



JAWORLD



OPEN SOURCE JAVA PROJECTS

By Steven Haines | Follow

About |

A working developer's guide to open source tools and frameworks for Java application development.

HOW-TO

Open source Java projects: Jenkins with Docker, Part 1

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

JavaWorld | SEP 29, 2016 2:05 PM PT



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. (We'll explore the more subtle differences between those two terms in the second half of this article.)

This installment of **Open source Java projects** gets you started with 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 and troubleshoot build failures. You'll also install and run a handful of popular Jenkins plugins for static code analysis testing and reporting.

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.

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, as you'll learn in Part 2.

Getting to know 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.

Jenkins in a local installation

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.



Figure 1. Jenkins setup 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.

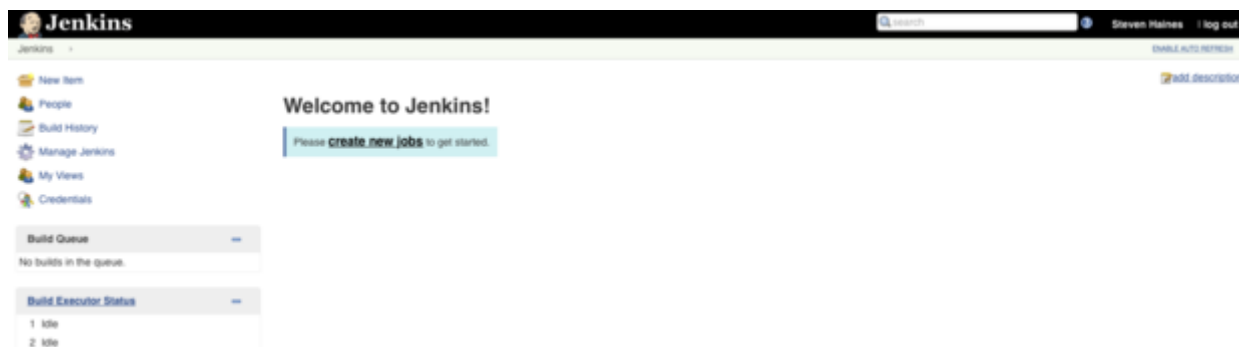


Figure 2. Jenkins homepage

Setup the example app

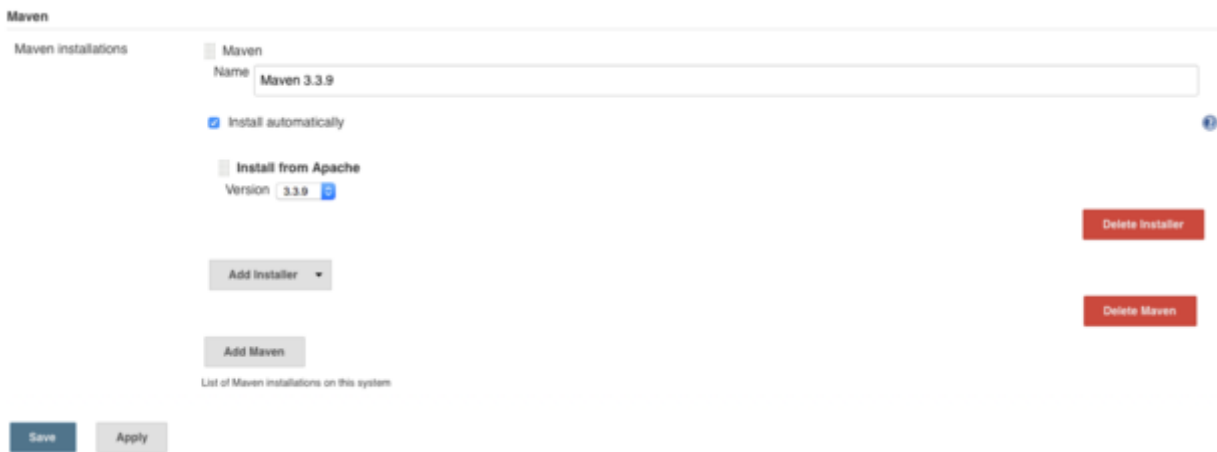
Before we can use Jenkins to build a Java web project with Maven, we need to setup both of these technologies. Under-the-hood, Jenkins will checkout source code from a source code repository to a local directory and execute the Maven targets that you specify. For that to work, you need to install one 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.

The screenshot shows the 'JDK' configuration page in Jenkins. At the top, there's a 'JDK installations' section. Below it, a 'JDK' entry is shown with a 'Name' field containing 'JDK8'. The 'Install automatically' checkbox is checked. Under the 'Install from java.sun.com' section, the 'Version' is set to 'Java SE Development Kit 8u102'. A checkbox for 'I agree to the Java SE Development Kit License Agreement' is checked. A red error message states 'Installing JDK requires Oracle account. Please enter your username/password'. On the right side, there are two red buttons: 'Delete Installer' and 'Delete JDK'. At the bottom left, there are buttons for 'Add Installer' and 'Add JDK'. A small text at the very bottom reads 'List of JDK installations on this system'.

Figure 3. Configuring JDK on 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

Maven

Name

☒ Install automatically

☒ Install from Apache

Version

List of Maven installations on this system

Figure 4. Configuring Maven on 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.

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:

```
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 8080:8080 -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

*****
*****
*****
```

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.

The screenshot shows the Jenkins 'New Project' configuration page, specifically the 'General' tab. The 'Project name' field is filled with 'hello-world-servlet'. The 'Description' field is empty. Below the description, there is a '[Plain text] Preview' link. The 'Discard old builds' checkbox is unchecked, and the 'GitHub project' checkbox is checked. The 'Project url' field is filled with 'https://github.com/ligado/hello-world-servlet'. There are two 'Advanced...' buttons. Below the 'General' tab, the 'Source Code Management' section is visible, with 'Git' selected as the SCM type. The 'Repository URL' field is filled with 'https://github.com/ligado/hello-world-servlet'.

General Source Code Management Build Triggers Build Environment Build Post-build Actions

Project name

Description

[Plain text] [Preview](#)

☐ Discard old builds

☒ GitHub project

Project url

[Advanced...](#)

☐ This project is parameterized

☐ Throttle builds

☐ Disable this project

☐ Execute concurrent builds if necessary

[Advanced...](#)

Source Code Management

☐ None

☒ Git

Repositories

Repository URL

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. We'll explore pipelines more in Part 2.
- 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.

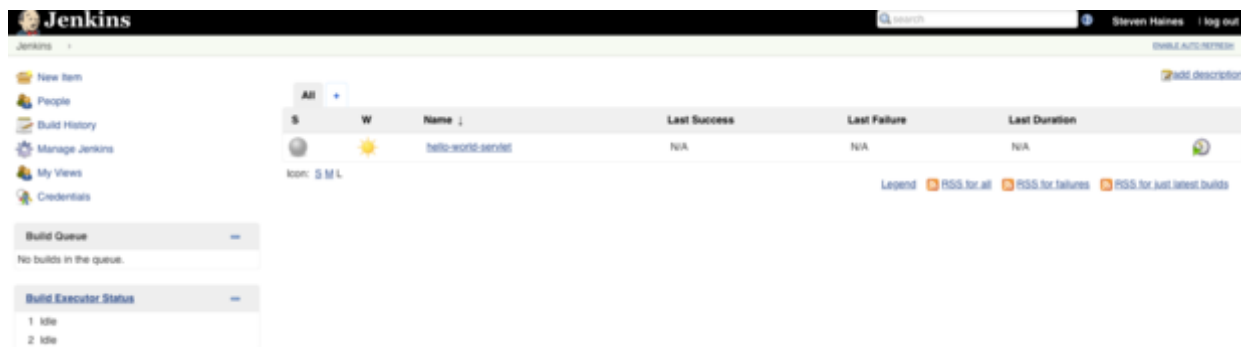


Figure 6. Jenkins dashboard with the new project added

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
 [trend =](#)

#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](#)

Figure 7. A successful build

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.

1 | 2 | 3 | **NEXT** ➤