CHAPTER: 1

INTRODUCING JENKINS

* 1. INTRODUCTION

Continuous Integration, also know as CI, is a cornerstone of modern software development. A good CI infrastructure can streamline the development process right through to deployment, help detect and fix bugs faster, provide a useful project dashboard for both developers and non-developers, and ultimately, help teams deliver more real business value to the end user.

* 1. CI Fundamentals

Back in the days of waterfall projects and Gantt charts, before the introduction of CI practices,

development team time and energy was regularly drained in the period leading up to a release by what was known as the Integration Phase. During this phase, the code changes made by individual developers or small teams were brought together piecemeal and forged into a working product. This was hard work, sometimes involving the integration of months of conflicting changes. It was very hard to anticipate the types of issues that would crop up, and even harder to fix them, as it could involve reworking code that had been written weeks or months before. This painful process, fraught with risk and danger, often lead to significant delivery delays, unplanned costs and, as a result, unhappy clients. Continuous Integration was born to address these issues.

Continuous Integration, in its simplest form, involves a tool that monitors your version control system for changes. Whenever a change is detected, this tool automatically compiles and tests your application. If something goes wrong, the tool immediately notifies the developers so that they can fix the issue immediately.

But Continuous Integration can do much more than this. Continuous Integration can also help you keep tabs on the health of your code base, automatically monitoring code quality and code coverage metrics, and helping keep technical debt down and maintenance costs low. The publicly-visible code quality metrics can also encourage developers to take pride in the quality of their code and strive to improve it. Combined with automated end-to-end acceptance tests, CI can also act as a communication tool, publishing a clear picture of the current state of development efforts. And it can simplify and accelerate delivery by helping you automate the deployment process, letting you deploy the latest version of your application either automatically or as a one-click process.

Continuous Integration can open and facilitate communication channels between team members and encourage collaborative problem solving and process improvement. And, by automating the deployment process, Continuous Integration helps you get your software into the hands of the testers and the end users faster, more reliably, and with less effort. The practice of automatically deploying every successful build directly into production is generally known as Continuous Deployment.

The notion of Continuous Delivery is a slight variation on the idea of Continuous Deployment that takes into account these considerations. With Continuous Delivery, any and every successful build that has passed all the relevant automated tests and quality gates can potentially be deployed into production via a fully automated one-click process, and be in the hands of the end-user within minutes. However, the process is not automatic: it is the business, rather than IT, that decides the best time to deliver the latest changes.

But Continuous Integration is a mindset as much as a toolset. To get the most out of CI, a team needs to adopt a CI mentality. For example, your projects must have a reliable, repeatable, and automated build process, involving no human intervention. Fixing broken builds should take an absolute priority, and not be left to stagnate. The deployment process should be automated, with no manual steps involved. And since the trust you place in your CI server depends to a great extent on the quality of your tests, the team needs to place a very strong emphasis on high quality tests and testing practices.

* 1. INTRODUCING JENKINS

Jenkins, originally called Hudson, is an open source Continuous Integration tool written in Java.

Boasting a dominant market share, Jenkins is used by teams of all sizes, for projects in a wide variety of languages and technologies, including .NET, Ruby, Groovy, Grails, PHP and more, as well as Java.

However Jenkins does not sacrifice power or extensibility: it is also extremely flexible and easy to adapt to your own purposes. Hundreds of open source plugins are available, with more coming out every week. These plugins cover everything from version control systems, build tools, code quality metrics, build notifies, integration with external systems, UI customization, games, and much more. And installing them is quick and easy.

* 1. From Hudson to Jenkins-A short History
  2. Should I use Jenkins or Hudson?
* Enterprise integration and Sonatype tools. Hudson is likely to place a strong emphasis on integration with enterprise tools such as LDAP/Active Directory, and the Sonatype products such as Maven 3, Nexus and M2Ecipse, whereas Jenkins is more open to other competing tools such as Artifactory and Gradle.
* Plugin architecture. If you intend to write your own Jenkins/Hudson plugins, you should be aware that Sonatype is working on providing JSR-330 dependency injection for Hudson plugins. New developers may find this approach easier to use, though it does raise issues about future plugin compatibility between Jenkins and Hudson.
  1. Introducing CI into your Organization

Continuous Integration is not an all-or-nothing affair. In fact, introducing CI into an organization takes you on a path that progresses through several distinct phases. Each of these phases involves incremental improvements to the technical infrastructure as well as, perhaps more importantly, improvements in the practices and culture of the development team itself.

* + 1. Phase 1-No Build Server
* No central build server
* Builds are built manually on developer’s machine using Ant scripts or similar to do so.
* A central source code repository.
* No regular basis commits to repo.
  + 1. Phase 2-Nightly Builds
* Central build server available
* Automated build are scheduled on regular basis
* Here automated tests using unit tests are not mandatory part of build process (i.e. no reliable and repeatable unit tests).
* Commits are at regular basis
* Build servers are used to alert the team via email regarding commit conflicts.
* But broken builds may stay on the build server for some time before fix done.
  + 1. Phase 3-Nightly builds and Basic Automated Tests
* Build server kick off the build whenever new code is committed to the version control system.
* Also able to see what changes in the source code triggered a particular build and what issues these changes address.
* Build script compiles the application and runs a set of automated unit and integration tests
* Also alerts through Instant Messaging. And broken builds are fixed quickly.
  + 1. Phase 4-Enter the Metrics
* Now quality of code can be analyzed using automated code quality and code coverage metrics.
* Code quality build automatically generates API doc for application.
* “Build radiator” also set up i.e. dashboard view of the project status.
* All this helps teams keep the quality of the code base high, alerting team members if good testing practices are slipping.
  + 1. Phase 5-Getting more serious about testing
* The benefits of Continuous Integration are closely related to solid testing practices. Now, practices like **Test-Driven Developmen**t are more widely practiced, resulting in a growing confidence in the results of the automated builds. The application is no longer simply compiled and tested, but if the tests pass, **it is automatically deployed to an application server for more comprehensive end-to-end tests and performance tests.**
  + 1. Phase 6-Automated Acceptance Test and More Automated Deployment
* These automated tests use Behavior-Driven Development and Acceptance-Test Driven Development tools to act as communication and documentation tools and documentation as much as testing tools, publishing reports on test results in business terms that non-developers can understand. Since these high-level tests are automated at an early stage in the development process, they also provide a clear idea of what features have been implemented, and which remain to be done.
* The application is automatically deployed into test environments for testing by the QA

team either as changes are committed, or on a nightly basis; a version can be deployed (or “promoted”) to UAT and possibly production environments using a manually-triggered build when testers consider it ready. The team is also capable of using the build server to back out a release, rolling back to a previous release, if something goes horribly wrong.

* + 1. Phase 7-Continuous Deployment

CHAPTER 2: YOUR FIRST STEPs WITH JENKINS

2.1 Introduction

Will get Jenkin’s key features…

2.2 Preparing Your Environment

* The most basic function of any Continuous Integration tool is to monitor source code in

a version control system and to fetch and build the latest version of your source code whenever any changes are committed. So you’ll need a version control system. E.g. git or SVN.

* Build tool like Maven (for Java applications).
* As Jenkins is Java web application we need JDK with JRE
* Configuring SSH Keys

2.3 Staring Up Jenkins

a. Using **“Java Web Start”**

Java Web Start is a technology that lets you start up a Java application on your local machine via a URL on a web page—it comes bundled with the Java JRE. In our case, this will start a Jenkins server running on your machine, and let you experiment with it as if it were installed locally.

b. Download Jenkins executable WAR file and run with command line i.e.

$ java –jar jenkins.war

c. By deploying Jenkins WAR file to Java application web servers like tomcat.

2.4 Configuring the Tools (Jenkin’s Home Page >> Manage Jenkins)

* Tell Jenkins about the build tools (i.e. maven) and JDK versions will be using for builds.
* Install and upgrade plugins (i.e. Git plugin)
* Keep track of system load
* Manage distributed build servers
* Security configuration
* Email servers for notification
* Version Control System
* Integration with 3rd party software

2.5 Yours First Jenkins Build Job

Build jobs are at the heart of the Jenkins build process. Simply put, you can think of a Jenkins build job as a particular task or step in your build process. This may involve simply compiling your source code and running your unit tests. Or you might want a build job to do other related tasks, such as running your integration tests, measuring code coverage or code quality metrics, generating technical documentation, or even deploying your application to a web server. A real project usually requires many separate but related build jobs.

Jenkins supports several different types of build jobs. The two most commonly-used are the **freestyle builds and the Maven 2/3 builds.** The freestyle projects allow you to configure just about any sort of build job: they are highly flexible and very configurable. The Maven 2/3 builds understand the Maven project structure, and can use this to let you set up Maven build jobs with less effort and a few extra features. There are also plugins that provide support for other types of build jobs.

In a nutshell, Jenkins works by checking out the source code of your project and building it in its own workspace. So the next thing you need to do is to tell Jenkins where it can find the source code for your project. You do this in the **Source Code Management section** Jenkins provides support for **CVS and Subversion** out of the box, and many others such as **Git, Mercurial, ClearCase, Perforce and many more via plugins.**

Once we have told Jenkins where to find the source code for our application, we need to tell it how often it should check for updates **(Using Build Triggers section)**. We want Jenkins to monitor the repository and start a build whenever any changes have been committed. This is a common way to set up a build job in a Continuous Integration context, as it provides fast feedback if the build fails. Other approaches include building on regular intervals (for example, once a day), requiring a user to kick of the build manually **(i.e. Poll SCM Option)**, or even triggering a build remotely using a “**post-commit**” hook in your SCM.

The next step is to configure the actual build itself using **Build Section**. In a freestyle build job, you can break down your build job into a number of build steps. This makes it easier to organize builds in clean, separate stages. For example, a build might run a suite of functional tests in one step, and then tag the build in a second step if all of the functional tests succeed. In technical terms, a build step might involve invoking an Ant task or a Maven target, or running a shell script. There are also Jenkins plugins that let you use additional types of build steps: Gant, Grails, Gradle, Rake, Ruby, MSBuild and many other build tools are all supported.

Using **Post-Build Action Section** we can configure the outcome of Build task test results using JUnit Testing tool and producing reports in XML format (This is de facto standard in Java world). This format is also used by many other Java testing tools, such as TestNG, Spock and Easyb. Jenkins understands this format, so if your build produces JUnit XML test results, Jenkins can generate nice graphical test reports and statistics on test results over time, and also let you view the details of any test failures. Jenkins also keeps track of how long your tests take to run, both globally, and per test—this can come in handy if you need to track down performance issues. When Maven runs unit tests in a project, it automatically generates the XML test reports in a directory called **surefire-reports in the target directory**.

Another thing you often want to do is to archive your **build results**. Jenkins can store a copy of the binary artifacts generated by your build, allowing you to download the binaries produced by a build directly from the build results page. It will also post the latest binary artifacts on the project home page, which is a convenient way to distribute the latest and greatest version of your application. You can activate this option by ticking the “Archive the artifacts” checkbox and indicating which binary artifacts you want Jenkins to archive. For example, we have configured Jenkins to store all of the JAR files generated by this build job.

2.6 Your First Build Job in Action

Core business model of a Continuous Integration server—kicking off builds when someone changes the code!

1. Commit code change at GitHub from local clone.
2. Build job will trigger and complete with status failure/success at Build Job Home page.
3. Some job status indications

* Red: Due to compile failure
* Yellow: Due to Unit test failure and lack of code coverage ratio.
* Blue: Successful

2.7 More reporting - Displaying Javadocs

For many Java projects, **Javadoc comments are an important source of low-level technical documentation.** There are even tools, such **as UmlGraph**, that let you produce **Javadoc with embedded UML diagrams** to give you a better picture of how the classes fit together in the application.

Note : Get More on UML diagram and Javadoc

Jenkins can integrate Javadoc API documentation directly into the Jenkins website. This way, everyone can find the latest Javadoc easily, in a well known place. Often, this sort of task is performed in a separate build job, but for simplicity we can add another build step by adding “Javadoc:Javadoc –o” in Goals field. And **Under Post-build Action section** select the “**Publish Javadoc**” and mention the path where we can save the Javadoc which are created by “Javadoc:Javadoc” goal.

Once build job is completed a javadoc link is displayed at build job home page w.r.t. build completion.

2.8 Adding Code Coverage and other metrics

To see how the plugins work, we are going to integrate code coverage metrics using the **Cobertura plugin**.

Code coverage is an indication of how much of your application code is actually executed during your tests—it can be a useful tool…

* In particular for finding areas of code that have not been tested by your test suites.
* It can also give some indication as to how well a team is applying good testing practices such as Test-Driven Development or Behavior-Driven Development.

To configure code coverage feature…under **build step** by adding “**cobertura:cobertura**” in Goals field. And **Under Post-build Action section** select the “**Publish Cobertura Coverage Report**” and mention the path where we can save the code coverage report (i.e. in XML form) which are created by “cobertura:cobertura” goal.

We can also configure build acceptance level (like minimum acceptance level of code coverage and test coverage) by fine tuning using “**Coverage Metrics Target**” option under **Post-Build action section.**

After running the build for code coverage you will see coverage statistics for your build displayed on the build home page, along with a Coverage Report link when you can go for more details.

Code coverage metrics are a great way to isolate code that has not been tested, in order to add extra tests for corner cases that were not properly tested during the initial development, for example. The Jenkins code coverage graphs are also a great way of keeping track of your code coverage metrics as the project grows. Indeed, as you add new tests, you will notice that Jenkins will display a graph of code coverage over time, not just the latest results.

CHAPTER 3: INSTALLING JENKINS

3.1 Introduction

3.2 Downloading and Installing Jenkins

Since Jenkins is a Java application, you will need a recent version of Java on your machine.

3.3 Preparing a Build Server for Jenkins

3.4 The Jenkins Home Directory

3.5 Installing Jenkins on Debian or Ubuntu

3.8 Running Jenkins as a Stand-Alone Application