CHAPTER 1 - Introducing Jenkins

**Introduction**

Continuous Integration, also know as CI, is a cornerstone of modern software development. In fact it is a real game changer—when Continuous Integration is introduced into an organization, it radically alters the way teams think about the whole development process. It has the potential to enable and trigger a series of incremental process improvements, going from a simple scheduled automated build right through to continuous delivery into production. 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. Every professional development team, no matter how small, should be practicing CI.

# Continuous Integration 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.

In essence, Continuous Integration is about reducing risk by providing faster feedback. First and foremost, it is designed to help identify and fix integration and regression issues faster, resulting in smoother, quicker delivery, and fewer bugs. By providing better visibility for both technical and non-technical team members on the state of the project, 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.

This idea of automated deployment is important. Indeed, if you take automating the deployment process to its logical conclusion, you could push every build that passes the necessary automated tests into production. The practice of automatically deploying every successful build directly into production is generally known as Continuous Deployment.

However, a pure Continuous Deployment approach is not always appropriate for everyone. For example, many users would not appreciate new versions falling into their laps several times a week, and prefer a more predictable (and transparent) release cycle. Commercial and marketing considerations might also play a role in when a new release should actually be deployed.

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.

So Continuous Integration techniques, and in particular Continuous Deployment and Continuous Delivery, are very much about providing value to the end user faster. How long does it take your team to get a small code change out to production? How much of this process involves problems that could have been fixed earlier, had you known about the code changes that Joe down the corridor was making? How much is taken up by labor-intensive manual testing by QA teams? How much involves manual deployment steps, the secrets of which are known only to a select few? CI is not a silver bullet by any means, but it can certainly help streamline many of these problems.

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.

In this book we will be looking at how to implement a robust and comprehensive Continuous Integration solution using Jenkins

# Introducing Continuous Integration 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. In the following paragraphs, I have tried to paint an approximate picture of each phase.

## Phase 1—No Build Server

Initially, the team has no central build server of any kind. Software is built manually on a developer’s machine, though it may use an Ant script or similar to do so. Source code may be stored in a central source code repository, but developers do not necessarily commit their changes on a regular basis. Some time before a release is scheduled, a developer manually integrates the changes, a process which is generally associated with pain and suffering.

## Phase 2—Nightly Builds

In this phase, the team has a build server, and automated builds are scheduled on a regular (typically nightly) basis. This build simply compiles the code, as there are no reliable or repeatable unit tests. Indeed, automated tests, if they are written, are not a mandatory part of the build process, and may well not run correctly at all. However developers now commit their changes regularly, at least at the end of every day. If a developer commits code changes that conflict with another developer’s work, the build server alerts the team via email the following morning. Nevertheless, the team still tends to use the build server for information purposes only—they feel little obligation to fix a broken build immediately, and builds may stay broken on the build server for some time.

## Phase 3—Nightly Builds and Basic Automated Tests

The team is now starting to take Continuous Integration and automated testing more seriously. The build server is configured to kick off a build whenever new code is committed to the version control system, and team members are able to easily see what changes in the source code triggered a particular build, and what issues these changes address. In addition, the build script compiles the application and runs a set of automated unit and/or integration tests. In addition to email, the build server also alerts team members of integration issues using more proactive channels such as Instant Messaging. Broken builds are now generally fixed quickly.

## Phase 4—Enter the Metrics

Automated code quality and code coverage metrics are now run to help evaluate the quality of the code base and (to some extent, at least) the relevance and effectiveness of the tests. The code quality build also automatically generates API documentation for the application. All this helps teams keep the quality of the code base high, alerting team members if good testing practices are slipping. The team has also set up a “build radiator,” a dashboard view of the project status that is displayed on a prominent screen visible to all team members.

## Phase 5—Getting More Serious About Testing

The benefits of Continuous Integration are closely related to solid testing practices. Now, practices like Test-Driven Development 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.

## Phase 6—Automated Acceptance Tests and More Automated Deployment

Acceptance-Test Driven Development is practiced, guiding development efforts and providing high-level reporting on the state of the project. 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.

## Phase 7—Continuous Deployment

Confidence in the automated unit, integration and acceptance tests is now such that teams can apply the automated deployment techniques developed in the previous phase to push out new changes directly into production.

The progression between levels here is of course somewhat approximate, and may not always match real-world situations. For example, you may well introduce automated web tests before integrating code quality and code coverage reporting. However, it should give a general idea of how implementing a Continuous Integration strategy in a real world organization generally works.