

Unit-4 Notes

Why do we build code?

Most developers are familiar with the process of building code. When we work in the field of DevOps, however, we might face issues that developers who specialize in programming a particular component type won't necessarily experience.

For the purposes of this book, we define software building as the process of molding code from one form to another. During this process, several things might happen:

- The compilation of source code to native code or virtual machine bytecode, depending on our production platform.

- Linting of the code: checking the code for errors and generating code quality measures by means of static code analysis. The term "Linting" originated with a program called Lint, which started shipping with early versions of the Unix operating system. The purpose of the program was to find bugs in programs that were syntactically correct, but contained suspicious code patterns that could be identified with a different process than compiling.

- Unit testing, by running the code in a controlled manner.

- The generation of artifacts suitable for deployment.

It's a tall order!

Not all code goes through each and every one of these phases. Interpreted languages, for example, might not need compilation, but they might benefit from quality checks.

The many faces of build systems

There are many build systems that have evolved over the history of software development. Sometimes, it might feel as if there are more build systems than there are programming languages.

Here is a brief list, just to get a feeling for how many there are:

- For Java, there is Maven, Gradle, and Ant

- For C and C++, there is Make in many different flavors

- For Clojure, a language on the JVM, there is Leiningen and Boot apart from Maven

- For JavaScript, there is Grunt

- For Scala, there is sbt

- For Ruby, we have Rake

- Finally, of course, we have shell scripts of all kinds

Depending on the size of your organization and the type of product you are building, you might encounter any number of these tools. To make life even more interesting, it's not uncommon for organizations to invent their own build tools.

As a reaction to the complexity of the many build tools, there is also often the idea of standardizing a particular tool. If you are building complex heterogeneous systems, this is rarely efficient. For example, building JavaScript software is just easier with Grunt than it is with

Maven or Make, building C code is not very efficient with Maven, and so on. Often, the tool exists for a reason.

Normally, organizations standardize on a single ecosystem, such as Java and Maven or Ruby and Rake. Other build systems besides those that are used for the primary code base are encountered mainly for native components and third-party components.

At any rate, we cannot assume that we will encounter only one build system within our organization's code base, nor can we assume only one programming language.

I have found this rule useful in practice: *it should be possible for a developer to check out the code and build it with minimal surprises on his or her local developer machine.*

This implies that we should standardize the revision control system and have a single interface to start builds locally.

If you have more than one build system to support, this basically means that you need to wrap one build system in another. The complexities of the build are thus hidden and more than one build system at the same time are allowed. Developers not familiar with a particular build can still expect to check it out and build it with reasonable ease.

Maven, for example, is good for declarative Java builds. Maven is also capable of starting other builds from within Maven builds.

This way, the developer in a Java-centric organization can expect the following command line to always build one of the organization's components:

```
mvn clean install
```

One concrete example is creating a Java desktop application installer with the Nullsoft NSIS Windows installation system. The Java components are built with Maven. When the Java artifacts are ready, Maven calls the NSIS installer script to produce a self-contained executable that will install the application on Windows.

While Java desktop applications are not fashionable these days, they continue to be popular in some domains.

The Jenkins build server

A build server is, in essence, a system that builds software based on various triggers. There are several to choose from. In this book, we will have a look at Jenkins, which is a popular build server written in Java.

Jenkins is a fork of the Hudson build server. Kohsuke Kawaguchi was Hudson's principal contributor, and in 2010, after Oracle acquired Hudson, he continued work on the Jenkins fork. Jenkins is clearly the more successful of the two strains today.

Jenkins has special support for building Java code but is in no way limited to just building Java.

Setting up a basic Jenkins server is not particularly hard at the outset. In Fedora, you can just install it via dnf:

```
dnf install jenkins
```

Jenkins is handled as a service via systemd:

```
systemctl start Jenkins
```

You can now have a look at the web interface at <http://localhost:8080>:

The screenshot shows the Jenkins dashboard with the following components:

- Header:** Jenkins logo, search bar, and "ENABLE AUTO REFRESH" link.
- Left Sidebar:**
 - New Item
 - People
 - Build History
 - Manage Jenkins
 - Credentials
 - Build Queue (No builds in the queue)
 - Build Executor Status (1 Idle, 2 Idle)
- Main Content Area:**
 - Buttons: All, +, add description
 - Table of jobs:

S	W	Name	Last Success	Last Failure	Last Duration
		emacs	50 min - #384	9 mo 8 days - #239	11 min
		fortune	4 days 6 hr - #2	N/A	81 ms
		inkscape	12 hr - #102	10 hr - #103	5 min 16 sec

Icon: S M L

Legend: RSS for all, RSS for failures, RSS for just latest builds

The Jenkins instance in the screenshot has a couple of jobs already defined. The fundamental entity in Jenkins is the job definition, and there are several types to choose from. Let's create a simple job in the web interface. To keep it simple, this job will just print a classic Unix fortune quote:

1. Create a job of the type **Freestyle project**:

The screenshot shows the Jenkins 'New Item' form with the following components:

- Header:** Jenkins logo, search bar, and "All" button.
- Left Sidebar:**
 - New Item
 - People
 - Build History
 - Manage Jenkins
 - Credentials
 - Build Queue (No builds in the queue)
 - Build Executor Status (1 Idle, 2 Idle)
- Main Content Area:**
 - Item name: fortunes
 - Project type selection:
 - ☒ **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.
 - ☐ **Maven project**
Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.
 - ☐ **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.
 - ☐ **Copy existing item**
Copy from:
 - OK button

1. Add a **shell build** step.
2. In the shell entry (Command), type fortune:

The screenshot shows the Jenkins web interface for configuring a project named 'fortune'. The top navigation bar includes the Jenkins logo, a search bar, and the breadcrumb 'Jenkins > fortune > configuration'. On the left sidebar, there are links for 'Back to Dashboard', 'Status', 'Changes', 'Workspace', 'Build Now', 'Delete Project', and 'Configure'. Below these is a 'Build History' section showing two builds from September 6, 2015, with links for 'RSS for all' and 'RSS for failures'. The main configuration area is divided into several sections: 'Project name' (fortune), 'Description' (empty), 'Advanced Project Options' (with checkboxes for 'Discard Old Builds', 'This build is parameterized', 'Disable Build', and 'Execute concurrent builds if necessary'), 'Source Code Management' (with radio buttons for 'None', 'Bazaar', 'CVS', 'CVS Projectset', 'Git', and 'Subversion'), 'Build Triggers' (with checkboxes for 'Build after other projects are built', 'Build periodically', and 'Poll SCM'), and 'Build' (with a checkbox for 'Execute shell' and a 'Command' field containing 'fortune'). There are also links for 'See the list of available environment variables' and a 'Delete' button. At the bottom, there are buttons for 'Add build step', 'Post-build Actions', 'Add post-build action', 'Save', and 'Apply'.

Whenever you run the job, a fortune quote will be printed in the job log.

Jobs can be started manually, and you will find a history of job executions and can examine each job log. This keeps a history of previous executions, which is very handy when you are trying to figure out which change broke a build and how to fix it.

If you don't have the fortune command, install it with `dnf install fortune-mod`, or you might opt to simply run the `date` command instead. This will just output the date in the build log instead of classic quotes and witticisms.

Managing build dependencies

In the previous, simple example, we printed a fortune cookie to the build log.

While this exercise can't be compared in complexity with managing real builds, we at least learned to install and start Jenkins, and if you had issues installing the fortune utility, you got a glimpse of the dark underbelly of managing a Continuous Integration server: managing the build dependencies.

Some build systems, such as the Maven tool, are nice in the way that the Maven POM file contains descriptions of which build dependencies are needed, and they are fetched automatically by Maven if they aren't already present on the build server. Grunt works in a similar way for JavaScript builds. There is a build description file that contains the dependencies required for the build. Golang builds can even contain links to GitHub repositories required for completing the build.

C and C++ builds present challenges in a different way. Many projects use GNU Autotools; among them is Autoconf, which adapts itself to the dependencies that are available on the host rather than describing which dependencies they need. So, to build Emacs, a text editor, you first run a configuration script that determines which of the many potential dependencies are available on the build system.

While this is a useful feature if you want your software to work in many different configurations depending on which system it should run on, it's not often the way we would like our builds to behave in an enterprise setting. In this case, we need to be perfectly sure which features will be available in the end. We certainly don't want bad surprises in the form of missing functionality on our production servers.

The RPM (short for Red Hat Package Manager) system, which is used on systems derived from Red Hat, offers a solution to this problem. At the core of the RPM system is a build descriptor file called a spec file, short for specification file. It lists, among other things, the build dependencies required for a successful build and the build commands and configuration options used. Since a spec file is essentially a macro-based shell script, you can use it to build many types of software. The RPM system also has the idea that build sources should be pristine. The spec file can adapt the source code by patching the source before building it.

Jenkins plugins

Jenkins has a plugin system to add functionality to the build server. There are many different plugins available, and they can be installed from within the Jenkins web interface. Many of them can be installed without even restarting Jenkins. This screenshot shows a list of some of the available plugins:

The screenshot shows the Jenkins Plugin Manager interface. At the top, there's a search bar and a filter input. Below the navigation bar, there are tabs for 'Updates', 'Available', 'Installed', and 'Advanced'. The 'Available' tab is selected, showing a list of plugins under the 'Artifact Uploaders' category. Each plugin entry includes a checkbox, the plugin name, a brief description, and the version number.

Install	Name	Version
<input type="checkbox"/>	Appaloosa Plugin Publish your mobile applications (Android, iOS, ...) to the appaloosa-store.com platform.	1.4.2
<input type="checkbox"/>	Appetize.io Plugin Stream iOS & Android builds directly within Jenkins via Appetize.io's cloud-based iOS Simulators & Android Emulators.	1.1.0
<input type="checkbox"/>	apptwack A Jenkins CI plugin for running Android/iOS mobile tests on 100s of real devices using AppThwack.	1.9
<input type="checkbox"/>	ArtifactPromotionPlugin Using this plugin you can promote artifacts by moving release candidates into release repositories.	0.3.5
<input type="checkbox"/>	Artifact Deployer Plug-in This plugin makes it possible to copy artifacts to remote locations.	0.33
<input type="checkbox"/>	aws-device-farm AWS Device Farm Jenkins Plugin	1.9
<input type="checkbox"/>	AWS Lambda Plugin This plugin adds AWS Lambda invocation and deployment abilities to build steps and post build actions	0.3.0
<input type="checkbox"/>	AWS Elastic Beanstalk Deployment Plugin This plugin allows you to deploy into AWS Elastic Beanstalk by Packaging, Creating a new Application Version, and Updating an Environment	0.0.3
<input type="checkbox"/>	Backlog plugin This plugin integrates Backlog (for Japanese users) to Jenkins.	1.10
<input type="checkbox"/>	Hudson Build-Publisher plugin This plugin allows records from one Jenkins to be published on another Jenkins.	1.21
<input type="checkbox"/>	Capitomecat Plugin This plugin deploy the WAR file to multiple remote Tomcat servers by using Capistrano 3	0.1.0
<input type="checkbox"/>	Cloud Foundry Plugin Pushes a project to Cloud Foundry or a CF-based platform (e.g. Stackato) at the end of a build.	1.4.2
<input type="checkbox"/>	AWS CodeDeploy Plugin for Jenkins Adds a post-build step to integrate Jenkins with AWS CodeDeploy	1.7
<input type="checkbox"/>	Confluence Publisher This plugin allows you to publish build artifacts as attachments to an Atlassian Confluence wiki page.	1.8
<input type="checkbox"/>	Criticicism dSYM Plugin A Jenkins CI plugin for uploading dSYM files to Crittericism.	1.1
<input type="checkbox"/>	CRX Content Package Deployer Plugin Deploys content packages to Adobe CRX applications, like Adobe CQ 5.4, CQ 5.5, and AEM 5.6. Also allows downloading packages from one CRX server and uploading them to one or more other CRX servers.	1.3.2
<input type="checkbox"/>	Deploy to container Plugin This plugin takes a war/ear file and deploys that to a running remote application server at the end of a build	1.10
<input type="checkbox"/>	Deploy to Websphere container Plugin This plugin is an extension of the Deploy Plugin . It takes a war/ear file and deploys that to a running remote WebSphere Application Server at the end of a build.	1.0
<input type="checkbox"/>	XebiaLabs XL Deploy Plugin The XL Deploy Plugin integrates Jenkins with XebiaLabs XL Deploy	5.0.0

At the bottom of the plugin list, there are two buttons: 'Install without restart' and 'Download now and install after restart'. To the right, it says 'Update information obtained: 6 hr 30 min ago'.

Among others, we need the Git plugin to poll our source code repositories.

Our sample organization has opted for Clojure for their build, so we will install the Leiningen plugin.

The host server

The build server is usually a pretty important machine for the organization. Building software is processor as well as memory and disk intensive. Builds shouldn't take too long, so you will need a server with good specifications for the build server – with lots of disk space, processor cores, and RAM.

The build server also has a kind of social aspect: it is here that the code of many different people and roles integrates properly for the first time. This aspect grows in importance if the servers are fast enough. Machines are cheaper than people, so don't let this particular machine be the area you save money on.

Build slaves

To reduce build queues, you can add build slaves. The master server will send builds to the slaves based on a round-robin scheme or tie specific builds to specific build slaves.

The reason for this is usually that some builds have certain requirements on the host operating system.

Build slaves can be used to increase the efficiency of parallel builds. They can also be used to build software on different operating systems. For instance, you can have a Linux Jenkins master server and Windows slaves for components that use Windows build tools. To build software for the Apple Mac, it's useful to have a Mac build slave, especially since Apple has quirky rules regarding the deployment of their operating system on virtual servers.

In essence, there must be a way for the Jenkins master to issue commands to the build slave. This command channel can be the classic SSH method, and Jenkins has a built-in SSH facility. You can also start a Jenkins slave by downloading a Java JNLP client from the master to the slave. This is useful if the build slave doesn't have an SSH server.

Software on the host

Depending on the complexity of your builds, you might need to install many different types of build tool on your build server. Remember that Jenkins is mostly used to trigger builds, not perform the builds themselves. That job is delegated to the build system used, such as Maven or Make.

In my experience, it's most convenient to have a Linux-based host operating system. Most of the build systems are available in the distribution repositories, so it's very convenient to install them from there.

To keep your build server up to date, you can use the same deployment servers that you use to keep your application servers up to date.

Triggers

You can either use a timer to trigger builds, or you can poll the code repository for changes and build if there were changes.

It can be useful to use both methods at the same time:

- Git repository polling can be used most of the time so that every check in triggers a build.

- Nightly builds can be triggered, which are more stringent than continuous builds, and thus take a longer time. Since these builds happen at night when nobody is supposed to work, it doesn't matter if they are slow.

- An upstream build can trigger a downstream build.

You can also let the successful build of one job trigger another job.

Job chaining and build pipelines

It's often useful to be able to chain jobs together. In its simplest form, this works by triggering a second job in the event that the first job finishes successfully. Several jobs can be cascaded this way in a chain. Such a build chain is quite often good enough for many purposes. Sometimes, a

nicer visualization of the build steps as well as greater control over the details of the chain is desired.

In Jenkins terminology, the first build in a chain is called the upstream build, and the second one is called the downstream build.

While this way of chaining builds is often sufficient, there will most likely be a need for greater control of the build chain eventually. Such a build chain is often called a pipeline or workflow.

There are many plugins that create improved pipelines for Jenkins, and the fact that there are several shows that there is indeed a great desire for improvements in this area.

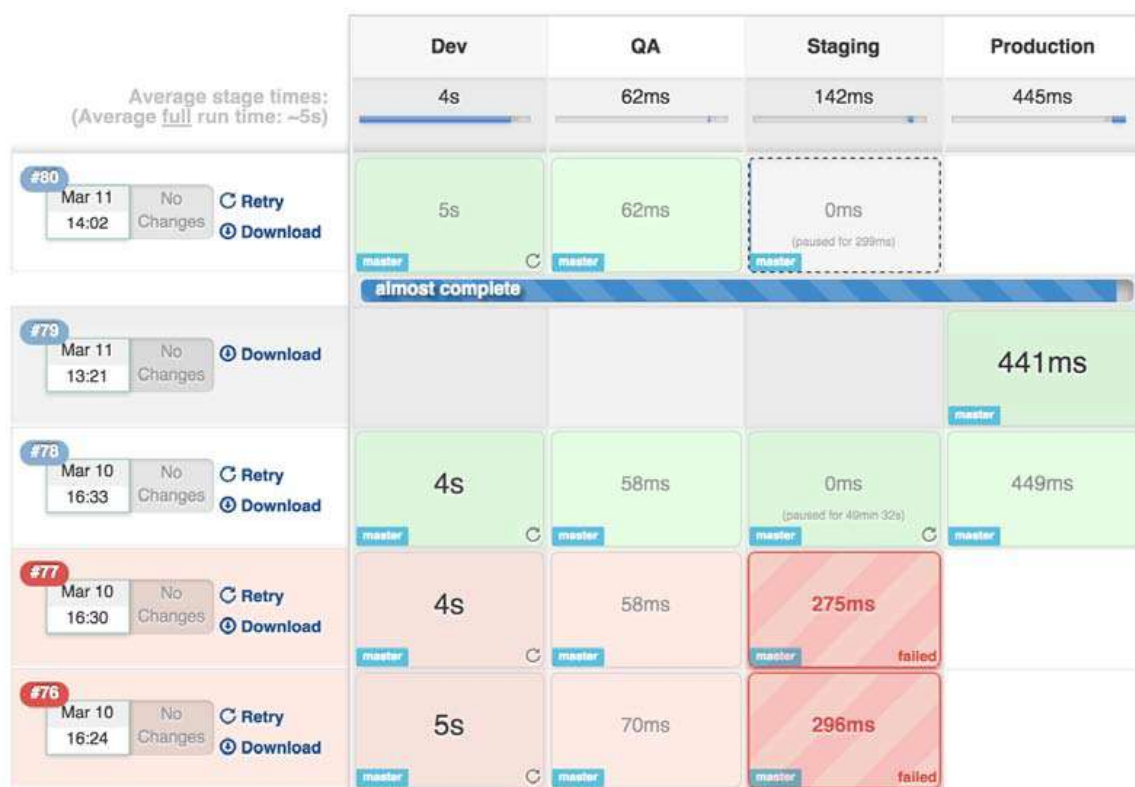
Two examples are the multijob plugin and the workflow plugin.

The workflow plugin is the more advanced and also has the advantage that it can be described in a Groovy DSL rather than fiddling about in a web UI.

The workflow plugin is promoted by CloudBees, who are the principal contributors to Jenkins today.

An example workflow is illustrated here:

Stage View



When you have a look at the Groovy build script that the workflow plugin uses, you might get the idea that Jenkins is basically a build tool with a web interface, and you would be more or less correct.

Build servers and infrastructure as code

While we are discussing the Jenkins file structure, it is useful to note an impedance mismatch that often occurs between GUI-based tools such as Jenkins and the DevOps axiom that infrastructure should be described as code.

One way to understand this problem is that while Jenkins job descriptors are text file-based, these text files are not the primary interface for changing the job descriptors. The web interface is the primary interface. This is both a strength and weakness.

It is easy to create ad-hoc solutions on top of existing builds with Jenkins. You don't need to be intimately familiar with Jenkins to do useful work.

On the other hand, the out-of-the-box experience of Jenkins lacks many features that we are used to from the world of programming. Basic features like inheritance and even function definitions take some effort to provide in Jenkins.

The build server feature in GitLab, for example, takes a different approach. Build descriptors are just code right from the start. It is worth checking out this feature in GitLab if you don't need all the possibilities that Jenkins offers.

Building by dependency order

Many build tools have the concept of a build tree where dependencies are built in the order required for the build to complete, since parts of the build might depend on other parts.

In Make-like tools, this is described explicitly; for instance, like this:

```
a.out : b.o c.o
b.o : b.c
c.o : c.c
```

So, in order to build a.out, b.o and c.o must be built first.

In tools such as Maven, the build graph is derived from the dependencies we set for an artifact. Gradle, another Java build tool, also creates a build graph before building.

Jenkins has support for visualizing the build order for Maven builds, which is called the **reactor** in Maven parlance, in the web user interface.

This view is not available for Make-style builds, however.

The screenshot shows the Jenkins web interface. The top navigation bar includes links for 'Back to Dashboard', 'Status', 'Changes', 'Workspace', 'Build Now', 'Delete Maven project', 'Configure', 'Modules', and 'Git Polling Log'. The main content area is titled 'Modules' and displays a table of modules. The table has columns for 'S' (Status), 'W' (Warning), 'Name', 'Last Success', 'Last Failure', and 'Last Duration'. The modules listed include 'acd', 'adminWeb', 'alarm-handling-service', 'alarm-to-client', 'applet-client-download', 'applet-xmirc-interface-servlet', 'client-update-servlet', 'customer-web-resources', 'licence-manager', 'mmx', 'mmx-eat', 'mmx-RPM', and 'mmxCOREBeans-eb3'. The 'Build History' section on the left shows a list of recent builds with their timestamps.

S	W	Name	Last Success	Last Failure	Last Duration
		acd	4 hr 48 min - #2017	N/A	2.7 sec
		adminWeb	4 hr 48 min - #2017	N/A	5 sec
		alarm-handling-service	4 hr 48 min - #2017	N/A	1 sec
		alarm-to-client	4 hr 48 min - #2017	N/A	0.93 sec
		applet-client-download	4 hr 48 min - #2017	N/A	1 sec
		applet-xmirc-interface-servlet	4 hr 48 min - #2017	N/A	3 sec
		client-update-servlet	4 hr 48 min - #2017	N/A	1.4 sec
		customer-web-resources	4 hr 48 min - #2017	N/A	0.68 sec
		licence-manager	4 hr 48 min - #2017	N/A	1.1 sec
		mmx	4 hr 48 min - #2017	N/A	0.54 sec
		mmx-eat	4 hr 48 min - #2017	N/A	10 sec
		mmx-RPM	4 hr 48 min - #2017	N/A	1 min 56 sec
		mmxCOREBeans-eb3	4 hr 48 min - #2017	N/A	7.3 sec

Build phases

One of the principal benefits of the Maven build tool is that it standardizes builds.

This is very useful for a large organization, since it won't need to invent its own build standards. Other build tools are usually much more lax regarding how to implement various build phases. The rigidity of Maven has its pros and cons. Sometimes, people who get started with Maven reminisce about the freedom that could be had with tools such as Ant.

You can implement these build phases with any tool, but it's harder to keep the habit going when the tool itself doesn't enforce the standard order: building, testing, and deploying.

We will examine testing in more detail in a later chapter, but we should note here that the testing phase is very important. The Continuous Integration server needs to be very good at catching errors, and automated testing is very important for achieving that goal.

Alternative build servers

While Jenkins appears to be pretty dominant in the build server scene in my experience, it is by no means alone. Travis CI is a hosted solution that is popular among open source projects. Buildbot is a buildserver that is written in, and configurable with, Python. The Go server is

another one, from ThoughtWorks. Bamboo is an offering from Atlassian. GitLab also supports build server functionality now.

Do shop around before deciding on which build server works best for you.

When evaluating different solutions, be aware of attempts at vendor lock-in. Also keep in mind that the build server does not in any way replace the need for builds that are well behaved locally on a developer's machine.

Also, as a common rule of thumb, see if the tool is configurable via configuration files. While management tends to be impressed by graphical configuration, developers and operations personnel rarely like being forced to use a tool that can only be configured via a graphical user interface.

Collating quality measures

A useful thing that a build server can do is the collation of software quality metrics. Jenkins has some support for this out of the box. Java unit tests are executed and can be visualized directly on the job page.

Another more advanced option is using the Sonar code quality visualizer, which is shown in the following screenshot. Sonar tests are run during the build phase and propagated to the Sonar server, where they are stored and visualized.

A Sonar server can be a great way for a development team to see the fruits of their efforts at improving the code base.

The drawback of implementing a Sonar server is that it sometimes slows down the builds. The recommendation is to perform the Sonar builds in your nightly builds, once a day.