

Getting Your Project Started with Groovy, Java and One-Step Builds

When it comes to completing a successful a project, everything a developer has to do *in addition to* writing the code is nearly as important as the code itself. I have seen more than one project get bogged down because of complicated builds and the lack of adequate testing. It's easy to download and run a "hello world" example, but it can often be challenging trying pull a new technology into a real project.

In this series of articles, I'll demonstrate how to write a one-step build script so that you can start using Groovy for non-trivial application development. Because it is common to mix Java and Groovy for production business applications, I will focus on getting both Java and Groovy source to compile in the same project. Since a one-step build should do more than compile source, I'll also show how easy it is to include unit tests and code coverage reports for your Java and Groovy source code.

The Value of One-Step Builds

Before you start writing a line of Java or Groovy, it's critical to have a one-step build in place. A one-step build is a program that can completely build your software application with a single command. It should run on any machine and anyone on the team should be able to execute the build. I attended the Chicago "No Fluff Just Stuff" show in October, 2004. At the show, I had the fortune of listening to Mike Clark talk about the value of one-step builds and immediately purchased his book "Pragmatic Project Automation". Jared Richardson and Will Gwaltney also discuss the topic in great detail in another Pragmatic Bookshelf book "Ship It". Both of these books go into great detail about the short term and long-term benefits have a one-step build in place. They also provide great insights into the value of using a version control system as well as having a unit test for every piece of application code.

Personally, I have also found having a one-step build in place *from day one* of coding to be very important for the success of a project. Everyone on the team from the very first day knows how to do a build and, just as importantly, they know when they've broke the build. This is especially important when introducing a new technology, like Groovy, into a project. The tension level on a project team usually rises until the project team becomes proficient in the new technology. Having unit tests in place that everyone can easily run significantly reduces the anxiety associated with checking in new code that utilizes a new technology. Using a one-step build that will compile all of the source and run all of the unit tests gives the developer confidence that the new code being checked into version control didn't accidentally break something else in the project. In a word, it greatly reduces stress caused by fear of the unknown.

Installing Groovy and Ant

Installing Groovy and Ant is very easy and requires a few simple steps. Groovy can be downloaded from <http://groovy.codehaus.org>. First, set the `GROOVY_HOME` environment variable to point to the directory where you unpacked the distribution archive. Second, add the `bin` directory to the path. For example, that Groovy was unpacked into `c:\groovy-1.0` on a Windows based system, you would set `GROOVY_HOME = c:\groovy-1.0` and you would add `%GROOVY_HOME%\bin` to your path. See <http://groovy.codehaus.org/Installing+Groovy> for information about installing on Mac OS X or Linux.

Ant can be downloaded from <http://ant.apache.org>. After unpacking the archive, set the `ANT_HOME` environment variable to your installation directory. Second, add the `bin` directory your path. If you've installed Ant into `c:\apache-ant-1.6.5` on a Windows system, you would set `ANT_HOME =`

[c:\apache-ant-1.6.5](http://ant.apache.org/manual/index.html) and add %ANT_HOME%\bin to your path. See <http://ant.apache.org/manual/index.html> for detailed information about installation on other operating systems. The final step in setting up Ant is to copy a JUnit jar file into the %ANT_HOME%\lib directory. This is the easiest way to setup Ant so that you can run JUnit tests from Ant. JUnit is part of the standard Groovy distribution and the jar file is located in %GROOVY_HOME%\lib.

Using Ant for One-Step Java Builds

The most common tool used for building Java projects is Ant and that will be the focus of these articles. I'll start with the typical minimal Ant script and work up from there. If you've ever worked with Ant, then the script below should look familiar. If you are not familiar with Ant, a great place to start is with Chapter 2 in "Pragmatic Project Automation". For more comprehensive information about Ant, the book "Java Development with Ant" is a great resource.

In the script below, the default Ant *target*, `jar`, depends on the `compile` target to compile all of the source code into a build directory. The `jar` target then invokes the Ant *task* by the same name to create an archive, `myproject.jar`, that contains the compiled classes.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Groovy and Java Bootstrap" default="jar" basedir=".">
  <property name="src.dir" value="src"/>
  <property name="build.dir" value="build"/>
  <property name="dist.dir" value="dist"/>
  <property name="lib.dir" value="lib"/>

  <path id="lib.classpath">
    <fileset dir="${lib.dir}" includes="**/*.jar"/>
  </path>

  <target name="clean" description="Remove all generated files.">
    <delete dir="${build.dir}"/>
    <delete dir="${dist.dir}"/>
  </target>

  <target name="compile" depends="init">
    <javac srcdir="${src.dir}" destdir="${build.dir}">
      <classpath refid="lib.classpath"/>
    </javac>
  </target>

  <target name="jar" depends="clean,compile">
    <jar jarfile="${dist.dir}/myproject.jar" basedir="${build.dir}"/>
  </target>
</project>
```

Listing 1: build.xml - Typical Ant script for Java projects

This is a typical boiler-plate Ant script. Even so, even this simple script has great power. It will run the same way on any platform where Java and Ant is installed. There is no IDE required to create the jar and no special set of paper instructions to follow. With this script, anyone on the team can build the project.

Incorporating the Groovy Compiler into the One-Step Build

The script above works great for a Java project. Now let's look at modifying the build script to compile Groovy source code. In the attached sample project, there is a Groovy class, `CsvReader` that parses each line in a comma separated file and puts the contents of each line into a `HashMap`. The source for this file is seen below in [Listing 3](#). The field names are the keys in the map and are defined in the first line of the file. The `read()` method returns the contents of the file as a list of these maps.

```
"firstName","lastName","datePlayed","scoreHole1","scoreHole2","scoreHole3",..
.
"Scott","Hickey","200601014",4,7,4,2,4,4,4,4,5
"Scott","Hickey","200601015",4,6,5,5,3,4,4,4,4
```

Listing 2: comma separated file

```
package org.hickey.io;

/**
 * This class will produce a list of maps where
 * each map represents a row in the table.
 */

class CsvReader {
    def recordList = []
    def fieldNames = []
    def isFirstRow = true

    List read(String fileName){
        println ("file=$fileName")
        new File(fileName).splitEachLine(",") { line ->
            if (isFirstRow) {
                fieldNames = line
                isFirstRow = false
            }else {
                def record = [:]
                recordList << record
                line.eachWithIndex{fieldValue, fieldNum->
                    record.put(fieldNames[fieldNum],fieldValue)
                }
            }
            return recordList
        }
    }
}
```

Listing 3: CsvReader.groovy

In order to compile the Groovy source file, it is necessary to add the `groovyc` Ant task to the `compile` target. The `groovyc` Ant task is an *external* task ships as part of the standard Groovy distribution. In the Ant source code listing below, the `groovyc` Ant task is imported into the script and then used in the compilation step.

```
<!-- import the task that compiles Groovy source files-->
  <taskdef name="groovyc" classpathref="lib.classpath"
           classname="org.codehaus.groovy.ant.Groovyc"/>

  <!-- compile Groovy and Java source -->
  <target name="compile" depends="init" >
    <javac srcdir="${src.dir}" destdir="${build.dir}">
      <classpath refid="lib.classpath"/>
    </javac>
    <groovyc srcdir="${src.dir}" destdir="${build.dir}">
      <classpath refid="lib.classpath"/>
    </groovyc>
  </target>
```

Listing 4: build.xml - adding the Groovy Compiler

The `taskdef` statement assumes that the required Groovy files are located on the classpath. The easiest way to include the Groovy runtime is to copy the `groovy-all` jar from `%GROOVY_HOME%\embeddable` into the project `lib` directory.

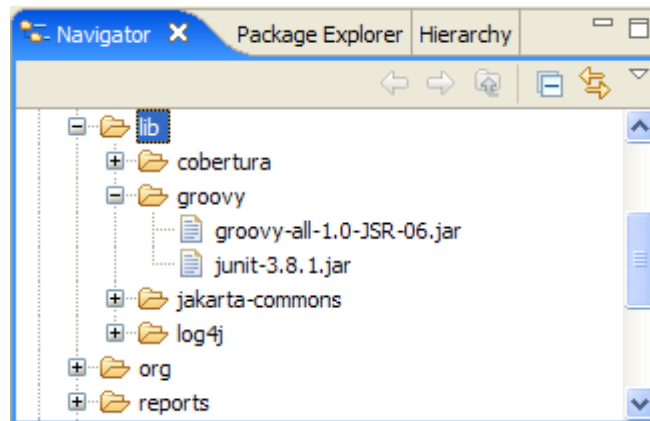


Illustration 1: lib directory – adding the Groovy runtime to the project classpath

If you build the attached sample application and inspect at the contents of the build directory, you can see there are .class files created by the `groovyC` Ant task. Like most other languages that run on the JVM, Groovy scripts can be interpreted and executed at runtime. More importantly for large project work, the Groovy compiler is also able to generate .class files. This is one of the really great things that distinguish Groovy from many other languages built on Java Virtual Machine (JVM). This important feature helps enables easy integration with many standard Java project tools such as Ant, JUnit and Cobertura.

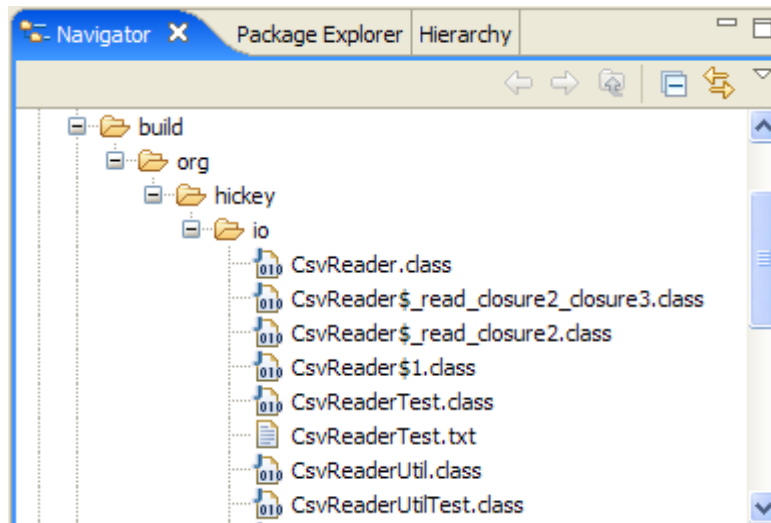


Illustration 2: build directory - class files created from Java and Groovy source

Mixing Java and Groovy

Now that Groovy and Java compilation are included in the build script, let's make the example more realistic. In my experience on a large business application with thousands of lines of Groovy source, there was the need to optimize the performance of the application in a couple of places. After some analysis, it was discovered that by replacing a couple of small Groovy classes with classes written in Java we were able to meet our customer's performance requirement. In the source code example below, I'll demonstrate how Groovy and Java can be used together within the same project to enhance the performance of the CSV file reader.

The `CsvReader` class works great as written in Groovy and is very easy to read. However, if I have to read millions of rows at a time, I will get faster results if I optimize the map creation. An easy way to optimize this is to write a Java utility class that will accept the list of field names and a list of Strings. The Java utility transforms the inputs into a map with field names as the keys and with the Strings, converted to numbers where appropriate, as values. Adding this utility is a simple change to the original `CsvReader`, as seen below. The complete source for the Groovy and Java is available in the attached project.

```

List read(String fileName){

    // create the Java utility class
    def util = new CsvReaderUtil()

    new File(fileName).splitEachLine(",") { line ->
        if (isFirstRow) {
            fieldNames = line
            isFirstRow = false
        }else {

            //use the Java utility class
            recordList << util.parse(fieldNames,line)

        }
    }
    return recordList
}

```

Listing 5: updated CsvReader that uses a Java utility class

In this situation, the Groovy class `CsvReader` *depends* on the Java class, `CsvReaderUtil`. Therefore, it is important that I compile the Java class first *as well as* include the build directory in the classpath for the Groovy compiler in order for `CsvReader` to compile correctly. In Listing 6 below, I have created an Ant path `id, build.classpath` that includes the build directory. The `groovyc` task is updated to use `build.classpath`.

```

<path id="build.classpath">
    <pathelement location="${build.dir}"/>
    <path refid="lib.classpath"/>
</path>

<target name="compile" depends="init">
    <javac srcdir="${src.dir}" destdir="${build.dir}">
        <classpath refid="lib.classpath"/>
    </javac>
    <groovyc srcdir="${src.dir}" destdir="${build.dir}">
        <classpath refid="build.classpath"/>
    </groovyc>
</target>

```

Listing 6: `build.xml` - adding Java build directory to the Groovy build path

The order of these two compilation tasks really depends how you will intend to use Groovy in your project. The compilation target is arranged differently depending on if you are primarily coding Java and calling Groovy for business logic versus primarily using Groovy and maybe using Java for some optimized utilities. The order matters because neither compiler recognizes the other's source file at compile time. Although Groovy generates `.class` files that can be referenced in the classpath for Java compiler, the Java compiler won't recognize the uncompiled classes defined in Groovy source at compile time. The reverse is true for the Groovy compiler as well. Since the example application is primarily written in Groovy with a Java helper class, I make sure the Java source is compiled first.

Creating the Distribution Jar

The original build script has been updated to invoke the Groovy compiler. Ultimately, the goal of the one step build is to compile *and* create the project jar file with one command. Because the Groovy files are being compiled into .class files that are written to the same build directory as the Java classes, there is nothing that needs to change! At this point, Ant treats both Groovy and Java class files the same way. Thus, with no changes to the `jar` target, the build script now enables any to simply type `ant jar` at the operating system command prompt create the distribution archive.

Conclusion

In this first part of the series, I've shown how easy it is to compile Groovy and Java into a typical project and create a one-step build with Ant. In particular, I've shown how it's important to structure your `compile` target to make the appropriate class files available for each compilation task when mixing Groovy and Java source code. In addition, I shown how compiling the Groovy source into .class files enables easy integration with the Ant `jar` task. In the next article, I'll expand the sample project to incorporate JUnit into the one-step build. I'll include examples for running individual unit tests as well and running all the unit tests at the same time. I'll also demonstrate creating a JUnit HTML report that summarizes the results from running all the tests.

Getting Your Project Started with Groovy, Java and One-Step Builds

In the first part of this series, I walked through creating a one-step build script using Ant that would compile both Groovy and Java source. The script also packages up the compiled classes into an archive for distribution. Of course, compiling source is just one part of a good build script. It also needs to make it easy to run unit tests as well. In this article, I'll walk through adding JUnit support for the Groovy and Java project. I'll demonstrate how to add support to run individual tests, run all the tests and create a JUnit report. All of these Ant targets will support application code written in either Groovy and Java.

Adding a Unit Test to the Build Script

In our build script, we want to include the ability to run an individual unit test and as well as all of the tests together. First, let's focus on running one test at a time to support the code-compile-debug cycle.

As noted in the first article, once all of the Groovy source code is compiled into .class files, they integrate into the build just like any other Java binary. This is illustrated by examining the Ant task that runs the JUnit test for `CsvReader`. Our unit tests for both the Groovy and Java application classes are written in Groovy. These unit tests extend `GroovyTestCase`, which in turn extends JUnit's `TestCase`. The Ant `junit` task knows nothing about Groovy and it doesn't need to. It's job is to execute the compiled test class. Once compiled, there is no distinction between a unit test written in Groovy and one written in Java.

Below is an Ant target to run the unit test for my Groovy `CsvReader`. I have created a generic JUnit target so that I may run any test from the command line by passing the test case in on the command line. For example, typing `ant -Dtestcase=org.hickey.io.CsvReaderTest` will execute my test case for `CsvReaderTest`. Notice that I have also created an Ant target to explicitly run this test. This is convenient for invoking frequently run tests. It enables me to simply type “`ant testCsvReader`” at the operating system prompt to run the unit test. These targets are easy to run inside of an IDE as well since they will also show up in a GUI that displays Ant targets, such as the Eclipse Ant view.

```
<target name="testCsvReader">
  <antcall target="runSingleJUnitTest">
    <param name="testcase" value="org.hickey.io.CsvReaderTest"/>
  </antcall>
</target>

<target name="runSingleJUnitTest" depends="compile" >
  <junit fork="true" haltonfailure="yes">
    <classpath refid="runtime.classpath"/>
    <formatter type="plain" usefile="false"/>
    <test name="${testcase}"/>
  </junit>
</target>
```

Listing 7: build.xml - run a single unit test

Running All Tests in the One-Step Build

Running one test is important while coding and debugging a specific source file, but during real project development I also need to easily run all of the JUnit tests before checking in changes to the version control system. Additionally, when refactoring or upgrading a dependent library, it is helpful to run all the tests and examine the results using the JUnit report task.

There are several ways to execute all of the tests. JUnit has a `TestSuite` class that allows developers to explicitly group tests to be run together. There are also utilities that will automatically search the classpath for classes that extend JUnit's `TestCase` class and execute all of these tests. Either of these techniques will work with tests written in Groovy.

Another technique is to follow a standard naming convention for test classes. This allows a file name pattern to be passed to the JUnit Ant task. I have had success by following the naming convention of ending all of my test classes with “Test” and the using `batchtest` property of the JUnit Ant task to execute all of these at once. This is illustrated in the code below.

```
<target name="run-all-junit-tests" depends="compile">
  <!-- set default classpath if not already set -->
  <property name="junitClasspathId" value="runtime.classpath"/>
  <property name="haltOnFailureFlag" value="true"/>

  <!-- clean up old junit xml files -->
  <delete dir="${junit.data.dir}"/>
  <mkdir dir="${junit.data.dir}"/>

  <!-- run the tests -->
  <junit printsummary="yes" haltonfailure="${haltOnFailureFlag}"
    failureproperty="testsDidNotPass" fork="yes">
    <formatter type="plain" usefile="false"/>
    <formatter type="xml"/>
    <batchtest todir="${junit.data.dir}">
      <fileset dir="${build.dir}" includes="**/*Test.class" />
    </batchtest>
    <classpath refid="${junitClasspathId}"/>
  </junit>
</target>
```

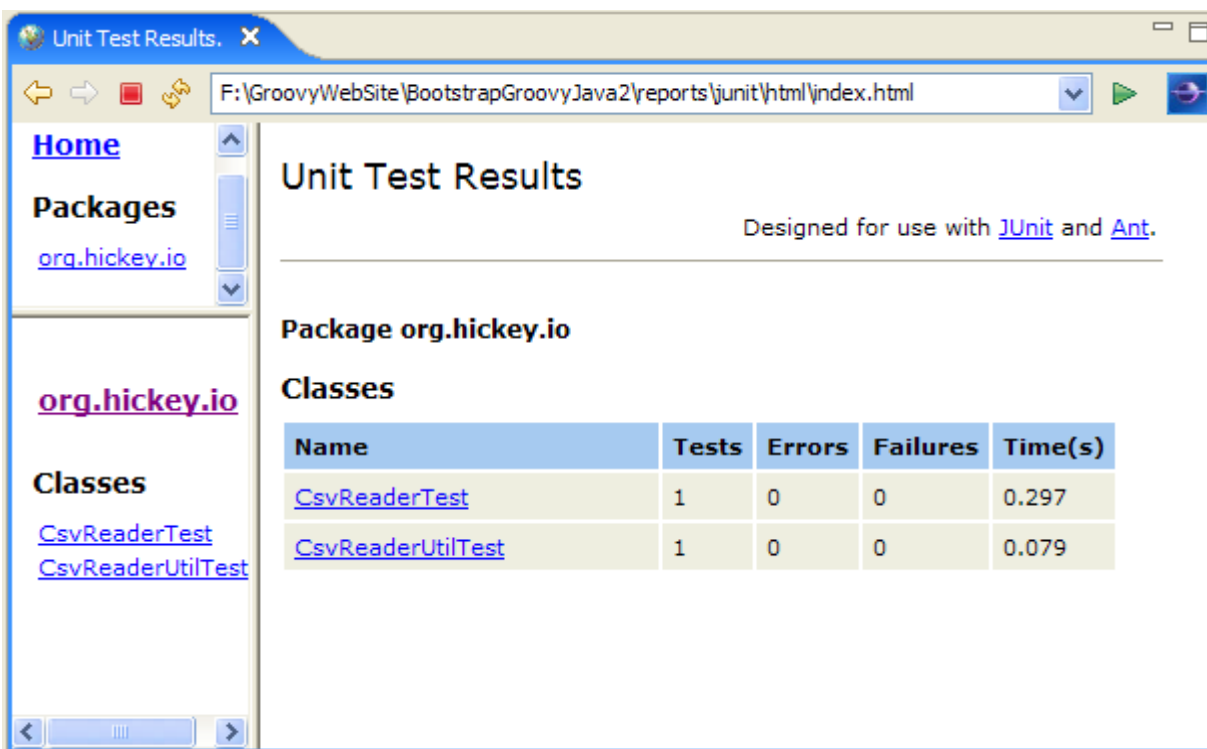
Listing 8: build.xml - run all JUnit tests

I have seen where some programmers like to follow the naming convention of having all tests being with `Test`. This doesn't work well with if you write test cases in Groovy. Anytime you use a closure such a `each`, an additional new class will be created that begins with the same name as the original class but has a `$` and some other text appended to it. In this situation, there will now be a `.class` file created that begins with `Test` but doesn't extend JUnit's `TestCase`. This will cause the JUnit task to fail.

Whether running a single test or multiple tests, Groovy integration is seamless. In either situation, the JUnit task isn't aware of the source code for the tests or the code being tested is written in Java or Groovy. JUnit only cares that the compiled test extends `TestCase`. Again, we are able to take advantage of the ability to compile Groovy source into `.class` files so that we can use standard Java build tools.

Publishing Test Results with JUnit Report

There are times when it makes sense to not only run all of the tests but also create a report that summarizes the results of running all the unit tests. For example, at the start of a major refactoring that will break many classes, a JUnit report can help understand the extent of the damage. It is also nice for tech leads and managers to be able to create a JUnit report on a scheduled basis to understand the current state of the code base.



The screenshot shows a web browser window titled "Unit Test Results" with the URL `F:\GroovyWebSite\BootstrapGroovyJava2\reports\junit\html\index.html`. The page content includes a sidebar with navigation links for "Home", "Packages" (pointing to `org.hickey.io`), and "Classes" (listing `CsvReaderTest` and `CsvReaderUtilTest`). The main content area displays "Unit Test Results" for "Package org.hickey.io" and a table of "Classes".

Name	Tests	Errors	Failures	Time(s)
CsvReaderTest	1	0	0	0.297
CsvReaderUtilTest	1	0	0	0.079

Illustration 3: Sample JUnit report showing Groovy unit tests

In order to create a useful report, it is necessary to change the default behavior of JUnit stopping when an error occurs. When running a single test, as shown in listing 7, the property `haltonfailure` is set to `true`. This stops Ant from proceeding further when an assertion fails or an exception occurs. This works well when fixing errors while working on a single class. This is also the default behavior for when running all the tests.

When creating a report, I really want all of the tests to be attempted so I can review how many tests are currently failing. The most common way I've seen this handled is to cut and paste the JUnit task as defined in listing 8 in the target `run-all-junit-tests` and the change value for `haltonfailure` from `true` to `false`. Of course, that would be violating the Don't-Repeat-Yourself (DRY) principle. A better option is to provide the ability to override the default behavior of stopping when a test fails which allows the JUnit report target to reuse the target `runAllUnitTests`. This is demonstrated in the attached sample project by avoiding the hard-coding the value for `haltonfailure`. The value it is set by the Ant property `haltOnFailureFlag`, defaulted to `true`, in the target `runAllUnitTests`.

To support the JUnit reports, as illustrated below in Listing 9, an additional Ant target, `junit-haltonfailure-false`, provides the ability to set this property to `false`. Since Ant properties are immutable once set, placing the `junit-haltonfailure-false` target in front of `run-all-junit-`

tests in the list of dependencies for the JUnit report target will allow enable the JUnit report to run for all of the tests in the project.

```
<target name="junit-haltonfailure-false">
  <property name="haltOnFailureFlag" value="false"/>
</target>
<target name="junit-report"
  depends="junit-haltonfailure-false, run-all-junit-tests"
  description="Run all unit tests and create JUnit Report">

  <!-- clean up old html JUnit report files -->
  <delete dir="${junit.report.dir}"/>
  <mkdir dir="${junit.report.dir}"/>

  <junitreport todir="${junit.report.dir}">
    <fileset dir="${junit.data.dir}">
      <include name="TEST-*.xml"/>
    </fileset>
    <report todir="${junit.report.dir}"/>
  </junitreport>
  <fail if="testsDidNotPass"/>
</target>
```

Listing 9: build.xml - create a JUnit report

The final piece of our JUnit report puzzle is to make sure that our JUnit report target causes Ant to fail if any of our JUnit tests fail. This is especially important when setting up automated builds using a tool like CruiseControl. It is useful to have CruiseControl automatically run all of the tests and create the JUnit report. But we definitely want CruiseControl to report on the success or failure of the tests running, not the success or failure of the ability to create a report.

Fortunately, the JUnit Ant task allows a property to be defined that will only be created when a test fails. This is illustrated in the run-all-junit-tests target above in Listing 9, with the option `failureproperty="testsDidNotPass"`. This allows any downstream target to check this property for test failure. In the example, this property is referenced in last line of our junit-report target and causes the target fail if the testsDidNotPass property is set.

Conclusion

In the first article of this series, I've shown how compiling the Groovy source into .class files enables simple integration with the Ant jar task. In this article, I expanded the sample project to incorporate JUnit into the one-step build. As demonstrated above, working with .class files generated from by Groovy compiler also allows application and test classes to integrate seamlessly JUnit. In the next installment of this series, I'll show how Cobertura, an open source code coverage tool, can be integrated into the build script to help verify the quality of the unit tests.

Adding Code Coverage to the One-Step Build

In the first article of this series, we built an Ant script to compile the Groovy source into .class files and used the Ant `jar` task to create a distribution archive. In the second article, we expanded the sample project by incorporating unit tests written with JUnit into the one-step build. As demonstrated in both articles, creating .class files with Groovy compiler allows for seamless integration with standard Java tools. In this installment of this series, we will add Cobertura, a free open source code coverage tool, to the build script to help verify the quality of the unit tests. First, let's walk through the reports Cobertura creates and see what percent of the application code is being executed by the unit tests. Then we can dive into the details for adding this functionality to the one step build script.

Cobertura Overview

Below in Illustration 4 is a code coverage report for our sample project. The report layout is very similar to standard Java Doc and JUnit reports. The frames on the left side of the report support navigation of the package and class hierarchy. In the content frame on the right, the red bars make it easy to identify which classes have source code that wasn't executed by the unit tests. Notice that both the class written in Groovy, `CsvReader`, and the class written in Java, `CsvReaderUtil`, are included on this report. When incorporated into a build script, anyone on the team can build the code, run all of the unit tests and inspect the not only which tests passed but also how much code was executed by the tests.

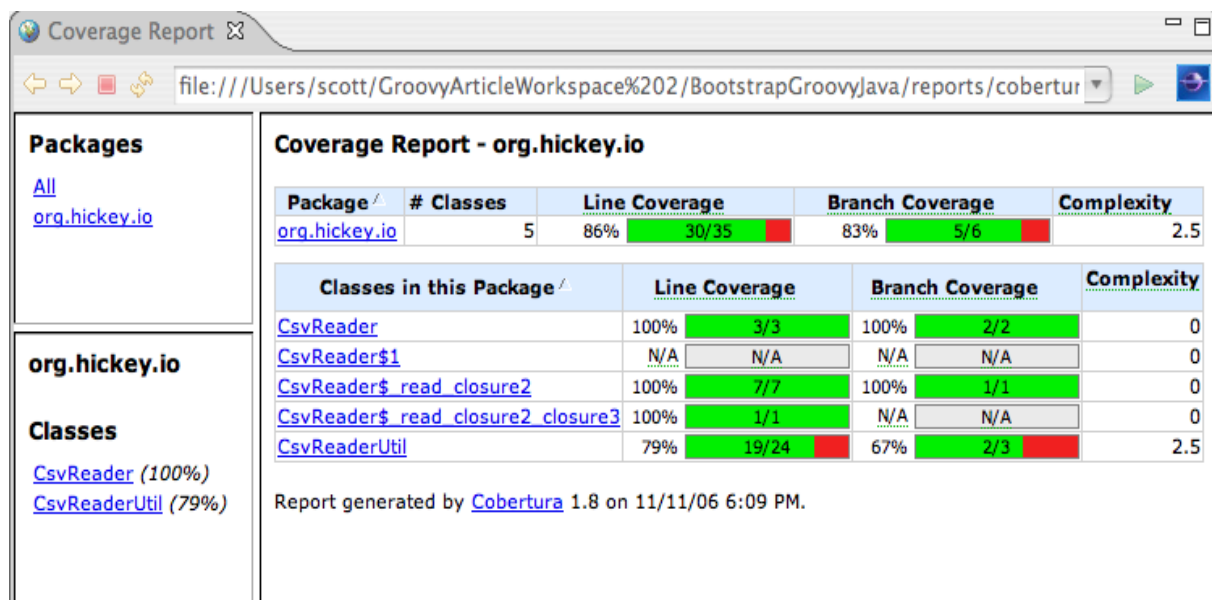


Illustration 4: Sample Cobertura Code Coverage Report

The summary report in Illustration 4 above is useful for the big picture. However, it is also possible to drill down into each class to see which lines of source code were not executed. In Illustration 5, the detail report for the Groovy class shows that all of the source lines were executed as well as how many times the lines were run.

Coverage Report - org.hickey.io.CsvReader

Classes in this File	Line Coverage	Branch Coverage	Complexity
CsvReader	100% 3/3	100% 2/2	0
CsvReader\$1	N/A N/A	N/A N/A	0
CsvReader\$ read_closure2	100% 7/7	100% 1/1	0
CsvReader\$ read_closure2_closure3	100% 1/1	N/A N/A	0

```

1 package org.hickey.io;
2
3 /**
4  * This class will produce a list of maps where
5  * each map represents a row in the table.
6  */
7 class CsvReader {
8     def recordList = []
9     def fieldNames = []
10    def isFirstRow = true
11
12    List read(String fileName){
13        println ("file=$fileName")
14        new File(fileName).splitEachLine(",") { line ->
15            if (isFirstRow) {
16                fieldNames = line
17                isFirstRow = false
18            }else {
19                def record = [:]
20                recordList << record
21                line.eachWithIndex{fieldValue, fieldNum->
22                    record.put(fieldNames[fieldNum],fieldValue)
23                }
24            }
25        }
26        return recordList
27    }
28 }

```

Illustration 5: Cobertura report with Groovy source

Of course, we can drill into Java source as well. In illustration 6, the detail report highlights the source that was missed by the unit tests in red. These reports make it easy to find where tests need to be improved. In this example, the test cases didn't exercise the code for creating BigDecimal and Integer numbers.

Classes in this File	Line Coverage	Branch Coverage	Complexity
CsvReaderUtil	79% 19/24	67% 2/3	2.5

```

1  package org.hickey.io;
2
3  import java.math.BigDecimal;
4  import java.util.ArrayList;
5  import java.util.List;
6  import java.util.Map;
7  import java.util.HashMap;
8  import java.util.StringTokenizer;
9
10
11 1 public class CsvReaderUtil {
12     public Map parse(List fieldNames, String line){
13 2         Map m = new HashMap();
14 2         line = line.replaceAll("[\r\n]", "\\n");
15 2         line = line.trim();
16         // replace all quotes
17         // line = line.replaceAll("\"", "");
18 2         StringTokenizer st = new StringTokenizer(line, ",");
19 2         int k=0;
20 6         while (st.hasMoreElements()) {
21 4             String token = (String)st.nextElement();
22 4             token = token.replaceAll("\"", "");
23 4             String fieldName = (String) fieldNames.get(k);
24 4             Object fieldValue = createFieldObject(token);
25 4             m.put(fieldName, fieldValue);
26 4             k++;
27 4         }
28 2         return m;
29     }
30     private Object createFieldObject(String token){
31 4         Object fieldValue = null;
32 4         if (token.matches("[0-9+-.]+")){
33 0             if (token.charAt('.') >= 0) {
34 0                 fieldValue = new BigDecimal(token);
35 0             } else {
36 0                 fieldValue = new Integer(token);
37     }

```

Illustration 6: Coverage detail for Java source

In practice, I've found that these reports are really useful for covering complex business logic. For non-trivial applications in insurance and finance, it can be challenging to write good unit tests. There so many combinations of variables that it can be difficult to ensure that every line of code has been tested. Although code coverage by itself doesn't guarantee the application won't have bugs, running coverage reports is a great way to ensure that the first time code is executed *isn't* in production.

Integrating Cobertura into the One-Step Build

Now that we've reviewed these great code coverage reports, let's walk through what's involved to add this functionality to our Ant build script. After making the Cobertura Ant tasks available to our script, there are four steps we need to follow. First, we need instrument our class files so that invocation data can be recorded as the code executes. Second, we will define an Ant classpath id that uses these instrumented class files instead of the originals. Third, we will re-run our JUnit tests with the instrumented classes. As these classes run, coverage data is recorded. Finally, we will run a coverage report task that will generate HTML reports from the coverage data.

To get started, we need to define some coverage related properties and import the Cobertura Ant tasks into our build file. At the top of Listing 10, we define a property for the directory that will hold the instrumented class files since we want to store them separately from the classes created for a normal build. All of the coverage data is stored in a serialized file, so we'll define a property for that as well. Finally, we want to define a place to store the HTML reports we create. In the next section of Listing 10, we setup a classpath id that includes all of the jars in the Cobertura distribution. We use that classpath id in last section of Listing 10 where we import the Cobertura Ant tasks.

```
<!-- properties for creating code coverage reports -->
<property name="cobertura.instrumented-classes.dir"
value="${report.dir}/cobertura/instrumented-classes"/>
<property name="cobertura.data.file" value="cobertura.ser"/>
<property name="cobertura.report.dir" value="${report.dir}/cobertura/html"/>

<!-- classpath for Cobertura jars -->
<path id="cobertura.classpath">
  <fileset dir="${lib.dir}/cobertura" includes="**/*.jar"/>
</path>

<!-- add support for creating code coverage reports -->
<taskdef classpath="${lib.dir}/cobertura/cobertura.jar"
resource="tasks.properties"
classpathref="cobertura.classpath"/>
```

Listing 10: Set coverage related properties and import Cobertura tasks

Cobertura provides the `cobertura-instrument` Ant task which adds code coverage instrumentation to our existing classes. As seen in listing 11, we provide a source directory that contains our original

```
<!-- instrument compiled class files with cobertura reporting code-->
<target name="instrument" depends="compile" >
  <delete quiet="false" failonerror="false">
    <fileset dir="${cobertura.instrumented-classes.dir}"/>
  </delete>
  <delete file="${cobertura.data.file}"/>
  <cobertura-instrument todir="${cobertura.instrumented-classes.dir}" >
    <fileset dir="${build.dir}">
      <include name="**/*.class"/>
      <exclude name="**/*Test.class"/>
    </fileset>
  </cobertura-instrument>
  <copy todir="${cobertura.instrumented-classes.dir}">
    <fileset dir="${src.dir}" casesensitive="yes">
      <patternset id="resources.ps"/>
    </fileset>
  </copy>
</target>
```

Listing 11: Ant task to instrument classes for collecting coverage data

class files and a destination for the new, instrumented version of the files. The `copy` task in listing 11 task will copy the extra files we need to run our unit tests, just like our original JUnit target.

Once we've created the instrumented classes we want to run, we need to setup a classpath environment

that includes these new classes, instead of the original. In listing 12 below, the Ant path id `cover-test.classpath` includes the `todir` directory specified in `cobertura-instrument` Ant task from listing 10. Once we've run all of our tests with the newly instrumented classes and gathered on the coverage information, we can finally create the coverage reports.

```
<path id="cover-test.classpath">
  <fileset dir="${lib.dir}" includes="**/*.jar"/>
  <pathelement location="${cobertura.instrumented-classes.dir}"/>
  <pathelement location="${build.dir}"/>
</path>

<target name="cover-test" depends="instrument" >
  <junit printsummary="yes"
    haltonerror="no" haltonfailure="no" fork="yes">
    <formatter type="plain" usefile="false"/>
    <batchtest>
      <fileset dir="${build.dir}" includes="**/*Test.class" />
    </batchtest>
    <classpath refid="cover-test.classpath"/>
  </junit>
</target>
```

Listing 12: Ant task for running JUnit tests using instrumented classes

The final step is to create the HTML reports using the coverage data gathered from running JUnit using the `cobertura-report` Ant task. In listing 13, you can see that we specify a location for the reports as well as the source code that the coverage data ties back to. The result of this is the report we saw at the beginning of this article.

```
<target name="coverage-report" depends="cover-test">
  <delete dir="${cobertura.report.dir}"/>
  <mkdir dir="${cobertura.report.dir}"/>
  <cobertura-report srcdir="${src.dir}"
    destdir="${cobertura.report.dir}"/>
</target>
```

Listing 13: Ant target for creating Cobertura report

Conclusion

A one-step build should be in place the first day a project starts and Ant continues to be a great tool to use for automatically building a Java based project. As the examples above have shown, it is not complicated to incorporate Groovy into a Java project and still utilize Ant, JUnit and Cobertura. By compiling Groovy into .class files, Groovy can seamlessly integrate into your Java project.

Resources:

[Ship It](http://www.pragmaticprogrammer.com/titles/prj/) <http://www.pragmaticprogrammer.com/titles/prj/>

[Pragmatic Project Automation](http://www.pragmaticprogrammer.com/starter_kit/auto/index.html) http://www.pragmaticprogrammer.com/starter_kit/auto/index.html

[Java Development with Ant](http://www.manning.com/hatcher/) <http://www.manning.com/hatcher/>

[Ant Elements of Style](http://wiki.apache.org/ant/TheElementsOfAntStyle) <http://wiki.apache.org/ant/TheElementsOfAntStyle>

CruiseControl Project Home : <http://cruisecontrol.sourceforge.net/>