

Continuous integration with Jenkins

Setup a continuous integration process to build and test a Java web app with Jenkins and JUnit

Looking back at how software was built and deployed even 15 years ago, it seems surprising that our applications actually worked. In those days, a software development lifecycle consisted of running builds on a local machine, manually copying the artifacts to a staging server, and manually testing each application through multiple iterations. When the dev team was satisfied with the build, we would manually deploy the application into production. The most consistent thing about this style of development was inconsistency--in process and in results.

Over a decade ago, agile developers began to embrace and promote test-driven development and [continuous integration](#) (CI). With these techniques we could automatically build source code whenever a developer checked it into a source repository, executing an exhaustive unit test suite to ensure that an application was working properly. Many test-driven developers also started doing integration testing and performance testing in a secondary CI process.

With continuous integration we could detect errors more quickly and release code much faster than we had done in previous years. It's no exaggeration to say that CI tamed the "build" side of the build-and-deploy cycle. These days many dev teams have moved beyond CI to CD, which stands for either [continuous delivery](#) or [continuous deployment](#). Whatever the designation, CD is a process that moves software from code check-in to staging, or even production deployments.

This installment of **Open source Java projects** introduces continuous integration with Jenkins, a leading automation server for CI/CD. We'll begin with an overview of the CI and CD process, then setup a Java web project using Maven and Jenkins. You'll learn how to build and unit test the project in Jenkins with JUnit, as well as how to troubleshoot build failures. You'll also install and run a handful of popular Jenkins plugins for static code analysis testing and reporting.

Table of Contents

- [Introduction to CI/CD](#)
- [Download and install Jenkins](#)
- [Setup Jenkins CI for a Java web app](#)
- [CI reports with Jenkins and JUnit](#)
- [Using static code analysis tools with Jenkins](#)

Show More

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](#) or [Sonatype Nexus](#)
- 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.

[[Learn Java from beginning concepts to advanced design patterns in this comprehensive 12-part course!](#)]

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](#) 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](#) and host Jenkins there.

3. Leverage a Jenkins cloud provider such as [CloudBees](#).
4. Setup Jenkins in a test installation using Docker.

I'll show you how to setup both the local install and the Docker test installation.

Download and install Jenkins locally

Start by [downloading Jenkins](#) 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](#). As of this writing the most current version of Tomcat 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.

Unlock Jenkins

To ensure Jenkins is securely set up by the administrator, a password has been written to the log ([not sure where to find it?](#)) and this file on the server:

```
/Users/shaines/.jenkins/secrets/initialAdminPassword
```

Please copy the password from either location and paste it below.

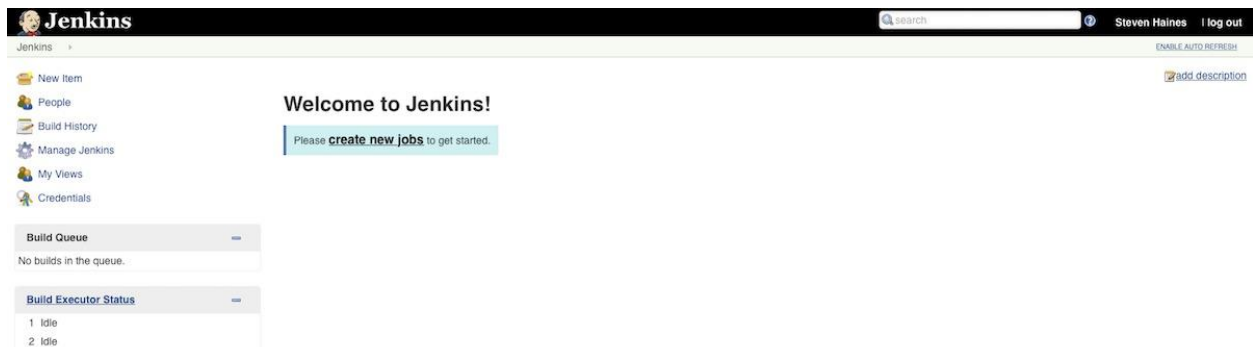
Administrator password

Steven Haines

Figure 1. The Jenkins start screen

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.



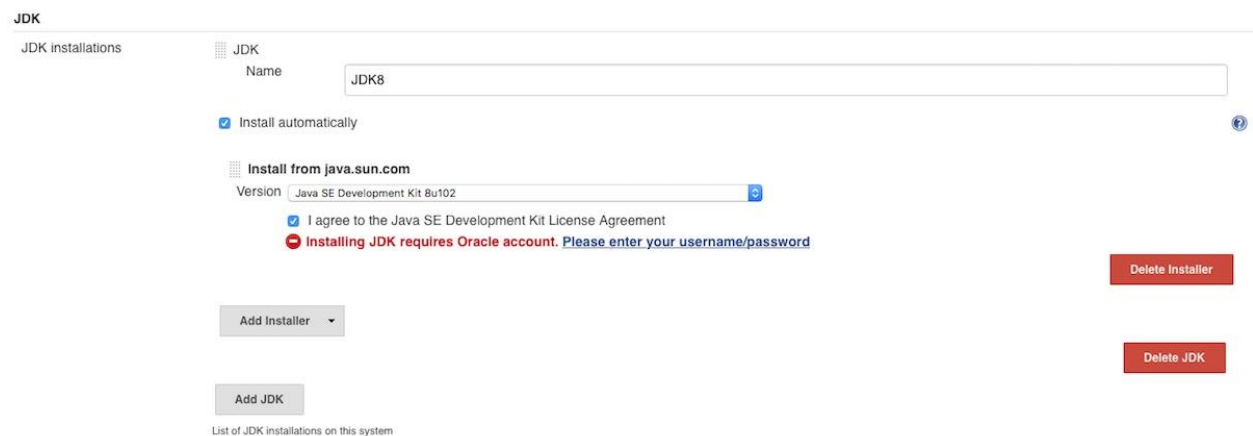
Steven Haines

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.



Steven Haines

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.

Maven

Maven installations

Name
Maven 3.3.9

☒ Install automatically

☒ Install from Apache

Version: 3.3.9

Add Installer

Add Maven

Delete Installer

Delete Maven

List of Maven installations on this system

Save Apply

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.

Install Jenkins in a Docker container

If you don't want to install Jenkins on your local machine, you have the option of running it in a Docker container. The official [Jenkins Docker image](#) lets you run and test an installation of Jenkins without actually configuring it on a local machine.

Installing Docker

See my [introduction to Docker](#) for a beginner's guide to Docker, including installation and setup instructions.

Assuming you already have Docker setup in your development environment, you can launch Jenkins from the Docker the command line:

```
docker run -p 8080:8080 -p 50000:50000 -v  
/your/home/jenkins:/var/jenkins_home -d jenkins
```

This command tells Docker to run the latest release of `jenkins` with the following options:

- `-p 8080:8080`: Maps port 8080 on the Docker container to port 8080 on the Docker host, so that you can connect to the Jenkins web app on port 8080.
- `-p 50000:50000`: Maps port 50000 on the Docker container to port 50000 on the Docker host. Jenkins uses this port internally to allow build slave executors to connect to the master Jenkins server.
- `-v /your/home/jenkins:/var/jenkins_home`: Maps Jenkins data storage to your local directory, so that you can restart your Docker container without losing your data.
- `-d`: Lets you run the Docker container in a detached mode, or as a daemon process.

The following shows the output for running these commands:

```
$ docker run -p 8000:8080 -v
/Users/shaines/jenkins/:/var/jenkins_home -d jenkins
cc16573ce71ae424d4122e9e4afd3a294fda6606e0333838fe332fc4e11d0d53
```

Because we're running our Docker container in detached mode, we need to follow the logs that are output by Jenkins. You can do so with the `docker logs -f` command. Just pass in the first few hexadecimal numbers of the container ID, in this case `cc16573ce71ae424d4122e9e4afd3a294fda6606e0333838fe332fc4e11d0d53`:

```
$ docker logs -f cc1

Running from: /usr/share/jenkins/jenkins.war
webroot: EnvVars.masterEnvVars.get("JENKINS_HOME")

...

*****

*****

*****
```

Jenkins initial setup **is** required. **An** admin user has been created **and** a password generated.

Please use the following password to proceed to installation:

205be6fe69c447dd933a3c9ce7420496

This may also be found at:

/var/jenkins_home/secrets/initialAdminPassword

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](#) 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.


Enter an item name

hello-world-servlet


» Required field

**Freestyle project**


This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

**Pipeline**


Orchestrates long-running activities that can span multiple build slaves. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

**External Job**


This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system. See [the documentation for more details](#).

**Multi-configuration project**

Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

**Folder**

Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.

**Multibranch Pipeline**

Creates a set of Pipeline projects according to detected branches in one SCM repository.

OK

Steven Haines

Figure 5. Setting up a new project in Jenkins

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.



Steven Haines

Figure 6. Jenkins dashboard with the new project added

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.

Jenkins > hello-world-servlet >

Back to Dashboard

 Status

 Changes

 Workspace

 Build Now

 Delete Project

 Configure

 GitHub Hook Log

 Move

 GitHub

Project hello-world-servlet

[Workspace](#)

[Recent Changes](#)

Permalinks

- [Last build \(#5\), 9 min 16 sec ago](#) ▼
- [Last stable build \(#5\), 9 min 16 sec ago](#)
- [Last successful build \(#5\), 9 min 16 sec ago](#)
- [Last failed build \(#3\), 44 min ago](#)
- [Last unsuccessful build \(#3\), 44 min ago](#)
- [Last completed build \(#5\), 9 min 16 sec ago](#)

Build History

X

#5	Jul 24, 2016 7:28 PM
#4	Jul 24, 2016 6:54 PM
#3	Jul 24, 2016 6:52 PM
#2	Jul 24, 2016 6:49 PM
#1	Jul 24, 2016 6:39 PM

[RSS for all](#)
[RSS for failures](#)

Steven Haines

Figure 7. A successful build in Jenkins

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.

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/remotes/origin/master)*

```
> git config core.sparsecheckout # timeout=10
```

```
> git checkout -f a3da5d8d82d11475e5e1fc899871f1f7a975ed07
```

First time build. Skipping changelog.

Unpacking

*https://repo.maven.apache.org/maven2/org/apache/maven/apache-maven/3.3.9/apache-maven-3.3.9-bin.zip to
/Users/shaines/.jenkins/tools/hudson.tasks.Maven_MavenInstallation/Maven_3.3.9 on Jenkins*

*[hello-world-servlet] \$
/Users/shaines/.jenkins/tools/hudson.tasks.Maven_MavenInstallation/Maven_3.3.9/bin/mvn clean install*

[INFO] Scanning for projects...

[WARNING]

[WARNING] Some problems were encountered while building the effective model for com.geekcap.vmturbo:hello-world-servlet-example:war:1.0-SNAPSHOT

[WARNING]

'dependencies.dependency.(groupId:artifactId:type:classifier)' must be unique: junit:junit:jar -> version 3.8.1 vs 4.12 @ line 23, column 19

[WARNING]

[WARNING] It is highly recommended to fix these problems because they threaten the stability of your build.

[WARNING]

[WARNING] For this reason, future Maven versions might no longer support building such malformed projects.

[WARNING]

[INFO]

*[INFO] -----
-----*

[INFO] Building hello-world-servlet-example Maven Webapp 1.0-SNAPSHOT

*[INFO] -----
-----*

[INFO]

[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ hello-world-servlet-example ---

[INFO]

[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ hello-world-servlet-example ---

[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!

*[INFO] skip non existing resourceDirectory
/Users/shaines/.jenkins/workspace/hello-world-servlet/src/main/resources*

[INFO]

[INFO] --- maven-compiler-plugin:2.0.2:compile (default-compile) @ hello-world-servlet-example ---

*[INFO] Compiling 2 source files to
/Users/shaines/.jenkins/workspace/hello-world-servlet/target/classes*

[INFO]

[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ hello-world-servlet-example ---

[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!

*[INFO] skip non existing resourceDirectory
/Users/shaines/.jenkins/workspace/hello-world-servlet/src/test/resources*

[INFO]

[INFO] --- maven-compiler-plugin:2.0.2:testCompile (default-testCompile) @ hello-world-servlet-example ---

*[INFO] Compiling 1 source file to
/Users/shaines/.jenkins/workspace/hello-world-servlet/target/test-classes*

[INFO]

[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ hello-world-servlet-example ---

*[INFO] Surefire report directory:
/Users/shaines/.jenkins/workspace/hello-world-servlet/target/surefire-reports*

T E S T S

Running com.geekcap.vmturbo.ThingTest

*Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.05 sec*

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO]

[INFO] --- maven-war-plugin:2.2:war (default-war) @ hello-world-servlet-example ---

[INFO] Packaging webapp

[INFO] Assembling webapp [hello-world-servlet-example] in
[/Users/shaines/.jenkins/workspace/hello-world-servlet/target/helloworld]

[INFO] Processing war project

[INFO] Copying webapp resources

[/Users/shaines/.jenkins/workspace/hello-world-servlet/src/main/webapp]

[INFO] Webapp assembled in [61 msecs]

[INFO] Building war: /Users/shaines/.jenkins/workspace/hello-world-servlet/target/helloworld.war

[INFO] WEB-INF/web.xml already added, skipping

[INFO]

[INFO] --- maven-install-plugin:2.4:install (default-install) @ hello-world-servlet-example ---

[INFO] Installing /Users/shaines/.jenkins/workspace/hello-world-servlet/target/helloworld.war to

```
/Users/shaines/.m2/repository/com/geekcap/vmturbo/hello-world-  
servlet-example/1.0-SNAPSHOT/hello-world-servlet-example-1.0-  
SNAPSHOT.war  
  
[INFO] Installing /Users/shaines/.jenkins/workspace/hello-world-  
servlet/pom.xml to  
/Users/shaines/.m2/repository/com/geekcap/vmturbo/hello-world-  
servlet-example/1.0-SNAPSHOT/hello-world-servlet-example-1.0-  
SNAPSHOT.pom  
  
[INFO] -----  
-----  
  
[INFO] BUILD SUCCESS  
  
[INFO] -----  
-----  
  
[INFO] Total time: 2.932 s  
  
[INFO] Finished at: 2016-08-20T06:35:54-04:00  
  
[INFO] Final Memory: 23M/316M  
  
[INFO] -----  
-----  
  
Finished: SUCCESS
```

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_MavenInstallati`

on/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:

```
/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, 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.



Console Output

```
Started by user Steven Haines
Building in workspace /var/jenkins_home/workspace/hello-world-servlet
Cloning the remote Git repository
Cloning repository https://github.com/ligado/hello-world-servlet
> git init /var/jenkins_home/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 a3da5d8d82d11475e5elfc899871f1f7a975ed07 (refs/remotes/origin/master)
> git config core.sparsecheckout # timeout=10
> git checkout -f a3da5d8d82d11475e5elfc899871f1f7a975ed07
First time build. Skipping changelog.
[hello-world-servlet] $ mvn clean install
FATAL: command execution failed
java.io.IOException: Cannot run program "mvn" (in directory "/var/jenkins_home/workspace/hello-world-servlet"): error=2, No such file or directory
at java.lang.ProcessBuilder.start(ProcessBuilder.java:1048)
```

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!

Table of Contents

- [Introduction to CI/CD](#)
- [Download and install Jenkins](#)
- [Setup Jenkins CI for a Java web app](#)

- [CI reports with Jenkins and JUnit](#)
- [Using static code analysis tools with Jenkins](#)

Show More

[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
```

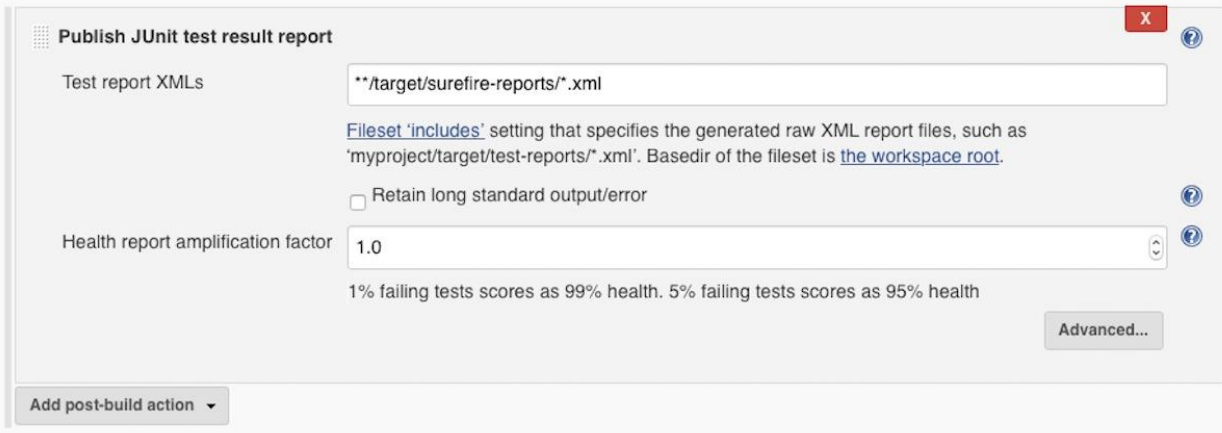
The Test Reports XMLs field points to the directory where your Surefire tests are published. Maven creates a target directory, a `surefire-reports` subdirectory, and then publishes a set of XML files summarizing the test results to the new directory. You can navigate to the directory yourself to confirm:

```
~/.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.

Post-build Actions



Publish JUnit test result report

Test report XMLs:

[Fileset 'includes' setting](#) that specifies the generated raw XML report files, such as 'myproject/target/test-reports/*.xml'. Basedir of the fileset is [the workspace root](#).

☐ Retain long standard output/error

Health report amplification factor:

1% failing tests scores as 99% health. 5% failing tests scores as 95% health

Advanced...

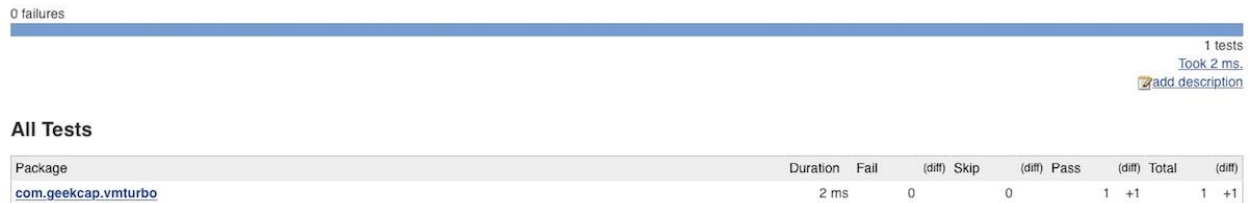
Add post-build action ▼

Steven Haines

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.

Test Result



0 failures

1 tests
Took 2 ms.
[add description](#)

All Tests

Package	Duration	Fail	(diff) Skip	(diff) Pass	(diff) Total	(diff)
com.geekcap.vmturbo	2 ms	0	0	1 +1	1 +1	

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](#). 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](#) checks the style of your code against the official Java code style.
- [FindBugs](#) searches for and reports common bugs.
- [PMD](#) validates build rules.

- [Cobertura](#) checks the coverage of your unit tests.

You need to do the following in order to use these tools and generate reports 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.
4. 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>

    </configuration>

  </plugin>

</plugins>
```

```
<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>
```



```
                                <!--  
                                <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.

From 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:

Filter:

Updates

Available

Installed

Advanced

Install	Name	Version
<input checked="" type="checkbox"/>	Static Analysis Collector Plug-in This plug-in is an add-on for the plug-ins Checkstyle , Dry , FindBugs , PMD , Task Scanner , and Warnings ; the plug-in collects the different analysis results and shows the results in a combined trend graph. Additionally, the plug-in provides health reporting and build stability based on these combined results.	1.48

Install without restart

Download now and install after restart

Update information obtained: 2 hr 25 min ago

Check now

Steven Haines

Figure 11. Installing the Static Analysis Collector plugin

Filter:

Updates

Available

Installed

Advanced

Install	Name	Version
<input type="checkbox"/>	Violations plugin This plug-in generates reports static code violation detectors such as checkstyle, pmd, cpd, findbugs, codenarc, fxcop, stylecop and simian.	0.7.11
<input checked="" type="checkbox"/>	Checkstyle Plug-in This plugin generates the trend report for Checkstyle , an open source static code analysis program. 	3.46
<input type="checkbox"/>	Box UK - JSLint Lint JavaScript files, outputting to checkstyle format. Supports all JSLint options. Developed by Box UK.	0.8.2

Install without restart

Download now and install after restart

Update information obtained: 2 hr 40 min ago

Check now

Steven Haines

Figure 12. Installing the Checkstyle plugin

Filter:

Updates

Available

Installed

Advanced

Install	Name	Version
<input type="checkbox"/>	Violations plugin This plug-in generates reports static code violation detectors such as checkstyle, pmd, cpd, findbugs, codenarc, fxcop, stylecop and simian.	0.7.11
<input checked="" type="checkbox"/>	FindBugs Plug-in This plugin generates the trend report for FindBugs , an open source program which uses static analysis to look for bugs in Java code. 	4.65

Install without restart

Download now and install after restart

Update information obtained: 2 hr 40 min ago

Check now

Steven Haines

Figure 13. Installing the FindBugs plugin

Filter:

Updates

Available

Installed

Advanced

Install	Name	Version
<input type="checkbox"/>	Violations plugin This plug-in generates reports static code violation detectors such as checkstyle, pmd, cpd, findbugs, codenarc, fxcop, stylecop and simian.	0.7.11
<input type="checkbox"/>	DRY Plug-in This plugin generates the trend report for duplicate code checkers like CPD or Simian .	2.45
<input checked="" type="checkbox"/>	PMD Plug-in This plugin generates the trend report for PMD , an open source static code analysis program. 	3.45
<input type="checkbox"/>	NOPMD verify Trac plugin When you use Trac to tracking issues, this plugin makes a report about whether 'NOPMD' in your codes are right.	0.9
<input type="checkbox"/>	NOPMD check plugin This plugin checks the keyword 'NOPMD' in your codes.	0.9

Install without restart

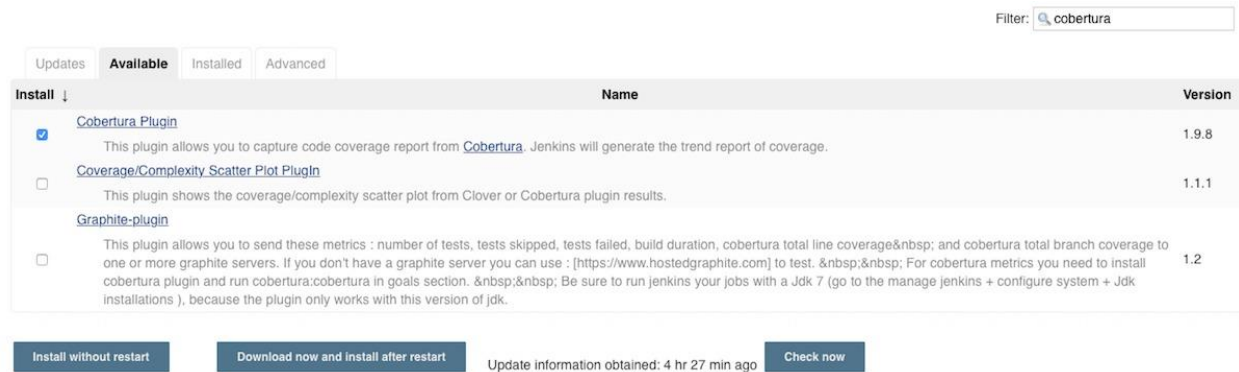
Download now and install after restart

Update information obtained: 4 hr 21 min ago

Check now

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
```

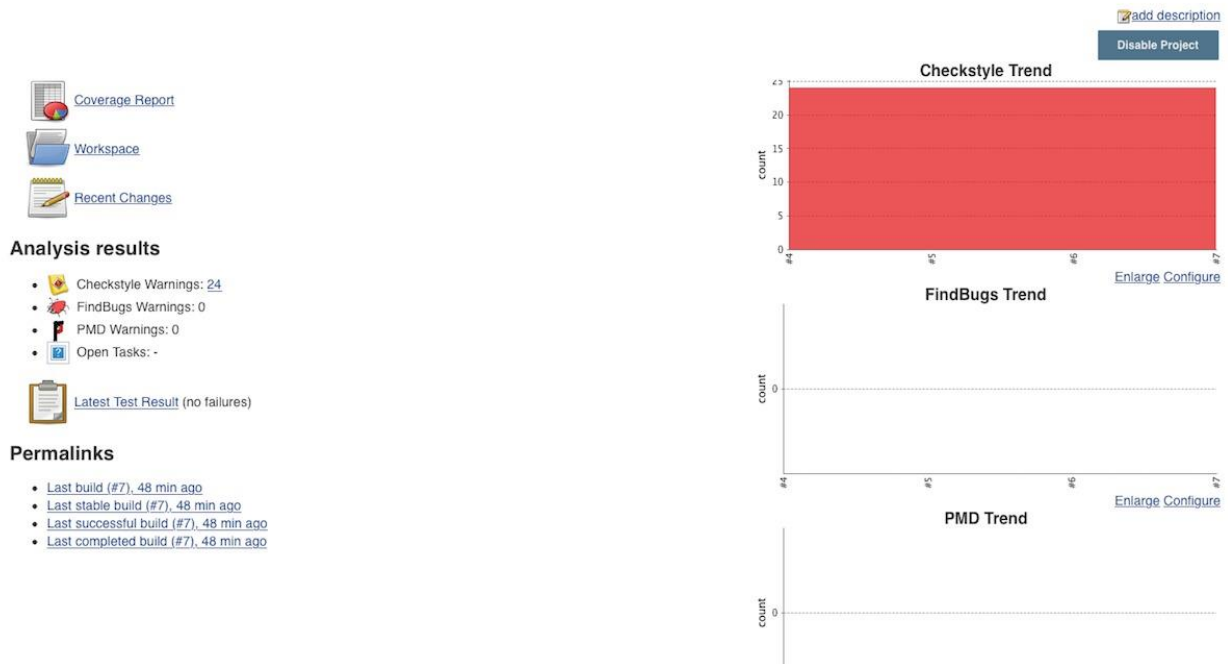
Step 4. Add your post-build actions

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.

Project hello-world-servlet



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.

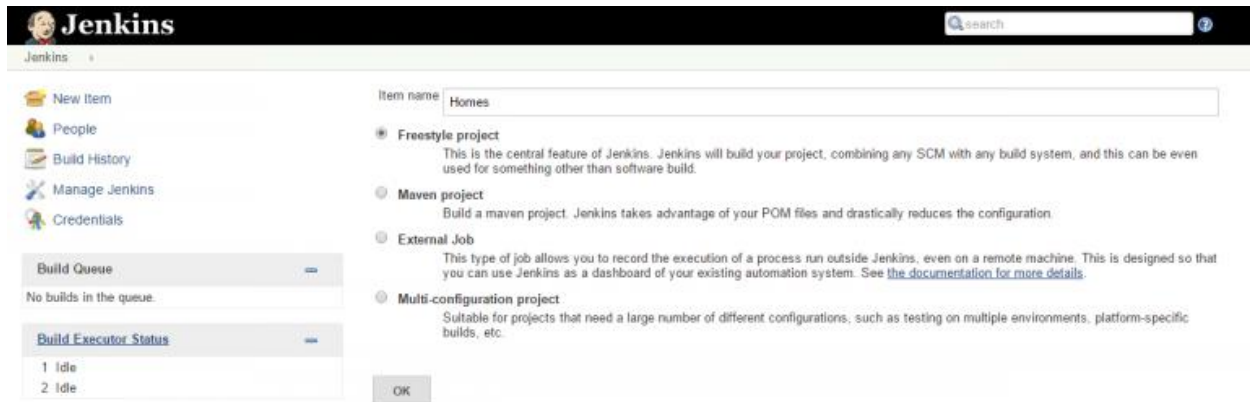
Conclusion

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.

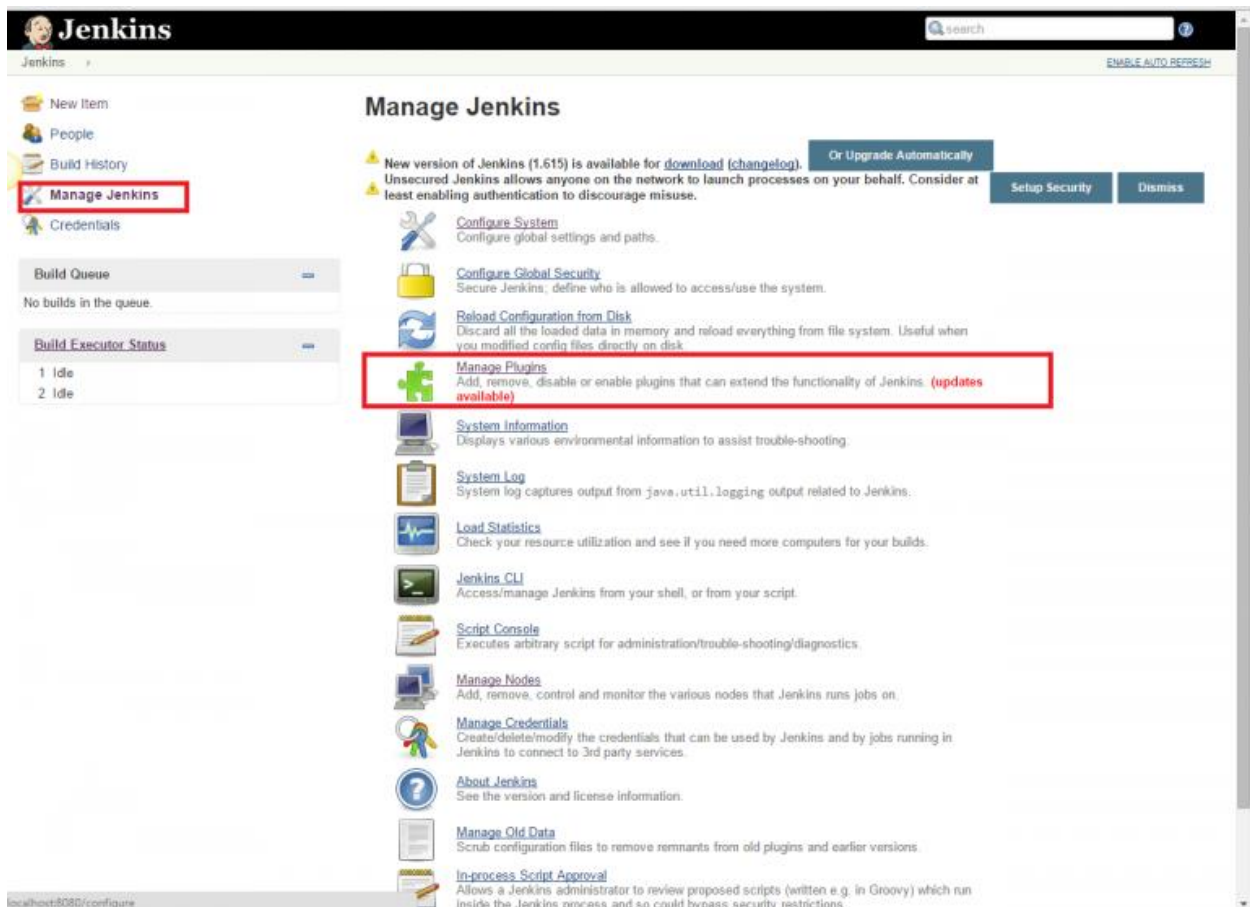
Setting up and testing an example project in Jenkins is a good way to familiarize yourself with how the Jenkins directory structure is organized. The more you understand how Jenkins works internally, the better, especially as you start to do more complex operations.

How to Configure Email Notification in Jenkins?

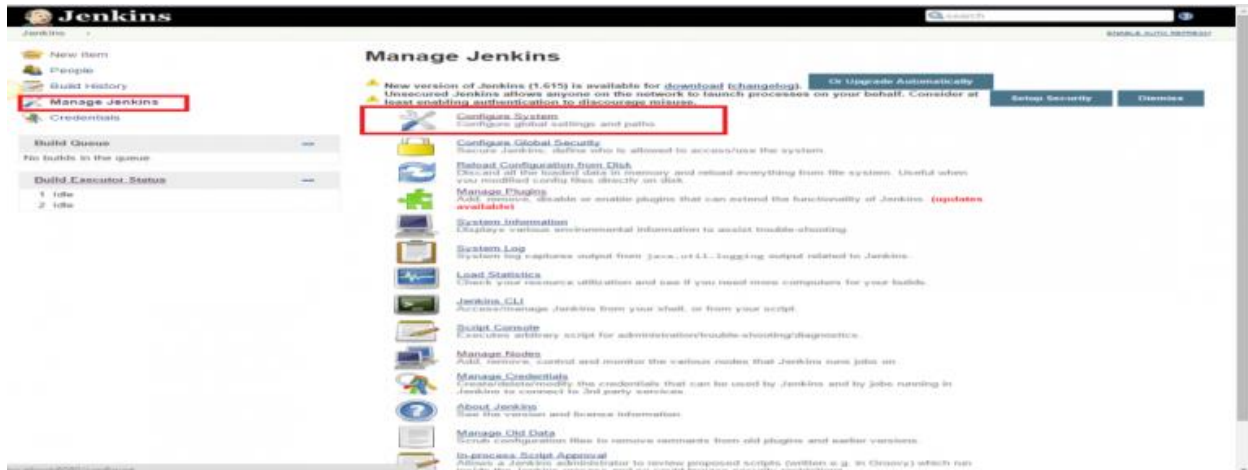
1. Open Jenkins using the following URL: <http://localhost:8080/> on any browser.



2. Click the 'Manage Jenkins' menu option displayed at the right side of the screen. You will be redirected to the 'Manage Jenkins' page, where you need to select the 'Manage Plugin' option.

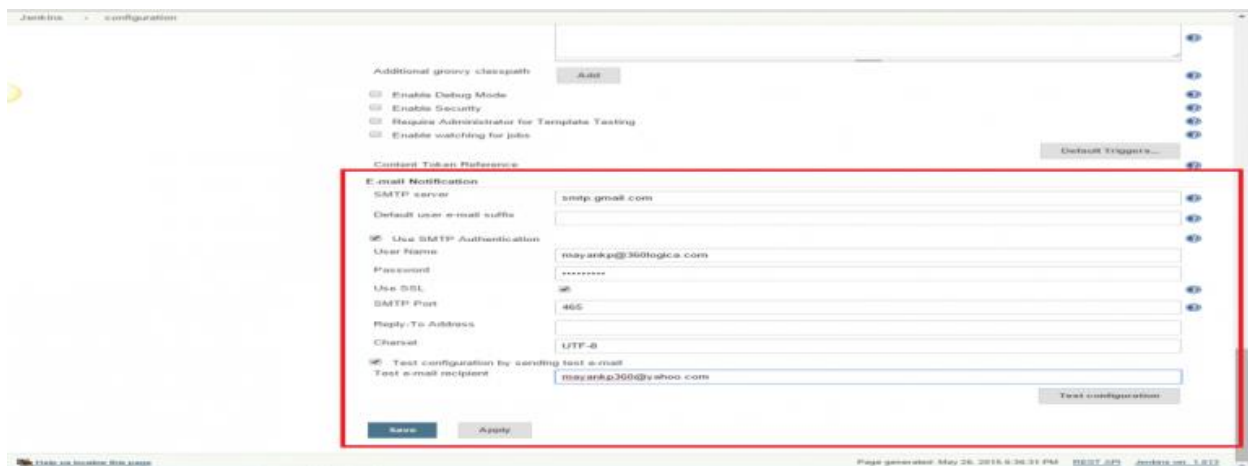


3. Click the 'Available' tab present at the top of the 'Manage Plugin' page.



7. Enter the SMTP server name under 'Email Notification'. Click the 'Advanced' button and then click the checkbox next to the 'Use SMTP Authentication' option. Now, set the following fields.

- - **SMTP server name** : smtp.gmail.com
 - **User name**: user_email_id@gmail.com
 - **Password**: 123456
 - **Use SSL** : Checked
 - **SMTP Port**: 456



8. Check the email notification functionality by clicking the checkbox next to the 'Test configuration by sending Test e-mail recipient' option. Enter a valid email id and click the 'Test configuration' button to check whether the email id is valid or not.

E-mail Notification

SMTP server

smtp.gmail.com

?

Default user e-mail suffix

?

☒ Use SMTP Authentication

?

User Name

mayankp@360logica.com

Password

Use SSL

☒

?

SMTP Port

465

?

Reply-To Address

Charset

UTF-8

☒ Test configuration by sending test e-mail

Test e-mail recipient

mayankp360@yahoo.com

Test configuration

Save

Apply

9. Go to the home page and click on a created job, like Homes. Then, click the 'Configure' option.

Jenkins

New Item

People

Build History

Manage Jenkins

Credentials

Build Queue

No builds in the queue.

Build Executor Status

1 Info

2 Info

All

S	W	Name	Last Success	Last Failure	Last Duration
Icon	S M L	Homes	N/A	N/A	N/A

Legend

RSS for all

RSS for failures

RSS for just latest builds

Link to Jenkins Wiki page

Page generated: May 26, 2015 6:10:29 PM

DEBT_JEN

Jenkins ver. 1.612

Jenkins

Homes

Back to Dashboard

Status

Changes

Workspace

Build Now

Delete Project

Configure

Build History

RSS for all

RSS for failures

Workspace

Recent Changes

Permalinks

edit description

Create Project

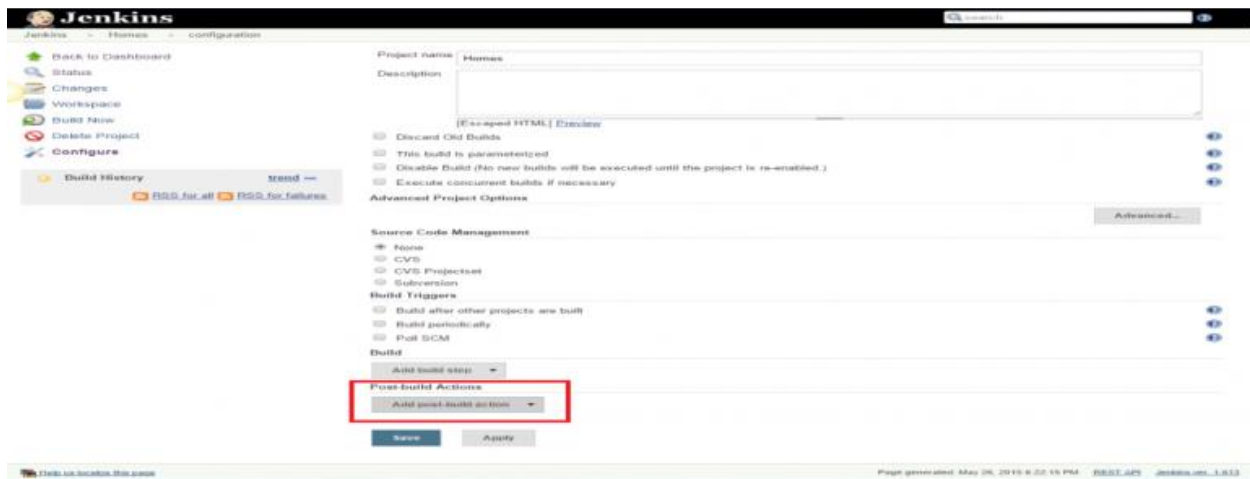
Link to Jenkins Wiki page

Page generated: May 26, 2015 6:10:29 PM

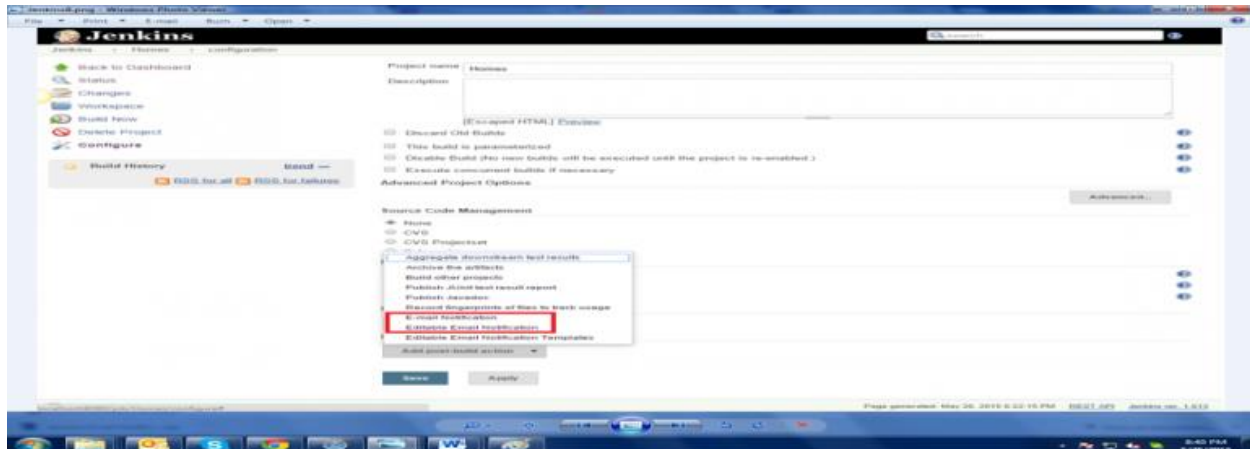
DEBT_JEN

Jenkins ver. 1.612

10. Click the 'Add post-build action' drop-down.



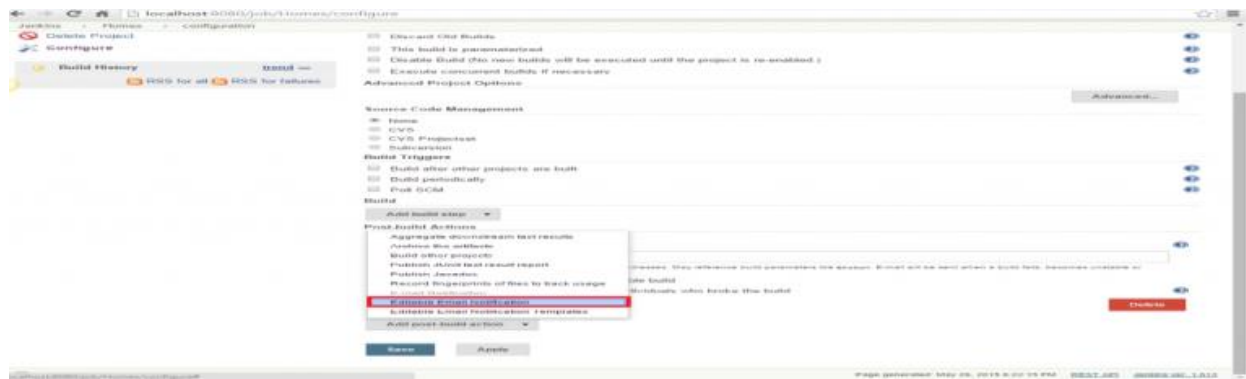
11. Select the 'E-mail Notification' value.



12. Enter the recipient email id in the 'E-mail Notification' box and select the checkbox next to the 'Send e-mail for every unstable build' option.

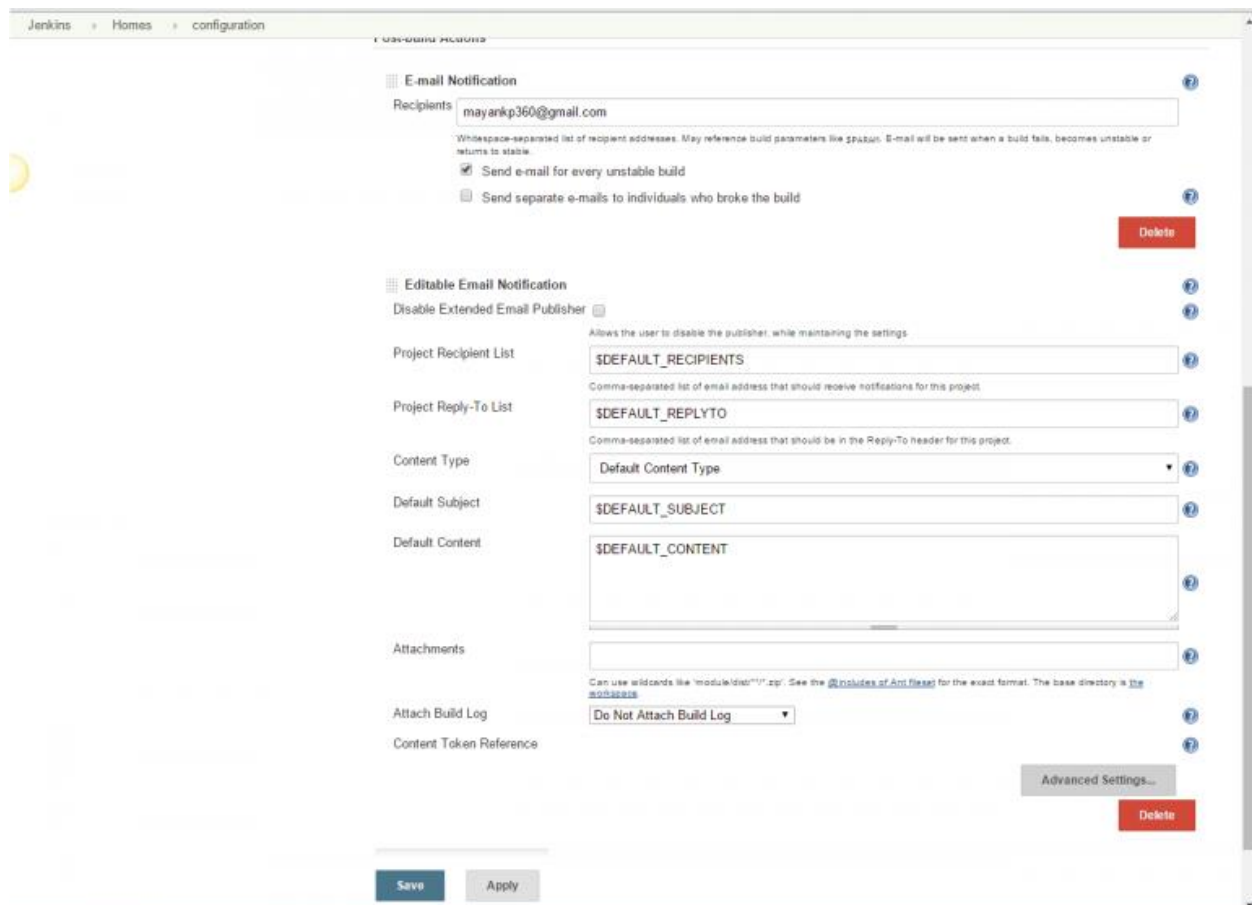


13. Click the 'Add post-build action' drop-down and select the 'Editable Email Notification' value.



14. Fill the 'Editable Email Notification' fields.

- Project Recipient List : email_id@gmail.com



15. Click the 'Advance Settings...' button in the 'Editable Email Notification' box.

16. Click the 'Add Trigger' drop-down and select the 'Always' option.

19. Click the 'Build now' link and check the email id after the job execution.