Module 4

Building the code and testing

Why do we build code?

Software building is define 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.

Build systems

There are many build systems that have evolved over the history of software development.

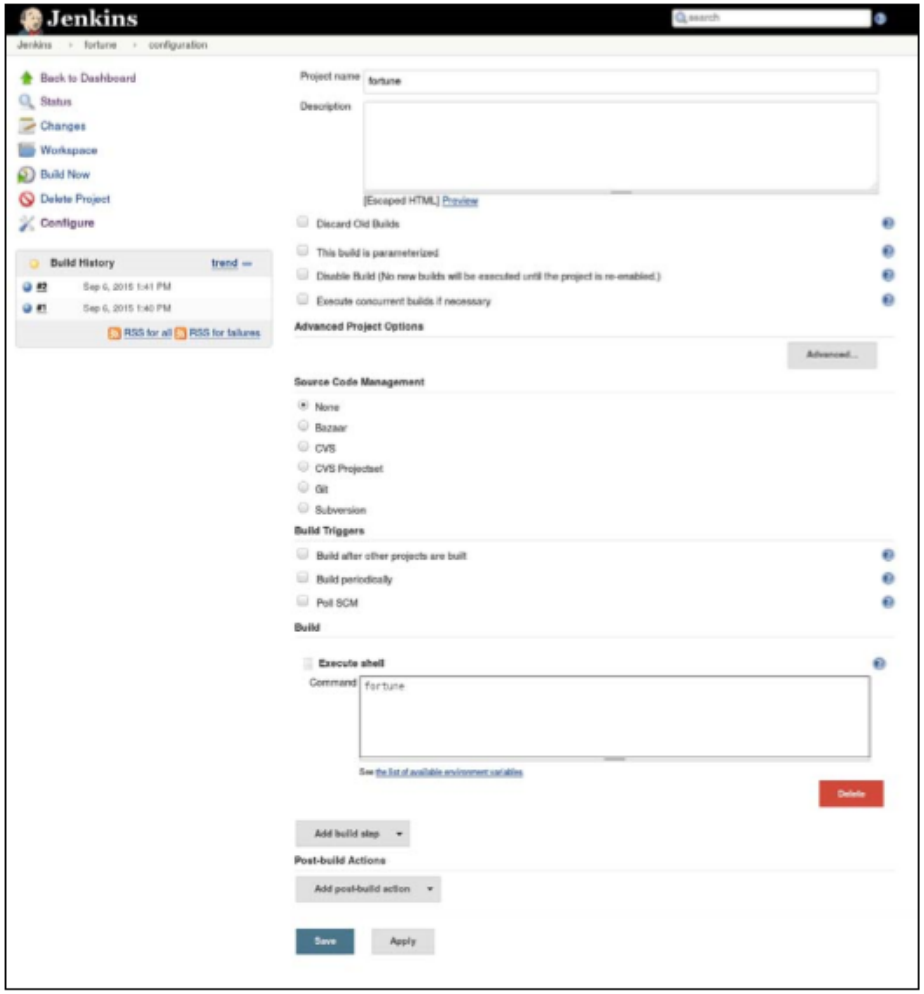
- For Java, there is Maven, Gradle, and Ant
- For C and C++, there is Make in many different flavors
- For JavaScript, there is Grunt
- For Scala, there is sbt
- For Ruby, we have Rake

Jenkins build server

A build server is, in essence, a system that builds software based on various triggers.

- Jenkins is a popular build server written in Java.
- Jenkins is a fork of the Hudson build server.
- Jenkins has special support for building Java code but is in no way limited to just building Java.

Jenkins build server



How-to-build-a-Jenkins Build Server

Please write the steps.

How-to-build-a-Jenkins Build Server

Setup Jenkins Job

- Go to Jenkins top page, select “New Job”, then choose “Build a free-style software project”.
  Then you can tell the elements of this freestyle job:
  - Optional SCM, such as CVS or Subversion where your source code resides.
  - Optional triggers to control when Jenkins will perform builds.
  - Some sort of build script that performs the build (ant, maven, shell script, batch file, etc.) where the real work happens.
  - Optional steps to collect information out of the build, such as archiving the artifacts and/or recording java doc and test results.
  - Optional steps to notify other people/systems with the build result, such as sending e-mails, IMs, updating issue tracker, etc..

Managing 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.

- https://cloudacademy.com/course/implementing-dependency-management-with-azure-devops/what-is-dependency-management/
- https://docs.gradle.org/current/userguide/core_dependency_management.html
- https://docs.microsoft.com/en-us/learn/modules/manage-build-dependencies/2-plan-build-dependencies-for-your-pipeline

Managing build dependencies

POM File:

A Project Object Model or POM is the fundamental unit of work in Maven. It is an XML file that contains information about the project and configuration details used by Maven to build the project.

- It contains default values for most projects.
- Examples for this is the build directory, which is target; the source directory, which is src/main/java; the test source directory, which is src/test/java; and so on. When executing a task or goal, Maven looks for the POM in the current directory. It reads the POM, gets the needed configuration information, then executes the goal.
- Some of the configuration that can be specified in the POM are the project dependencies, the plugins or goals that can be executed, the build profiles, and so on.
- Other information such as the project version, description, developers, mailing lists and such can also be specified.

The final Artifact

The final output from a Maven build is usually an enterprise archive, or Enterprise archive file(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.

- The result of a build is called artifact and can be, for example, an executable JAR file or zip file.
- 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.

The final Artifact

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 **Red-hat Package Manager(**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

RPM Package Manager is a free and open-source package management system. The name RPM refers to .rpm file format and the package manager program itself. RPM was intended primarily for Linux distributions; the file format is the baseline package format of the Linux Standard Base.

An RPM package typically contains **binary executables, along with relevant configuration files and documentation**. The rpm program is a powerful package manager, which can be used to install, query, verify, update, erase and build software packages in the RPM format.

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
- 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.

- 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.
  - Artifact repository is a collection of binary software artifacts and metadata stored in a defined directory structure which is used by clients such Maven, Mercury, or Ivy to retrieve binaries during a build process.
  - A Maven repository provides a platform for the storage, retrieval, and management of binary software artifacts and metadata.

4.2 Build server and Phases

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.

Build slaves

There are several methods to add build slaves to a Jenkins master;

See https://www.edureka.co/blog/jenkins-master-and-slave-architecture-a-complete-guide/.

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 Network Launch Protocol(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.

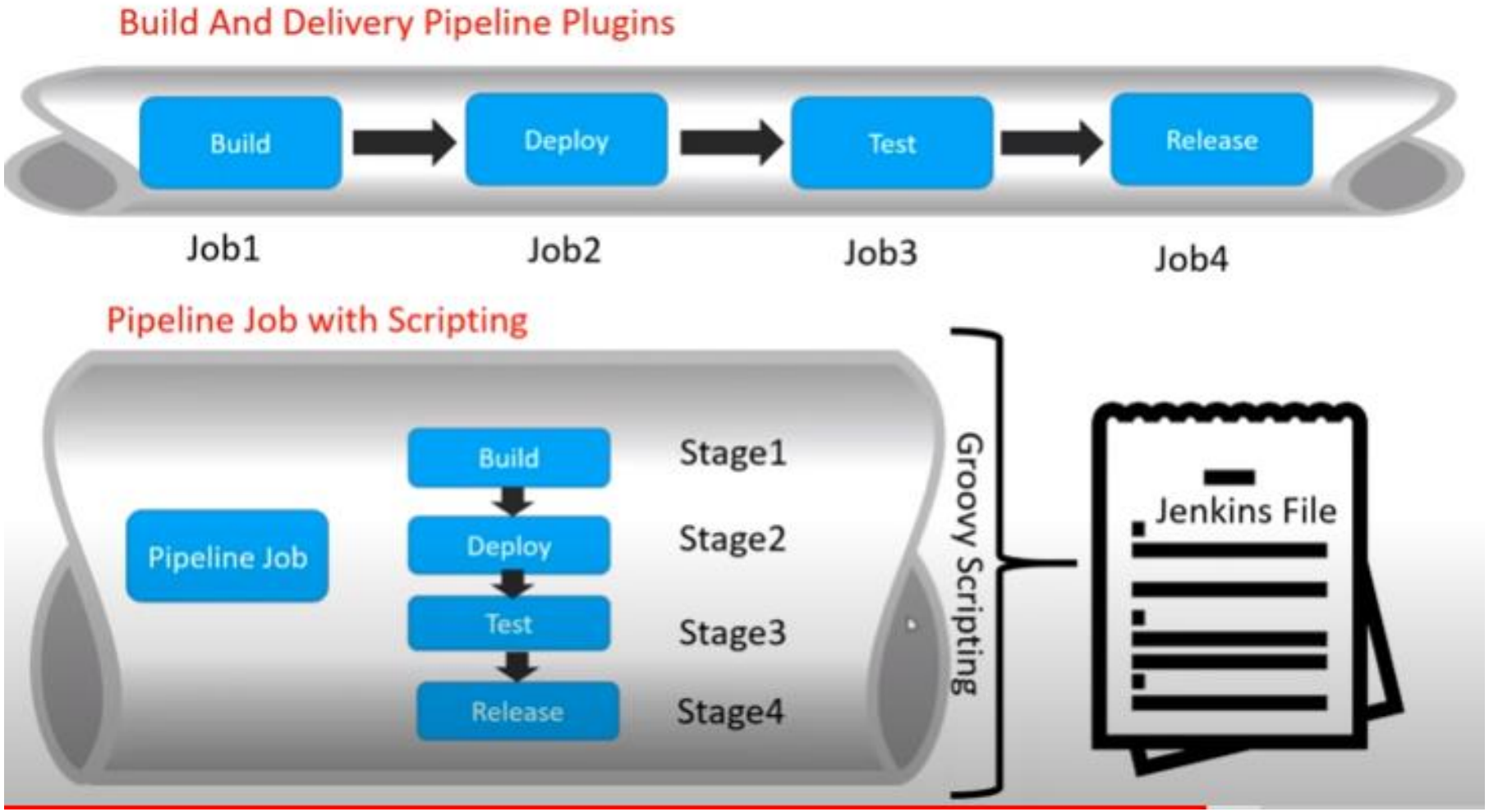
• An upstream build can trigger a downstream build. You can also let the successful build of one job trigger another job.

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.

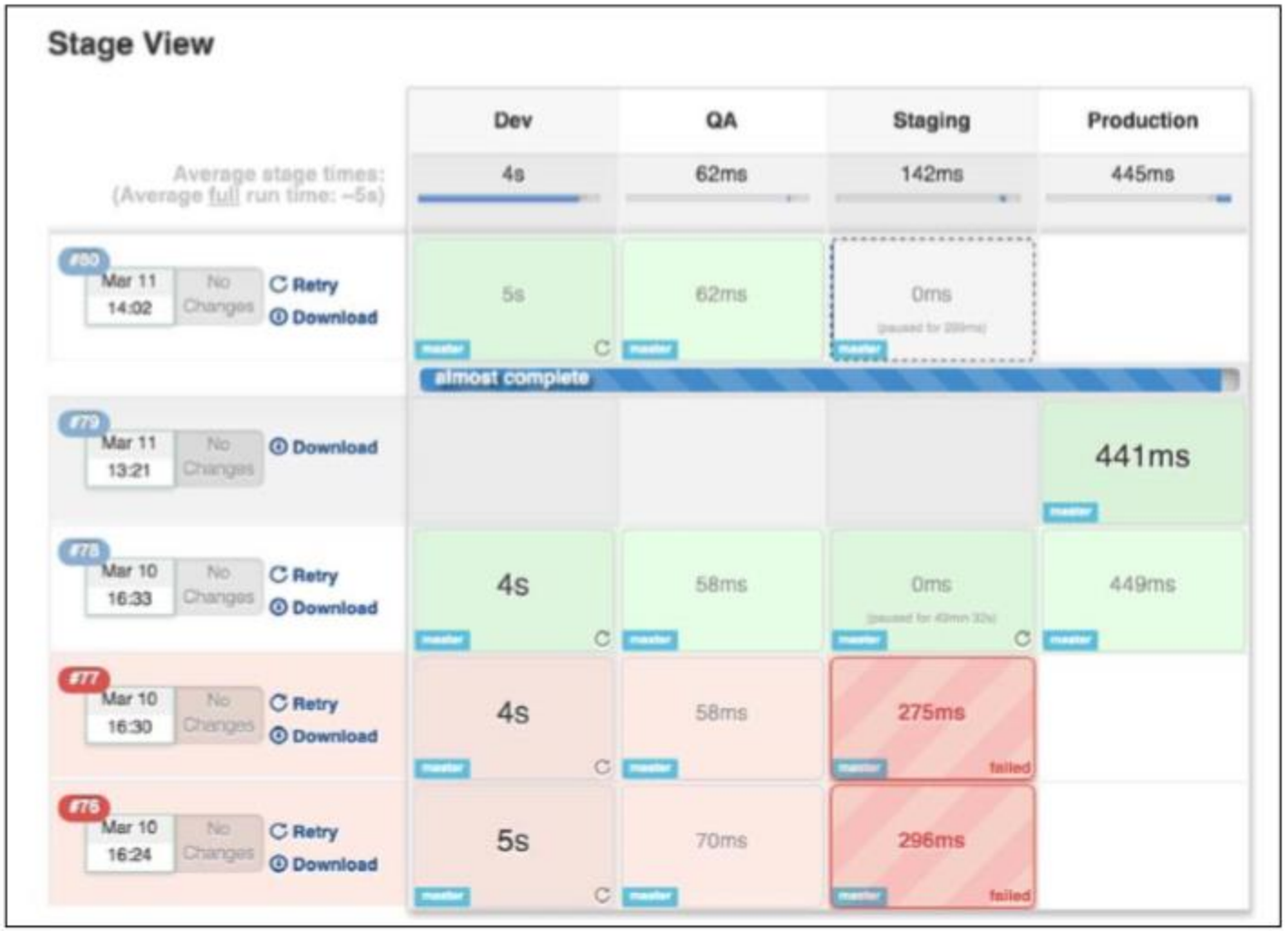
- In Jenkins terminology, the first build in a chain is called the upstream build, and the second one is called the downstream build.
- While this way of chaining builds is often sufficient, there will most likely be a need for greater control of the build chain eventually. Such a build chain is often called a pipeline or workflow.
- There are many plugins that create improved pipelines for Jenkins:
- 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.
  - Jenkins Pipeline Tutorial: https://www.guru99.com/jenkins-pipeline-tutorial.html
  - Jenkins Pipeline Tutorial | CI/CD Pipeline Jenkins | Jenkins Tutorial | DevOps Training Edureka: https://www.youtube.com/watch?v=_fmX31VFbL8&t=1796s

Job chaining and build pipelines



Job chaining and build pipelines

An example workflow is illustrated here:



Declarative pipeline and scripted pipeliine

- The **key difference** between Declarative pipeline and Scripted pipeline would be with respect to their **syntaxes** and their **flexibility**.
- Declarative pipeline is a relatively new feature that supports the pipeline as code concept. It makes the pipeline code easier to read and write. This code is written in a Jenkinsfile which can be checked into a source control management system such as Git.
- Whereas, the scripted pipeline is a traditional way of writing the code. In this pipeline, the Jenkinsfile is written on the Jenkins UI instance.
- Though both these pipelines are based on the groovy DSL, the scripted pipeline uses stricter groovy based syntaxes because it was the first pipeline to be built on the groovy foundation. Since this Groovy script was not typically desirable to all the users, the declarative pipeline was introduced to offer a simpler and more optioned Groovy syntax.
- The declarative pipeline is defined within a block labelled ‘pipeline’ whereas the scripted pipeline is defined within a ‘node’.
  - https://www.edureka.co/community/54705/difference-between-declarative-pipeline-scripted-pipeliine
  - https://www.edureka.co/blog/jenkins-pipeline-tutorial-continuous-deliver

These lifecycle phases (plus the other lifecycle phases not shown here) are executed sequentially to complete the default lifecycle. Given the lifecycle phases abov

Build phases

A Build Lifecycle is Made Up of Phases

Each of these build lifecycles is defined by a different list of build phases, wherein a build phase represents a stage in the lifecycle.

For example, the default lifecycle comprises of the following phases (for a complete list of the lifecycle phases, refer to the Lifecycle Reference):

- `validate` - validate the project is correct and all necessary information is available
- `compile` - compile the source code of the project
- `test` - test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- `package` - take the compiled code and package it in its distributable format, such as a JAR.
- `verify` - run any checks on results of integration tests to ensure quality criteria are met
- `install` - install the package into the local repository, for use as a dependency in other projects locally
- `deploy` - done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

These lifecycle phases (plus the other lifecycle phases not shown here) are executed sequentially to complete the default lifecycle. Given the lifecycle phases above, this means that when the default lifecycle is used, Maven will first validate the project, then will try to compile the sources, run those against the tests, package the binaries (e.g. jar), run integration tests against that package, verify the integration tests, install the verified package to the local repository, then deploy the installed package to a remote repository.

https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

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.
- The Continuous Integration server needs to be very good at catching errors, and automated testing is very important for achieving that goal.

Build servers and infrastructure as code

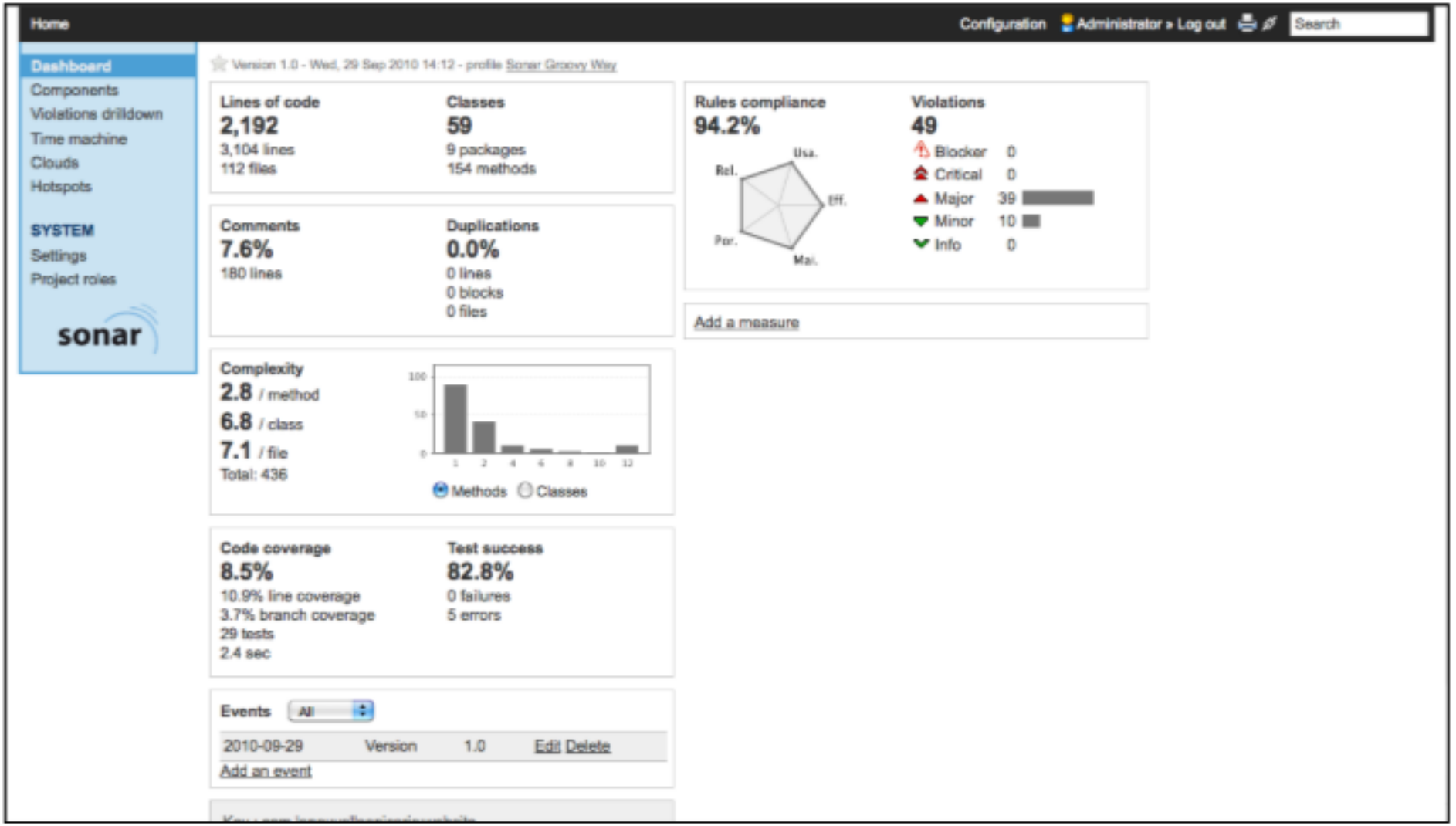
- While we are discussing the Jenkins file structure, it is useful to note an impedance mismatch that often occurs between GUI-based tools such as Jenkins and the DevOps axiom that infrastructure should be described as code.
- One way to understand this problem is that while Jenkins job descriptors are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

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.
  - What is SonarQube | Introduction SonarQube | SonarQube Tutorial | SonarQube Basics | Intellipaat: https://www.youtube.com/watch?v=efHas35j_yg

Collating quality measures



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.

About build status visualization

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.

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.