Extending Jenkins

Jenkins CI is the leading open source continuous integration server. It is written in Java and has a wealth of plugins to support the building and testing of virtually any project. Jenkins supports multiple software configuration management tools, such as Git, Subversion, and Mercurial.

This book explores and explains the many extension points and customizations that Jenkins offers its users, and teaches you how to develop your own Jenkins extensions and plugins.

First, you will learn how to adapt Jenkins and leverage its abilities to empower DevOps, continuous integration, continuous deployment, and Agile projects. Next, you will find out how to reduce the cost of modern software development, increase the quality of deliveries, and thereby reduce the time to market. We will also teach you how to create your own custom plugins using extension points.

Finally, we will show you how to combine everything you learned over the course of the book into one real-world scenario.

Who this book is written for

This book is aimed primarily at developers and administrators who are interested in taking their interaction and usage of Jenkins to the next level.

What you will learn from this book

- Retrieve and act upon Jenkins information in real time
- Find out how to interact with Jenkins through a variety of IDEs
- Develop your own form and input validation and customization
- Explore how extension points work, and develop your own Jenkins plugin
- See how to use the Jenkins API and command-line interface
- Get to know how to remotely update your Jenkins configuration
- Design and develop your own Information Radiator
- Discover how Jenkins customization can help improve quality and reduce costs

Extending Jenkins

Get a complete walkthrough of the many interfaces available in Jenkins with the help of real-world examples to take you to the next level.

Donald Simpson


Free Sample

Prices do not include local sales tax or VAT

Download from Packt at http://www.packtpub.com

Get in touch with us:

Email: contact@packtpub.com

www.packtpub.com
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 8 'Testing and Debugging Jenkins Plugins'
- A synopsis of the book’s content
- More information on Extending Jenkins
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 the adoption of process automation and Agile methodologies.

Starting out as a Java developer, Donald's interest in application servers, networking, and automation led him to a career as a build engineer. He remains highly technical and hands-on and enjoys learning about new technologies and finding ways to automate and improve manual processes.

He can be reached at www.donaldsimpson.co.uk.
Preface

Jenkins provides many interfaces and extension points to enable users to customize and extend its functionality. In this book, we will explore these interfaces in depth and provide practical real-world examples that will take your usage of Jenkins to the next level.

In this book, you will learn how to develop and test your own Jenkins plugin, find out how to set up fully automated build pipelines and development processes, discover how to interact with the API and CLI, and how to enhance the user interface.

What this book covers

Chapter 1, Preparatory Steps, will cover the initial setup steps—getting your development environment set up, an overview of Jenkins and some options to install and run it as well as extend the basic setup. We will also review the principles of Continuous Integration, which are explored in greater detail later.

Chapter 2, Automating the Jenkins UI, will discuss how several common issues and bottlenecks may be alleviated through the automation and adaptation of the Jenkins frontend. Here, we will look at four fairly typical use cases, identify the root cause of the issues, and propose some possible improvements that can be made through the alteration and automation of the GUI.

Chapter 3, Jenkins and the IDE, builds on the Continuous Integration principals that we looked at earlier and provides an introduction to the Mylyn project.

It then details how to set up a process that enables developers to interact with Jenkins directly from within their IDE. A selection of examples covers Eclipse, NetBeans, and IntelliJ.
Chapter 4, *The API and the CLI*, illustrates how we can automate and extend Jenkins through its API and CLI. In this chapter, we will illustrate how to use these interfaces by working through the high-level "building blocks" of an example "Information Radiator" project.

This chapter will explain how to create a dynamic application that consumes information from Jenkins via its exposed interfaces.

We will also review other ways in which you could extend Jenkins via the CLI—by kicking off jobs and making other changes to Jenkins automatically and remotely.

Chapter 5, *Extension Points*, introduces many important concepts that provide a foundation for the Jenkins Extension points topics in the subsequent chapters. We will run through Java interfaces, Design by Contract, abstract classes, and Singletons. We will then take a look at how these patterns are used in the real world when we define our own Extension Point in Jenkins.

Chapter 6, *Developing Your Own Jenkins Plugin*, will combine the skills, concepts, and tools from the preceding chapters to build our first Jenkins plugin.

We will take a look at Maven and learn how to set it up and use it for Jenkins plugin development. We will then create our first Jenkins plugin, learn how to install it locally, and then learn how to quickly make, build, and deploy subsequent changes using Maven.

Chapter 7, *Extending Jenkins Plugins*, makes use of a simple plugin with the "Hello world" functionality we created in the previous chapter to keep the focus on getting to grips with the processes and tools. This chapter takes a look at the best way to get started with adding your own implementations. You will learn how to reuse existing code and functionality and understand how and where to find them.

After taking a look at some existing plugins and using those as examples, we will then take a detailed look at some of the additional resources and technologies you could take advantage of in your own projects.

Chapter 8, *Testing and Debugging Jenkins Plugins*, explains how to test and debug your own code and how to apply the same approach to existing plugins for troubleshooting.

It covers running tests with Maven, examines some existing tests from a popular plugin, and uses these to demonstrate how you can adapt these approaches to suit your own projects.

We will also take a look at debugging live code through the IDE and show how to integrate these useful functions into popular development IDEs. The final section of this chapter will introduce the inbuilt Jenkins Logger Console.
Chapter 9, *Putting Things Together*, takes a look at how Jenkins can be extended to work with other technologies and languages. We will start off with a look at the Jenkins Scripting console and see how useful it can be when combined with some Groovy scripting by providing some examples. We will then discuss developing applications using Groovy, Grails, and Gradle as possible alternatives to Maven and Java. The final part of this chapter covers Jenkins and Docker and then discusses how to set up build and deployment pipelines for iOS and Android development.
8

Testing and Debugging Jenkins Plugins

In this chapter, we will take a look at the testing and debugging of Jenkins plugins. We will explore several popular options and approaches that are currently available, and we will review the benefits and suitability of each approach.

Testing Jenkins plugins is reasonably straightforward if you are happy to simply run standard Java Unit tests, but if you wish to test and mimic interactions via the user interface, testing can become a little bit more involved. We will start off with a simple example and then look at some of the approaches and tools you may want to investigate further for more complex scenarios.

Being able to debug a Jenkins plugin is a valuable addition to your development skills—it can help you understand what is going on with your own plugin while you are developing it, and it can also help you to resolve issues in other plugins or Jenkins itself.

In this chapter, we will take a look at the following topics:

- Testing: Under Testing, we'll cover the following topics:
  - Running tests for an existing project
  - Writing your own tests
  - Available tools
  - Techniques—HTML scraping, Mocking, and so on
Testing and Debugging Jenkins Plugins

• Debugging: Under Debugging, we'll cover the following topics:
  ○ Standard log files
  ○ Using the local Jenkins debug session
  ○ Connecting from an IDE
  ○ The `mvnDebug` command

Running tests with Maven

When we were exploring plugin development earlier, we learned where to find and how to fetch the source code for any given Jenkins plugin.

The full source code for most plugins can be quickly and easily downloaded from GitHub and then built on your local machine. In many cases, this also includes Unit tests, which are bundled with the source code and can be found in the expected (by Maven convention) location of `src/test`. Examining a selection of popular plugins would provide you with useful information and a great starting point to write your own test cases.

The Maven `test` target will execute all of the tests and produce a summary of the outcome by detailing all the usual statistics such as the number of tests run along with how many failures and errors there were and the number of skipped tests.

To demonstrate this process, we will take a look at the very popular Green Balls plugin, which simply replaces the standard blue balls in Jenkins with green ones.

This link explains why Jenkins has blue balls as default:
http://jenkins-ci.org/content/why-does-jenkins-have-blue-balls

The Green Balls plugin homepage links to this GitHub location, where you can download the source and configuration files in a zip file or clone it using the URL provided:
https://github.com/jenkinsci/greenballs-plugin

We're looking at this example plugin, as it contains a good variety of tests that cover the main topics and styles of testing—we will take a closer look at the contents shortly. Once you have the source code downloaded to your local machine, you should be able to kick off the tests by simply running the Maven `test` target:

`mvn test`
This target will then run through all the prerequisite setup steps before executing all the tests and then report on the outcome as follows:

Note that a single test can be run by specifying the name of the test, as shown here:

```
mvn test -Dtest=GreenBallIntegrationTest
```

This will result in one test being run, or you can use wildcards such as this:

```
mvn test -Dtest=*ilter*
```

The preceding code results in four tests being run in this case.

This approach could be used to categorize your tests into logical suites—integration tests, nightly tests, regression tests, or unit tests—whatever you like, simply by applying a consistent naming convention to your test classes and then setting up Jenkins jobs, or running Maven targets that will perform the corresponding actions, for example:

```
mvn test -Dtest=*Integration*
```

The Green Balls plugin contains two test classes: GreenBallFilterTest and GreenBallIntegrationTest, which illustrate two different approaches of plugin testing—taking a look through their source code should help you to see how you can develop your own tests.
GreenBallFilterTest performs some simple pattern matching tests to ensure that correct images are in place:

```java
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNull;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.fail;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.fail;

public class GreenBallFilterTest {
    private GreenBallFilter greenBallFilter;

    @Before
    public void setUp() {
        greenBallFilter = new GreenBallFilter();
    }

    @Test
    public void testMatch() {
        Matcher matcher = greenBallFilter.matcher("/mynamespace/myimage.png");
        assertTrue(matcher.find());
        assertEquals("myimage.png", matcher.group());
    }

    @Test
    public void testNoMatch() {
        Matcher matcher = greenBallFilter.matcher("/mynamespace/mynotherelement.png");
        assertFalse(matcher.find());
    }
}
```

GreenBallIntegrationTest, as shown in the following screenshot, extends HudsonTestCase and uses com.gargoylesoftware.htmlunit.WebResponse to test and interact directly with the deployed web components, asserting that they return the expected results:
This Jenkins page provides useful resources for further reading that would cater to more detailed and complex testing scenarios:

https://wiki.jenkins-ci.org/display/JENKINS/Unit+Test

This link covers topics such as Mocking, HTML scraping, submitting forms, JavaScript, and web page assertions.

**Debugging Jenkins**

The remainder of this chapter focuses on debugging in a number of different ways in order to help in further understanding the application and its behavior at run time.

The main focus is on using a local instance of Jenkins and an IDE to debug development sessions; however, it is still useful to know about the options available through the inbuilt logging options in Jenkins, which are sophisticated and highly customizable. These are often a good starting point for any kind of issue, so we will start with a quick overview of the options here before moving on to the type of debugging that you'll probably want to set up and use when developing your own code.

**Server debugging – a quick recap**

Jenkins uses the `java.util.logging` package for logging; the details of this can be found here:

https://docs.oracle.com/javase/7/docs/api/java/util/logging/package-summary.html

The Jenkins documentation on logging is available here:

https://wiki.jenkins-ci.org/display/JENKINS/Logging

This page explains how to go about setting up your own custom log recorders—this can be very useful to separate and filter all the log output to help in finding what you are interested in, as *everything* is often piped to the default log, which can make analyzing difficult.
Testing and Debugging Jenkins Plugins

The Jenkins system log can be checked out using the user interface at Manage Jenkins | System Log | All Jenkins Logs, and there are also links to the RSS feeds available at the bottom of the page:

These can help identify and filter the different types of events within the system.

For issues with slave nodes, there are log files available in the following location:

`~/.jenkins/logs/slaves/{slavename}`.

For job issues, historic log files are kept at `~/.jenkins/jobs/{jobname}/builds/{jobnumber}`.

You can also start Jenkins at a specific logging level by adding an additional `-D` argument to your startup process:

```
-Djava.util.logging.loglevel={level}
```

Here, `level` is one of the following:

- SEVERE (highest value)
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST (lowest value)

The Off and All levels are also available—see this page for further details and options:

http://docs.oracle.com/javase/7/docs/api/java/util/logging/Level.html
Debugging with IntelliJ

To debug from within IntelliJ, point IntelliJ to the pom.xml file of the project and then select the option from the Run menu to create a new Run/Debug configuration. This should lead you to a screen similar to this:

![Run/Debug Configurations](image)

IntelliJ will have already parsed the POM file and will be aware of the available targets it contains. As soon as you start typing, for example, hpi, you would be presented with a drop-down list of all matching options to select from.

Select and run the required target (hpi:run again in this case) from the dropdown and then click on **Debug**.

You should see the familiar Jenkins startup process in the console and then be able to connect to a local debug session at:

http://localhost:8080/jenkins
Add a debug point to the code at the same place where we made our "Hello World" text change previously (double-click on the left margin of the line that says `hello world...`). and then run the Jenkins job. This should run up to the break point you have set and produce this:
You can then use the debug arrows and buttons to drive through the debug process:

These allow you to step in to, over, or out of the current debug point, and you should be able to inspect the listed variables that are being updated to reflect the live state of the application being debugged.

For more information on debugging with IntelliJ, see this link:
https://www.jetbrains.com/idea/help/stepping-through-the-program.html

**Debugging with Eclipse**

Debugging with Eclipse is very similar to the process described for IntelliJ previously.

To set your breakpoint, double-click on the left-hand side margin in the code window, like this:
Next, right-click on the POM file in your Eclipse project and select **Debug as...** and the following window appears:

Specify the `hpi:run` target and then click on **Debug**: Jenkins should start up as usual in the Eclipse console window.

As before, point your browser to `http://localhost:8080/jenkins` and then create or run a job that hits the breakpoint you set earlier—when this code/point is reached, Jenkins will freeze and the focus will switch to Eclipse, where you can inspect the current state of the variables and properties and navigate through the various debugging steps to drill further into issues or step over areas to see what changes and happens.

**mvnDebug**

The `mvnDebug` tool provides an alternative approach that may be of interest to you. To use this, run `mvnDebug hpi:run` in the command line.
This should start up Maven in debug mode and a listener on port 8000 of local host, like this:

Now switch to your IDE and connect a debug session to this port. For example, in Eclipse, select Run | Debug Configurations...

This should produce the following window from which you can select Remote Java Application. Check whether the host and the port match:
Next, select **Debug** to connect to the `mvnDebug` session you started in the console. At this point, the `hpi:run` target will start up (in the console) and run Jenkins in debug mode in Maven while connected to your chosen debugger—for example, Eclipse.

If you examine the `mvnDebug` executable, you will see that it simply sets `MAVEN_DEBUG_OPTS` before running the normal `mvn` binary, as follows:

```
MAVEN_DEBUG_OPTS="-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=8000"
```

```
echo Preparing to Execute Maven in Debug Mode
env MAVEN_OPTS="$MAVEN_OPTS $MAVEN_DEBUG_OPTS" $(dirname $0)/mvn "$@"
```

This reveals that it would be easy to specify a different port if you wish, or you could adjust this script to add any additional parameters or settings you may want to include.

## The Jenkins Logger Console

The final topic in this chapter is the **Logger Console** that is built in to the debug versions of Jenkins.

When you start up a local dev instance of Jenkins via Maven (whether through the command line or an IDE), you will notice the additional **Logger Console** box that is included on the left-hand side of the screen:
Expanding this box will reveal a live log output window, which you can customize in real time to adjust and filter in or out the types and severities of log items that you want to see or hide.

Keeping info selected provides a very verbose level of output, which includes information on mouseover events and other UI interactions. These can be very useful when debugging UI issues.

Unchecking the info box leaves just the warn and error messages. The log output can be managed by pausing and optionally clearing the output and adjusting the filters to suit your need. The following screenshot shows the Logger Console:

![Logger Console](image)

**Summary**

As you can see, there is a large range of options and approaches available for both testing and debugging within Jenkins. This chapter introduced some of the main tools and approaches that you may hopefully find useful for your own development processes.

Knowing how to test and debug your code and set up a productive development environment that suits your needs and preferences should improve the quality of your own development. It should also make things much easier further down the line, when you look at distributing your own plugin and are considering alternative development options. We will take a look at some alternative technologies and languages in the next chapter.
Where to buy this book

You can buy Extending Jenkins from the Packt Publishing website.

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

Click here for ordering and shipping details.