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 plugins 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

- 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

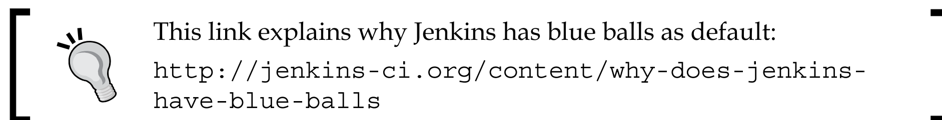
Running tests with Maven

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`

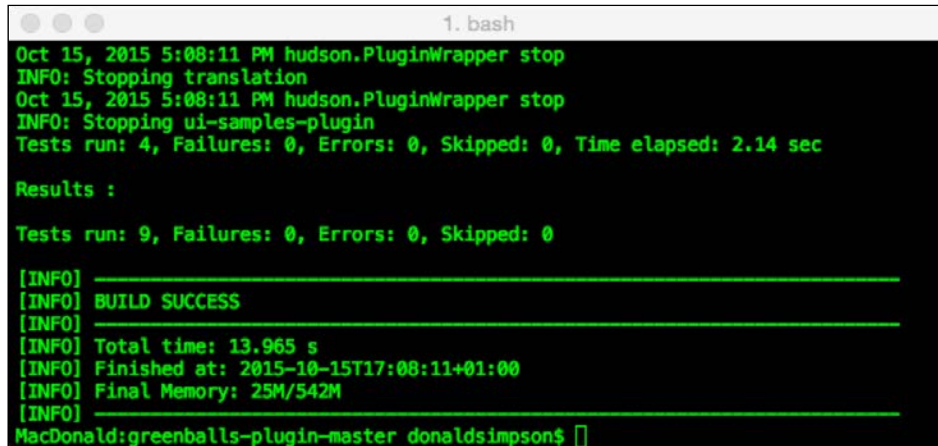
The Green Balls plugin homepage links to this GitHub location, where you can download the source and configuration files in a zip file or clone it using the URL provided:

`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:

```
GreenBallFilterTest.java ...on/plugins/greenballs) - VIM

import static org.hamcrest.CoreMatchers.equalTo;
import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertThat;

import java.util.regex.Matcher;

import org.junit.Before;
import org.junit.Test;

public class GreenBallFilterTest {

  GreenBallFilter greenBallFilter;

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

  @Test
  public void patternShouldMatch() {
    final Matcher m = greenBallFilter.patternBlue.matcher("/nocacheImages/48x48/blue.gif");
    assertThat(m.find(), is(true));
    assertThat(m.group(1), equalTo("48x48"));
    assertThat(m.group(2), equalTo(""));
    assertThat(m.group(3), equalTo("gif"));
  }

  @Test
  public void patternShouldMatchPNG() {
    final Matcher m = greenBallFilter.patternBlue.matcher("/nocacheImages/48x48/blue.png");
    assertThat(m.find(), is(true));
    assertThat(m.group(1), equalTo("48x48"));
    assertThat(m.group(2), equalTo(""));
    assertThat(m.group(3), equalTo("png"));
  }
```

`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:

```
GreenBallIntegrationTest....n/plugins/greenballs) - VIM1
package hudson.plugins.greenballs;

import java.math.BigInteger;
import java.net.URL;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Date;

import org.apache.commons.httpclient.util.DateUtil;
import org.jvnet.hudson.test.HudsonTestCase;

import com.gargoylesoftware.htmlunit.WebResponse;

/**
 *
 * @author Asgeir S. Nilsen
 */
public class GreenBallIntegrationTest extends HudsonTestCase {

    static String join(String first, String second) {
        if (first.endsWith("/"))
            first = first.substring(0, first.length() - 1);
        if (second.startsWith("/"))
            second = second.substring(1);
        return first + "/" + second;
    }

    static String hash(String algorithm, byte[] data) throws NoSuchAlgorithmException {
        byte[] hash = MessageDigest.getInstance(algorithm).digest(data);
        BigInteger bi = new BigInteger(1, hash);
        String result = bi.toString(16);
        if (result.length() % 2 != 0) {
            return "0" + result;
        }
        return result;
    }
```

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:

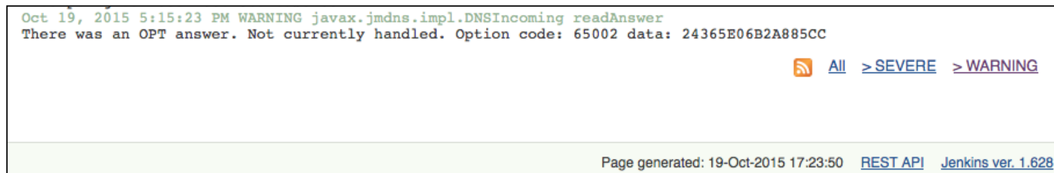
`https://docs.oracle.com/javase/7/docs/api/java/util/logging/package-summary.html`

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.

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:

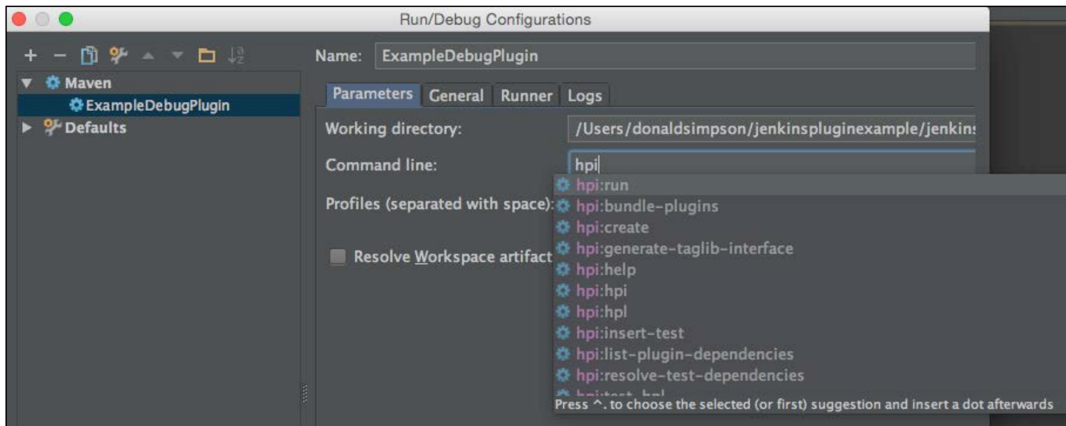
- `SEVERE (highest value)`
- `WARNING`
- `INFO`
- `CONFIG`
- `FINE`
- `FINER`
- `FINEST (lowest value)`

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:



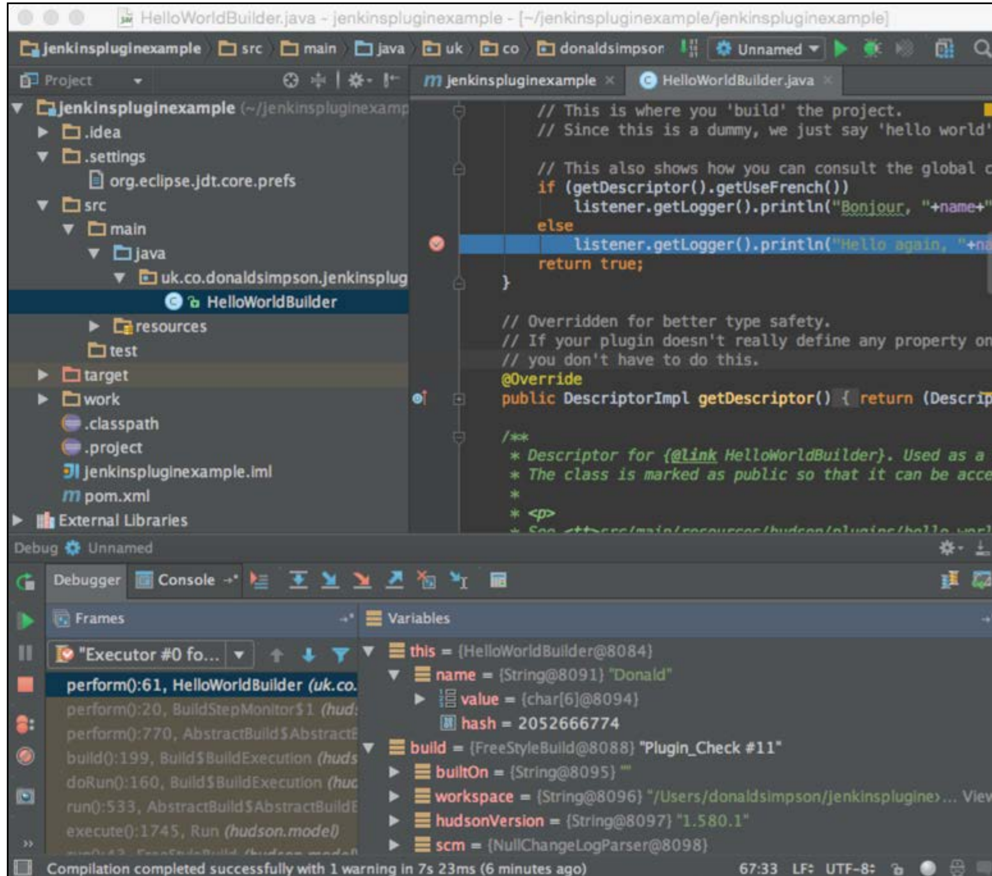
IntelliJ will have already parsed the POM file and will be aware of the available targets it contains. As soon as you start typing, for example, `hpi`, you would be presented with a drop-down list of all matching options to select from.

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

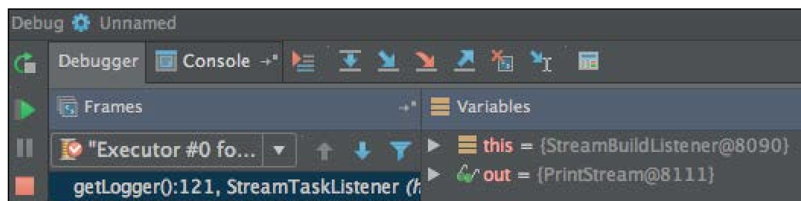
You should see the familiar Jenkins startup process in the console and then be able to connect to a local debug session at:

`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.

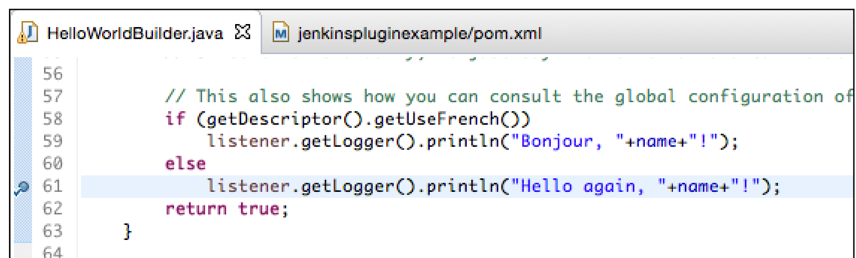
For more information on debugging with IntelliJ, see this link:

`https://www.jetbrains.com/idea/help/stepping-through-the-program.html`

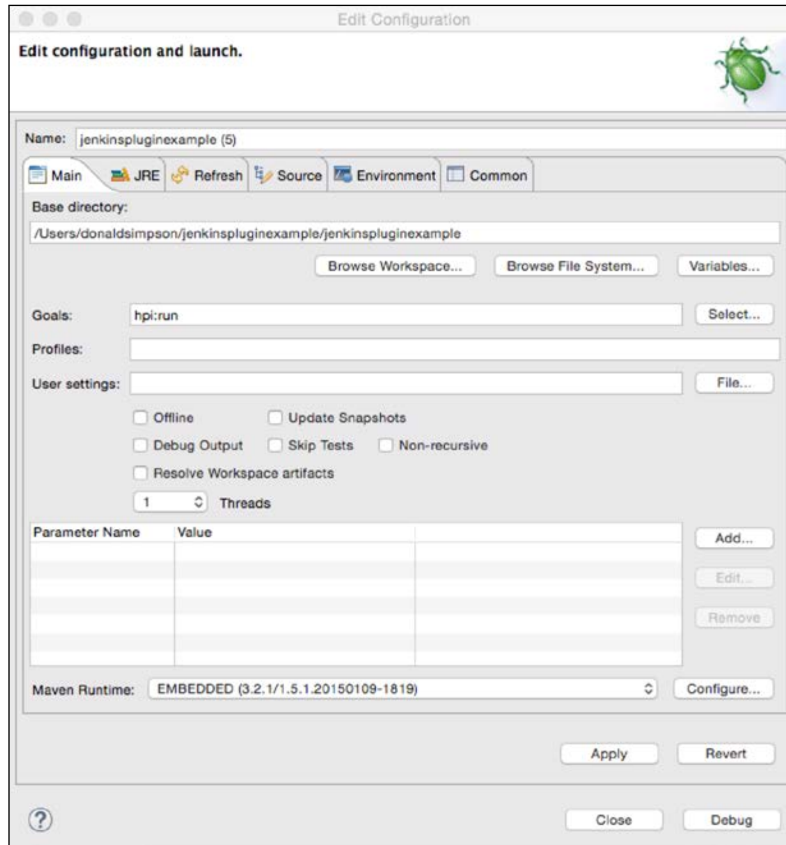
Debugging with Eclipse

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

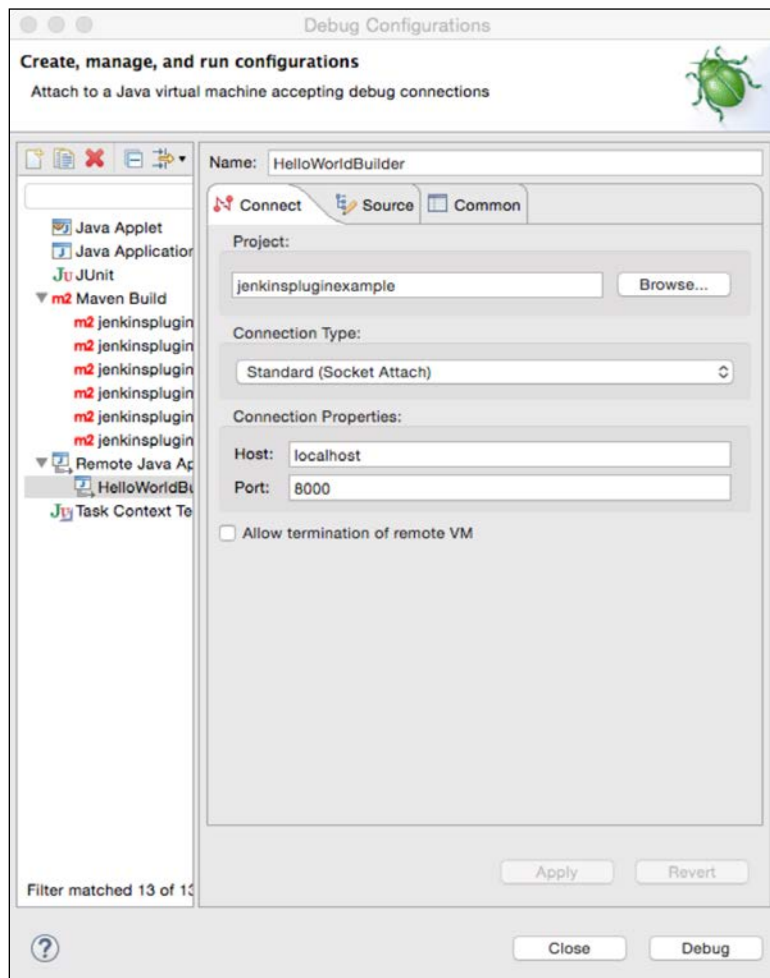
The `mvnDebug` tool provides an alternative approach that may be of interest to you. To use this, run `mvnDebug hpi:run` in the command line.

This should start up Maven in debug mode and a listener on port `8000` of local host, like this:

```
MacDonald:jenkinspluginexample donaldsimpson$ mvnDebug hpi:run
Preparing to Execute Maven in Debug Mode
Listening for transport dt_socket at address: 8000
```

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:

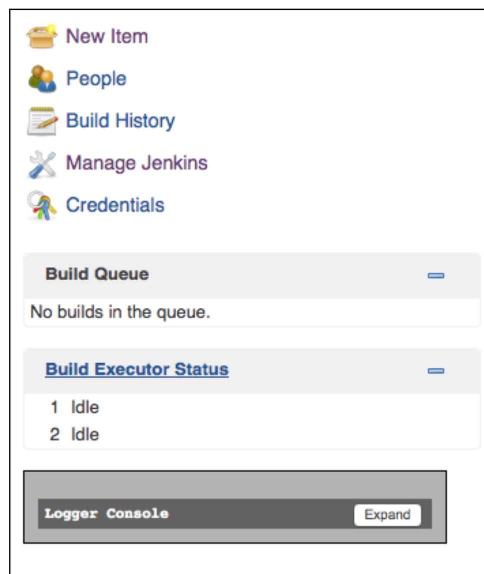
```
MAVEN_DEBUG_OPTS="-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspe
nd=y,address=8000"
echo Preparing to Execute Maven in Debug Mode
env MAVEN_OPTS="$MAVEN_OPTS $MAVEN_DEBUG_OPTS" $(dirname $0)/mvn "$@"
```

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.

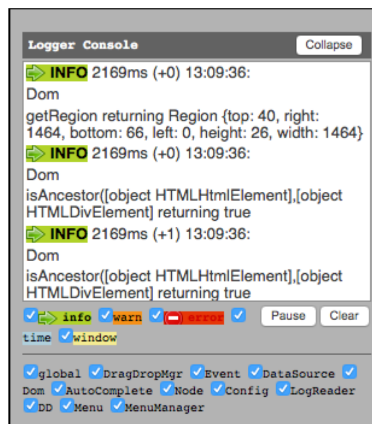
When you start up a local dev instance of Jenkins via Maven (whether through the command line or an IDE), you will notice the additional **Logger Console** box that is included on the left-hand side of the screen:



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**:



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.