Unit 04

Build System

You need a system to build your code, and you need somewhere to build it.

Jenkins is a flexible open source build server that grows with your needs. Some alternatives to Jenkins will be explored as well.

We will also explore the different build systems and how they affect our DevOps work.

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.

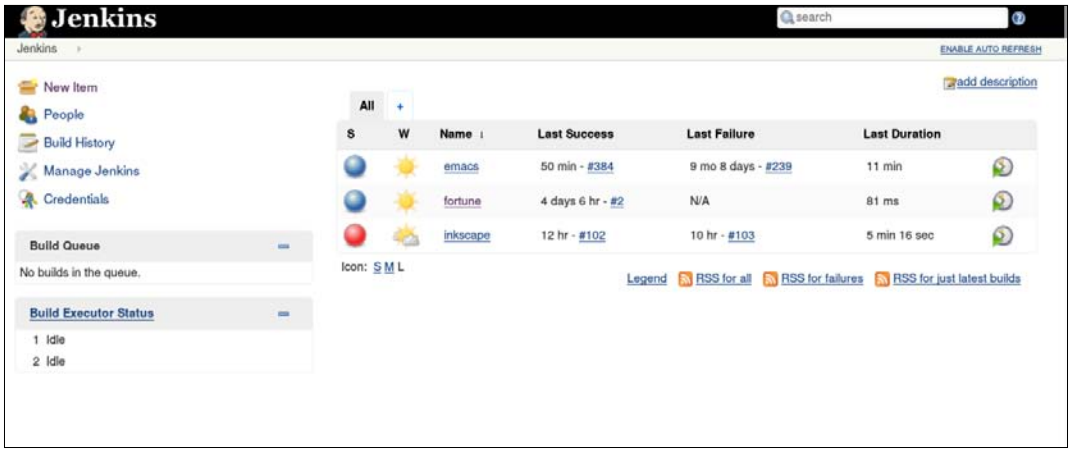
Setting up a basic Jenkins server is not particularly hard at the outset. In Fedora, you can just install it via `dnf`:

```
dnf install jenkins
```

Jenkins is handled as a service via `systemd`:

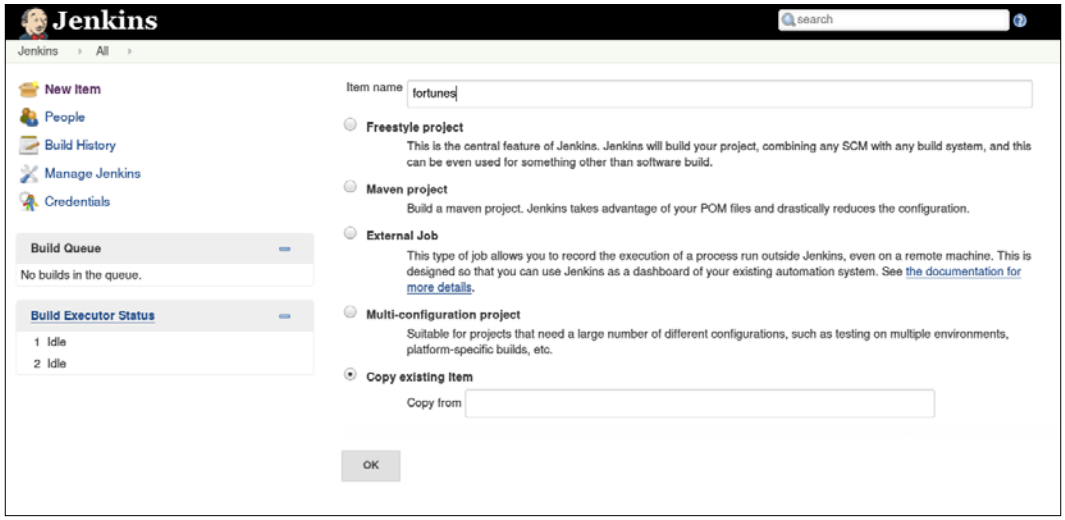
```
systemctl start jenkins
```

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

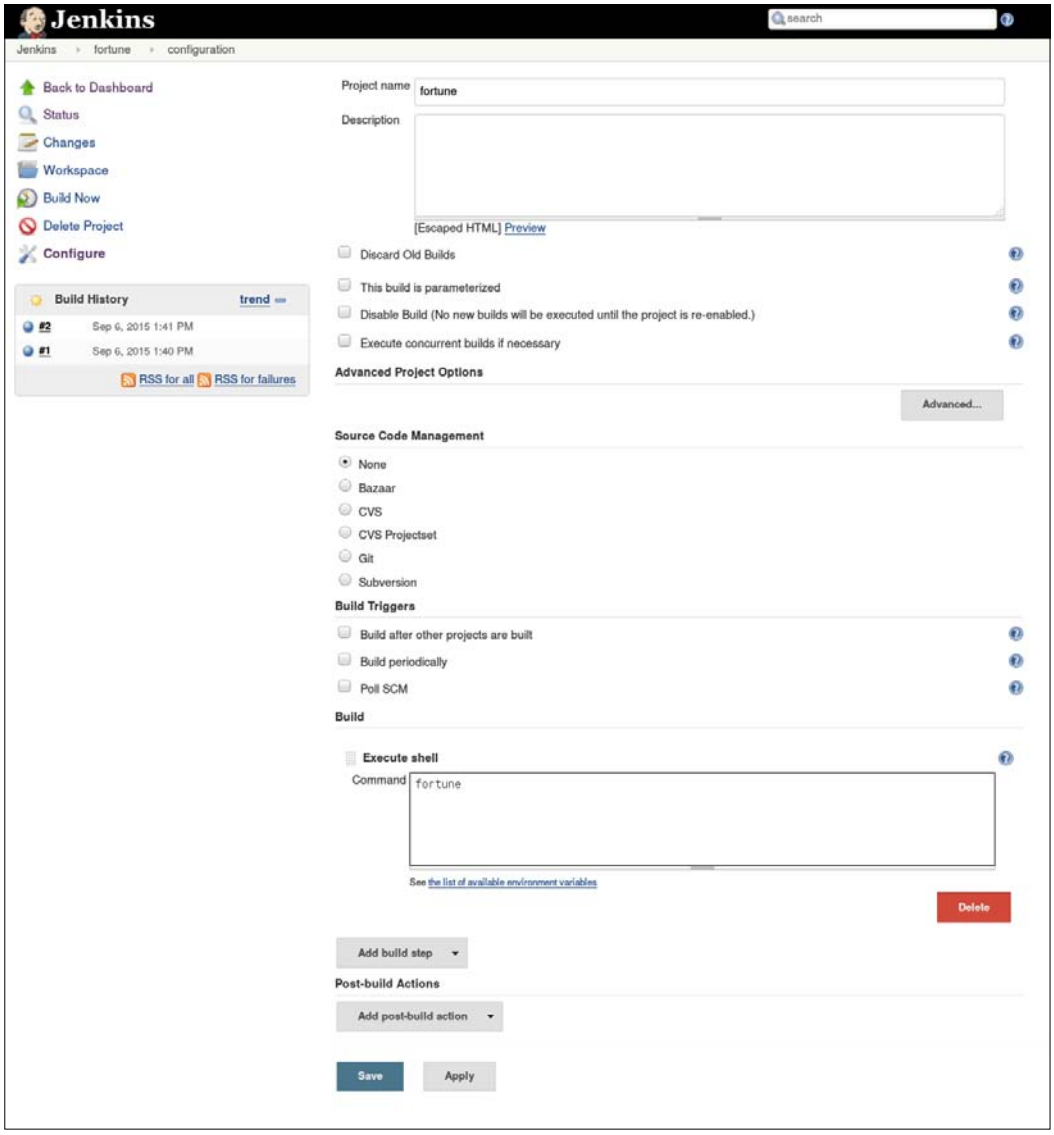


The Jenkins instance in the screenshot has a couple of jobs already defined. The fundamental entity in Jenkins is the job definition, and there are several types to choose from. Let's create a simple job in the web interface. To keep it simple, this job will just print a classic Unix `fortune` quote:

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



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



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.

If an optional dependency is missing, such as image libraries for image support, the optional feature will not be available in the final executable. You can still build the program, but you won't get features that your build machine isn't prepared for.

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.

The final artifact

After finishing the build using the RPM system, you get an RPM file, which is a very convenient type of deployment artifact for operating systems based on Red Hat. For Debian-based distributions, you get a `.deb` file.

The final output from a Maven build is usually an enterprise archive, or EAR file for short. This contains Java Enterprise applications.

It is final deployment artifacts such as these that we will later deploy to our production servers.

In this chapter, we concern ourselves with building the artifacts required for deployment, and in *Chapter 7*, *Deploying the Code*, we talk about the final deployment of our artifacts.

However, even when building our artifacts, we need to understand how to deploy them. At the moment, we will use the following rule of thumb: OS-level packaging is preferable to specialized packaging. This is my personal preference, and others might disagree.

Let's briefly discuss the background for this rule of thumb as well as the alternatives.

As a concrete example, let's consider the deployment of a Java EAR. Normally, we can do this in several ways. Here are some examples:

- Deploy the EAR file as an RPM package through the mechanisms and channels available in the base operating system
- Deploy the EAR through the mechanisms available with the Java application server, such as JBoss, WildFly, and Glassfish

It might superficially look like it would be better to use the mechanism specific to the Java application server to deploy the EAR file, since it is specific to the application server anyway. If Java development is all you ever do, this might be a reasonable supposition. However, since you need to manage your base operating system anyway, you already have methods of deployment available to you that are possible to reuse.

Also, since it is quite likely that you are not just doing Java development but also need to deploy and manage HTML and JavaScript at the very least, it starts to make sense to use a more versatile method of deployment.

Nearly all the organizations I have experience of have had complicated architectures comprising many different technologies, and this rule of thumb has served well in most scenarios.

The only real exception is in mixed environments where Unix servers coexist with Windows servers. In these cases, the Unix servers usually get to use their preferred package distribution method, and the Windows servers have to limp along with some kind of home-brewed solution. This is just an observation and not a condoning of the situation.

Cheating with FPM

Building operating system deliverables such as RPMs with a spec file is very useful knowledge. However, sometimes you don't need the rigor of a real spec file. The spec file is, after all, optimized for the scenario where you are not yourself the originator of the code base.

There is a Ruby-based tool called FPM, which can generate source RPMs suitable for building, directly from the command line.

The tool is available on GitHub at `https://github.com/jordansissel/fpm`.

On Fedora, you can install FPM like this:

```
yum install rubygems
yum install ruby
yum install ruby-devel gcc
gem install fpm
```

This will install a shell script that wraps the FPM Ruby program.

One of the interesting aspects of FPM is that it can generate different types of package; among the supported types are RPM and Debian.

Here is a simple example to make a "hello world" shell script:

```
#!/bin/sh
echo 'Hello World!'
```

We would like the shell script to be installed in `/usr/local/bin`, so create a directory in your home directory with the following structure:

```
$HOME/hello/usr/local/bin/hello.sh
```

Make the script executable, and then package it:

```
chmod a+x usr/local/bin/hello.sh
fpm -s dir -t rpm -n hello-world -v 1 -C installdir usr
```

This will result in an RPM with the name `hello-world` and version `1`.

To test the package, we can first list the contents and then install it:

```
rpm -qivp hello-world.rpm
rpm -ivh hello-world.rpm
```

The shell script should now be nicely installed in `/usr/local/bin`.

FPM is a very convenient method for creating RPM, Debian, and other package types. It's a little bit like cheating!

Continuous Integration

The principal benefit of using a build server is achieving Continuous Integration. Each time a change in the code base is detected, a build that tests the quality of the newly submitted code is started.

Since there might be many developers working on the code base, each with slightly different versions, it's important to see whether all the different changes work together properly. This is called **integration testing**. If integration tests are too far apart, there is a growing risk of the different code branches diverging too much, and merging is no longer easy. The result is often referred to as "merge hell". It's no longer clear how a developer's local changes should be merged to the master branch, because of divergence between the branches. This situation is very undesirable. The root cause of merge hell is often, perhaps surprisingly, psychological. There is a mental barrier to overcome in order to merge your changes to the mainline. Part of working with DevOps is making things easier and thus reducing the perceived costs associated with doing important work like submitting changes.

Continuous Integration builds are usually performed in a more stringent manner than what developers do locally. These builds take a longer time to perform, but since performant hardware is not so expensive these days, our build server is beefy enough to cope with these builds.

If the builds are fast enough to not be seen as tedious, developers will be enthused to check in often, and integration problems will be found early.

Continuous Delivery

After the Continuous Integration steps have completed successfully, you have shiny new artifacts that are ready to be deployed to servers. Usually, these are test environments set up to behave like production servers.

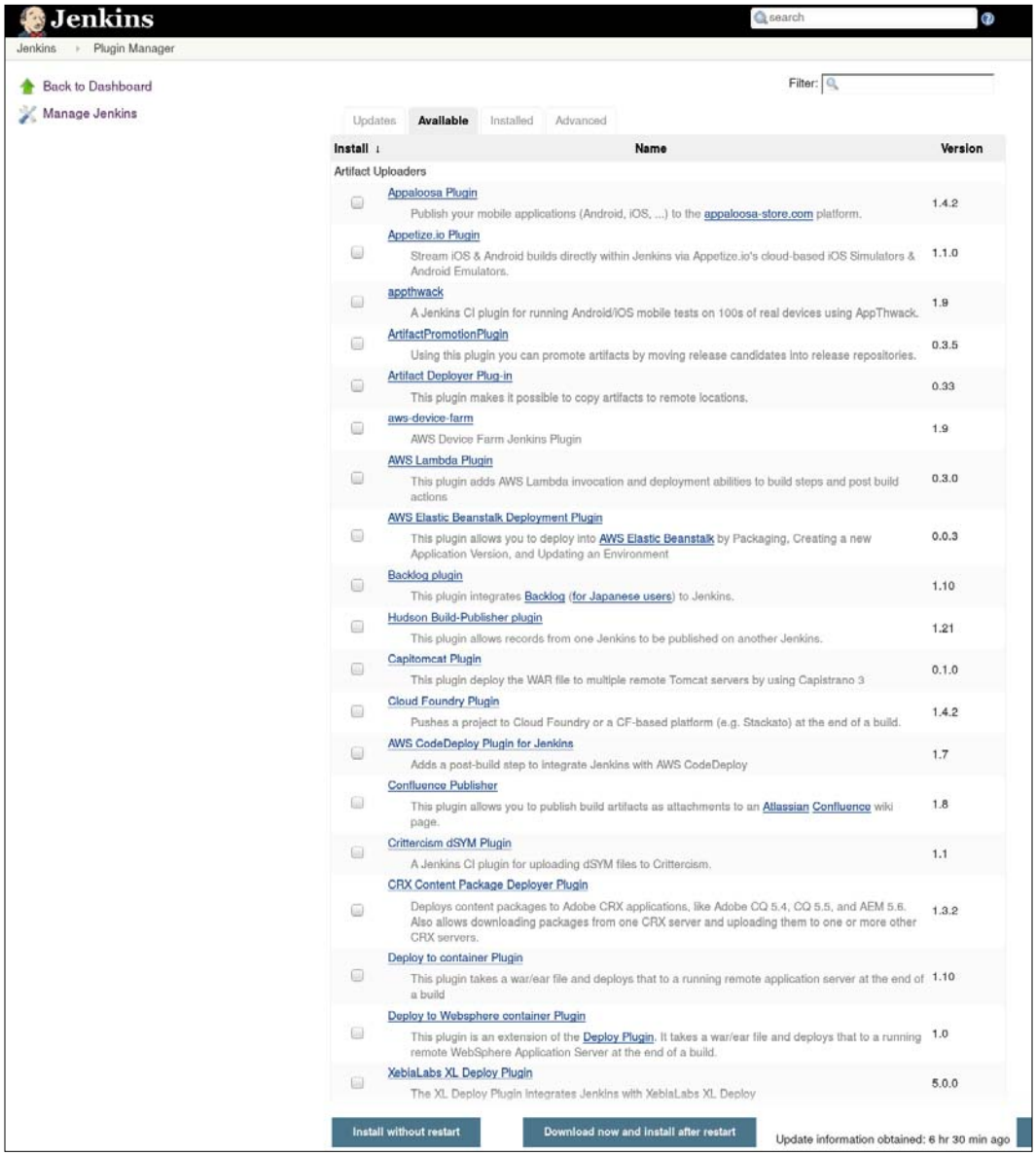
We will discuss deployment system alternatives later in the book.

Often, the last thing a build server does is to deploy the final artifacts from the successful build to an artifact repository. From there, the deployment servers take over the responsibility of deploying them to the application servers. In the Java world, the Nexus repository manager is fairly common. It has support for other formats besides the Java formats, such as JavaScript artifacts and Yum channels for RPMs. Nexus also supports the Docker Registry API now.

Using Nexus for RPM distributions is just one option. You can build Yum channels with a shell script fairly easily.

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:



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 Leingingen 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.

There are several methods to add build slaves to a Jenkins master; see `https://wiki.jenkins-ci.org/display/JENKINS/Distributed+builds`.

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.

A note on cross-compiling

While it's possible to use Windows build slaves, sometimes it's actually easier to use Linux to build Windows software. C compilers such as GCC can be configured to perform cross-compilation using the MinGW package.

Whether or not this is easier very much depends on the software being built.

A big system usually comprises many different parts, and some of the parts might contain native code for different platforms.

Here are some examples:

- Native android components
- Native server components coded in C for efficiency
- Native client components, also coded in C or C++ for efficiency

The prevalence of native code depends a bit on the nature of the organization you are working with. Telecom products often have a lot of native code, such as codecs and hardware interface code. Banking systems might have high-speed messaging systems in native code.

An aspect of this is that it is important to be able to build all the code that's in use conveniently on the build server. Otherwise, there's a tendency for some code to be only buildable on some machine collecting dust under somebody's desk. This is a risk that needs to be avoided.

What your organization's systems need, only you can tell.

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.

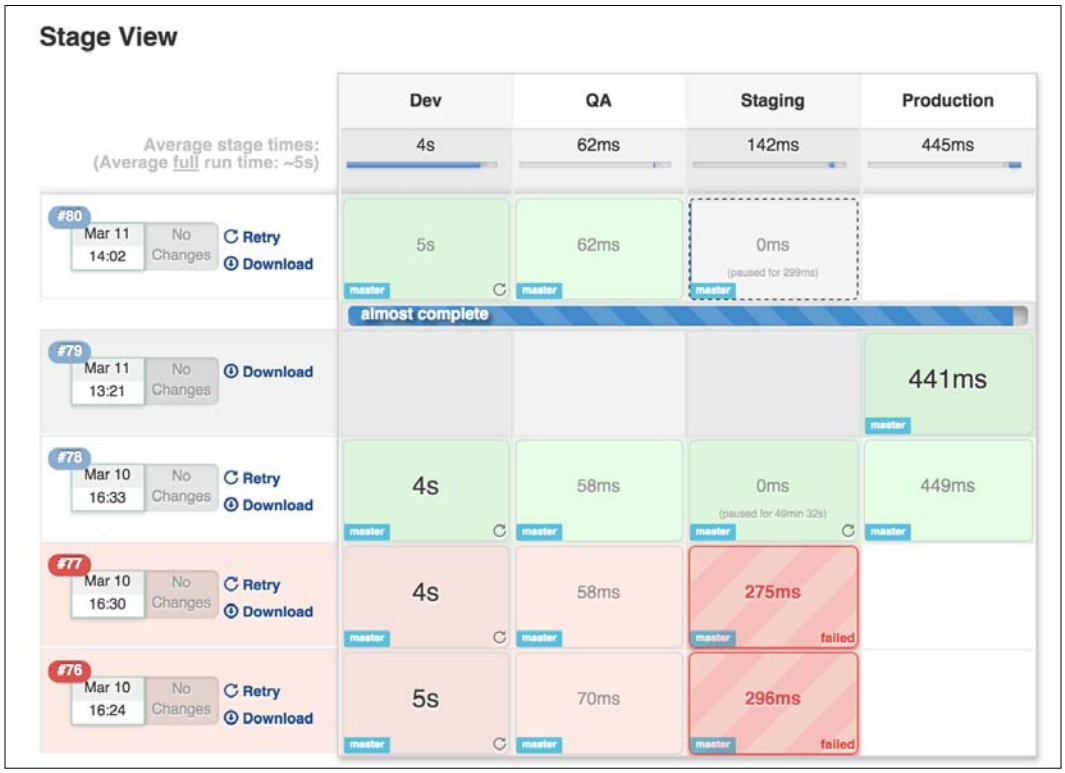
There are many plugins that create improved pipelines for Jenkins, and the fact that there are several shows that there is indeed a great desire for improvements in this area.

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:



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.

A look at the Jenkins filesystem layout

It is often useful to know where builds wind up in the filesystem.

In the case of the Fedora package, the Jenkins jobs are stored here:

```
/var/lib/jenkins/jobs
```

Each job gets its own directory, and the job description XML is stored in this directory as well as a directory for the build called workspace. The job's XML files can be backed up to another server in order to be able to rebuild the Jenkins server in the event of a catastrophic failure. There are dedicated backup plugins for this purpose as well.

Builds can consume a lot of space, so it may sometimes happen that you need to clean out this space manually.

This shouldn't be the normal case, of course. You should configure Jenkins to only leave the number of builds you have space for. You can also configure your configuration management tool to clear out space if needed.

Another reason you might need to delve into the filesystem is when a build mysteriously fails, and you need to debug the cause of the failure. A common cause of this is when the build server state does not meet expectations. For a Maven build, broken dependencies could be polluting the local repository on the build server, for example.

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:

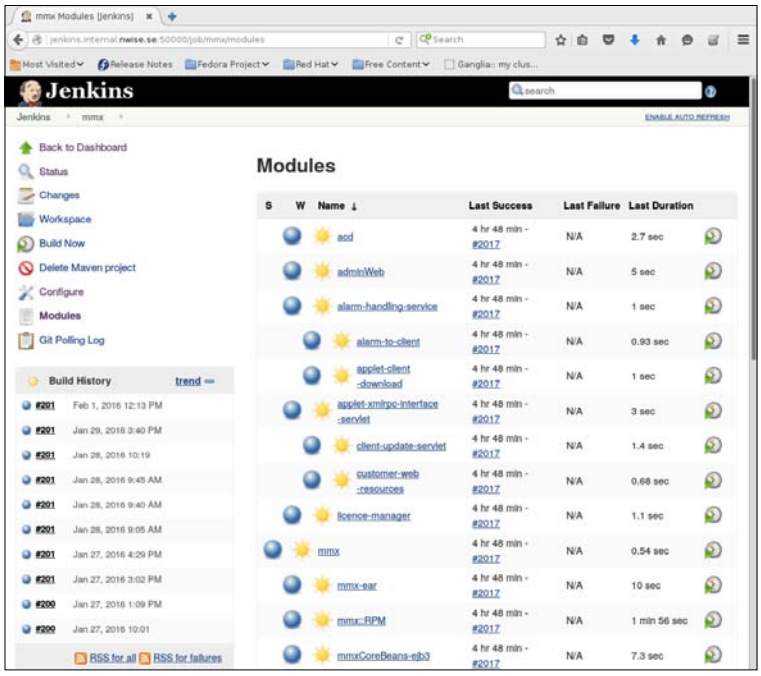
```
a.out : b.o c.o
b.o : b.c
c.o : c.c
```

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.



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.

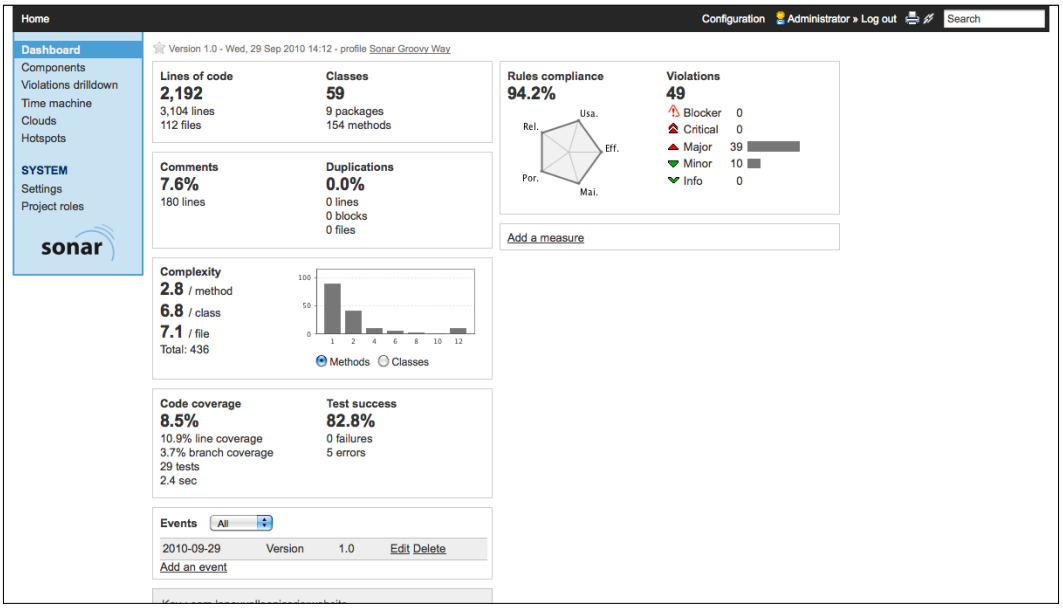
Collating quality measures

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.



About build status visualization

The build server produces a lot of data that is amenable to visualization on a shared display. It is useful to be immediately aware that a build has failed, for instance.

The easiest thing is to just hook up a monitor in a kiosk-like configuration with a web browser pointing to your build server web interface. Jenkins has many plugins that provide a simplified job overview suitable for kiosk displays. These are sometimes called **information radiators**.

It is also common to hook up other types of hardware to the build status, such as lava lamps or colorful LED lamps.

In my experience, this kind of display can make people enthusiastic about the build server. Succeeding with having a useful display in the long run is more tricky than it would first appear, though. The screen can be distracting. If you put the screen where it's not easily seen in order to circumvent the distraction, the purpose of the display is defeated.

A lava lamp in combination with a screen placed discreetly could be a useful combination. The lava lamp is not normally lit, and thus not distracting. When a build error occurs, it lights up, and then you know that you should have a look at the build information radiator. The lava lamp even conveys a form of historical record of the build quality. As the lava lamp lights up, it grows warm, and after a while, the lava moves around inside the lamp. When the error is corrected, the lamp cools down, but the heat remains for a while, so the lava will move around for a time proportional to how long it took to fix the build error.

Taking build errors seriously

The build server can signal errors and code quality problems as much as it wants; if developer teams don't care about the problems, then the investment in the notifications and visualization is all for nought.

This isn't something that can be solved by technical means alone. There has to be a process that everybody agrees on, and the easiest way for a consensus to be achieved is for the process to be of obvious benefit to everyone involved.

Part of the problem is organizations where everything is on fire all the time. Is a build error more important than a production error? If code quality measures estimate that it will take years to improve a code base's quality, is it worthwhile to even get started with fixing the issues?

How do we solve these kinds of problems?

Here are some ideas:

- Don't overdo your code quality metrics. Reduce testing until reports show levels that are fixable. You can add tests again after the initial set of problems is taken care of.
- Define a priority for problems. Fix production issues first. Then fix build errors. Do not submit new code on top of broken code until the issues are resolved.

Robustness

While it is desirable that the build server becomes one of the focal points in your Continuous Delivery pipeline, also consider that the process of building and deployment should not come to a standstill in the event of a breakdown of the build server. For this reason, the builds themselves should be as robust as possible and repeatable on any host.

This is fairly easy for some builds, such as Maven builds. Even so, a Maven build can exhibit any number of flaws that makes it non-portable.

A C-based build can be pretty hard to make portable if one is not so fortunate as to have all build dependencies available in the operating system repositories. Still, robustness is usually worth the effort.

Summary

In this chapter, we took a whirlwind tour through the systems that build our code. We had a look at constructing a Continuous Integration server with Jenkins. We also examined a number of problems that might arise, because the life of a DevOps engineer is always interesting but not always easy.

In the next chapter, we will continue our efforts to produce code of the highest quality by studying how we can integrate testing in our workflow.