Spot defects early with Continuous Integration

Your complete guide to creating a best-of-breed CI environment

Skill Level: Intermediate

Andrew Glover (aglover@stelligent.com)
President
Stelligent Incorporated

20 Nov 2007

Continuous Integration (or CI) is a process that consists of continuously compiling, testing, inspecting, and deploying source code. In many Continuous Integration environments, this means running a new build anytime code within a source code management repository changes. The benefit of CI is simple: assembling software often greatly increases the likelihood that you will spot defects early, when they still are relatively manageable. In this tutorial, a companion to his series *In pursuit of code quality* , Andrew Glover introduces the fundamental aspects of Continuous Integration and steps you through how to set up a CI process using best-of-breed open source technologies.

Section 1. Before you start

Learn what to expect from this tutorial and how to get the most out of it.

About this tutorial

This tutorial discusses the fundamental aspects of Continuous Integration: what it is, why you need it, how it works, even the pace of development in a CI environment. It shows you how to set up a CI process that delivers a repeatable and reliable build process.

You'll learn how to properly configure a CI server to poll an SCM repository and run

an Ant build process anytime a change is detected in your source code. You'll also learn how to run automated JUnit tests and how to make the most of both PMD and FindBugs for software inspection. Finally, you'll see for yourself how Hudson, truly a best-of-breed CI server, informs you of issues as they arise, ultimately enabling you to build reliable software more quickly.

Objectives

This tutorial guides you step-by-step through the fundamental concepts of Continuous Integration using Hudson, Ant, and Subversion as your framework. When you are done with this one-hour tutorial, you will understand the benefits of Continuous Integration as well as how to set up and properly configure Hudson, Ant, and Subversion to work together. The resulting build process will run both tests and software inspections and will report back violations almost as quickly as they occur.

Prerequisites

To get the most from this tutorial, you should be familiar with Java™ development in general. This tutorial also assumes that you understand the value of building software with an acceptable level of quality and that you are familiar with JUnit.

System requirements

A Continuous Integration environment requires an automated build tool, a code repository, and a CI server. To follow along and try out the code for this tutorial, you'll need a working installation of the Java platform as well as Hudson 1.150, Ant 1.7, JUnit 3.8.1, and Subversion 1.4.x.

The recommended system configuration for this tutorial is as follows:

- A system supporting either the Sun JDK 1.5.0_09 (or later) or the IBM Developer Kit forJava technology 1.5.0 SR3 with at least 500 MB of main memory
- At least 20MB of disk space to install the software components and examples covered

The instructions in the tutorial are based on a Microsoft® Windows® operating system. All the tools covered in the tutorial also work on Linux® and Unix® systems.

Section 2. Continuous Integration in a nutshell

The process of CI is about building software components often, which, in many instances, means any time code within a repository (such as Subversion or ClearCase) changes. The benefit of CI is simple: building software often ensures that you will encounter issues (such as code defects) early, as opposed to later in the software development life cycle when they are more expensive to address.

Tools versus process

While CI is actually a process, the term *Continuous Integration* often is associated with one or more particular tools. In this tutorial, I show you how to install, configure, and use Hudson as a CI server, but keep in mind that CI is much more than a tool. In fact, the tool you use is probably the least important aspect of CI because all it does is run your build when it detects a change within a code repository. The build itself is far more important than the tool you use to run it.

Getting started with CI

Getting started with CI requires three things:

- An automated build process with a platform like Ant or Maven
- A code repository like CVS or Subversion
- A CI server such as Hudson, although a cron job could suffice

Let's consider each of these components in detail.

Automated builds

The process of CI is about integrating software often, which is accomplished through the use of a build. In the Java world, Ant stands as the ubiquitous build platform. With Ant, you can reliably perform (in an automated fashion) otherwise manual tasks like compilation, testing, and even more interesting things like software inspection and deployment. As you'll see once everything has been wired together, your build strategy is by far the most important aspect of a successful CI process. In the absence of a solid build that does more than compile your code, CI withers.

Source code management

For CI to properly take shape, you need a source code management (SCM) system

or repository such as Subversion or CVS. A CI server polls the SCM repository for changes. On finding changes, the CI server performs a checkout (or an update of the local sandbox) and executes a build. More often than not, this is the same build you would execute in your local environment.

The CI server

For a successful CI process, it's also mighty helpful to have an automated process that monitors your SCM repository and runs builds when changes are detected. A host of CI servers are available for the Java platform, both open source and commercial. All are similar in their basic configuration and are optimized to monitor a particular SCM and run builds when changes are detected. All CI servers come with their own bells and whistles. Hudson is particularly interesting because of its ease of configuration and compelling plug-ins, which make aspects like test result trends particularly visible.

Section 3. Hudson explained

Hudson is an innovative open source CI server that arguably builds on lessons learned from previous CI servers. One of the most compelling features of Hudson is its ease of configuration: you would be hard-pressed to find an easier CI server to set up, or one that offers half as many features out of the box. Second to ease of use is Hudson's impressive plug-in framework, which makes it easy to add features. For instance, Hudson has a plug-in for tracking FindBugs issues over time as well as code coverage. It also trends test results (from either JUnit or TestNG), as well as build results and corresponding execution times.

As I'll show you, Hudson requires Java 5 to run — you only need a Servlet 2.4 container if you want to use something other than the embedded container Hudson ships with, which is Winstone. Winstone is adequate for most needs.

Section 4. Starting out with CI

This section looks at how the various components of a CI process fit together and why they fit together that way. After that, you'll be ready to start coding!

Prerequisite: A build system!

A repeatable and reliable build is the cornerstone of a predictable software process. As previously noted, Ant is a popular build tool for the Java platform, whose primary purpose is to automate common tasks like compilation and deployment. Ant also facilitates unit testing with test frameworks like JUnit and TestNG, and it integrates well with a myriad of other tools like PMD and FindBugs, which automate static code analysis. Compiling source files using Ant is as easy as issuing an `ant compile` command at a command prompt.

Reliable versus repeatable

While Ant facilitates repeatability, *reliability* is up to you. When it comes to software builds, reliability means that Joe and Fran will experience the same behavior when issuing commands like compile or test. If for some reason, Fran cannot compile because of a required library that is not explicitly referenced in the build itself (which Joe for some reason has), then the build isn't considered reliable. You could argue that it isn't repeatable either!

The aspects of reliability and repeatability are paramount when it comes to CI. Because CI is an automated process that occurs without human intervention, the build process that it will ultimately invoke must work flawlessly. It must be a simple event independent of aspects that the CI server cannot control, such as libraries needing to be in precise, undocumented locations, and databases manually refreshed with data. Surely, some builds will fail — for instance when someone checks in non-compiling code — but a build must not fail because the build itself is broken.

A basic build process

For CI to add value to a software development process, the build itself must do more than just compile code. Because a build is run every time code changes, it presents an excellent opportunity to run tests and code inspections.

A basic build process has tasks for the following:

- Compiling source code, including tests
- Executing tests, such as those written with JUnit or TestNG
- Running code inspections, such as PMD
- Archiving the final product into a JAR, WAR, or series of files
- Deploying the final assets (assuming the final product warrants it)

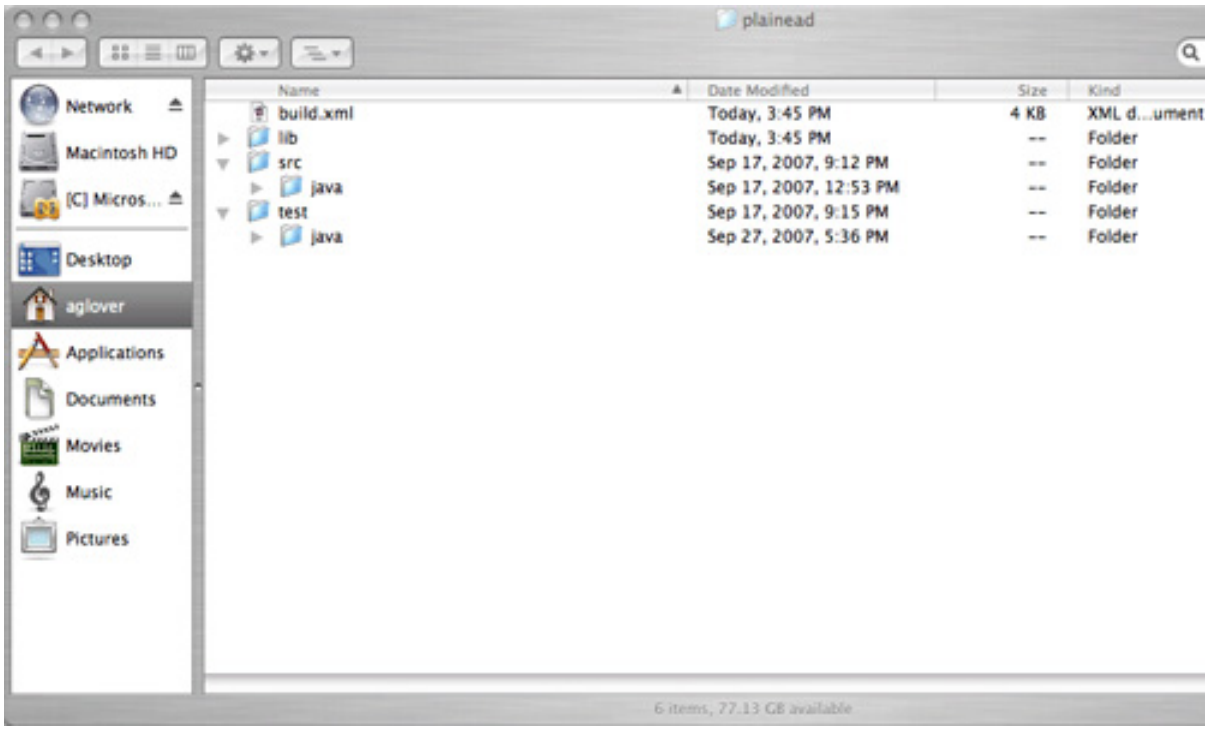
Keep in mind that these steps are minimal and don't even tackle more advanced options, such as dealing with databases or other aspects of a software product.

Setting up the directory structure

Myriad strategies exist for setting up the directory structure of a software project, and each one has its pros and cons. I happen to like separating *source code* from *test code*, and I always find myself creating different directory structures for these code lines. A common technique for handling third-party libraries is to create a lib directory as well. Keep in mind the more manageable mechanisms for dealing with third-party dependencies, however, such as Ivy.

Regardless of how you decide to set up your project's directory structure, the more detail and organization you bring to bear, the better. This is especially true as the project's assets (such as source files and tests) grow in number. For instance, Figure 1 illustrates a simple project directory structure that contains a directory for third-party libraries, and two separate directories for source files and their related tests:

Figure 1. A simple project structure



Note the build.xml file located in the project's root directory, shown in Figure 1. This is the project's build file, which defines a minimal set of automated actions related to moving source code into a production-ready state.

Section 5. CI basics: Code compilation

As promised, you're ready to start coding! This section helps you set up the basic infrastructure for a software project, which means setting up a project classpath and compilation. Believe it or not, without these pedestrian steps, nothing else is possible.

Compiling with Ant

The first step in creating a reliable and repeatable build is to limit hardcoded values, especially those related to filesystem paths, like directories. Accordingly, Listing 1 defines a number of Ant properties, which you can later reference in various associated targets:

Listing 1. Setting up properties in Ant

```
<property name="default.target.dir" value="target" />
<property name="classes.dir"
value="${default.target.dir}/classes" />
<property name="test.classes.dir"
value="${default.target.dir}/test-classes" />
<property name="test.report.dir"
value="${default.target.dir}/test-reports" />
<property name="lib.dir" value="${basedir}/lib" />

<property name="source.dir" value="src/java" />
<property name="test.source.dir" value="test/java" />

<property name="test.pattern" value="**/**Test.java" />
```

Creating a classpath

Because all your third-party libraries are located in a lib directory, you can quickly create a classpath by scanning that directory, shown in Figure 2. (Note that the directory is referenced via the `lib.dir` variable in Listing 1.)

Listing 2. Creating a classpath from a series of JAR files in the lib directory

```
<target name="init">
 <mkdir dir="${classes.dir}" />
 <mkdir dir="${test.classes.dir}" />

 <path id="build.classpath">
  <fileset dir="${lib.dir}">
   <include name="**/*.jar" />
```

```
  </fileset>
 </path>
</target>
```

Compiling the source code

With the classpath defined, you can create a `target` that compiles source code, as shown in Listing 3:

Listing 3. Using Ant's javac task to compile the source code

```
<target name="compile-source" depends="init"
  description="compiles all .java files in source directory ">
 <javac destdir="${classes.dir}" srcdir="${source.dir}" classpathref="build.classpath" />
</target>
```

Ant makes compilation easy to define through a task called `javac`. This task both compiles code in a directory using a classpath and places the class files in another directory.

Section 6. Testing, monitoring, and archiving

If you'd like CI to be of real value to you, I highly recommend you do more than just compilation continuously. If you want to get the biggest bang for your quality buck, so to speak, start with testing. You can use either JUnit or TestNG, the choice of testing framework doesn't matter. What does matter is that you run those tests *often* — meaning every time the code changes.

Testing with JUnit

Luckily, setting up JUnit with Ant couldn't be any easier: Ant ships with the ability to run JUnit tests via its `junit` task. When configuring the `junit` task, you need to think about where your JUnit tests can be found and how you would like to receive the results of those tests.

Listing 4. Running JUnit tests via Ant

```
<target name="test" depends="compile-tests" description="runs
JUnit tests">
 <mkdir dir="${test.report.dir}" />
 <junit dir="${basedir}" printSummary="on" fork="true"
```

```
haltonfailure="true">
  <sysproperty key="basedir" value="${basedir}" />
  <formatter type="xml" />
  <classpath>
   <path refid="build.classpath" />
   <pathelement path="${test.classes.dir}" />
   <pathelement path="${classes.dir}" />
  </classpath>
  <batchtest todir="${test.report.dir}">
   <fileset dir="${test.source.dir}">
    <include name="${test.pattern}" />
   </fileset>
  </batchtest>
 </junit>
</target>
```

In Listing 4, the `junit` task runs all tests found in the test.source.dir directory (which you can see defined in Listing 1). What's more, XML reports are written to another directory (test.report.dir). It will be handy to know where these test reports are written when you begin configuring a CI server, so keep it in mind.

Software inspections

With an automated mechanism like Ant at your beck and call, a whole world of options for monitoring code quality opens up. You have quite a few options for scanning source and binary files, but two of the most popular ones are PMD and FindBugs. PMD scans source files and checks the code in them against a series of rules. If there is an issue, it will report a violation. FindBugs does something similar but scans binary files (i.e., class files) and reports, well, bugs. Both of these tools integrate quite nicely with Hudson.

Running PMD

Running PMD via Ant is easy: just download it and follow the instructions! Just like with running JUnit tests, be sure to specify that an XML report is generated as a part of a PMD run.

Listing 5. Finding coding violations with PMD

```
<target name="pmd" depends="init">
 <taskdef name="pmd" classname="net.sourceforge.pmd.ant.PMDTask"
  classpathref="build.classpath" />
 <pmd>
  <ruleset>rulesets/basic.xml</ruleset>
  <ruleset>rulesets/braces.xml</ruleset>
  <ruleset>rulesets/javabeans.xml</ruleset>
  <ruleset>rulesets/unusedcode.xml</ruleset>
  <ruleset>rulesets/strings.xml</ruleset>
  <ruleset>rulesets/design.xml</ruleset>
  <ruleset>rulesets/coupling.xml</ruleset>
  <ruleset>rulesets/codesize.xml</ruleset>
  <ruleset>rulesets/imports.xml</ruleset>
  <ruleset>rulesets/naming.xml</ruleset>
  <formatter type="xml" toFile="${default.target.dir}/pmd_report.xml" />
```

```
  <fileset dir="${source.dir}">
   <include name="**/*.java" />
  </fileset>
 </pmd>
</target>
```

PMD has a cornucopia of rules you can run against your code base. Listing 5 shows quite a few of the options, but the choice of what to run is entirely up to you.

Running FindBugs

FindBugs scans binary files, and it's often easier just to scan a JAR file containing a project's files. The `findbugs` target in Listing 6 depends on the `jar` target:

Listing 6. Discovering bugs is easy with FindBugs

```
<target name="findbugs" depends="jar">
 <taskdef name="findbugs" classname="edu.umd.cs.findbugs.anttask.FindBugsTask"
    classpathref="build.classpath" />
 <findbugs classpathref="build.classpath" pluginlist="${lib.dir}/coreplugin-1.0.jar"
    output="xml" outputFile="${default.target.dir}/findbugs.xml">
  <sourcePath path="${source.dir}" />
  <class location="${default.target.dir}/plainead.jar" />
 </findbugs>
</target>
```

As with the JUnit and PMD tasks, you should specify that FindBugs will create an XML file for later analysis.

Achieving binary files into a jar

Regardless of how your project is ultimately deployed, chances are you'll need to either create a WAR or JAR file. For this project, you'll create a JAR file. As Listing 7 shows, creating a JAR file from a series of class files is as simple as compilation in Ant:

Listing 7. Achieving binary assets couldn't be easier with Ant

```
<target name="jar" depends="test">
 <jar jarfile="${default.target.dir}/plainead.jar" basedir="${classes.dir}" />
</target>
```

In this section and the previous one, you created just a few targets, but the end result is a repeatable and reliable build system for just about any code base. Not only does this build file facilitate compilation, but it also runs tests. It also utilizes two different inspection tools, which you can use to evaluate specific aspects of code quality. Lastly, this build file enables you to package your code base into a JAR file. Remember that this last step will vary from project to project. If you are building a

Web application, you'll probably want to build in a bit more control than you see in Listing 7; for instance, you might want to create a WAR file with the application's assets and then deploy them to a Web container.

Section 7. No SCM, no CI

After a reliable build process, the next requirement for a CI process is an SCM repository. There are myriad SCM repositories on the market ranging from open source to commercial. All serve the basic purpose of managing source code assets. With an SCM system, source code versioning and history management — both quite important when multiple people are working on the same file — are done for you. If you are currently working on a software project that doesn't make use of an SCM repository, you should probably stop reading this tutorial right now and focus on getting one up and running as soon as possible.

SCM integration with Hudson

Out of the box, Hudson supports CVS and Subversion, which are both freely available. Hudson also has plug-ins for integrating with Rational ClearCase, if you're looking for a commercial alternative.

Working with Subversion

For the purpose of this tutorial, I assume you're using Subversion. If you aren't using Subversion, don't fret: the basic principles are the same regardless of your SCM system. In fact, at a high level, the process of CI is quite simple when it comes to an SCM. A CI server, in this case, Hudson, is essentially configured to poll an SCM for changes to a specific project. If a change is detected, the CI server executes an update (which means it grabs the latest copy in an SCM) and runs a build. In this example, Hudson runs the build defined earlier in the tutorial.

URL-based access

In Subversion's lingo, projects live in a repository. Depending on how it has been configured, the repository can be accessible via a URL, which is essentially a combination of the repository's path plus the name of a project. If you are using a different SCM from Subversion, you may be using a different mechanism to access your repository. Whatever the case, it is imperative that you be able to properly configure a CI server to access the project repository. For now, let's assume you're using Subversion, so all you need is the URL to check out a desired project.

Let's say the project's name is *solar-ci* and the repository's URL is:

```
http://scm.acme.com/svn-repo/main/trunks/
```

In that case, the project's access URL is:

```
http://scm.acme.com/svn-repo/main/trunks/solar-ci
```

You have to properly configure repository access based on how your SCM is set up; for example, Subversion requires a user name and password. Oftentimes when setting up a CI process, it makes sense to create a new user that the CI server will act as. For this project, you'll create a new user named `buildmaster`. Later, when you configure Hudson to monitor your project, you explicitly specify the `buildmaster` credentials to perform checkouts and other functions.

Section 8. Hudson

The final aspect of a successful CI process is the CI server itself. The CI server's main role in the entire developmental dance is that of the director: When the server detects a change within a code repository, it merely defers the responsibility of running a build to the build process itself. If the build indicates it failed, then the CI server will notify interested parties and go back to monitoring a repository for changes. Its role may seem passive in nature; however, it is the vehicle by which rapid feedback of issues is made possible.

Installing Hudson

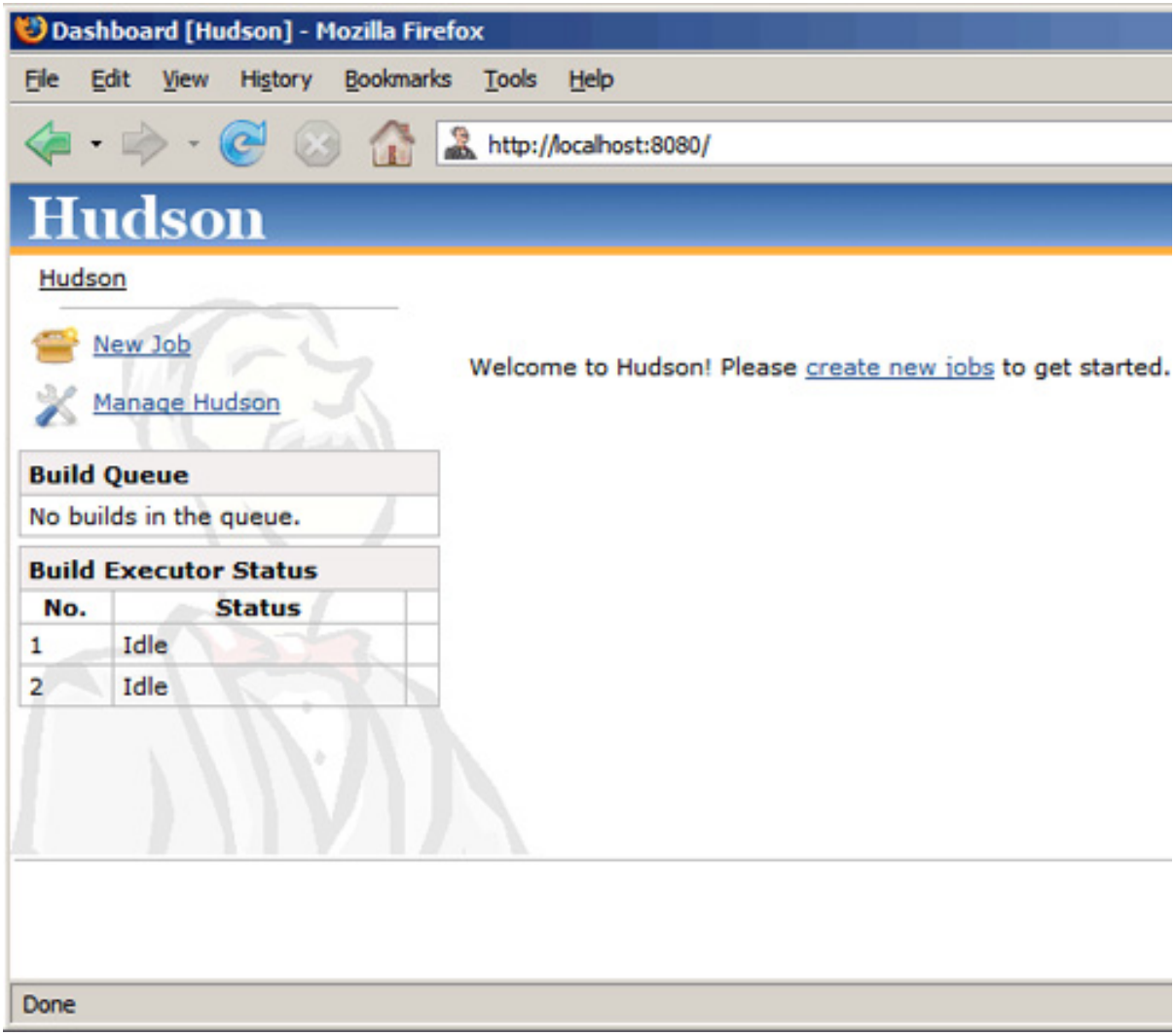
One of the primary benefits of using Hudson is its simple set up. At a minimum, Hudson requires two steps:

1. Download the latest version (which is bundled as a WAR file).
2. Run `java -jar hudson.war`.

That's it. Because the download is a WAR file, you are free to deploy it to a container such as Tomcat or JBoss, if you want. The choice is entirely up to you. Of course, Hudson's installation does assume you've got Java 5 running on the box it is destined to live on, and Hudson will use the `JAVA_HOME` environmental variable if you've defined it. (As I mentioned earlier, Hudson requires Java 5.)

After you've got Hudson up and running (either by deploying the war file to a servlet container or issuing `java -jar hudson.war` from the command line), fire up a browser and go to the default location of your installation. If you ran Hudson via the command line and you are on the local machine, you can surf to `http://localhost:8080/`.

Figure 2. Hudson is ready to go!



Assuming everything went well (it would be pretty hard to mess this up), you should see the Hudson start page displayed in Figure 2.

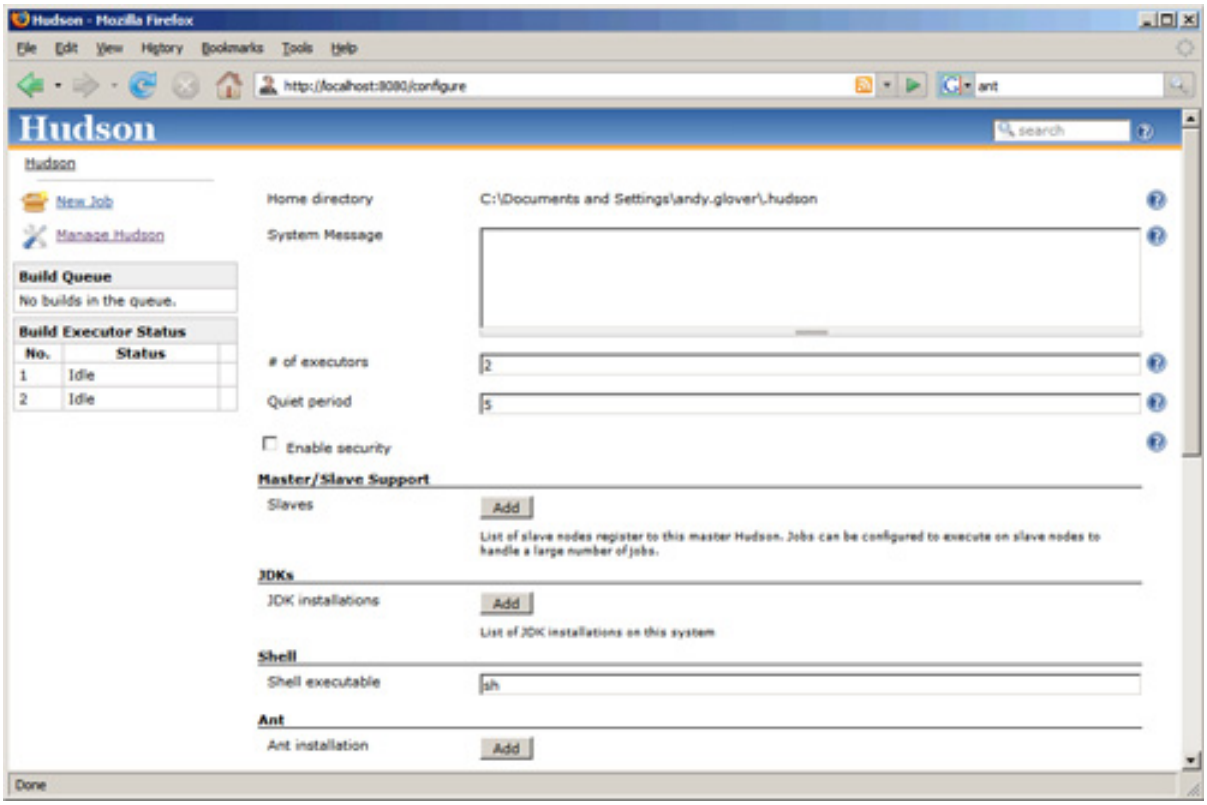
Configuring Hudson

The first step to configuring Hudson is to let it know where it can find your build platform's executables. In this example, you've defined Ant as your build system, so you need to tell Hudson where it can find Ant on the local machine. If you are using

Maven, you'll have to do the same thing: tell Hudson where it can find Maven. Remember, you aren't telling Hudson where your build file lives; you are telling Hudson where it can find the executable so it can *call* your build file. In this example, you need to tell Hudson where it can find the Ant command so that it can issue the command `ant -f build.xml`.

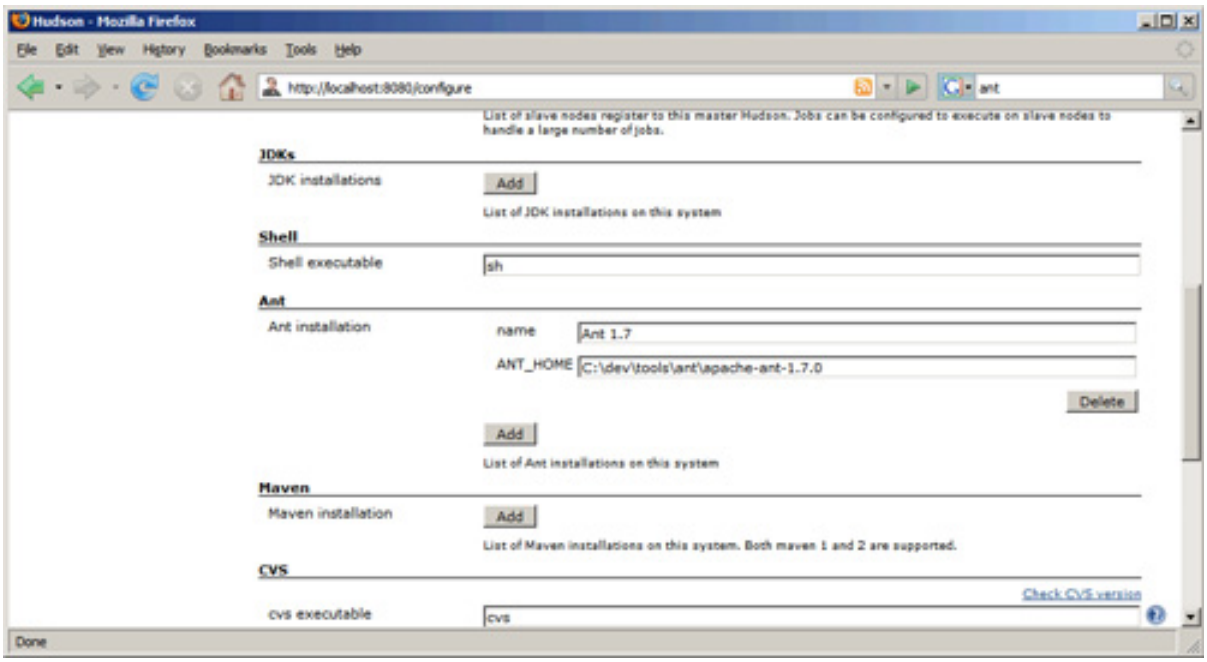
If you go to your local instance of Hudson's homepage and click the **Manage Hudson** link in the top left corner, you should see a list of configurable options as shown in Figure 3.

Figure 3. Configuring Hudson is easier than you think



In the Ant section, you need to provide a path to where Ant is installed, as shown in Figure 4:

Figure 4. Pointing Hudson Antward



You can configure a few other aspects of the server, such as pointing Hudson to an e-mail server, which comes in handy if you want to receive e-mails when a build fails. Depending on how e-mail is set up for your organization, you may need to work with your system administrator to get this feature going. Setting up e-mail isn't required; Hudson also supports RSS as a notification mechanism, which for some, is less obtrusive than e-mail. Your choice of notification mechanisms is entirely up to you. (Also note the plural: you can configure as many as you like!)

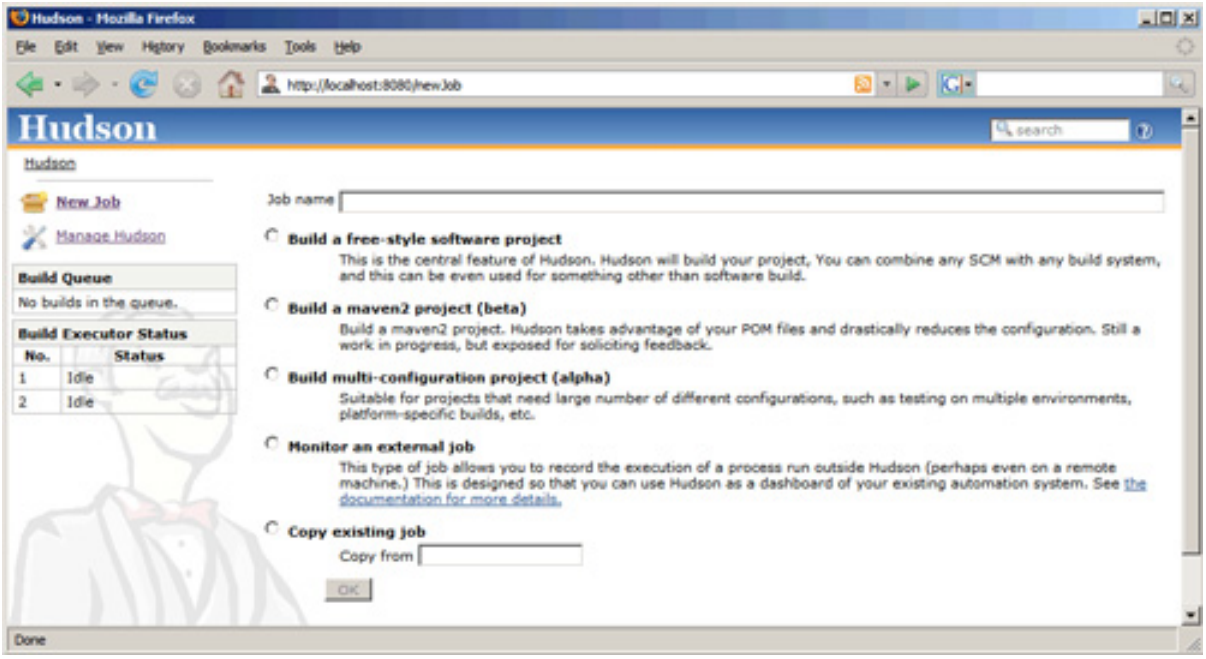
Lastly, before you can configure a project, you need to set up Hudson to talk to your particular SCM. In this case, you need to tell Hudson the path to your Subversion repository and the credentials to work with it.

Simply point your browser to `http://localhost:8080/scm/SubversionSCM/enterCredential` and specify the repository URL to a project along with the proper credentials (such as buildmaster, etc). This one-time step will validate that Hudson can correctly communicate with Subversion. (Of course, you'll have to do it again if you decide to point to another Subversion repository at a different URL.)

Configuring a project in Hudson

Now that Hudson is properly configured to talk to your SCM repository, you're ready to configure a project. The project for this example is called `solar-ci`, so from the Hudson home page, click the **New Job** link on the top left. You get the screen shown in Figure 5:

Figure 5. Configuring a job in Hudson



Give the job a name (Solar Project for this example) and select the **Build a free-style software project** option. As you can see, if you are a Maven 2 user, Hudson can quickly configure your project based upon your project's configuration files.

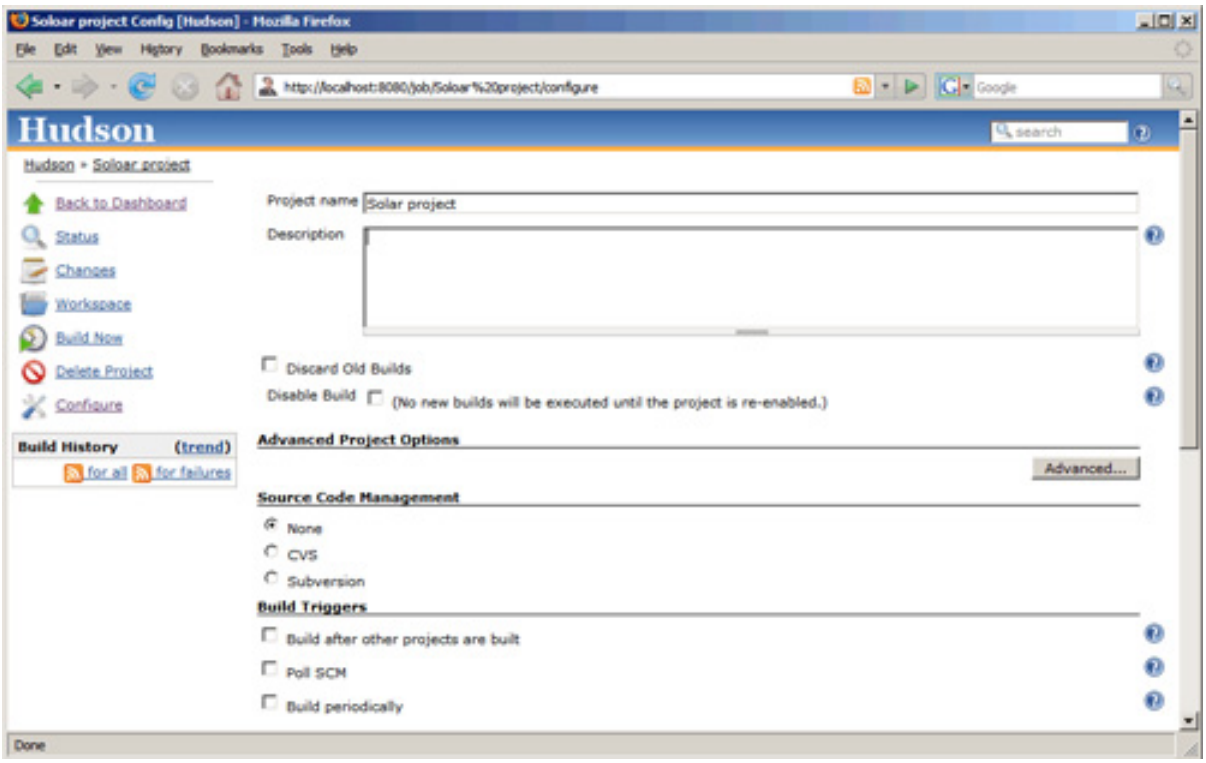
Project details

Once you click **OK**, you'll be presented with a project-configuration screen that allows you to specify the following:

- The SCM to connect to
- How often to build the project
- Which build platform to invoke (Ant, Maven, etc.)

In addition, you can configure some post-build actions, such as sending e-mails or publishing related assets. The options shown in Figure 6 are fairly self explanatory. Setting up a CI project in Hudson couldn't be easier.

Figure 6. Configuring project details in Hudson



For this project, you need to select the **Subversion** option, which then requires you to specify at least the project's URL.

Scheduling inspections

In the Build Triggers section, you can see that you have a number of options. I find the Poll SCM option very useful for creating a schedule for how often Hudson will check an SCM for changes. The schedule you create will vary based on your needs; if you have a large team making changes often, you'll want to poll frequently (say, every five minutes).

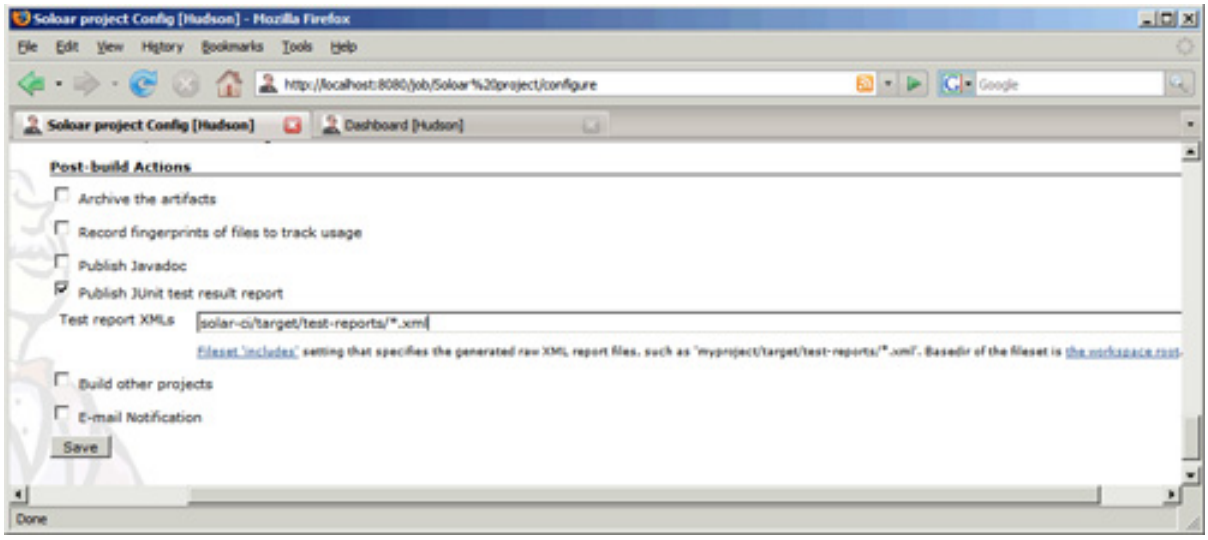
So, select the **Poll SCM** option and then in the Schedule box, type * * * * *, which tells Hudson to poll every minute. This is helpful for demonstration purposes (as you won't have to wait long for builds to be triggered after a change made to your SCM) but remember to specify something more reasonable once you're comfortable with the process. Click the question-mark (?) icon for more details on configuring the cron tab.

Configuring JUnit test trending

In the Build section, select the **Invoke Ant** option, select the Ant version you configured earlier, and then tell Hudson which target or targets to execute in the build defined earlier. For now, you can just tell Hudson to execute the test target. You may remember that this target will compile all source files and then run any JUnit tests you have defined.

In the Post-Build Actions section, select the **Publish JUnit test result report** option. You'll see that you have to specify a location for where Hudson can find the XML files that JUnit produces when run through Ant. Given that the project's name in Subversion is `solar-ci` and the build file is configured to write those reports in the `target/test-reports` directly, you should enter `solar-ci/target/test-reports/*.xml`, as shown in Figure 7:

Figure 7. Configuring JUnit trending with Hudson



Click **Save** and you're good to go.

The project home page

As you can see in Figure 8, Hudson presents a project's homepage with a number of options for modifying the configuration, forcing a build, viewing detailed changes related to the project's assets, and more! The next section walks you through using these options to gain detailed insight into all aspects of your software projects.

Figure 8. A CI project's homepage in Hudson

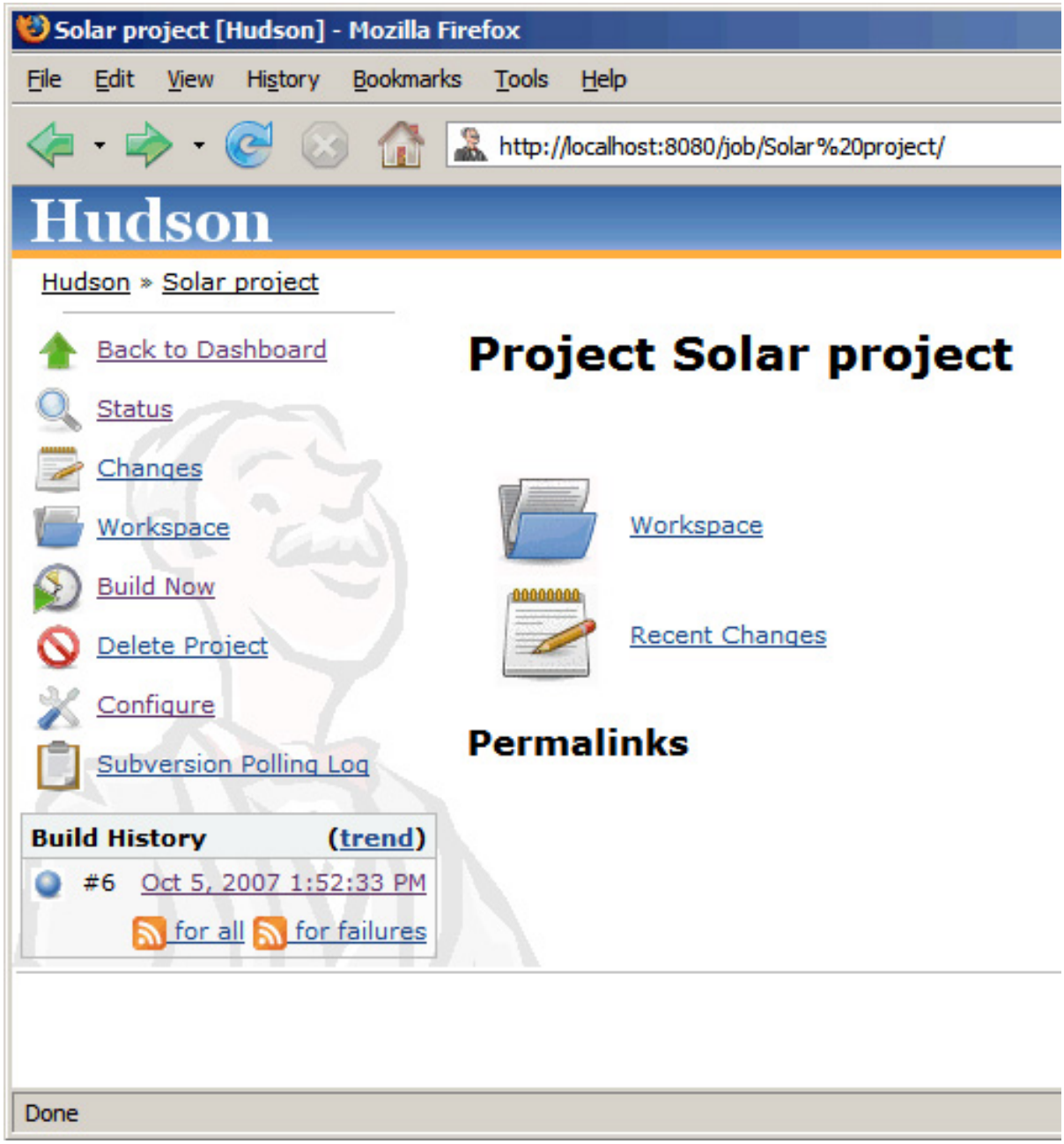
Section 9. Are you ready for CI?

You've properly configured a CI project in Hudson, so it's *almost* time to start doing something with it! Before we get into that, though, let's make sure your project is properly set up.

Checking the build status

You've configured Hudson to poll every minute, and the project hasn't ever been built, so pretty soon Hudson will automatically trigger a build. Check the project's home page and you'll see a new entry in the Build History box on the bottom left-hand side of the page, as shown in Figure 9:

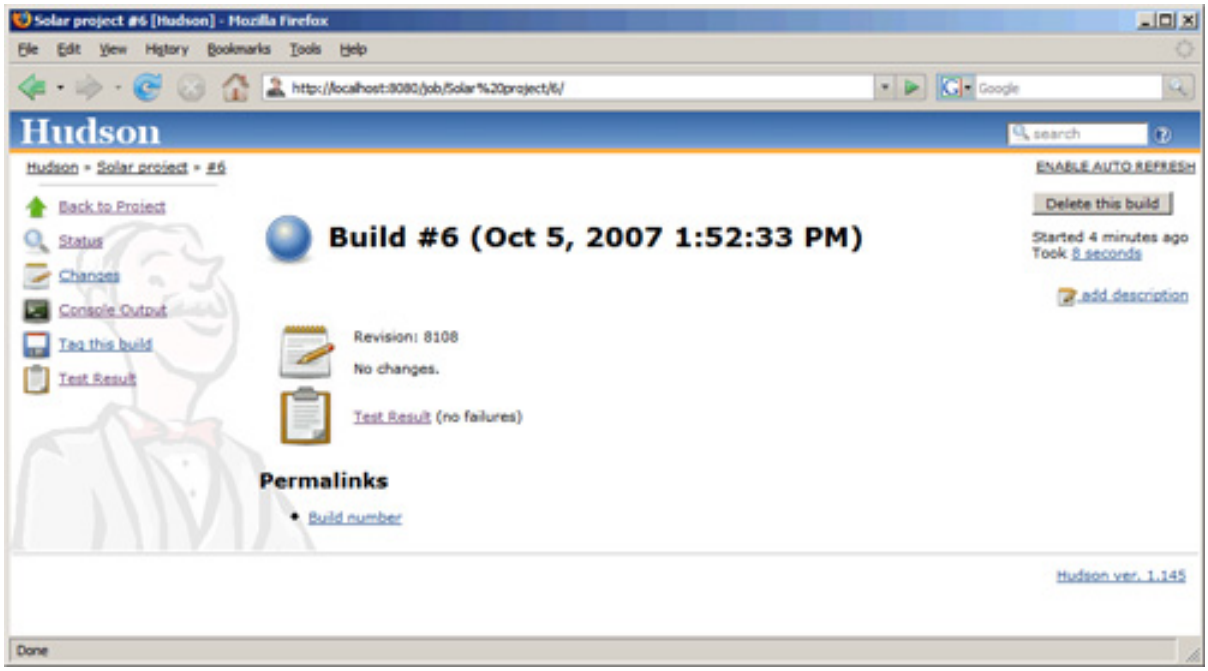
Figure 9. Hudson has run a build!



Further details

Click the build's date, which leads you to specific information about the build, as shown in Figure 10:

Figure 10. Hudson's Build Status page



The Build Status page shown in Figure 10 allows you to view specific changes that occurred on the source code (this initial build will report none) as well as test results. Additionally, you can view the output of the build by clicking the **Console Output** link.

What else can you do?

If you go back to the project's homepage, you'll note that you can subscribe to the project's Build Status page via RSS. Also note that you can subscribe for all builds or just failures. I usually just subscribe to *failures* because I need to be alerted to take an action when those happen.

In addition to checking the build status, Hudson lets you force a build (click the **Build Now** link), view changes to multiple builds, and even view the workspace where Hudson performs builds. (If you ever need to retrieve project assets, like WAR files for example, you'll do it from the workspace.)

Section 10. Let's write some code!

So, all you need for a CI process is three things: a build, an SCM, and CI server. You have all three and they're set up and ready to go. Of course, for CI to be useful, you need to write some code and set it in motion. In this section, you'll see how a CI process works from start to finish.

The cadence of CI

Before diving into a start-to-finish CI scenario, it's probably helpful to define what I like to call the *cadence of development* when working with CI. Having a CI server run builds doesn't preclude you from running local builds yourself. In fact, with a proper CI process in place, which notifies you of a failure in the SCM, it pays to ensure everything is working smoothly before you check in any new or modified code. Doing so will spare everyone the needless noise that could otherwise occur.

Accordingly, the cadence of development in a CI environment looks something like this:

1. Do an update of your local sandbox.
2. Modify some code.
3. Write some tests.
4. Run a local build to ensure you haven't broken anything.
5. Execute an update to ensure you have the latest code from the SCM, then run another build to see that things are still good.
6. Check in your changes.

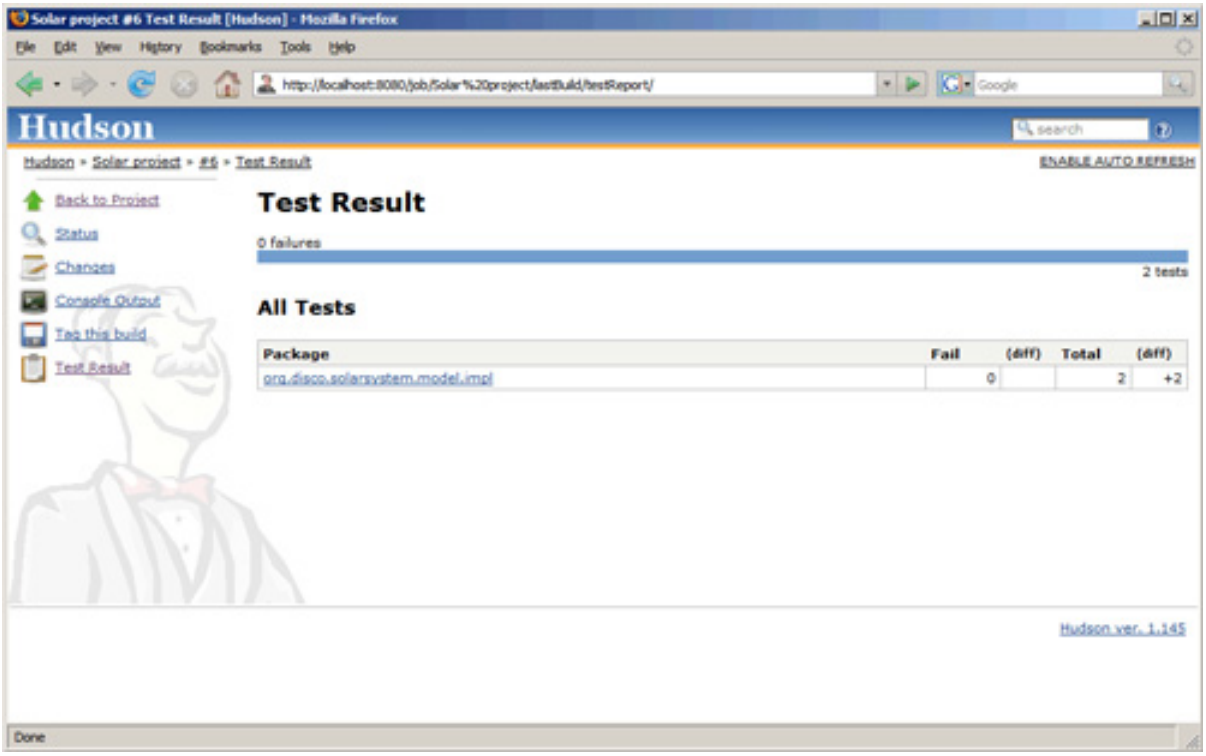
Note that Step 5 is crucial when you're part of a team where people are working simultaneously on the code base, especially if you've been at it for a long time.

Hopefully most developers will find the cadence of CI quite natural. It's a good working rhythm in or out of a CI environment!

Two tests, and two tests, and two ...

So now it's time to start doing some real work. For example, when Hudson ran the first build of the example project, it discovered two JUnit tests and ran them; accordingly, it reported that two passed and that two were new, as shown in Figure 11:

Figure 11. These tests ran swimmingly well



You're right to be pleased that these tests passed, but let's say you have a hunch that they don't offer enough coverage by themselves. (Of course, you could obtain a coverage report to avoid hunches, but for now we'll stick with intuition.) To ensure the code is solid, you'll write two additional tests, run them locally to verify they work, do a quick update, run the build again, and then check them in. After that, you'll write *another two tests* and re-execute the same steps.

Checking the build status

After you check in your two new tests, you can go to the project homepage to check their status. Within a few moments, in the Build History box, you'll notice a pending build. Yep, Hudson has detected a change in the SCM repository!

Figure 12. Hudson responds to a change in Subversion

Checking the build's details

Hudson's Build History box shows a pending build in Figure 12. After the build completes (hopefully in a passing state), you can click the build's date to receive more detailed information, as shown in Figure 13:

Figure 13. Details after running two new tests

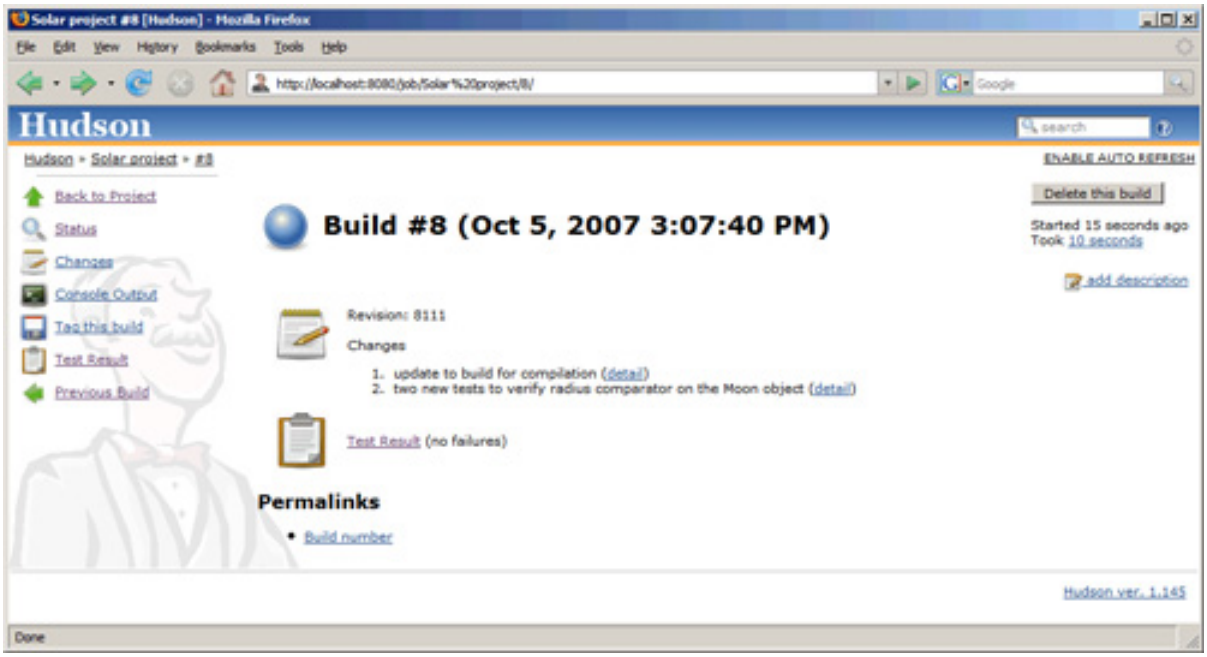
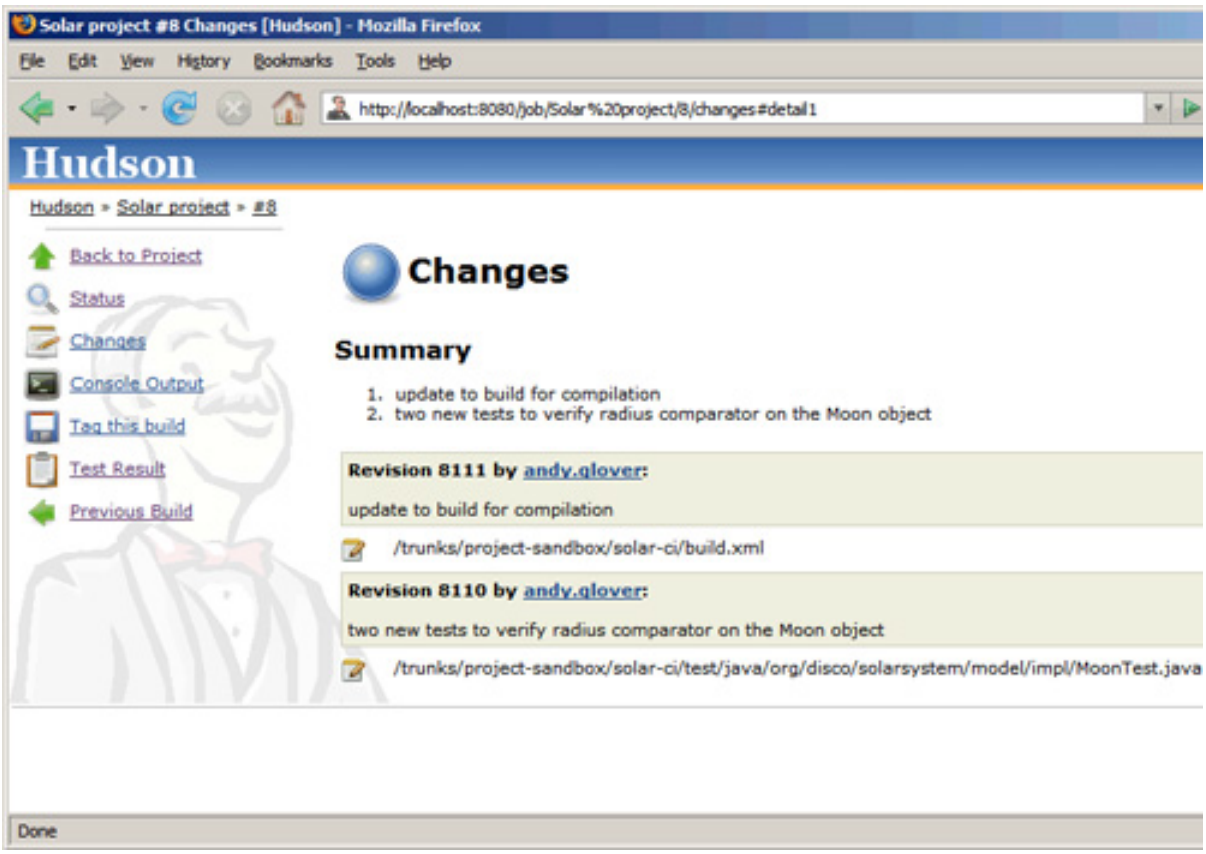


Figure 13 shows the changes that triggered this build. You can click the detail link next to each change to see more about the changes, such as who made the changes and their commit statement, as well as the particular detail shown in Figure 14:

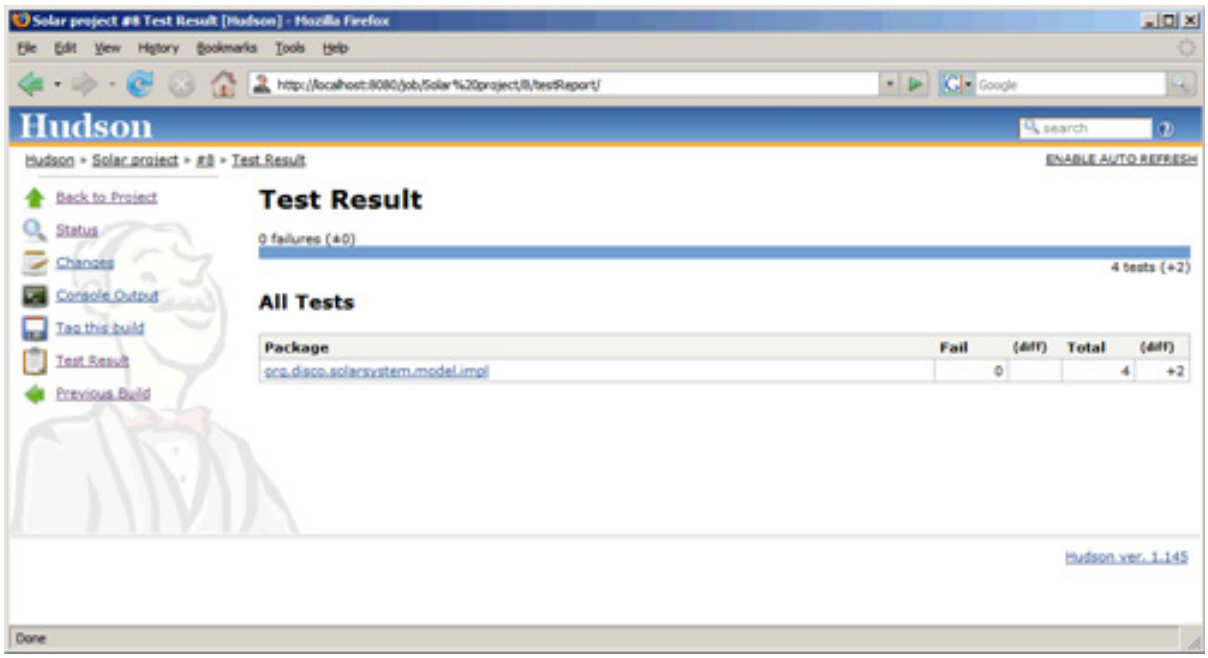
Figure 14. Two revisions were found in this build



Did the tests pass?

If you go back to the build status page and click the **Test Result** link, you can also see that two new tests were run and that all four tests run so far have passed, as shown in Figure 15:

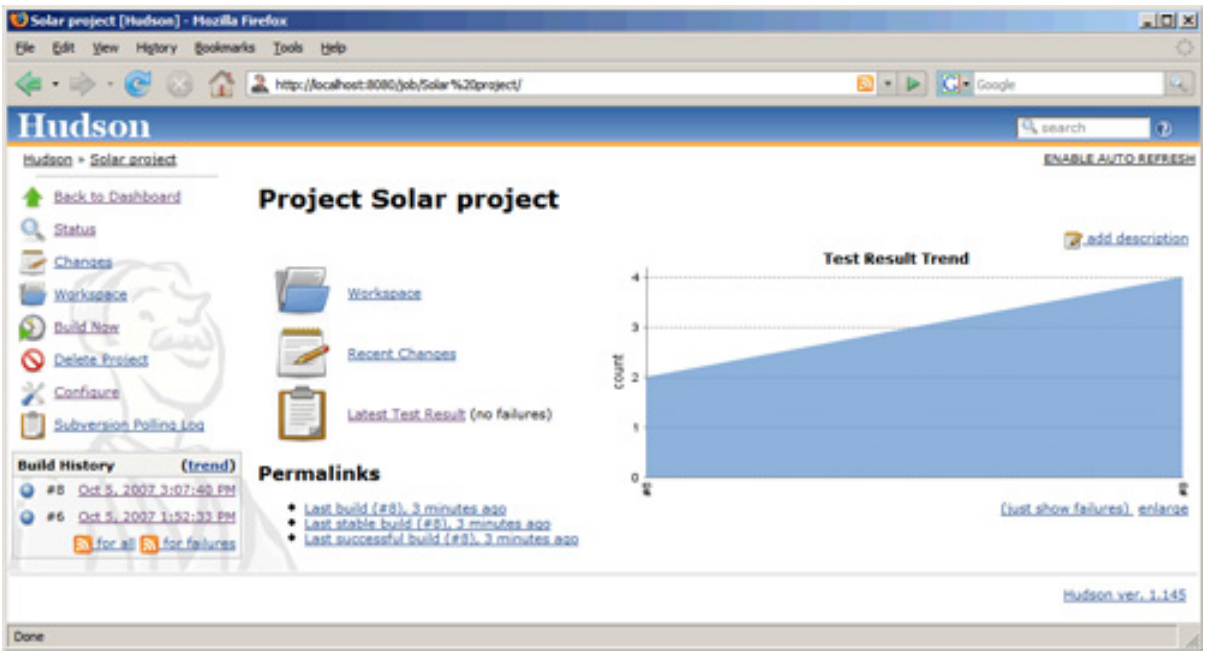
Figure 15. Test report



Test trending

When you configured this project, I had you point Hudson to JUnit's result files but didn't tell you why. I wanted to demonstrate a great out-of-the-box feature that Hudson supports. For each build, it will parse the corresponding JUnit result XML files and build a trend graph. In fact, if you go back to the project's homepage, you'll see a trend graph showing that for two builds so far, the test count has doubled.

Figure 16. Not a bad trend 100% growth

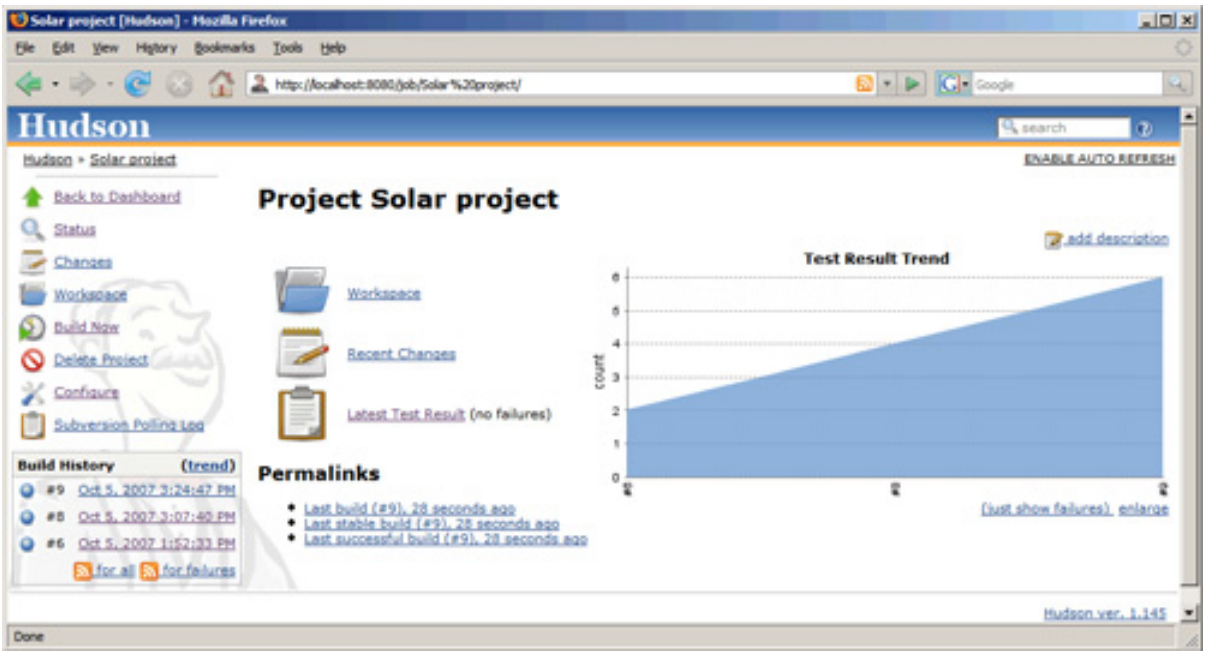


The trend graph in Figure 16 shows that the number of tests has doubled between Builds 6 and 8 (which, in this case, are sequential builds).

The same graph after two more tests

If you write two more tests, the trend graph in Figure 16 will continue to show a very good trajectory. That is, as each build is run, the amount of tests run is increasing. You can also see that those tests are consistently passing because the graph is blue, rather than red, shown in Figure 17:

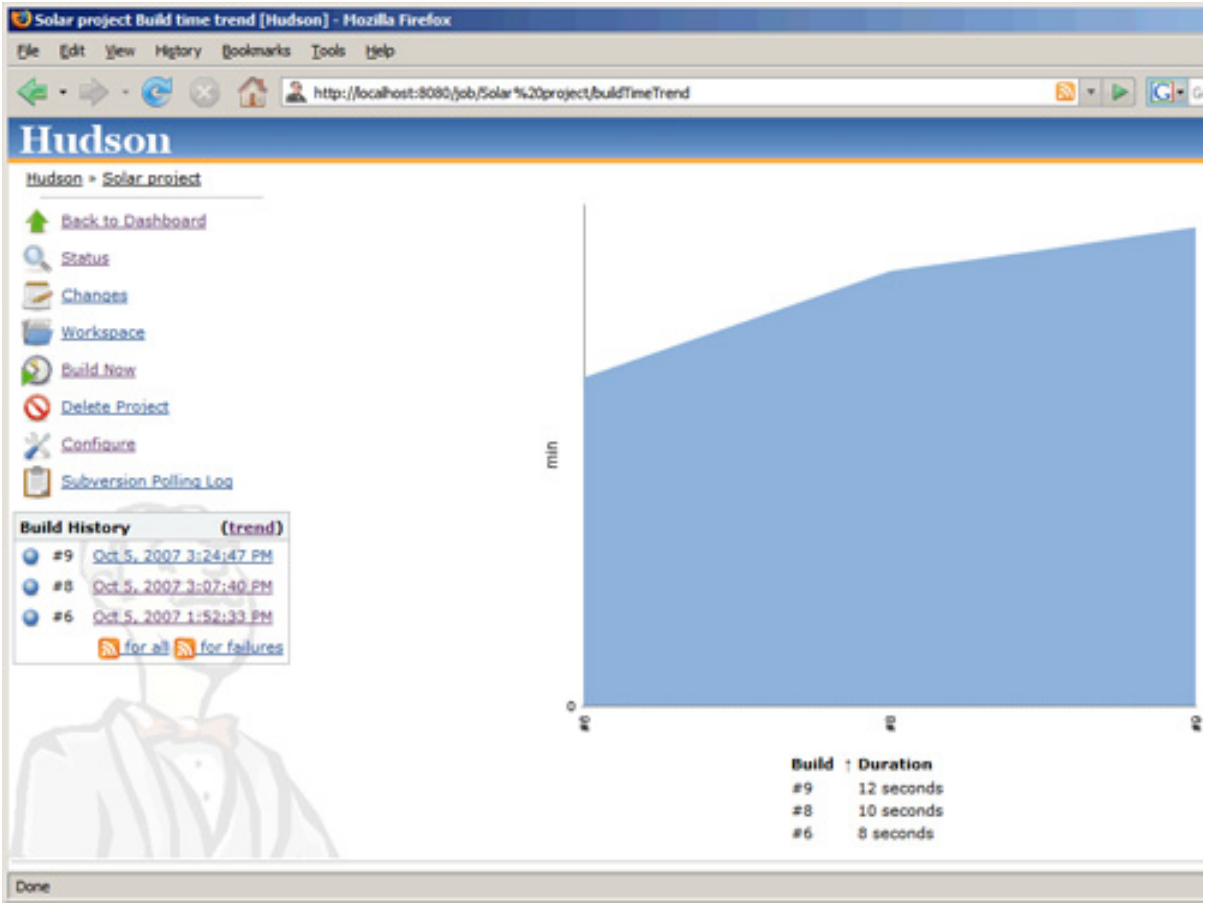
Figure 17. Keep that test trend up!



Check the time

Tests aren't the only aspect that Hudson will trend out of the box. Hudson also trends build execution times, which can be handy in ascertaining build performance. For example, as builds begin to take longer to execute, you may have to start thinking of different ways of running tests, for example, categorizing them for quicker builds.

Figure 18. Hey, those tests take time to run!

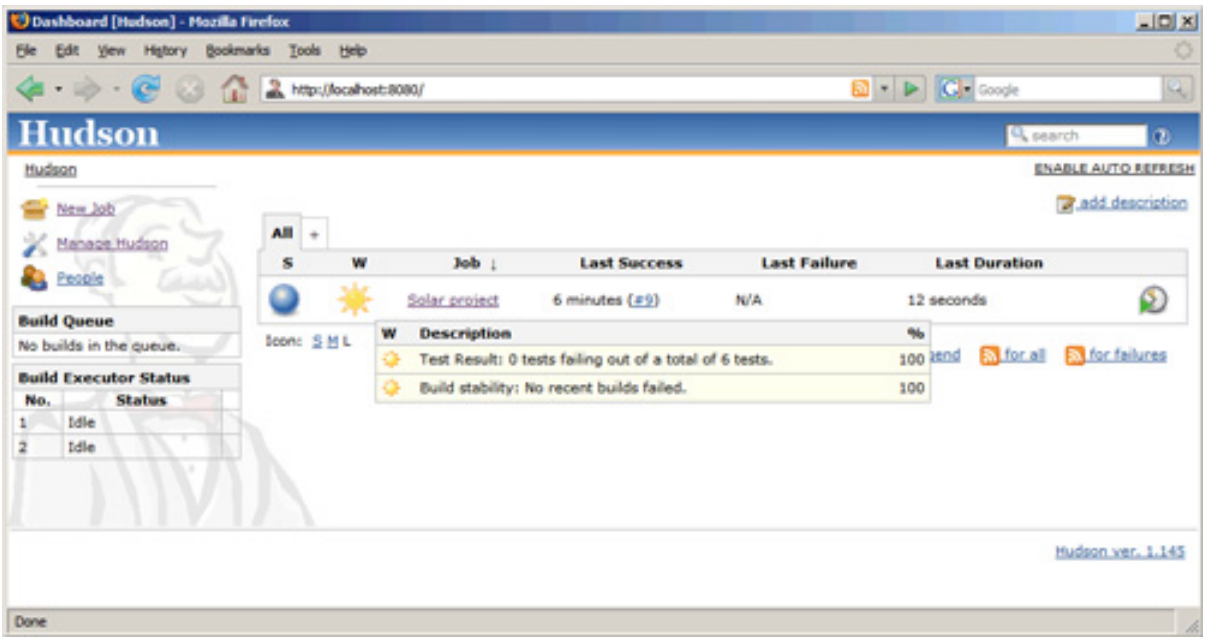


The data shown in Figure 18 clearly indicates that tests add to the overall time of a build — compiling those tests takes time, but you'll find that it is a much smaller amount of time compared to running them.

Report card: A+

Speaking of trending, if you go back to Hudson's homepage, you can see a list of project's that Hudson is currently managing. Various data items are found in its project table. What is particularly interesting is the second icon in the W column: an intuitive visual clue corresponding to the overall health of a monitored project. In your case, as shown in Figure 19, the icon is a bright sun indicating that tests are passing, and so are all builds:

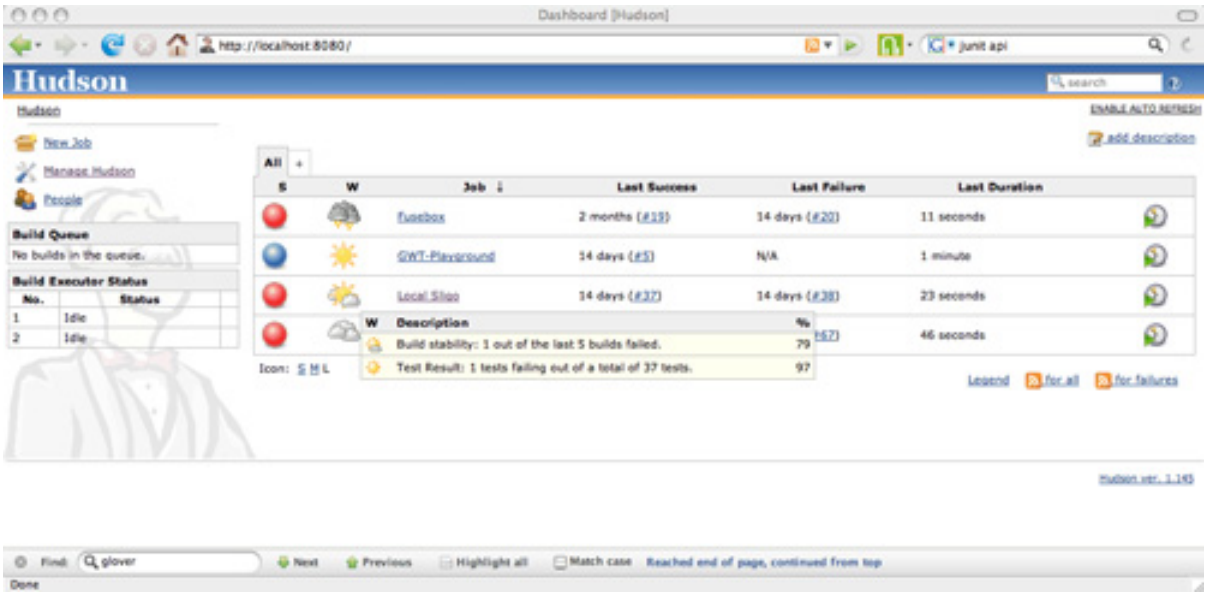
Figure 19. Sunny days lie ahead



Alternate report card: D-

Just like in real life, not all days are bright and sunny; when things are going bad, for example, projects aren't building or tests are failing, Hudson reports a much different picture, as shown in Figure 20:

Figure 20. Watch out for storm clouds



If you look closely, you'll see the pop-up indicating that one particular project has a build stability of 79% and a test stability of 97%, which amounts to a partly cloudy day. The top project is really having a bad day though — it's practically a hurricane!

Section 11. Hudson's plug-ins

The build you defined at the start of this project runs software inspectors FindBugs and PMD to analyze code and report violations. These are excellent tasks to incorporate into a CI process, providing you with a detailed window into your code base. Running them often means you can begin to trend their results, much like the test and build-time trends you saw earlier.

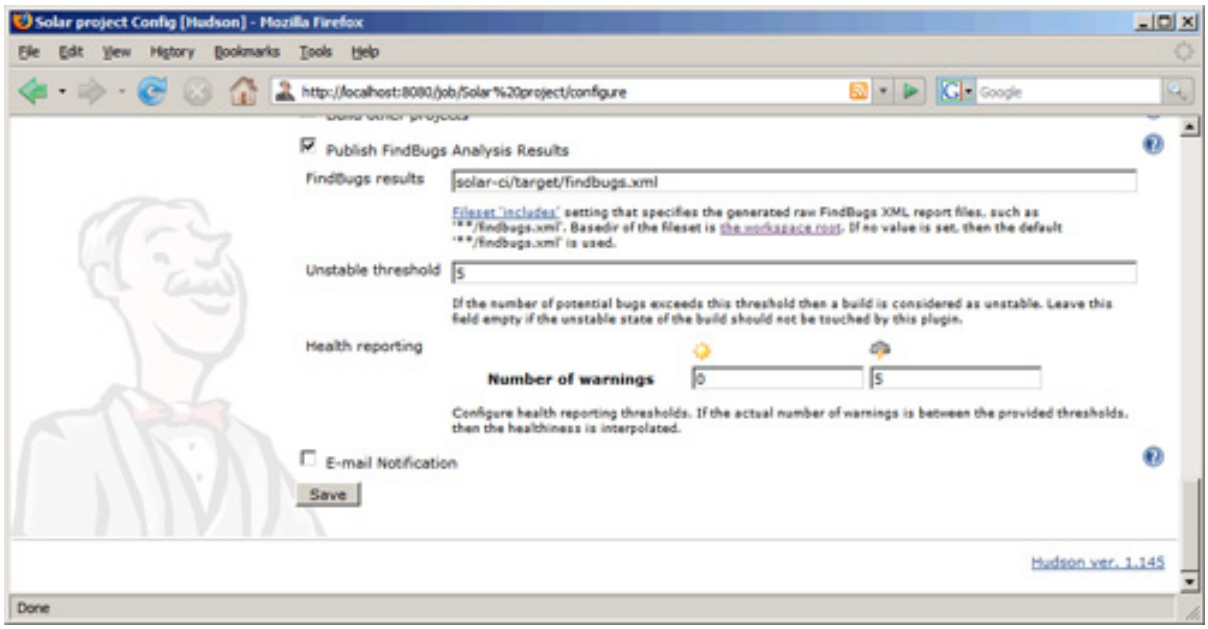
One of the more interesting features of Hudson is its plug-in API, which facilitates creating new features that may be installed on demand. Hudson supports an extensive list of plug-ins, including one for trending FindBugs data as well as one for trending PMD violations (this plug-in can trend other tools as well, including CheckStyle).

Installing a Hudson plug-in is simple, much like the act of installing Hudson itself. All you need to do is download the latest version of the plug-in from Hudson's Web site, click the **Manage Hudson** link from Hudson's homepage, and then click the **Manage Plugins** link, where you'll be presented with a form to upload the plug-in archive file. Once the plug-in has been uploaded, you'll have to restart Hudson.

FindBugs

Once you've downloaded and installed the FindBugs Hudson plug-in, you need to configure it. The configuration of a plug-in happens at the project level, so go to the project's home page and click the **Configure** link, where you'll find some new options specific to FindBugs, as shown in Figure 22:

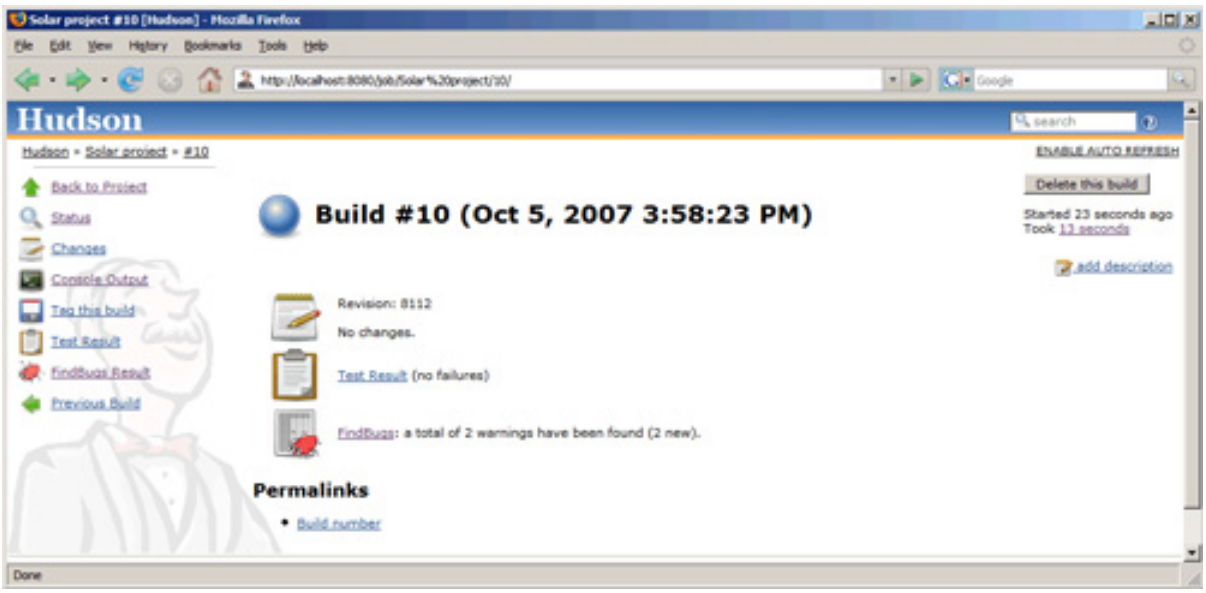
Figure 22. Configuring FindBugs in Hudson



Configuring the FindBugs plug-in is similar to the steps you followed to configure JUnit trending. You have to tell Hudson where to find the XML reports that FindBugs will output (which you took care of when you defined the FindBugs target in the build file). In addition to telling Hudson where to find the reports, you can specify a threshold count. As you can see in Figure 22, if you specify that if there are more than five FindBugs violations, the build will be considered unstable. You can also use thresholds to affect the weather, so to speak, of your build reporting.

Once you click **Save**, you need to update the build preferences of your project, meaning you need to make sure that Hudson knows to run FindBugs as part of a build. Once that is done, you need to force a build (or just wait for someone to check in some changes, I suppose) to start gathering FindBugs data.

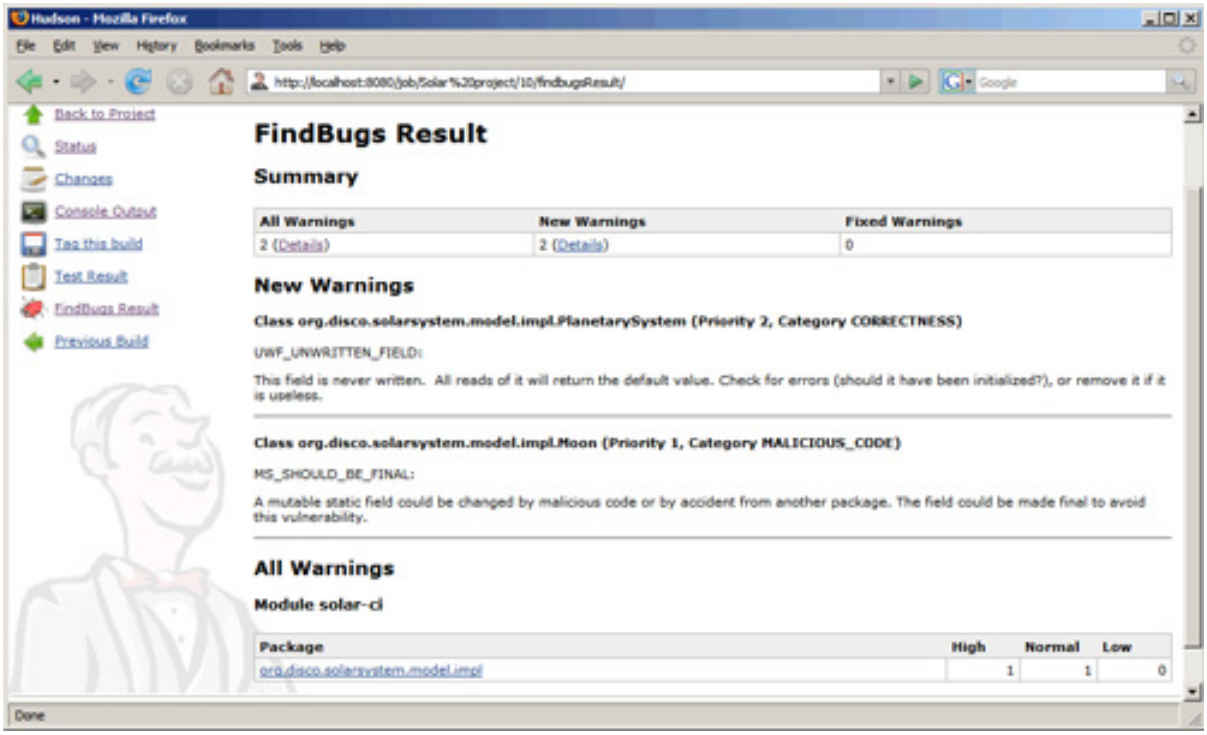
Figure 23. Two warnings were found



Using FindBugs

After that first build, you'll notice some new data points on the build status page, as shown in Figure 23. In this case, Hudson reports that two warnings were found when FindBugs was run. Looks like you have some work to do, starting with digging for more detail, which you can get by clicking the **FindBugs** link on Hudson's report page.

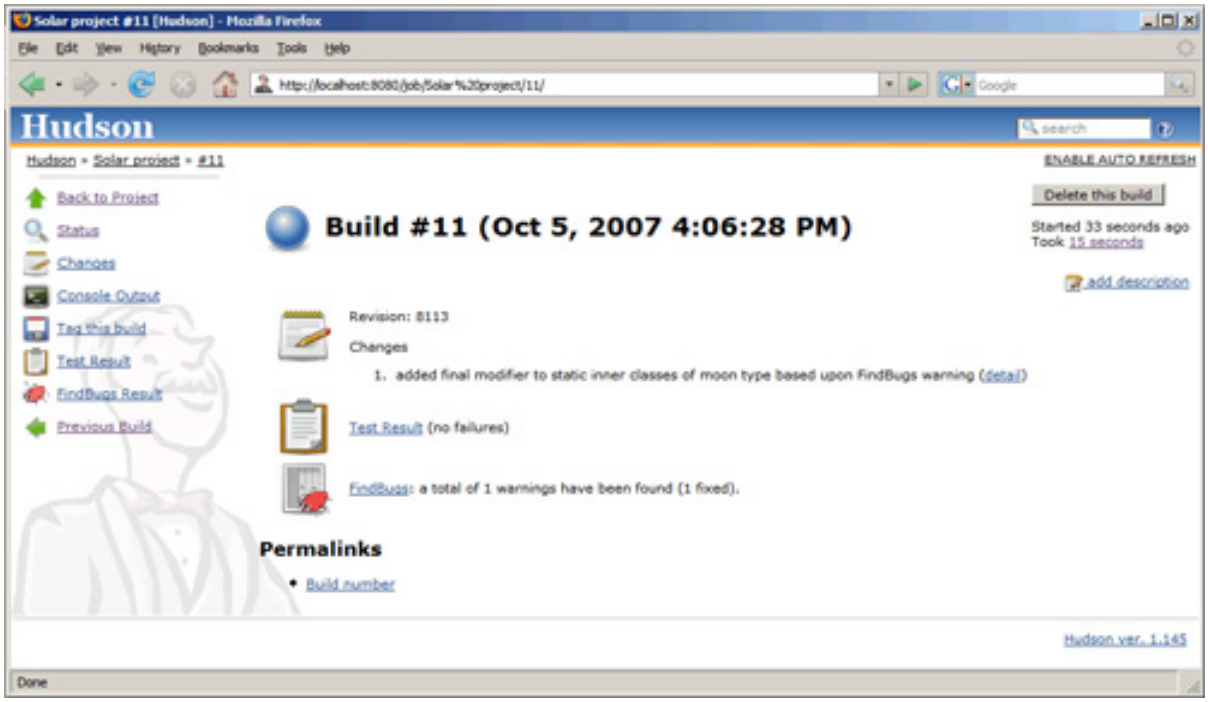
Figure 24. FindBugs violation details



Viewing the details if helpful for fixing the issues, as shown in Figure 24. In this case, you can see that two of your project's classes have violations. One is a Priority 1 violation and the other is a Priority 2. Because you're a high-speed developer, you'll begin to fix these issues right away, which is in accord with the cadence of CI development.

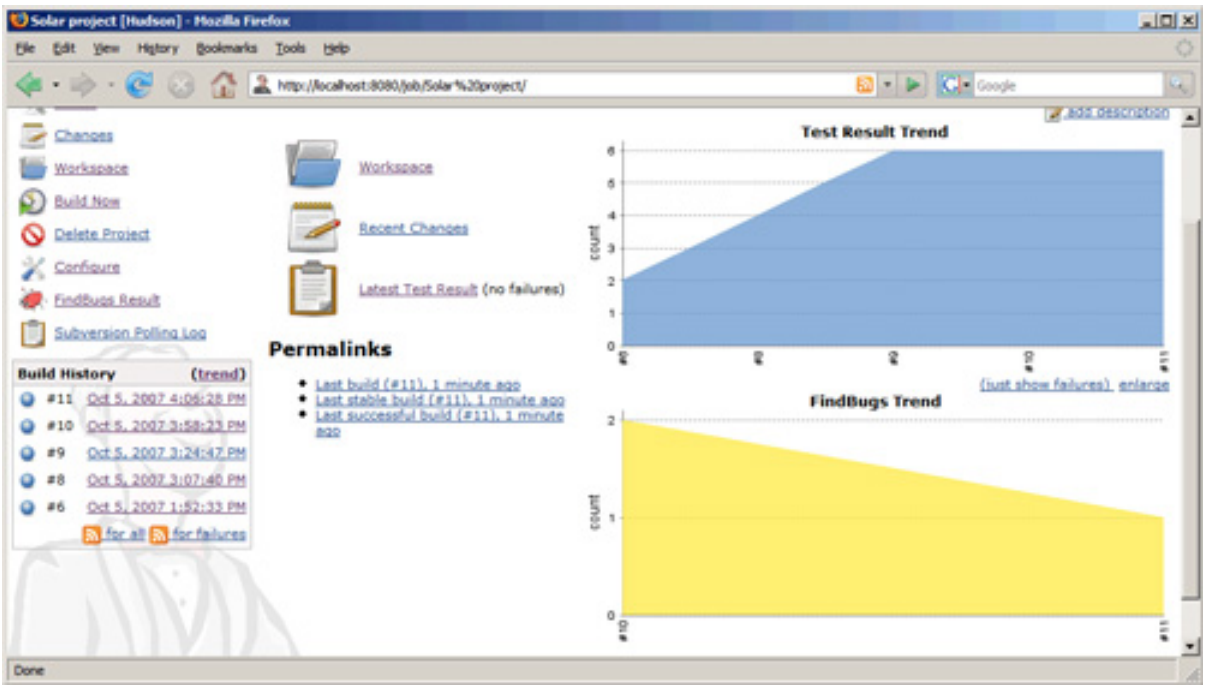
Once you make a correction and check the modifications into Subversion, Hudson will eventually kick off a build. When you click the **Build Status** link after that, you can see that Hudson is correctly pointing out that one bug was fixed, as shown in Figure 25:

Figure 25. One bug down, one to go



Furthermore, the new trend graph in Figure 26, displayed on the project home page, shows that you fixed one of the bugs:

Figure 26. A downward trend for bugs not bad!



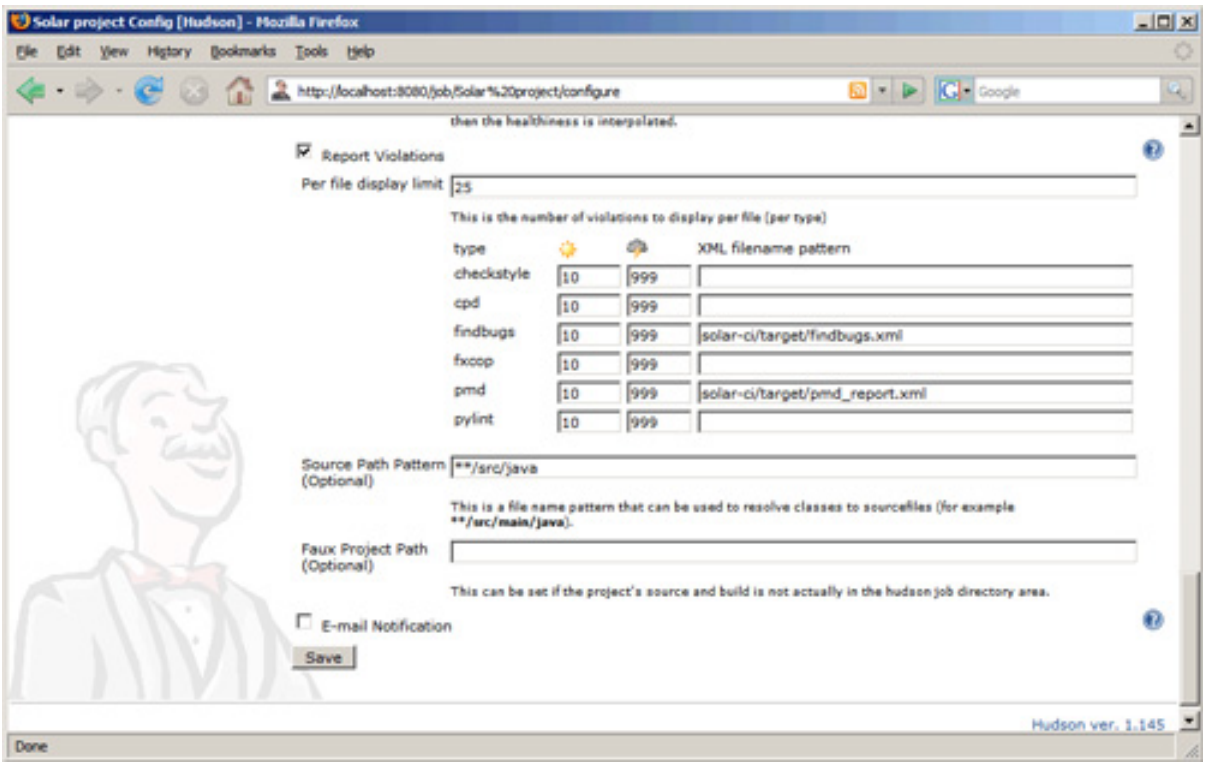
Trending FindBugs violations and using bug thresholds to affect the overall stability of a build are two slick features in Hudson. The price of admission is almost negligible, as I hope I've shown you. Just download the FindBugs plug-in, install it, and start running FindBugs during your project's build.

PMD

FindBugs is by no means the only cool plug-in for Hudson. A few others are related to software inspections, notably one dubbed the Violations plug-in, which trends various tools like PMD, CheckStyle, and FindBugs, to name a few. Given you're already running the FindBugs plug-in, you can use the Violations plug-in to specifically monitor PMD data. When you set up your Ant build earlier in this tutorial, you set up PMD to output an XML report. As you can probably guess, the Violations plug-in uses that report to analyze data for trending.

As you can see in Figure 27, this plug-in works just like other trending tools in Hudson: you simply need to tell Hudson where to find the XML reports of your desired tool. There are other options, of course, but for the basic features just point Hudson to the XML and click **Save**.

Figure 27. Configuring the Violations plug-in

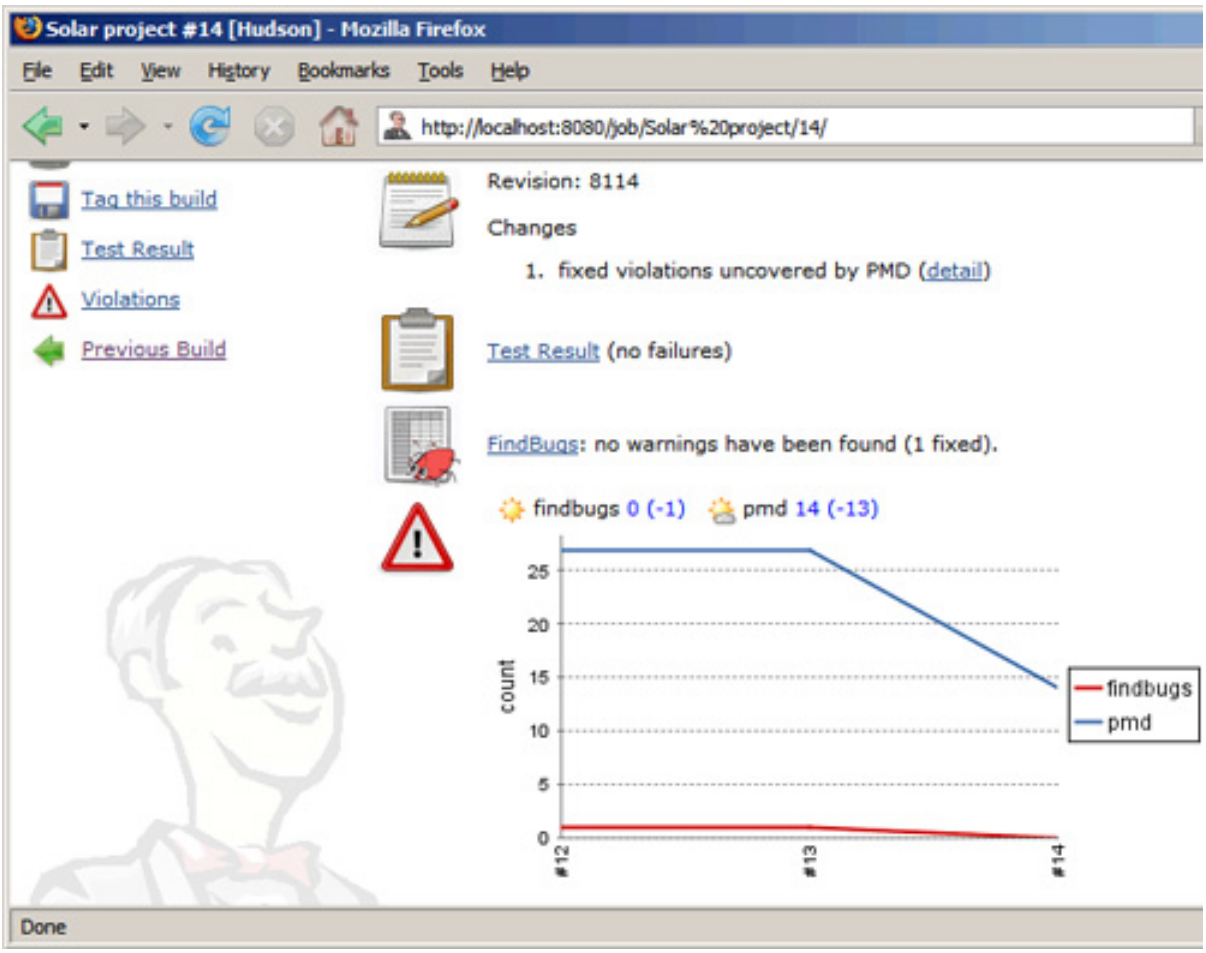


Remember, you need to configure Hudson to actually invoke Ant targets that run these tools; in this case, you've defined a PMD target, so all you need to do is update Hudson to run PMD (as well as FindBugs and your tests which will then force a compilation).

Repair and repeat

After you've forced a build and studied the data, you can start to make corrections like you did with the FindBugs data. As you can see in Figure 28, between builds 13 and 14, you managed to reduce 13 PMD violations. Not bad, don't you think? Already your code is getting better!

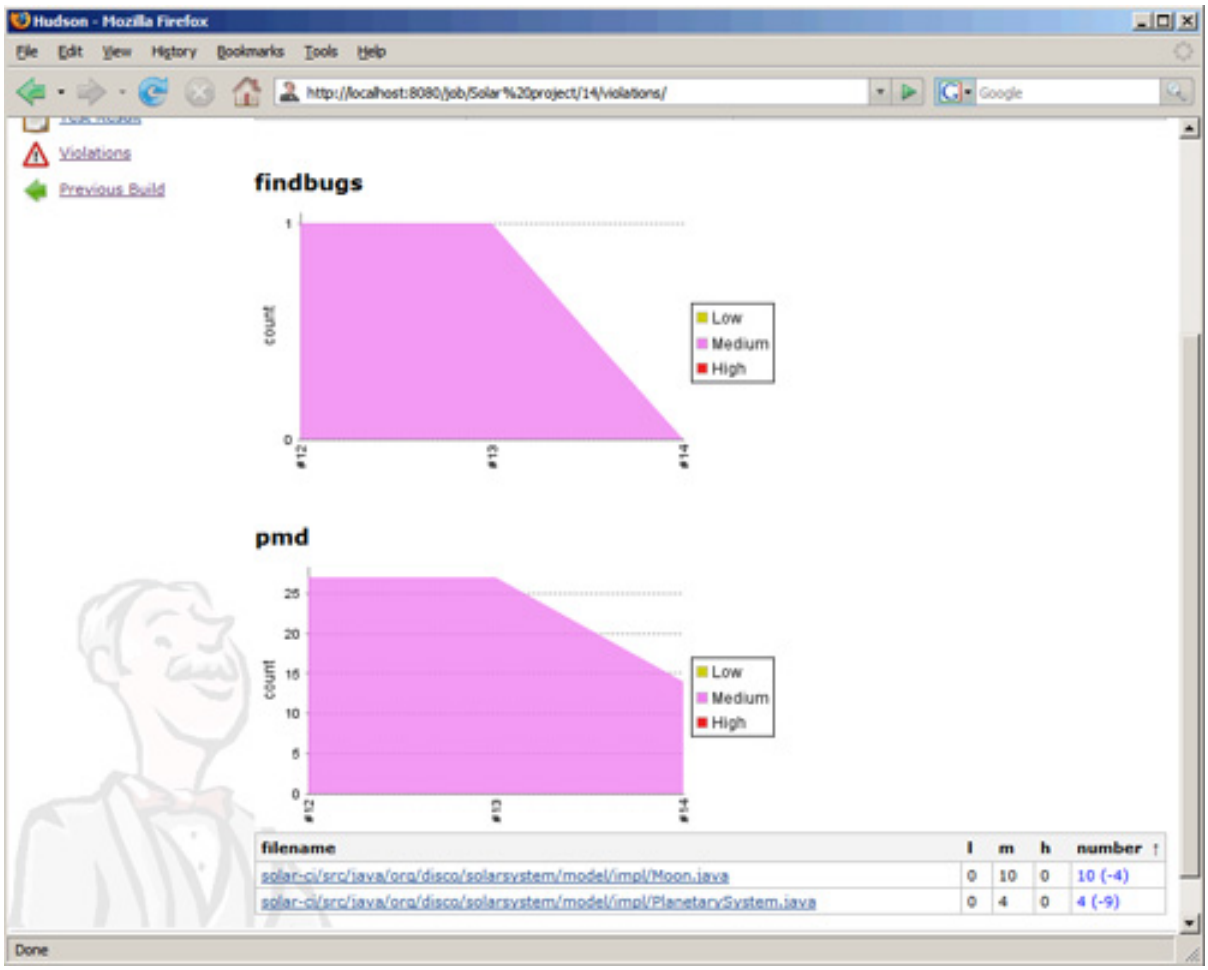
Figure 28. You've fixed some PMD violations!



Trending

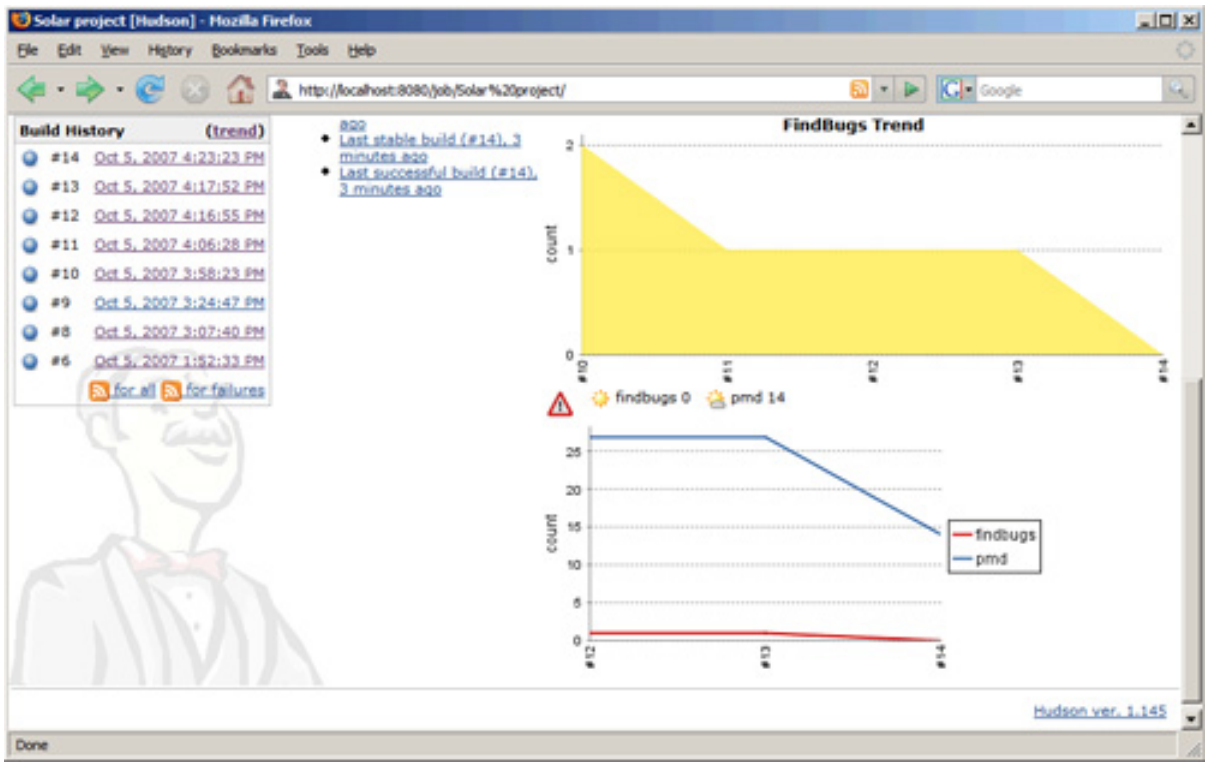
The Violations plug-in supports trending as well, and as you can see in Figure 29, it even trends data by tool; in this case, it is finding both FindBugs data as well as PMD's. By the way, can you see in Figure 29 that you've managed to knock out some more FindBugs violations as well?

Figure 29. More downward trends when it comes to violations



And just like the test trending and FindBugs trending you saw you earlier, the Violations plug-in also displays a historical look at total violations on the project's home page as shown below in Figure 30:

Figure 30. Violations trended over time now that's visibility!



It's clear that Hudson's plug-ins give you unprecedented visibility into your code base; what's more, the ability to trend this data is quite nice as it allows you to set goals, for example, and monitor their progress without having to cycle through pages and pages of custom reports. Hudson, in a sense, creates a project dashboard, which facilitates understanding data quickly.

Section 12. In conclusion

The process of Continuous Integration is about compiling, testing, inspecting, and deploying source code often. The benefit of CI is easy to grasp: by assembling software often, issues can be found early when they are easier and less expensive to resolve.

In this tutorial, you learned about the fundamental aspects of CI. I've introduced some tools for CI (namely Hudson, Ant, and Subversion) and shown you how to set up a CI environment using those tools. The framework you worked with in this tutorial provides a repeatable and reliable build process that runs JUnit tests, as well as software inspections via PMD and FindBugs. You learned how to properly configure the Hudson CI server to poll an SCM repository and run your build process any time a change is detected, and how to use Hudson's plug-ins to trend various

results.

Now go forth and build software rapidly with confidence!

Resources

Learn

- " *Automation for the people*: Continuous Inspection" (Paul Duvall, developerWorks, August 2006): Looks at how automated inspectors like CheckStyle, JavaNCSS, and CPD enhance the development process and when you should use them.
- " *Automation for the people*: Continuous feedback" (Paul Duvall, developerWorks, November 2006): Examines various feedback mechanisms that you can incorporate into CI systems.
- "Automate your team's build and unit-testing process" (Mark Wilkinson, developerWorks, October 2005): Guides you through the practical issues involved in setting up your own Linux-based build server for Java projects.
- "Don’t belie CI" (Andrew Glover, thediscoblog.com): Continuous Integration is much more than a CI server — in fact, the build process is arguably the more important aspect of CI.
- "Don't focus on the elephant" (Andrew Glover, thediscoblog.com): Compares manual code reviews to software inspections.
- "The Future of Continuous Integration" (Paul Duvall, testearly.com): CI is reactionary because the code has already been committed — so how do you prevent broken code from entering your SCM system in the first place?
- *In pursuit of code quality* (Andrew Glover, developerWorks): See all the articles in Andrew's code quality series, with topics ranging from code metrics to testing frameworks to refactoring.

Get products and technologies

- Hudson: See for yourself how easy it is to get going with this clever CI server.
- JUnit: Write effective developer tests with the standard testing framework for the Java platform.
- FindBugs: Find defects quickly by scanning byte code with FindBugs.
- PMD: Put PMD's hundreds of rules to the test and catch errors in your code.

Discuss

- developerWorks blogs: Get involved in the developerWorks community.

About the author

Andrew Glover

Andrew Glover is president of Stelligent Incorporated, which helps companies address software quality with effective developer testing strategies and continuous integration techniques that enable teams to monitor code quality early and often. Check out Andy's blog for a list of his published work.