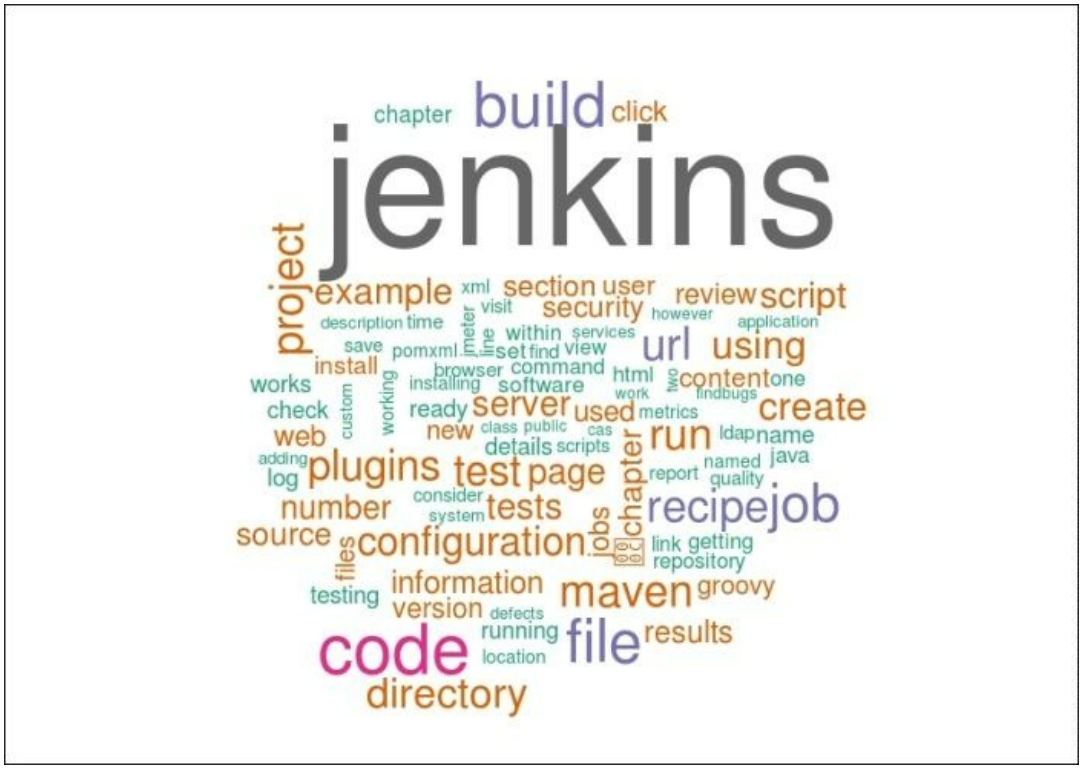
With 80 percent of information going through to your brain being visual and your brain being an excellent pattern recognizer, one of the tricks to understand the underlying complexity is to visualize the results of your Jenkins jobs.

SonarQube is an excellent starting point to visualize and gain an overview of the overall quality of projects and for delving into relationships and couplings between different areas of the code. For more information, refer to the *Integrating Jenkins with SonarQube* recipe in Chapter 5, *Using Metrics to Improve Quality*.

However, if you have specialized requirements, you will need to build graph generation. Test results are usually stored in XML or CSV format. Once you accumulated the results, you can easily transform them with your language of choice.

R is a language designed for statisticians and scientists. After data is gathered, it is explored to try and discover which variables are related. For this purpose, the R community has created many helper graphic packages. For more detail, refer to the *Analyzing project data with the R plugin* recipe in Chapter 5, *Using Metrics to Improve Quality*.

The following graphic is a wordcloud summarizing this book's content. Within 10 lines of code, the R language generated it. R uses a combination of the `tm` and `wordcloud` packages. You can download the code from the book's website.



The graphics libraries in R have great examples of what is achievable. The `example()` command placed around the function of your choice runs example code based on the function. The following code displays graphs for the `plot` and `hist` graphics functions. The code is included with the plots:

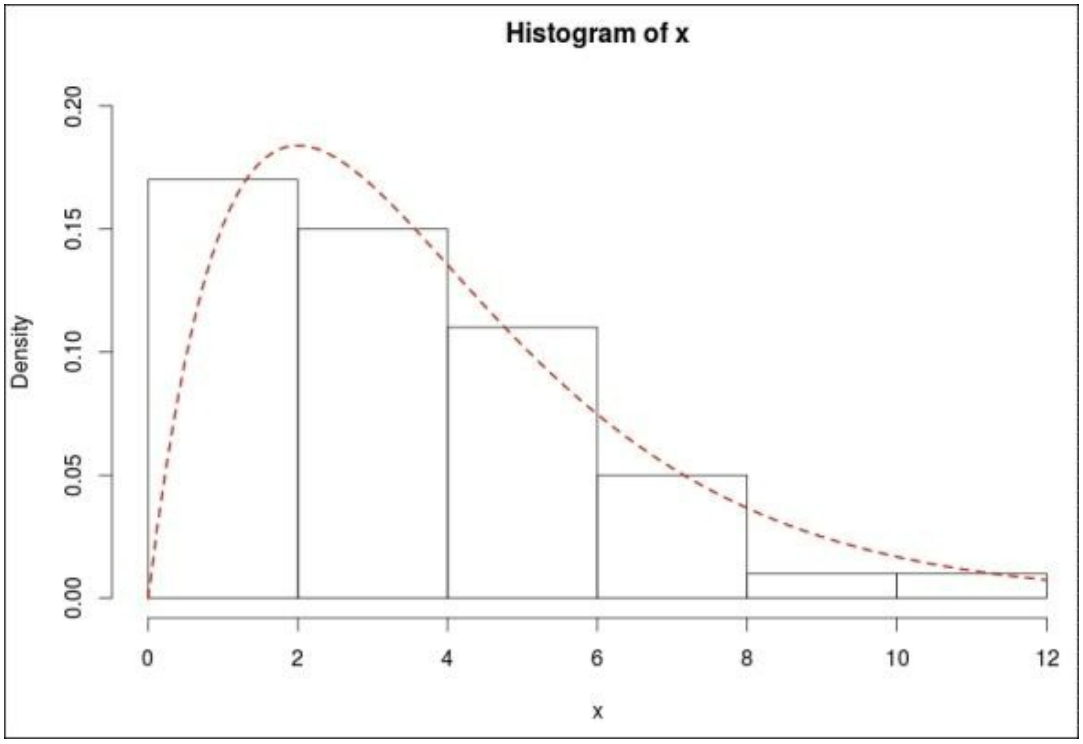
```
example(plot)
example(hist)

x <- rchisq(100, df = 4)
hist(x, freq = FALSE, ylim = c(0, 0.2))
curve(dchisq(x, df = 4), col = 2, lty = 2, lwd =
2, add = TRUE)
```

Once you have discovered a new, interesting function, you can explore its help further by searching the documentation. For example, typing `?rchisq` outputs the following information:

```
Density, distribution function, quantile function and random
generation for the chi-squared (chi^2) distribution with df
degrees of freedom and optional non-centrality parameter ncp.
```

For more information, refer to the Simplifying *powerful visualizations using the R plugin* recipe in Chapter 4, *Communicating Through Jenkins*.

The following screenshot shows the graph generated by the `hist` function:



The `rgl` package has a wide range of features for generating impressive graphics.

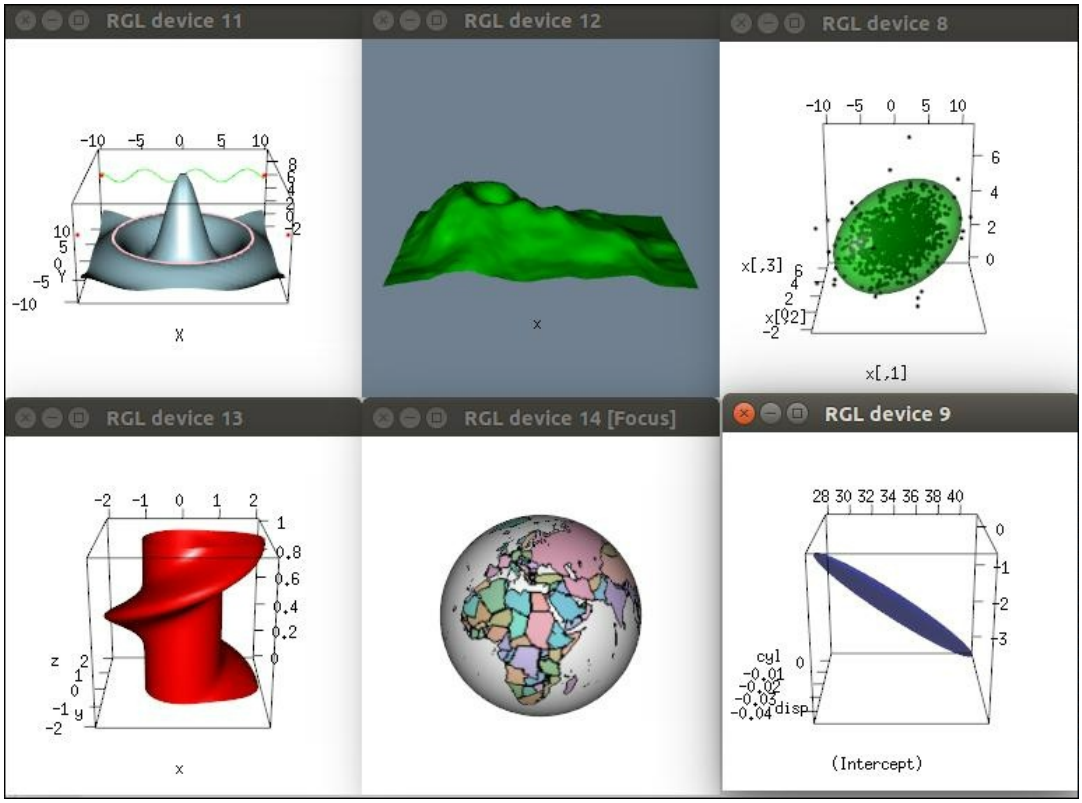
To install the `rgl` package and its dependencies from the Ubuntu command line, type the following command:

```
sudo apt-get install r-cran-rgl
```

To look at some examples, run the following commands from the R console:

```
library(rgl)
example(plot3d)
example(ellipse3d)
example(surface3d)
example(persp3d)
```

You will see output similar to the following screenshot:



The following are a few relevant resources:

- A full list of official R packages at http://cran.r-project.org/web/packages/available_packages_by_name.html
- Documentation for the `rgl` package at http://cran.r-project.org/web/packages/rgl/rgl.pdf
- Documentation for the `tm` package at http://cran.r-project.org/web/packages/tm/tm.pdf
- Documentation for the `wordcloud` package at http://cran.r-project.org/web/packages/wordcloud/wordcloud.pdf

Conventions are good

By following conventions, you decrease the amount of maintenance and lower the number of defects hidden in your code. Coding conventions are particularly important when more than one developer is involved in writing the code. Conventions aid readability. Consistently indented code focuses the eye on poorly written sections. Well-structured variable names help avoid naming collisions between codes written in different parts of the organization. Structure in naming highlights the data that you can later move to configuration files and increases the opportunity for semi-automatic refactoring using regular expressions, for example, you can write a short piece of R code to visualize the number of global variables you have per module. The more global variables you have, the greater the risk of using the same variable for multiple purposes. Hence, the plot is a rough indicator of smelly code.

The following are a few relevant resources:

- The *Deliberately failing builds through log parsing* recipe in Chapter 1, *Maintaining Jenkins*
- The *Plotting alternative code metrics in Jenkins* recipe in Chapter 3, *Building Software*
- The *Creating custom PMD rules* recipe in Chapter 5, *Using Metrics to Improve Quality*
- Google's code of conventions for Java programmers at https://google-styleguide.googlecode.com/svn/trunk/javaguide.html
- Coding and naming conventions for the GCC project at https://gcc.gnu.org/codingconventions.html

Test frameworks and commercial choices are increasing

In the past few years, there has been a lot of improvement in test automation. One example is that static code review is being used more thoroughly for security checks. SonarQube is an all-encompassing reporter of project quality and new frameworks are emerging to improve on the old. Here are a few implications:

- **SonarQube**: This measures project quality. Its community is active. SonarQube will evolve faster than the full range of Jenkins quality metrics plugins. Consider using Jenkins plugins for early warnings of negative quality changes and SonarQube for in-depth reporting. For more information, refer to the *Integrating Jenkins with SonarQube* recipe in Chapter 5, *Using Metrics to Improve Quality*.
- **Static code review tools**: These 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. For more detail, refer to the *Finding security defects with FindBugs* recipe in Chapter 5, *Using Metrics to Improve Quality*.
- **Code search**: Wouldn't it be great if code search engines 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 code's communities.
- **The cloud**: CloudBees allows you to create on-demand slave nodes in the cloud. Expect more kinds of cloud-like integrations around Jenkins.

For more information about SonarQube features and the CloudBees cloud service, visit http://www.sonarqube.org/features/ and http://www.CloudBees.com/products/dev.

Offsetting work to Jenkins nodes

Thanks to its wealth of plugins, Jenkins can easily connect to many types of system. Therefore, Jenkins usage can grow virally in an organization. Testing and JavaDoc generation takes up 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.

If you are using JMeter for your performance tests at scale, consider offloading from Jenkins to a cloud service such as BlazeMeter (http://blazemeter.com/).

For functional testing with Selenium, there is also a wide range of cloud services. Consider using them not only because of load, but also because of the use of a wide range of browser types and versions offered. One example of a commercial service is Sauce Labs (https://saucelabs.com/). It is worth periodically reviewing the market for new cloud services.

The following are a few relevant resources:

- The *Monitoring via JavaMelody* recipe in Chapter 1, *Maintaining Jenkins*
- The *Creating multiple Jenkins nodes* recipe in Chapter 6, *Testing Remotely*
- The *Custom setup scripts for slave nodes* recipe in Chapter 6, *Testing Remotely*

Starving QA/integration servers

A few hundred years ago, coal miners would die because of the build-up of methane and carbon monoxide in the mines. To give early warning of this situation, canaries 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 in your acceptance environment: deliberately starve them of resources. If they fall over, you will have enough time to review before watching the explosion in production.

For more information, refer to the *Monitoring via JavaMelody* recipe in Chapter 1, *Maintaining Jenkins*.

Reading the change log of Jenkins

Jenkins practices what it preaches. Minor version number releases occur about once a week. New features appear, bugs are resolved, and new bugs introduced. In general, the great majority of changes lead to improvement, but a few do not. However, when introduced, bugs are generally caught early and removed quickly.

Before updating Jenkins for new features and potential stability glitches, it's worth reading the changelog (http://jenkins-ci.org/changelog). Occasionally, you might want to speed up a deployment to production because of a security issue or miss a version due to a stability blooper.

> **Note**
>
> If you are focused on stability rather than feature richness, consider using the older but more stable long-term support release. For more details visit: https://wiki.jenkins-ci.org/display/JENKINS/LTS+Release+Line.

Avoiding human bottlenecks

The simpler your testing environment is, the less skill you'll need to maintain it. As you learn to use the plugins and explore the potential of new tools and scripting languages, the more knowledge the organization needs to maintain a stable system. If you wish to go on holidays without random text messages asking for advice, make sure that your knowledge is transferred to at least a second person. This sounds obvious, but in the rush of your daily load, this principle is often forgotten or put to one side.

One of the easiest ways to share knowledge is to send a couple of developers off to the same conferences and events together (https://www.CloudBees.com/company/events/juc).

This is where managers play a significant role in knowledge dissemination. They need to plan in time and activities for the sharing of knowledge, rather than expecting it to happen by magic.

Avoiding groupthink

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 groupthink (http://en.wikipedia.org/wiki/Groupthink) by the project team or resource owners. If the team has the wrong collective attitude, as a quality assurance professional it is much harder to inject hard-learnt realism. Quality assurance is not only about finding and capturing defects as early as possible, it is also about injecting 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 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 metrics that fail a build. One approach is to compare a small successful project to a small failed project. If later there is a disagreement, then the debate is about process and numbers rather than personality.

For more information, refer to the *Looking for "smelly code" through code coverage* recipe in Chapter 5, *Using Metrics to Improve Quality*.

Training and community

Training and participating in Jenkins and the wider tester community are vital for long-term learning paths that lead to optimized environments. Here are a few relevant resources:

- CloudBees is a commercial company working with Jenkins cloud services. At the time of writing this book, CloudBees' CTO is Kohsuke Kawaguchi, the father of Jenkins. CloudBees provides a number of training opportunities and conference events. The company's training information can be found at http://www.CloudBees.com/jenkins/training.
- When starting with an online community, it is wise to first review and participate in the mailing lists. This allows you to judge your own standard and gradually become recognized. The mailing lists are summarized at http://jenkins-ci.org/content/mailing-lists. Once you are confident that you can productively participate, consider progressing to real-time interactions through the IRC channel at https://wiki.jenkins-ci.org/display/JENKINS/IRC+Channel.
- The ISTQB software certification body keeps example documentation on its website for its software tester exams. You can find the documentation at http://www.istqb.org/downloads.html.
- The slideshow from Kohsuke Kawaguchi explaining the funnel of participation as a method to build a community is available at http://www.slideshare.net/kohsuke/building-developer-community.

Visibly rewarding successful developers

This is a call to resource managers. Developers and testers specialize in technical matters that are at times hard to explain to those outside their problem domain. To reach the highest level of expertise and to keep track of trends requires time (sometimes a lot of their own time), energy, and motivation. Undermining their motivation or underestimating the time required to build their skills will ultimately decrease the quality of your products and will cost more in the end. Consider what you can do to support them, from pay scale jumps, learning paths, reserving time in the week for developers to read and practice new ideas, to conferences and gadgets. For example, after a pay rise, Kickstarter (https://www.kickstarter.com/) is a great place to look for motivational rewards and to stimulate the developers' creative muscle.

Finally, do not make developers do non-development stuff. In general, they need to be highly focused on the complex task of understanding detailed requirements and turning them into rock-solid code.

Stability and code maintenance

This book mentions many plugins and a number of languages and testing tools. It is OK to experiment in development and then push to acceptance, but the more diversity you have in production, the more skills are needed to maintain and especially to write a fluent workflow. Subtle choices, such as pinning Jenkins plugins at known versions and keeping the production version of your Jenkins server stable for fixed periods, help with up-time. Just as importantly, monitoring the load and offsetting most of the jobs away from the master Jenkins ensues a high degree of determinism in the timing of the jobs.

To limit job maintenance implies keeping configuration simple and similar. This is not realistic in a complex organization with a high degree of diversity. Using a test-driven approach helps; conventions also simplify configuration. As the diversity increases, communicating and agreeing the conventions becomes important. Simple strategies—such as one source of documentation wisdom (for example, a communal wiki), regular lessons, learned meetings, and weekly reviews—become vital.

Resources on quality assurance

It is a mistake to consider testing to be the sole responsibility of the testers. Coders should feel responsible for the quality of their code, architects for the quality of their designs, managers for the ethos of the project and project planning, and so on. Here are some examples of a range of practical resources on actionable quality assurance—this is not just for the testers:

- There are many wise words on avoiding classic mistakes based on years of hard knocks and bruising. One well thought-out set of comments can be found at http://www.exampler.com/testing-com/writings/classic/mistakes.html.
- If unit tests cover your code thoroughly, then if you break a piece of code during an update, you will know this quickly during the next build. JUnit is arguably the most well-known framework in this genre. You can find the framework's home page at http://junit.org/.
- The Jenkins home page (http://jenkins-ci.org/) covers a wealth of information around the practicalities of configuration, plugins, the community, and hints and tips.
- The **Open Web Application Security Project** (**OWASP**) is a great source of information and tools on security testing. OWASP is focused on improving the security of software. Its mission is to make software security visible so that individuals and organizations worldwide can make informed decisions about true software security risks. You can find the OWASP home page at https://www.owasp.org/index.php/Main_Page.
- If you want a hardcopy or e-book of any of the OWASP material, then you can download or buy from Lulu (http://www.lulu.com/spotlight/owasp).
- You can find OWASP's security testing guide at https://www.owasp.org/index.php/OWASP_Testing_Guide_v4_Table_of_Contents.
- One popular example of a commercial company selling Jenkins infrastructure in the cloud is Sauce Labs (https://docs.saucelabs.com/ci-integrations/jenkins/).
- There are a number of excellent and free-to-download software testing magazines. The Professional Tester is one such example and is available at http://www.professionaltester.com/.
- uTest is the world's largest open community dedicated to professional testers and software testing. Its sole purpose is to promote and advance the testing profession, and the people who do this vital work. For more information, visit http://www.utest.com/about-us.
- There are more and more free MOOC courses and a number of them support the learning paths of software testers. You can find a full list of currently running MOOC courses at https://www.mooc-list.com/.
- There are many hundreds of excellent blogs centered on testing. Martin Fowler's blog (http://martinfowler.com/) is a great resource and

the father of Jenkins, Kohsuke Kawaguchi, has another (http://kohsuke.org/).

- Static code review allows jobs to find a wide range of issues without human intervention. A set of articles I wrote about static code review for the Free Software Magazine is available at http://www.freesoftwaremagazine.com/articles/destroy_annoying_bugs_part_1.

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 it is that the relevant tests capture the quintessential details of user expectations. Look for new Jenkins plugins that support tools that lower the learning curve. For more information, refer to 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.

> **Tip**
>
> **Warning**
>
> Before adding defects, make sure that the team has agreed to the process or you might be getting angry e-mails late at 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 unexpected input. Not only is this good for security testing, but also for boundary testing. If your server returns an unexpected error, then use a fuzzer to trigger a more thorough review. Fuzzers and link crawlers are a cheap way to increase the code coverage of your tests. For more information, refer to the *Finding 500 errors and XSS attacks in Jenkins through fuzzing* recipe in Chapter 2, *Enhancing Security*.
- In your development environment, periodically review for new Jenkin plugins. The number of plugins is increasing rapidly and there may be new ways for Jenkins to connect different parts of your organization's infrastructure to Continuous Integration.

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 great attention to detail. Jenkins can pay attention to many of the details and then shout loudly when violations occur.

Each project is different and there are many ways to structure the workflow. Luckily, with over 1,000 plugins and the number rising rapidly, 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 Java programmers to adapt or create their own plugin.

> **Note**
>
> 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. If you agree, consider participating.

Index

A

- Abstract Syntax Tree (AST)
  - URL / How it works...
  - about / How it works...
- Access Control Lists (ACLs) / There's more...
- Analysis Collector
  - URL / Introduction, Faking Checkstyle results
- Android 1.6 / Android 1.6 and Hudson apps
- Android x86 project / Virtualbox and the Android x86 project
- Ant-contrib library
  - URL / Running Ant through Groovy in Maven
- AntBuilder instance
  - URL / How it works...
- Ant filesets
  - URL / Introduction
- AntRun plugin
  - URL / Running Ant through Groovy in Maven
- Ant tasks, Maven
  - running, through Groovy / Running Ant through Groovy in Maven, How to do it...
- Apache Archiva
  - URL / There's more...
- Apache Rat project
  - URL / Looking at license violations with Rat
- Apereo Open Academic Environment (OAE)
  - URL / There's more...

- archiving
    - notifying for / Signaling the need to archive, How to do it..., How it works...
- Artifactory
    - URL / There's more...
- audit logs
    - missing / Missing audit logs
- Audit Trail plugin
    - working with / Working with the Audit Trail plugin, How it works...
- Avatar plugin
    - URL / Getting ready
    - about / How it works...

B

- backups
    - about / Backing up and restoring
    - performing / Getting ready, How to do it..., How it works...
    - options / How it works...
    - permission error, checking / There's more...
- BadBoy
    - URL / There's more...
- banner
    - adding, to job descriptions / Adding a banner to job descriptions, How to do it..., How it works...
- Behavior Driven Development (BDD)
    - about / Introduction
- Blamer
    - URL / How to do it...
- bots

    - about / More information on bots

- Bourne Again SHell
    - URL / How it works...

- build descriptions
    - information, exposing through / Exposing information through build descriptions, How to do it..., How it works...

C

- CAPTCHA
    - about / Avoiding sign-up bots with JCaptcha

- CAS
    - about / Introduction
    - server, installing / Installing a CAS server, How to do it..., How it works...
    - resources / A few useful resources

- CAS SSO server
    - URL / Installing OpenLDAP

- changelog
    - URL / Reading the change log of Jenkins

- Checkstyle
    - about / Checking code style using an external pom.xml file
    - used, for checking code style with pom.xml file / Checking code style using an external pom.xml file, How to do it..., How it works..., There's more...
    - installing / Getting ready
    - URL / Getting ready, Faking Checkstyle results
    - reports, faking / Faking Checkstyle results, How to do it..., How it works...

- Claim plugin
    - URL / There's more...

- CloudBees

    - URL / How it works...
    - about / Training and community

- Cobertura
    - URL / Introduction, Looking for "smelly" code through code coverage
    - about / Looking for "smelly" code through code coverage
    - used, for testing code coverage / Looking for "smelly" code through code coverage, How to do it..., How it works...
    - installing / Getting ready, How to do it...

- COCOMO model
    - about / Software cost estimation
    - URL / Software cost estimation

- code.ohoh.net
    - about / Goodbye Google code search; hello code.ohoh.net

- coding conventions
    - URL / Introduction

- command-line interface (CLI) / Modifying the Jenkins configuration from the command line
- Common Name (CN) / How it works...
- config.xml file / How it works...
- Continuous Integration (CI)
    - about / Using a test Jenkins instance

- Copy Data to Workspace plugin
    - URL / Getting ready

- cross-site request forgery
    - URL / How it works...

- CSS 3
    - about / CSS 3

- CSS 3 quirks / How it works...
- custom ListView plugin

    - creating / Creating my first ListView plugin, How to do it..., How it works...

- custom PMD rules
    - creating / Creating custom PMD rules, How to do it..., How it works...
- custom security flaw / My own custom security flaw
- custom setup scripts
    - running, on slave nodes / Custom setup scripts for slave nodes, How to do it..., How it works...
- custom sound
    - creating, with HTML5 browsers / Making noise with HTML5 browsers, How to do it...
- cyclic dependency
    - about / There's more...
- Cygwin
    - URL / How it works...

D

- Dashboard View plugin
    - using / Saving screen space with the Dashboard View plugin, How to do it...
    - about / Saving screen space with the Dashboard View plugin
    - installing / Getting ready
- data
    - exporting / Exporting data, How to do it..., How it works...
- Data Camp
    - URL / For more information
    - about / For more information
- Debian Almquist Shell
    - URL / How it works...

- Denial Of Service (DOS) attack / Avoiding sign-up bots with JCaptcha
  - about / There's more...
- Deploy plugin
  - URL / Getting ready
- differential backups
  - about / How it works...
  - cleaning / How it works...
- disk usage
  - reporting / Reporting overall storage use, How it works...
- disk usage plugin
  - about / Reporting overall storage use
  - installing / Getting ready, How it works..., There's more...
- disk usage violations
  - warning / Adding a job to warn of storage use violations through log parsing, How to do it..., How it works...
- Distinguished Name (DN) / How it works...
- Distributed Denial Of Service attack (DDOS) / Avoiding sign-up bots with JCaptcha
- Don't Repeat Yourself (DRY) principle
  - about / The Don't Repeat Yourself principle
  - URL / The Don't Repeat Yourself principle

  / Considering test automation as a software project
- DropDown ViewsTabBar plugin
  - about / Efficient use of views
  - installing / Getting ready
  - using / How to do it..., How it works...
- Dynamic Kernel Module Support (DKMS) / How it works...

E

- Eclipse
  - about / Eclipse templates for JSP pages
  - URL / Eclipse templates for JSP pages

- elements, JMeter test plans
  - Thread Group / How it works...
  - Cookie manager / How it works...
  - CSV Data Set Config / How it works...
  - View Results Tree / How it works...
- Email plugin
  - URL / Installing OpenLDAP
- Emma
  - URL / Introduction, There's more...
  - about / There's more...
- envfile plugin
  - URL / Manipulating environmental variables
- envinject plugin
  - URL / Manipulating environmental variables
  - about / How it works...
- environmental variables
  - manipulating / Manipulating environmental variables, How to do it..., How it works..., There's more...
- events
  - triggering, on startup / Triggering events on startup, How it works...
  - triggering, if web content changes / Triggering events when web content changes, How to do it..., There's more...
- evil URL / Finding 500 errors and XSS attacks in Jenkins through fuzzing
- example site generation configurations
  - searching for / Searching for example site generation configurations
- exclude patterns
  - testing / Testing exclude patterns
- eXtreme Feedback Panel plugin

    - URL / Getting ready

- extreme feedback plugin
    - about / An extreme view for reception areas
    - adding / An extreme view for reception areas, How to do it...
    - appearance, modifying / How to do it...

F

- failsafe integration tests
    - triggering, with Selenium WebDriver / Triggering failsafe integration tests with Selenium WebDriver, How to do it..., How it works...

- Failsafe plugin
    - URL / How it works...

- Favorite plugin
    - about / Personalizing Jenkins
    - URL / Getting ready

- fb-contrib project
    - URL / Enabling extra FindBug rules

- file system SCM plugin
    - inner workings, reviewing / Changing the help of the FileSystem SCM plugin, How to do it..., How it works...

- File System SCM plugin
    - setting up / Setting up a File System SCM

- FindBugs
    - about / Introduction, Finding bugs with FindBugs
    - used, for searching bugs / Finding bugs with FindBugs, How to do it..., How it works...
    - installing / Getting ready
    - URL / Getting ready
    - extra bug detectors, adding / Enabling extra FindBug rules, How to do it..., How it works...

    - used, for detecting security defects / Finding security defects with FindBugs, How to do it..., There's more...

- FindBugs Eclipse plugin
    - about / The FindBugs Eclipse plugin
    - URL / The FindBugs Eclipse plugin

- FitNesse
    - about / Introduction, Testing with FitNesse
    - URL / Testing with FitNesse
    - testing with / Testing with FitNesse, How to do it..., How it works..., There's more...
    - HtmlUnit fixtures, activating / Activating FitNesse HtmlUnit fixtures, How to do it..., How it works..., There's more...

- functional testing
    - performing, with JMeter assertions / Functional testing using JMeter assertions, How to do it..., How it works...

- fuzzer
    - about / Finding 500 errors and XSS attacks in Jenkins through fuzzing

G

- Geb
    - URL / How it works...

- generated data
    - reacting, with groovy-postbuild plugin / Reacting to generated data with the groovy-postbuild plugin, How to do it..., How it works..., There's more...

- GMaven plugin
    - using / Running Groovy scripts through Maven, How to do it..., How it works...
    - URL / Running Groovy scripts through Maven
    - warnings, tracking / Keeping track of warnings
    - source code / Where's my source?
    - Maven phases / Maven phases

- Google Analytics account

    - creating / Knowing your audience with Google Analytics, How it works...

- Google Analytics plugin
    - installing / Getting ready

- Google Calendar plugin
    - using, for mobile presentation / Mobile presentation using Google Calendar, How to do it..., How it works..., There's more...
    - installing / Getting ready

- Green Balls plugin
    - URL / Getting ready
    - about / How it works...

- Groovy
    - used, for global modifications of jobs / Global modifications of jobs with Groovy, How to do it..., There's more...
    - used, for running Ant in Maven / Running Ant through Groovy in Maven, How to do it..., There's more...

- groovy-postbuild plugin
    - about / Reacting to generated data with the groovy-postbuild plugin
    - generated data, reacting to / Reacting to generated data with the groovy-postbuild plugin, How to do it..., How it works..., There's more...

- Groovy hook scripts
    - placing, on startup / Groovy hook scripts and triggering events on startup, How to do it..., How it works..., There's more...

- Groovy scripts
    - running, through Maven / Running Groovy scripts through Maven, How to do it..., How it works...

- group2.pl script / How it works...
- GUI samples plugin
    - viewing / Looking at the GUI samples plugin, How to do it..., How it works..., There's more...

H

- help.start() command / Quickly getting help
- home page
  - generating / Generating a home page, How to do it..., How it works..., There's more...
- HTML5 browsers
  - custom sound, creating with / Making noise with HTML5 browsers, How to do it..., There's more...
- HTML publisher plugin
  - URL / Getting ready
- HTML reports
  - creating / Creating HTML reports, How to do it..., There's more...
- HtmlUnit fixtures
  - activating / Activating FitNesse HtmlUnit fixtures
- HTML validity
  - verifying, Unicon Validation plugin used / Verifying HTML validity, How to do it..., There's more...
- Hudson2Go Lite
  - URL / How to do it...
- Hudson apps / Android 1.6 and Hudson apps
- Hudson Helper
  - URL / How to do it...

I

- information
  - exposing, through build descriptions / Exposing information through build descriptions, How to do it..., How it works...
- Information radiators

    - about / An extreme view for reception areas

- installation, Checkstyle / Getting ready
- installation, Cobertura / Getting ready, How to do it...
- installation, FindBugs

    - about / Getting ready

- installation, JavaNCSS

    - about / Getting ready

- installation, PMD / Getting ready
- installation, SLOCCount

    - about / Getting ready

- installation, Unicorn Validation plugin / Getting ready

J

- Jacoco

    - URL / There's more...
    - about / There's more...

- JASIG

    - URL / Getting ready

- java.lang.IllegalArgumentException / There's more...
- java.lang.NullPointerException / There's more...
- java.lang.NumberFormatException / There's more...
- java.lang.SecurityException / There's more...
- JavaMelody

    - URL / Monitoring via JavaMelody
    - about / Monitoring via JavaMelody, How it works...
    - used, for monitoring / Monitoring via JavaMelody, How to do it..., How it works...
    - used, for troubleshooting / Troubleshooting with JavaMelody – memory, Troubleshooting with JavaMelody – painful jobs

- JavaNCSS

    - about / Reporting with JavaNCSS
    - URL / Reporting with JavaNCSS
    - used, for creating reports / Reporting with JavaNCSS, How it works...

    - installing / Getting ready

- Java puzzlers
    - URL / There's more...

- JavaScript library frameworks / Included JavaScript library frameworks
- JavaServer Pages (JSP)
    - about / Failing Jenkins jobs based on JSP syntax errors
    - URL / Failing Jenkins jobs based on JSP syntax errors
    - compiling / Failing Jenkins jobs based on JSP syntax errors, How to do it..., How it works...
    - different server types / Different server types
    - Eclipse templates / Eclipse templates for JSP pages

- JCaptcha
    - used, for avoiding sign-up bots / Avoiding sign-up bots with JCaptcha, How to do it...

- JDepend
    - about / There's more...
    - URL / There's more...

- Jelly
    - URL / Introduction

/ Introduction
- Jelly framework
    - URL / How it works...

- Jenkins
    - about / Introduction, Introduction, Jenkins is an enterprise-friendly agnostic
    - test instance, using / Using a test Jenkins instance, Getting ready, How to do it..., How it works...
    - security / Introduction
    - pom.xml template / A pom.xml template
    - alternative code metrics, plotting / Plotting alternative code metrics in Jenkins, Getting ready, How to do it...
    - skinning, with themes plugin / Skinning Jenkins with the simple themes plugin, How to do it..., How it works...
    - skinning, with WAR overlay / Skinning and provisioning Jenkins using a WAR overlay, How to do it..., How it works...

    - integrating, with SonarQube / Integrating Jenkins with SonarQube, How to do it..., How it works...
    - personalizing / Personalizing Jenkins, How to do it..., How it works...

- Jenkins bug [Jenkins-22252]
    - URL / Getting ready

- Jenkins CLI
    - about / Scripting the Jenkins CLI
    - URL / Scripting the Jenkins CLI
    - scripting / Scripting the Jenkins CLI, Getting ready, How it works...
    - JAR file, downloading / Getting ready

- Jenkins configuration
    - modifying, from command line / Modifying the Jenkins configuration from the command line, How it works...
    - security, turning off / Turning off security
    - JavaDoc, searching for custom plugin extensions / Finding JavaDoc for custom plugin extensions
    - garbage, effects / The effects of adding garbage

- Jenkins Job configuration
    - multiple approaches / Multiple approaches and antipatterns

- JenkinsMobi
    - URL / How to do it...

- JenkinsMobi application / How it works...
- Jenkins Mobile Monitor
    - URL / How to do it...

- Jenkins Mood widget
    - URL / How to do it...

- Jenkins performance plugin
    - URL / Getting ready

- Jenkins plugins
    - about / Introduction

- Jenkins Publish Over SSH plugin
  - URL / Introduction
- Jenkins sounds plugin
  - installing / Getting ready
- Jenkins user
  - viewing, through Groovy / Looking at the Jenkins user through Groovy, How to do it..., How it works..., There's more...
- Jenkins XML API
  - about / There's more...
  - URL / There's more...
- Jenkins xUnit plugin / Getting ready
- Jetty
  - configuring, for integration tests / Configuring Jetty for integration tests, How to do it..., How it works...
- Jlint
  - URL / How it works...
- JMeter
  - about / Introduction
  - URL / Creating JMeter test plans
- JMeter assertions
  - used, for functional testing / Functional testing using JMeter assertions, How to do it..., How it works...
- JMeter performance metrics
  - reporting / Reporting JMeter performance metrics, How to do it..., There's more...
- JMeter test plans
  - creating / Creating JMeter test plans, How to do it..., How it works...
- JobConfigHistory plugin / A complementary plugin – JobConfigHistory
- Job Exporter plugin

    - about / Exporting data
    - URL / Getting ready
    - working / How it works...

- jobs
    - global modifications, with Groovy / Global modifications of jobs with Groovy, How to do it..., How it works...
    - testing / Testing and then promoting builds
    - promoting / Testing and then promoting builds, How to do it..., There's more...

- Jobs
    - triggering remotely, through Jenkins API / Remotely triggering jobs through the Jenkins API, How to do it..., How it works...
    - running, from Maven / Running jobs from within Maven
    - generating remotely / Remotely generating jobs

- jQuery plugin
    - URL / Included JavaScript library frameworks

- JSGames plugin
    - pinning / Fun with pinning JSGames, How to do it...
    - URL / Getting ready

- JUnit
    - URL / There's more...

K

- Keep It Simple Stupid (KISS) / Introduction
- Keep It Simple Stupid (KISS) principle / Considering test automation as a software project
- Kickstarter
    - URL / Visibly rewarding successful developers

L

- LDAP plugin
    - configuring / Configuring the LDAP plugin, How it works...

    - misconfiguration, versus bad credentials / The difference between misconfiguration and bad credentials
    - searching / Searching

- LDIF
    - about / Installing OpenLDAP, How it works...

- license violations
    - searching, with Rats / Looking at license violations with Rat, How it works..., There's more...
    - reviewing, from Maven / Reviewing license violations from within Maven, How to do it..., How it works...

- Lightweight Directory Access Protocol (LDAP)
    - about / Installing OpenLDAP

- listeners / Creating JMeter test plans
- ListView plugins
    - reviewing / Reviewing three ListView plugins, Getting ready, How to do it..., How it works...
    - Last Figure Version Column plugin / Getting ready
    - Last Success Description Column plugin / Getting ready
    - Last Success Version Column plugin / Getting ready
    - Extra Columns plugin / There's more...
    - Cron Column plugin / There's more...
    - Emma Coverage plugin / There's more...
    - Progress Bar plugin / There's more...

- log-parser plugin
    - installing / Getting ready, How to do it...
    - configuring / How it works...

- login2.pl script / How it works...

M

- manager.addShortText / How it works...
- manager.addWarningBadge(title) / How it works...
- manager.build.logFile / How it works...
- manager.getMatcher / How it works...
- markup plugin / How it works...
- Mask Passwords plugin
    - installing / Getting ready

    - about / How it works...

- Maven
    - about / Introduction
    - Ant, running through Groovy / Running Ant through Groovy in Maven, How to do it..., There's more...

- maven-jetty-jspc-plugin
    - URL / Failing Jenkins jobs based on JSP syntax errors

- maven-soapui plugin
    - URL / How it works...

- Maven 2
    - about / Maven changes

- Maven 3 / There's more...
    - about / Maven 2 and 3 pitfalls

- Maven dashboard
    - about / The Xradar and Maven dashboards
    - URL / The Xradar and Maven dashboards

- Maven site generation
    - about / Adaptive site generation, Getting ready, How to do it..., How it works..., Maven 2 and 3 pitfalls

- mobile apps, for Android / Mobile apps for Android and iOS, How to do it...
- mobile apps, for iOS / Mobile apps for Android and iOS, How to do it...
- mobile presentation
    - creating, with Google Calendar / Mobile presentation using Google Calendar, How to do it..., How it works..., There's more...

- monitoring
    - performing, via JavaMelody / Getting ready, How to do it..., How it works...
    - about / There's more...

- Monitoring plugin

    - installing / Monitoring via JavaMelody, How to do it...

- MOOCs
    - URL / For more information
- multiple Jenkins nodes
    - creating / Creating multiple Jenkins nodes, How to do it..., How it works..., There's more...
- Multi slave config plugin
    - URL / Getting ready

N

- Nexus
    - URL / There's more...
- Nginx
    - about / Installing Nginx
    - advantages / Installing Nginx
    - installing / Getting ready, How to do it...
    - working / How it works...
    - logfiles, naming / Naming logfiles
    - configuration, backing up / Backing up configuration
    - configuring, as reverse proxy / Configuring Nginx as a reverse proxy, How to do it..., How it works...
    - complex configuration, testing / Testing complex configuration
    - SSL, offloading / Offloading SSL
- Nikto
    - URL / More tools of the trade
- Nmap
    - URL / More tools of the trade
- nonce / How it works...
- NonCommenting Source Statements (NCSS) / Reporting with JavaNCSS
- no tests pattern / How it works...

O

- objectClass
  - about / There's more...
- open hub
  - URL / How it works...
- OpenLDAP
  - installing / Installing OpenLDAP, How to do it..., How it works...
  - administering / Administering OpenLDAP, How to do it..., How it works...
- OWASP
  - about / Testing for OWASP's top 10 security issues
  - URL / Testing for OWASP's top 10 security issues
  - security issues, testing / Testing for OWASP's top 10 security issues, Getting ready, How to do it...
  - storefront / Testing for OWASP's top 10 security issues
  - target practice, with Webgoat / Target practice with Webgoat
- OWASP Dependency-Check plugin
  - about / Exploring the OWASP Dependency-Check plugin
  - installing / Getting ready, How to do it..., How it works..., There's more...
- OWASP project
  - URL / There's more...
- OWASP_TOP10 profile / How it works...

P

- Perl script
  - about / How it works...
  - URL / There's more...
- plot plugin
  - about / How it works..., There's more...
- Pluggable Authentication Modules (PAM) / There's more...
- plugins

    - information / Introduction

- plugin tutorial
    - URL / Introduction

- PMD
    - about / Introduction
    - installing / Getting ready
    - URL / How it works...
    - Don't Repeat Yourself (DRY) principle / The Don't Repeat Yourself principle
    - URL, for installation / Getting ready
    - URL, for logging rules / How it works...
    - JSP files, scanning / How it works...

- PMD rulesets
    - activating / Activating more PMD rulesets, How to do it..., How it works...
    - basic / Activating more PMD rulesets
    - unused code / Activating more PMD rulesets
    - imports / Activating more PMD rulesets
    - throttling down / Throttling down PMD rulesets
    - URL / Throttling down PMD rulesets

- pom.xml file
    - used, for checking code style with Checkstyle / Checking code style using an external pom.xml file, How to do it..., How it works..., There's more...

- pom.xml template
    - about / A pom.xml template

- profile2 tool
    - about / How it works...

- project-based matrix strategy
    - reviewing, via custom group script / Reviewing project-based matrix tactics via a custom group script, How to do it..., How it works...

- Promoted Builds plugin
    - URL / Getting ready

    - about / How it works...

- promotion

    - about / Testing and then promoting builds

Q

- QJPro

    - URL / How it works...

- quality assurance

    - failing early / Fail early
    - data-driven testing / Data-driven testing
    - learning from history / Learning from history
    - test automation, as software project / Considering test automation as a software project
    - results, visualizing / Visualize, visualize, visualize
    - conventions / Conventions are good
    - work, offsetting / Offsetting work to Jenkins nodes
    - QA/integration servers, starving / Starving QA/integration servers
    - human bottlenecks, avoiding / Avoiding human bottlenecks
    - groupthink, avoiding / Avoiding groupthink
    - training / Training and community
    - community / Training and community
    - developers, rewarding / Visibly rewarding successful developers
    - stability / Stability and code maintenance
    - code maintenance / Stability and code maintenance
    - resources / Resources on quality assurance
    - points to consider / And there's always more
    - final comments / Final comments

- quality metrics

    - reference link / Introduction

R

- repository managers

    - Artifactory / There's more...
    - Apache Archiva / There's more...
    - Nexus / There's more...

    - advantages / There's more...

- restoring
    - performing / How to do it..., How it works...
    - about / How it works...

- retention policy / There's more...
- R language
    - resources / For more information

- robots.txt
    - URL / Search engines and robots.txt

- Roles Validation script / How it works...
- RootAction plugin
    - creating / Creating a RootAction plugin, How to do it..., How it works..., There's more...

- R plugin
    - about / Simplifying powerful visualizations using the R plugin
    - powerful visualizations, simplifying with / Simplifying powerful visualizations using the R plugin, How to do it..., How it works...
    - URL / Simplifying powerful visualizations using the R plugin
    - used, for analyzing project data / Analyzing project data with the R plugin, How to do it..., How it works..., There's more...

- R programming
    - URL / For more information

- RSS credentials / RSS credentials
- RSS feeds
    - extracting, FireFox used / Keeping in contact with Jenkins through Firefox, How to do it..., How it works...

- RStudio
    - URL / RStudio or StatET

- R style guide
    - URL / For more information

S

- Sakai
  - URL / How it works...
- Sakai CLE
  - about / Enabling Sakai web services, There's more...
  - download link / Getting ready
- Sakai Learning Management System
  - URL / How it works...
- Sakai project
  - URL / How to do it...
- Sakai web services
  - enabling / Getting ready, How to do it..., How it works...
- samplers / Creating JMeter test plans
- SCP task
  - URL / How it works...
- screen space
  - saving, with Dashboard View plugin / Saving screen space with the Dashboard View plugin, How it works...
- Script console / There's more...
- Scriptler
  - URL / Getting ready
- Scriptler plugin
  - installing / Getting ready
- Script Realm authentication
  - used, for provisioning / Using Script Realm authentication for provisioning, How to do it..., How it works...
- scripts

    - managing, with Scriptler plugin / Keeping track of script glue, How it works...

- search engines / Search engines and robots.txt
- security, Jenkins

    - about / Introduction
    - OWASP security issues, testing / Testing for OWASP's top 10 security issues, Getting ready, How to do it...
    - XSS attacks, identifying with fuzzer / Finding 500 errors and XSS attacks in Jenkins through fuzzing, How to do it..., How it works...
    - 500 errors, identifying with fuzzer / Finding 500 errors and XSS attacks in Jenkins through fuzzing, How to do it..., How it works...
    - improving, via modest configuration changes / Improving security via small configuration changes, How to do it..., How it works...
    - sign-up bots, avoiding with JCaptcha / Avoiding sign-up bots with JCaptcha, How to do it..., There's more...
    - Jenkins user, viewing through Groovy / Looking at the Jenkins user through Groovy, How to do it..., How it works..., There's more...
    - Audit Trail plugin, working with / Working with the Audit Trail plugin, How it works...
    - OpenLDAP, installing / Installing OpenLDAP, How to do it..., How it works...
    - Script Realm authentication, using for provisioning / Using Script Realm authentication for provisioning, How to do it..., How it works...
    - project-based matrix strategy, reviewing via custom group script / Reviewing project-based matrix tactics via a custom group script, Getting ready, How to do it..., How it works...
    - OpenLDAP, administering / Administering OpenLDAP, How to do it..., How it works...
    - LDAP plugin, configuring / Configuring the LDAP plugin, How it works...
    - CAS server, installing / Installing a CAS server, How to do it..., How it works...
    - SSO, enabling / Enabling SSO in Jenkins, How to do it..., How it works...
    - OWASP Dependency-Check plugin, installing / Exploring the OWASP Dependency-Check plugin, How to do it..., How it works..., There's more...

- security issues, OWASP

    - A2-Cross-site Scripting (XSS) / Testing for OWASP's top 10 security issues
    - A6-Security Misconfiguration / Testing for OWASP's top 10 security issues
    - A7-Insecure Cryptographic Storage / Testing for OWASP's top 10 security issues
    - A9-Insufficient Transport Layer Protection / Testing for OWASP's top 10 security issues

- Selenium / Introduction
- Selenium HTML report plugin
    - URL / Getting ready

- Selenium IDE tests
    - about / Running Selenium IDE tests
    - running / Getting ready, How to do it..., How it works..., There's more...

- Selenium WebDriver
    - failsafe integration tests, triggering / Triggering failsafe integration tests with Selenium WebDriver, How to do it...

- shortened URL
    - URL / How it works...

- sign-up bots
    - avoiding, JCaptcha used / Avoiding sign-up bots with JCaptcha, How to do it...

- Single sign-on (SSO) / Introduction
- Skipfish
    - URL / More tools of the trade

- slave nodes
    - custom setup scripts, running / Custom setup scripts for slave nodes, How to do it..., How it works...

- Slave Setup plugin
    - URL / Getting ready

- Slide me

    - URL / Virtualbox and the Android x86 project

- SLOCCount
    - about / Estimating the value of your project through sloccount
    - URL / Estimating the value of your project through sloccount
    - used, for estimating project value / Estimating the value of your project through sloccount, How to do it..., How it works...
    - installing / Getting ready
    - URL, for installation / Getting ready

- snapshots
    - about / Snapshots

- SoapUI
    - about / Introduction, Writing test plans with SoapUI, How it works...
    - URL / Writing test plans with SoapUI
    - test plans, writing with / Writing test plans with SoapUI, How to do it...

- SoapUI test results
    - reporting / Reporting SoapUI test results, How to do it..., How it works...

- Sonar plugin
    - URL, for installation / Getting ready

- SonarQube
    - about / Integrating Jenkins with SonarQube
    - Jenkins, integrating / Integrating Jenkins with SonarQube, How to do it..., How it works...
    - installing / Getting ready
    - URL, for installation / Getting ready
    - extra plugins, adding / SonarQube plugins

/ Test frameworks and commercial choices are increasing
- SourceForge
    - URL / Getting ready

- SSH Public Keys section / There's more...
- SSO

    - enabling / Enabling SSO in Jenkins, How to do it..., How it works...

- Stapler
    - URL / There's more...

/ Introduction
- Startup Trigger plugin
    - URL / Getting ready
    - working / How it works...

- StatET plugin
    - URL / RStudio or StatET

- static code review / Static code review, tainting, and untainting
- static code review tools
    - about / Test frameworks and commercial choices are increasing

- stringi library
    - about / There's more...
    - URL / There's more...

- Swatch
    - about / Swatch

T

- Test Driven Development (TDD)
    - about / Introduction

- test instance
    - setting up / Using a test Jenkins instance
    - advantages / Using a test Jenkins instance
    - using / Getting ready, How to do it..., How it works...

- TestNG framework / Introduction
- test plans
    - writing, with SoapUI / Writing test plans with SoapUI, Getting ready, How to do it...

- themes plugin
  - about / Skinning Jenkins with the simple themes plugin
  - installing / Getting ready
- thinBackup plugin / How it works...
- tm package
  - about / There's more...
  - URL / There's more...
- token-macro plugin
  - URL / There's more...
- Tomcat 7
  - URL / Getting ready
- troubleshooting
  - JavaMelody used / Troubleshooting with JavaMelody – memory, Troubleshooting with JavaMelody – painful jobs
- two-minute tutorials
  - URL / For more information

U

- Ubuntu image
  - URL / Using a test Jenkins instance
- Unicorn Validation plugin
  - used, for verifying HTML validity / Verifying HTML validity, How to do it..., There's more...
  - URL / Getting ready
  - installing / Getting ready
- URLTrigger plugin / How it works...

V

- violations plugin
  - URL / Introduction

    - about / Alternative aggregator – the violations plugin

- VirtualBox
    - URL / Using a test Jenkins instance
- Virtualbox / Virtualbox and the Android x86 project

W

- w3af
    - about / Getting ready, More tools of the trade
    - installing / How to do it...
- wapiti
    - about / How it works...
- Wapiti
    - URL / More tools of the trade
- WAR file
    - deploying, from Jenkins to Tomcat / Deploying a WAR file from Jenkins to Tomcat, How to do it..., How it works...
- wget
    - URL / Getting ready
- Wiki books on R
    - URL / For more information
- WSDL
    - URL / How it works...

X

- X-SSH-Endpoint / There's more...
- Xpath
    - about / How it works...
    - URL / How it works...
- Xradar dashboard

    - about / The Xradar and Maven dashboards
    - URL / The Xradar and Maven dashboards

- XSS attacks
    - about / Finding 500 errors and XSS attacks in Jenkins through fuzzing
    - identifying, fuzzer used / Getting ready, How to do it...
    - URL / How it works...

- XStream
    - URL / How it works..., Introduction

/ Introduction

- XTrigger plugin
    - URL / Getting ready

- xUnit plugin
    - about / There's more...
    - URL / There's more...

- xUnit standard
    - reference link / Introduction

- Xvfb
    - URL / Running Selenium IDE tests

Y

- Yale CAS
    - URL / Installing a CAS server
    - about / Installing a CAS server
    - downloading / Getting ready
    - backend authentication / Backend authentication
    - alternative installation, ESUP CAS used / An alternative installation recipe using ESUP CAS
    - LDAP SSL, trusting / Trusting LDAP SSL

- YUI library
    - URL / Included JavaScript library frameworks