**Introduction to CI/CD**

In a continuous Integration process, code that has been checked into a source code repository can be automatically checked out, built, tested in a variety of ways, and published to a repository. For continuous integration to work, you need a CI server like Jenkins, which is able to monitor your source code repository for new changes and respond in configurable ways.

Take a Java application built using Maven as an example. On detecting code changes, your CI server could respond by executing a mvn clean install. In a typical Maven build configuration, it would execute a fresh set of unit tests as part of the build command. While the source code was being built, the server could execute any number of additional actions:

* Merge your feature branch back into your main or master branch once the committed code passed the unit test.
* Execute static code analysis, such as code coverage, code complexity, checks for common bugs, etc.
* Publish your build artifacts to a repository, such as [Artifactory](https://www.jfrog.com/artifactory/" \t "_blank) or [Sonatype Nexus](https://www.sonatype.com/nexus-lifecycle" \t "_blank)
* Deploy your application to an integration test environment
* Execute integration tests
* Deploy your application to a performance test environment
* Execute a load test against your application
* Deploy your application to a User Acceptance Testing Environment (UAT)
* Deploy your application to production

These steps are all types of activities that you might perform as part of a CI/CD process. CI typically encompasses the building-and-testing phases of the development lifecycle, whereas CD extends that process to deploying a build artifact to a server for testing. In some environments, CD goes all the way to production.

Continuous Integration is typically done using a tool like Jenkins, Bamboo, or TeamCity, which orchestrates your build steps into an integration pipeline. Jenkins is probably the most popular CI/CD product, and it pairs well with Docker.

**Download and install Jenkins**

[Jenkins](https://jenkins.io/doc/) is a continuous integration server and more. It consists of an automation engine and a plugin ecosystem that supports continuous integration, automated testing, and continuous delivery. You customize the delivery pipeline depending on your need.

There are many ways to run Jenkins:

1. Download a WAR file and install it on a servlet container on your local computer.
2. Setup a virtual machine in a public cloud like [AWS](https://turbonomic.com/blog/on-technology/devs-are-from-venus-ops-are-from-mars-containers-amazon-elastic-container-service-ecs-part-i/) and host Jenkins there.
3. Leverage a Jenkins cloud provider such as [CloudBees](http://www.infoworld.com/article/3046038/application-development/why-jenkins-is-becoming-the-engine-of-devops.html" \t "_blank).
4. Setup Jenkins in a test installation using Docker.

I'll show you how to setup both the local install and the D

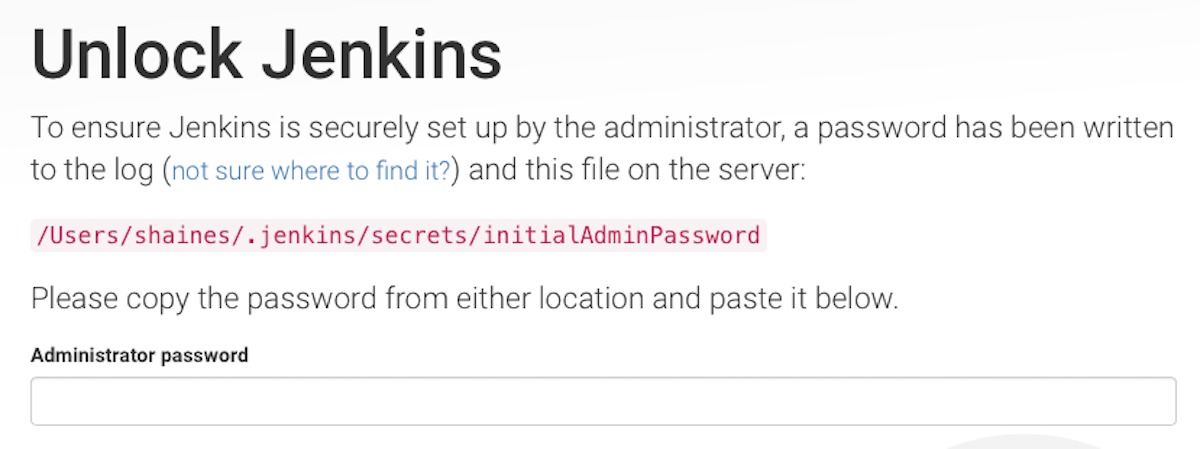
### Download and install Jenkins locally

Start by [downloading Jenkins](https://jenkins.io/) and selecting the Long-Term Support (LTS) release from the Jenkins homepage. Because I'm on a Mac, the install automatically downloaded a pkg file, which placed a jenkins.war in my Application/Jenkins folder. The WAR file can be deployed to any servlet container.

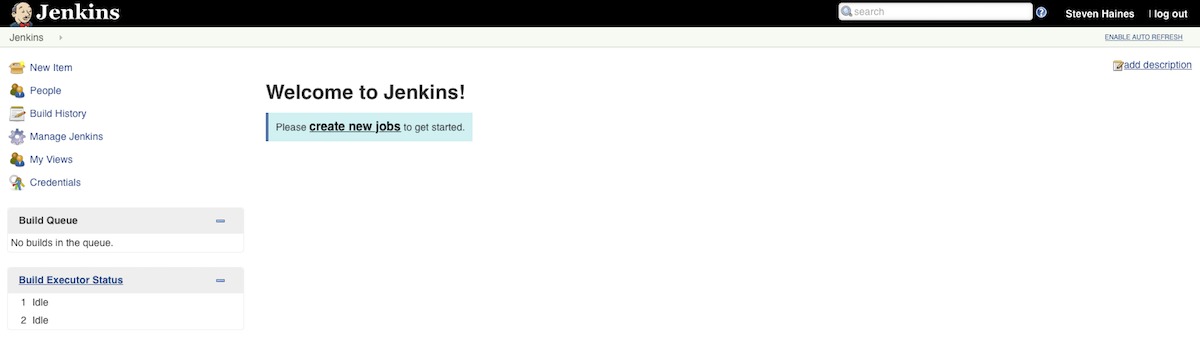
You'll also want to download and install [Apache Tomcat](http://tomcat.apache.org/). As of this writing the most current version of Tomcat is is 8.5.4, but you should be able to run any recent version. Download the zip or tar.gz file and decompress it to your hard drive. Copy the jenkins.war file to Tomcat's webapps folder and then run the bin/startup.sh or bin/startup.bat file. You can test that it is running by opening your browser to: http://localhost:8080.

To start Jenkins, open a browser to the URL: http://localhost:8080/jenkins.

You should get a screen that looks like Figure 1.

Next, Jenkins creates an administration password and writes that both to Tomcat's logs/catalina.out log file and to the following home directory: .jenkins/secrets/initialAdminPassword. Retrieve the password, enter it in Administration password form element (shown in Figure 1), and press **Continue**. You'll be prompted to either install suggested plugins or select plugins to install. For now I recommend installing the suggested plugins.

Now you'll be prompted to create an admin user. Enter your admin user information and press **Save and Finish**. Finally, click **Start using Jenkins**. You'll now see the Jenkins homepage, as shown in Figure 2.

Steven Haines

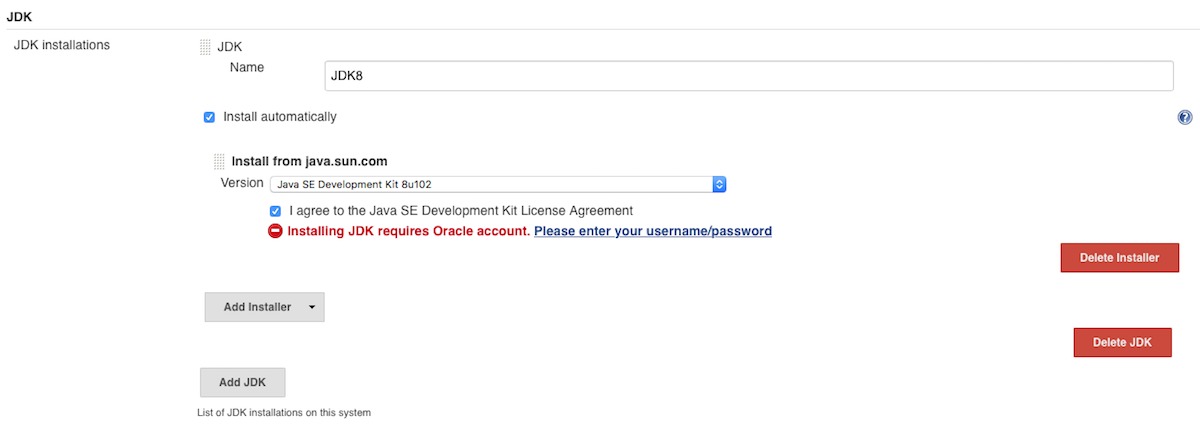
*Figure 2. Screenshot of the Jenkins homepage*

### Configure the example app with Maven

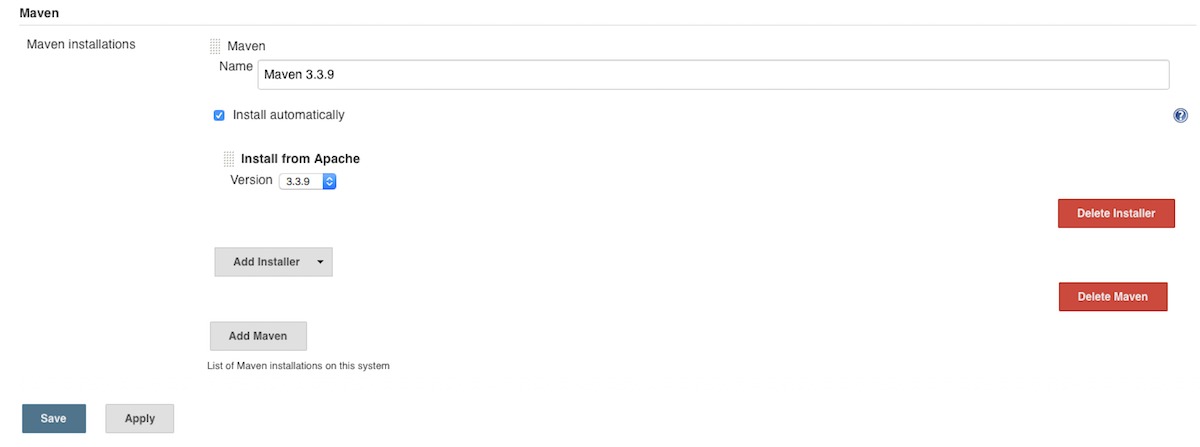
Before we can use Jenkins to build a Java web project with Maven, we need to setup both of these technologies. Under-the-hood, Jenkins will

checkout source code from a source code repository to a local directory and execute the Maven targets that you specify. For that to work, you need to install one or more versions of Maven, tell Jenkins where they're installed, and configure the version of Maven that you want Jenkins to use when building your application.

From the Jenkins dashboard, click **Manage Jenkins** and choose **Global Tool Configuration**. The first thing we'll do is configure a JDK. Under the JDK section, click **Add JDK**, give it a name (mine is "JDK8"), and leave the default **Install from java.sun.com** checked. Accept the Oracle license agreement, then click the "Please enter your username/password" link. Enter your Oracle username and password and press **Close**. You'll be presented with a screen similar to Figure 3.

*Figure 3. Configuring the JDK in Jenkins*

Click **Apply** to save your work, then scroll down to the Maven section and click **Add Maven**. Enter a name for Maven (mine is "Maven 3.3.9"), leave "Install Automatically" and "Install from Apache" checked. Click **Save** when you're ready. You should be presented with a screen similar to Figure 4.

Steven Haines

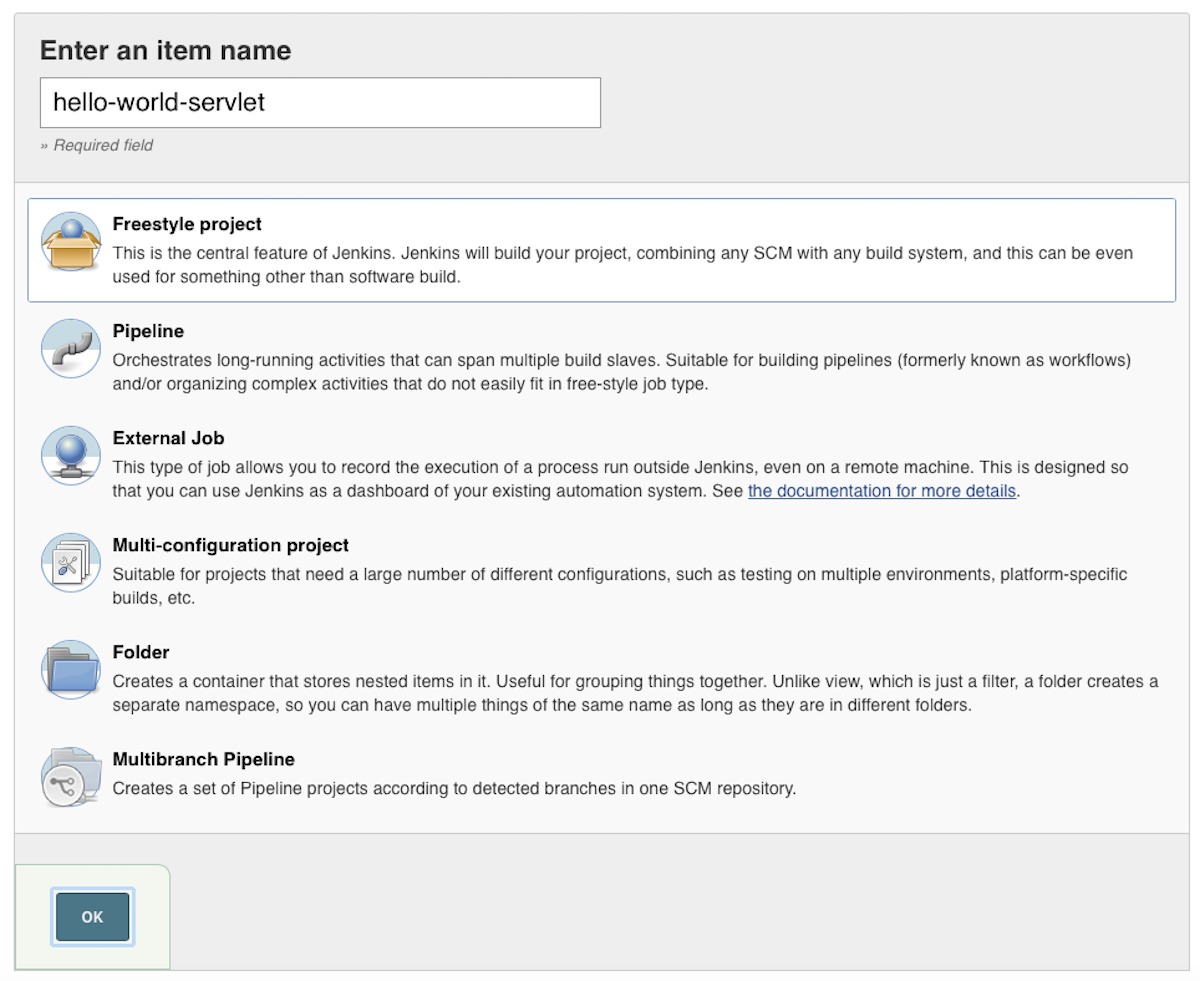
*Figure 4. Configuring Maven in Jenkins*

Git comes preconfigured with Jenkins, so you should now have all the tools installed that you need to checkout and build a Java project from Git with Maven.

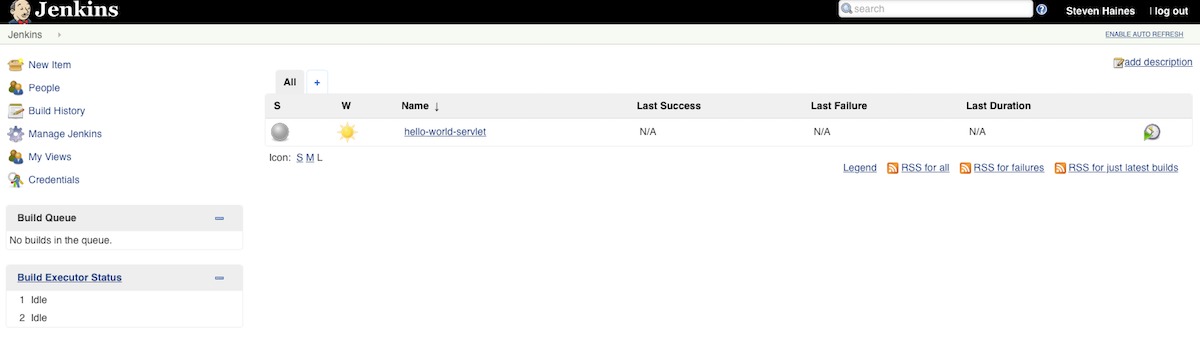
## Setup Jenkins CI for a Java web app

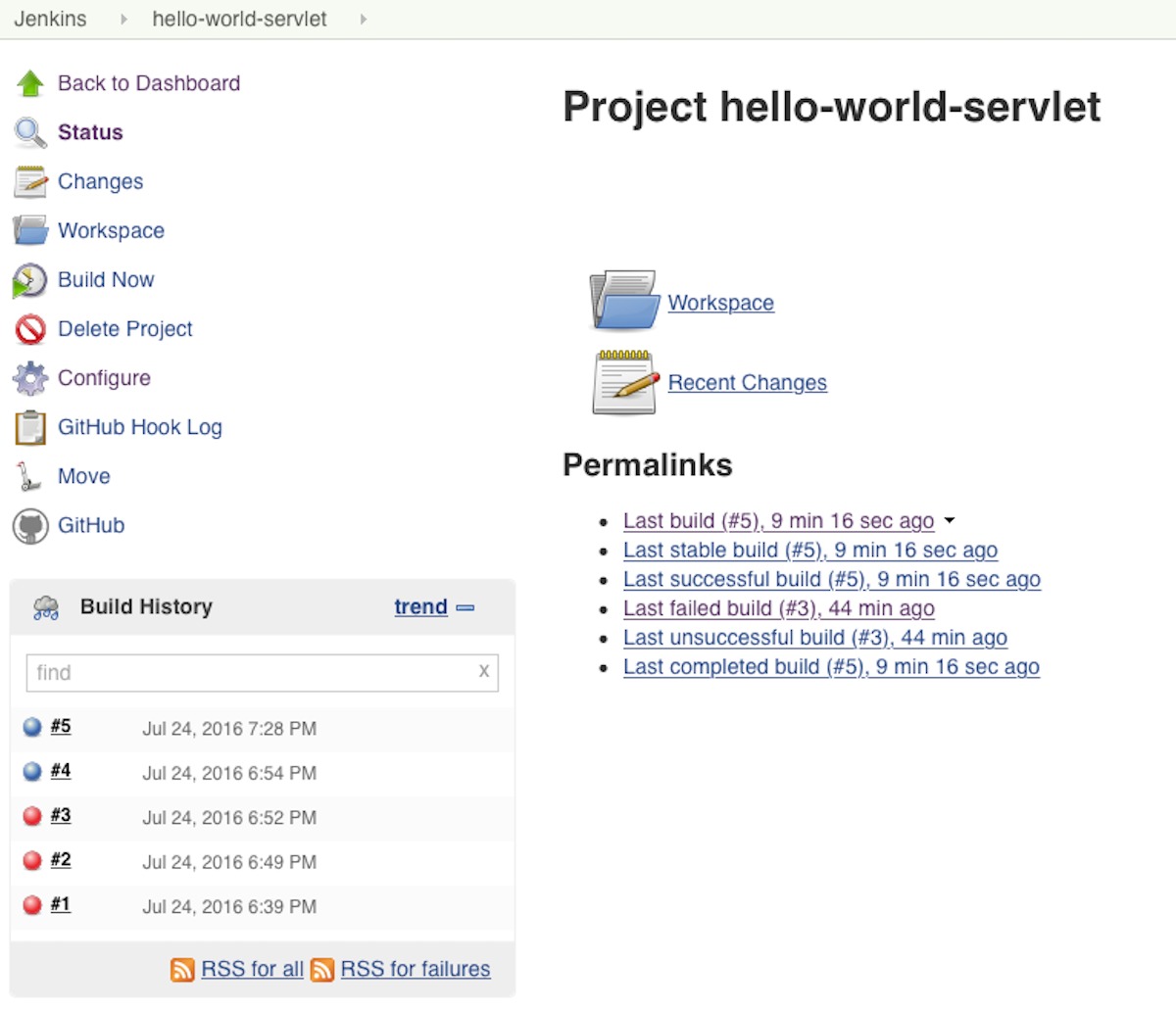
Next we'll setup a simple Java web application job in Jenkins. Because the application isn't important for this tutorial, we'll use my simple [Hello, World Servlet](https://github.com/ligado/hello-world-servlet) example app, which I've hosted on GitHub.

In order to test Jenkins you'll need to be able to commit changes to a source code repository, so you should create that repository now. On the Jenkins homepage, click the **Create new jobs** button and enter the name of your project. You'll be asked to choose the project type, as shown in Figure 5.

We'll choose the Freestyle project type for this project, but you should be aware of your options:

* Freestyle project: This most common type of project allows you to monitor a source code repository and use any build system, such as Maven and Ant.
* Pipeline: Choose this project type for complicated projects with moving parts that you need to coordinate across multiple build slaves.
* External job: Use this to configure an automated external job that you want to track in Jenkins as part of your build.
* Multi-configuration project: This is the job type for projects that require different configurations for different environments, such as production, staging, and test.
* Folder: When you have a complicated build then you might want to organize things into folders, each with their own distinct namespace.
* Multi-branch pipeline: automatically create a set of pipeline projects, based on the code branches that are defined in your source code repository
* Enter a project name, in this case "hello-world-servlet", and choose "OK". Next, choose **GitHub project**, then enter the GitHub URL of your project: https://github.com/ligado/hello-world-servlet.
* Under Source Code Management, choose **Git** and enter the same project URL.
* In the Build Triggers section, choose **Build when a change is pushed to GitHub** so that Jenkins will build your code anytime you push a change to GitHub.
* In the Build section, add a new build step, choose **Invoke top-level Maven targets**, choose the Maven instance that you configured previously (such as "Maven 3.3.9") and enter **clean install** in the goals field. Leave the Post-build Actions empty for now. When you're finished, press **Save**.
* When you return to the dashboard you should see a screen similar to Figure 6.

To test your configuration, press the **Build Now** button next to the hello-world-servlet project. You should see a build executed successfully in the Build History on the left-hand side of the project page, shown in Figure 7.

To see exactly what happened, click on the build and then click **Console Output**, which will show you all the steps that Jenkins performed and their results. The console output is below. [PREVIOUS](https://www.infoworld.com/article/3123117/open-source-java-projects-jenkins-with-docker-part-1.html) [1](https://www.infoworld.com/article/3123117/open-source-java-projects-jenkins-with-docker-part-1.html) [2](https://www.infoworld.com/article/3123117/open-source-java-projects-jenkins-with-docker-part-1.html?page=2) [3](https://www.infoworld.com/article/3123117/open-source-java-projects-jenkins-with-docker-part-1.html?page=3) [NEXT](https://www.infoworld.com/article/3123117/open-source-java-projects-jenkins-with-docker-part-1.html?page=3)

#### Listing 1. Console output from a successful project build

**Started** **by** user **Steven** **Haines**

**Building** **in** workspace /**Users**/shaines/.jenkins/workspace/hello-world-servlet

**Cloning** the remote **Git** repository

**Cloning** repository https:*//github.com/ligado/hello-world-servlet*

> git init /**Users**/shaines/.jenkins/workspace/hello-world-servlet *# timeout=10*

**Fetching** upstream changes **from** https:*//github.com/ligado/hello-world-servlet*

> git --version *# timeout=10*

> git -c core.askpass=**true** fetch --tags --progress https:*//github.com/ligado/hello-world-servlet +refs/heads/\*:refs/remotes/origin/\**

> git config remote.origin.url https:*//github.com/ligado/hello-world-servlet # timeout=10*

> git config --add remote.origin.fetch +refs/heads*/\*:refs/remotes/origin/\* # timeout=10*

*> git config remote.origin.url https://github.com/ligado/hello-world-servlet # timeout=10*

*Fetching upstream changes from https://github.com/ligado/hello-world-servlet*

*> git -c core.askpass=true fetch --tags --progress https://github.com/ligado/hello-world-servlet +refs/heads/\*:refs/remotes/origin/\**

*> git rev-parse refs/remotes/origin/master^{commit} # timeout=10*

*> git rev-parse refs/remotes/origin/origin/master^{commit} # timeout=10*

*Checking out Revision a3da5d8d82d11475e5e1fc899871f1f7a975ed07 (refs/r*

From the output shown in Listing 1, the first thing that Jenkins does is check out the source code from GitHub to the directory /Users/shaines/.jenkins/workspace/hello-world-servlet.

You can open a terminal window and navigate to that directory to confirm. In order to check out your project, you can see several command-line Git statements: git init to setup the project, git config to setup the remote repository references, and git checkout to retrieve the source code.

Next, because this is the first time you've run the project build and you didn't already have Maven installed, Jenkins downloaded Maven from Apache (you can see the URL in the console output) and decompressed it to /Users/shaines/.jenkins/tools/hudson.tasks.Maven\_MavenInstallation/Maven\_3.3.9. When we get deeper into Maven, you'll be able to see more of the command-line interface tools installed under the tools folder.

Finally, Jenkins executed your Maven goals with the following command:

Volume 0%

/**Users**/shaines/.jenkins/tools/hudson.tasks.**Maven\_MavenInstallation**/**Maven\_3**.3.9/bin/mvn clean install

In Listing 1, you can see the source getting built, the unit tests executing,

n Listing 1, you can see the source getting built, the unit tests executing, and finally Maven publishing the build artifact (helloworld.war) to a local machine's local Maven repository.

### Troubleshooting build failures in Jenkins

What if a build fails? In that case you can retrace your steps to troubleshoot the failure. Just click on the failed build and choose **Console Output**. As you can see from the console output from my first failed build, I initially forgot to install Maven.

Steven Haines

*Figure 8. A failed build alert in Jenkins*

Fortunately, I received an error message that Jenkins couldn't run Maven (mvn), which alerted me to my oversight!

## CI reports with Jenkins and JUnit

Once you have a working build, you'll want to start gathering information into a set of reports. In the previous section, when you executed mvn clean install, the install goal implicitly ran your unit tests. Unit tests are finely grained tests that validate your application from a class and method perspective. In this section we'll configure Jenkins to generate a report that summarizes the unit test execution. Then I'll show you how to run and report static code analysis reports for things like test coverage, code complexity, and detecting common bugs.

### JUnit test report

Open your **hello-world-servlet** project and click **Configure**. Scroll down to the bottom of the page, find the Post-build Actions section, and click **Add post-build action** and choose Publish JUnit test result report. Enter the following in the Test Reports XMLs text field:

\*\*/target/surefire-reports*/\*.xml*

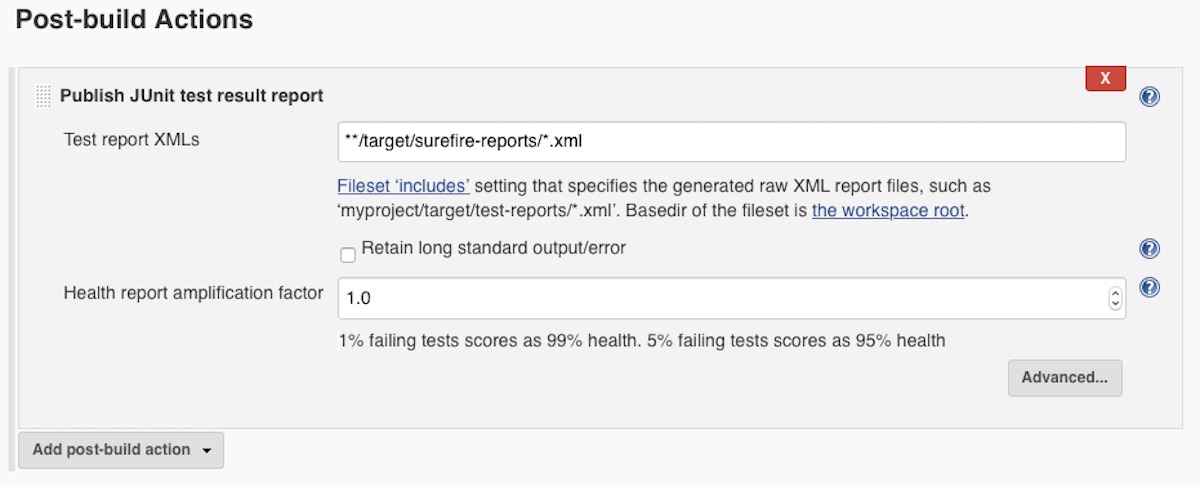
The Test Reports XMLs field points to the directory where your Surefire

tests are published. Maven creates a target directory, a surefire-reports subdirectory, and then publishes a set of XML files summarizing the test results to the new directory. You can navigate to the directory yourself to confirm:

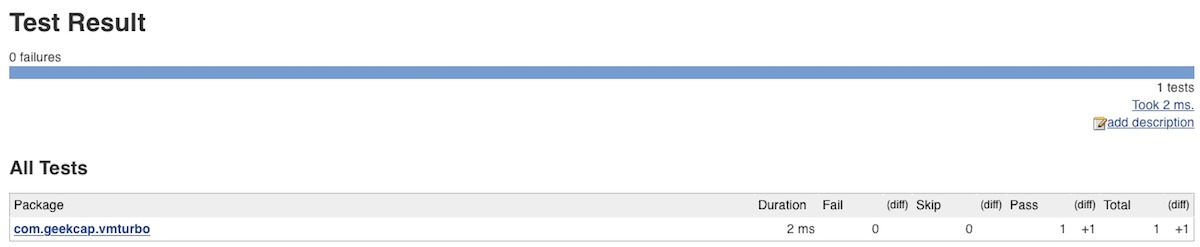
~/.jenkins/workspace/hello-world-servlet/target/surefire-reports

Because this is a simple example with only one test in it, Maven creates a single XML file named TEST-com.geekcap.vmturbo.ThingTest.xml. In a production application you would have dozens or even hundreds of XML files.

When you're finished, press **Save**. You should see something similar to Figure 9.

*Figure 9. Adding a JUnit report post-build action*

From your project page, press the **Build Now** button. When that's complete, click on the new build. You should see a new section at the bottom of the page that reads Test Results (no failures). Click on the link and you should be presented with a screen similar to Figure 10.

Steven Haines

*Figure 10. A JUnit report test summary*

This report shows a summary of all passes and failures and allows you to drill down into packages, classes, and individual tests to see the results.

### The JUnit Plugin

If you're curious about why the new report is available in your new builds, but not retroactively available in previous builds, you'll need to dive further into the inner workings of Jenkins, and specifically of the JUnit plugin.

When you configured a post-build action to get the JUnit test report, you invoked the [JUnit Plugin](https://wiki.jenkins-ci.org/display/JENKINS/JUnit+Plugin). When a build is complete, Jenkins creates a directory for it in its .jenkins/jobs/builds folder. You can navigate to that folder to see all of your build artifacts. Here's the difference that I have between build "1" and build "2" (the before and after for the JUnit report):

$ ls -l 1

total 32

-rw-r----- 1 shaines staff 2341 **Aug** 20 06:35 build.xml

-rw-r----- 1 shaines staff 0 **Aug** 20 06:35 changelog.xml

-rw-r----- 1 shaines staff 8748 **Aug** 20 06:35 log

$ ls -l 2

total 40

-rw-r----- 1 shaines staff 2658 **Aug** 21 16:46 build.xml

-rw-r----- 1 shaines staff 0 **Aug** 21 16:46 changelog.xml

-rw-r----- 1 shaines staff 707 **Aug** 21 16:46 junitResult.xml

-rw-r----- 1 shaines staff 8209 **Aug** 21 16:46 log

The build directory contains a build.xml file that specifies the configuration that was in place when that build was executed, a

changelog.xml file, and a log file that saves the console output from each build. In directory 2 you can see a new junitResult.xml file that contains your results. The JUnit Plugin provides a stylesheet to render the XML file in a more readable fashion.

**Using static code analysis tools with Jenkins**

Static code analysis uses various standard measures to test the overall health of your application. In Jenkins, you can combine several good static code analysis tools for an assessment. In this case, we'll use the following tools and generate a report of their findings:

* [Checkstyle](https://maven.apache.org/plugins/maven-checkstyle-plugin/) checks the style of your code against the official Java code style.
* [FindBugs](http://gleclaire.github.io/findbugs-maven-plugin/) searches for and reports common bugs.
* [PMD](https://maven.apache.org/plugins/maven-pmd-plugin/) validates build rules.
* [Cobertura](http://www.mojohaus.org/cobertura-maven-plugin/) checks the coverage of your unit tests.

You need to do the following in order to use these tools and generate

eports for them in Jenkins:

1. Add reporting plugins to your Maven pom.xml file.
2. Install the Jenkins plugins for these tools.
3. Update your Maven goals to call these plugins.

#### Add post-build actions to publish the static code analysis reports. Step 1. Add the reporting plugins to your Maven pom.xml

I updated the POM file locally and pushed it to my GitHub with sample configurations that define Maven plugins for these tools. Listing 2 is for your reference.

#### Listing 2. pom.xml

**<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd"**>**

**<modelVersion>**4.0.0**</modelVersion>**

**<groupId>**com.geekcap.vmturbo**</groupId>**

**<artifactId>**hello-world-servlet-example**</artifactId>**

**<packaging>**war**</packaging>**

**<version>**1.0-SNAPSHOT**</version>**

**<name>**hello-world-servlet-example Maven Webapp**</name>**

**<url>**http://maven.apache.org**</url>**

**<reporting>**

**<plugins>**

**<plugin>**

**<groupId>**org.codehaus.mojo**</groupId>**

**<artifactId>**cobertura-maven-plugin**</artifactId>**

**<version>**2.7**</version>**

**<configuration>**

**<formats>**

**<format>**xml**</format>**

**</formats>**

**</configuration>**

**</plugin>**

**<plugin>**

**<groupId>**org.apache.maven.plugins**</groupId>**

**<artifactId>**maven-checkstyle-plugin**</artifactId>**

**<version>**2.17**</version>**

**</plugin>**

**<plugin>**

**<groupId>**org.apache.maven.plugins**</groupId>**

**<artifactId>**maven-pmd-plugin**</artifactId>**

**<version>**3.6**</version>**

**<configuration>**

**<linkXref>**true**</linkXref>**

**<sourceEncoding>**utf-8**</sourceEncoding>**

**<minimumTokens>**100**</minimumTokens>**

**<targetJdk>**1.6**</targetJdk>**

*<!--*

*<rulesets>*

*<ruleset>/rulesets/braces.xml</ruleset>*

*</rulesets>*

*<excludes>*

*<exclude>\*\*/\*Bean.java</exclude>*

*<exclude>\*\*/generated/\*.java</exclude>*

*</excludes>*

*<excludeRoots>*

*<excludeRoot>target/generated-sources/stubs</excludeRoot>*

*</excludeRoots>*

*-->*

**</configuration>**

**</plugin>**

**<plugin>**

**<groupId>**org.codehaus.mojo**</groupId>**

**<artifactId>**taglist-maven-plugin**</artifactId>**

**<version>**2.3**</version>**

*<!--*

*<configuration>*

*<tags>*

*<tag>TODO</tag>*

*<tag><todo</tag>*

*<tag>FIXME</tag>*

*<tag>DOCUMENT\_ME</tag>*

*<tag>NOT\_YET\_DOCUMENTED</tag>*

*</tags>*

*</configuration>*

*-->*

**</plugin>**

**<plugin>**

**<groupId>**org.codehaus.mojo**</groupId>**

**<artifactId>**findbugs-maven-plugin**</artifactId>**

**<version>**3.0.4**</version>**

**<configuration>**

**<effort>**Max**</effort>**

**</configuration>**

**</plugin>**

**</plugins>**

**</reporting>**

**<dependencies>**

**<dependency>**

**<groupId>**javax.servlet**</groupId>**

**<artifactId>**servlet-api**</artifactId>**

**<version>**2.5**</version>**

**<scope>**provided**</scope>**

**</dependency>**

**<dependency>**

**<groupId>**junit**</groupId>**

**<artifactId>**junit**</artifactId>**

**<version>**4.12**</version>**

**<scope>**test**</scope>**

**</dependency>**

**</dependencies>**

**<build>**

**<finalName>**helloworld**</finalName>**

**<plugins>**

**<plugin>**

**<groupId>**org.apache.maven.plugins**</groupId>**

**<artifactId>**maven-compiler-plugin**</artifactId>**

**<version>**2.0.2**</version>**

**<configuration>**

**<source>**1.6**</source>**

**<target>**1.6**</target>**

**</configuration>**

**</plugin>**

**<plugin>**

**<groupId>**org.jolokia**</groupId>**

**<artifactId>**docker-maven-plugin**</artifactId>**

**<version>**0.13.4**</version>**

**<configuration>**

**<dockerHost>**tcp://192.168.99.100:2376**</dockerHost>**

**<certPath>**/Users/shaines/.docker/machine/machines/default**</certPath>**

**<useColor>**true**</useColor>**

**<images>**

**<image>**

**<name>**lygado/tomcat-with-my-app:0.1**</name>**

**<alias>**service**</alias>**

**<build>**

**<from>**tomcat**</from>**

*<!--*

*<ports>*

*<port>8080</port>*

*</ports>*

*<entryPoint>*

*<exec>*

*<args>catalina.sh</args>*

*<args>start</args>*

*</exec>*

*</entryPoint>*

*-->*

**<assembly>**

**<mode>**dir**</mode>**

**<basedir>**/usr/local/tomcat/webapps**</basedir>**

**<descriptor>**assembly.xml**</descriptor>**

**</assembly>**

**</build>**

**<run>**

**<ports>**

**<port>**8080:8080**</port>**

**</ports>**

**</run>**

**</image>**

**</images>**

**</configuration>**

**</plugin>**

**</plugins>**

**</build>**

**</project>**

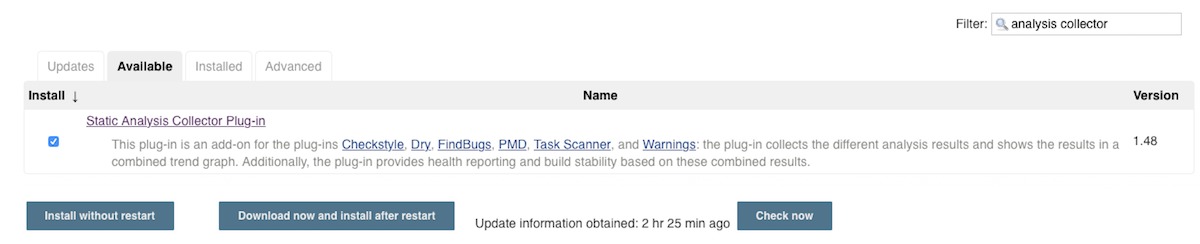
The reporting section adds the four aforementioned static code analysis plugins. Each of these plugins defines Maven goals that can be invoked as follows to generate their respective reports:

mvn clean install checkstyle:checkstyle findbugs:findbugs pmd:pmd pmd:cpd cobertura:cobertura -**Dcobertura**.report.format=xml

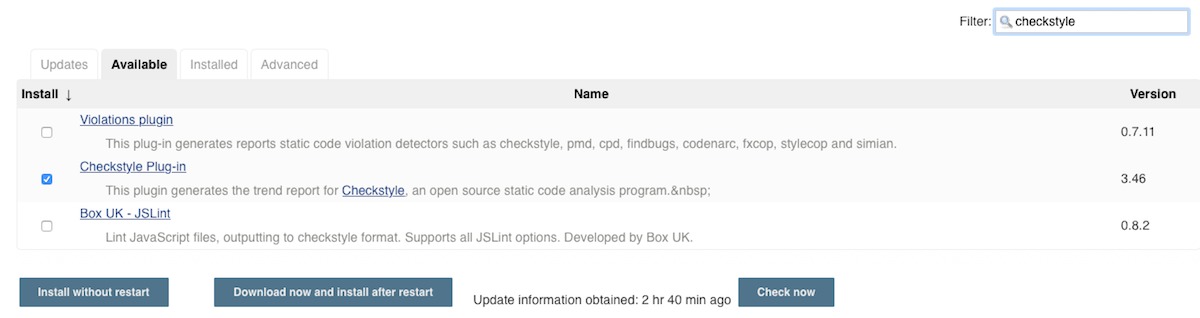
#### Step 2. Install the Jenkins plugins

Next, you'll need to install the Jenkins plugins that will retrieve the generated reports, associate them with a build, and make them available through the Jenkins dashboard.

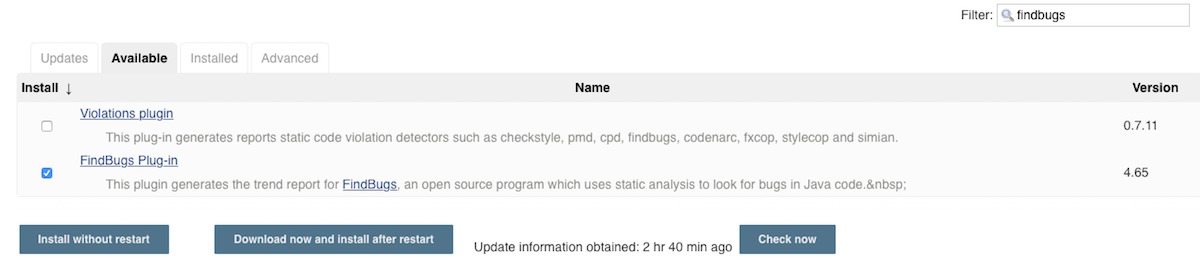
om the Jenkins homepage, click on **Manage Jenkins**, then **Manage Plugins**. Choose the Available tab and search for the following plugins, checking each one as you find it:

Steven Haines

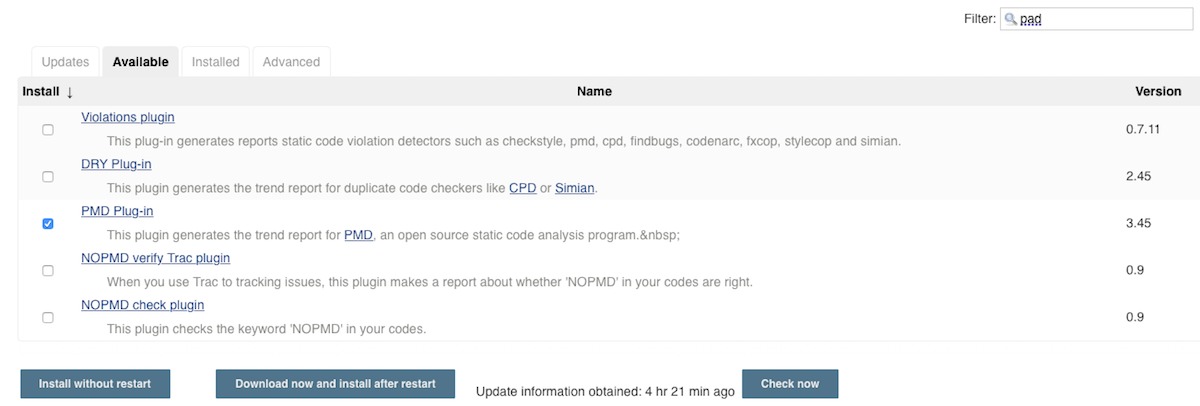
*Figure 11. Installing the Static Analysis Collector plugin*

Steven Haines

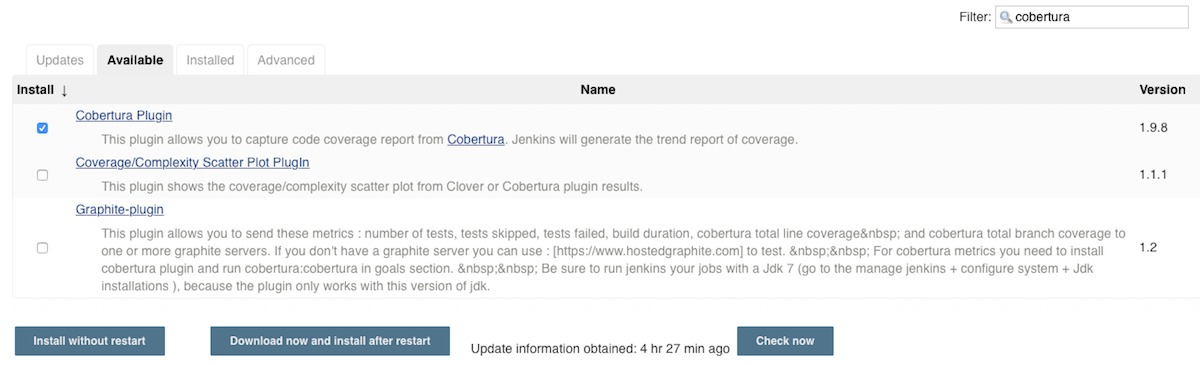
*Figure 12. Installing the Checkstyle plugin*

Steven Haines

*Figure 13. Installing the FindBugs plugin*

Steven Haines

*Figure 14. Installing the PMD plugin*

Supplied by author for JavaWorld.

*Figure 15. Installing the Cobertura plugin*

#### Step 3. Update the project build config

Next you'll update your project build configuration. Navigate to your project page and click the **Configure** button. Scroll down to the Build section and update your Maven goals to the following:

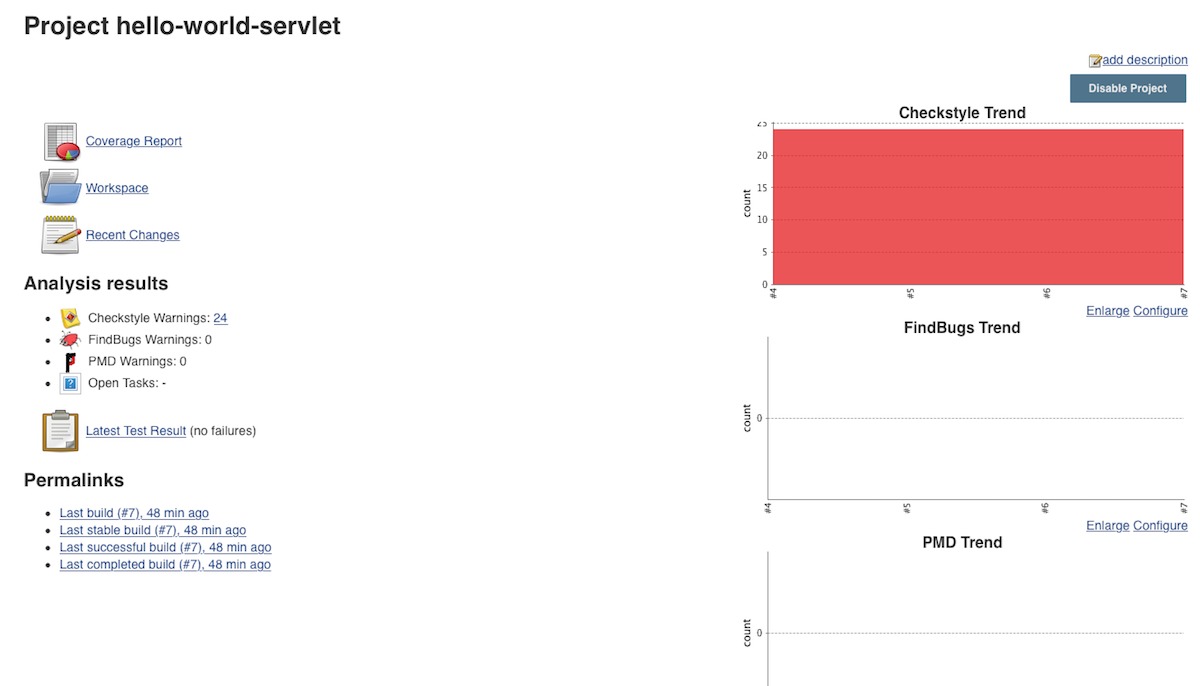
clean install checkstyle:checkstyle findbugs:findbugs pmd:pmd pmd:cpd cobertura:cobertura -**Dcobertura**.report.format=xml

#### Step 4. Add your post-build actions

Finally, scroll to the post-build actions section and do the following:

* Publish Checkstyle analysis results: Enter the following into the Checkstyle results text field, so that the Checkstyle plugin can find your test results: \*\*/checkstyle-result.xml
* Publish FindBugs analysis results: Enter the following into the FindBugs results text field, so that the FindBugs plugin can find your test results: \*\*/findbugsXml.xml
* Publish PMD analysis results: Enter the following into the PMD results text field, so that the PMD plugin can find your test results: \*\*/pmd.xml
* Publish Cobertura Coverage Report: Enter the following into the Cobertura.xml report pattern text field, so that the Cobertura plugin can find your test results: \*\*/target/site/cobertura/coverage.xml
* Publish the combined analysis results.

Save your project and click **Build Now**. As shown in Figure 16, you should now see that reports are being collected and trends aggregated for Checkstyle, FindBugs, PMD, and Cobertura.

Steven Haines

*Figure 16. New project homepage*

You can click through the various reports and review the information to learn more about the health of your application.

## Conclusion

In this tutorial you've taken your first steps with Jenkins CI. To start, you integrated Jenkins with GitHub and learned how to poll for source code changes, checkout a project, build the project from a Maven command, and publish the project's unit test results to the console. You then added a set of static code analysis tools to your project's Maven pom.xml file, installed the five Jenkins plugins, and published them to the Jenkins build. Finally, you updated your Maven build goals, and added post-build actions to generate a set of static code analysis reports.