

A photograph of industrial pipes and valves at dusk or dawn. The pipes are illuminated by a warm, golden light, creating a dramatic effect against the dark, cloudy sky. The pipes are arranged in a complex, overlapping pattern, with some running horizontally and others curving upwards or downwards. The overall scene conveys a sense of industrial scale and complexity.

# Continuous Delivery for Java Apps

Build a CD Pipeline Step by Step Using  
Kubernetes, Docker, Vagrant, Jenkins,  
Spring, Maven and Artifactory

JORGE ACETOZI

# Continuous Delivery for Java Apps: Kubernetes and Jenkins in Practice

Build a CD Pipeline Step by Step Using Kubernetes, Docker, Vagrant, Jenkins, Spring, Maven and Artifactory

Jorge Acetozi

This book is for sale at <http://leanpub.com/continuous-delivery-for-java-apps>

This version was published on 2018-07-06



Leanpub

This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2017 - 2018 Jorge Acetozi

# Also By **Jorge Acetozi**

Graylog

# Contents

Why You Should Read This Book . . . . .	i
About the Author . . . . .	iii
<b>Introduction . . . . .</b>	<b>1</b>
Agile . . . . .	2
Scrum . . . . .	5
Scrum and Continuous Integration . . . . .	9
Deployed vs Released . . . . .	10
Scrum and Continuous Delivery . . . . .	11
Extreme Programming and Continuous Delivery . . . . .	14
Automated Tests . . . . .	15
Continuous Integration . . . . .	17
Feature Branch . . . . .	19
Continuous Delivery . . . . .	20
Continuous Delivery Pipeline . . . . .	21
Continuous Delivery vs Continuous Deployment . . . . .	26
Canary Release . . . . .	27
A/B Tests . . . . .	30

## CONTENTS

Feature Flags . . . . .	31
-------------------------	----

<b>Notepad App: Automated Tests, Maven and Flyway . . . . .</b>	<b>32</b>
---	-----------

Pre-Requisites . . . . .	33
--------------------------	----

<b>The Notepad Application . . . . .</b>	<b>34</b>
The Note Model . . . . .	35
The Note Controller . . . . .	37

<b>Unit Tests . . . . .</b>	<b>40</b>
NoteTest.java . . . . .	41

<b>Integration Tests . . . . .</b>	<b>43</b>
NoteServiceTest.java . . . . .	44
NoteControllerTest.java . . . . .	47

<b>Acceptance Tests . . . . .</b>	<b>49</b>
Page Object: NewNotePage.java . . . . .	50
CreateNoteTest.java . . . . .	52
AcceptanceTestsConfiguration.java . . . . .	54
Distributed Acceptance Tests with Selenium-Grid . . . . .	54

<b>Smoke Tests . . . . .</b>	<b>59</b>
------------------------------	-----------

<b>Performance Tests . . . . .</b>	<b>60</b>
Gatling . . . . .	61
HomeSimulation.scala . . . . .	64

<b>Apache Maven . . . . .</b>	<b>66</b>
Maven Snapshot vs Release . . . . .	67
The Default Lifecycle and its Phases . . . . .	68
Maven Repositories . . . . .	70
Repository Manager . . . . .	71

<b>Introduction to Docker . . . . .</b>	<b>73</b>
---	-----------

## CONTENTS

<b>Difference Between Container and Image . . . . .</b>	<b>75</b>
<b>Jenkins Overview . . . . .</b>	<b>77</b>
Node, Master, and Agent (or Slave) . . . . .	82
<b>ChatOps . . . . .</b>	<b>84</b>
<b>Why Kubernetes? . . . . .</b>	<b>86</b>
Namespaces . . . . .	88
Pods . . . . .	93
<b>Kubernetes Architecture . . . . .</b>	<b>98</b>
Kubernetes Master Components . . . . .	100
Etcd . . . . .	100
API Server . . . . .	101

## **Hands-on Project: Continuous Delivery Pipeline** **104**

New User Story Development Flow . . . . .	105
---	-----

# Why You Should Read This Book

Dear reader,

This book will guide you through the implementation of the **real-world** Continuous Delivery using top-notch technologies that are in high demand by the best companies around the world. Instead of finishing this book thinking “I know what Continuous Delivery is, but I have no idea how to implement it”, you will end up with your machine set up with a Kubernetes cluster running Jenkins Pipelines in a distributed and scalable fashion (each Pipeline run on a new Jenkins `slave` dynamically allocated as a Kubernetes pod) to test (unit, integration, acceptance, performance and smoke tests), build (with Maven), release (to Artifactory), distribute (to Docker Hub) and deploy (on Kubernetes) a Spring Boot application to testing, staging and production environments implementing the [Canary Release](https://martinfowler.com/bliki/CanaryRelease.html)<sup>1</sup> deployment pattern to mitigate risks.

This book is intended for Java Developers or SysAdmins interested in learning how to build a continuous delivery pipeline step by step using Kubernetes, Docker, Vagrant, Jenkins, Spring, Maven and Artifactory.

If you are a Java Developer, **it’s not required** that you have prior knowledge with Kubernetes, Docker, Vagrant, Jenkins, etc. If you are a SysAdmin, **it’s not required** that you know about Java, Spring, Maven, and so on. Although this is a hands-on book, all the theory needed to build the CD pipeline is provided step by step throughout the book.

I have worked as a Java Developer for many years and now I’m particularly interested in subjects such as these:

- Agile
- DevOps / Continuous Delivery
- NoSQL Databases (Cassandra, Redis, etc)
- Cloud Computing

---

<sup>1</sup><https://martinfowler.com/bliki/CanaryRelease.html>

- Containers
- Distributed Systems
- Linux / Infrastructure / Security

This is the book I wish I had found when I was learning how to implement Continuous Delivery in practice; **that's why I wrote it**. It will bring to you years of experience implementing Continuous Delivery in many different Java projects.

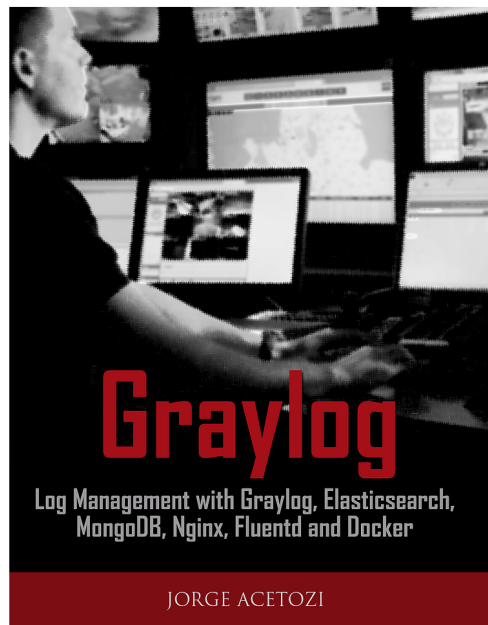
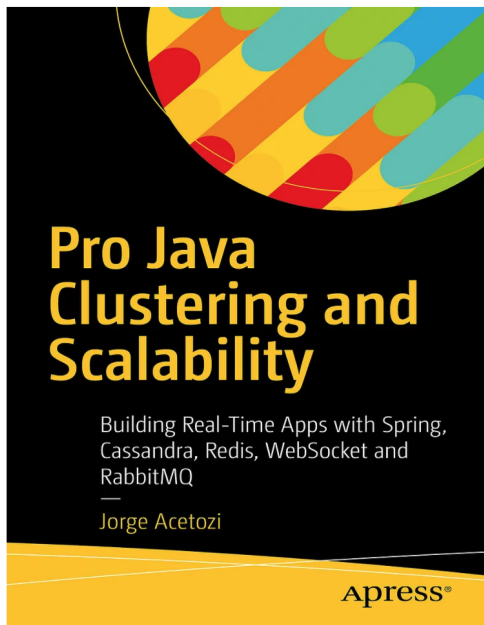
It's also worth to mention that this book is a **forever edition**, which means that no matter how fast the technologies used here evolve, this book will be always kept up to date and of course you will get notified and will pay nothing to take advantage of the updates.

I hope you buy this book and have a very pleasant reading. Thank you very much!



# About the Author

Jorge Acetozi is a software engineer who spends almost his whole day having fun with things such as AWS, Kubernetes, Docker, Terraform, Ansible, Cassandra, Redis, Elasticsearch, Graylog, New Relic, Sensu, Elastic Stack, Fluentd, RabbitMQ, Kafka, Java, Spring, and much more! He loves deploying applications in production while thousands of users are online, monitoring the infrastructure, and acting quickly when monitoring tools decide to challenge his heart's health!



Author's Books

You can reach him at:

- [Web site<sup>2</sup>](#)

---

<sup>2</sup><https://www.jorgeacetozi.com>

- [This Book's web site](https://www.continuous-delivery-java.com)<sup>3</sup>
- [GitHub](https://github.com/jorgeacetozi)<sup>4</sup>
- [LinkedIn](https://www.linkedin.com/in/jorgeacetozi)<sup>5</sup>
- [Facebook](https://www.facebook.com/jorgeacetozi)<sup>6</sup>
- [Twitter](https://twitter.com/jorgeacetozi)<sup>7</sup>
- [Medium](https://medium.com/jorgeacetozi)<sup>8</sup>

---

<sup>3</sup><https://www.continuous-delivery-java.com>

<sup>4</sup><https://github.com/jorgeacetozi>

<sup>5</sup><https://www.linkedin.com/in/jorgeacetozi>

<sup>6</sup><https://www.facebook.com/jorgeacetozi>

<sup>7</sup><https://twitter.com/jorgeacetozi>

<sup>8</sup><https://medium.com/jorgeacetozi>

# Introduction

In practice, implementing continuous delivery requires technical skills in different technologies, and that's why this book has more than 500 pages with a lot of technical stuff (and it could have more!). However, a smooth continuous delivery process requires more than that. Actually, everything starts at the planning stage; if your team makes some (often) mistakes here, the whole flow can be compromised. In this section, we are going to learn how to avoid making these mistakes.

Besides, we define crucial concepts such as Continuous Integration, Continuous Delivery, Continuous Deployment, Feature Branch, Canary Release, Test A/B and Feature Flags. This will ensure you won't get confused about these concepts anymore and that we are actually on the same page.

# Agile

Tell me, how many consecutive years did you fail to accomplish your New Year's promises and long-term plans for the next year? Long-term plans are just a way for human beings to feel less uncomfortable with the unknown. We kind of get the feeling we have control over stuff when we have a long-term plan.

The point is that all of the sudden, on January 15th, when you think you are in control of your life, you receive an email with a job offer paying twice more than your current job or you simply get fired.

What we don't immediately realize is that a job offer paying twice more sometimes is not good (the job and people there are boring, they don't care for development best practices, the environment does not encourage communication, and so on) and sometimes getting fired is not bad (your next job can be much better). This happens because human beings tend to react without thinking first. Very often we caught ourselves being thankful for something that initially we thought it would be bad.

In the software development context, the same happens when the client asks for a change in something that was already delivered or even is being developed. Sometimes, our first reaction is to become upset and think "why didn't they think about it before asking? These guys don't know what they want!". However, after a while we did realize that the change is actually for the best and it will increase a lot of value to the whole product; the client will be more satisfied and the end users will benefit. The second Agile principle states: **Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.**

Agile is not about being faster. It is about being able to adapt well to new situations. Imagine that you are in a building on fire. You can either be really fast and escape from it by throwing yourself out through the window or you can find the fire escape and escape from it in safety. Option one is faster, option two is agile.

Agile Methodologies for Software Development such as Scrum and Extreme Programming are based on the [Agile Manifesto](http://agilemanifesto.org/)<sup>9</sup>, which states the following:

---

<sup>9</sup><http://agilemanifesto.org/>

1. Individuals and interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Also, let's check the [Twelve Principles of Agile Software](#)<sup>10</sup>.

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

I want you to read carefully the first Agile Principle again: **Our highest priority**

---

<sup>10</sup><http://agilemanifesto.org/principles.html>

**is to satisfy the customer through early and continuous delivery of valuable software.**

Most of the companies nowadays use Scrum as a framework for managing software development. Have you ever thought if there is something in Scrum that may affect the implementation of Continuous Delivery?

# Scrum

Developing software in a non-iterative fashion is probably the reason why a lot of good projects with highly-skilled people have failed over the past years.

As I mentioned, the whole problem with this approach is that when you set up a long-term plan and strictly follow it, like the [Waterfall model](#)<sup>11</sup>, you lose the ability of making changes and adapting the product to what the client really needs, which most of the times happens to be figured out only during the project development phase rather than the in the very first talks to the client.

In order to avoid this issue, the current recommended way for approaching software projects is by iterations, that is, shorter periods of time (weeks) where the whole process happens (planning, development, tests, release, client feedback). This not only allows the client to start using the software earlier (adding value to his business) but also gives the opportunity for him to provide faster feedback, ask for changes earlier, change feature priorities, and so on.

A very well-known framework for software management that implements this idea is called **Scrum**. It has been successfully used along many years in companies all around the world.



Of course that there are many other factors that influence the success of a software project such as the code quality, the release process, the application architecture, and so on. Scrum does not define software engineering best practices, it is all about **managing** the software project. There is another Agile Methodology called **Extreme Programming (XP)** that approaches these subjects. Software development teams often use Scrum and XP together to take advantage of both worlds.

---

<sup>11</sup>[https://en.wikipedia.org/wiki/Waterfall\\_model](https://en.wikipedia.org/wiki/Waterfall_model)



This is not a Scrum book, so I'm just providing an overview and pointing out what part of Scrum is extremely decisive regarding Continuous Delivery. If you wish to have further information about Scrum, I would suggest you read the [Scrum Guide](http://www.scrumguides.org)<sup>12</sup>.

The Scrum artifacts are:

- **Product Backlog:** a list of items that represent the client “wish list”. **These items can be [user stories], bug fixes, non-functional requirements, and so on.**
- **Sprint Backlog:** a list of items chosen in the Sprint Planning to be developed in a particular sprint.
- **Product Increment:** the sum of all the product backlog items completed during a sprint, integrated with the work of all previous sprints.

A **Scrum Team** consists of:

- **Product Owner:** defines and prioritize the items that make up the Product Backlog, answer to business questions that the Development Team might have, etc.
- **Development Team:** make up the people that are actually technical, such as developers, system administrators, database administrators and so on.
- **Scrum Master:** remove possible impediments that might show up for the development team during the development phase, facilitates Scrum Events, etc.

In short, Scrum calls each iteration as a **Sprint**. A sprint should always have the same duration (it can vary from 1 to 4 weeks) and it's essentially composed by these events (in the presented order):

- **Spring Planning:** a meeting that takes place at the beginning of each sprint where the Scrum Team define which items from the Spring Backlog should be developed in the development phase. Basically, the Product Owner picks the most priority items and answer to business doubts that arise. Then, the

---

<sup>12</sup><http://www.scrumguides.org>

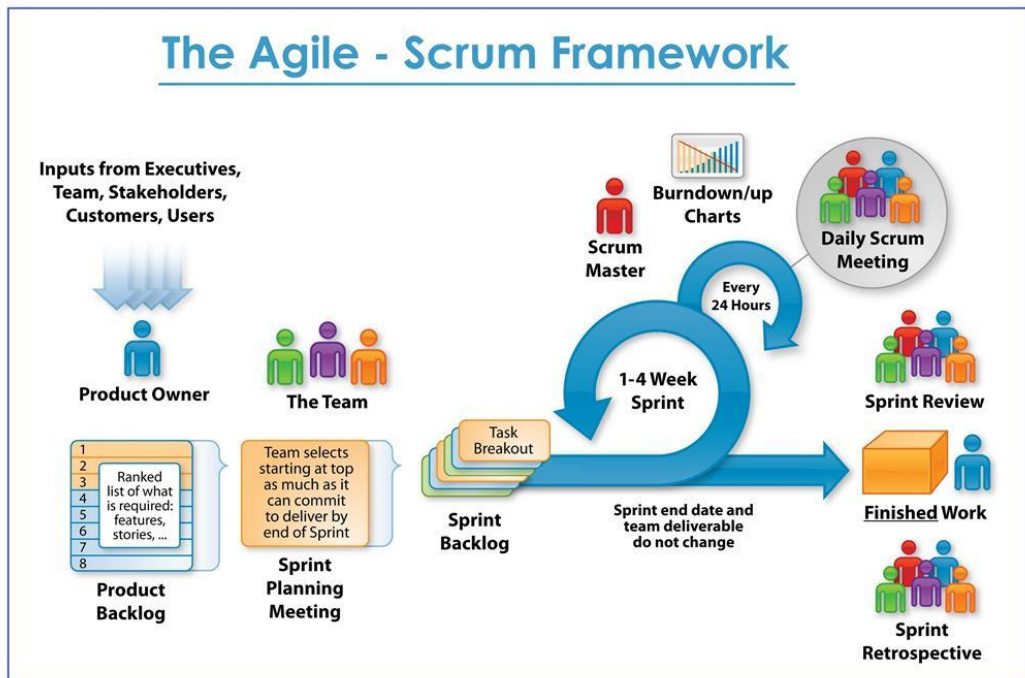


developers estimate the effort to develop these items, optionally using the [Planning Poker](https://en.wikipedia.org/wiki/Planning_poker)<sup>13</sup>, and based on the average points the team is used to deliver in every Sprint, the Product Owner decides whether he puts some items back to his Product Backlog or he includes more items to the Sprint Backlog. **After they all agree on which items will compose the Sprint Backlog, it's time for developers to break these items into tasks. Tasks are the technical pieces that must be done in order to deliver an item. A single item can raise many tasks. The important thing here is that each task should not last more than a day to be developed, including automated tests, of course.**

- **Daily Scrum:** a quick meeting that takes place every day so that team members can inform what they did yesterday, what they are doing today and if they have any impediments.
- **Sprint Review:** a meeting that involves the Scrum Team and any stakeholder to review the work that was completed and the planned work that was not completed along with a presentation on the completed work (for example, a demo). It is also a time where everyone collaborates on what to do next, so it provides a valuable input to the subsequent Sprint Planning.
- **Sprint Retrospective:** a meeting where the Scrum team discusses what went well in the Sprint, what could be improved and what they will commit to improve in the next Sprint.

---

<sup>13</sup>[https://en.wikipedia.org/wiki/Planning\\_poker](https://en.wikipedia.org/wiki/Planning_poker)



Scrum Framework

## Scrum and Continuous Integration

Let's understand why I highlighted the Sprint Planning. Anticipating a little bit the definition of Continuous Integration:

Continuous Integration (CI) is a development practice from Extreme Programming that requires developers to **integrate code into a mainline as often as possible, at least once a day**, and each check-in is then verified by an automated build that compiles the code and run the suite of automated tests against it, allowing teams to detect problems early.

If the development team estimates tasks in such a way they will take forever to be coded, how can they **integrate code into a mainline as often as possible, at least once a day**? That's the reason why I started writing this book approaching Agile, Scrum and XP. To implement continuous integration, the whole team must have a clear understanding that:

- Tasks should be small. That means they have to last no more than a day to be coded (including automated tests) and integrated to the mainline. This is one of the requirements for implementing continuous integration.
- Continuous integration is a requirement for implementing continuous delivery (although continuous delivery is not a requirement for implementing continuous integration; you will understand more about this in minutes, don't worry).

## Deployed vs Released

From now on, it's very important that you be aware of the difference between these terms:

- **Deployed:** A technical concern that applies in the domain of the team and means the product is introduced in a chosen environment (testing, staging, production, and so on).
- **Released:** A business term that defines functionality being available to an end user.

Note that **deployed doesn't necessarily mean released**. That's because you can deploy code to production that is not active for the end user. In other words, although the code is indeed in production, the feature behind it is not visible to the end user. This mechanism is called [Feature Flag](#) and we will discuss it later in this book.

## Scrum and Continuous Delivery

In the [Scrum and Continuous Integration](#) section, I mentioned that **continuous integration is a requirement for implementing continuous delivery**. Well, if continuous delivery requires continuous integration and continuous integration requires small tasks that are coded and integrated to the mainline in no longer than a day, we can conclude that continuous delivery also requires small tasks that are coded and integrated to the mainline in no longer than a day, right? It's like simple math: if  $a > b$  and  $b > c$  then  $a > c$ .

In short, if tasks are wrongly split (that is, in a way that it would take more than a day to develop it with automated tests and integrate it back to the mainline) in the sprint planning, then continuous integration will be compromised and, as continuous delivery depends on continuous integration, continuous delivery will also be compromised. **In fact, everything starts by correctly splitting the tasks.**

Now, read the definition of sprint review again: "Sprint Review is a meeting that involves the scrum team and any stakeholder to review the work that was completed and the planned work that was not completed along with a presentation on the completed work (for example, a demo)". I don't know if it's only me, but doesn't it give you the false impression that Scrum is something like...

1. Planning
2. Development (most of the sprint time)
3. **Review: deploy the software to a controlled environment, such as staging, where end users are not using it yet. A presentation takes place and if people are all happy with what they see, the software is eventually deployed to production and released to end users.**
4. Retrospective

Well, especially some years ago when continuous delivery wasn't a reality yet, this would sound really reasonable. Actually, many years ago I worked for a company that ran Scrum exactly like this. We used to call the day before the sprint review as the "release day", because it indeed used to take the whole day (and sometimes the whole night) to create a single .war package and deploy it to the staging environment.

People did know that their work would only be published after many days for a demo in the sprint review (and it would be only available for the product owner and stakeholders, not the end users) so they didn't really worry about integrating their codes as fast as possible at least once a day as the continuous integration definition states. The "release day" was the day when people actually were trying to merge their branches into master, and guess what used to happen every time? That's right: merging issues, or, like we say in the continuous integration context, the **merge hell**.

Sometimes people were at the company late at night trying to solve these merging issues, but very often the guy that actually wrote that code wasn't in the office anymore.

Because of these issues and others like automated tests failing, very often even the sprint review presentation used to fail, and of course the whole team used to feel disappointed, leading the sprint retrospective to be very negative every time. If deploying to staging were that panic, imagine deploying to production.

It was a time when we couldn't imagine deploying code to production every day, many times a day, or even on every single commit. I mean, we couldn't imagine it even if we had enough knowledge to actually implement it, which actually we didn't. Thus, deploying code to production only after the product owner's approval in the sprint review sounded quite reasonable to me (and to the whole team as well).

The point here is that, where in the Scrum Guide does it states that you must deploy to production and release features to your end users only after the sprint review? In fact, it simply states that the development team shows a demo of what is **done**. In other words, **the release is not related to the sprint review meeting**.

If you read the Scrum Guide, you will notice that there is a whole topic called [Definition of Done<sup>14</sup>](http://www.scrumguides.org/scrum-guide.html#artifact-transparency-done), which states: "When a Product Backlog item or an Increment is described as Done, everyone must understand what Done means. Although this varies significantly per Scrum Team, members must have a shared understanding of what it means for work to be complete, to ensure transparency."

If the definition of done for your scrum team includes "deployed to production", then the idea of the "release day" I mentioned before does not make sense anymore and we are absolutely fine even deploying to production many times a day.

---

<sup>14</sup><http://www.scrumguides.org/scrum-guide.html#artifact-transparency-done>

If there was an automated, safe and fast way to deploy code to production, the decision to release features to end users would be much more in the business hands than the IT. That would be great, wouldn't it? Good news: fortunately, this way is called **Continuous Delivery**, and it is exactly what you are going to implement step by step using modern technologies in this book.

# Extreme Programming and Continuous Delivery

Extreme Programming (XP) is another important and well-known Agile Methodology that is often used together with Scrum. XP is an agile software development framework that provides important guidelines on how to plan, manage, design, code and test software projects. In this book, I'm only approaching a subset of these practices that are **strictly required** in order to implement continuous delivery, which happens to be **Automated Tests and Continuous Integration**.

Obviously, there are many other software engineering practices encouraged by XP that directly improve the software quality and hence the release quality, such as [Pair Programming](http://www.extremeprogramming.org/rules/pair.html)<sup>15</sup>, [Test Driven Development](http://www.extremeprogramming.org/rules/testfirst.html)<sup>16</sup>, [Refactoring](http://www.extremeprogramming.org/rules/refactor.html)<sup>17</sup>, [Simple Design](http://www.extremeprogramming.org/rules/simple.html)<sup>18</sup>, and so on. I would highly recommend that you practice each of them within your development team every single day. Although they don't really prevent you from implementing continuous delivery, I'm pretty sure that you want to deliver the highest quality software possible to your customers, and these techniques and others encouraged by XP will certainly help you achieve that.



If you wish to have further information on Extreme Programming, I would suggest you to read [extremeprogramming.org](http://www.extremeprogramming.org)<sup>19</sup>.

---

<sup>15</sup><http://www.extremeprogramming.org/rules/pair.html>

<sup>16</sup><http://www.extremeprogramming.org/rules/testfirst.html>

<sup>17</sup><http://www.extremeprogramming.org/rules/refactor.html>

<sup>18</sup><http://www.extremeprogramming.org/rules/simple.html>

<sup>19</sup><http://www.extremeprogramming.org>



## Automated Tests

Humans are good at creative tasks but horrible at repetitive tasks. Machines are horrible at creative tasks but excellent at repetitive tasks. Why not take advantage of both humans and machines? In other words, let humans and machines do what they do best!

Testing is a repetitive, difficult, and boring task. Humans are not able to test thousands of use cases without making mistakes. Machines are. Also, machines can accomplish a testing task in seconds or minutes, while humans can spend entire days doing it. This also means that replacing humans with machines on these kinds of tasks is cheaper.

Having a huge and high quality suite of automated tests to your application means that you can **safely** make changes, develop new features, refactor parts of the system or fix a bug. After your changes, you can simply run the automated tests and they will give almost immediate feedback whether your changes affected other parts of the system or everything is still working good. Just bear in mind that every tiny change you introduce to the code should also be accompanied by more automated tests.

Some time ago, a friend that worked for a big company migrating and evolving their IT projects to implement continuous delivery told me that he was working hard creating a continuous delivery pipeline for an application and automating every single infrastructure step, but the application itself didn't have any automated tests and the developers used to take weeks to integrate their codes into the mainline. He asked me what was my opinion about that and what I simply said is that they were trying to implement continuous delivery without continuous integration, and that is simply not possible (unless you want to release a lot of garbage in production to your end users in an automated fashion, which I bet you don't). The point in my answer is: No automated tests? No continuous integration! No continuous integration? No continuous delivery!

I complemented saying that the development team had to start learning about automated tests and coding it for every new feature, refactoring or bug fixes, otherwise they would be delivering code to production that they don't really know whether it works or not. That's not the idea of continuous delivery. Instead, continuous delivery aims to deliver changes (such as new features, configuration changes, bug fixes,

and so on) into production **safely** and quickly in a **sustainable way**. Having no automated tests is neither safe nor sustainable.

## Continuous Integration

Continuous Integration (CI) is a development practice from Extreme Programming that requires developers to **integrate code into a mainline as often as possible, at least once a day**, and each check-in is then verified by an automated build that compiles the code and run the suite of automated tests against it, allowing teams to detect problems early.

In the Git context, the mainline often turns out to be the `master` branch and each check-in is a new `commit` that emerges into the remote repository.

Implementing continuous integration provides several benefits:

- Merge hell, that is, endless merge conflicts while merging code into the mainline, is avoided because the code should be integrated to the mainline as often as possible (at least once a day).
- Allows to detect errors quickly as automated tests run on every single commit that checks-in the remote repository. Hence, less bugs get shipped to production.
- When an error is detected, it's much easier to fix because the amount of code involved in each commit is much smaller.
- Promotes human communication earlier when merge conflicts occur. People can sit together and understand how their codes are actually influencing the other's code.
- Building the release is easy as all integration issues have been solved early.
- Less bugs get shipped to production as regressions are captured early by the automated tests.

However, as mentioned earlier in this book, everything comes at a price. To implement continuous integration, the development team must understand that:

- Tasks should be split as small as possible because the code should be developed (including automated tests) and integrated to the mainline as often as possible, at least once a day.
- Automated tests should be present for each new feature, improvement or bug fix. They will ensure that a change (even a small one) is not breaking any other part of the system.

The second part of the definition states that “each check-in is then verified by an automated build that compiles the code and run the suite of automated tests”. In order to run the automated tests on every single commit that emerges in the remote repository mainline, the team will need to set up a **Continuous Integration server**. The most essential purpose of the CI server is to run automated tests on every new commit that emerges in the remote repository mainline, but it is actually capable of doing much more than that, such as issuing custom notifications when a test or build fail, triggering releases generation, triggering the deployment to a specific environment, and so on. In this book, we will use [Jenkins](https://jenkins.io)<sup>20</sup>, a leading open source CI server written in Java that provides hundreds of plugins to support building, deploying and automating any project.

---

<sup>20</sup><https://jenkins.io>

# Feature Branch

**Feature Branch**<sup>21</sup> is a development practice that requires every task to be developed in its own branch apart from the mainline. When the code is completed it is then integrated back into the mainline. This is useful because it ensures that the code being developed doesn't interfere the mainline code.



Before you move on I would like to ask you something: based on what you have learned about Continuous Integration, is using Feature Branch a good or bad idea?

Well, depending on how Feature Branch is being used, it might be a good idea or not. If a developer creates a branch and works there for a week, then it's very bad. Why? Because he would be breaking one of the Continuous Integration premise: integrate to the mainline as often as possible, at least once a day.

On the other hand, if the branch is created and merged in the same day, then it's fine.

Note that the problem is not actually the Feature Branch itself, but how much time it takes to merge the branch into the mainline. Particularly, I have been successfully using Feature Branch for many years.

---

<sup>21</sup><https://martinfowler.com/bliki/FeatureBranch.html>

# Continuous Delivery

I really like Jez Humble's definition for continuous delivery:

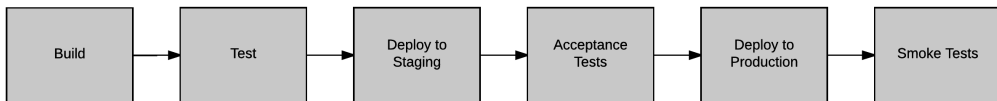
Continuous Delivery is the ability to get changes of all types; including new features, configuration changes, bug fixes and experiments into production, or into the hands of users, safely and quickly in a sustainable way.

In other words, when your team really implements continuous delivery what it actually means that the mainline is always in a deployable state, and one could decide to deploy it to production at any time at the touch of a button. This in turns is only possible because when this button is touched, an **automated pipeline** is triggered. The key element for achieving continuous delivery is automation!

Well, if you can deploy to production at any time, the question is: when is it the right time? Of course the answer may vary depending on your business requirements, but the truth is that if you want to get the benefits of continuous delivery, you should deploy to production as early as possible to make sure that you release small batches that are easy to troubleshoot in case of a problem.

# Continuous Delivery Pipeline

The continuous delivery pipeline is a set of automated steps that your code changes goes through until it is finally deployed to production. Pipeline steps can vary from project to project but they all include building and testing the application as part of the CI process.

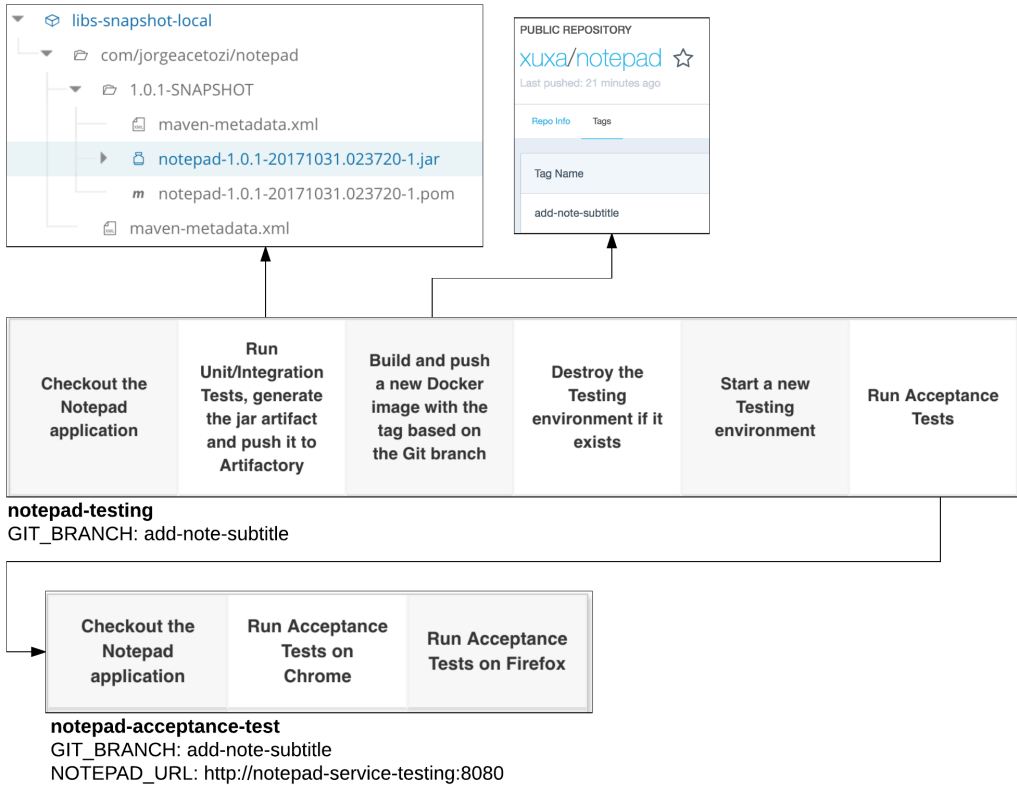


Pipeline Example

In this book we are going to implement the continuous delivery pipeline for the Notepad application using [Jenkins Pipelines](https://jenkins.io/doc/book/pipeline/)<sup>22</sup>. We'll let the details for later, but just take a look how nice the pipeline responsible for setting up the **testing** environment is:

---

<sup>22</sup><https://jenkins.io/doc/book/pipeline/>



### Testing Pipeline



In this book, the testing environment is immutable, which means that it is dynamically provisioned on top of Kubernetes for each new test execution.

Below is the code that makes all this magic possible. Don't worry if you don't get it now! By the end of this book you will be reading it as if it was the newspaper!



```
podTemplate(label: 'testing',
  containers: [
    containerTemplate(
      name: 'kubect1',
      image: 'jorgeacetozi/kubect1:1.7.0',
      ttyEnabled: true,
      command: 'cat'
    ),
    containerTemplate(
      name: 'mysql',
      image: 'mysql:5.7',
      envVars: [
        envVar(key: 'MYSQL_DATABASE', value: 'notepad'),
        envVar(key: 'MYSQL_ROOT_PASSWORD', value: 'root')
      ]
    ),
    containerTemplate(
      name: 'maven',
      image: 'maven:3.3.9-jdk-8-alpine',
      ttyEnabled: true,
      command: 'cat'
    ),
    containerTemplate(
      name: 'docker',
      image: 'docker',
      ttyEnabled: true,
      command: 'cat'
    )
  ],
  volumes: [
    hostPathVolume(
      hostPath: '/var/run/docker.sock',
      mountPath: '/var/run/docker.sock'
    ),
    secretVolume(
      mountPath: '/etc/maven/',
      secretName: 'maven-settings-secret'
    )
  ],
  envVars: [
    secretEnvVar(
      key: 'DOCKERHUB_USERNAME',
      secretName: 'dockerhub-username-secret',
      secretKey: 'USERNAME'
```

```

    ),
    secretEnvVar(
        key: 'DOCKERHUB_PASSWORD',
        secretName: 'dockerhub-password-secret',
        secretKey: 'PASSWORD'
    )
]
)
{
    node ('testing') {
        def image_name = "notepad"

        checkout scm

        dir('app') {
            stage('Checkout the Notepad application') {
                git url: 'https://github.com/jorgeacetozi/notepad.git', branch: "${GIT_BRANCH}"
            }

            stage('Run Unit/Integration Tests, generate the jar artifact and push it to Artifactory') {
                container('maven') {
                    sh 'mvn -B -s /etc/maven/settings.xml clean deploy'
                }
            }

            stage('Build and push a new Docker image with the tag based on the Git branch') {
                container('docker') {
                    sh """
                        docker login -u ${DOCKERHUB_USERNAME} -p ${DOCKERHUB_PASSWORD}
                        docker build -t ${DOCKERHUB_USERNAME}/${image_name}:${GIT_BRANCH} .
                        docker push ${DOCKERHUB_USERNAME}/${image_name}:${GIT_BRANCH}
                    """
                }
            }
        }

        stage('Destroy the Testing environment if it exists') {
            container('kubectl') {
                sh 'kubectl delete all -l env=testing'
            }
        }

        stage('Start a new Testing environment') {

```

```

    container('kubect1') {
        sh """
            sed -i "s/NOTEPAD_CONTAINER_IMAGE/${DOCKERHUB_USERNAME}\\/${image_name}:${GIT_BRANCH}\\
RANCH}/" ./notepad/k8s/testing/notepad-testing-deployment.yaml

            kubect1 apply -f notepad/k8s/testing/ -l app=mysql
            sleep 20

            kubect1 apply -f notepad/k8s/testing/ -l app=notepad
            kubect1 rollout status deployment notepad-deployment-testing

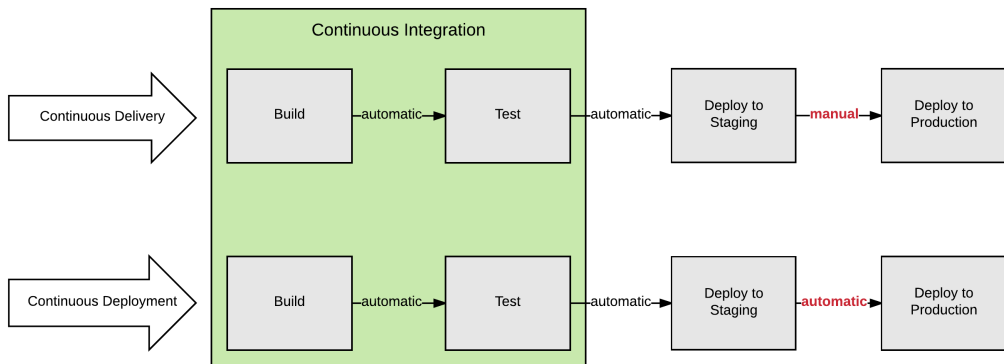
            kubect1 get service notepad-service-testing
            kubect1 get endpoints notepad-service-testing
        """
    }
}

stage ('Run Acceptance Tests') {
    build job: 'notepad-acceptance-test', parameters: [
        [$class: 'StringParameterValue', name: 'GIT_BRANCH', value: "${GIT_BRANCH}"],
        [$class: 'StringParameterValue', name: 'NOTEPAD_URL', value: 'http://notepad-serv\
ice-testing:8080']
    ]
}
}

```

# Continuous Delivery vs Continuous Deployment

Continuous deployment is very similar to continuous delivery, but it takes even one step further regarding automation. In continuous delivery, every change pushed to the main repository is ready to be deployed to production, but starting this process still requires human interaction. In continuous deployment, the deployment to production is automatically triggered for every change that passes the test suite.



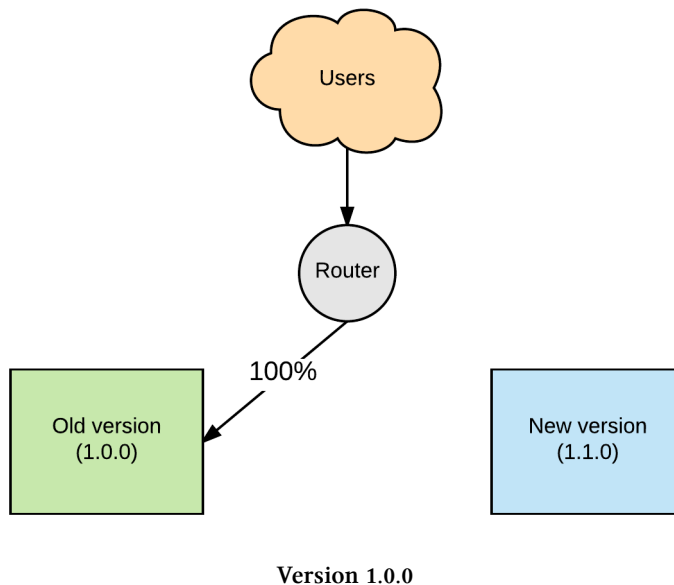
Continuous Delivery vs Continuous Deployment

As in continuous deployment every change that passes the test suite is deployed to production, it heavily relies on [feature flags](#) to release features to end users.

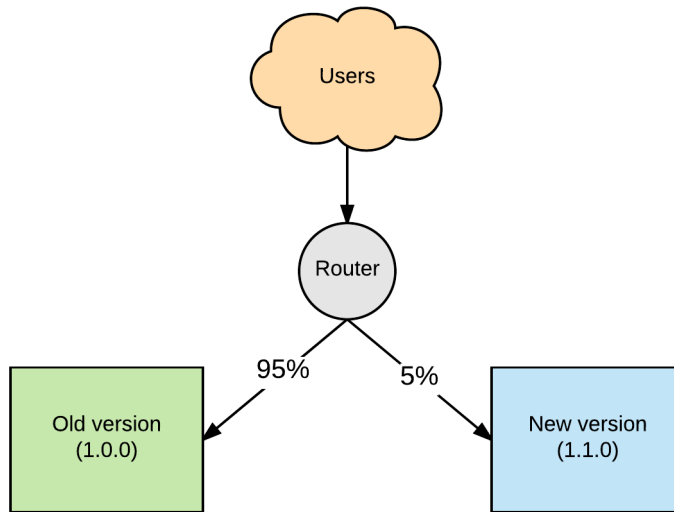
# Canary Release

Canary release is a technique to reduce the risk of introducing a new software version in production by gradually rolling it out through the servers of your infrastructure.

Suppose you are currently running your application in version 1.0.0 and you want to deploy a new version, say 1.1.0. Then, you set up version 1.1.0 to a subset of the infrastructure (say, a single machine) and run smoke tests on it to ensure that version 1.1.0 main functionalities still work.

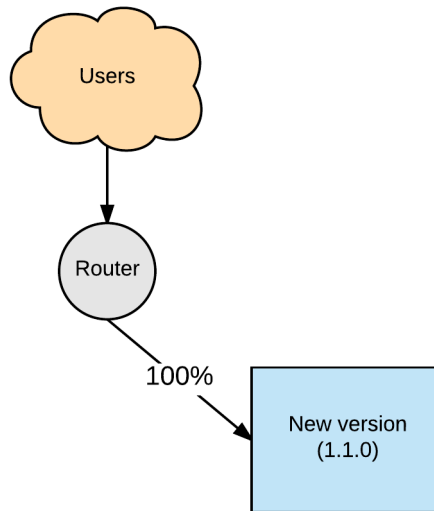


Now, you gradually start routing traffic to version 1.1.0 (say, 95% to the old version and 5% to the new version). Note that the router is basically a load balancer.



**Canary with version 1.1.0 starts receiving traffic**

As soon as you get more confident, you can gradually increase the traffic for version 1.1.0 until it reaches 100%, then you can simply get rid of version 1.0.0.



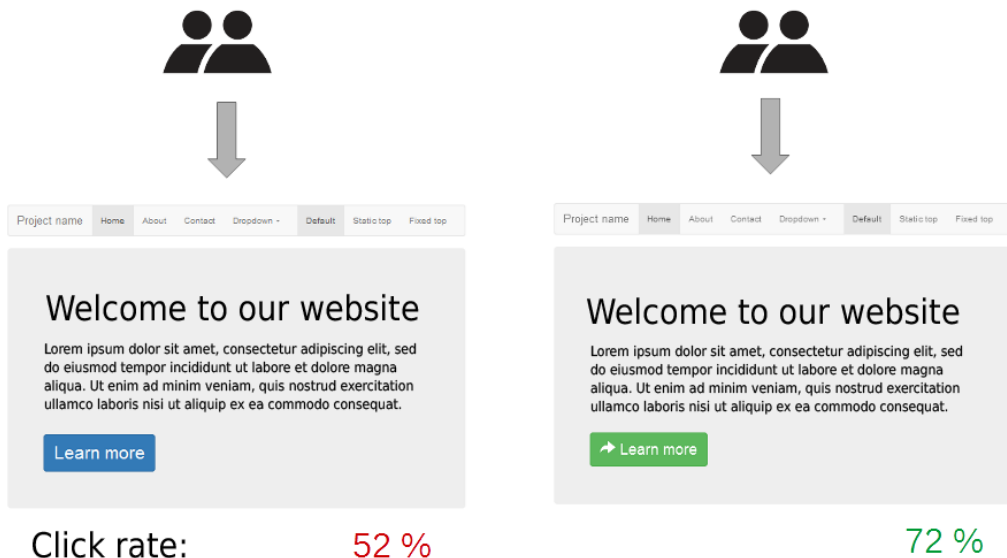
**Version 1.1.0 fully deployed**

This whole process is meant to last some minutes or at most hours. The intention here is just to be sure that the new version is working as expected and then rolling it out to the whole fleet of servers.

Bear in mind that it only makes sense to implement canary release if you've got a comprehensive business and IT monitoring for your application, because that's how you ensure that everything still working fine as you gradually increase traffic to the new version.

# A/B Tests

A/B testing is a method for comparing two versions of a feature (usually in the User Interface) against each other to determine which one performs better.



A/B test example

**Can you use canary release to perform A/B tests? Yes! Should you use canary release specifically to perform A/B tests? No!**

A/B tests might last for days to actually have statistical significance. Canary release is supposed to last for minutes or at most hours running different software versions. The intention of using canary release is just to be sure that the new version being deployed is working as intended.

If you want to perform A/B tests, use **Feature Flags**.



# Feature Flags

Feature flag is a technique that allows activate or deactivate features in a live system without having to change any code. Functionality can be deployed off, then turned on via the feature flag. One could activate a feature by using different strategies, such as a specific group of users, specific percentage of users, client IP, and so on.

That's quite handy for implementing A/B tests, because the feature to be tested could be deployed off and then turned on for metrics collection. After some days, there would be a conclusion whether the new feature performs better then the older one or not.

## Can canary release and feature flags be used together?

Absolutely, but they have different purposes! You should use canary release regardless of using feature flags or not. That's because even if the feature flag in the new release version is turned off, you have no idea if it actually didn't break any other part of the system (one that might not be covered with automated tests, for example). That is, even with the flag off, the new release version could have issues and you would only figure that out when your end users start using it.



A handy framework for implementing feature flags in Java is [Togglz](https://www.togglz.org)<sup>23</sup>.

---

<sup>23</sup><https://www.togglz.org>

# Notepad App: Automated Tests, Maven and Flyway

In this section, we are going to have an overview of the Notepad application code. The goal here is not to dive into implementation details, but to make sure that you get the general idea of Spring Profiles, know how to trigger the different types of automated tests, generate a snapshot version and generate a release version. This is important because when we get to the hands-on project later on this book you will not be lost asking yourself the meaning of some commands such as `mvn -B -s /etc/maven/settings.xml clean deploy`.

# Pre-Requisites

Before moving on, please make sure you have [Git](#)<sup>24</sup>, at least [JDK 8](#)<sup>25</sup>, [Maven](#)<sup>26</sup> and [Docker](#)<sup>27</sup> installed on your machine.

Now, let's clone the GitHub repositories used along this chapter into your machine. Create a new directory named `ebook-project`, `cd` into it and clone the Notepad repositories:

```
$ mkdir ebook-project && cd ebook-project
$ git clone https://github.com/jorgeacetozi/notepad.git
$ git clone https://github.com/jorgeacetozi/notepad-performance-test.git

$ ls -lh
drwxr-xr-x 9 jorgeacetozi staff 306B 1 Nov 20:08 notepad
drwxr-xr-x 7 jorgeacetozi staff 238B 1 Nov 20:08 notepad-performance-test
```

---

<sup>24</sup><https://www.atlassian.com/git/tutorials/install-git>

<sup>25</sup><http://www.oracle.com/technetwork/java/javase/downloads/index.html>

<sup>26</sup><https://maven.apache.org/>

<sup>27</sup><https://www.docker.com/>

# The Notepad Application

The Notepad application will be used all along this book as the sample application. Later on this book we are going to implement a complete continuous delivery pipeline for this application, so it's important for you to get familiar with it.

Basically, the Notepad application is a simple Spring Boot based app that allow users to save notes. Each note is represented by a `title` and a `content`. The notes are stored in a MySQL database instance. That's how the application looks like:

Jorge Acetozi - Notepad App New Note

Title	Content	Word Count
What is Cassandra?	Cassandra	14

New Note

Title

Kubernetes

Content

Best container orchestration tool ever!

Close

Create

## Create Note

Jorge Acetozi - Notepad App		New Note
Title	Content	Word Count
What is Cassandra?	Cassandra is a NoSQL database that belongs to the Column Family NoSQL database category.	14
Kubernetes	Best container orchestration tool ever!	5

## List Notes

# The Note Model

Let's take a look at the Note model:

```
@Entity
public class Note {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Integer id;
    @NotEmpty
    private String title;
    @NotEmpty
    private String content;

    // Makes Hibernate happy
    private Note () {}

    public Note (String title, String content) {
        this.title = title;
        this.content = content;
    }

    public Integer getId() {
        return id;
    }

    public String getTitle() {
        return title;
    }

    public String getContent() {
        return content;
    }

    public Integer getWordCount() {
        return this.content.split(" ").length;
    }

    @Override
    public String toString() {
        return "Note [id=" + id + ", title=" + title + ", content=" + content + ", wordCount=\n" + this.getWordCount() + "]\n";
    }
}
```

```
}
```

Here, we use the [Hibernate Validator](http://hibernate.org/validator/)<sup>28</sup> `@NotEmpty` annotation to indicate that the attributes `title` and `content` provided by the user cannot be empty. Also, we just provide getter methods since the `title` and `content` attributes are set through the public constructor. The `getWordCount()` method implements a simple logic to return the amount of words in the `content` attribute, for example, if the content is “Kubernetes is amazing”, this method should return 2. Well, it’s wrong, isn’t it? Actually, the method should return 3, but that was just a reminder that you should never trust a code without looking at the automated tests first (which, by the way, we are going to have a look at in a minute).

---

<sup>28</sup><http://hibernate.org/validator/>

## The Note Controller

When the user fills the title, the content and clicks the button **Create**, what happens under the hoods is that the title and the content provided are converted into a JSON object and sent to the notes API `/notes` using the HTTP `POST` method. If the API returns success, the note is appended to the grid and a success message is shown, otherwise a error message is shown.

Your note was successfully saved!

Title	Content	Word Count
What is Cassandra?	Cassandra is a NoSQL database that belongs to the Column Family NoSQL database category.	14
Kubernetes	Kubernetes is amazing	3

### Success Message

Title and Content cannot be empty

Title	Content	Word Count
What is Cassandra?	Cassandra	14
Kubernetes	Best contain	5

New Note

Title

Content

Close

Create

### Error Message

Below is the Javascript `createNewNote` function that is executed when the user clicks the **Create** button.

```
$(document).ready(function() {  
  var newNoteModal = $("#newNoteModal");  
  var btnCreateNewNote = $("#btnCreateNewNote");  
  var txtNewNoteTitle = $("#newNoteTitle");  
  var txtNewNoteContent = $("#newNoteContent");  
  
  function createNewNote() {  
    var newNote = {  
      'title' : txtNewNoteTitle.val(),  
      'content' : txtNewNoteContent.val()  
    };  
  
    $.ajax({  
      type : "POST",  
      url : "/notes",  
      data : JSON.stringify(newNote),  
      contentType : "application/json",  
      success : function(note) {  
        //add note to the grid  
  
        noty({  
          text: "Your note was successfully saved!",  
          type: 'success'  
        });  
      },  
      error(error) {  
        noty({  
          text: "Title and Content cannot be empty",  
          type: 'error'  
        });  
      }  
    });  
  }  
  btnCreateNewNote.on("click", createNewNote);  
})
```

When the backend API receives the HTTP POST request, the `create` method in the `NoteController` class is invoked. Basically, it first validates that the `title` and the `content` are not empty, then it logs the request data using the `DEBUG` level and calls the `NoteService` component to actually persist the note to the database.



```
@Controller
public class NoteController {
    @Autowired
    private NoteService noteService;

    private final Logger logger = LoggerFactory.getLogger(this.getClass());

    @PostMapping("/notes")
    @ResponseBody
    @ResponseStatus(code = HttpStatus.CREATED)
    public Note create(@RequestBody @Valid Note note) {
        logger.debug("Create note request: {}", note);
        return noteService.create(note);
    }
}
```



Just out of curiosity, the `NoteService` is just a Spring `@Service` that injects the `NoteRepository` repository, which is a simple [Spring Data JPA Repository](https://docs.spring.io/spring-data/jpa/docs/current/reference/html/)<sup>29</sup>.

---

<sup>29</sup><https://docs.spring.io/spring-data/jpa/docs/current/reference/html/>

# Unit Tests

Unit tests provide very fast feedback, that is, they run in seconds or even milliseconds. Basically, they test only an isolated unit of code such as a method in a class and are great for ensuring the behavior of a code isolated from its dependencies.

However, there are cases where a unit test would not be appropriate. For example, an issue with a SQL query would not be caught from a unit test since it does not actually hit the database.

In the Notepad application, the unit tests are written using [JUnit](http://junit.org)<sup>30</sup> and [Hamcrest](http://hamcrest.org)<sup>31</sup>.

---

<sup>30</sup><http://junit.org>

<sup>31</sup><http://hamcrest.org>

## NoteTest.java



There is no need to copy and paste the whole class here because my goal is just to illustrate how a unit tests look like for those who are seeing it for the first time. Please check the [NoteTest.java<sup>32</sup>](#) class to see the whole test suite.

When we were learning about the [Note Model](#), I stated that:

- The `title` should not be empty.
- The content should not be empty.
- The `getWordCount()` method should return the word count for the content attribute.

Let's cover these statements with unit tests.

```
@Test
public void shouldRaiseViolationWhenTitleIsEmpty() {
    Note note = new Note("", "Unit tests provide fast feedback");
    Set<ConstraintViolation<Note>> constraintViolations = validator.validate(note);
    assertThat(constraintViolations.size(), is(1));
}
```

In the `shouldRaiseViolationWhenTitleIsEmpty` test, first we create a `Note` with an empty title and the “Unit tests provide fast feedback” content, then we call the `validator.validate(note)` to check if there are any constraint violations for this note. Well, if the `Note` model is implemented correctly, then there should be a violation and the `constraintViolations` set size should be equals to 1.

---

<sup>32</sup><https://github.com/jorgeacetozi/notepad/blob/master/src/test/java/com/jorgeacetozi/notepad/unitTests/note/domain/model/NoteTest.java>

```
@Test
public void shouldRaiseViolationWhenContentIsEmpty() {
    Note note = new Note("Unit Tests", "");
    Set<ConstraintViolation<Note>> constraintViolations = validator.validate(note);
    assertThat(constraintViolations.size(), is(1));
}
```

The `shouldRaiseViolationWhenContentIsEmpty` test has pretty much the same logic, but here we want to ensure that the content should not be empty.

```
@Test
public void shouldCountWordsFromNoteContent() {
    Note note = new Note(
        "Unit Tests",
        "Unit tests provide fast feedback, but they test only an isolated unit of code"
    );
    assertThat(note.getWordCount(), is(14));
}
```

The `shouldCountWordsFromNoteContent` test creates a note with a content containing 14 words and calls the `getWordCount()` method in the note object. If it's implemented correctly, the return should be equals to 14.

# Integration Tests

Integration tests are more comprehensive than unit tests because they actually test an integration between the application and its external dependencies, such as a database or a webservice. Thus, they could be used for example to ensure that a SQL query is fetching the correct data, an external API is returning the correct JSON, and so on. However, integration tests also have their downsides, such as:

- Take more time to run than unit tests.
- More difficult to set up than unit tests because they expect the application's external dependencies to be up and running before they are executed.



The more similar to the production environment the tests run the better. So, in the testing pipeline we are going to implement later on this book, for our relational database integration tests, instead of setting up an in-memory database such as [H2<sup>33</sup>](http://www.h2database.com/html/main.html), we will actually set up a MySQL instance on top of a Docker container.

---

<sup>33</sup><http://www.h2database.com/html/main.html>

## NoteServiceTest.java

```
@RunWith(SpringRunner.class)
@SpringBootTest
@ActiveProfiles("test")
public class NoteServiceTest {
    @Autowired
    private NoteService noteService;
    private Note note;

    @Before
    public void setUp() {
        note = new Note("Kubernetes", "Best container orchestration tool ever");
    }

    @After
    public void destroy() {
        noteService.delete(note);
    }

    @Test
    public void shouldCreateNoteWithTitleAndContent() {
        Note createdNote = noteService.create(note);
        assertThat(createdNote.getId(), notNullValue());
        assertThat(createdNote.getWordCount(), is(5));
    }
}
```

Note the `@ActiveProfiles("test")` annotation. It instructs Spring to set up the datasource to MySQL based on the `test` [Spring Profile](https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-profiles.html)<sup>34</sup> configuration, which is described in the `application.yml` file.

---

<sup>34</sup><https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-profiles.html>

```
spring:
  profiles: test
  datasource:
    url: jdbc:mysql://localhost:3306/notepad
    username: root
    password: root
    testWhileIdle: true
    validationQuery: SELECT 1
  jpa:
    show-sql: true
    hibernate:
      ddl-auto: validate
      naming-strategy: org.hibernate.cfg.ImprovedNamingStrategy
    properties:
      hibernate:
        dialect: org.hibernate.dialect.MySQL5Dialect

flyway.enabled: true
endpoints.sensitive: false
```



Spring Profiles provide a way to segregate parts of your application configuration (the `application.yml` file, in our case) and make it only available to certain environments. Look at the complete [application.yml](#) and note that there is a profile set for each environment and the default profile is `dev`, which is defined by the property `spring.profiles.active: dev`. This is a handy feature because it allows us to define which environment configuration to use when starting the application through the command-line just by providing a simple system property. For example, to run the Notepad application using the staging configuration (the staging Spring Profile), just issue the `$ java -jar -Dspring.profiles.active=staging notepad.jar` command. To run the Notepad application using the production configurations (the production Spring Profile), just issue the `$ java -jar -Dspring.profiles.active=production notepad.jar` command, and so on.

Basically, in the `shouldCreateNoteWithTitleAndContent` integration test, it creates a note and persists it to the database. After that, it ensures that the note has an `id` and that the word count is 5. This test uses `setUp()` method to create the note object and the `destroy()` method to delete the note from the database after the test is executed.

Note that the test will create the note in a MySQL database running on `localhost`. Therefore, if you want to run this test in your local machine, you will have to set up a MySQL instance in your machine with username and password `root` and create a database called `notepad` in it. Don't worry, you will do that by instantiating a Docker container in minutes.

If you are reading it carefully, you might be asking yourself “where the tables for the `notepad` database are coming from?”. Actually, [Flyway](#) is taking care of it. Just relax and keep following me. For now, just assume that “someone” automatically created the database tables.



## NoteControllerTest.java

Before, the `NoteServiceTest.java` integration test ensured the integration between the application and the MySQL database instance is working correctly. Now, we have to ensure that the API is actually returning the JSON in the format that the frontend is expecting. This is exactly what the `NoteControllerTest.java` does.

```
@RunWith(SpringRunner.class)
@SpringBootTest
@ActiveProfiles("test")
@WebAppConfiguration
public class NoteControllerTest {

    @Autowired
    private WebApplicationContext wac;
    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build();
    }

    @Test
    public void shouldCreateNoteWithTitleAndContentAndReturnHttp201Created() throws Exception {
        Note note = new Note(
            "Integration Tests",
            "Test the external integrations such as databases or web services."
        );

        this.mockMvc.perform(post("/notes")
            .contentType(MediaType.APPLICATION_JSON)
            .content(new ObjectMapper().writeValueAsString(note))
        )
        .andDo(print())
        .andExpect(status().isCreated())
        .andExpect(jsonPath("$.id", is(notNullValue())))
        .andExpect(jsonPath("$.title", is(note.getTitle())))
        .andExpect(jsonPath("$.content", is(note.getContent())))
        .andExpect(jsonPath("$.wordCount", is(note.getWordCount())));
    }

    @Test
```

```

    public void shouldNotCreateNoteWhenTitleIsEmptyAndReturnHttp400BadRequest() throws Exception {
        Note note = new Note("", "Test the external integrations such as databases or web services.");

        this.mockMvc.perform(post("/notes")
            .contentType(MediaType.APPLICATION_JSON)
            .content(new ObjectMapper().writeValueAsString(note))
        )
            .andDo(print())
            .andExpect(status().isBadRequest());
    }

    @Test
    public void shouldNotCreateNoteWhenContentIsEmptyAndReturnHttp400BadRequest() throws Exception {
        Note note = new Note("Integration Tests", "");

        this.mockMvc.perform(post("/notes")
            .contentType(MediaType.APPLICATION_JSON)
            .content(new ObjectMapper().writeValueAsString(note))
        )
            .andDo(print())
            .andExpect(status().isBadRequest());
    }
}

```

In the `shouldCreateNoteWithTitleAndContentAndReturnHttp201Created` test, a note with title and content filled is sent to the `/notes` endpoint through an HTTP POST request. The test then expects that the HTTP response code is 201 (Created) and the JSON returned in the response body contains a note with a not null `id` and the `title`, `content` and `wordCount` corresponding exactly to the note sent in the request.

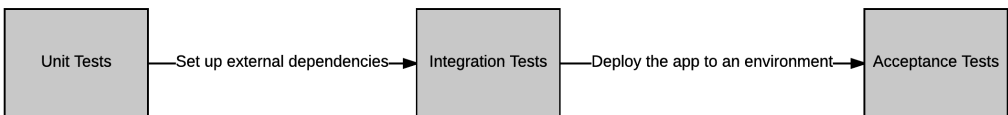
In the `shouldNotCreateNoteWhenTitleIsEmptyAndReturnHttp400BadRequest` test, a note with an empty title is sent to the `/notes` endpoint through an HTTP POST request. The test then expects that the HTTP response code is 400 (Bad Request). Similarly, the `shouldNotCreateNoteWhenContentIsEmptyAndReturnHttp400BadRequest` test sends a note with an empty content and expects a response code 400 (Bad Request).

# Acceptance Tests

We have tested the backend with unit and integration tests, so theoretically, if an HTTP POST request is sent to the /notes endpoint with a JSON corresponding to a note with title and content filled, the note will be successfully persisted to the database.

Now we need to ensure the behavior for the frontend by simulating the actual end user interacting with the page in the web browser (clicking buttons and links, filling in and submitting forms, and so on). For that, we code acceptance tests using [Selenium](http://www.seleniumhq.org/)<sup>35</sup>.

Acceptance tests provide even slower feedback than integration tests. They are even more difficult to set up than integration tests because they actually require the Notepad application to be deployed to an environment whereas integration tests could be run just by setting up a MySQL instance on localhost.



Testing Steps

---

<sup>35</sup><http://www.seleniumhq.org/>

## Page Object: NewNotePage.java

Interacting directly with the Selenium `WebDriver` in the test code may lead to a test that is hard to read and understand. In order to avoid that, a common pattern implemented is the [Page Object](https://martinfowler.com/bliki/PageObject.html)<sup>36</sup>. Basically, it encapsulates the web page functions and states in a separated component that exposes a simple and easy-to-use API for your tests.

Below is `NewNotePage.java` class, which is the page object used to encapsulate the web page that allow users to create new notes.

```
public class NewNotePage {
    @FindBy(id="newNote")
    private WebElement newNoteModal;

    @FindBy(id="newNoteTitle")
    private WebElement title;

    @FindBy(id="newNoteContent")
    private WebElement content;

    @FindBy(id="btnCreateNewNote")
    private WebElement createNoteButton;

    private Long sleep = 2000L;

    private WebDriver driver;

    public NewNotePage(WebDriver driver) {
        this.driver = driver;
        PageFactory.initElements(driver, this);
    }

    public void create(Note newNote) throws InterruptedException {
        newNoteModal.click();
        sleep(sleep);

        title.sendKeys(newNote.getTitle());
        content.sendKeys(newNote.getContent());
        createNoteButton.click();
    }
}
```

---

<sup>36</sup><https://martinfowler.com/bliki/PageObject.html>

```
        sleep(sleep);  
    }  
  
    public String getMessage() {  
        return driver.findElement(By.className("noty_text")).getText();  
    }  
}
```

The `NewNotePage` constructor uses the driver to map the web page elements into the `newNoteModal`, `title`, `content` and `createNoteButton` `WebElement` attributes in the class.

The `create` method uses the mapped attributes to click the `New Note` button, which will open the modal. Then, it waits `sleep` milliseconds, fill the title and content form inputs and clicks on the `Create` button.

The `getMessage` method just uses the driver to find the element in the page that contains the message shown after the `Create` button in the modal is clicked.

## CreateNoteTest.java

Now that the web page is encapsulated by the `NewNotePage` class, the acceptance tests become much clearer.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = { AcceptanceTestsConfiguration.class })
public class CreateNoteTest {

    @Autowired
    private WebDriver driver;

    @Autowired
    private URI notepadBaseUrl;

    private NewNotePage newNotePage;
    private final String newNoteSuccessMessage = "Your note was successfully saved!";
    private final String newNoteFailMessage = "Title and Content cannot be empty";

    @Before
    public void setUp() {
        driver.get(notepadBaseUrl.toString());
        newNotePage = new NewNotePage(driver);
    }

    @Test
    public void shouldCreateNewNoteWithTitleAndContent() throws InterruptedException {
        Note newNote = new Note("Acceptance Test", "Creating note from the acceptance test");
        newNotePage.create(newNote);
        assertEquals(newNotePage.getMessage(), newNoteSuccessMessage);
    }

    @Test
    public void shouldNotCreateNewNoteWhenTitleIsEmpty() throws InterruptedException {
        Note newNote = new Note("", "Creating note from the acceptance test");
        newNotePage.create(newNote);
        assertEquals(newNotePage.getMessage(), newNoteFailMessage);
    }

    @Test
    public void shouldNotCreateNewNoteWhenContentIsEmpty() throws InterruptedException {
        Note newNote = new Note("Acceptance Test", "");
        newNotePage.create(newNote);
        assertEquals(newNotePage.getMessage(), newNoteFailMessage);
    }
}
```

```
}  
}
```

The `setUp` method uses the driver to navigate to the application web page. The `notepadBaseUrl` contains the application URL.

The `shouldCreateNewNoteWithTitleAndContent` test creates a note and ensure that the success message “Your note was successfully saved!” is shown.

The `shouldNotCreateNewNoteWhenTitleIsEmpty` and the `shouldNotCreateNewNoteWhenContentIsEmpty` tests try to create a note with empty fields and they ensure that the error message “Title and Content cannot be empty” is shown.

The configurations used to set up the acceptance tests are coded in the `AcceptanceTestsConfiguration.java` class.

## AcceptanceTestsConfiguration.java

As mentioned before, the acceptance tests must run against an environment already set up. The point now is: how will the driver know the URL for this environment? The answer is that when you issue the command to run the acceptance tests you should inform it, for example:

```
$ mvn verify -Dacceptance.notepad.url=${NOTEPAD_URL} -Dselenium.browser=firefox
```



The same stills for the `selenium.browser` system property.

The `acceptance.notepad.url` property is then caught inside the test configuration using the `System.getProperty("acceptance.notepad.url", "http://localhost:8080")` method invocation. The second argument is only the default value when the `acceptance.notepad.url` property is not provided.

```
@Configuration
public class AcceptanceTestsConfiguration {
    private final static String NOTEPAD_URL = System.getProperty("acceptance.notepad.url", \
"http://localhost:8080");

    // Ommited the webDriver method here. See the Selenium-Grid section!

    @Bean
    public URI baseUrl() throws URISyntaxException {
        return new URI(NOTEPAD_URL);
    }
}
```

Now, let's understand what Selenium-Grid is before diving into the `webDriver` method.

## Distributed Acceptance Tests with Selenium-Grid

Sometimes the same frontend code behaves different in different browsers and platforms (Android, Windows, Linux, etc). If you want to ensure that your application



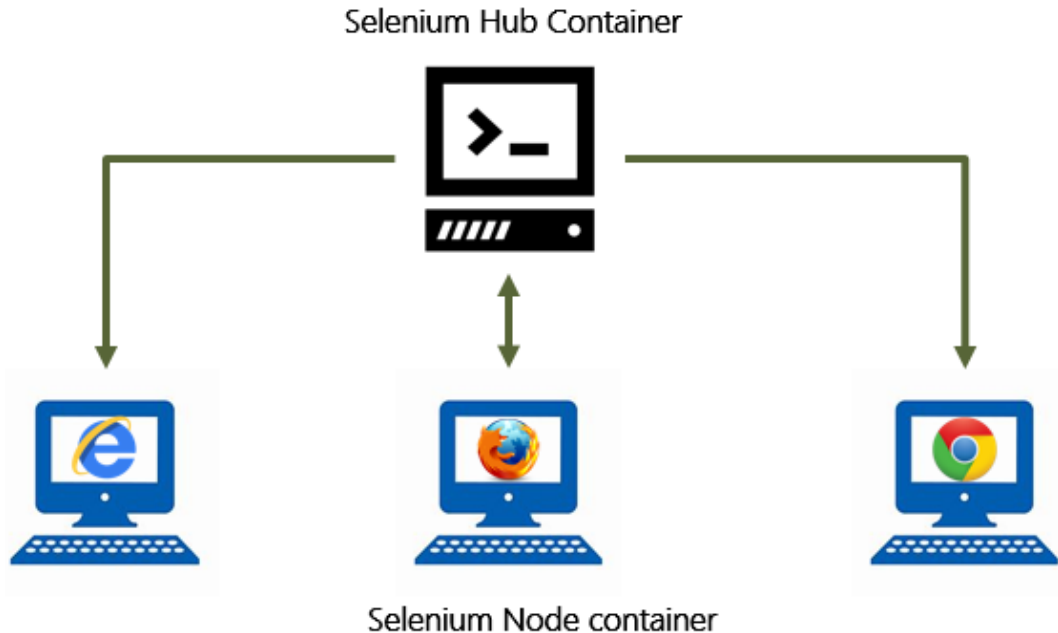
works in different browsers and platforms you have to execute your acceptance tests in different browsers and platforms; there is no magic. Fortunately, Selenium-Grid provides you the ability to do that very easily.

Selenium-Grid allows you run your tests on different machines against different browsers in parallel. That is, running multiple tests at the same time against different machines running different browsers and operating systems. Essentially, Selenium-Grid support distributed test execution. It allows for running your tests in a distributed test execution environment.

Selenium-Grid gives you the ability to:

- Run your tests against multiple browsers, multiple versions of browser, and browsers running on different operating systems.
- Reduce the time it takes for the test suite to complete a test pass.

This is architecture diagram for the Selenium-Grid:



Selenium-Grid with Hub and Nodes Architecture

A grid consists of a single hub and one or more nodes.



As in this book we use Kubernetes to orchestrate Docker containers, both the hub and the nodes are actually created as Docker containers.

Basically, the test connects to the hub using the address and port provided in the `SELENIUM_URL` constant got from the `selenium.url` system property. Then, the hub receives the test to be executed along with the information on which browser and platform where the test should run (the browser information is the `SELENIUM_BROWSER` constant got from the `selenium.browser` system property and the platform is set as `Platform.ANY`). Based on these informations, the hub selects an available node that has the requested browser-platform combination. For example, if there is a node with Linux platform and Firefox browser and the test should run on Linux and Firefox, then the hub will send Selenium commands to be executed in this node. The node runs the browser and executes the Selenium commands within that browser against the application under test.

So, consider you have the following situation (which, by the way, is exactly what is going to happen in our continuous delivery pipeline later on this book):

- Hub with two nodes connected: Node1 and Node2.
- Node1 is running Chrome.
- Node2 is running Firefox.

If you trigger two acceptance test executions at the same time and the first has the `selenium.browser=chrome` system property and the second has the `selenium.browser=firefox` system property, then the hub will assign an execution for each node and obviously the total amount of time spent on the whole execution would be the half compared to a sequential execution. Nice, isn't it?

```
@Configuration
public class AcceptanceTestsConfiguration {
    private final static String NOTEPAD_URL = System.getProperty("acceptance.notepad.url", \
"http://localhost:8080");
    private final static String SELENIUM_URL = System.getProperty("selenium.url", "http://1\
ocalhost:4444/wd/hub");
    private final static String SELENIUM_BROWSER = System.getProperty("selenium.browser", "\
chrome");

    @Bean(destroyMethod = "quit")
    public WebDriver webDriver() throws Exception {
        DesiredCapabilities capabilities = new DesiredCapabilities(SELENIUM_BROWSER, "", Plat\
form.ANY);
        WebDriverException ex = null;
        for (int i = 0; i < 10; i++) {
            try {
                return new RemoteWebDriver(new URL(SELENIUM_URL), capabilities);
            } catch (WebDriverException e) {
                ex = e;
                System.out.println(String.format("Error connecting to %s: %s. Retrying", SELENIUM\
_URL, e));
                Thread.sleep(1000);
            }
        }
        throw ex;
    }
}

@Bean
```

```
public URI baseUri() throws URISyntaxException {  
    return new URI(NOTEPAD_URL);  
}  
}
```

# Smoke Tests

Smoke tests are a simple subset of integration/acceptance tests that are executed against the production servers after a deployment (but before they serve traffic to end users) to ensure that the new version deployed is up and running. These tests are intended to make sure that the main application functionalities are still working properly.

In the Notepad application, there is only a single smoke test, which is the `should-CreateNewNoteWithTitleAndContent` test of the [CreateNoteTest](#) class, responsible for ensuring that a user can create a note with title and content.

# Performance Tests

Almost every application has non-functional requirements regarding performance such as “even with 100 users connecting at the same time, the main page should take no more than 2s to load”. You can ensure that new release candidates of your software are still meeting the performance requirements by integrating to your continuous delivery pipeline the so called performance tests. A very handy tool that allow us to create performance tests is [Gatling](http://gatling.io/)<sup>37</sup>.

---

<sup>37</sup><http://gatling.io/>

# Gatling

[Gatling](#)<sup>38</sup> is a powerful tool that allow us to create performance tests to cover a variety of scenarios. Basically, a performance test for a web application consists of:

1. Simulating a large number of users with complex behaviors.
2. Collecting and aggregating all the requests' response times.
3. Creating reports and analyzing data.

Gatling's code-like scripting enables you to easily create and maintain testing scenarios. It provides an easy-to-use [Domain Specific Language](#)<sup>39</sup> in [Scala](#)<sup>40</sup> and requires very little knowledge in programming to get started.

Gatling's architecture is asynchronous as long as the underlying protocol, such as HTTP, can be implemented in a non blocking way. This kind of architecture allows implementing virtual users as messages instead of dedicated threads, making them very resource cheap. Thus, running thousands of concurrent virtual users is not an issue.

At the end of the tests, Gatling automatically generates a dynamic and colorful [report](#)<sup>41</sup> with useful information. Have a look at some examples:

---

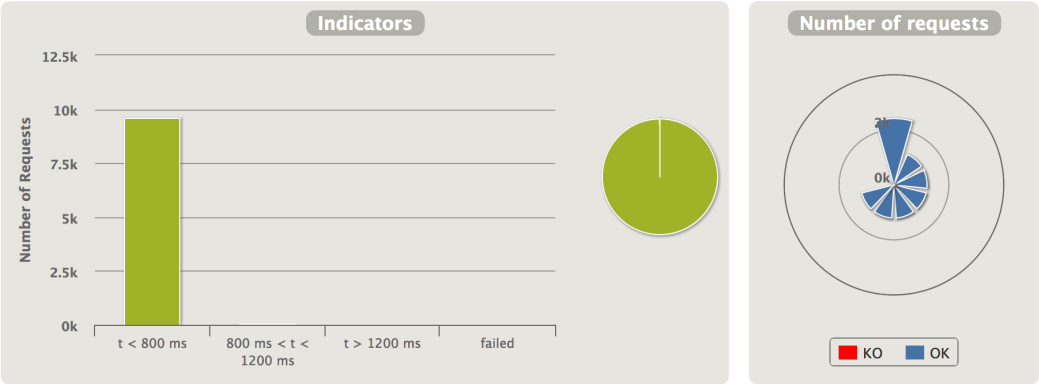
<sup>38</sup><http://gatling.io/>

<sup>39</sup>[https://en.wikipedia.org/wiki/Domain-specific\\_language](https://en.wikipedia.org/wiki/Domain-specific_language)

<sup>40</sup><https://www.scala-lang.org>

<sup>41</sup><http://gatling.io/docs/2.3/general/reports>

Global Information



Response times distribution among standard ranges

STATISTICS

Expand all groups | Collapse all groups

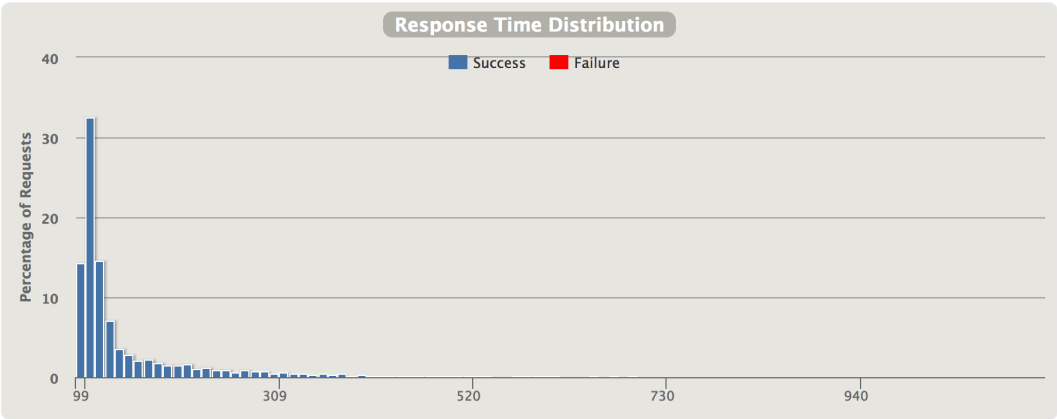
Requests ^	Executions				Response Time (ms)						
	Total ^	OK ^	KO ^	% KO ^	Min ^	Max ^	Mean ^	Std Dev ^	95th pct ^	99th pct ^	Req/s ^
Global Information	9661	9659	2	0 %	94	1145	159	108	363	663	140.6
Home Redirect 1	2410	2410	0	0 %	94	1145	150	101	335	625	35.06
Search	1205	1205	0	0 %	97	965	156	106	366	656	17.53
Select	1205	1205	0	0 %	95	980	155	108	352	684	17.53
Page 0	1205	1205	0	0 %	100	1134	169	122	389	733	17.53
Page 1	1205	1205	0	0 %	100	1122	163	113	374	687	17.53
Page 2	1205	1205	0	0 %	100	992	165	110	367	677	17.53
Page 3	1205	1205	0	0 %	100	978	163	101	365	561	17.53
Form	7	7	0	0 %	98	512	194	148	453	500	0.10
Post Redirect 1	14	12	2	14 %	99	483	150	103	354	457	0.20

ERRORS

Error ^	Count ^	Percentage ^
status.is(201), but actually found 200	2	100.0 %

Standard statistics





Response time distribution

## HomeSimulation.scala

In the Notepad application, we want to ensure that even when 100 users connects to the home page at once there are no failed requests.

```
package notepad

import io.gatling.core.Predef._
import io.gatling.http.Predef._
import scala.concurrent.duration._

class HomeSimulation extends Simulation {

  val httpConf = http
    .baseUrl(System.getProperty("notepadUrl")) // Get the URL from a system property
    .acceptHeader("text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8") // Here are the common headers
    .acceptEncodingHeader("gzip, deflate")
    .acceptLanguageHeader("en-US,en;q=0.5")
    .userAgentHeader("Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8; rv:16.0) Gecko/20100101 Firefox/16.0")

  val scn = scenario("Should List Notes")
    .exec(http("request")
      .get("/")
    )

  setUp(scn.inject(atOnceUsers(100)).protocols(httpConf)).assertions(
    global.failedRequests.percent.is(0)
  )
}
```

The baseUrl, which corresponds to the Notepad URL in which the performance tests will be executed on, is obtained from a system property just the acceptance tests. The scenario is just opening the application home page by issuing an HTTP GET request to /. Then, 100 users are injected at once in the scenario and the test ensures that the percentage of failed requests must be 0.



If for example you want to ensure that the maximum response time allowed is 3s, then you could just include the assertion `global.responseTime.max.lt(3000)` to the assertions. Gatling is really easy to use, isn't it? Check the [assertions](http://gatling.io/docs/2.3/general/assertions)<sup>42</sup> documentation for further information.

---

<sup>42</sup><http://gatling.io/docs/2.3/general/assertions>

# Apache Maven

The Notepad application uses [Apache Maven](https://maven.apache.org/)<sup>43</sup> as the build automation tool. Using Maven you can easily run automated tests, package the application, deploy a snapshot version to a [repository manager](https://www.sonatype.com/nexus-repository-sonatype) such as [Nexus](https://www.sonatype.com/nexus-repository-sonatype)<sup>44</sup> or [Artifactory](https://www.jfrog.com/open-source)<sup>45</sup> and much more. Maven also manages the application dependencies that you declare in a special file called **pom.xml** in your Maven project. For example, if your project uses Selenium for acceptance tests, you just have to declare its dependency to pom.xml file:

```
<dependencies>
  <dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
  </dependency>
  ...
</dependencies>
```

---

<sup>43</sup><https://maven.apache.org/>

<sup>44</sup><https://www.sonatype.com/nexus-repository-sonatype>

<sup>45</sup><https://www.jfrog.com/open-source>

## Maven Snapshot vs Release

A snapshot version in Maven is one that has not been released yet, which means it is still under development and may still change a lot. For example, as we have not started the continuous delivery pipeline project for this book, we assume that the Notepad application is under development and has not been released yet, therefore you will find a `1.0.0-SNAPSHOT` version in the Notepad's [pom.xml](#) file:

```
<groupId>com.jorgeacetozi</groupId>
<artifactId>notepad</artifactId>
<version>1.0.0-SNAPSHOT</version>
<packaging>jar</packaging>
```

After we set up the continuous delivery pipeline and release the application for the first time, the released version will be **1.0.0** and the next development version (snapshot) will automatically be updated to **1.0.1-SNAPSHOT**, which means that the version 1.0.0 (the release version) will not change anymore and is a candidate to be deployed to production.

## The Default Lifecycle and its Phases

Maven has three built-in build lifecycles: default (or build), clean and site. The default lifecycle handles the project deployment. Below are some phases that make up the default lifecycle. Note that the order matters, that is, the phases are executed in the presented order. (for a complete list of the lifecycle phases, refer to the [Lifecycle Reference](#)<sup>46</sup>).

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

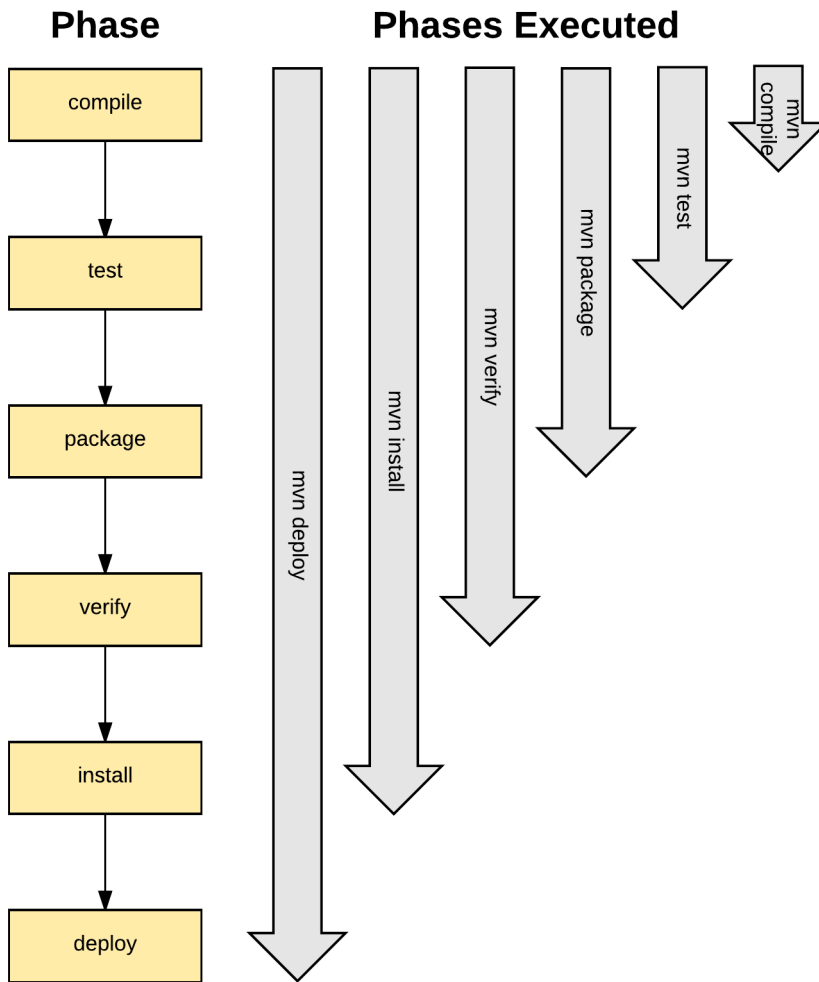
To invoke a specific phase in the command line, just issue the `mvn phase_here` command. For example, to invoke the `deploy` phase, issue the command:

```
$ mvn deploy
```

When you invoke a specific phase, each phase prior to it is executed too (in the presented order). So, when you execute `mvn deploy`, the phases are executed in the following order: `validate`, `compile`, `test`, `package`, `verify`, and finally `deploy`.

---

<sup>46</sup><https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>



Maven Phases Ordered Execution

# Maven Repositories

The artifacts produced and consumed by Maven projects are stored in repositories. There are two types of repositories:

- **Local:** The local repository is located in your local machine at `$HOME/.m2/repository` (by default). When you issue the `mvn install` command, your project is deployed into the local repository.
- **Remote:** When a requested artifact is not found in the local repository (for example, the Selenium dependency for acceptance tests), Maven will download it from a remote repository and then store it in the local repository. Next time this dependency is needed, it will be already in the local repository and there will be no need to spend bandwidth downloading it. By default, Maven will download from the [Maven Central Repository](http://repo.maven.apache.org/maven2/)<sup>47</sup>.



Some dependencies are not available in the Maven Central Repository. In this case, you can declare additional remote repositories to your projects' `pom.xml` file.

---

<sup>47</sup><http://repo.maven.apache.org/maven2/>



## Repository Manager

A repository manager is a dedicated server application (such as [Nexus](#)<sup>48</sup> or [Artifactory](#)<sup>49</sup>) designed to manage repositories of binary components. In this book's context, the binary components are the Notepad application dependencies that have to be downloaded and the Notepad artifacts (snapshots and releases) that have to be published.

If you are a developer working alone on a project, you might not need a repository manager because once you have downloaded the project's dependencies once, they would be stored in your local Maven repository so you would not have to download them over and over again. However, if you work for a company where people are constantly downloading dependencies and sharing their artifacts with other teams, the usage of a repository manager is considered essential.

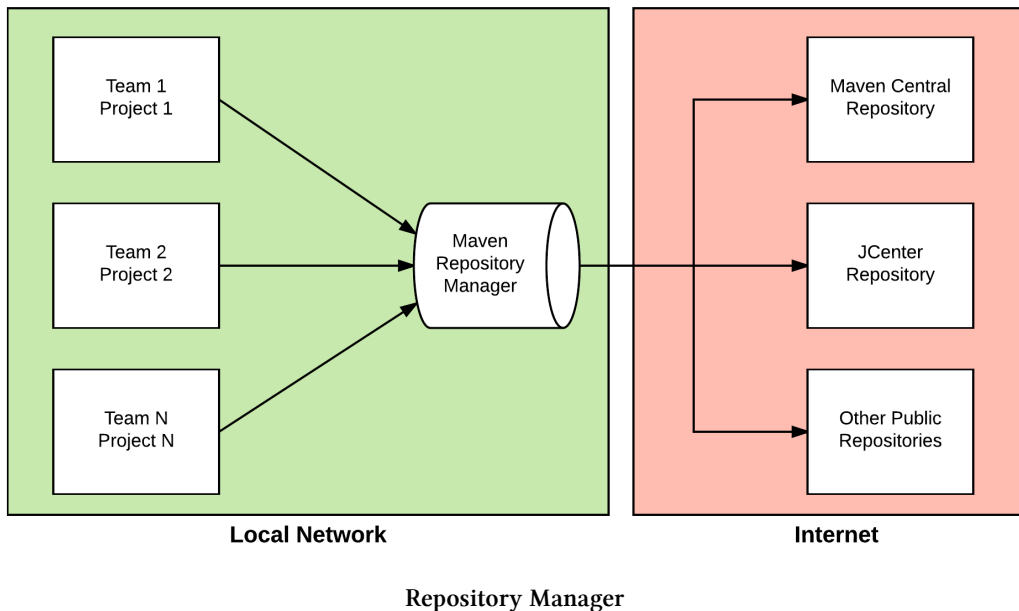
In short, a repository manager serves two essential purposes:

1. Act as dedicated proxy server for public Maven repositories. It means that every artifact or dependency that a project might need would be stored on a single location available on the internal network, which happens to be more reliable and faster than downloading from public remote repositories. Besides, this would avoid spending a lot of outbound network traffic, which may represent cost reduction for companies.
2. Provide repositories as a deployment destination for your Maven project outputs. For example, a team might develop a SDK while other team uses it in another project.

---

<sup>48</sup><https://www.sonatype.com/nexus-repository-sonatype>

<sup>49</sup><https://www.jfrog.com/open-source>



Using a repository manager also provides other handy features such as fine-grained access controlling (which user is allowed to use which artifacts or deploy to which repositories), RESTful APIs, and so on.

In this book, we are going to set up and use Artifactory (open-source version) as the repository manager.

**In the continuous delivery pipeline we are going to implement later on this book, when the `mvn deploy` command is issued, a snapshot version is created and published to the Maven snapshot repository in Artifactory.**

# Introduction to Docker

In the continuous delivery pipeline we are going to implement later on this book, each Notepad release version is distributed as a Docker image and each environment runs a MySQL instance as a Docker container.

The question now is: Why? The artifact (`notepad-1.0.0.jar`) is already stored and available in the [Repository Manager](#) and I can install MySQL on any operating system just by following the official installation instructions. So, why do I need Docker?

Well, have you ever heard the sentence “I don’t know what is happening; it works on my machine”? What about the famous “this version was working perfectly on staging but it’s not working on production”? One of the main reasons why these issues happen is because environments often become different over time, especially those managed manually (like your workstation if you don’t use something like [Vagrant](#)<sup>50</sup>, for example). In different environments, People are often installing different libraries on different versions in different environments (which can also run different operating systems). The result is that, for example, the staging environment is no longer a mirror from production over time. If the staging is not a mirror from production anymore, of course that the application running on these environments could behave different on each environment, what leads to bugs that happen in a environment but not in another.

In short, Docker allows you to easily run services (MySQL, Jenkins, Nginx, your own application like Notepad, etc) on a machine. Docker guarantees that these services will always be in the same state across executions, regardless of the underlying operating system or system libraries. In other words, if your staging environment is very different from your production environment (system libraries, operating systems, etc) and you run a application like Notepad as a Docker container on both environments, it’s guaranteed that the application would behave exactly the same on these environments regardless of their differences.

---

<sup>50</sup><https://www.vagrantup.com>

In my humble opinion, this is the most important benefit got from using Docker, but there are many more:

- It's easy to run services as Docker containers (you did that when running integration tests. It's quite easy, isn't it?). Thus, it also helps a lot in the development phase because you don't have to waste time installing and configuring tools on your operating system.
- Docker is a highly collaborative tool. You can reuse Docker images that people build and share publicly.
- It encourages the infrastructure as code model because a Docker image is entirely described on a file called a Dockerfile that can (and should) be versioned in the source control.
- Docker has a great community, and it's expanding quickly.

# Difference Between Container and Image

Docker images are binary files that contain everything needed to run a specific service. When you instantiate a service from a Docker image, you say that you create a Docker container. As an analogy, if a Docker image is a Java class, then a Docker container is an object. Many containers can be created from a single image.

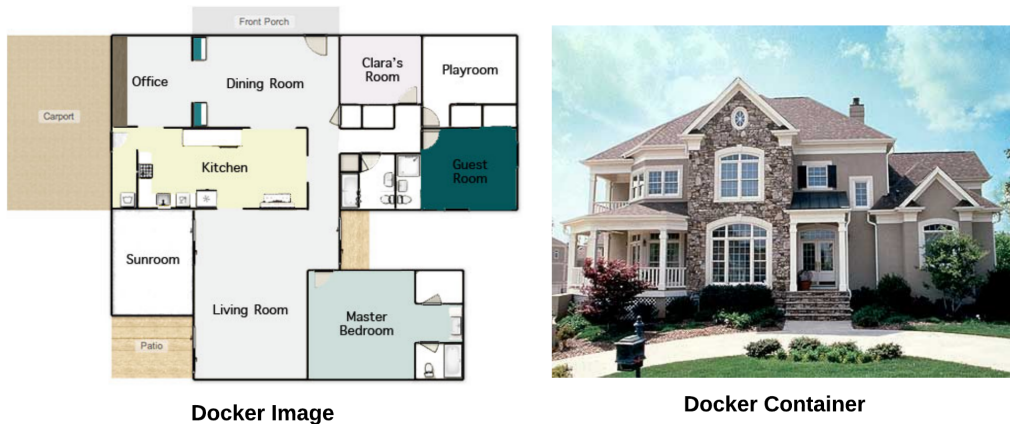


Image vs Container

As the figure above shows, an image is like a house scheme whereas the container is the concrete house where people actually live in.



Technically speaking, a container is a group of processes contained in an isolated environment, but running on the same kernel as the host operating system. This isolation is provided by concepts like `cgroups` and `namespaces`. For example, if you create a file inside a container, this file cannot be accessed from the host operating system (unless you explicitly specify this).

Docker images can be either created from a container state (like a snapshot of a

running container) or describing the image state (like the operating system, libraries and applications) on a special file called `Dockerfile`. Both work, but which way do you think it's better? That's right, the `Dockerfile` way, because it encourages the infrastructure as code model, that is, the image state is easily tracked using a source control and safely evolves as every single change would go through a pull request, code review, and so on.



If you remember, that's pretty much the idea behind [Flyway](#). Having database migrations versioned in the source control allows to gain control over the database state across different application versions.

You create a Docker image from a Dockerfile using the **`docker image build`** command and create a Docker container by executing the **`docker container run`** command.

# Jenkins Overview

[Jenkins](#)<sup>51</sup> is a leading open source CI server written in Java. It provides hundreds of plugins to support building, deploying and automating any project. As Jenkins is a CI server, **its main purpose is to run automated tests on every new commit that emerges in the remote repository mainline**, but it is actually capable of doing much more than that such as triggering releases generation, deploying to a specific environments, executing database scripts, and so on. Actually, Jenkins can be used to automate all sorts of tasks related to building, testing, and deploying software.

Let's set up Jenkins master using Docker. Start Jenkins as a Docker container binding the 8080 port to the web user interface, the 50000 port for connecting Jenkins slaves and mounting a volume to the ~/jenkins\_home directory in your machine to make Jenkins data persistent:

```
$ docker container run -d --name jenkins-master -p 8080:8080 -p 50000:50000 -v ~/jenkins_\nhome:/var/jenkins_home jenkins/jenkins:lts
```

Follow the container initialization logs and copy the initial admin password automatically generated:

```
$ docker container logs -f jenkins-master
```

...

```
Jenkins initial setup is required. An admin user has been created and a password generate\nd.
```

Please use the following password to proceed to installation:

```
b71ac23964344032820affa87dae0737
```

This may also be found at: /var/jenkins\_home/secrets/initialAdminPassword

...

---

<sup>51</sup><https://jenkins.io>

Open your browser and navigate to `http://localhost:8080`. Paste the password and click on **Continue**.

Getting Started

## Unlock Jenkins

To ensure Jenkins is securely set up by the administrator, a password has been written to the log ([not sure where to find it?](#)) and this file on the server:

```
/var/jenkins_home/secrets/initialAdminPassword
```

Please copy the password from either location and paste it below.

Administrator password

Continue

### Jenkins Initial Password

Click on **Install Suggested Plugins**:



Getting Started

## Customize Jenkins

Plugins extend Jenkins with additional features to support many different needs.

### Install suggested plugins

Install plugins the Jenkins community finds most useful.

### Select plugins to install

Select and install plugins most suitable for your needs.

### Install Suggested Plugins

It should start installing the plugins as shown bellow:

Getting Started

## Getting Started

✓ Folders Plugin	✓ OWASP Markup Formatter Plugin	⚙ build timeout plugin	⚙ Credentials Binding Plugin	** bouncycastle API Plugin
⚙ Timestamper	⚙ Workspace Cleanup Plugin	⚙ Ant Plugin	⚙ Gradle Plugin	Folders Plugin
⚙ Pipeline	⚙ GitHub Branch Source Plugin	⚙ Pipeline: GitHub Groovy Libraries	⚙ Pipeline: Stage View Plugin	** Struts Plugin
⚙ Git plugin	⚙ Subversion Plug-in	⚙ SSH Slaves plugin	⚙ Matrix Authorization Strategy Plugin	** JUnit Plugin
✓ PAM Authentication plugin	⚙ LDAP Plugin	⚙ Email Extension Plugin	⚙ Mailer Plugin	OWASP Markup Formatter Plugin
				PAM Authentication plugin
				** Windows Slaves Plugin

### Installing Plugins

Fill in the fields as shown bellow and click on **Save and Finish**:

**Getting Started**

# Create First Admin User

Username:

Password:

Confirm password:

Full name:

E-mail address:

Jenkins 2.60.3

[Continue as admin](#)

[Save and Finish](#)

## Create First Admin User

Click on **Start using Jenkins**:

## Getting Started

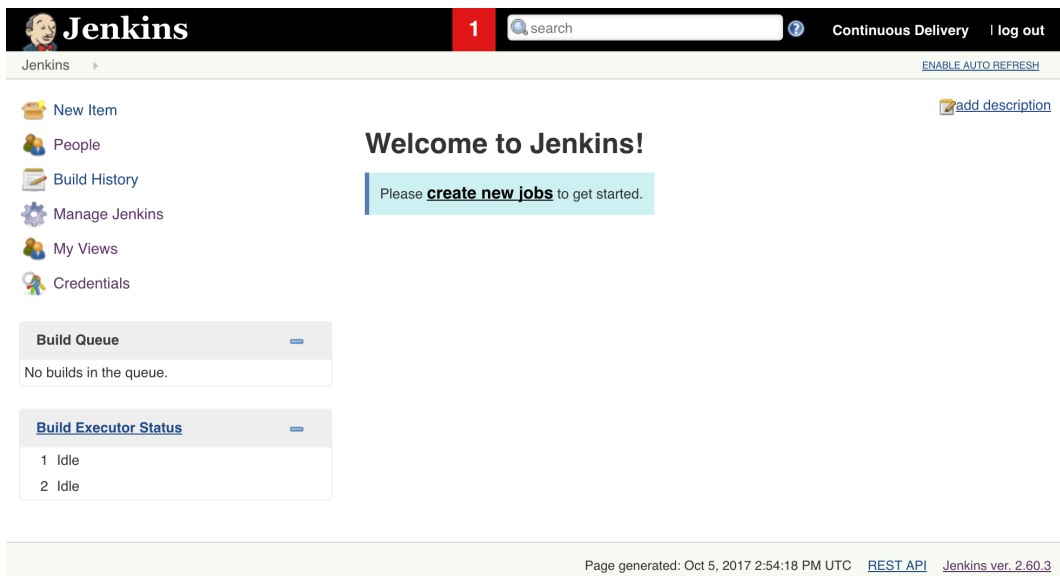
# Jenkins is ready!

Your Jenkins setup is complete.

Start using Jenkins

### Start Using Jenkins

You should see the Jenkins home:



The screenshot shows the Jenkins home page. At the top is a black header with the Jenkins logo, a red box with the number '1', a search bar, and links for 'Continuous Delivery' and 'log out'. Below the header is a sidebar with links: 'New Item', 'People', 'Build History', 'Manage Jenkins', 'My Views', and 'Credentials'. The main content area has a 'Welcome to Jenkins!' message and a button to 'create new jobs'. Below this are two sections: 'Build Queue' (showing 'No builds in the queue.') and 'Build Executor Status' (showing two 'Idle' executors). At the bottom, a footer indicates the page was generated on Oct 5, 2017, and provides links for the REST API and Jenkins version 2.60.3.

Jenkins

1 search

Continuous Delivery | log out

Jenkins

ENABLE AUTO REFRESH

New Item

People

Build History

Manage Jenkins

My Views

Credentials

add description

### Welcome to Jenkins!

Please **create new jobs** to get started.

**Build Queue**

No builds in the queue.

**Build Executor Status**

1 Idle

2 Idle

Page generated: Oct 5, 2017 2:54:18 PM UTC [REST API](#) [Jenkins ver. 2.60.3](#)

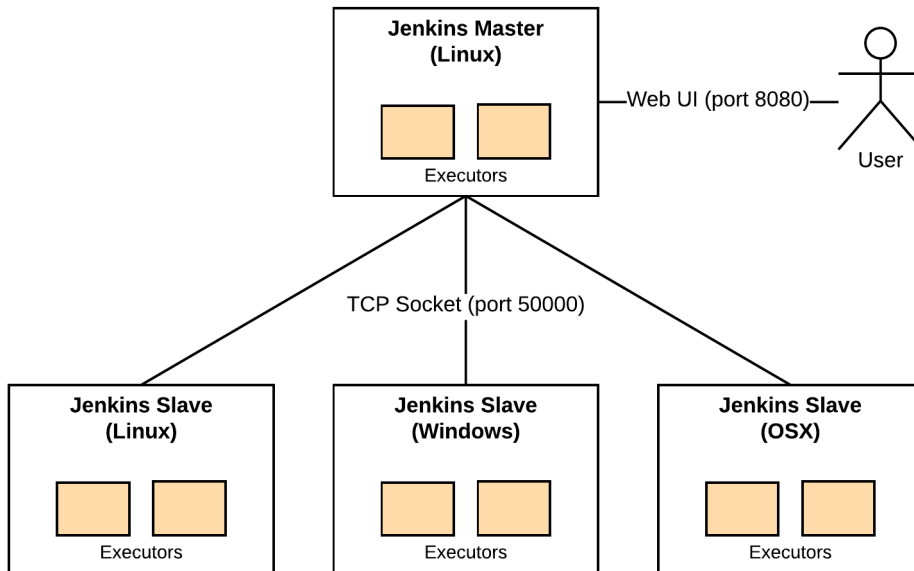
### Jenkins Home

## Node, Master, and Agent (or Slave)

It's important to have a clear understanding on the Node, Master, and Agent (or Slave) concepts.

The most basic Jenkins setup relies on a single instance **Jenkins master** installation. In this case, the master is responsible for everything: coordinating process which stores configuration, loading plugins, rendering the user interface, scheduling build jobs, executing build jobs (the heavy work part), and so on. At first this might not be an issue, but if you start to use Jenkins a lot with just the Jenkins master, you will most likely find that you will run out of resources (memory, CPU, etc.). At this point, you will have either to scale up your instance (adding more resources to it) or to scale out by setting up **Jenkins agents (also called as slaves)**, which are typically a machine, or container, which connect to a Jenkins master and executes tasks when directed by the master, relieving the master from the heavy work.

A node is a machine, or container, which is part of the Jenkins environment. **Both the master and agents are considered to be nodes.**



**Jenkins Distributed Architecture Diagram**

You may be thinking now “Isn’t running two slaves on the same Jenkins master machine with the number of executors set to zero the same as having a single Jenkins master with the number of executors set to two?” and that is a very good question. Actually, in terms of the ability to execute two builds concurrently, maybe yes. However, you must be aware that starting a slave actually starts a main process, which consumes the main memory to run and connect to the master. On the other hand, the executor is just a thread, that is, a sub-process of the main process, which is cheaper in terms of memory. Therefore, if you are looking for scalability, the real power of scaling out with Jenkins slaves comes when you actually run them on separate machines, making up a distributed system.

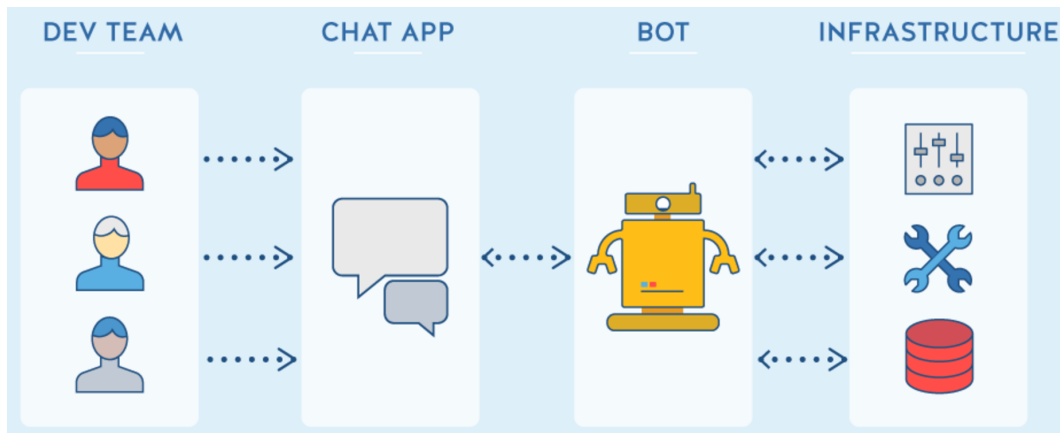
We are going to set up Jenkins slaves on Docker in the [Scaling Jenkins with Slaves](#) section.

# ChatOps

Jenkins plugins allows to extend the Jenkins core functionalities. You have already installed some when you set up the Jenkins installation. Now, we are going to create an integration between [Slack](https://slack.com/)<sup>52</sup> and Jenkins so that our jobs can send notifications to a Slack Channel.

[Slack](https://slack.com/)<sup>53</sup> is an amazing chat and collaboration tool. You can integrate all your tools within Slack (like we are going to do with Jenkins now) and have a single point of communication connecting people, bots, and tools in an automated and transparent flow. This is also known as **ChatOps**.

ChatOps is the use of chat clients, chat bots and real-time communication tools to facilitate how software development and operation tasks are communicated and executed.



ChatOps

ChatOps give users access to important information directly from the chat window, drastically reducing context switching. This not only serves as an important source

---

<sup>52</sup><https://slack.com/>

<sup>53</sup><https://slack.com/>

of information but also makes up an history, allowing people that were away to catch up with situations quickly when they become available.

ChatOps also allows interaction between people and bots to perform a variety of actions. What these actions could be? Well, basically anything (seriously!):

- Show [Grafana](#)<sup>54</sup> dashboards for Login activity in the last 30 minutes.
- Pull the production ERROR logs in the last 5 minutes from Elasticsearch.
- Scale the number of canary replicas on a Canary Release deployment.
- Change the log level of an application at runtime.

And so on! In the end of this chapter you will trigger the `my-first-job` job we have been working on the last section using Slack and Hubot.



**Jorge Acetozi** 11:43 AM ☆

xuxa jenkins build my-first-job



**xuxa** APP 11:43 AM

jorge.acetozi: (201) Build started for my-first-job <http://jenkins-master:8080/job/my-first-job>



**jenkins** APP 11:43 AM

my-first-job - #6 Started by user Continuous Delivery ([Open](#))

my-first-job - #6 Success after 24 sec ([Open](#))

### Trigger Build From Slack Using Hubot

---

<sup>54</sup><https://grafana.com/>

# Why Kubernetes?

[Kubernetes](https://kubernetes.io/)<sup>55</sup> is an open-source system for automating deployment, scaling, and management of containerized applications. Kubernetes builds upon 15 years of experience of running production workloads at Google, combined with best-of-breed ideas and practices from the community.

Let's start with a question. Would you deploy a Jenkins distributed architecture for the production environment of your company using Docker on a single machine with several containers in it, just like we did in the [Scaling Jenkins with Slaves](#) section? Of course not! Why not? What is wrong with this approach? Let's list out some arguments below:

- The containers (Jenkins master and slaves) would be running on a single machine. What would happen if this machine crashes?
- If some of those containers crash, how would you know about that? Besides, you would have to bring them up again by yourself logging into the machine and executing the “docker run” command manually (and that is not funny when it's 3:00 AM!).

Actually, even if you run these containers on separate machines, you would still have to manage them one by one by yourself.

Of course that you could address these issues by running a [systemd unit](#)<sup>56</sup> for each container on each machine that would ensure the containers would always be running even if they crash. However, again, what if some machines crash? Who would have to start new containers in other machines in order to bring your Jenkins distributed architecture up again? That's correct, you! However, if Kubernetes were managing these containers for you, it would be in charge of relaunching them again on the most appropriate machines available in the Kubernetes cluster.

---

<sup>55</sup><https://kubernetes.io/>

<sup>56</sup><https://www.freedesktop.org/software/systemd/man/systemd.unit.html>



Well, that might be arguments enough to use Kubernetes, but there are still much more. Forget about the Jenkins thing and imagine now that you are running a stateless web application using Docker containers. How would you deploy a new application release? Manually removing containers and launching new ones based on the new Docker image? Well, as a DevOps Engineer (or whatever you want to call it), does it sound good to read the word “manually”? I bet you know the key for the DevOps culture is automation.

I think you are already convinced that, for production purposes, running and managing (or a more sophisticated word “orchestrating”) container life cycles by yourself may not be a good idea. So, who is going to do this for you? That’s right, Kubernetes! That’s why we call tools like Kubernetes, [Docker Swarm](#)<sup>57</sup>, [Amazon ECS](#)<sup>58</sup> and others as **Container Orchestration Tools**.

Kubernetes has a set of great features that are very useful specially when it comes to production environments.

- Container replication among different nodes to ensure high availability.
- Automatic container recovery when it crashes for whatever reason.
- Container auto-scaling based on Kubernetes cluster metrics such as CPU consumption.
- Rolling deployments and deployment rollbacks.
- Service Discovery. That’s neat for deploying Microservices!
- Load Balancing and Volume management.
- Container health checks.
- Configuration management.
- Logical resources isolation using Namespaces.
- Controlling resources and quotas by Namespaces.
- Managing cron jobs.
- Providing authentication and authorization.

Actually, there are even more than that! In the next sections I will dive into some Kubernetes architectural concepts, then we will cover concepts such as Pods, Replica Sets, Services, Deployments, and so on.

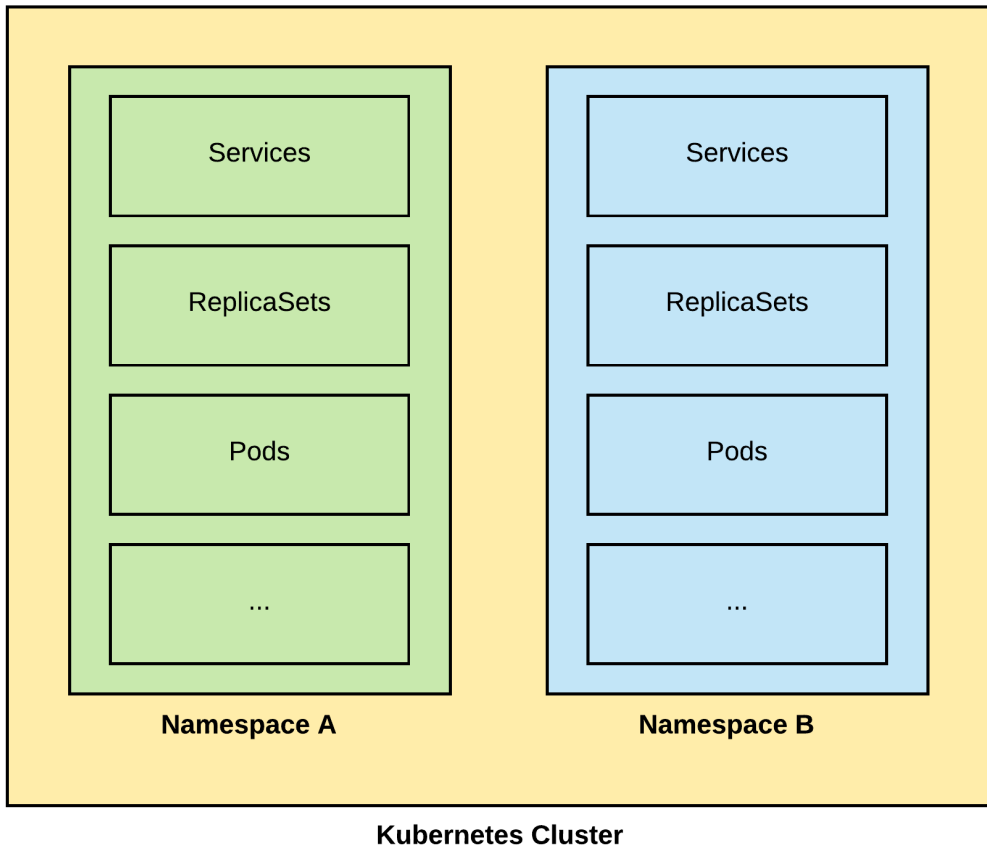
---

<sup>57</sup><https://github.com/docker/swarm>

<sup>58</sup><https://aws.amazon.com/ecs/>

# Namespaces

Kubernetes supports splitting your cluster into multiple isolated virtual clusters. These virtual clusters are called Namespaces. For example, you can use a namespace per project, per team, per environment or whatever makes sense to your scenario.



## Namespaces



Namespaces are useful in environments with many users spread across multiple teams or projects. For clusters with a few to tens of users, you should not need to create or think about namespaces at all. Start using namespaces when you need the features they provide.

Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces. When you create a resource without specifying a namespace, it's assigned to the **default** namespace, which is automatically created by Kubernetes.

When using namespaces, it's important to define resource quotas to avoid a namespace consuming all the resources of the cluster. Resource quotas can restrict the amount of [memory](#)<sup>59</sup>, [CPU](#)<sup>60</sup>, [number of pods](#)<sup>61</sup>, and so on.



It is not necessary to use multiple namespaces just to separate slightly different resources such as different versions of the same software. Use [labels](#) to distinguish resources within the same namespace.



As you can specify a namespace for every object you create (such as pods, services, deployments and so on), I decided it was important to start the Kubernetes Concepts chapter approaching namespaces. Although we will mostly use the **default** namespace along this book, now you already know that it's possible to split your cluster into multiple isolated virtual clusters.

To list the existing namespaces in your cluster, execute the following command on `tab-kubectl`:

```
$ kubectl get namespaces
NAME          STATUS    AGE
default       Active   40m
kube-public   Active   40m
kube-system   Active   40m
```

List the pods in the **default** namespace:

---

<sup>59</sup><https://kubernetes.io/docs/tasks/administer-cluster/memory-constraint-namespace/>

<sup>60</sup><https://kubernetes.io/docs/tasks/administer-cluster/cpu-constraint-namespace/>

<sup>61</sup><https://kubernetes.io/docs/tasks/administer-cluster/quota-pod-namespace/>

```
$ kubectl get pods
No resources found.
```

List the pods in the **kube-system** namespace:

```
$ kubectl get pods --namespace kube-system
```

NAME	READY	STATUS	RESTARTS	AGE
kube-apiserver-172.17.8.101	1/1	Running	0	40m
kube-controller-manager-172.17.8.101	1/1	Running	0	40m
kube-dns-4223714313-1q0ph	3/3	Running	0	40m
kube-proxy-172.17.8.101	1/1	Running	0	39m
kube-proxy-172.17.8.102	1/1	Running	0	39m
kube-proxy-172.17.8.103	1/1	Running	0	37m
kube-scheduler-172.17.8.101	1/1	Running	0	40m
kubernetes-dashboard-524272573-kfps8	1/1	Running	0	40m

Note that it shows many pods. The **kube-system** namespace is where Kubernetes runs its internal components (some of them were already mentioned before, such as the Scheduler and the API Server).

Just to illustrate, create a file named `namespace-example.yaml` and paste the following content in it:

```
apiVersion: v1
kind: Namespace
metadata:
  name: namespace-example
```



You could create the namespace by directly using the `kubectl create namespace namespace-example` command. I like to create resources using the YAML definition though because you can (and should) put them into the version control. The same is valid for pods, services and so on. Along this book, we are going to create our Kubernetes objects using YAML definitions.

Create the namespace:

```
$ kubectl apply -f namespace-example.yaml
namespace "namespace-example" created
```

Check the existing namespaces again:

```
$ kubectl get namespaces
NAME              STATUS    AGE
default           Active    1h
kube-public       Active    1h
kube-system       Active    1h
namespace-example Active    4s
```

Let's create a ResourceQuota object to limit the amount of running pods in this namespace to 2. Create a file named `pod-quota.yaml` and paste the following content in it:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: pod-demo
spec:
  hard:
    pods: "2"
```

Create the ResourceQuota object:

```
$ kubectl apply -f pod-quota.yaml --namespace=namespace-example
resourcequota "pod-demo" created
```

If you followed the [Hands-on Introduction to Kubernetes](#), you should have a file named `nginx-deployment.yaml` in your filesystem. Just ensure that it declares 3 Nginx replicas and then create the deployment in the `namespace-example` namespace:

```
$ kubectl apply -f nginx-deployment.yaml --namespace=namespace-example
deployment "nginx-deployment" created
```

Check the deployment status:

```
$ kubectl get deployment nginx-deployment --namespace=namespace-example
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment    3         2         2            2           3m
```

Note that only 2 replicas were allowed to run on namespace-example. Let's check the detailed message:

```
$ kubectl get deployment nginx-deployment --namespace=namespace-example --output=yaml
status:
  availableReplicas: 2
  conditions:
  - lastTransitionTime: 2017-10-18T02:47:32Z
    lastUpdateTime: 2017-10-18T02:47:32Z
    message: 'unable to create pods: pods "nginx-deployment-431080787-" is forbidden:
      exceeded quota: pod-demo, requested: pods=1, used: pods=2, limited: pods=2'
    reason: FailedCreate
    status: "True"
    type: ReplicaFailure
```

We exceeded the number of pods defined in the ResourceQuota. Of course this was just a simple example, but it shows the power of resource quotas in action. Delete the namespace and let's move on to the next section:

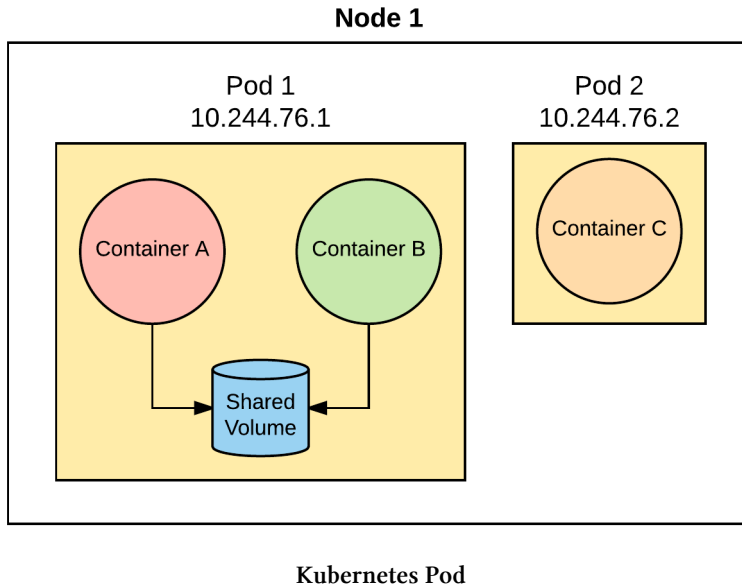
```
$ kubectl delete -f namespace-example.yaml
namespace "namespace-example" deleted
```



When you delete a namespace, everything under it is also deleted!

## Pods

A pod is a group of one or more **related** containers with shared namespaces and shared volumes. A pod is the basic building block of Kubernetes; it's the smallest and simplest unit in the Kubernetes object model that you create. For example, what is the smallest deployable unit on Docker? The container, right? When it comes to Kubernetes, it's the pod. On Kubernetes, a container cannot run outside a pod and every pod run on a node, which can “host” many pods (and these pods don't require to be replicas of the same application, for example, pod 1 can run app A and pod 2 can run app B, and these applications may have absolutely nothing in common).



Although it's pretty common to use Kubernetes with Docker as the container engine, you could use another container engine such as [rkt](https://coreos.com/rkt/)<sup>62</sup> too.

A pod has an independent lifecycle, which means that if you create a pod specifically from pod definition (not from a deployment as we did earlier) and the pod suddenly

---

<sup>62</sup><https://coreos.com/rkt/>

“dies”, then it will not be recreated. If you want to ensure that a number of pods is always running in the cluster, you have to rely on the `ReplicaSet` object.

**All the containers in a pod have the same IP address and port space, which means they can communicate using `localhost`.** Remember when we created a [Jenkins Pipeline to run the Notepad integration tests](#)? We explicitly created a MySQL container and a Maven container and linked them together so that the Notepad container could reach MySQL. If these two containers were grouped into a Kubernetes pod, they would not need this link because they could reach each other by simply using `localhost`.

Containers within a pod can also share volumes. Depending on the volume type, the data is persistent only as long the pod lives or can be permanently persistent. We are going to learn more about volumes later.



In order to avoid unexpected behaviors, please make sure there are no resources running on the cluster before creating the samples along this chapter.

Create a pod definition named `nginx-pod.yaml` and paste the following content in it:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.7.9
    ports:
    - containerPort: 80
```





Typically, you will rarely create pods directly using a pod definition. In practice, you will manipulate other higher-level Kubernetes objects (like deployments, as we did in the [Hands-on Introduction to Kubernetes](#)) and these other objects will actually create the pods that you declare.

On `tab-kubectl`, create the Nginx pod:

```
$ kubectl apply -f nginx-pod.yaml
pod "nginx" created
```

List the pods:

```
$ kubectl get pods -o wide
NAME      READY   STATUS    RESTARTS   AGE   IP            NODE
nginx     1/1     Running   0           2m    10.244.76.3   172.17.8.103
```

Describe the details of the Nginx pod by using the `kubectl describe` command:

```
$ kubectl describe pod nginx
```

```
# huge output here...
```

The `describe` command is pretty useful when it comes to troubleshooting; the `Events` section shows each step of the pod startup:

Events:							
FirstSeen	LastSeen	Count	From	SubObjectPath	Type	Reason	Message
10m	10m	1	default-scheduler		Normal	Scheduled	Successfully assigned nginx to 172.17.8.103
10m	10m	1	kubelet, 172.17.8.103		Normal	SuccessfulMountVolume	MountVolume.SetUp succeeded for volume "default-token-k7nrw"
10m	10m	1	kubelet, 172.17.8.103	spec.containers(nginx)	Normal	Pulled	Container image "nginx:1.7.9" already present on machine
10m	10m	1	kubelet, 172.17.8.103	spec.containers(nginx)	Normal	Created	Created container
10m	10m	1	kubelet, 172.17.8.103	spec.containers(nginx)	Normal	Started	Started container

### Pod Events



As well as `kubectl get`, `kubectl apply`, `kubectl delete`, and others, the `kubectl describe` command is also used for different Kubernetes resources. For example, you could describe a service named `nginx-service` executing the command `kubectl describe service nginx-service`.

Do you remember we have used the `docker exec` command to execute a command inside a Docker container running on your machine? In Kubernetes, we use `kubectl` to do the same:

```
$ kubectl exec nginx -- ls -lh /etc/nginx
drwxr-xr-x. 2 root root 4.0K Jan 27 2015 conf.d
-rw-r--r--. 1 root root 964 Dec 23 2014 fastcgi_params
-rw-r--r--. 1 root root 2.8K Dec 23 2014 koi-utf
-rw-r--r--. 1 root root 2.2K Dec 23 2014 koi-win
-rw-r--r--. 1 root root 3.9K Dec 23 2014 mime.types
-rw-r--r--. 1 root root 643 Dec 23 2014 nginx.conf
-rw-r--r--. 1 root root 596 Dec 23 2014 scgi_params
-rw-r--r--. 1 root root 623 Dec 23 2014 uwsgi_params
-rw-r--r--. 1 root root 3.6K Dec 23 2014 win-utf
```



This is equivalent to `ssh node-02` and execute `docker exec NGINX_-CONTAINER_ID ls -lh /etc/nginx`.

Delete the pod and check that it is **not** automatically recreated:

```
$ kubectl delete pod nginx
pod "nginx" deleted
```

Wait a few seconds and list the pods again:

```
$ kubectl get pods
No resources found.
```

As mentioned before, when there isn't a `ReplicaSet` watching for the pod, once the pod is gone for whatever reason it's not automatically recreated.



A pod definition also lets you specify many other configurations such as container health checks using probes, resources quotas, and so on. We are going to learn more about them later.

Below is a list of useful commands for manipulating pods:

- `kubectl get pods`: list and retrieve basic information about the pods running on the cluster.

- `kubectl get pods -o wide`: list pods showing additional information such as the pod IP and the node IP.
- `kubectl get pod <pod>`: retrieve only information about the specific <pod> (the option `-o wide` can be used here too).
- `kubectl describe pod <pod>`: show detailed information about the specific <pod>.
- `kubectl logs <pod>`: print the logs from a container in the <pod> (1 container case).
- `kubectl logs <pod> -c <container>`: print the logs from a specific container in the <pod> (multi-container case).
- `kubectl exec <pod> -- <command>`: execute a command on a container in the <pod> (1 container case).
- `kubectl exec <pod> -c <container> -- <command>`: execute a command on a specific container in the <pod> (multi-container case).
- `kubectl top pods`: show metrics for all running pods (requires [Heapster](#) running).
- `kubectl top pod <pod>`: show metrics for the specific <pod> (requires [Heapster](#) running).



When using `kubectl`, you can refer to pods as `pod`, `pods` and `po`. For example, you can use `kubectl get po` instead of `kubectl get pods`, it's up to you. Personally, I like to use plural when listing resources (`kubectl get pods`, `kubectl get deployments`, etc) and singular for a particular resource (`kubectl get pod nginx`, `kubectl get deployment my-app`, etc).

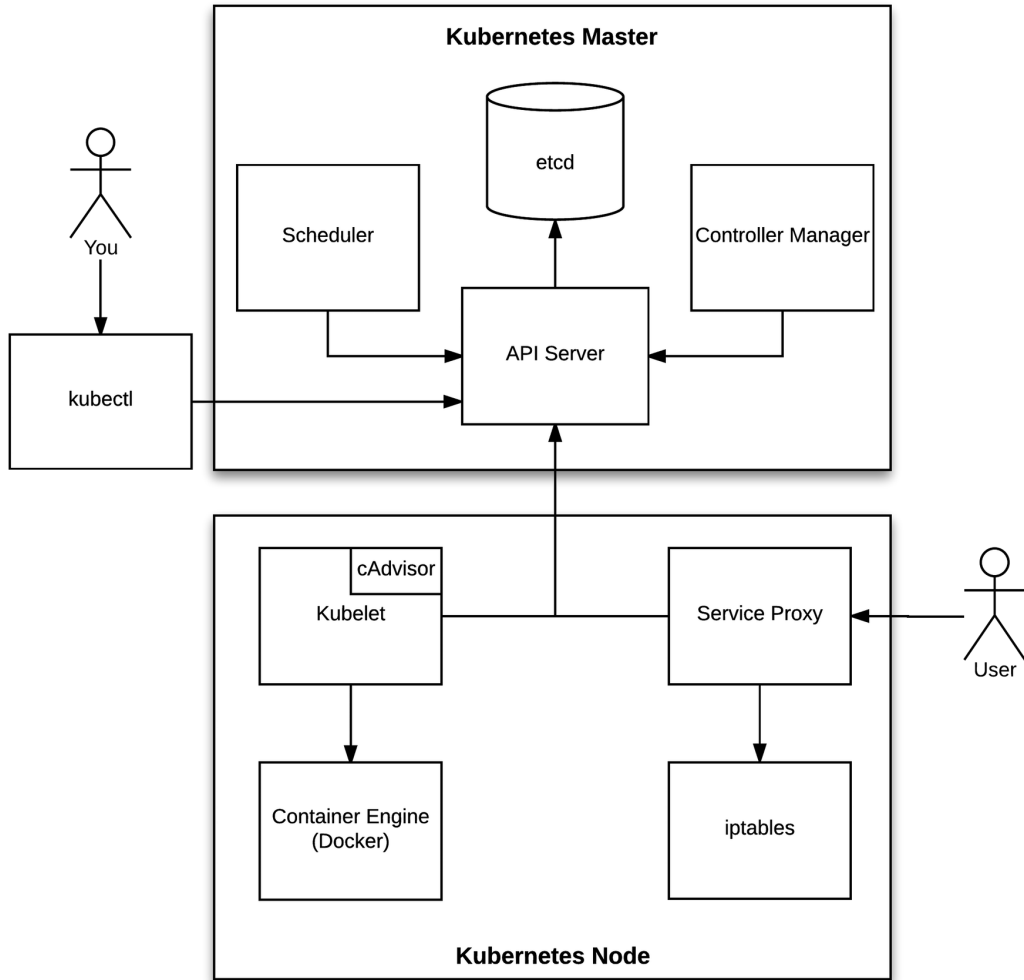


`kubectl` is a powerful tool and it provides a lot of options for each command. Please use the `kubectl help` to get detailed information.

# Kubernetes Architecture

If you have followed me in the last sections, you should have a good understanding on how to deploy containerized applications on Kubernetes. Although this may be enough for you to move on to the hands-on project of this book, this chapter is intended for those who are more curious and like to understand how things work behind the scenes.

Let's take a look at the following Kubernetes architecture diagram:



Kubernetes Architecture Diagram

Well, this might look complicated at first glance, but actually it's not so much. I'm pretty sure that by the end of this chapter you will have a clear understanding of how these components interact with each other.

# Kubernetes Master Components

As mentioned before, the master runs the Scheduler, Controller Manager, API Server and etcd components and is responsible for managing the Kubernetes cluster. Essentially, it's the brain of the cluster! Now, let's dive into each master component.



In production, you should set up Kubernetes with multiple masters for high availability. See how to [build high-Availability clusters](#)<sup>63</sup> official guides for further information.

## Etcd

[Etcd](#)<sup>64</sup> is a distributed, consistent key-value store used for configuration management, service discovery, and coordinating distributed work.

When it comes to Kubernetes, etcd reliably stores the configuration data of the Kubernetes cluster, representing the state of the cluster (what nodes exist in the cluster, what pods should be running, which nodes they are running on, and a whole lot more) at any given point of time.



As all cluster data is stored in etcd, you should always have a backup plan for it. You can easily back up your etcd data using the `etcdctl snapshot save` command. In case you are running Kubernetes on AWS, you can also back up etcd by taking a snapshot of the EBS volume.

Etcd is written in Go and uses the Raft consensus algorithm to manage a highly-available replicated log.

---

<sup>63</sup><https://kubernetes.io/docs/admin/high-availability/>

<sup>64</sup><https://github.com/coreos/etcd>



Raft is a consensus algorithm designed as an alternative to [Paxos](#)<sup>65</sup>. The Consensus problem involves multiple servers agreeing on values; a common problem that arises in the context of replicated state machines. Raft defines three different roles (Leader, Follower, and Candidate) and achieves consensus via an elected leader. For further information, please read the [Raft paper](#)<sup>66</sup>.

[Etcctl](#)<sup>67</sup> is the command-line interface tool written in Go that allows manipulating an etcd cluster. It can be used to perform a variety of actions, such as:

- Set, update and remove keys.
- Verify the cluster health.
- Add or remove etcd nodes.
- Generating database snapshots.



You can play online with a 5-node etcd cluster at <http://play.etcd.io><sup>68</sup>.

Etcd also implements a watch feature, which provides an event-based interface for asynchronously monitoring changes to keys. Once a key is changed, its “watchers” get notified. This is a crucial feature in the context of Kubernetes, as the API Server component heavily relies on this to get notified and call the appropriate business logic components to move the current state towards the desired state.

## API Server

When you interact with your Kubernetes cluster using the `kubectl` command-line interface, you are actually communicating with the master **API Server** component.

The API Server is the main management point of the entire cluster. In short, it processes REST operations, validates them, and updates the corresponding objects in

---

<sup>65</sup>[https://en.wikipedia.org/wiki/Paxos\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Paxos_(computer_science))

<sup>66</sup><https://raft.github.io/raft.pdf>

<sup>67</sup><https://github.com/coreos/etcd/tree/master/etcctl>

<sup>68</sup><http://play.etcd.io>

`etcd`. The API Server serves up the [Kubernetes API](#)<sup>69</sup> and is intended to be a relatively simple server, with most business logic implemented in separate components or in plugins.



The API Server is the only Kubernetes component that connects to `etcd`; all the other components must go through the API Server to work with the cluster state.

The API Server is also responsible for the authentication and authorization mechanism. All API clients should be authenticated in order to interact with the API Server. For example, the `kubectl` is working on your machine because it has been configured with the appropriate certificates when you executed the `vagrant up` command.

```
$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority: kubernetes-vagrant-coreos-cluster/artifacts/tls/ca.pem
    server: https://172.17.8.101
  name: default-cluster
contexts:
- context:
    cluster: default-cluster
    user: default-admin
  name: local
current-context: local
kind: Config
preferences: {}
users:
- name: default-admin
  user:
    client-certificate: kubernetes-vagrant-coreos-cluster/artifacts/tls/admin.pem
    client-key: kubernetes-vagrant-coreos-cluster/artifacts/tls/admin-key.pem
```

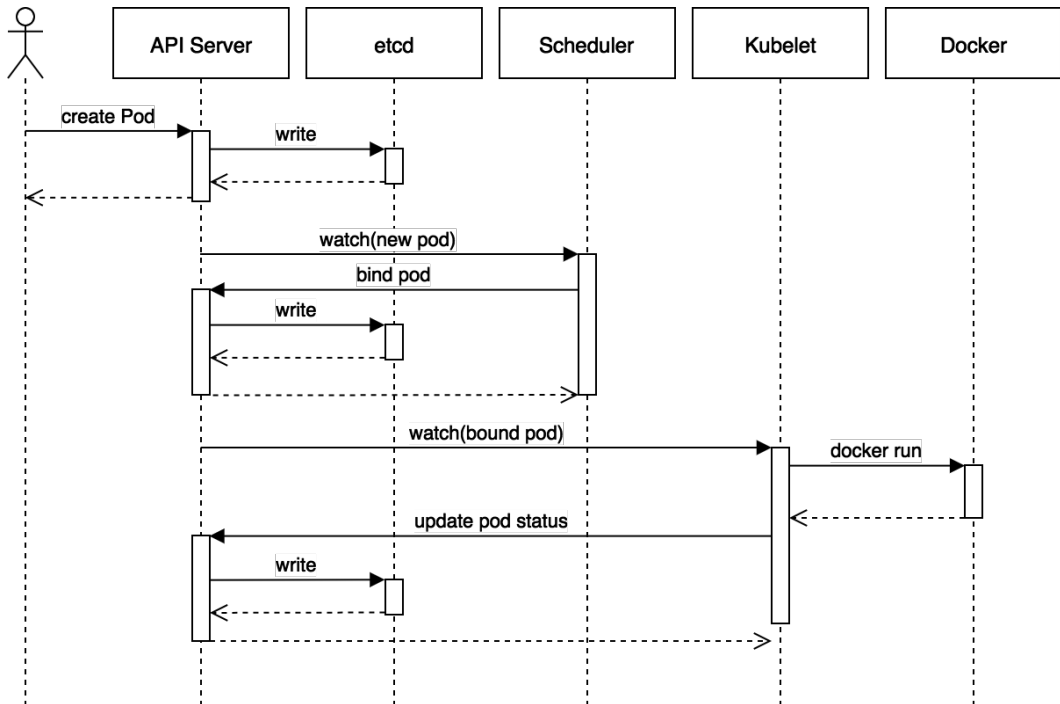
The API Server also implements a watch mechanism (similar to `etcd`) for clients to watch for changes. This allows components such as the Scheduler and Controller Manager to interact with the API Server in a loosely coupled manner.

---

<sup>69</sup><https://kubernetes.io/docs/concepts/overview/kubernetes-api/>



This pattern is extensively used in Kubernetes. For example, when you create a pod using `kubectl`, this what happens:



Create Pod Flow. Source: [heptio.com](https://heptio.com)

1. `kubectl` writes to the API Server.
2. API Server validates the request and persists it to etcd.
3. etcd notifies back the API Server.
4. API Server invokes the Scheduler.
5. Scheduler decides where to run the pod on and return that to the API Server.
6. API Server persists it to etcd.
7. etcd notifies back the API Server.
8. API Server invokes the Kubelet in the corresponding node.
9. Kubelet talks to the Docker daemon using the API over the Docker socket to create the container.
10. Kubelet updates the pod status to the API Server.
11. API Server persists the new state in etcd.

# Hands-on Project: Continuous Delivery Pipeline

Congratulations! After reading about 400 pages you finally got to the hands-on project; the fun part of the book. I'm going to guide you through the creation of a complete [continuous delivery pipeline](#) for a real-world case using all you have learned so far.

In order to make this chapter easier to understand and well-organized, it was divided into three parts: Configuration, Usage and Code.

- **Configuration:** set up Kubernetes, GitHub, Docker Hub, Artifactory, Jenkins, Pipeline Projects, and so on. By the way, as the Pipeline Projects are explained in next parts, they are **really** just focused on the **configurations**.
- **Usage:** review the complete development flow for a new user story and reproduce it using the Jenkins Pipeline Projects. Besides, describes the role of each Pipeline Project.
- **Code:** dive into each Jenkins Pipeline code focusing on technical implementation details and show the Kubernetes resources that make up the testing, staging and production environments.

For example, in both **Usage** and **Code** parts you will find the `Generating Release Version 1.1.0` section, but with different approaches. In the first, you will interact with Jenkins Pipeline Projects, create a pull request and merge it on GitHub, etc, whereas in the last you will understand the implementation details for the Jenkins Scripted Pipeline code that is executed behind the scenes to generate a release version.

## New User Story Development Flow

Before we start using our Jenkins Pipeline Projects, let's understand the complete flow for a new feature in the Notepad application.

For this example, consider a [Scrum](#) team that relies on [Extreme Programming](#) techniques to create high quality code, uses Git as the source control system, [Feature Branches](#) and is already running the Notepad application in production (version 1.0.0 stable).

Suppose that the new feature is to add a new field named `subtitle` to the [Note model](#) so that users can add not only the `title` and `content` of their notes but also the `subtitle`.



As you already know, everything starts with the product owner transforming this feature into a user story and adding it to his product backlog. If the user story is a high priority one, the product owner introduces it to the team in the next sprint planning so that it can be estimated and then broken into smaller tasks that last no more than a day to be coded (including automated tests) and integrated to the mainline (branch `master`). These tasks make up the spring backlog.

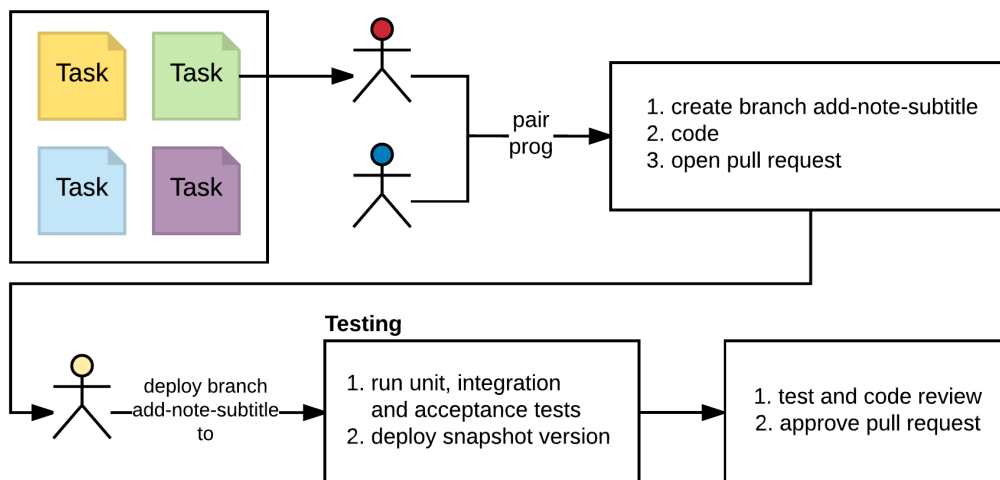
The product owner decide to transform this feature into a user story, add it to the highest priority of his product backlog, introduced it to the team in the sprint planning and it generates a single task.

Once the sprint starts, two developers pick up the task, create a new branch (say `add-note-subtitle`) from branch `master` and start a pair programming section committing and pushing their code to this branch. When the task is done, they create a [pull request on GitHub](#)<sup>70</sup> and go grab some coffee (isn't that true?) while other developer performs a code review and tests the branch `add-note-subtitle` on the **testing** environment, which is an immutable environment dynamically provisioned for each new test. When bootstrapping the testing environment, automated tests are automatically executed (unit, integration and acceptance tests), so the developer testing it can just focus on the code review and manual tests. If everything is fine, he

---

<sup>70</sup><https://help.github.com/articles/about-pull-requests/>

approves the pull request, which means the branch `add-note-subtitle` is now ready to be merged into `master`.



#### Before Merging `add-note-subtitle` Into `master`

After merging the branch `add-note-subtitle` into `master`, an automated build process compile the code on branch `master` and run the automated tests as part of the [Continuous Integration](#) process. Then, a release version `1.1.0` is generated (based on `master`, of course) and the application version is incremented according to [Semantic Versioning](#)<sup>71</sup>. The release version is deployed to the **staging** environment (which is pretty similar to production) where acceptance and performance tests are executed again.



The staging environment also allows the product owner and the QA team (if you have one) to verify that the release version works as intended before releasing it to end users without requiring a special deployment or access to a local developer machine.

If everything is fine, the release version is deployed to **production** using the [Canary Release](#)<sup>72</sup> deployment pattern, allowing just a small percentage of end users start

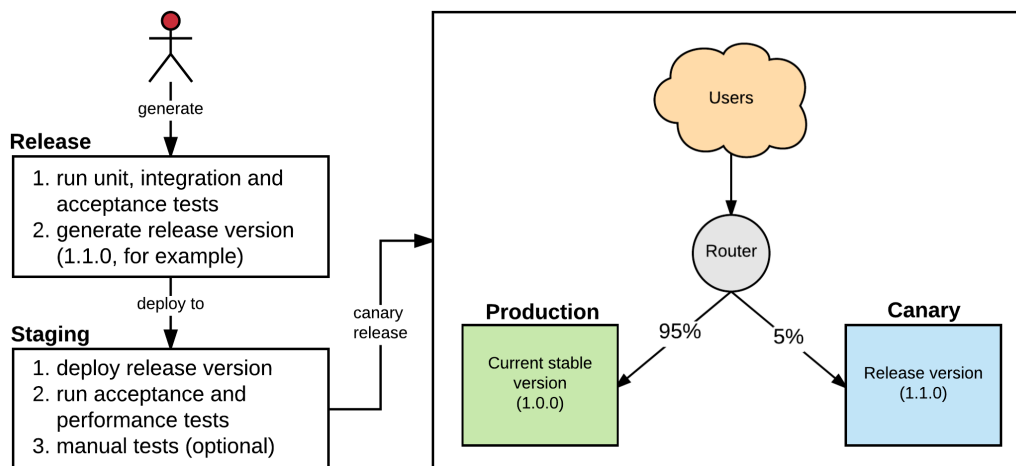
<sup>71</sup><http://semver.org/>

<sup>72</sup><https://martinfowler.com/bliki/CanaryRelease.html>

using it. Thus, the team has the opportunity to collect and analyze metrics and logs to ensure that the release version is actually working as intended in production too. Once the team feels confident, more traffic is gradually routed to the release version, until it eventually reaches all users and release version becomes the stable version.

There are a few things to point out about this user story development flow:

1. I used a user story as an example, but it could be a bug fix, a configuration, etc. The flow would still be the same.
2. The flow presented here is a default one and actually works pretty fine most of cases, but it doesn't mean that every development flow should follow these exact same steps all the time. A continuous delivery pipeline can (and probably will) be different for each project.



After Merging add-note-subtitle Into master

Now we are going to reproduce exactly the same flow using the Jenkins Pipeline Projects you have set up in the last section.