Carlos Pérez Sánchez, Pablo Solar Vilariño

PHP Microservices

Transit from monolithic architectures to highly available, scalable, and fault-tolerant microservices

www.allitebooks.com

PHP Microservices

Transit from monolithic architectures to highly available, scalable, and fault-tolerant microservices

Carlos Pérez Sánchez
Pablo Solar Vilariño

BIRMINGHAM - MUMBAI

www.allitebooks.com

PHP Microservices

Copyright © 2017 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: March 2017

Production reference: 1240317

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B32PB, UK.
ISBN 978-1-78712-537-7

www.packtpub.com

www.allitebooks.com

Credits

Authors

Carlos Pérez Sánchez
Pablo Solar Vilariño

Reviewers

Gabor Zelei

Commissioning Editor

Aaron Lazar

Acquisition Editor

Divya Poojari

Content Development Editor

Anurag Ghogre

Technical Editor

Jijo Maliyekal
Subhalaxmi Nadar

Copy Editor

Shaila Kusanale

Project Coordinator

Vaidehi Sawant

Proofreader

Safis Editing

Indexer

Mariammal Chettiyar

Graphics

Jason Monteiro

Production Coordinator

Melwyn Dsa

www.allitebooks.com

About the Authors

Carlos Pérez Sánchez is a backend web developer with more than 10 years of experience in working with the PHP language. He loves finding the best solution for every single problem on web applications and coding, looking to push forward the best practices for development, ensuring a high level of attention to detail.

He has a bachelors degree in computer engineering from the University of Alicante in Spain, and he has worked for different companies in the United Kingdom and Spain. He has also worked for American companies and is currently working for Pingvalue. You can connect with him on LinkedIn at `https://www.linkedin.com/in/mrcarlosdev`.

To my girlfriend, Becca, family, and friends—thanks for your unconditional support in every single stage of my life.

Pablo Solar Vilariño is a software developer who became interested in web development when PHP 4 started becoming a popular language.

Over the last few years, he has worked extensively with web, cloud, and mobile technologies for medium-to-large companies and is currently an e-commerce developer at NITSNETS.

He has a passion for new technologies, code standards, scalability, performance, and open source projects.

Pablo can be approached online at `https://pablosolar.es/`.

To everyone who has ever believed in me.

www.allitebooks.com

About the Reviewer

Gabor Zelei is a polyglot software engineer with a versatile background in both software engineering and operations. He's had a decade-long love for PHP and LAMP/LEMP stacks in general; he enjoys working with upcoming technologies and methodologies and is a big fan of clean, well-structured and standards-compliant code.

During his career, he has worked for small startups as well as large enterprise companies. Currently, he lives in Dublin, Ireland, where he works as a senior software developer for a leading online marketplace company.

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

https://www.packtpub.com/mapt

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at `https://www.amazon.com/dp/1787125378`.

If you'd like to join our team of regular reviewers, you can e-mail us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface 1

Chapter 1: What are Microservices? 6

- **Monolithic versus microservices** 7
- **Service Oriented Architectures versus microservices** 10
- **Microservices characteristics** 11
  - Successful cases 12
  - Disadvantages of microservices 13
- **How to focus your development on microservices** 14
  - Always create small logical black boxes 14
  - Network latency is your hidden enemy 15
  - Always think about scalability 15
  - Use a lightweight communication protocol 15
  - Use queues to reduce a service load or make async executions 15
  - Be ready for the worst case scenario 16
  - Each service is different, so keep different repositories and build environments 16
- **Advantages of using PHP on microservices** 16
  - A short history of PHP 17
  - PHP milestones 18
    - Version 4.x 18
    - Version 5.x 18
    - Version 6.x 18
    - Version 7.x 19
  - Advantages 19
  - Disadvantages 21
- **Summary** 22

Chapter 2: Development Environment 23

- **Design and architecture to build the basic platform for microservices** 24
- **Requirements to start working on microservices** 25
  - Docker installation 26
    - Installing Docker on macOS 26
      - Docker for Mac (alias, native implementation) versus Docker toolbox 27
      - Minimum requirements 27
      - Docker for Mac installation process 27
    - Installing Docker on Linux 28

CentOS/RHEL 29
Minimum requirements 29
Installing Docker using yum 29
Post-install setup – creating a Docker group 30
Installing Docker on Ubuntu 30
Minimum requirements 31
Installing Docker using apt 31
Common issues on Ubuntu 32
UFW forwarding 33
DNS server 33
Post-install setup – creating a Docker group 34
Starting Docker on boot 34
Installing Docker on Windows 35
Minimum requirements 35
Installing the Docker tools 35
How to check your Docker engine, compose, and machine versions 37
Quick example to check your Docker installation 37
Common management tasks 38
Version control – Git versus SVN 39
Git 39
Hosting 42
GitHub 42
BitBucket 43
Version control strategies 43
Centralized work 43
Feature branch workflow 43
Gitflow workflow 44
Forking workflow 44
Semantic versioning 45
Setting up a development environment for microservices 45
Autodiscovery service 47
Microservice base core – NGINX and PHP-FPM 48
Frameworks for microservices 56
PHP-FIG 56
PSR-7 57
Messages 58
Headers 58
Host header 59
Streams 59
Request targets and URIs 59
Server-side requests 60
Uploaded files 60
Middleware 61
Available frameworks 65
Phalcon 65

Slim framework 66
Lumen 66
Zend Expressive 67
Silex (based on Symfony) 67
Summary 68
Chapter 3: Application Design 69
Microservices structure 69
Microservice patterns 71
API gateway 71
Service discovery and registry 72
Shared database or database per service 73
Database per service 73
Shared database 74
RESTful conventions 74
Security 75
Standards 75
Consumer amenities 76
Caching strategy 77
General caching strategy 77
HTTP caching 80
Static files caching 82
Domain-driven design 83
How domain-driven design works 83
Using domain-driver design in microservices 86
Event-driven architecture 87
Event-driven architecture in microservices 88
Continuous integration, continuous delivery, and tools 91
Continuous integration – CI 91
What is continous integration? 91
Benefits of CI 92
Tools for continuous integration 92
Continuous delivery 93
Benefits of continuous delivery 94
Tools for a continuous delivery pipeline 95
Summary 95
Chapter 4: Testing and Quality Control 96
The importance of using tests in your application 96
Testing in microservices 97
Test-driven development 98
How to do TDD? 99

Why should I use TDD? 100
TDD algorithm 102
Red – writing the unit tests 102
Green – make the code work 103
Refactor – eliminate redundancy 103
Behavior-driven development 104
What is BDD? 104
How does it work? 104
Cucumber as DSL for BDD 104
Acceptance test-driven development 106
User stories 107
ATDD algorithm 108
Discuss 108
Distill 109
Develop 109
Demo 109
Tools 110
Composer 110
PHPUnit 110
Unit testing 112
Running the tests 113
Assertions 114
assertArrayHasKey 114
assertClassHasAttribute 115
assertArraySubset 116
assertClassHasStaticAttribute 116
assertContains() 117
assertDirectory() and assertFile() 117
assertString() 118
assertRegExp() 119
assertJson() 119
Boolean assertions 120
Type assertions 120
Other assertions 120
Unit testing from scratch 121
Behat 128
Installation 128
Test execution 128
Behat example from scratch 129
Selenium 132
Selenium WebDriver 133
Selenium IDE 133
Summary 133
Chapter 5: Microservices Development 134

Dependency management 134
Routing 136
Postman 139
Middleware 141
Implementing a microservice call 142
Request life cycle 145
Communication between microservices with Guzzle 146
Database operations 149
Error handling 158
Validation 158
Manage exceptions 161
Async and queue 164
Caching 167
Summary 171
Chapter 6: Monitoring 172
Debugging and profiling 172
What is debugging? 172
What is profiling? 173
Debugging and profiling in PHP with Xdebug 173
Debugging installation 173
Debugging setup 175
Debugging the output 176
Profiling installation 177
Profiling setup 178
Analyzing the output file 178
Error handling 179
What is error handling? 179
Why is error handling important? 179
Challenges when managing error handling with microservices 180
Basic die() function 180
Custom error handling 181
Report method 182
Render method 182
Error handling with Sentry 183
Application logs 188
Challenges in microservices 189
Logs in Lumen 190
Application monitoring 191
Monitoring by levels 192
Application level 192
Datadog 193

Infrastructure level 194
Prometheus 195
Weave Scope 198
Hardware/hypervisor monitoring 200
Summary 200
Chapter 7: Security 201
Encryption in microservices 201
Database encryption 202
Encryption in MariaDB 203
InnoDB encryption 205
Performance overhead 206
TSL/SSL protocols 206
How the TSL/SSL protocol works 207
TSL/SSL termination 208
TSL/SSL with NGINX 209
Authentication 211
OAuth 2 215
How to use OAuth 2 on Lumen 215
OAuth 2 installation 215
Setup 216
Let's try OAuth2 217
JSON Web Token 219
How to use JWT on Lumen 219
Setting up JWT 220
Let's try JWT 224
Access Control List 225
What is ACL? 225
How to use ACL 226
Security of the source code 228
Environment variables 228
External services 229
Tracking and monitoring 229
Best practices 230
File permissions and ownership 230
PHP execution locations 231
Never trust users 231
SQL injection 231
Cross-site scripting XSS 231
Session hijacking 231
Remote files 232
Password storage 232

Password policies 232
Source code revelation 233
Directory traversal 233
Summary 233
Chapter 8: Deployment 234
Dependency management 234
Composer require-dev 234
The .gitignore file 235
Vendor folder 235
Deployment workflows 235
Vendor folder on repository 235
Composer in production 236
Frontend dependencies 236
Grunt 236
Gulp 238
SASS 238
Bower 239
Deploy automation 240
Simple PHP script 240
Ansible and Ansistrano 241
Ansible requirements 242
Ansible installation 242
What is Ansistrano? 243
How does Ansistrano work? 243
Deploying with Ansistrano 245
Other deployment tools 247
Advanced deployment techniques 248
Continuous integration with Jenkins 248
Blue/Green deployment 249
Canary releases 250
Immutable infrastructure 251
Backup strategies 252
What is backup? 252
Why is it important? 252
What and where we need to back up 253
Backup types 253
Full backup 253
Incremental and differential backups 254
Backup tools 254
Bacula 254
Percona xtrabackup 255
Custom scripts 256

Validating backups 256
Be ready for the apocalypse 256
Summary 257
Chapter 9: From Monolithic to Microservices 258
Refactor strategies 258
Stop diving 258
Divide frontend and backend 260
Extraction services 262
How to extract a module 262
Tutorial: From monolithic to microservices 263
Stop diving 264
Divide frontend and backend 270
Extraction services 275
Summary 284
Chapter 10: Strategies for Scalability 285
Capacity planning 285
Knowing the limits of your application 286
Availability math 289
Load testing 292
Apache JMeter 292
Installing Apache JMeter 293
Executing load tests with Apache JMeter 294
Load testing with Artillery 302
Installing Artillery 302
Executing loading tests with Artillery 303
Creating Artillery scripts 305
Advanced scripting 306
Load testing with siege 307
Installing siege on RHEL, CentOS, and similar operating systems 307
Installing siege on Debian or Ubuntu 308
Installing siege on other OS 308
Quick siege example 308
Scalability plan 309
Step #0 311
Step #1 311
Step #2 312
Step #3 312
Step #4 313
Step #5 313
Summary 313

Chapter 11: Best Practices and Conventions 314
Code versioning best practices 314
Caching best practices 315
Performance impact 315
Handle cache misses 315
Group requests 316
Size of elements to be stored in cache 316
Monitor your cache 316
Choose your cache algorithm carefully 316
Performance best practices 317
Minimize HTTP requests 317
Minimize HTML, CSS, and JavaScript 318
Image optimization 319
Use sprites 319
Use lossless image compression 319
Scale your images 320
Use data URIs 320
Cache, cache, and more cache 321
Avoid bad requests 322
Use Content Delivery Networks (CDNs) 322
Dependency management 322
Semantic versioning 323
How semantic versioning works 324
Semantic versioning in action 324
We have been told to add a new feature to the project 324
We have been told that there is a bug in our project 324
We have been asked for a big change 325
Error handling 325
Client request successful 325
Request redirected 326
Client request incomplete 326
Server errors 327
Coding practices 327
Dealing with strings 327
Single quotes versus double quotes 328
Spaces versus tabs 328
Regular expressions 328
Connection and queries to a database 329
Using the === operator 329
Working with release branches 329
Quick example 330

Summary 331
Chapter 12: Cloud and DevOps 332
What is Cloud? 332
Autoscalable and elastic 333
Lower management efforts 334
Cheaper 334
Grow faster 334
Time to market 335
Select your Cloud provider 335
Amazon Web Services (AWS) 335
Microsoft Azure 336
Rackspace 337
DigitalOcean 337
Joyent 338
Google Compute Engine 338
Deploying your application to the Cloud 339
Docker Swarm 339
Installing Docker Swarm 340
Adding services to our Swarm 343
Scaling your services in Swarm 346
Apache Mesos and DC/OS 347
Kubernetes 347
Deploying to Joyent Triton 349
What is DevOps? 350
Summary 352
Index 353

Preface

Microservices are the new kids on the block. They are a way of solving problems, but it does not mean that they will always be the best solution. Microservices are a full and complex architecture that you need to understand deeply in order to be able to apply them successfully in the real world.

We know that understanding and having everything in place for microservices can take time and can be very complex, so for this reason we wrote this book--to give you a small guide to microservices, from A to Z. This book will only be a glimpse of what you can do with microservices, but we hope that it will be at least a helpful start into this world.

We will start our journey in this book by explaining the foundations of this architecture, the basic knowledge you will need to have. As you advance through the book, we will be increasing the level of difficulty. However, we will guide you through every step of the process, so at the end of the book you will be able to know whether a microservice architecture is the best solution for your problem and how can you apply this architecture.

Enjoy your microservices!

What this book covers

`Chapter 1`, *What are Microservices?*, will teach you all the basics of microservices.

`Chapter 2`, *Development Environment*, will take you through setting up your development machine to build microservices successfully.

`Chapter 3`, *Application Design*, will help you start designing your application, creating the foundation of your project.

`Chapter 4`, *Testing and Quality Control*, looks at how important testing your application is and how can you add tests to your application.

`Chapter 5`, *Microservices Development*, will cover building a microservices application, explaining each step involved.

`Chapter 6`, *Monitoring*, covers how to monitor your application so that you can always know how your application is behaving.

`Chapter 7`, *Security*, focuses on how you can add an extra layer of complexity to your application to make it more secure.

`Chapter 8`, *Deployment*, explains how you can deploy your application successfully.

`Chapter 9`, *From Monolithic to Microservices*, discusses an example of how a monolithic application can be transformed into a microservice.

`Chapter 10`, *Strategies for Scalability*, outlines how you can create a scalable application.

`Chapter 11`, *Best Practices and Conventions*, will refresh your knowledge about the best practices and conventions you should use in an application.

`Chapter 12`, *Cloud and DevOps*, looks at the different cloud providers and the DevOps world.

What you need for this book

To follow along with this book, you'll need a computer with an Internet connection to download the required software and Docker images. At some point, you will need at least a text editor or an IDE, and we highly recommend PHPStorm for this job.

Who this book is for

This book is for experienced PHP developers who want to go a step ahead in their careers and start building successful scalable and maintainable applications, all based on microservices. After reading the book, they will be able to know whether microservices are the best solutions to their problems and if it is the case, create a successful application.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The next lines of code read the link and assign it to the to the `BeautifulSoup` function."

A block of code is set as follows:

```
version: '2'
services:
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
version: '2'
services:
```

Any command-line input or output is written as follows:

```
$ uname -r
3.10.0-229.el7.x86_64
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Click on the toolbar whale for **Preferences...** and other options."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support`and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/PHP-Microservices`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support`and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
What are Microservices?

Good projects need good solutions; this is why developers are always looking for better ways to do their jobs. There is no best solution for all projects because every single project has different needs and the architect (or the developer) has to find the best solution for that specific project.

Microservices are maybe a good approach to solve problems; in the last few years, companies such as Netflix, PayPal, eBay, Amazon, and Spotify have chosen to use microservices in their own development teams because they believed them to be the best solution for their projects. To understand why they chose microservices and understand the kinds of projects you should use them in, it is necessary to know what a microservice is.

Firstly, it is essential to understand what a monolithic application is, but basically, we can define a microservice as an extended Service Oriented Architecture. In other words, it is a way to develop an application by following the required steps to turn it into various little services. Each service will execute itself and communicate with others through requests, usually using APIs on HTTP.

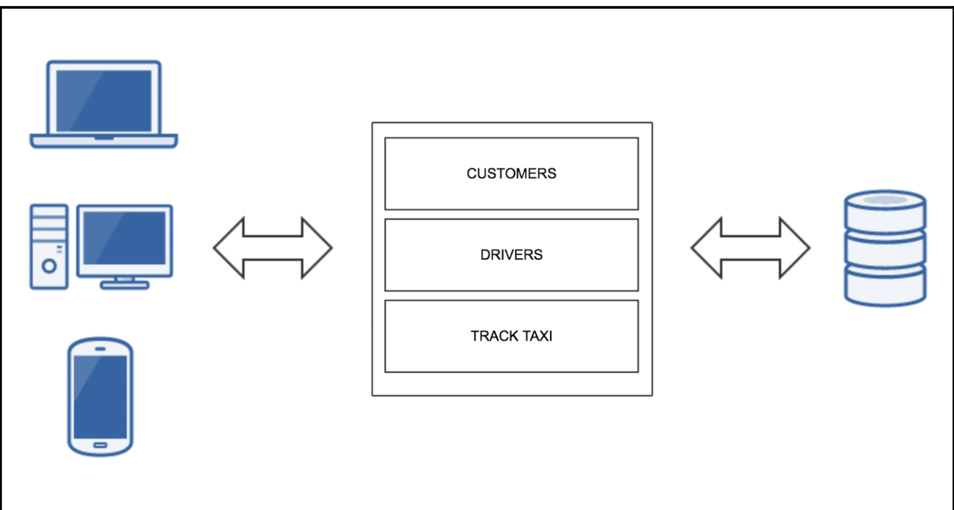
To further understand what microservices are, we first need to understand what a monolithic application is. It is the typical application that we have been developing for the last few years, for example in PHP, using a framework like Symfony; in other words, all the applications we have been developing are divided into different parts, such as frontend, backend, and database, and also use the **Model-View-Controller** (**MVC**) pattern. It is important to differentiate between MVC and microservices. MVC is a design pattern and microservices are a way to develop an application; therefore, applications developed using MVC could still be monolithic applications. People may think that if we split our application into different machines and divide the business logic from the model and the view, the application is then based on microservices, but this is not correct.

However, using a monolithic architecture still has its advantages. There are also various huge web applications, such as Facebook, that use it; we just need to know when we need to use a monolithic architecture and when we need to use microservices.

Monolithic versus microservices

Now, we will discuss the advantages and disadvantages of using monolithic applications and how microservices improve a specific project by giving a basic example.

Imagine a taxi platform like Uber; the platform in this example will be a small one with only the basic things in order to understand the definitions. There are customers, drivers, cities, and a system to track the location of the taxi in real time:



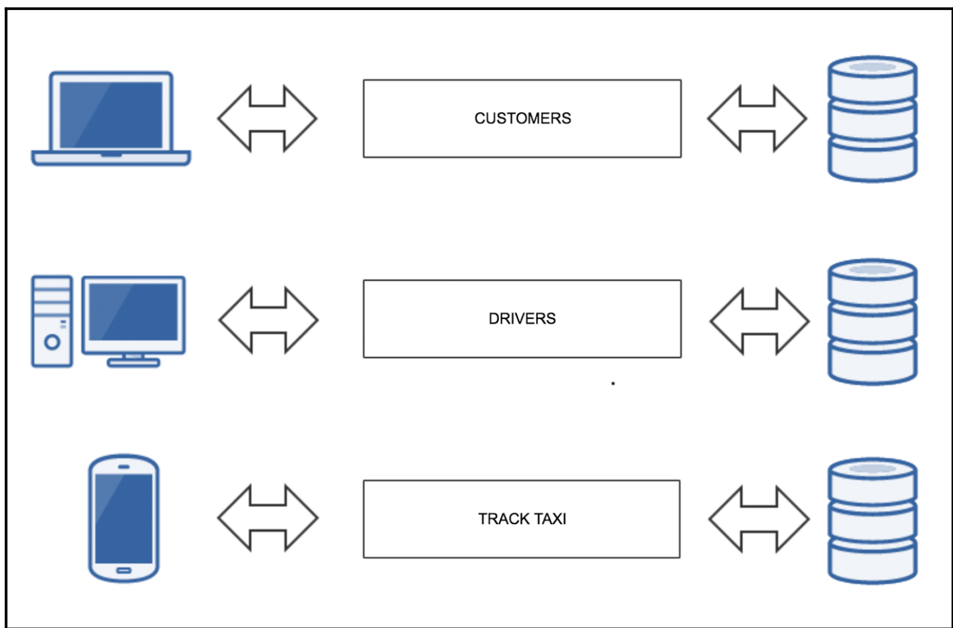
In a monolithic system, we have all this together—we have a common database with the customers and drivers linked to cities and all these are linked to the system to track taxis using foreign keys.

All this can also be hosted on different machines using master-slave databases; the backend can be in different instances using a load balancer or a reverse proxy and the frontend can use a different technology using Node.js or even plain HTML. Even so, the platform will be a monolithic application.

Lets see an example of possible problems faced in a monolithic application:

- Two developers, Joe and John, are working on the basic taxi example project; Joe needs to change some code on drivers and John has to change some code on customers. The first problem is that both the developers are working on the same base code; this is not an exclusive monolithic problem but if Joe needs to add a new field on drivers, he may need to change the customer's model too, so Joe's work does not finish on the driver's side; in other words, his work is not delimited. This is what happens when we use monolithic applications.
- Joe and John have realized that the system to track taxis has to communicate with third parties calling external APIs. The application does not have a very big load but the system to track taxis has a lot of requests from customers and drivers, so there is a bottleneck on that side. Joe and John have to scale it to solve the problem on the tracking system: they can get faster machines, more memory, and a load balancer, but the problem is that they have to scale the entire application.
- Joe and John are happy; they just finished fixing the problem on the system to track taxis. Now they will put the changes on the production server. They will have to work tonight when the traffic on the web application is lower; the risk is high because they have to deploy the entire application and not just the system to track taxis that they fixed.
- In a couple of hours, an error 500 appears within the application. Joe and John know that the problem is related to the tracking system, but the entire application will be down only because there is a problem with a part of the application.

A microservice is a simple, isolated entity with a concrete proposal. It is independent and works with the rest of the microservices by communicating through an agreed channel as you can see in the next picture:



For developers who are used to working on object-oriented programming, the idea of a microservice would be something like an encapsulated object working on a different machine and isolated from the other ones working on different machines too.

Following the same example as before, if we have a problem on the system to track taxis, it would be necessary to isolate all the code related to this part of the application. This is a little complex and will be explained in detail in `Chapter 9`, *From Monolithic to Microservices*, but in general terms, it is a database used exclusively by the system to track taxis, so we need to extract the part for this purpose and the code needs to be modified to work with the extracted database. Once the goal is achieved, we will have a microservice with an API (or any other channel) that can be called by the rest of the monolithic application.

This will avoid the problems mentioned before—Joe and John can work on their own issue because once the application is divided into microservices, they will work on the customer or driver microservice. If Joe has to change code or include a new field, he will only need to change it in his own entity and John will consume the drivers API to communicate with it from the customer's part.

The scalability can be done just for this microservice, so it is not necessary to scale the entire application by spending money and resources and if the system to track taxis is down, the rest of the application will work without any problems.

Another advantage of using microservices is that they are agnostic to the language; in other words, it is possible to use different languages for each microservice. A microservice written in PHP can talk to others written in Python or Ruby because they only give the API to the rest of the microservices, so they just have to share the same interface to communicate with each other.

Service Oriented Architectures versus microservices

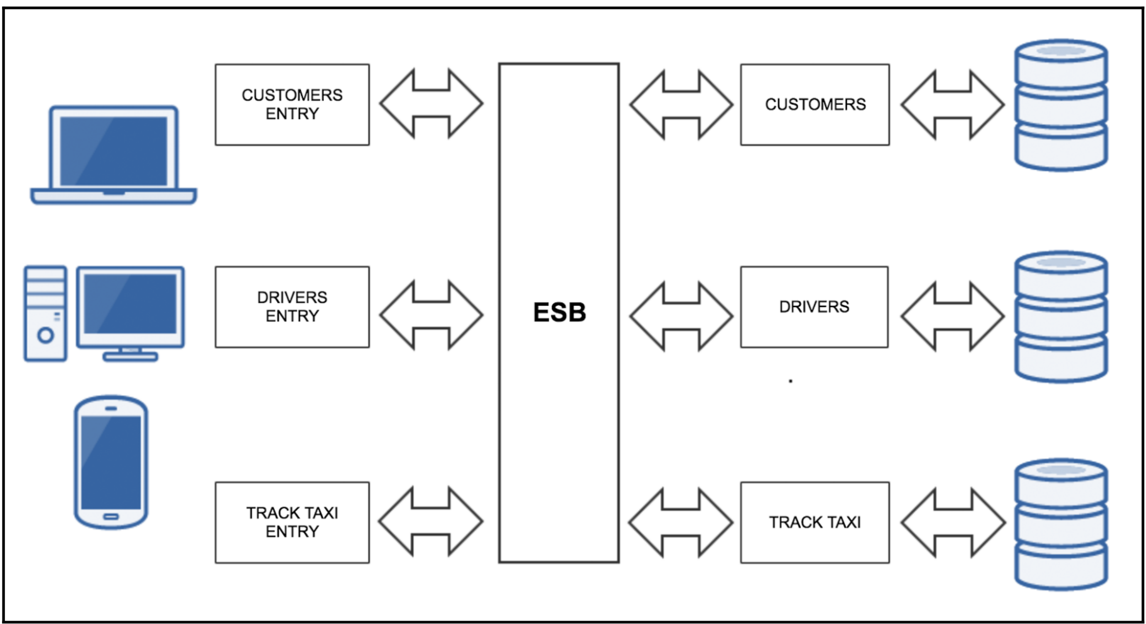
When a developer encounters microservices and they know the **Service Oriented Architecture** (**SOA**) style of software design, the first question they ask themselves is whether SOA and microservices are the same thing or whether they are related, and the answer is a little bit controversial; depending on who you ask, the answer will be different.

According to Martin Fowler, SOA focuses on integrating monolithic applications between themselves, and it uses an **Enterprise Service Bus** (**ESB**) to achieve this.

When the SOA architectures began to appear, these ones tried to connect different components between themselves, and this is one of the characteristics of microservices, but the problem with SOA was that it needed many things surrounding the architecture to work properly, such as ESB, **Business process management** (**BPM**), service repositories, register, and more things, so it made it more difficult to develop. Also, in order to change some parts of a code, it was necessary to agree with the other development teams before doing it.

All these things made the maintenance and code evolution difficult, and the time to market long; in other words, this architecture was not the best for applications that often needed to make changes live.

There are other opinions regarding SOA. Some say that SOA and microservices are the same, but SOA is the theory and microservices is a good implementation. The need to use an ESB or communicate using WSDL or WADL was not a must, but it was defined as the SOA standard. As you can see on the next picture, your architecture using SOA and ESB will look like this:



The requests arrive via different ways; this is the same way microservices work, but all the requests reach the ESB and it knows where it should call to get the data.

Microservices characteristics

Next, we will look at the key elements of a microservice architecture:

- **Ready to fail**: Microservices are designed to fail. In a web application, microservices communicate with each other and if one of them fails, the rest should work to avoid a cascading failure. Basically, microservices attempt to avoid communicating synchronously using async calls, queues, systems based on actors, and streams instead. This topic will be discussed in the subsequent chapters.

- **Unix philosophy**: Microservices should follow the Unix philosophy. Each microservice must be designed to do one thing only, and should only be small and independent. This allows us as developers to adjust and deploy each microservice independently.
  The Unix philosophy emphasizes building simple, short, clear, modular, and extensible code that can be easily maintained and repurposed by developers as well, in addition to its creators.

- **Communication layer**: Each microservice communicates with the others through HTTP requests and messages, executing the business logic, querying the database, exchanging messages with the required systems and, at the end, returning a JSON (or HTML/XML) response.
- **Scalability**: The main reason to choose a microservice architecture is that it is possible to scale the application easily. The bigger an application is and the more traffic the application has, the more sense the proper selection of choosing microservices makes. A microservice can scale the required part without any impact on the rest of the application.

Successful cases

The best way to understand how important microservices are in real life is knowing some platforms that decided to evolve and use microservices thinking about the future and making the maintenance and scalability easier, faster, and more effective:

- **Netflix**: The number one application for online video streaming turned its architecture into microservices a few years ago. There is a story about the reason they decided to use microservices. Making changes on a review module, a developer forgot to put the `;` at the end of the line and Netflix was down for many hours. The platform gets around 30% of the total traffic from the USA every day, so Netflix has to offer a stable service to its customers who pay every month. To get this, Netflix makes five requests to its different servers for each request that we make and it can get requests from 800 different devices using its streaming video API.
- **eBay**: In 2007, eBay decided to change its architecture to microservices; it used a monolithic application on C++ and Perl, later they moved to services built on Java, and finally they implemented their architecture using microservices. Its main application has many services and each one executes its own logic to be used by the customers in each area.

- **Uber**: Microservices allowed this company to grow quickly because it allowed it to use different languages (Node.js, Go, Scala, Java, or Python) for each microservice and the process of hiring engineers was easier because they were not limited by the language code.
- **Amazon**: Maybe Amazon is not the king of Internet traffic, but it moved to microservices a few years ago, being one of the first ones to use live microservices. The engineers said that it was not possible to provide all the services they offer, such as the web service, using the old monolithic application.
- **Spotify**: The need to be faster than its opponent was a must for Spotify. The main engineer, Niklas Gustavsson, said that being fast, automating everything, and having smaller teams of development is really important for the application. This is the reason Spotify uses microservices.

Disadvantages of microservices

Next, we will look at the disadvantages of the microservices architecture. When asking developers about this, they agree that the major problem with microservices is the debugging on the production server.

Debugging an application based on microservices can be a little tedious when you have hundreds of microservices in your application and you have to find where a problem is; you need to spend time looking for the microservice that is giving you the error. Each microservice works like an individual machine, so to look at a specific log you have to go to the specific microservice. Fortunately, there are some tools to help us out with this subject by getting the logs from all the different microservices in your application and putting them together into a single location. In the subsequent chapters, we will look at these kinds of tools.

Another disadvantage is that it is necessary to maintain every single microservice as an entire server; in other words, every single microservice can have one or more databases, logs, different services, or library versions, and even the code can be in a different language, so if it is difficult to maintain a single server and doing it with hundreds will waste money and time.

Also, the communication between microservices is very important—they have to work like clocks, so communication is essential for the application. To do this, the communication between the development teams will be necessary to tell each other what they need and also to write good documentation for each microservice.

A good practice to work with microservices is having everything automatized or at least everything possible. Maybe the most important part is the deployment. If it is necessary to deploy hundreds of microservices, it can be difficult. So, the best way is to automatize these kinds of tasks. We will look at how to do this in the subsequent chapters.

How to focus your development on microservices

Developing microservices is a new way of thinking. Therefore, it can be difficult when you encounter how the application will be designed and built for the first time. However, if you always remember that the most important idea behind microservices is the need to decompose your application into smaller logical pieces, you are already halfway there.

Once you understand this, the following core ideas will help you with the design and build process of your application.

Always create small logical black boxes

As a developer, you always start with the big picture of what you will build. Try to decompose the big picture into small logical blocks that only do one thing. Once the multiple small pieces are ready, you can start building complex systems, ensuring that the foundations of your application are solid.

Each of your microservices is like a black box with a public interface, which is the only way to interact with your software. The main recommendation you need to have always in mind is to build a very stable API. You can change the implementation of an API call without many problems, but if you change the way of calling or even the response of this call, you will be in big trouble. In the case of deep changes to your API, ensure that you use some kind of versioning so that you can support the old and new versions.

Network latency is your hidden enemy

The communication between services is made through API calls using the network as a connection pipe. This message exchange takes some time and it will not always be the same due to multiple factors. Imagine that you have *service_a* on one machine and a *service_b* on a different machine. Do you think that the network latency will always be the same? What happens if, for example, one of the servers is under a high load and takes some time to process requests? To reduce time, always keep an eye on your infrastructure, monitor everything, and use compression if it is available.

Always think about scalability

One of the main advantages of a microservice application is that each service can be scaled up or down. This flexibility can be achieved by reducing the number of stateful services to the minimum. A stateful service relies on data persistence, making it difficult to move or share the data without having data consistency problems.

Using autodiscovery and autoregistry techniques, you can build a system that knows which one will deal with each request at all times.

Use a lightweight communication protocol

Nobody likes to wait, not even your microservices. Don't try to reinvent the wheel or use an obscure but cool communication protocol, use HTTP and REST. They are known by all web developers, they are fast, reliable, easy to implement, and very easy to debug. If you need to increase the security, implement SSL/TSL.

Use queues to reduce a service load or make async executions

As a developer, you want to make your system as fast as possible. Therefore, it makes no sense to increase the execution time of an API call just because it is waiting for some action that can be done in the background. In these cases, the best approach is the use of queues and job runners in charge of this background processing.

Imagine that you have a notification system that sends an e-mail to a customer when placing an order on your microservice e-commerce. Do you think that the customer wants to wait to see the payment successful page only because the system is trying to send an e-mail? In this case, a better approach is to enqueue the message so that the customer will have a pretty instant thank you page. Later, a job runner will pick up the queued notification and the e-mail will be sent to the customer.

Be ready for the worst case scenario

You have a nice, new, good-looking site built on top of microservices. Are you ready for the moment when everything goes wrong? If you are suffering a network partition, do you know if your application will recover from that situation? Now, imagine that you have a recommendation system and it is down, are you going to give the customers a *default* recommendation while you try to recover the dead service? Welcome to the world of distributed systems where when something goes wrong, it can get worse. Always keep this in mind and try to be ready for any scenario.

Each service is different, so keep different repositories and build environments

We are decomposing an application into small parts which we want to scale and deploy independently, so it makes sense to keep the source in different repositories. Having different build environments and repositories means that each microservice has its own lifecycle and can be deployed without affecting the rest of your application.

In the subsequent chapters, we will take a deeper look at all these ideas and how to implement them using different driven developments.

Advantages of using PHP on microservices

To understand why PHP is a suitable programming language for building microservices, we need to take a small peek into its history, where it comes from, which problems it was trying to solve, and the evolution of the language.

A short history of PHP

In 1994, Rasmus Lerdorf created what we can say was the first version of PHP. He built a small suite of **Common Gateway Interfaces** (**CGIs**) in the C programming language to maintain his personal web page. This suite of scripts was called *Personal Home Page Tools*, but it was more commonly referenced as *PHP Tools*.

Time passed and Rasmus rewrote and extended the suite so that it could work with web forms and have the ability to communicate with databases. This new implementation was called *Personal Home Page/Forms Interpreter* or *PHP/FI* and served as a framework upon which other developers could build dynamic web applications. In June 1995, the source code was opened to the public under the name of *Personal Home Page Tools* (*PHP Tools*) *version 1.0*, allowing developers from all over the world to use it, fix bugs and improve the suite.

The first idea around PHP/FI was not to create a new programming language, and Lerdorf let it grow organically, leading to some problems like the inconsistency of function names or their parameters. Sometimes the function names were the same as the low-level libraries that PHP was using.

In October 1995, Rasmus released a new rewrite of the code; this was the first release that was considered as an advanced scripting interface and PHP started to become the programming language that it is today.

As a language, PHP was designed to be very similar to C in structure so that it would be easier to be adopted by developers who were familiar with C, Perl, or similar languages. Along with the growth of the features of the language, the number of early adopters also began to grow. A Netcraft survey of May, 1998 indicated that nearly 60,000 domains had headers containing PHP (around 1% of the domains on the Internet at the time), which indicated that the hosting server had it installed.

One important point in PHP history was when Andi Gutmans and Zeev Suraski of Tel Aviv, Israel, joined the project in 1997. At this time, they did another complete rewrite of the parser and started the development of a new and independent programming language. This new language was named PHP, with the meaning becoming a recursive acronym—**PHP** (**Hypertext Preprocessor**).

The official release of PHP 3 was in June 1998, including a great number of features that made the language suitable for all kinds of projects. Some of the features included were a mature interface for multiple databases, support for multiple protocols and APIs, and the ease of extending the language itself. Among all the features, the most important ones were the inclusion of object-oriented programming and a more powerful and consistent language syntax.

Andi Gutmans and Zeev Suraski founded Zend Technologies and started the rewrite of PHP's core, creating the Zend Engine in 1999. Zend Technologies became the most important PHP company and the main contributor to the source code.

This was only the beginning and as years passed, PHP grew in features, language stability, maturity, and developer adoption.

PHP milestones

Now that we have some historical background, we can focus on the main milestones achieved by PHP throughout the years. Each release increased the language stability and added more and more features.

Version 4.x

PHP 4 was the first release, which included the Zend Engine. This engine increased the average performance of PHP. Along with the Zend Engine, PHP 4 included support for more web servers, HTTP sessions, output buffering, and increased security.

Version 5.x

PHP 5 was released on July 13, 2004 using the Zend Engine II, which increased the language performance once again. This release included important improvements for **object-oriented** (**OO**) programming, making the language more flexible and robust. Now, users were able to choose between developing applications in a procedural or a stable OO way; they could have the best of both worlds. In this release, one of the most important extensions used to connect to data stores was also included—the **PHP Data Objects** (**PDO**) extension.

With PHP 5 becoming the most stable version in 2008, many open source projects started ending their support for PHP 4 in their new code.

Version 6.x

This release was one of the most famous failures for PHP. The development of this major release started in 2005 but, in 2010, it was abandoned due to difficulties with the Unicode implementation. Not all the work was thrown away and most of the features, one of them being namespaces, were added to the previous releases.

As a side note, version 6 is generally associated with a failure in the tech world: PHP 6, Perl 6, and even MySQL 6 were never released.

Version 7.x

This was a long awaited release—one release to rule them all and a release with performance levels seen never before.

On December 3, 2015, version 7.0.0 was released with the last Zend Engine available. The performance increase obtained only by changing the running version on your machine reached up to 70%, with a very small memory footprint.

The language also evolved, and PHP now had a better 64-bit support and a secure random number generator. Now you could create anonymous classes or define return types among other major changes.

This release became serious competition to the other so-called *enterprise languages*.

Advantages

PHP is one of the most used programming languages you can use to build your web projects. In case you are not yet sure if PHP is the appropriate language for your next application, let us now to tell you the main advantages of using PHP:

- **Big community**: It's very easy to engage in conferences, events, and meetups all over the world with ZendCon, PHP[world], or International PHP Conference. Not only can you talk to other PHP developers at events and conferences, but you can join IRC/Slack channels or mailing lists to ask questions and keep yourself updated. One of the many locations where you can find events near your area is on the official site at `http://php.net/cal.php`.
- **Great documentation**: The main point of information about the language is available on the PHP website (`http://php.net/docs.php`). This reference guide covers every aspect of PHP, from basic control flows to advanced topics. Do you want to read a book? No problem, it will be easy to find a suitable book among more than 15,000 Amazon references. Even if you need more information, a quick search on Google will give you more than 9,300,000,000 results.

- **Stable**: The PHP project makes frequent releases and maintains several major releases at the same time until their scheduled **End Of Life** (**EOL**). Usually, the time between a release and its EOL is enough to move to the next mainstream version.
- **Easy to deploy**: PHP is the most popular server-side programming language with an 81.8% usage in August 2016, so the market moves in the same direction. It is very easy to find a hosting provider with PHP preinstalled and ready to use, so you only need to deal with the deploy.
  There are a number of ways of deploying your code into production. For instance, you can track your code in a Git repository and do a Git pull on your server later. You could also push your files through FTP to the public location.You can even build a **Continuous Integration/Continuous Delivery** (**CI/CD**) system with Jenkins, Docker, Terraform, Packer, Ansible, or other tools. The complexity of the deploy will always match the project complexity.

- **Easy to install**: PHP has prebuilt packages for the major operating systems: it can be installed on Linux, Mac, Windows, or Unix. Sometimes the packages are available inside the package system (for instance, apt). In other cases, you need external tools to install it, such as Homebrew. In the worst case scenario, you can download the source code and compile it on your machine.
- **Open Source**: All the PHP source code is available at GitHub, so it is really simple for any developer to take a deep look at how everything works. This openness allows a programmer to participate in the project, extending the language or fixing the bugs. The license used in PHP is a **Berkeley Software Distribution** (**BSD**) style license without the *copyleft* restrictions associated with GPL.
- **High running speed (PHP 7)**: In the past, PHP was not fast enough, but this situation changed completely with PHP 7. This major release is based on the **PHP Next-Gen** (**PHPNG**) project with Zend Technologies as the leader. The idea behind PHPNG was to speed up PHP applications and they did this very well. The performance gains can vary between 25% and 70% only by changing the PHP version.
- **High number of frameworks and libraries available**: The PHP ecosystem is very rich in libraries and frameworks. The common way to find a suitable library for your project is using PEAR and PECL.
  Regarding the available frameworks, you can use one of the best, for example Zend Framework, Symfony, Laravel, Phalcon, CodeIgniter, or you can build one from scratch if you can't find one that meets your requirements. The best part about this is that all of these are open source, so you can always extend them.

- **High development speed**: PHP has a relatively small learning curve, which can help you start coding from the beginning. Also, the similarities in syntax with C, Perl, and other languages can make you understand how everything works in no time. PHP avoids wasting time waiting for the compiler to generate our build.
- **Easy to debug**: As you probably know, PHP is an interpreted language. Therefore, when trying to solve an issue, you have multiple options to succeed. The easy way is dropping a few `var_dump` or `print_r` calls to view what the code is doing. For a deeper view of the execution, you can link your IDE to Xdebug or Zend Debug and start tracing your application.
- **Easy to test**: No modern programming language will survive in the wild without an appropriate test suite, so you have to ensure that your application will continue running as expected. Don't worry, the PHP community has your back as you have available multiple tools to do any kind of tests. For instance, you can use PHPUnit for your **Test Driven Development** (**TDD**), or Behat if you are following a **Behavior Driven Development** (**BDD**).
- **You can do anything**: PHP is a modern programming language with multiple applications in the real world. Therefore, only sky is the limit! You can build a GUI application using the GTK extension, or you can create a terminal command with all the required files in a phar archive. The language has no limitations, so anything can be built.

Disadvantages

Like any programming language, PHP also has some disadvantages. Some of the most common mentioned disadvantages flagged are: security issues, unsuitable for large applications, and weak types. PHP started as a collection of CGIs and has become more modern and robust throughout the years, so it is pretty robust and flexible for a young programming language (in comparison with other languages).

In any case, an experienced developer will have no problem overcoming these disadvantages when building their application if they use the best practices.

As you can already see, the evolution of PHP was enormous. It has one of the most vibrant communities, was made for the web, and has all the power you need to create any kind of project. Without a doubt, PHP will be the right language for you to express your best ideas.

Summary

In this chapter, we looked at how microservices stand against monolithic architecture and SOA. We learned about the essential components of microservices architecture along with its advantages and disadvantages. Further along the chapter, we looked at how to implement the microservices architecture and the prerequisites to take into consideration before switching to microservices. Later, we covered the history of PHP versions along with their advantages and disadvantages.

2
Development Environment

In this chapter, we will start building our application based on microservices now that we know why microservices are necessary for the development of our application and the advantages we can enjoy if we base the application on microservices.

The application that we will develop in this book (which is similar to Pokemon GO) is called *Finding secrets*. This application will be like a game using geolocation to find different secrets around the world. The entire world keeps a lot of hidden secrets and the players will have to find them as soon as possible. There are 100 different kinds of secrets, and they will generate and appear in different parts of the world every day, so the players will be able to find them by walking around different areas and checking to see if there are any kind of secrets nearby.

The secrets will be saved in the application wallet and if the player finds a secret that they already have in their wallet, they will not be able to collect it.

The players will be able to duel against other players if they are close. The duel consists of throwing a dice to get the highest number, and the player who loses will give a random secret to the other player.

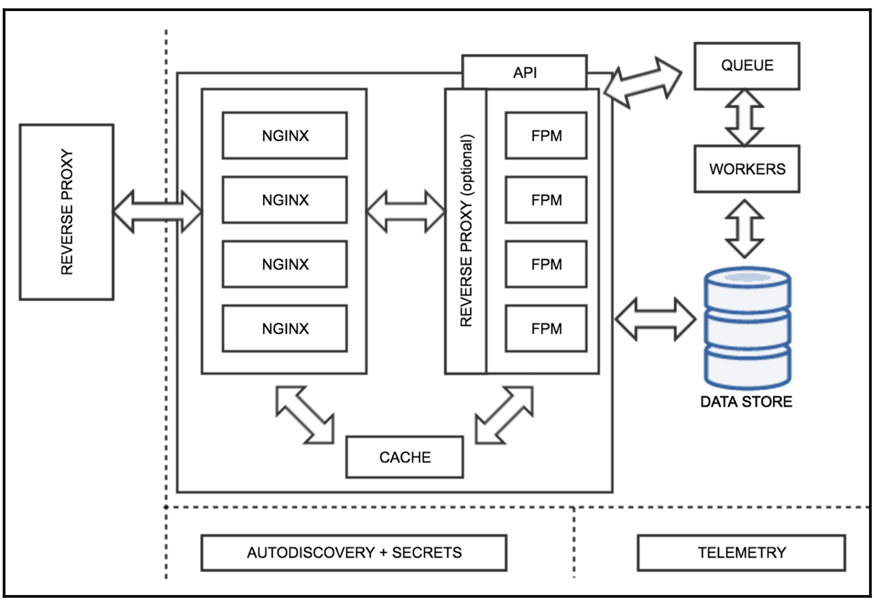
In the subsequent chapters, the specific functions will be more detailed, but in this chapter, we only need to know how the applications works in order to have a general overview of the entire application to start building the basic platform based on microservices.

Design and architecture to build the basic platform for microservices

Creating an application based on microservices is not like a monolithic application. For this reason, we have to divide our functionalities into different services. To do this, it is important to follow an adequate design and structure each of the microservices according to its requirements.

The design takes care of dividing the application into logical parts and groups them according to their existing relationship. The architecture takes care of defining which concrete elements support each of the microservices, for example, where the data is stored or the communication between the services.

Throughout the book, we will follow the given structure for each microservice. In the following image, you will see the structure of one of the microservices, the rest of them are similar; however, some parts are optional:



All the requests for our microservices come from a **REVERSE PROXY** as this allows us to balance the load. Also, we use **NGINX** as a gateway for the **API** built in PHP. To reduce the load and increase the performance of PHP and **NGINX**, we can use a **CACHE** layer.

In case we need to execute big, resource consuming tasks, or the tasks do not need to be executed in a concrete time window, our **API** can use a **QUEUE** system.

In case we need to store some data, our **API** is responsible for managing the access and saving the data in our **DATA STORE**.

In this book, we will be using containerization, a new virtualization method which spins ups containers instead of full virtual machines. Each container will have only the minimum resources and software installed to run your application.

We can use Telemetry (it is a system that gets the stats from the container) and autodiscovery (it is a system that helps us to see which containers are working properly) to supervise the container ecosystem.

Requirements to start working on microservices

Now that you understand why you can use PHP (especially the latest release, version 7) for your next project, it is time to talk about other requirements for the success of your microservices project.

You probably have the importance of the scalability of your application in mind, but how can you do it within a budget? The response is virtualization. With this technology, you will be wasting less resources. In the past, only one **Operating System** (**OS**) could be executed at a time on the same hardware, but with the birth of virtualization, you can have multiple OSes running concurrently. The greatest achievement in your project will be that you will be running more servers dedicated to your microservices but using less hardware.

Given the advantages provided by the virtualization and containerization, nowadays using containers in the development of an application based on microservices is a default standard. There are multiple containerization projects, but the most used and supported is Docker. For this reason, this is the main requirement to start working with microservices.

The following are the different tools/software that we will be using in our Docker environment:

- PHP 7
- Data storage: Percona, MySQL, PostgreSQL
- Reverse proxy: NGINX, Fabio, Traefik,
- Dependency management: composer
- Testing tools: PHPUnit, Behat, Selenium
- Version control: Git

In `Chapter 5`, *Microservices Development*, we will explain how to add each one to our project.

Due to the fact that our main requirement is a containerization suite, we will explain how to install and test Docker in the following sections.

Docker installation

Docker can be installed from two different channels, each one with advantages and disadvantages:

- **Stable channel**: As the name indicates, everything you install from this channel is fully tested and you will have the latest GA version of the Docker engine. This is the most reliable platform and therefore, suitable for production environments. Through this channel, the releases follow a version schedule with long testing and beta times, only to ensure that everything should work as expected.
- **Beta channel**: If you need to have the latest features, this is your channel. All the installers come with the experimental version of the Docker engine where bugs can be found, so it is not recommended for production environments. This channel is a continuation of the Docker beta program where you can provide feedback and there is no version schedule, so you will have more frequent releases.

We will be developing for a stable production environment, so you can forget about the beta channel for now as all you need is on the stable releases.

Docker was born in Linux, so the best implementation was done for this OS. With other OSes, such as Windows or macOS, you have two options: a native implementation and a toolbox installation if you cannot use the native implementation.

Installing Docker on macOS

On macOS, you have two options to install Docker depending on whether your machine matches the minimum requirements or not. With relatively new machines (OS X 10.10 Yosemite and higher), you can install the native implementation that uses Hyperkit, a lightweight OS X virtualization solution built on top of **Hypervisor.Framework**. If you have an older machine that does not match the minimum requirements, you can install the Docker Toolbox.

Docker for Mac (alias, native implementation) versus Docker toolbox

The Docker Toolbox was the first implementation of Docker on macOS and it does not have a deep OS integration. It uses VirtualBox to spin a Linux virtual machine where Docker will be running. Using a **Virtual Machine** (**VM**) where you will be running all your containers has numerous problems, the most noticeable being the poor performance. However, it is the desired choice if your machine does not match the requirements for the native implementation.

Docker for Mac is a native Mac application with a native user interface and auto-update capability, and it is deeply integrated with OS X native virtualization (Hypervisor.Framework), networking, and filesystem. This version is faster, easier to use, and more reliable than the Docker Toolbox.

Minimum requirements

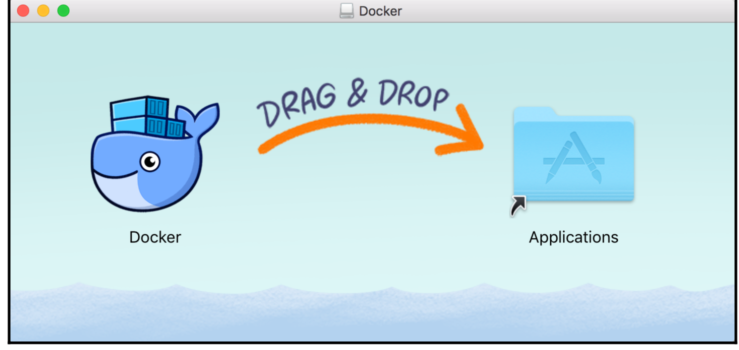
- Mac must be a 2010 or newer model, with Intel's hardware support for **Memory Management Unit** (**MMU**) virtualization; that is, **Extended Page Tables** (**EPT**)
- OS X 10.10.3 Yosemite or newer
- At least 4 GB of RAM
- VirtualBox prior to version 4.3.30 must not be installed (it is incompatible with Docker for Mac)

Docker for Mac installation process

If your machine passes the requirements, you can download the Docker for Mac from the official page, that is `https://www.docker.com/products/docker`.

Once you have the image downloaded on your machine, you can carry out the following steps:

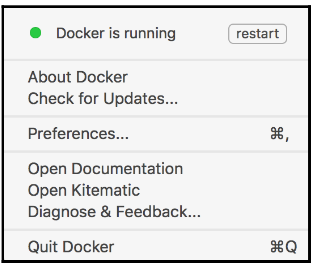
1. Double-click on the downloaded image (called `Docker.dmg`) to open the installer. Once the image is mounted, you need to drag and drop the Docker app into the `Applications` folder:



The `Docker.app` may ask you for your password during the installation process to install and set up network components in the privileged mode.

2. Once the installation is complete, Docker will be available on your Launchpad and in your `Applications` folder. Execute the application to start Docker.
Once Docker starts up, you will see the whale icon in your toolbar. This will be your quick access to settings.

3. Click on the toolbar whale for **Preferences...** and other options:



4. Click on **About Docker** to find out if you are running the latest version.

Installing Docker on Linux

The Docker ecosystem was developed on top of Linux, so the installation process on this OS is easier. In the following pages, we will only cover the installation on **Community ENTerprise Operating System** (**CentOS**)/**Red Hat Enterprise Linux** (**RHEL**) (they use yum as the package manager) and Ubuntu (uses apt as the package manager).

CentOS/RHEL

Docker can be executed on CentOS 7 and on any other binary compatible EL7 distribution but Docker is not tested or supported on these compatible distributions.

Minimum requirements

The minimum requirement to install and execute Docker is to have a 64-bit OS and a kernel version 3.10 or higher. If you need to know your current version, you can open a terminal and execute the following command:

```
$ uname -r
3.10.0-229.el7.x86_64
```

Note that it's recommended to have your OS up to date as it will avoid any potential kernel bugs.

Installing Docker using yum

First of all, you need to have a user with root privileges; you can log in on your machine as this user or use a `sudo` command on the terminal of your choice. In the following steps, we assume that you are using a root or privileged user (add sudo to the commands if you are not using a root user).

First of all, ensure that all your existing packages are up to date:

```
yum update
```

Now that your machine has the latest packages available, you need to add the official Docker `yum` repository:

```
# tee /etc/yum.repos.d/docker.repo <<-'EOF'
[dockerrepo]
name=Docker Repository
baseurl=https://yum.dockerproject.org/repo/main/centos/7/
enabled=1
gpgcheck=1
```

```
gpgkey=https://yum.dockerproject.org/gpg
EOF
```

After adding the yum repository to your CentOS/RHEL, you can easily install the Docker package with the following command:

```
yum install docker-engine
```

You can add the Docker service to the startup of your OS with the `systemctl` command (this step is optional):

```
systemctl enable docker.service
```

The same `systemctl` command can be used to start the service:

```
 systemctl start docker
```

Now you have everything installed and running, so you can start testing and playing with Docker.

Post-install setup – creating a Docker group

Docker is executed as a daemon that binds to a Unix socket. This socket is owned by root, so the only way for other users to access it is with the `sudo` command. Using the `sudo` command every time you use any Docker command can be painful, but you can create a Unix group, called `docker`, and assign users to this group. By making this small change, the Docker daemon will start and assign the ownership of the Unix socket to this new group.

Listed are the commands to create a Docker group:

```
groupadd docker
usermod -aG docker my_username
```

After these commands, you need to log out and log in again to refresh your permissions.

Installing Docker on Ubuntu

Ubuntu is officially supported and the main recommendation is to use an LTS (the last version is always recommended):

- Ubuntu Xenial 16.04 (LTS)
- Ubuntu Trusty 14.04 (LTS)
- Ubuntu Precise 12.04 (LTS)

Just as in the previous Linux installation steps, we are assuming that you are using a root or privileged user to install and set up Docker.

Minimum requirements

As with other Linux distributions, a 64-bit version is required and your kernel version needs to be at least a 3.10. Older kernel versions have known bugs that cause data loss and frequent kernel panics.

To check your current kernel version, open your favorite terminal and run:

```
$ uname -r
3.11.0-15-generic
```

Installing Docker using apt

First of all, ensure that you have your apt sources pointing to the Docker repository, especially if you have previously installed Docker from the apt. In addition to this, update your system:

```
apt-get update
```

Now that you have your system up to date, it is time to install some required packages and the new GPG key:

```
apt-get install apt-transport-https ca-certificates
apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys \
58118E89F3A912897C070ADBF76221572C52609D
```

With Ubuntu, it is very easy to add the official Docker repository; you only need to create (or edit) the `/etc/apt/sources.list.d/docker.list` file in your favorite editor.

In the case of having the previous lines from old repositories, delete all the content and add one of the following entries. Ensure that you match your current Ubuntu version:

```
Ubuntu Precise 12.04 (LTS):
deb https://apt.dockerproject.org/repo ubuntu-precise main
Ubuntu Trusty 14.04 (LTS):
deb https://apt.dockerproject.org/repo ubuntu-trusty main
Ubuntu Xenial 16.04 (LTS):
deb https://apt.dockerproject.org/repo ubuntu-xenial main
```

After saving the file, you need to update the `apt` package index:

```
apt-get update
```

In the case of having a previous Docker repo on your Ubuntu, you need to purge the old repo:

```
apt-get purge lxc-docker
```

On Trusty and Xenial, it is recommended that you install the `linux-image-extra-*` kernel package that allows you to use the AFUS storage driver. To install them, run the following command:

```
apt-get update && apt-get install linux-image-extra-$(uname -r) linux-image-extra-virtual
```

On Precise, Docker requires a 3.13 kernel version, so ensure that you have the correct kernel; if your version is older, you must upgrade it.

At this point, your machine will be more than ready to install Docker. It can be done with a single command, as with `yum`:

```
apt-get install docker-engine
```

Now that you have everything installed and running, you can start playing and testing Docker.

Common issues on Ubuntu

If you see errors related to swap limits while you are using Docker, you need to enable memory and swap on your system. It can be done on GNU GRUB by following the given steps:

1. Edit the `/etc/default/grub` file.
2. Set the `GRUB_CMDLINE_LINUX` as follows:

   ```
   GRUB_CMDLINE_LINUX="cgroup_enable=memory swapaccount=1"
   ```

3. Update grub:

   ```
   update-grub
   ```

4. Reboot the system.

UFW forwarding

Ubuntu comes with **Uncomplicated Firewall** (**UFW**) and if it is enabled on the same host as the one where you run Docker, you will need to make some adjustments because, by default, UFW will drop any forwarding traffic. Also, UFW will deny any incoming traffic, making the reach of your containers from a different host impossible. The Docker default port is 2376 when TLS is enabled and 2375 in other cases. On a clean installation, Docker runs without TLS enabled. Let's configure UFW!

First, you can check if UFW is installed and enabled:

```
ufw status
```

Now that you are sure that UFW is installed and running, you can edit the `config` file, `/etc/default/ufw`, with your favorite editor and set up the `DEFAULT_FORWARD_POLICY`:

```
vi /etc/default/ufw
DEFAULT_FORWARD_POLICY="ACCEPT"
```

You can now save and close the `config` file and, after the restart of the UFW, your changes will be available:

```
ufw reload
```

Allowing incoming connections to the Docker port can be done with the `ufw` command:

```
ufw allow 2375/tcp
```

DNS server

Ubuntu and its derivatives use 127.0.0.1 as the default name server in the `/etc/resolv.conf` file, so when you start containers with this configuration, you will see warnings because Docker can't use the local DNS nameserver.

If you want to avoid these warnings you need to specify a DNS server to be used by Docker or disable `dnsmasq` in the `NetworkManager`. Note that disabling `dnsmasq` will make DNS resolutions a little bit slower.

To specify a DNS server, you can open the `/etc/default/docker` file with your favorite editor and add the following setting:

```
DOCKER_OPTS="--dns 8.8.8.8"
```

Replace `8.8.8.8` with your local DNS server. If you have multiple DNS servers, you can add multiple `--dns` records separated with spaces. Consider the following example:

```
DOCKER_OPTS="--dns 8.8.8.8 --dns 9.9.9.9"
```

As soon as you have your changes saved, you need to restart the Docker daemon:

```
service docker restart
```

If you don't have a local DNS server and you want to disable `dnsmasq`, open the `/etc/NetworkManager/NetworkManager.conf` file with your editor and comment out the following line:

```
dns=dnsmasq
```

Save the changes and restart the NetworkManager and Docker:

```
restart network-manager
restart docker
```

Post-install setup – creating a Docker group

Docker is executed as a daemon that binds to a Unix socket. This socket is owned by root, so the only way for other users to access it is with the `sudo` command. Using the `sudo` command every time you use any docker command can be painful, but you can create a Unix group, called `docker`, and assign users to this group. Making this small change, the Docker daemon will start and assign the ownership of the Unix socket to this new group.

Perform the following steps to create a Docker group:

```
groupadd docker
usermod -aG docker my_username
```

After these steps, you need to log out and log in again to refresh your permissions.

Starting Docker on boot

Ubuntu 15.04 onwards uses the `systemd` system as its boot and service manager, while for versions 14.10 and the previous ones, it uses the `upstart` system.

For the 15.04 and up systems, you can configure the Docker daemon to start on boot by running the given command:

```
systemctl enable docker
```

In the case of using older versions, the installation method automatically configures upstart to start the Docker daemon on boot.

Installing Docker on Windows

The Docker team has made a huge effort to bring their entire ecosystem to any OS and they didn't forget about Windows. As on macOS, you have two options to install Docker on this OS: a toolbox and a more native option.

Minimum requirements

Docker for Windows requires a 64-bit Windows 10 Pro, Enterprise, and Education (1511 November update, Build 10586 or later), and the `Hyper-V` package must be enabled.

In case your machine is running a different version, you can install the Toolbox that requires a 64-bit OS running at least Windows 7 and with virtualization enabled on the machine. As you can see, it has lighter requirements.

Due to the fact that Docker for Windows requires at least a Pro/Enterprise/Education version and the majority of computers are sold with a different version, we will explain how to install Docker with the toolbox.

Installing the Docker tools

The Docker tools use VirtualBox to spin a virtual machine that will run the Docker engine. The installation package can be downloaded from `https://www.docker.com/products/docker-toolbox`.

Once you have the installer, you only need to double-click on the downloaded executable to start the installation process.

The first window shown by the installer allows you to send debug information to Docker to improve the ecosystem. Allowing the Docker engine to send debug information from your development environment can help the community to find bugs and improve the ecosystem. The recommendation on this option is to have it enabled at least in your development environment:

Just like any other Windows installer, you can choose where it will be installed. In most cases, the default will be fine for your development environment.

By default, the installer will add all the required packages and some extras to your machine. In this step of the installation, you can purge some non-required software. Some of the optional packages are as follows:

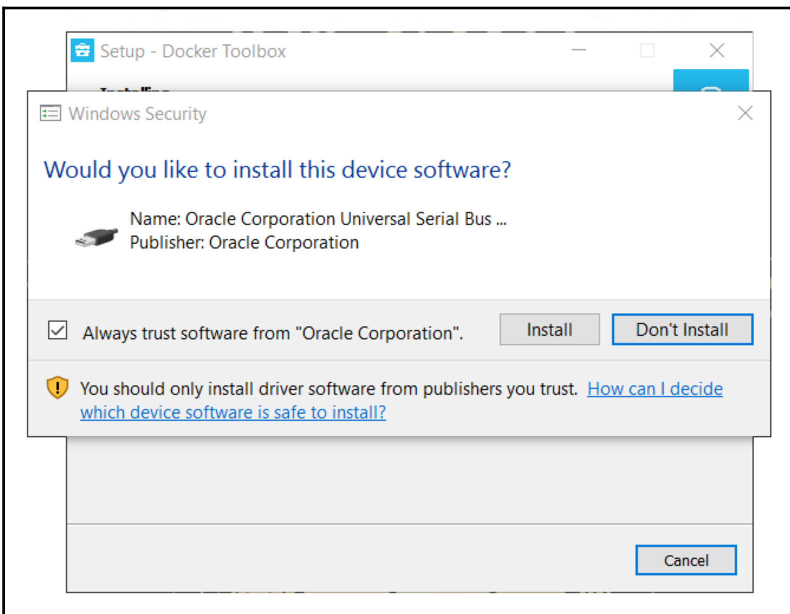
- **Docker compose for Windows**: In our opinion, this is a must as we will be using the package in our book.
- **Kitematic for Windows:** This application is a GUI to manage your containers easily. If you are not comfortable with the command line, you can install this package.
- **Git for Windows:** This is another must-install package; we will be using Git to store and manage our project.

After choosing the packages we want to install, it is time for some additional tasks. The default selected tasks will be fine for your dev environment.

Now, you only need to confirm all the setup you have done in the previous steps before the installation starts.

The installation can take several minutes to complete, so be patient.

While the installation progresses, you may be alerted about the installation of an Oracle device. This is due to the fact that the tools are using VirtualBox to spin up a virtual machine to run the Docker engine. Install this device to avoid future headaches:



Congratulations! You have Docker installed on your Windows machine. Don't waste another minute and start testing and playing with your Docker ecosystem.

How to check your Docker engine, compose, and machine versions

Now that you have Docker installed, you only need to open your favorite terminal and type in the following command:

```
$ docker --version && docker-compose --version && docker-machine --version
```

Quick example to check your Docker installation

You should have the Docker running and execute the following command:

```
$ docker run -d -p 8080:80 --name webserver-test nginx
```

The preceding command will do the following things:

- Execute the container in the background with `-d`
- Map the 8080 port of your machine to the 80 port of the container with `-p 8080:80`
- Assign a name to your container with `--name webserver-test`
- Get the NGINX Docker image and make the container to run this image.

Now, open your favorite browser and navigate to `http://localhost:8080`, where you will see a default NGINX page.

Common management tasks

By executing `docker ps` on your terminal, you can see the running containers.

The preceding command gives us a deeper view of the containers that are running on your machine, the image they are using, when they were created, the status, and the port mapping or the assigned name.

Once you finish playing with your container, it's time to stop it. Execute `docker stop webserver-test` and the container will end its life.

Oops! You need the same container again. No problem, because a simple `docker start webserver-test` will again spin up the container for you.

Now, it is time to stop and remove the container because you are not going to use it anymore. Executing `docker rm -f webserver-test` on your terminal will do the trick. Note that this command will remove the container but not the downloaded image we have used. For this last step, you can do a `docker rmi nginx`.

Version control – Git versus SVN

Version control is a tool that helps you recall the previous versions of your source code to check them and work with them; it is agnostic of the language or technology used and it is possible to use a version control in all softwares developed in plain text.

We can categorize the versioning control tools into the following categories:

- **Centralized version**: Control system needs a centralized server to work and all developers need to be connected to it so that they synchronize and download the changes from it.
- **Distributed version**: Control system is not centralized; in other words, each developer has the entire management version control system on their own machine, so it is possible to work locally and then synchronize it with a common server or with each developer. **Distributed Version Control Systems** (**DVCS**) are faster because they need less changes on the centralized or shared server.

Subversion (SVN) is a centralized version control system and, for this reason, some developers think that it is the best way to work respecting the entire project, so the developer just needs to write and read the access controllers in one place.

The entire code is hosted in one place, so it is possible to think that this way, it is easier to understand SVN better than Git. The truth is that the SVN command line is easier and there are more GUIs available for SVN. The reason is clear: SVN has existed since the year 2000, and Git came 5 years later.

Another advantage of SVN is that the system to number the versions is clearer; it uses a sequential number system (1,2,3,4…) and Git uses SHA-1 codes, which are more difficult to read and understand.

Finally, with SVN, it is possible to get a subdirectory to work with it without the need of having the entire project. This is not a problem for small projects, but it can be difficult when you have a large project.

Git

In this book, we will use Git for our version control. We made this decision because Git is definitely faster and more lightweight (it takes up 30 times less disk space than SVN). Also, Git became the standard version control on web development version control and our goal is to create an application based on microservices using PHP, so Git is a great solution for the project.

The advantages of Git are as follows:

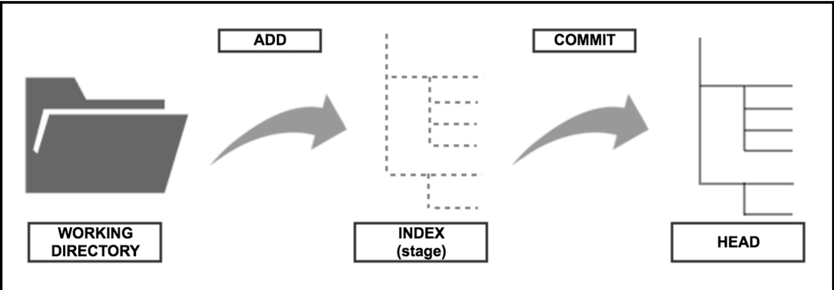
- The branches are more lightweight than SVN
- Git is a lot of faster than SVN
- Git is a DVCS from the start, so the developer has total control of its local
- Git provides better auditory in branching and merging

In the subsequent chapters, we will use Git commands on our project, explaining each one and giving examples, but until then, let's take a look at the basic ones:

How to create a new repository: Create a new folder, open it, and execute Git in it to create a new Git repository.

How to checkout an existing repository: Create a local copy from the repository executing Git clone `/path/to/repository`. If you are using a remote server (hosting centralized servers will be explained in the following lines), execute Git clone `username@host:/path/to/repository`.

To understand the Git workflow, it is necessary to know that there are three different trees:

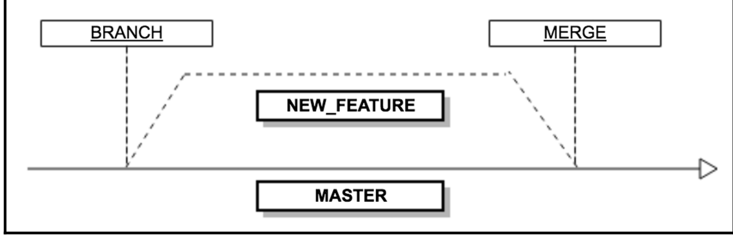


- **WORKING DIRECTORY**: This contains the files of your project
- **INDEX**: This works as the intermediate area; files will be here until they are committed
- **HEAD**: This points to the last commit done

Adding files to the INDEX and committing them are easy tasks. After working with files on your project and changing them, you have to add them to the index:

- **How to add files to the INDEX**: Checking the files before adding them to the INDEX is recommended. You can do this by executing `git diff <file>`. It will show you the added and deleted lines, and some more interesting information about the file you modified. Once you are happy with the changes made, you can execute `git add <filename>` to add a specific file or add Git to all the files you have modified.
- **How to add files to the HEAD**: Once you have included all the necessary files in the INDEX, you have to commit them. You can do this by executing `git commit -m "Commit message"` . Now the files are included in the HEAD in your local copy, but they are still not in the remote repository.
- **How to send changes to the remote repository**: To send the changes included in your HEAD local copy to the remote repository, execute `git push origin <branch name>;` you can choose the branch where you want to include the changes, for example, master. If you did not clone an existing repository and you want to connect your local repository to a remote one, execute `git remote add origin <server>`.

The branches are used to develop isolated functions and can be merged with the main branch in the future. The default branch when a new repository is created is called master. The workflow overview will be something like this:



- **How to create a new branch**: Once you are in the branch from where you want to create a new one, execute `git checkout -b new_feature`
- **How to change the branch**: You can navigate through the branches by executing `git checkout <branch name>`

- **How to delete a branch**: You can delete a branch by executing `git branch -d new_feature`
- **How to make your branch available to everyone**: A branch will not be available to the rest of developers until you upload your branch to the remote repository by executing `git push origin <branch>`

If you want to update your local copy with the changes made on the remote repository, you can execute `git fetch` to check if there are any new updates and then `git pull` to get that update.

To merge your active branch with a different branch, execute `git merge <branch>` and Git will attempt to fuse both branches but, sometimes, if two or more developers changed the same file, there could be conflicts and you will need to solve those conflicts manually before the merge and then put the modified files into the INDEX again.

If you fail, maybe you want to trash your local changes and get the ones from the repository again. You can do this by executing `git checkout -- <filename>`. In case you want to trash all your local changes and commits, execute `git fetch origin and git reset --hard origin/master`.

Hosting

When we are working in a team, maybe we want to have a common repository with a centralized server. Remember that Git is a DVCS and it is not necessary to use a place to have the code centralized, but in case you want to use it for different reasons, we will look at the two famous ones.

The hostings provide you with a better way to manage your repository using a web interface.

GitHub

GitHub is the place to host the code chosen by a majority of developers. It is based on Git, and companies, such as Twitter and Facebook, use this service to put their open source projects. GitHub became the most famous source host in just a few years and currently, many companies ask for a candidate's GitHub repository before their technical interview.

This hosting is free for all developers; they can create unlimited projects with unlimited collaborators and only one condition; the project needs to be open source and public. If you want to have a private project, you will have to pay.

Having your project as public on GitHub is a good opportunity to show your project to the world and take advantage of the big GitHub community. It is possible to ask for help because there are many registered and active developers.

You can visit the official website at `https://github.com/`.

BitBucket

BitBucket is an alternative place to host your project. It uses Git, but you can use Mercurial too. The interface is pretty similar to GitHub. A great advantage of BitBucket is the company that makes it possible–Atlassian. It has many functionalities for developers included in their hosting, for example, the possibility of integrating other Atlassian tools or a small continuous delivery tool which allows you to build, test, and deploy your application.

This one is free regardless of the project you want: public or private. The only limitation is that it only allows five collaborators for each project; if you need more people working on your project you will have to pay.

Official website: `https://bitbucket.org/`.

Version control strategies

When you are developing an application, it is important to keep your code nice and clean, but it is even more important when you work with other developers. In this section, we will give you a small introduction to the most known version control strategies you can use in your project.

Centralized work

This strategy is the most common for developers who were previously using SVN (old school) or similar version controls. Like Subversion, the project is hosted in a central repository with a unique point of entry. This strategy does not need more branches except master (trunk in SVN).

Developers clone the entire project on their local machines, work on the project, and then they commit the changes. When they want to publish the changes, they execute push.

Feature branch workflow

This is the next step from centralized work. It works with a centralized repository also, but developers create a local feature branch in their copy and this one is published on the centralized repository too, so all the developers have the chance to participate in that feature. The branches will have descriptive names or number of issues.

In this strategy, the master never contains errors, so this is a great improvement for continuous integration. Also, having specific branches for each feature is a good encapsulation to not disturb the main code base.

Gitflow workflow

Gitflow workflow does not add more new concepts than feature branch workflow. It just assigns different roles for each branch. Gitflow workflow works with a centralized repository too, and developers create branches on it, such as feature branch workflow. However, the branches have a specific function, for example, development, release, or feature. So, the feature branches will be merged with a specific release and then with master. This way, it is possible to have different releases for the same project.

This strategy is used for large projects or projects that need releases.

Forking workflow

The last strategy is quite different to others that we have looked at in this chapter. Instead of cloning a copy from the centralized server and working on it, this one gives a fork of that to every developer. This means that every developer has two copies of the project: a *private* one and the *server-side*.

Once the developer makes the changes they want, they are sent to the project maintainer to be reviewed and checked so that they do not break the project and then they are merged with the main repository.

This strategy is used in open source projects, so the developers cannot break the current project.

Semantic versioning

It is really important to have a versioning system in our microservice or API. This allows the consumers and yourself to have a coherent system of versioning so that everyone can know the importance of a release or feature.

Set the version number as `MAJOR.MINOR.PATCH`:

1. `MAJOR` will be incremented when incompatible API changes are made, so developers need to trash the current API version and use the new one.
2. `MINOR` will be incremented when a new feature is added and it is compatible with the current code. The developer, therefore, does not need to change the entire API just to update the current one.
3. `PATCH` will be incremented when a new bug fix for the current version is done.

This is a summary of semantic versioning, but you can find more information at `http://semver.org/`.

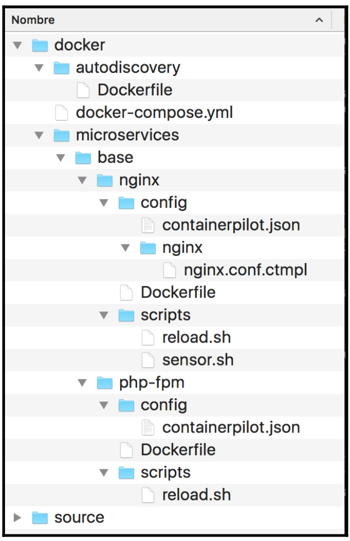
Setting up a development environment for microservices

One of the greatest benefits of using Docker and its container ecosystem is that you don't need to install anything else on your machine. For example, if you need a MySQL database, you don't need to install anything on your local dev; it is easier to spin a container with the version you want and start using it.

This way of developing is more flexible, therefore we will be working with Docker containers throughout the whole book. In this section, we will learn how to build a basic Docker environment; it will be our foundation and we will be improving and adapting this base to each of our microservices in the subsequent chapters.

To simplify the folder structure of our project, we will have some root folders on our development machine:

- `Docker`: This folder will contain all the Docker environment
- `Source`: This folder will have the source of each of our microservices



Note that this structure is flexible and can be changed and adapted to your specific requirements without any problems.

All the required files are available on our GitHub repository, at `https://github.com/php-microservices/docker`, on the master branch with the `chapter-02` tag.

Let's dig deeper into the Docker setup. Open your docker folder and create a file called `docker-compose.yml` with the following content:

```
version: '2'
services:
```

These two lines indicate that we are using the latest syntax for Docker compose and they define a list of services that we will be spinning every time we do a `docker-compose up`. All our services will be added after the `services` declaration.

Autodiscovery service

Autodiscovery is a mechanism in which we don't specify the endpoints of each of our microservices. Each one of our services use a shared registry in which they say that they are available. When a microservice needs to know the location of another microservice, it can consult our autodiscovery registry to know the required endpoint.

For our application, we will be using an autodiscovery mechanism to ensure that our microservices can be scaled easily and if a node is not healthy, we stop sending requests to it. Our choice for this purpose is to use Consul (by HashiCorp), a very small application that we can add to our project. The main role for our Consul container is to keep everything in order, keeping a list of the available and healthy services.

Let's start the project by opening your `docker-compose.yml` file with your favorite IDE/editor and adding the next piece of code just after the `services:` line:

```
version: '2'
services:
    autodiscovery:
        build: ./autodiscovery/
        mem_limit: 128m
        expose:
            - 53
            - 8300
            - 8301
            - 8302
            - 8400
            - 8500
        ports:
            - 8500:8500
        dns:
            - 127.0.0.1
```

In a Docker compose file, the syntax is very easy to understand and always follows the same flow. The first line defines a container type (it is like a class name for devs); in our case it is `autodiscovery`, and inside this container type we can specify several options to adapt the container to our requirements.

With `build: ./autodiscovery/`, we are telling Docker where it can find a Dockerfile that describes what we want in our container in detail.

The `mem_limit: 128m` sentence will limit the memory consumption of any container of the `autodiscovery` type to not more than 128 Mb. Note that this instruction is optional.

Each container needs different ports open and, by default, when you spin a container, none of them are open. For this reason, you need to specify which ports you want open for each container. For example, a container with a web server will need the `port 80` open but for a container that runs MySQL, the required port may be `3306`. In our case, we are opening the ports `53`, `8300`, `8301`, `8302`, `8400`, and `8500` for each one of our `autodiscovery` containers.

If you try to reach the container on one of the opened ports, it will not work. The container ecosystem resides in a separate network and you can only access it if you create a bridge between your environment and the Docker network. Our `autodiscovery` container runs Consul and it has a nice web UI on port `8500`. We want to be able to use this UI; so, when we use `ports`, we are mapping our local `8500` port to the container `8500` port.

Now, it's time to create a new folder called `autodiscovery` in the same path of your `docker-compose.yml file`. Inside this new folder, place a file called `Dockerfile` with the following line:

```
FROM consul:v0.7.0
```

This small sentence inside the `Dockerfile` indicates that we are using a Docker `consul` image with tag `v0.7.0`. This image will be fetched from the official Docker hub, a repository for container images.

At this point, doing a `$ docker-compose up` will spin up a Consul machine, give it a try. Since we didn't specify the `-d` option, the Docker engine will output all the logs to your terminal. You can stop your container with a simple *CTRL+C*. When you add the `-d` option, the Docker compose runs as a daemon and returns the prompt; you can do a `$ docker-compose stop` to stop the containers.

Microservice base core – NGINX and PHP-FPM

PHP-FPM is an alternative to the old way of executing PHP in our web server. The main benefit of using PHP-FPM is its small memory footprint and the high performance under hight loads. The best web server you can find nowadays to run your PHP-FPM is NGINX, a very light web server and reverse proxy used in the most important projects.

Since our application will be using an autodiscovery pattern, we need an easy way of dealing with the service registering, deregistering, and health check. One of the simplest and fastest applications you can use is ContainerPilot, a small micro-orchestration application created by Joyent that works with your favorite container scheduler, in our case Docker compose. This small app is being executed as PID 1 and forks the application we want to run inside the container.

We will be working with ContainerPilot because it relieves the developer of dealing with the autodiscovery, so we need to have the latest version on each container we will be using.

Let's start defining our base `php-fpm` container. Open the `docker-compose.yml` and add a new service for the `php-fpm`:

```
microservice_base_fpm:
  build: ./microservices/base/php-fpm/
links:
  - autodiscovery
expose:
  - 9000
environment:
  - BACKEND=microservice_base_nginx
  - CONSUL=autodiscovery
```

In the preceding code, we are defining a new service and one interesting attribute is links. This attribute defines which other containers our service can see or connect. In our example, we want to link this type of container to any `autodiscovery` container. Without this explicit definition, our `fpm` container won't see the `autodiscovery` service.

Now, create the `microservices/base/php-fpm/Dockerfile` file on your IDE/editor with the following content:

```
FROM php:7-fpm

RUN apt-get update && apt-get -y install \
  git g++ libcurl4-gnutls-dev libicu-dev libmcrypt-dev
  libpq-dev libxml2-dev
  unzip zlib1g-dev \
  && git clone -b php7
  https://github.com/phpredis/phpredis.git
  /usr/src/php/ext/redis \
  && docker-php-ext-install curl intl json mbstring
  mcrypt pdo pdo_pgsql
  redis xml \
  && apt-get autoremove && apt-get autoclean \
  && rm -rf /var/lib/apt/lists/*
```

```
RUN echo 'date.timezone="Europe/Madrid"' >>
  /usr/local/etc/php/conf.d/date.ini
RUN echo 'session.save_path = "/tmp"' >>
  /usr/local/etc/php/conf.d/session.ini

ENV CONSUL_TEMPLATE_VERSION 0.16.0
ENV CONSUL_TEMPLATE_SHA1
064b0b492bb7ca3663811d297436a4bbf3226de706d2b76adade7021cd22e156

RUN curl --retry 7 -Lso /tmp/consul-template.zip \
  "https://releases.hashicorp.com/
  consul-template/${CONSUL_TEMPLATE_VERSION}/
  consul-template_${CONSUL_TEMPLATE_VERSION}_linux_amd64.zip" \
&& echo "${CONSUL_TEMPLATE_SHA1}  /tmp/consul-template.zip"
| sha256sum -c \
&& unzip /tmp/consul-template.zip -d /usr/local/bin \
&& rm /tmp/consul-template.zip

ENV CONTAINERPILOT_VERSION 2.4.3
ENV CONTAINERPILOT_SHA1 2c469a0e79a7ac801f1c032c2515dd0278134790
ENV CONTAINERPILOT file:///etc/containerpilot.json

RUN curl --retry 7 -Lso /tmp/containerpilot.tar.gz \
  "https://github.com/joyent/containerpilot/releases/download/
  ${CONTAINERPILOT_VERSION}/containerpilot-
  ${CONTAINERPILOT_VERSION}.tar.gz"
  \
  && echo "${CONTAINERPILOT_SHA1}  /tmp/containerpilot.tar.gz"
  | sha1sum -c \
  && tar zxf /tmp/containerpilot.tar.gz -C /usr/local/bin \
  && rm /tmp/containerpilot.tar.gz

COPY config/ /etc
COPY scripts/ /usr/local/bin

RUN chmod +x /usr/local/bin/reload.sh

CMD [ "/usr/local/bin/containerpilot", "php-fpm", "--nodaemonize"]
```

What we have done on this file is tell Docker how it needs to create our `php-fpm` container. The first line declares the official version we want to use as a foundation for our container, in this case php7 fpm. Once the image is downloaded, the first `RUN` line will add all the extra `PHP` packages we will be using.

The two `RUN` sentences will add bespoke PHP configurations; feel free to adapt these lines to your requirements.

Once all the PHP tasks are done, it is time to install a small application on the container that will help us to deal with templates–`consul-template`. This application is used to build configuration templates on the fly using the information we have stored on our Consul service.

As we said before, we are using ContainerPilot. So, after the `consul-template` installation, we are telling Docker how to install this application.

At this point, Docker finishes installing all the required packages and copies some configuration and shell scripts needed by ContainerPilot

The last line starts ContainerPilot as PID 1 and forks `php-fpm`.

Now, let's explain the configuration file required by ContainerPilot. Open your IDE/editor and create the `microservices/base/php-fpm/config/containerpilot.json` file with the following content:

```
{
  "consul": "{{ if .CONSUL_AGENT }}localhost{{ else }}{{ .CONSUL }}
  {{ end }}:8500",
  "preStart": "/usr/local/bin/reload.sh preStart",
  "logging": {"level": "DEBUG"},
  "services": [
    {
      "name": "microservice_base_fpm",
      "port": 80,
      "health": "/usr/local/sbin/php-fpm -t",
      "poll": 10,
      "ttl": 25,
      "interfaces": ["eth1", "eth0"]
    }
  ],
  "backends": [
    {
      "name": "{{ .BACKEND }}",
      "poll": 7,
      "onChange": "/usr/local/bin/reload.sh"
    }
  ],
  "coprocesses": [{{ if .CONSUL_AGENT }}
    {
      "command": ["/usr/local/bin/consul", "agent",
        "-data-dir=/var/lib/consul",
        "-config-dir=/etc/consul",
        "-rejoin",
        "-retry-join", "{{ .CONSUL }}",
        "-retry-max", "10",
```

```
            "-retry-interval", "10s"],
            "restarts": "unlimited"
          }
        {{ end }}]
      }
```

This JSON configuration file is very easy to understand. First, it defines where we can find our Consul container and which command we want to run on the ContainerPilot preStart event. In `services`, you can define all the services you want to declare that the current container is running. On the `backends`, you can define all the services you are listening for changes. In our case, we are listening for changes to services called `microservice_base_nginx` (the `BACKEND` variable is defined on the `docker-compose.yml`). If something changes on Consul on these services, we will execute the `onChange` command in the container.

For a more information about ContainerPilot, you can visit the official page, that is, `https://www.joyent.com/containerpilot`.

It's time to create the `microservices/base/php-fpm/scripts/reload.sh` file with the following content:

```
#!/bin/bash

SERVICE_NAME=${SERVICE_NAME:-php-fpm}
CONSUL=${CONSUL:-consul}
preStart() {
  echo "php-fpm preStart"
}

onChange() {
  echo "php-fpm onChange"
}

help() {
  echo "Usage: ./reload.sh preStart
  => first-run configuration for php-fpm"
  echo "      ./reload.sh onChange
  => [default] update php-fom config on
  upstream changes"
}

until
  cmd=$1
  if [ -z "$cmd" ]; then
         onChange
  fi
```

```
    shift 1
    $cmd "$@"
    [ "$?" -ne 127 ]
  do
    onChange
    exit
  done
```

Here, we created a dummy script, but it is up to you to adapt it to your requirements. For example, it can be changed to `run execute consul-template` and rebuild the NGINX configuration once ContainerPilot fires the script. We will be explaining a more complex script later.

We have our base `php-fpm` container ready, but our basic environment can't be complete without a web server. We will be using NGINX, a very light and powerful reverse proxy and web server.

The way we will build our NGINX server is very similar to the `php-fpm`, so we will only explain the differences.

Remember that all the files are available in our GitHub repository.

We will add a new service definition for NGINX to the `docker-compose.yml` file and link it to our `autodiscovery` service and also to our `php-fpm`:

```
microservice_base_nginx:
  build: ./microservices/base/nginx/
  links:
    - autodiscovery
    - microservice_base_fpm
  environment:
    - BACKEND=microservice_base_fpm
    - CONSUL=autodiscovery
  ports:
    - 8080:80
```

In our `microservices/base/nginx/config/containerpilot.json`, we now have a new option `telemetry`. This config setting allows us to specify a remote telemetry service used to collect stats from our service. Having this kind of service included in our environment allows us to see how our containers are performing:

```
"telemetry": {
  "port": 9090,
  "sensors": [
    {
      "name": "nginx_connections_unhandled_total",
      "help": "Number of accepted connnections that were not handled",
      "type": "gauge",
      "poll": 5,
      "check": ["/usr/local/bin/sensor.sh", "unhandled"]
    },
    {
      "name": "nginx_connections_load",
      "help": "Ratio of active connections (less waiting) to
      the maximum
      worker connections",
      "type": "gauge",
      "poll": 5,
      "check": ["/usr/local/bin/sensor.sh", "connections_load"]
    }
  ]
}
```

As you can see, we use a bespoke bash script to obtain the container stats, and the content of our `microservices/base/nginx/scripts/sensor.sh` script is as follows:

```
#!/bin/bash
set -e

help() {
  echo 'Make requests to the Nginx stub_status endpoint and
  pull out metrics'
  echo 'for the telemetry service. Refer to the Nginx docs
  for details:'
  echo 'http://nginx.org/en/docs/http/ngx_http_stub_status_module.html'
}

unhandled() {
  local accepts=$(curl -s --fail localhost/nginx-health | awk 'FNR == 3
  {print $1}')
  local handled=$(curl -s --fail localhost/nginx-health | awk 'FNR == 3
  {print $2}')
  echo $(expr ${accepts} - ${handled})
```

```
}

connections_load() {
  local scraped=$(curl -s --fail localhost/nginx-health)
  local active=$(echo ${scraped}
  | awk '/Active connections/{print $3}')
  local waiting=$(echo ${scraped} | awk '/Reading/{print $6}')
  local workers=$(echo $(cat /etc/nginx/nginx.conf | perl -n -e'/
  worker_connections *(\d+)/ && print $1') )
  echo $(echo "scale=4; (${active} - ${waiting}) / ${workers}" | bc)
}

cmd=$1
if [ ! -z "$cmd" ]; then
  shift 1
  $cmd "$@"
  exit
fi

help
```

This bash script gets some `nginx` stats that we will be sending to our `telemetry` server with ContainerPilot.

Our `microservices/base/nginx/scripts/reload.sh` is a little more complex than the one we created before for `php-fpm`:

```
#!/bin/bash
SERVICE_NAME=${SERVICE_NAME:-nginx}
CONSUL=${CONSUL:-consul}

preStart() {
  consul-template \
        -once \
        -dedup \
        -consul ${CONSUL}:8500 \
        -template "/etc/nginx/nginx.conf.ctmpl:/etc/nginx/nginx.conf"
}

onChange() {
  consul-template \
        -once \
        -dedup \
        -consul ${CONSUL}:8500 \
        -template "/etc/nginx/nginx.conf.ctmpl:/etc/nginx/
        nginx.conf:nginx -s reload"
}
```

```
help() {
  echo "Usage: ./reload.sh preStart
  => first-run configuration for Nginx"
  echo "      ./reload.sh onChange  => [default] update Nginx config on
  upstream changes"
}

until
  cmd=$1
  if [ -z "$cmd" ]; then
         onChange
  fi
  shift 1
  $cmd "$@"
  [ "$?" -ne 127 ]
do
   onChange
   exit
done
```

As you can see, we use `consul-template` to rebuild our NGINX config on the startup or when ContainerPilot detects a change in the list of backend services we will be monitoring. This behavior allows us to stop sending requests to unhealthy nodes.

At this point, we have our base environment ready and we are ready to test it with a simple `$ docker-compose up`. We will be using all these pieces to create bigger and more complex services. In the upcoming chapters, we will be adding the telemetry service or a data storage among others.

Frameworks for microservices

A framework is a skeleton that we can use for sofware development. Using a framework will help us use standard and robust patterns in our application, making it more stable and well known by other developers. PHP has many different frameworks you can use in your daily work. We will see some standards used on the most common frameworks so that you can pick the best for your project.

PHP-FIG

For years, the PHP community has been working on their own projects and following their own rules. Thousands of different projects with different developers have been released since the first years of PHP, and none followed any common standards.

This was a problem for PHP developers, firstly because there was no way of knowing if the steps they were following to build applications were the correct ones. Only their own experience and the Internet could help the developer guess if their code was written properly and if it would be readable by other developers in the future.

Secondly, the developers felt that they were trying to reinvent the wheel. Developers were making the same existing applications for their projects because there was no standard to fit third-party application into their projects.

In 2009, the **PHP Framework Interoperability Group** (**PHP-FIG**) was born with the main goal of creating a unique standard for development in PHP. PHP-FIG is a big community of members that works on the **PHP Standards Recommendation** (**PSR**), discussing what the best way to use the PHP language is.

PHP-FIG is supported by big projects such as Drupal, Magento, Joomla!, Phalcon, CakePHP, Yii, Zend, and Symfony and that is the reason the PSRs they propose are implemented by the PHP frameworks.

Some standards, such as PSR-1 and PSR-2, are about the use of code and its style (using tabs or spaces, opening tags of PHP, using camelCasing or filenames) and others are about the autoloading (PSR-0 and then PSR-4). Since PHP 5.3, namespaces were included and it was the most important thing to implement `autoloading`.

The `autoloading` was possibly one of the most important improvements to PHP. Before PHP-FIG, frameworks had their own methods to implement the autoloading, their own way to format them, initialize them, and name them, and each one was different, so it was a disaster (Java already solved this problem using its beans system).

Finally, Composer implemented autoloading, which was written by PHP-FIG. So, developers don't need to worry about `require_once()`, `include_once()`, `require()`, or `include()` anymore.

You can find more information about PHP-FIG at `http://www.php-fig.org/`.

PSR-7

In this book, we will use the **PHP Standard Recommendation 7** (**PSR-7**). It is about the HTTP messages interfaces. This is the essence of web development; it is the way to communicate with a server. An HTTP client or web browser sends an HTTP request message to a server and it replies with an HTTP response message.

These kinds of messages are hidden from normal users, but developers need to know the structure to manipulate them. The PSR-7 talks about the recommended ways to manipulate them, doing it simply and clearly.

We will use HTTP messages to communicate with microservices, so it is necessary to know how they work and what their structure is.

An HTTP request has the following structure:

```
POST /path HTTP/1.1 Host: example.com
foo=bar&baz=bat
```

In the request, there are all the necessary things to allow the server to understand the request message and to be able to reply with a response. In the first line, we can see the method used for the request (`GET`, `POST`, `PUT`, `DELETE`), the request target, and the HTTP protocol version, and then one or more HTTP headers, an empty line, and the body.

And the HTTP response will look like this:

```
HTTP/1.1 200 OK Content-Type: text/plain
```

This is the response body. The response has the HTTP protocol version, such as the request and the HTTP status code, followed by a text to describe the code. You will find all the available status codes in the next chapters. The rest of the lines are like the request: one or more HTTP headers, an empty line, and then the body.

Once we know the structure of HTTP request messages and HTTP response messages, we will understand what the recommendations of PHP-FIG on PSR-7 are.

Messages

Any message is an HTTP request message (RequestInterface) or an HTTP response message (ResponseInterface). They extend MessageInterface and may be implemented directly. Implementors should implement RequestInterface and ResponseInterface.

Headers

Case-insensitive for header field names:

```
$message = $message->withHeader('foo', 'bar');
echo $message->getHeaderLine('foo');
// Outputs: bar
echo $message->getHeaderLine('FoO');
// Outputs: bar
```

Headers with multiple values:

```
$message = $message ->withHeader('foo', 'bar') ->
withAddedHeader('foo', 'baz');
$header = $message->getHeaderLine('foo');
// $header contains: 'bar, baz'
$header = $message->getHeader('foo'); // ['bar', 'baz']
```

Host header

Usually, the host header is the same as the host component of the URI and the host used when establishing the TCP connection, but it can be modified.

`RequestInterface::withUri()` will replace the request host header.

You can keep the original state of the host by passing true for the second argument; it will keep the host header unless the message does not have a host header.

Streams

StreamInterface is used to hide the implementation details when a stream of data is read or written.

Streams expose their capabilities using the following three methods:

```
isReadable()
isWritable()
isSeekable()
```

Request targets and URIs

A request target is in the second segment of the request line:

```
origin-form
absolute-form
authority-form
asterisk-form
```

The `getRequestTarget()` method will use the URI object to make the origin-form (the most common request-target).

Using `withRequestTarget()`, the user can use the other three options. For example, an asterisk-form, as illustrated:

```
$request = $request
->withMethod('OPTIONS')
->withRequestTarget('*')
->withUri(new Uri('https://example.org/'));
```

The HTTP request will be as follows:

```
OPTIONS * HTTP/1.1
```

Server-side requests

RequestInterface gives the general representation for an HTTP request, but the server-side requests needs to be processed into an account **Common Gateway Interface** (**CGI**). PHP provides simplification via its superglobals:

```
$_COOKIE
$_GET
$_POST
$_FILES
$_SERVER
```

ServerRequestInterface provides an abstraction around these superglobals to reduce coupling to the superglobals by consumers.

Uploaded files

The `$_FILES` superglobal has some well-known problems when working with arrays or file inputs. For example, with the input name `files`, submitting files[`0`] and files[`1`], PHP will be represented like this:

```
array(
  'files' => array(
    'name' => array(
      0 => 'file0.txt',
      1 => 'file1.html',
    ),
    'type' => array(
      0 => 'text/plain',
      1 => 'text/html',
    ),
/* etc. */    ), )
```

The expected representation is as follows:

```
array(
  'files' => array(
    0 => array(
    'name' => 'file0.txt',
    'type' => 'text/plain',
  /* etc. */        ),
  1 => array(
    'name' => 'file1.html',
    'type' => 'text/html',
    /* etc. */
  ),
), )
```

So, the customers need to know these kinds of problems and write code to fix the given data.

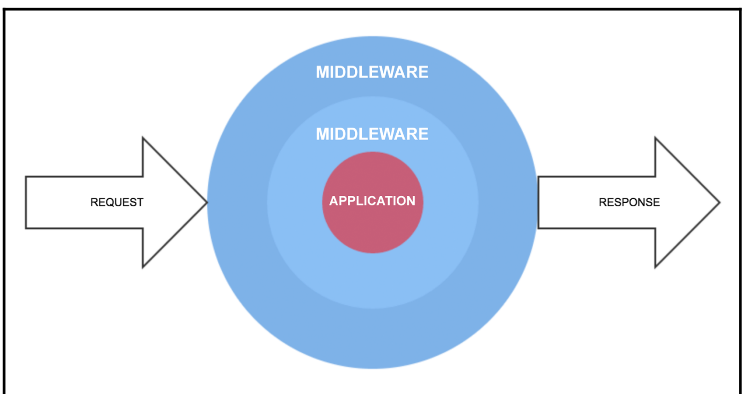
`getUploadedFiles()` provides the normalized structure for consumers.

You can find more detailed information and the interfaces and classes that we discussed at `http://www.php-fig.org/psr/psr-7/`.

Middleware

A middleware is a *mechanism to filter the HTTP requests on an application*; it allows you to add additional layers to the business logic. It executes before and after the piece of code we want to reach to handle input and output communication. The middleware uses the recommendations on PSR-7 to modify the HTTP requests. This is the reason PSR-7 and middleware are united.

The most common example of middleware is on the authentication. In an application where it is necessary to log in to get user privileges, the middleware will decide if the user can see specific content of the application:



In the preceding image, we can see a typical PSR-7 HTTP **REQUEST** and **RESPONSE** with two **MIDDLEWARE**. Usually, you will see the middleware implementations as lambda (λ).

Now, we will take a look at some examples of typical implementations of middlewares:

```
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;
function (ServerRequestInterface $request, ResponseInterface $response,
callable $next = null)
{
  // Do things before the program itself and the next middleware
  // If exists next middleware call it and get its response
  if (!is_null($next)) {
  $response = $next($request, $response);     }
  // Do things after the previous middleware has finished
  // Return response
  return $response;
}
```

The `request` and `response` are the objects, and the last param `$next` is the name of the next middleware to call. If the middleware is the last one, you can leave it empty. In the code, we can see three different parts: the first one is to modify and do things before the next middleware, in the second one the middleware calls the next middleware (if it exists), and the third one is to modify and do things after the previous middleware.

It is a good practice to see the code before and after the `$next($request, $response)` form as onion layers around the middleware, but it is necessary to be on the ball regarding the order of execution of the middlewares.

Another good practice is looking at the real application (the part of code the middlewares reach, usually a controller function) as a middleware too, because it receives a request and a response and it has to return a response, but this time without the next param because it is the last one; however, the execution continues after this. This is the reason we have to look at the end code in the same way as the last middleware.

We will see a complete example to understand how you can use it on your application based on microservices. As we saw in the preceding image, there were two middlewares, we will call them first and second, and then the end function will be called endfunction:

```
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;
$first = function (ServerRequestInterface $request,
ResponseInterface $response, callable $next)
{
  $request = $request->withAttribute('word', 'hello');
  $response = $next($request, $response);
  $response = $response->withHeader('X-App-Environment',
  'development');
  return $response;
} class Second {
  public function __invoke(ServerRequestInterface $request,
  ResponseInterface $response, callable $next)
  {
    $response->getBody()->write('word:'.
    $request->getAttribute('word'));
    $response = $next($request, $response);
    $response = $response->withStatus(200, 'OK');
    return $response;
  }
}
$second = new Second; $endfunction = function (
ServerRequestInterface $request, ResponseInterface $response)
{
  $response->getBody()->write('function reached');
  return $response;
}
```

Every framework has its own middleware handler, but each one works very similarly. The middleware handler is used to manage the stack, so you can add more middleware on it and they will be called sequentially. This is a generic middleware handler:

```
class MiddlewareHandler {
public function __construct()
{ //this depends the framework and you are using
    $this->middlewareStack = new Stack;
    $this->middlewareStack[] = $first;
    $this->middlewareStack[] = $second;
    $this->middlewareStack[] = $endfunction;
}
... }
```

The execution track will be something like this:

1. The user requests a specific path to the server. For example, `/`.
2. The `first` middleware is executed adding `word` equals to `hello`.
3. The `first` middleware sends the execution to `second`.
4. The `second` middleware adds the sentence `word: hello`.
5. The `second` middleware sends the execution to the `end` function.
6. The `end` function adds the sentence `function reached` and finishes its own job.
7. The execution continues by the `second` middleware and this one sets the HTTP status 200 to response.
8. The execution continues by the `first` middleware and this one adds a custom header.
9. The response is returned to the user.

Of course, if you get an error during the middleware execution, you can manage it using the following common code:

```
try { // Things to do }
catch (\Exception $e) {
  // Error happened return
  $response->withStatus(500);
}
```

The response will be sent immediately to the user without ending the entire execution.

Available frameworks

There are hundreds of frameworks available to develop your application, but when you need one to be used to make microframeworks, it is necessary to look for some characteristics:

- A microframework able to process the maximum requests per second
- Lightweight in terms of memory
- If it is possible, with a great community developing applications for that framework

Now we will look at possibly the five most-used frameworks at the moment. Finding the best framework is not a unique opinion, every developer has their own opinion, so let me introduce you to the following ones:

Framework	Request per second	Peak memory
Phalcon 2.0	1,746.90	0.27
Slim 2.6	880.24	0.47
Lumen 5.1	412.36	0.95
Zend Expressive 1.0	391.97	0.80
Silex 1.3	383.66	0.86

Source: *PHP Framework Benchmark*. The project attempts to measure minimum overhead PHP frameworks in the real world.

Phalcon

Phalcon is a popular framework, it gained fame because its speed made it the fastest framework. Phalcon is very optimized and modular; in other words, it only uses the things that you need or want, without adding extra things that you won't use.

The documentation is excellent, but its disadvantage is that the community is not as big as Silex or Slim, so the third-parties' community is small and it is sometimes a little difficult to find fast solutions when you have issues.

Phalcon ORM is based on the C language. This is really important if you are developing a microservice based on databases.

To sum up, it is the best framework; however, it is not recommended for beginners.

You can visit the official website at `https://phalconphp.com`.

Slim framework

Slim is one of the fastest micro RESTful frameworks available. It provides you with every feature that a framework should have. Also, Slim framework has a very big community: you can find a lot of resources, tutorials, and documentation on the Internet because there are a lot of developers using it.

Since the release of version 3, the framework has a better architecture, making it better in terms of overall architecture and security. This new version is a little slower than version 2, but all the introduced changes make this framework suitable for projects of all sizes.

The documentation is not bad, but it could be better. It is too short.

It is a good microframework for beginners.

Refer to the official website at `http://www.slimframework.com/`.

Lumen

Lumen is one of the fastest micro RESTful frameworks made by Laravel. This microframework was specially made to work with ultra fast microservices and APIs. Lumen is really fast, but there are some microframeworks that are faster, such as Slim, Silex, and Phalcon.

This framework became famous because it was pretty similar to CodeIgniter syntax, and it is very easy to use, so maybe this is why Lumen is the best microframework to start working with microservices using a microframework. Also, Lumen has very good and clear documentation; you can find it at `https://lumen.laravel.com/docs`. If you use this framework, you can start to work in little time as the setup is really fast.

Another advantages of Lumen is that you can start working with it and then, if you need the complete Laravel in the future, it is very easy to transform and update the framework into Laravel.

Please note that Lumen still enforces an application structure (convention over configuration) that might limit your options when you design your application.

If you are going to use Lumen, it is because you have used Laravel and you liked it; if you do not like Laravel, Lumen is not the best solution for you.

You can visit the official website at `https://lumen.laravel.com/`.

Zend Expressive

Lumen is the equivalent of Laravel and Zend Expressive is for Zend Framework. It is a microframework built to make microservices and is prepared to be used exclusively following PSR-7 and based on middleware.

It is possible to set it up in a few minutes and you have all the advantages of Zend Framework on it in terms of community. Also, being a product of Zend is a synonym of quality and security.

It comes with a minimal core and you can choose what components you want to include. It has very good flexibility and ability to extend it too.

Visit the official website at `https://zendframework.github.io/zend-expressive/`.

Silex (based on Symfony)

Silex is also a very good micro RESTful PHP framework. It is one of the five fastest microframeworks and currently, it is one of the best known because the Silex community is one of the bigger ones and they develop really good third parties, so developers have many solutions for their projects.

The community and its connection to Symfony guarantee stable implementation with many available resources. Also, the documentation is really good, and this microframework is specially good for large projects.

The advantages of Silex and Slim Framework are pretty similar; maybe the competition made them better.

Refer to the official website at `http://silex.sensiolabs.org/`.

Summary

In this chapter we talked about what we are going to build in this book as an example application. We also showed you how you can set up your development machine using Docker, and we even talked about the different microframeworks you can use .In the next chapter, we will learn how to go about designing our application and different types of microservices patterns.

3
Application Design

In the previous chapters, we looked at a brief description of the application we will build. It is now time to give you an in-depth look of the overall project.

Microservices structure

We want to build a geolocalization application and we chose to create it like a game so that it is more fun and easier to understand. Feel free to adapt the example to any other idea, for example, a tourism application with geolocalization embedded.

Our game will use geolocalization to find different secrets all around the world (or in a specific geographic area if you want a smaller map). The backend system will generate new secrets and place them randomly on our map, allowing the users to explore their environment to find them. As a player of our game, you will collect the different secrets and store them in your *wallet*, which is where you will find more information about each of them.

To make our game more fun, we will have a battle engine. While you are discovering our *secret world*, you can battle against other players to steal his/her *secrets*. The battle engine will be a simple one–just throw a dice and the highest score wins the battle.

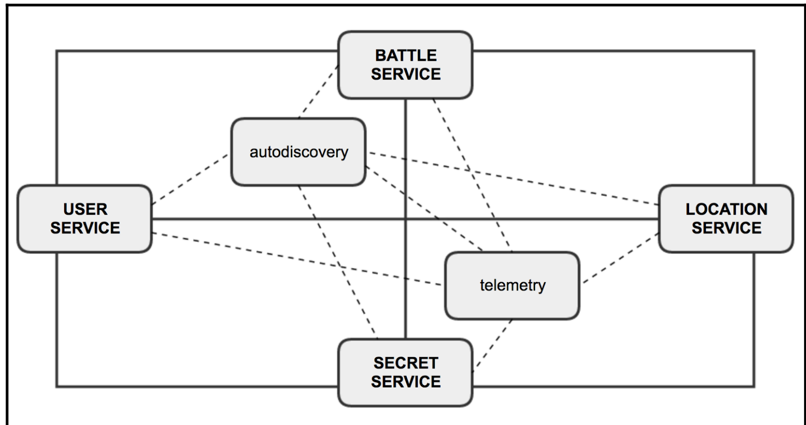
A project of this kind cannot be completed without other services, such as a user/player management system among others.

As a developer, you start with a specification and you try to decompose it into smaller parts. From our little description, we can start defining our microservices and their responsibilities as follows:

- **User service**: The main responsibility of this service is user registration and management. To keep the example small, we will also add extra functionalities, such as user notifications and secrets wallet management.
- **Battle service**: This service will be responsible for the users battle, keeping a record of each battle and moving secrets from the loser wallet to the winner.
- **Secret service**: This is one of the core services for our game because it will be responsible for all the secrets stuff.
- **Location service**: To add an extra layer of complexity, we decided to create a service to manage any task related to locations. The main responsibility is to know where everything is located; for example, if the user service needs to know if there are other players in the area, sending a message with the geolocalization to this service, the response will tell the User Service who is in the area.

Note that we are not only creating services for our game, but we will be using other supporting services to make everything work smoothly.

The following diagram describes the communication paths between our different services. Every service will be able to talk to other services so that we can compose bigger and more complex tasks. The following diagram depicts connections between our microservices:



Microservice patterns

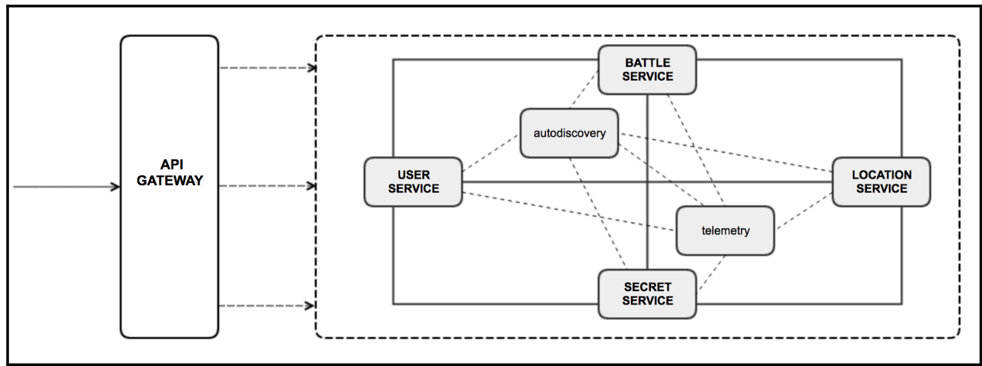
A design pattern is a reusable solution to a recurrent problem in a real-world application development. These solutions have a proven track record of success and they are widely used, so adding them to our project will make our software more stable and reliable.

We are building a microservice application and because we want it to be as stable and reliable as possible, we will use some microservice patterns, such as: API gateway, service discovery and registry, and shared database or database per service.

API gateway

We will have a frontend for the users to register and interact with our application and it will be the main client of our microservices. Also, we are planning to have native mobile applications in the future. Having different clients using our application can create headaches for us because their use of our microservices can be very different.

To unify the way any client uses our microservices, we will be adding an extra layer–an API gateway. This API gateway becomes the single entry point for any client (for example, browser and native application). In this layer, our gateway can handle the request in two ways: some requests are simply proxied and others are fanned out to multiple services. We can even use this API gateway as a security layer, checking whether each request from the client is allowed to use our microservices or not:



Assets' requests

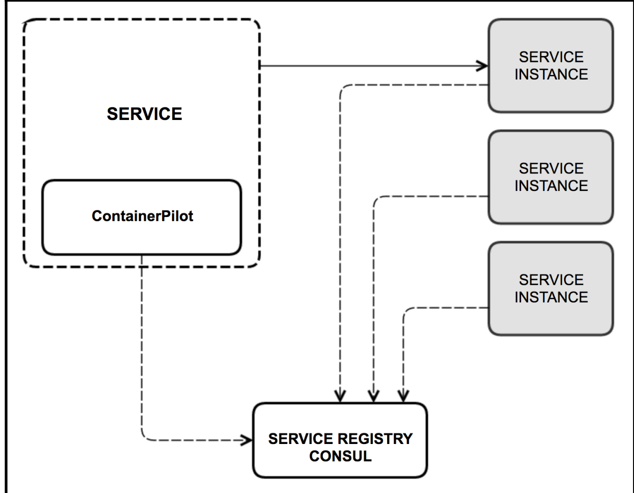
Having an API gateway has numerous benefits, among which we can highlight the following ones:

- Our application will have a single point of access, removing the problem of clients needing to know where each microservice is.
- We have a bigger control of how our services are used, and we can even provide custom endpoints for specific clients.
- It reduces the number of requests/roundtrips. With one single round-trip, a client can retrieve data from multiple services.

Service discovery and registry

Our services will need to call other services. On monolithic applications, the solution is very simple–we can call methods or use procedure calls. We are building a microservices application running in containers, so there is no easy way of knowing where some service is located. Our containers infrastructure is very flexible and we need to build a service discovery system.

Each of our services will obtain the location of all the other linked services by querying our service registry (a place where we store information about all our services using Consul). Our registry will know the locations of each service instance. The following figure shows the autodiscovery pattern:



To achieve this, we will use different tools:

- **Consul**: This is our service registry with loads of features, such as clustering support, among others.
- **Fabio**: This is a reverse proxy built on Go and has a deep integration with Consul. What we like about this proxy is the easy connection with Consul and its ability to do blue-green deploys. Another interesting tool you can try is Træfik.
- **NGINX**: A powerful HTTP server and reverse proxy, this is a very well-known tool for most of the web developers and was chosen due to its performance and low memory footprint. We will be using Fabio and NGINX as reverse proxies indistinctly.
- **ContainerPilot**: This is a small tool written in Go. We will use this software to register our services in Consul, send stats of our containers to a centralized telemetry system, send health checks to Consul, and detect changes in other services. We will create some kind of auto-healing system with this tool.

Shared database or database per service

In one way or another, an application generates data that we need to store. In a monolithic application, there is no doubt that all the data is stored at the same place. The problem is when you are dealing with a microservice application and there is no easy response. Each application domain is unique, so there is no rule of thumb to solve the problem; you need to analyze your data and decide if you want to store all the data in a shared store, if each service has its own data store, or a mixture.

In our sample application, we will cover both the approaches but let's explain the benefits of each option.

Database per service

In this approach, we keep each microservice's persistent data private to that service, the data is only accesible via its API and has numerous benefits:

- It makes the services loosely coupled; you can make changes to the battle service without impacting the user service, for example.
- It increases the flexibility of our application due to the fact that the data can only be accessed by its API; we can use different storage engines. For example, we can use a relational database in our user service and a NoSQL in the Location service.

Of course, this solution has a few drawbacks, the difficulty of joining data shared between different services being the most notable problem.

Shared database

This approach works like a database in a monolithic application–all the data is stored in the same engine. The main benefit is the simplicity of having everything in one place.

This simplicity has some drawbacks; we highlight the following ones among them:

- Any database change can break or impact other services
- Results in a less flexible application as you are using the same engine for all the data
- If the data store is down, all the services that use the shared database will note the problem

As a developer, your job is to find the best solution for each problem you need to solve. You need to decide how you are going to store the application data, always keeping in mind the benefits and drawbacks of each option.

RESTful conventions

Representational State Transfer is the name of the method used to communicate with the APIs. As the name suggests, it is *stateless*; in other words, the services do not keep the data transferred, so if you call a microservice sending data (for example, a username and a password), the microservice will not remember the data next time you call it. The state is kept by the client, so the client needs to send the state every time the microservice is called.

A good example of this is when a user is logged in and the user is able to call a specific method, so it is necessary to send the user credentials (username and password or token) every time.

The concept of a Rest API is not a service anymore; instead of that, it is like a resource container available to be communicated by identifiers (URIs).

In the following lines, we will define some interesting conventions about APIs. It is important to know these kinds of tips because you should do things as you would like to find them when you are working on an API. In other words, writing an API is like writing a book for yourself–it will be read by developers like you, so the perfect functionality is not the only important thing, the friendly way to talk is important too.

Creating a RESTful API will be easier for you and the consumers if you follow some conventions in order to make them happy. I have been using some recommendations on my RESTful APIs and the results were really good. They help to organize your application and its future maintenance needs. Also, your API consumers will thank you when they enjoy working with your application.

Security

Security in your RESTful API is important, but it is especially important if your API is going to be consumed by people you do not know, in other words, if it is going to be available to everybody.

- Use SSL everywhere–it is important for the security of your API. There are many public places without an SSL connection and it is possible to sniff packages and get other people's credentials.
- Use token authentication, but it is mandatory to use SSL if you want to use a token to authenticate the users. Using a token avoids sending entire credentials every time you need to identify the current user. If it is not possible, you can use OAuth2.
- Provide response headers useful to limit and avoid too many requests by the same consumer. One of the problems with big companies is the traffic or even people trying to do bad things with your API. It is good to have some kind of method to avoid these kinds of problems.

Standards

Bit by bit, more standards for PHP and microservices are appearing. As we saw in the last chapter, there are groups, such as PHP-FIG, trying to establish them. Here are some tips to make your API more standard:

- Use JSON everywhere. Avoid using XML; if there is a standard for RESTful APIs, it is JSON. It is more compact and can be easily loaded in web languages.
- Use camelCase instead of snake_case; it is easier to read.
- Use HTTP status code errors. There are standard statuses for each situation, so use them to avoid explaining every response of your API better.

- Include the versioning in the URL, do not put it on the header. The version needs to be in the URL to ensure browser explorability of the resources across versions.
- Provide a way to override the HTTP method. Some browsers only allow `POST` and `GET` requests, so it will be good to allow a `X-HTTP-Method-Override` header to override `PUT`, `PATCH`, and `DELETE`.

Consumer amenities

The consumers of your API are the most important, so you need to provide useful, helpful, and friendly ways to make the developer's job easier. Develop the methods thinking about them:

- Limit the response data. The developer will not need all the available data, so you can limit the response using a filter for the fields to return.
- Use query parameters to filter and sort results. It will help you simplify your API.
- Remember that your API is going to be used by different developers, so look after your documentation–it needs to be really clear and friendly.
- Return something useful on the `POST`, `PATCH`, and `PUT` requests. Avoid making the developer call to the API too many times to get the required data.
- It is good to provide a way to autoload related resource representations in the response. It would be helpful for the developers in order to avoid requesting the same thing many times to get all the necessary data. It is possible to do this by including filters to define specific parameters in the URL.
- Make the pagination using link headers, then the developers will not need to make the links on their own.
- Include response headers that facilitate caching. HTTP has included a framework to do this just by adding some headers.

There are a lot more tips, but these ones are enough for the first approach to RESTful conventions. In the subsequent chapters, we will see examples of these RESTful conventions and explain how they should be used better.

Caching strategy

> *"There are only two hard things in Computer Science: cache invalidation and naming things."*
>
> *– Phil Karlton*

A cache is a component that stores data temporarily so that future requests for that data can be served faster. This temporal storage is used to shorten our data access times, reduce latency, and improve I/O. We can improve the overall performance using different types of caches in our microservice architecture. Let's take a look at this subject.

General caching strategy

To maintain the cache, we have algorithms that provide instructions which tell us how the cache should be maintained. The most common algorithms are as follows:

- **Least Frequently Used (LFU)**: This strategy uses a counter to keep track of how often an entry is accessed and the element with the lowest counter is removed first.
- **Least Recently Used (LRU)**: In this case, the recently-used items are always near the top of the cache and when we need some space, elements that have not been accessed recently are removed.
- **Most Recently Used (MRU)**: The recently-used items are removed first. We will use this approach in situations where older items are more commonly accessed.

The perfect time to start thinking about your cache strategy is when you are designing each of the microservices required by your app. Every time your service returns data, you need to ask to yourself some questions:

- Are we returning sensible data we can't store at any place?
- Are we returning the same result if we keep the input the same?
- How long can we store this data?
- How do we want to invalidate this cache?

You can add a cache layer at any place you want in your application. For example, if you are using Percona/MySQL/MariaDB as a data storage, you can enable and set up the query cache correctly. This little setup will give your database a boost.

You need to think about cache even when you are coding. You can do lazy loading on objects and data or build a custom cache layer to improve the overall performance. Imagine that you are requesting and processing data from an external storage, the requested data can be repeated several times in the same execution. Doing something similar to the following piece of code will reduce the calls to your external storage:

```
<?php
class MyClass
{
   protected $myCache = [];

   public function getMyDataById($id)
   {
       if (empty($this->myCache[$id])) {
           $externalData = $this->getExternalData($id);
           if ($externalData !== false) {
               $this->myCache[$id] = $externalData;
           }
       }

            return $this->myCache[$id];
   }
}
```

Note that our examples omit big chunks of code, such as namespaces or other functions. We only want to give you the overall idea so that you can create your own code.

In this case, we will store our data in the `$myCache` variable every time we make a request to our external storage using an ID as the key identifier. The next time we request an element with the same ID as a previous one, we will get the element from `$myCache` instead of requesting the data from the external storage. Note that this strategy is only successful if you can reuse the data in the same PHP execution.

In PHP, you have access to the most popular cache servers, such as **memcached** and **Redis**; both of them store their data in a key-value format. Having access to these powerful tools will allow us to increase the performance of our microservices application.

Let's rebuild our preceding example using `Redis` as our cache storage. In the following piece of code, we will assume that you have a `Redis` library available in your environment (for example, phpredis) and a `Redis` server running:

```
<?php
class MyClass
{
  protected $myCache = null;
```

```
    public function __construct()
    {
      $this->myCache  = new Redis();
      $this->myCache->connect('127.0.0.1',  6379);
    }

    public function getMyDataById($id)
    {
      $externalData = $this->myCache->get($id);
        if ($externalData === false) {
          $externalData = $this->getExternalData($id);
          if ($externalData !== false) {
            $this->myCache->set($id, $externalData);
          }
        }

        return $externalData;
    }
  }
```

Here, we connected to the Redis server first and adapted the `getMyDataById` function to use our new Redis instance. This example can be more complicated, for example, by adding the dependence injection and storing a JSON in the cache, among other infinite options. One of the benefits of using a cache engine instead of building your own is that all of them come with a lot of cool and useful features. Imagine that you want to keep the data in cache for only 10 seconds; this is very easy to do with Redis–simply change the set call with `$this->myCache->set($id, $externalData, 10)` and after ten seconds your record will be wiped from the cache.

Something even more important than adding data to the cache engine is invalidating or removing the data you have stored. In some cases, it is fine to use old data but in other cases, using old data can cause problems. If you do not add a TTL to make the data expire automatically, ensure that you have a way of removing or invalidating the data when it is required.

Keep this example and the previous one in mind, we will be using both strategies in our microservice application.

As a developer, you don't need to be tied to a specific cache engine; wrap it, create an abstraction, and use that abstraction so that you can change the underlying engine at any point without changing all the code.

This general caching strategy can be used in any scope of your application–you can use it inside the code of your microservice or even between microservices. In our application example, we will deal with *secrets*; their data doesn't change very often, so we can store all this information on our cache layer (Redis) the first time they are accessed.

Future petitions will obtain the secrets' data from our cache layer instead of getting it from our data storage, improving the performance of our app. Note that the service that retrieves and stores the *secrets* data is the one that is responsible for managing this cache.

Let's see some other caching strategies that we will be using in our microservices application.

HTTP caching

This strategy uses some HTTP headers to determine whether the browser can use a local copy of the response or it needs to request a fresh copy from the origin server. This cache strategy is managed outside your application, so you don't have much control over it.

Some of the HTTP headers we can use are as listed:

- **Expires**: This sets a time in the future when the content will expire. When this point in the future is reached, any similar requests will have to go back to the origin server.
- **Last-modified**: This specifies the last time that the response was modified; it can be used as part of your custom validation strategy to ensure that your users always have fresh content.
- **Etag**: This header tag is one of the several mechanisms that HTTP provides for web cache validation, which allows a client to make conditional requests. An Etag is an identifier assigned by a server to a specific version of a resource. If the resource changes, the Etag also changes, allowing us to quickly compare two resource representations to determine if they are the same.
- **Pragma**: This is an old header, from the HTTP/1.0 implementation. HTTP/1.1 Cache-control implements the same concept.
- **Cache-control**: This header is the replacement for the expires header; it is well supported and allows us to implement a more flexible cache strategy. The different values for this header can be combined to achieve different caching behaviors.

The following are the available options:

- **no-cache**: This says that any cached content must be revalidated on each request before being sent to a client.
- **no-store**: This indicates that the content cannot be cached in any way. This option is useful when the response contains sensitive data.
- **public**: This marks the content as public and it can be cached by the browser and any intermediate caches.
- **private**: This marks the content as private; this content can be stored by the user's browser, but not by intermediate parties.
- **max-age**: This sets the maximum age that the content may be cached before it must be revalidated. This option value is measured in seconds, with a maximum of 1 year (31,536,000 seconds).
- **s-maxage**: This is similar to the max-age header; the only difference is that this option is only applied to intermediary caches.
- **must-revalidate**: This tag indicates that the rules indicated by max-age, s-maxage, or the expires header must be obeyed strictly.
- **proxy-revalidate**: This is similar to s-maxage, but only applies to intermediary proxies.
- **no-transform**: This header tells caches that they are not allowed to modify the received content under any circumstances.

In our example application, we will have a public UI that can be reached through any web browser. Using the right HTTP headers, we can avoid requests for the same assets again and again. For example, our CSS and JavaScript files won't change frequently, so we can set up an expiry date in the future and the browser will keep a copy of them; the future requests will use the local copy instead of requesting a new copy.

You can add an expires header to all `.jpg`, `.jpeg`, `.png`, `.gif`, `.ico`, `.css`, and `.js` files with a date of 123 days in the future from the browser access time in NGINX with a simple rule:

```
location ~*  .(jpg|jpeg|png|gif|ico|css|js)$ {
    expires 123d;
}
```

Static files caching

Some static elements are very cache-friendly, among them you can cache the following ones:

- Logos and non-auto generated images
- Style sheets
- JavaScript files
- Downloadable content
- Any media files

These elements tend to change infrequently, so they can be cached for longer periods of time. To alleviate your servers' load, you can use a **Content Delivery Network** (**CDN**) so that these infrequently changed files can be served by these external servers.

Basically, there are two types of CDNs:

- **Push CDNs**: This type requires you to **push** the files you want to store. It is your responsibility to ensure that you are uploading the correct file to the CDN and the pushed resource is available. It is mostly used with uploaded images, for example, the avatar of your user. Note that some CDNs can return an OK response after a push, but your file is not really ready yet.
- **Pull CDNs**: This is the lazy version, you don't need to send anything to the CDN. When a request comes through the CDN and the file is not in their storage, they get the resource from your server and it stores it for future petitions. It is mostly used with CSS, images, and JavaScript assets.

You need to have this in mind when you are designing your microservice application because you may allow your users to upload some files.

Where are you going to store these files? If they are to be public, why not use CDN to deliver these files instead of them being gutted from your servers.

Some of the well-known CDNs are CloudFlare, Amazon CloudFront, and Fastly, among others. What they all have in common is that they have multiple data centers around the world, allowing them to try to give you a copy of your file from the closest server.

By combining HTTP with static files caching strategies, you will reduce the asset requests on your server to a minimum. We will not explain other cache strategies, such as full page caching; with what we have covered, you have enough to start building a successful microservice application.

Domain-driven design

Domain-driven design (**DDD** from here on) is an approach for the development when it has complex needs. This concept is not new; it was created by Eric Evans in his book with the same title in 2004, but now it is mainstream as microservices are popular among developers and very common in huge projects.

This is happening as there is great compatibility between the microservices concepts (regarding the software architecture, dividing every functionality into services) and DDD concepts (about the bounded contexts).

Before knowing where and how we can use DDD in our microservices project, it is necessary to understand what DDD is and how it works, so let me introduce you to the main concepts as a summary of this approach.

How domain-driven design works

Evans introduced some concepts that are necessary to understand to learn how domain-driven design works:

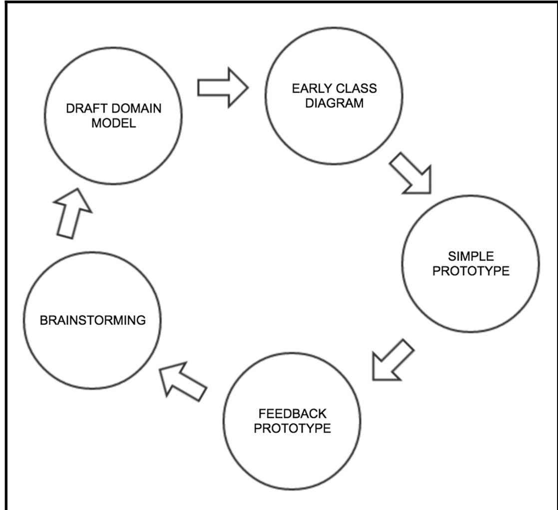
- **Context**: This is the setting in which a word or statement appears that determines its meaning.
- **Domain**: This is a sphere of knowledge (ontology), influence, or activity. The subject area to which the user applies a program is the domain of the software.
- **Model**: This is a system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain.
- **Ubiquitous language**: This is a language structured around the domain model and used by all team members to connect all the activities of the team with the software.

The **software domain** is not related to the technical terms, programming or computers in any way. In most projects, the most challenging part is to understand the business domain, so DDD suggests using a **model domain**; this is abstract, ordered, and selective knowledge reproduced in a diagram, code, or just words.

The model domain is like the roadmap to build projects with complex functionalities, and it is necessary to follow five steps to achieve it. These five steps need to be agreed on by the development team and the domain expert:

1. **Brainstorming and refinement**: There should be a communication channel between the development team and the domain expert. So, all the people in the project should be able to talk to everyone because they all need to know how the project should work.
2. **Draft domain model**: During the conversation, it is necessary to start drawing a draft of the domain model, so that it can be checked and corrected by the domain expert until they both agree.
3. **Early class diagram**: Using the draft, we can start building an early version of the class diagram.
4. **Simple prototype**: Using the draft of the early class diagram and domain model, it is possible to build a very simple prototype. Evans suggests avoiding things that are not related to the domain to ensure that the domain business was modeled properly. It can be a very simple program as a trace.
5. **Prototype feedback**: The domain expert interacts with the prototype in order to check whether all the needs are met and then the entire team will improve the model domain and the prototype.

This process will have all the iterations needed until the domain model is correct:



The model, code, and design must evolve and grow together. They cannot be unsynchronized at all. If a concept is updated on the model, it should also be updated on the code and on the design automatically, and the same goes for the rest.

A model is an abstraction system that describes selective concepts of a domain and it can be used to resolve problems related to that domain. If there is a piece of the model that is not reflected in the code, it should be removed.

Finally, the domain model is the base of the common language in a project. This common language in DDD is called **ubiquitous language** and it should have the following things:

- Class names and their functions related to the domain
- Terms to discuss the domain rules included on the model
- Names of analysis and design patterns applied to the domain model

The ubiquitous language should be used by all the members of the project, including developers and domain experts, so the developers should be able to describe all the tasks and functions.

It is absolutely necessary to use this language in all the discussions between the team, such as meetings, diagrams, or documentation, but this language was not born in the first iteration of the process, meaning that it can take many iterations of refactoring having the model, language, and code synchronized. If, for example, the developers discover that a class from the domain should be renamed, they cannot refactor this without refactoring the name on the domain model and the ubiquitous language.

The ubiquitous language, domain model, and code should evolve together as a single knowledge block.

There is controversial concept on DDD. Eric Evans says that it is necessary for the domain expert to use the same language as the team, but some people do not like this idea. Usually, the domain experts do not have knowledge of object-oriented concepts or microservices because they are too abstract for non-developers. Anyway, DDD says that if the domain expert does not understand the domain model, it is because there is something wrong with it.

There are diagrams in the domain model, but Evans suggests using text as well, because diagrams do not explain the concepts properly. Also, the diagrams should be superficial; if you want to see more details you have the code for it.

Some projects are affected by the connection between the domain model and the code. This happens because there is a division between analysis and design. The analysts make a model independent of the design and the developers cannot develop the functionalities because some information is missing. In addition, they cannot talk with the domain expert. The development team will not follow the model and, in the end, the domain model will not be updated and it will not work. Therefore, the project will not meet the requirements.

To sum up, DDD works to achieve the software development as an iterative process of refinement of the model, design, and code as a single task in a block.

Using domain-driver design in microservices

As we said before, DDD meets the microservice's needs perfectly. A common problem with microservices appears because they have decentralized data management; this has advantages but can be problematic sometimes.

The concept model between two services will be different, and it can cause problems in huge companies. For example, a user can differ depending on the service, the attributes for each service regarding the user can differ and, also, the attribute semantic can differ.

It is even more complex when, in a big company, the application evolves a lot and has updates for many years. Each service can have different attributes for the user and, generally, they do not match. So, a great way to solve this is using DDD.

As microservices do, DDD divides a complex domain into different contexts, making relationships between them and asking for the collaboration of all the members to get a ubiquitous language in a particular domain and bounded context, iterating this process until they achieve a real concept regarding the problem.

Evans suggests designing each microservice as a DDD-bounded context so that it will provide a logical boundary for microservices inside a system. Every single microservice (or team working on it) will be responsible for that part of the system, and it will give clearer and maintainable code.

Michael Plöd gave more ideas about how DDD can help microservices. There are four significant areas regarding building microservices:

- **Strategic design**: This is basically bounded context, but context maps and other patterns are important too. A context map should show all the bounded contexts of the project and their relationships with each other; it also describes the contract between them. The context map is very useful for monolithic applications wanting to move into microservices.

- **Internal building blocks**: This refers to using tactical patterns, such as aggregates, entities, or repositories, when designing the inside of a bounded context.
- **Large-scale structures**: This is used to create a structure using evolving order and responsibility layers. This is a concept in microservices too. In huge projects, it is helpful to create large-scale structures into boundary contexts. They should be designed to evolve individually.
- **Distillation**: Distilling a core domain from an already grown system is very useful when migrating a monolithic application into microservices. The most important part should be identifying and extracting the core domain, along with the iteration process of identifying a subdomain, extracting it from the core, and refactoring.

To sum up, microservices and DDD match perfectly, but it is necessary to have a larger scope and understand more than the boundary contexts.

Event-driven architecture

Event-driven architecture (**EDA**) is a pattern of architecture for applications following the tips of production, detection, consumption of, and reaction to events.

It is possible to describe an event as a change of state. For example, if a door is closed and somebody opens it, the state of the door changes from closed to opened. The service to open the door has to make this change like an event, and that event can be known by the rest of the services.

An event notification is a message that was produced, published, detected, or consumed asynchronously and it is the status changed by the event. It is important to understand that an event does not move around the application, it just happens. The term *event* is a little controversial because it usually means the message event notification instead of the event, so it is important to know the difference between the event and the event notification.

This pattern is commonly used in applications based on components or microservices because they can be applied by the design and implementation of applications. An application driven by events has event creators and event consumers or sink (they have to execute the action as soon as the event is available).

An **event creator** is the producer of the event; it only knows that the event has occurred, nothing else. Then we have the event consumers, which are the entities responsible of knowing that the event was fired. The consumers are involved in processing or changing the event.

The **event consumers** are subscribed to some kind of middleware event manager which, as soon as it receives notification of an event from a creator event, forwards the event to the registered consumers to be taken by them.

Developing applications as microservices around an architecture such as EDA allows these applications to be constructed in a way that facilitates more responsiveness because the EDA applications are, by design, ready to be in unpredictable and asynchronous environments.

The advantages of using EDA are as follows:

- **Uncoupling systems**: The creator service does not need to know the rest of the services, and the rest of the services do not know the creator. So, it allows it to uncouple the system.
- **Interaction publish/subscribe**: EDA allows many-to-many interactions, where the services publish information about some event and the services can get that information and do what is necessary with the event. So, it enables many creator events and consumer events to exchange status and respond to information in real time.
- **Asynchronous**: EDA allows asynchronous interactions between the services, so they do not need to wait for an immediate response and it is not mandatory to have a connection working while they are waiting for the response.

Event-driven architecture in microservices

Microservices are commonly used in large projects to divide their services into smaller ones. So, it is really important to have good and organized communication between them. Event-driven architecture can be used to solve the common issues of communication between microservices.

In a project based on microservices, usually every microservice communicates with each other using HTTP requests. This has some problems that we will now explain.

In our `Finding secrets` project, there is a function to create events for the users. When a new event is created, the event name and the images attached in the event form need to be sent to a service to create a video from the data received. Once the video is generated, the event will be updated and sent to the users by e-mail.

If we make HTTP requests for each service, the problem is that all the services need to know about the others. For example, the service to generate the video needs to know how to update the event once the video is generated; in other words, the service has to contain code to do this update.

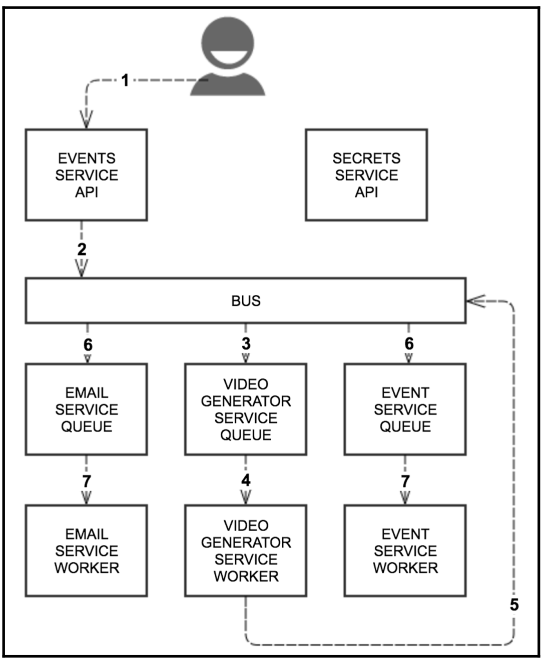
Also, this becomes more and more difficult once we add many services because it will need more communication between them. It will have more failures and the main problem is that if a microservice is down, the video will not be generated. So, using HTTP requests is not going to scale pretty well and we should use a different strategy to communicate microservices in projects like this one.

What if we do the things differently? In other words, the service to generate the video will not update the event directly and the event will not ask the video service to generate the video. So, how can we make the microservices communicate? The answer is with event-driven architecture.

To do this, we need the following things:

- A queue of events for each microservice
- All the microservices have to send the event to a centralized BUS (we can use AWS to do this)
- Every queue of microservices has to be subscribed to the centralized BUS
- Every microservice has a background worker listening to the queue of events and it will execute the necessary action when receiving an event

In the following figure, you can see the different services involved and the process flow, indicated with arrows. The following figure shows the event-driven workflow:



When we create a new event on the **EVENTS SERVICE API** (**1**), the event goes to the centralized BUS (**2**) and the corresponding worker gets the event from the centralized BUS (**3**); the rest of them just ignore the event. The event is placed in the video generator service queue and it waits to be executed by the service (**4**).

Once the video has been generated by a service worker, the service launches a new event to the centralized BUS (5). However, it will be taken by a different worker this time (6) and the res of workers will ignore this event as earlier. The worker to update the event and the worker to send the e-mail will put the event into their queues and it will be executed (**7**) doing the corresponding action for each service and they will send a new event into the centralized BUS if it is necessary.

This is a loop of events that improves the HTTP requests method for the communication between services. The advantages of using event-driven architecture are as mentioned:

- If there are any errors or exceptions on a service, the event does not get lost, it stays in the queue and it will be executed later. For example, if the service to send e-mails is down, the event to send the e-mail will be kept in the queue waiting for the service to go up again.
- The services do not need to know how to update other services. It means that the logic of the service can be isolated in each service.
- It is possible to add more microservices without impact.
- It will scale better.

Continuous integration, continuous delivery, and tools

A software project cannot be successful without a strategy for code commit or a testing/deploying workflow. Having a strategy is even more important when you work in a team. There is nothing more annoying than working on a messy project where there are no rules or nobody is accountable for the work they have done. In this section, we will explain the most common and successful development practices.

Continuous integration – CI

Continuous integration is a software development practice where all the team members integrate their work frequently. Every time new code is pushed to the shared repository, an automated build will be fired to detect any kind of integration errors as fast as possible. The main goal is to avoid long and unpredictable integrations.

What is continous integration?

Let's explain it better with a brief example of what the CI process is like. Imagine that you have our game example ready and working well in production and you have a new idea for a small feature that the users of your application will love. This new feature can be done in a few hours.

Begin by getting a copy of the current source code on your development machine; you will be using a source control system, so you only need to check out a working copy from the mainline.

Now that you have a working copy of the source, you can do whatever you need to complete the feature, add new code, create new tests, and so on. The CI practice assumes that a high part of your code will be covered by automated tests. A popular unit test suite available for PHP is PHPUnit, a simple and powerful tool that we will cover in the later chapters. Having our code tested will help us in the future steps of the process and will assure high quality in our code.

Now you have ended your new feature and it is time to launch an automated build on your development environment. This process will take the source code, check for errors, and run the automated tests. Only if the build and all the tests pass without errors, we can consider the build as good and it can be added to our repository.

The result of doing this process is that we have a stable piece of software that works properly and contains very few bugs.

Benefits of CI

The main goal of continuous integration is to reduce risk, but this is not the only benefit of adopting this development practice. Among others, we can highlight the following benefits:

- Reduced integration times
- Early bug detection due to the fact that we are pushing small changes and each change is tested again and again
- Constant availability of a stable build that we can use, for example, to make new tests, use as a demo for our customers, or even to deploy it again
- Continuous monitoring of the project quality metrics

Tools for continuous integration

As a developer, you can be worried about how to automate this process. Don't worry, in the market you have multiple ways to create and manage your CI pipeline. The best recommendation from us is, before you decide which CI software you will use in your projects, spend some time testing all your options. Some of the CI software available with an easy integration with PHP are:

- **Jenkins**: This is an open source project that is very easy to install and manage. Its versatility makes this software perhaps one of the most widely used for CI.
- **Bamboo**: This is a subscription-based software. Atlassian is well known in the development world for its productivity and development support tools. It is a nice option if you need deep integration with other Atlassian tools.
- **Travis**: This is another subscription based software with a free plan for open source projects.
- **PHP CI**: This new open source tool was built on PHP and is available to installed on your server or as a cloud-based tool.

In our sample project, we will use Jenkins and spin up a Docker container. In the meantime, you can start testing Jenkins with this simple command:

```
$ docker run -p 8080:8080 -p 50000:50000 jenkins
```

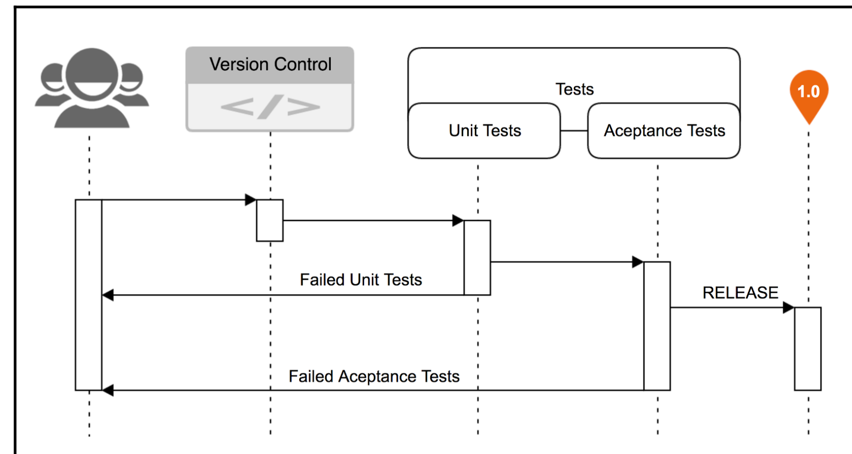
This command will create a container with the official Docker image for Jenkins and map the 8080 and 50000 from your local environment to the container. If you open your browser on `http://localhost:8080`, you will have access to the Jenkins UI.

Continuous delivery

Continuous delivery is the continuation of the continuous integration and the main goal is to be able to deploy any version of your software at any point, without a point of failure. We can achieve this by ensuring that our code is always available to be deployed and by following a continuous integration practice, we can ensure the quality and level of integration of our source.

With continuous delivery, every time we make a change to our code, this change is built, tested, and then released to a stage environment. The following diagram shows the basic workflow on a CD pipeline. As you can see, if any testing step fails, we need to start again until our code passes the tests. Working this way, we can always ensure that our project meets the highest quality standards.

The following is the diagram for continuous delivery workflow:



Benefits of continuous delivery

Continuous delivery has numerous benefits; among them, we highlight the following ones:

- **Reduced deployment risk**: We will be deploying smaller changes, so there is less space for something to go wrong and it will be easier to fix any problems. Even if we apply a deploy pattern, such as blue-green deployments, our deployment will be undetectable to our users.
- **Progress tracking**: Since not all developers and managers track the work progress in the same way, we are now deploying small releases very quickly; there is no doubt when a task is done–if it is on production, the task is done.
- **Higher quality**: With continuous delivery, we work in small batches; this allows us to get feedback from users throughout the delivery life cycle. We can even use A/B testing to test ideas before building the full feature. Having automatic testing tools in our pipeline allows the developers to discover regressions quickly and avoid the release of unstable software.
- **Faster time to market**: The integration and test phase of the traditional software life cycle can take weeks, but if we manage to automate the build and deployment, and environment provisioning and testing processes, we can reduce the times to the minimum and incorporate them in the developer's daily work.
- **Lower costs**: If we invest in build, test, deployment, and environment automation, we reduce the cost of software by eliminating many fixed associated costs.

Tools for a continuous delivery pipeline

As mentioned before, continuous delivery is a continuation of continuous integration, so we can use most of the CI tools we mentioned earlier and expand our pipeline with our favorite testing framework. In PHP, we have a great number of testing frameworks available, but the most well known are the following:

- **phpUnit**: This is the most well-known framework used to create unit tests. Every PHP developer needs to know this framework as it will be the foundation of their tests. It is a standard in the industry.
- **Codeception**: This is one of the complete testing suites available for PHP. With Codeception, you can build unit, functional, and acceptance tests.
- **Behat**: This is the most popular behavior-driven PHP testing framework. Instead of writing code, you write stories and the framework will transform and test them.
- **PHPSec**: This is another important framework that follows the behavior-driven testing.
- **Selenium**: This is one of the most sophisticated testing frameworks used to automate browsers. With this framework, it is possible to write user acceptance tests.

In the subsequent chapters, we will use some of these testing frameworks. In the meantime, give each of them a go and select your favorite frameworks. Remember that you can mix them without any problems.

Summary

In this chapter, we talked about the different ways of designing and developing an application. We covered some patterns and strategies that you can easily integrate in your development workflow and we even talked about the most common development practices. In the subsequent chapters, we will apply all these concepts in our development workflow.

4
Testing and Quality Control

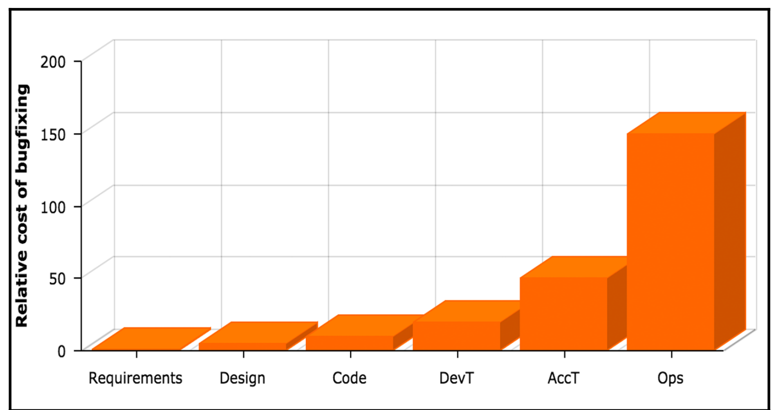
In this chapter, we will look at the different testing methodologies that you can use before, during, and after the development process. As you will know, testing your application avoids future issues and gives you a better project overview.

The importance of using tests in your application

It is very important to use testing in our application because these steps can avoid (or at least reduce) future problems or errors that can appear as we are humans and can make mistakes during the development process or because the structure of the project is not correct or even the understanding of the developer does not match the requirements of the customer.

The testing process will help improve the code quality and understanding of the functionalities, do regression testing in order to avoid the inclusion of old issues in continuous integration and reduce the time taken to finish the project.

Testing is used to reduce the fails or errors in our application. Development teams spend a lot of time doing bug fixing and, depending on the moment of the discovery of the bugs, the impact can be bigger or smaller. The following image shows the relative cost of bug fixing related to the stage of the development:



The reason for using testing methodologies in development is that we can find errors in our code in the early steps of the development so that we will spend less time doing bug fixing.

Testing in microservices

The challenge of testing an application based on microservices is not in the testing of every single microservice, but on the integration and data consistency. An application based on microservices will need a better understanding of the architecture of microservices and their workflow by the developers to be able to use testing on it. This is because it is necessary to check the information and also the functionality of every microservice at all the points of communication between the microservices.

The testing to use on microservices are as follows:

- **Unit testing**: In all the applications based on microservices or even in a monolithic one, it is necessary to use unit testing. Using it, we will check the necessary functionality of the methods or code modules.
- **Integration testing**: Unit testing only checks the isolated components, so we also need to check the behavior between methods. We will check the behavior between methods of the same microservice using integration testing, so the calls between microservices will need to be mocked.

- **API tests**: The microservices architecture depends on the communication between them. For each microservice, it is necessary to establish an API; this is like a *contract* to use that microservice. With this kind of test, we will check that the contract is working for each microservice and all the microservices are working with each other.
- **End-to-end tests**: These guarantee the application quality without any mockup method or call. A test will be run to assess the functionality between all the required microservices. There are some rules to avoid problems during these tests:
    - End-to-end tests are difficult to maintain, so only test the most important functionalities; the rest of them use unit testing
    - The user functionalities can be tested by simulating calls to the microservices
    - It is necessary to keep a clean environment to test it because the tests are very dependent on the data, so a previous test can manipulate the data and then, the next test

Once we know how to proceed with testing our application based on microservices, we will look at some strategies to do so during development.

Test-driven development

Test-driven development (**TDD**) is part of agile philosophy, and it appears to solve the common developer's problem that shows when an application is evolving and growing and the code is getting sick. The developers fix the problems to make it run but every single line that we add can be a new bug and it can even break other functions.

TDD is a learning technique that helps the developer to learn about the domain problem of the application they will build, doing it in an iterative, incremental, and constructivist way:

- **Iterative** because the technique always repeats the same process to get a value
- **Incremental** because for each iteration, we have more unit tests to be used
- **Constructivist** because it is possible to test everything we are developing during the process straightaway so that we can get immediate feedback

Also, when we finish developing each unit test or iteration, we can forget it because it will be kept throughout the entire development process, helping us to remember the domain problem through the unit test. This is a good approach for forgetful developers.

It is very important to understand that TDD includes four things: analysis, design, development, and testing. In other words, doing TDD is understanding the domain problem and correctly analyzing the problem, designing the application well, developing well, and testing it. It needs to be clear; TDD is not just about implementing unit tests, but the whole process of the software development.

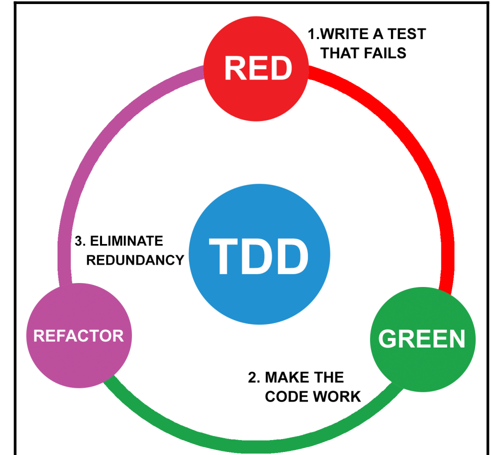
TDD perfectly matches projects based on microservices, because using microservices in a large project is dividing it into little microservices, our functionalities are like an aggrupation of little projects connected by a communication channel. The project size is independent of using TDD because, in this technique, you divide each functionality into little examples and, to do this, it does not matter whether the project is big or small, and even less when our project is divided by microservices. Also, microservices are still better than a monolithic project because the functionalities for the unit tests are organized in microservices and it will help the developers to know where they can begin to use TDD.

How to do TDD?

Doing TDD is not difficult, we just need to follow some steps and repeat them by improving our code and checking that we did not break anything:

1. **Write the unit test**: It needs to be the simplest and clearest test possible, and once it is done it has to fail; this is mandatory, if it does not fail, it means that there is something we are not doing properly.
2. **Run the tests**: If it has errors (it fails), this is the moment to develop the minimum code to pass the test; just do what is necessary, do not code additional things. Once you develop the minimum code to pass the test, run the test again; if it passes go to the next step, if not fix it and run the test again.
3. **Improve the test**: If you think it is possible to improve the code you wrote, do it and run the tests again. If you think it is perfect, write a new unit test.

The following image illustrates the mantra of TDD–**RED**, **GREEN**, **REFACTOR**:



To do TDD, it is necessary to write the tests before implementing the function; if the implementation is started and then the tests are written, it is not TDD, it is just testing.

If we create the unit tests after we start developing our application, we are doing the classic testing and we are not taking advantage of the TDD benefits. Having your unit tests in place will help you ensure that your abstract idea of the domain problem is correct throughout the developing process.

Obviously, doing testing is always better than not doing it, but doing TDD is still better than doing just classic testing.

Why should I use TDD?

TDD is the answer to questions such as "where shall I begin?", "how can I do it?", "how can I write code that can be modified without breaking anything?", and "how can I know what I have to implement?".

The goal is not to write many unit tests without sense, but to design TDD properly following the requirements. In TDD, we do not to think about implementing functions, but about good examples of the functions related to the domain problem in order to remove the ambiguity created by the domain problem.

In other words, we should reproduce a specific function or case of use in X examples by doing TDD until we get the necessary examples to describe the function or task without ambiguity or misinterpretations.

TDD can be the best way to document your application.

Using other methodologies of software development, we start thinking about how the architecture will be, what pattern will be used, how the communication between microservices will be, but what happens if once we have planned all this, we realize that it is not necessary? How much time will pass until we realize that? How much effort and money will we spend?

TDD defines the architecture of our application by creating little examples in many iterations until we realize what is the architecture. The examples will slowly show us the steps to follow in order to define what the best structure, pattern, or tools to use are, so that we avoid spending resources during the first stages of our application.

This does not mean that we are working without an architecture. Obviously, we have to know whether our application will be a website or a mobile app and use a proper framework (you can research which framework fits your needs in `Chapter 2`, *Development Environment*), and also know what the interoperability in the application will be; in our case, it will be an application based on microservices. So, it will give us support to start creating the first unit tests. TDD will be our guideline to develop an application and it will produce an architecture without ambiguity from the unit testing.

TDD is not cure-all; in other words, it does not give the same results to a senior and junior developer, but it is useful for the whole team. Let's look at some advantages of using TDD:

- **Code reuse**: This creates every functionality with only the necessary code to pass the tests in the second stage (Green). It allows you to see whether there are more functions using the same code structure or parts of a specific function; so, it helps you to reuse the code you wrote earlier.
- **Teamwork is easier**: It allows you to be confident with your team colleagues. Some architects or senior developers do not trust developers with poor experience, and they need to check their code before committing the changes, creating a bottleneck at that point, so TDD helps us to trust the developers with less experience.

- **Increases the communication**: It increases the communication between team colleagues. The communication is more fluent, so the team share their knowledge about the project reflected on the unit tests.
- **Avoid overdesign**: Do not overdesign the application in the first stages. As we said before, doing TDD allows you to have an overview of the application little by little, avoiding creating useless structures or patterns in your project that you will maybe trash in the future stages.
- **Unit tests are the best documentation**: The best way to give a good point of view of a specific functionality is by reading its unit test, which helps us understand how it works instead of human words.
- **Allows to discover more use cases** in the design stage: In every test you have to create, you will understand how the functionality should work better and all the possible stages that a functionality can have.
- **Increases the feeling of a job well done**: In every commit of your code, you will have the feeling that it was done properly because the rest of the unit tests pass without errors, so you will not be worried about breaking other functionalities.
- **Increases the software quality**: During the refactoring step, we spend our efforts in making the code more efficient and maintainable, verifying that the whole project still works properly after the changes.

TDD algorithm

The technical concepts and steps to follow the TDD algorithm are easy and clear, and the proper way to make it happen improves by practicing it. As we saw before, there are only three steps: red, green, and refactor.

Red – writing the unit tests

It is possible to write a test even when the code is not written, you just need to think about whether it is possible to write a specification before implementing it. So, in the first step, you should consider that the unit test you start writing is not like a unit test but like an example or specification of the functionality.

In TDD, this example or specification is not fixed; in other words, the unit test can be modified in the future. Before beginning to write the first unit test, it is necessary to think about how the **software under test** (**SUT**) will be, and just a little about how it will work. We need to think about how it will be the SUT code and how we will check that it works the way we want it to.

The way that TDD works drives us to firstly design what is more comfortable and clear if it fits the requirements.

Green – make the code work

Once the example is written, we have to code the minimum to make it pass the test; in other words, set the unit test to green. It does not matter if the code is ugly and not optimized, it will be our task in the next steps and iterations.

In this step, the important thing is only to write the necessary code for the requirements, without unnecessary things. It does not mean writing without thinking about the functionality, but thinking about it to be efficient. It looks easy, but you will realize that you will write extra code the first time.

If you concentrate on this step, you will think of new questions about the SUT behavior with different entries. However, you should be strong and avoid writing extra code about other functionalities related to the current one. As a rule of thumb, instead of coding the new features, take notes so you can convert them into functionalities in future iterations.

Refactor – eliminate redundancy

Refactoring is not the same as rewriting code. You should be able to change the design without changing the behavior.

In this step, you should remove the duplicity in your code and check whether the code matches the principles of the good practices, thinking about the efficiency, clarity, and the future maintainability of the code. This part depends on the experience of each developer.

The key to good refactoring is doing it in small steps.

To refactor a functionality, the best way to do it is to change a little part and execute all the available tests, and if they pass, continue with another little part until you are happy with the obtained result.

Behavior-driven development

Behavior-driven development (**BDD**) is a process that broadens the TDD technique and mixes it with other design ideas and business analysis provided to the developers in order to improve the software development.

In BDD, we test the scenarios and the class's behavior in order to meet the scenarios that can be composed by many classes.

It is very useful to use a DSL in order to have a common language to be used by the customer, project owner, business analyst, or developers. The goal is to have a ubiquitous language as we saw in `Chapter 3`, *Application Design*, in the domain-driven design section.

What is BDD?

As we said before, BDD is an agile technique based on TDD and ATDD, promoting the collaboration between the entire team of a project.

The goal of BDD is that the entire team understands what the customer wants, and the customer knows what the rest of the team understood from their specifications. Most of the times, when a project starts, developers don't have the same point of view as the customer, and during the development process the customer realizes that maybe they did not explain it or maybe the developer did not understand it properly, so it adds more time to change the code to meet the customer's needs.

So, BDD is writing test cases in human language using rules or a ubiquitous language so that the customer and developers can understand it. It also defines a DSL for the tests.

How does it work?

It is necessary to define the features as user stories (we will explain what this is in the ATDD section of this chapter) and examine their acceptance criteria.

Once the user story is defined, we have to focus on the possible scenarios that describe the project behavior for a concrete user or situation using DSL. The steps are: given (context), when (event occurs), and then (outcome).

To sum up, the defined scenario for a user story gives the acceptance criteria to check whether the feature is done.

Cucumber as DSL for BDD

Cucumber is a DSL tool that executes examples made in plain text as automatic tests, taking advantage of the benefits of BDD and bringing together the business layer and the technology in a project in order to know what are the functionalities most valued by the user and develop them at the same time that we are defining the case tests and documenting the project.

The most important thing for Cucumber is having the same point of understanding between developers and customers.

Gherkin is the language that Cucumber uses, and it allows you to translate the specifications of the project into a near human language so that the customer or other people without technical skills can understand it. This tool and language can be used for BDD and ATDD. Let's look at a piece of sample code:

```
Feature: Search secrets
 In order to find secrets
 Users should be able to search for near secrets

 Scenario: Search secrets by distance
   Given there are 996 secrets in the game which are no closer than 100
   meters from me
   And there are 4 secrets SEC001, SEC005, SEC054, SEC121 that are
   within 100
   meters from me
   When I search for closer secrets
   Then I should see the following secrets:
     | Secret code |
     | SEC001      |
     | SEC005      |
     | SEC054      |
     | SEC121      |
```

This allows us to define the software behavior without saying how it is implemented. Also, it allows us to document the functionalities at the same time we write the case automatic tests. The advantages of using Cucumber are as follows:

- Easy to read
- Easy to understand
- Easy to parse
- Easy to discuss

DSL has three steps in the code that the tool understands and processes; they are as follows:

1. **Given**: This is the necessary step to set the system in the proper status to check the tests.
2. **When**: This is the necessary step to be carried out by the user to action the functionality.
3. **Then**: This refers to the things that change in the system. Here, we are able to see if it does what we want.

Also, there are two more available optional steps: **And** and **But**, and they can be used in **Given** or **Then** when you need more than a sentence to match the requirements.

In this chapter, we will see how to use a tool, called Selenium, to do BDD. It is another DSL tool but is oriented to web development instead of plain text.

Acceptance test-driven development

Maybe the most important methodology in a project is the **Acceptance Test-Driven Development** (**ATDD**) or **Story Test-Driven Development** (**STDD**); it is TDD but on a different level.

The acceptance (or customer) tests are the written criteria that the project meets the business requirements that the customer demands. They are examples (like the examples in TDD) written by the project owner. It is the beginning of the development for each iteration, the bridge between scrum and agile development.

In ATDD, we start the implementation of our project in a different way to the traditional methodologies. The business requirements written in human language are replaced by executables agreed by some team members and the customer. It is not about replacing the whole documentation, but only part of the requirements.

The advantages of using ATDD are as mentioned:

- It provides real examples and a common language for the team to understand the domain
- It allows us to identify the domain rules properly
- It is possible to know if a user story is finished in each iteration
- The workflow works from the first steps
- The development does not start until the tests are defined and accepted by the team

User stories

The user stories of ATDD are like use cases in terms of name or description, but work in a different way. A user story does not define the requirement, avoiding the problem of ambiguity of the human language. The goal is to communicate the idea without problems for the rest of the team.

Each user story is a list of clear and concise examples about what the customer wants from the application. The name of the story is a sentence of human language defining what the function must do. Consider the following examples:

- Search available secrets around our position
- Check the secrets we already have stored
- Check who is the winner in a battle

They have the goals of listening to the customers and helping them define what they expect from the application. The user stories should be clear without ambiguity and should be written in human language, not in technical language; the customer should understand what they say.

Once we have defined a user story, some questions appear and they should be answered by associating acceptance tests for each story. For example, for the *Check who is the winner in a battle* story, some possible questions are as listed:

- What happens if they draw?
- What does the winner win?
- What does the loser lose?
- How long does a battle take?

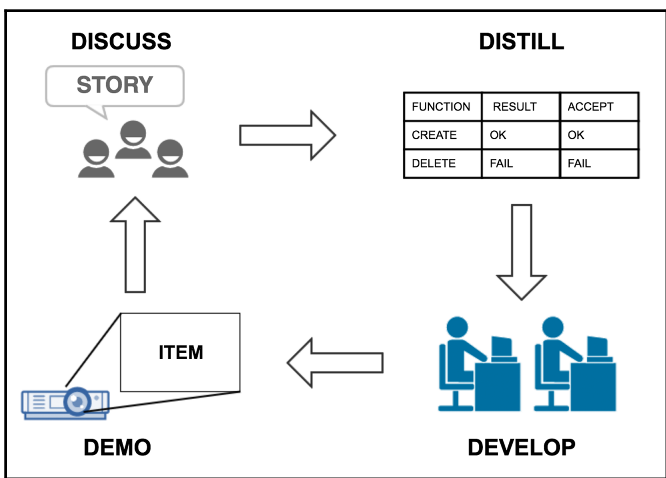
And the possible acceptance tests are as follows:

- If they draw, no one will win or lose anything; they will keep their secrets
- The winner will earn 10 points and get a secret from the loser's pocket
- The loser will give a secret to the winner
- A battle takes three throws of a dice

Maybe the questions and answers from a user story will generate new user stories to be added to the backlog.

ATDD algorithm

The algorithm of ATDD is like the TDD, but reaches more people than the developers. In other words, doing ATDD, the tests of each story are written in a meeting that includes the project owners, developers, and QA technicians because the team must understand what is necessary to do and why it is necessary so that they can see if it is what the code should do. The following image shows the ATDD cycle:



Discuss

The starting point of the ATDD algorithm is the discussion. In this step, the business has a meeting with the customer to clarify how the application should work and the analyst should create the user stories from that conversation. Also, they should be able to explain the conditions of satisfaction of every user story in order to be translated into examples as we explained earlier, in the user stories section.

By the end of the meeting, the examples should be clear and concise so that we can get a list of examples of user stories to cover all the needs that the customer reviewed and understood for himself. Also, the team will have a project overview in order to understand the business value of the user story, and in case the user story was too big, it can be divided into little user stories, getting the first one for the first iteration of this process.

Distill

The high level acceptance tests are written by the customer and the development team. In this step, the writing of the test cases that we got from the examples in the discussion step begins and the team can take part in the discussion and help clarify the information or specify its real needs.

The tests should cover all the examples that were discovered in the discussion step, and extra tests could be added during this process; little by little we are understanding the functionality better.

At the end of this step, we will obtain the necessary tests written in human language so that the team (including the customer) can understand what they will do in the next step. These tests can be used like documentation.

Develop

In the develop step, the acceptance test cases start to be developed by the development team and the project owner. The methodology to follow in this step is the same as TDD–the developers should create a test and watch it fail (red), and then develop the minimum amount of lines to pass (green). Once the acceptance tests are green, this should be verified and tested to be ready to be delivered.

During this process, the developers may find new scenarios that need to be added into the tests or even, if it needs a large amount of work, it could be pushed to the user story.

At the end of this step, we will have a software that passes the acceptance tests and maybe more comprehensive tests.

Demo

The created functionality is shown by running the acceptance test cases and manually exploring the features of the new functionality. After it is demonstrated, the team discusses whether the user story was done properly and whether it meets the product owner's needs and decide if they can continue with the next story.

Tools

Now that you know more about TDD and BDD, it is time to explain a few tools that you can use in your development workflow. There are a lot of tools available, but we will only explain the most used ones.

Composer

Composer is a PHP tool used to manage software dependencies. You only need to declare the libraries required by your project and Composer will manage them, installing and updating them when necessary. This tool has only a few requirements–if you have PHP 5.3.2+, you are ready to go. In the case of a missing requirement, Composer will warn you.

You can install this dependency manager on your development machine, but since we are using Docker, we will install it directly on our **PHP-FPM** (**FastCGI Process Manager**) containers. The installation of Composer in Docker is very easy; you only need to add the following rule to the Dockerfile:

```
RUN curl -sS https://getcomposer.org/installer
| php -- --install-dir=/usr/bin/ --filename=composer
```

PHPUnit

Another tool we need for our project is PHPUnit, a unit test framework. In our case, we will use version 4.0. The same as before, we will add this tool to our PHP-FPM containers to keep our development machine clean. If you are wondering why we are not installing anything on our development machine except for Docker, the response is clear–having everything in the containers will help you avoid any conflict with other projects and give you the flexibility of changing versions without being too worried.

As a quick way, you can add the following `RUN` command to your `PHP-FPM Dockerfile` and you will have the latest PHPUnit version installed and ready to use:

```
RUN curl -sSL https://phar.phpunit.de/phpunit.phar -o
/usr/bin/phpunit && chmod +x /usr/bin/phpunit
```

The preceding command will install the latest Composer version in your container, but the recommended way of installing it is through Composer. Open your `composer.json` file and add the following line:

```
"phpunit/phpunit": "4.0.*",
```

As soon as you update the `composer.json` file, you only need to do a Composer update in your container command line and Composer will install PHPUnit for you.

Now that we have all our requirements as well, it is time to install our PHP framework and start doing some TDD stuff. Later, we will continue updating our Docker environment with new tools.

In the previous chapters, we talked about some PHP frameworks and chose Lumen as our example. Feel free to adapt all the examples to your favorite framework. Our source code will be living inside our containers but, at this point of the development, we do not want immutable containers. We want every change we make to our code to be available instantaneously in our containers so that we will use a container as a storage volume.

To create a container with our source and use it as a storage volume, we only need to edit our `docker-compose.yml` file and create a source container for each of our microservices, as follows:

```
source_battle:
   image: nginx:stable
   volumes:
       - ../source/battle:/var/www/html
   command: "true"
```

The preceding piece of code creates a container image named `source_battle`, and it stores our battle source (located at `../source/battle` from the `docker-compose.yml` file current path). Once we have our source container available, we can edit each of our services and assign a volume. For instance, we can add the following line to our `microservice_battle_fpm` and `microservice_battle_nginx` container descriptions:

```
volumes_from:
           - source_battle
```

Our battle source will be available in our source container at the `/var/www/html` path and the remaining step to install Lumen is to do a simple Composer execution. First, you need to ensure that your infrastructure is up with a simple command:

```
$ docker-compose up
```

The preceding command spins up our containers and outputs the log to the standard IO. Now that we are sure that everything is up and running, we need to enter in our PHP-FPM containers and install Lumen.

If you need to know the names assigned to each of your containers, you can execute `docker ps` on your terminal and copy the container name.As an example, we will enter the following command into the battle PHP-FPM container:

```
$ docker exec -it docker_microservice_battle_fpm_1 /bin/bash
```

The preceding command opens an interactive shell in your container so that you can do anything you want. Let's install Lumen with a single command:

```
# cd /var/www/html && composer create-project --prefer-dist laravel/lumen .
```

Repeat the preceding commands for each of your microservices.

Now you have everything ready to start doing unit tests and code your application.

Unit testing

A unit test is a small piece of code that uses other code in a known context so that we can check whether the code we are testing is valid. Lumen comes with PHPUnit out of the box; therefore, we only need to add all our tests to the tests folder. The framework installation comes with a very small sample file by default–`ExampleTest.php`–where you can give the unit testing a try. In order to start with unit testing and until you become more comfortable creating unit tests, choose one of your microservices sources and create the `app/Dummy.php` file with the following content:

```
<?php
namespace App;

class Dummy
{
}
```

The easiest way of starting with unit testing is every time you create a new class in your code, you can create a new one for your tests. Working this way, you will remember that your new class needs to be covered with unit tests. For instance, imagine that you need a `Battle` class; therefore, when you create the class, you can also create a new one with a `Test` prefix in your `tests` folder.

In an ideal world, all your code is covered by unit tests but we know that this is an odd case. Most of the times, you will have 70% or 80% of code coverage if you are lucky. We encourage you to keep your code fully covered but if it is not possible, cover at least the core functionalities. There are two ways of creating unit tests:

- **Tests first, code later:** In our opinion, this workflow is better when you have enough time to develop your project. First, you create the tests so that you are sure that you really know each new functionality. After you have the tests in place, you will write the minimum code necessary to pass the tests. Coding this way, you will be thinking about what makes your code valid and what can make your code fail.
- **Code first, tests later:** This a very extended workflow when you don't have too much time for unit testing. You create your code as always and, as soon as you have finished, you create the unit tests. This approach creates a less robust code because you are adapting your unit test to the already created code instead of doing it the other way around.

Remember that it is important to always have time to test your code; it is a long-term investment. Spending time at the beginning will make your code more robust and will eliminate future bugs.

Running the tests

You are probably wondering how you can run and check your tests. Don't worry, it is very simple. You only need to enter one of your PHP-FPM containers. For example, to enter in the Battle PHP-FPM container, open your terminal and execute the following command:

```
$ docker exec -it docker_microservice_battle_fpm_1 /bin/bash
```

After executing the above command you will be inside the container. Now it is time to be sure that your current path is the `/var/www/html` folder. After you accomplish the previous step, you can execute phpunit inside that folder. All this actions can be done with the following commands:

```
# cd /var/www/html
# ./vendor/bin/phpunit
```

The `phpunit` command will read the `phpunit.xml` file. This XML describes where our tests are stored and executes all of them. The execution of this command will give us a pretty screen with the results of our passed or failed tests.

Assertions

An assertion is a statement in a known context that we are expecting to be true at some point in our code and that is the core of unit testing. Assertions are used inside test cases and a test case can include multiple assertions inside the same tests. In PHPUnit, it is very simple to create a test as you only need to add the `test` prefix to your method name. Easy, isn't it? To clarify all these concepts, let's look at some assertions you can use in your unit tests with some examples. Feel free to create more complex tests until you are comfortable with PHPUnit.

assertArrayHasKey

The `assertArrayHasKey(mixed $key, array $array[, string $message = ''])` assertion checks whether `$array` has an element with `$key`. Imagine that you have a method that generates and returns some kind of configuration data and there is a specific element identified by `storage` that you need to be sure is always present. Add the following method to our `Dummy` class to simulate the configuration generation:

```
public static function getConfigArray()
{
  return [
       'debug'   => true,
       'storage' => [
           'host' => 'localhost',
           'port' => 5432,
           'user' => 'my-user',
           'pass' => 'my-secret-password'
       ]
   ];
}
```

Now we can test the response of this `getConfigArray` in any way we want:

```
public function testFailAssertArrayHasKey()
{
   $dummy = new App\Dummy();

   $this->assertArrayHasKey('foo', $dummy::getConfigArray());
}
```

The preceding test will check if the array returned by `getConfigArray` has an element identified by `foo`, which fails in our example:

```
public function testPassAssertArrayHasKey()
{
   $dummy = new App\Dummy();

   $this->assertArrayHasKey('storage', $dummy::getConfigArray());
}
```

In this case, this test will ensure that `getConfigArray` returns an element identified by `storage`. If, for some reason, you change the implementation of the `getConfigArray` method in future, this test will help you ensure that you keep receiving at least an array element identified by `storage`.

You can use `assertArrayNotHasKey()` as the inverse of `assertArrayHasKey;` it uses the same arguments.

assertClassHasAttribute

The `assertClassHasAttribute(string $attributeName, string $className[, string $message = ''])` assertion checks whether our `$className` has defined the `$attributeName`. Modify our `Dummy` class and add a new attribute, as follows:

```
public $foo;
```

Now we can test the existence of this public attribute with the following tests:

```
public function testAssertClassHasAttribute()
{
  $this->assertClassHasAttribute('foo', App\Dummy::class);
  $this->assertClassHasAttribute('bar', App\Dummy::class);
}
```

The preceding code will pass the check of the `foo` attribute but will fail in checking the `bar` attribute.

You can use `assertClassNotHasAttribute()` as the inverse of `assertClassHasAttribute`; it uses the same arguments.

assertArraySubset

The `assertArraySubset(array $subset, array $array[, bool $strict = '', string $message = ''])` assertion checks whether a given `$subset` is available in our `$array`:

```
public function testAssertArraySubset()
{
   $dummy = new App\Dummy();

   $this->assertArraySubset(['storage' => 'failed-test'],
   $dummy::getConfigArray()]);
}
```

The preceding example test will fail because the `['storage' => 'failed-test']` subset does not exist in the response of our `getConfigArray` method.

assertClassHasStaticAttribute

The `assertClassHasStaticAttribute(string $attributeName, string $className[, string $message = ''])` assertion checks the existence of a static attribute in a given `$className`. We can add a static attribute, like the following one, to our `Dummy` class:

```
public static $availableLocales = [
       'en_GB',
       'en_US',
       'es_ES',
       'gl_ES'
];
```

Having this static attribute in place, we are free to test the existence of `$availableLocales`:

```
public function testAssertClassHasStaticAttribute()
{
  $this->assertClassHasStaticAttribute('availableLocales',
  App\Dummy::class);
}
```

In case of needing to assert the inverse, you can use `assertClassNotHasStaticAttribute()`; it uses the same arguments.

assertContains()

Sometimes you need to check if a haystack contains specific elements. You can do this using the `assertContains()` functions:

- `assertContains(mixed $needle, Iterator|array $haystack[, string $message = ''])`
- `assertNotContains(mixed $needle, Iterator|array $haystack[, string $message = ''])`
- `assertContainsOnly(string $type, Iterator|array $haystack[, boolean $isNativeType = null, string $message = ''])`
- `assertNotContainsOnly(string $type, Iterator|array $haystack[, boolean $isNativeType = null, string $message = ''])`
- `assertContainsOnlyInstancesOf(string $classname, Traversable|array $haystack[, string $message = ''])`

assertDirectory() and assertFile()

PHPUnit not only allows you to test the logic of your application, but you can even test the existence and permissions of folders and files. All this can be achieved with the following assertions:

- `assertDirectoryExists(string $directory[, string $message = ''])`
- `assertDirectoryNotExists(string $directory[, string $message = ''])`
- `assertDirectoryIsReadable(string $directory[, string $message = ''])`

- `assertDirectoryNotIsReadable(string $directory[, string $message = ''])`
- `assertDirectoryIsWritable(string $directory[, string $message = ''])`
- `assertDirectoryNotIsWritable(string $directory[, string $message = ''])`
- `assertFileEquals(string $expected, string $actual[, string $message = ''])`
- `assertFileNotEquals(string $expected, string $actual[, string $message = ''])`
- `assertFileExists(string $filename[, string $message = ''])`
- `assertFileNotExists(string $filename[, string $message = ''])`
- `assertFileIsReadable(string $filename[, string $message = ''])`
- `assertFileNotIsReadable(string $filename[, string $message = ''])`
- `assertFileIsWritable(string $filename[, string $message = ''])`
- `assertFileNotIsWritable(string $filename[, string $message = ''])`
- `assertStringMatchesFormatFile(string $formatFile, string $string[, string $message = ''])`
- `assertStringNotMatchesFormatFile(string $formatFile, string $string[, string $message = ''])`

Does your application rely on a writable file in order to work? Don't worry, PHPUnit has your back. You can add an `assertFileIsWritable()` to your tests so that the next time somebody removes the file you have specified in the assertion, the test will fail.

assertString()

In some cases, you need to check the content of some strings. For instance, if your code generates serial codes, you can check whether the codes you generate match your specifications. You can use the following assertions with strings:

- `assertStringStartsWith(string $prefix, string $string[, string $message = ''])`
- `assertStringStartsNotWith(string $prefix, string $string[, string $message = ''])`

- `assertStringMatchesFormat(string $format, string $string[, string $message = ''])`
- `assertStringNotMatchesFormat(string $format, string $string[, string $message = ''])`
- `assertStringEndsWith(string $suffix, string $string[, string $message = ''])`
- `assertStringEndsNotWith(string $suffix, string $string[, string $message = ''])`

assertRegExp()

The `assertRegExp(string $pattern, string $string[, string $message = ''])` assertion will be very useful for you as you have all the regex power in one assertion. Let's add a static function to our Dummy class:

```
public static function getRandomCode()
{
  return 'CODE-123A';
}
```

This new function returns a static string code. Feel free to complicate the generation. To test this generated string code, you can now do something like this in your test class:

```
public function testAssertRegExp()
{
   $this->assertRegExp('/^CODE\-\d{2,7}[A-Z]$/',
   App\Dummy::getRandomCode());
}
```

As you can see, we are using a simple regular expression to check the output generated by `getRandomCode`.

assertJson()

Working with microservices, you will probably be working very closely with JSON requests and responses. Therefore, it is very important that you have the ability to test our JSONs. You can have a JSON as a file or as a string:

- `assertJsonFileEqualsJsonFile()`
- `assertJsonStringEqualsJsonFile()`
- `assertJsonStringEqualsJsonString()`

Boolean assertions

Boolean results or types can be checked with the following methods:

- `assertTrue(bool $condition[, string $message = ''])`
- `assertFalse(bool $condition[, string $message = ''])`

Type assertions

Sometimes you need to ensure that an element is an instance of a specific class or has a specific internal type. You can use the following assertions in your tests:

- `assertInstanceOf($expected, $actual[, $message = ''])`
- `assertInternalType($expected, $actual[, $message = ''])`

Other assertions

PHPUnit has a high number of assertions and your tests can't be completed without some of the following assertions applied to results or object states of your features:

- `assertCount($expectedCount, $haystack[, string $message = ''])`
- `assertEmpty(mixed $actual[, string $message = ''])`
- `assertEquals(mixed $expected, mixed $actual[, string $message = ''])`
- `assertGreaterThan(mixed $expected, mixed $actual[, string $message = ''])`
- `assertGreaterThanOrEqual(mixed $expected, mixed $actual[, string $message = ''])`
- `assertInfinite(mixed $variable[, string $message = ''])`
- `assertLessThan(mixed $expected, mixed $actual[, string $message = ''])`
- `assertLessThanOrEqual(mixed $expected, mixed $actual[, string $message = ''])`
- `assertNan(mixed $variable[, string $message = ''])`

- `assertNull(mixed $variable[, string $message = ''])`
- `assertObjectHasAttribute(string $attributeName, object $object[, string $message = ''])`
- `assertSame(mixed $expected, mixed $actual[, string $message = ''])`

You can find more information about the assertions you can use on the PHPUnit website, that is, `https://phpunit.de/`.

Unit testing from scratch

At this point, you probably feel more comfortable with unit testing and you want to start coding your app as soon as possible, so let's get started with testing!

Our microservices application uses geolocalization to find secrets and other players. This means that your location microservice will need a way to calculate the distance between two geospatial points. We will also need, given an origin point, to get a list of the closest stored points (they can be the closest users or secrets). As this is a core feature, you need to ensure that what we described is fully tested.

In our app, the localization has its own service. Therefore, open the source code of the location microservice with your IDE and create the `app/Http/Controllers/LocationController.php` file with the following content:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class LocationController extends Controller
{

}
```

The preceding code has created our location controller in Lumen and, as we mentioned before, as soon as we have this class created, we need to create a similar class for our unit tests. In order to do this, you only need to create the `tests/app/Http/Controllers/LocationControllerTest.php` file. As you can see, we are even replicating the folder structure; this is the best approach to easily know which class we are testing for.

We want to start creating tests for the distance calculation and for the function that allows us to get the closest secrets given a specific geolocation point. One approach is to create two different tests. Therefore, fill your `LocationControllerTest.php` with the following code:

```
<?php

use Laravel\Lumen\Testing\DatabaseTransactions;

class LocationControllerTest extends TestCase
{
  public function testDistance()
  {
  }

  public function testClosestSecrets()
  {
  }
}
```

We didn't add anything special to our test class, we only declared two tests.

Let's start with `testDistance()`. In this test, we want to ensure that given two geospatial points, the calculated distance between them is accurate enough for our purposes. In unit testing, you need to start describing the known scenario–as points, we have chosen London (latitude: `51.50`, longitude: `-0.13`) and Amsterdam (latitude: `52.37`, longitude: `4.90`). The known distance between these two cities is around 358.06 km, and this is the result we aim to get from our distance calculator. Let's fill our test with the following code:

```
public function testDistance()
{
  $realDistanceLondonAmsterdam = 358.06;

  $london = [
       'latitude'  => 51.50,
       'longitude' => -0.13
  ];

  $amsterdam = [
       'latitude'  => 52.37,
       'longitude' => 4.90
  ];

  $location = new App\Http\Controllers\LocationController();

  $calculatedDistance = $location->getDistance($london, $amsterdam);
```

```
    $this->assertClassHasStaticAttribute('conversionRates',
    App\Http\Controllers\LocationController::class);
    $this->assertEquals($realDistanceLondonAmsterdam,
                        $calculatedDistance);
}
```

In the preceding piece of code, we defined the known scenario, the location of our two points and the distance known between them. Once we had our known scenario ready, we created an instance of our `LocationController` and we used the defined `getDistance` function to obtain a result we want to test. As soon as we got the result, we tested that our `LocationController` had a `conversionRate` static attribute that we can use to transform the distance to different units. Our last and most important assertion checks the match between the calculated distance and the known distance between these two points. We have our basic test ready, and it is time to start coding our `getDistance` function.

The distance calculation between two geospatial points can be calculated in very different ways. You can use a strategy pattern here, but to keep the example simple we will code the different calculation algorithms in different functions inside our controller.

Open your `LocationController` and add some auxiliary code:

```
const ROUND_DECIMALS = 2;

public static $conversionRates = [
   'km'    => 1.853159616,
   'mile'  => 1.1515
];

protected function convertDistance($distance, $unit = 'km')
{
  switch (strtolower($unit)) {
    case 'mile':
      $distance = $distance * self::$conversionRates['mile'];
      break;
    default :
      $distance = $distance * self::$conversionRates['km'];
      break;
  }

  return round($distance, self::ROUND_DECIMALS);
}
```

In the preceding code, we have defined our conversion rates, a constant we can use to round our results, and a simple conversion function. We will use this `convertDistance` function later.

Our first approach for distance calculation is to use the Euclidean function to get our result. A simple implementation is described in the following code:

```
public function getEuclideanDistance($pointA, $pointB, $unit = 'km')
    {
       $distance = sqrt(
           pow(abs($pointA['latitude'] - $pointB['latitude']), 2) +
pow(abs($pointA['longitude'] - $pointB['longitude']), 2)
       );

       return $this->convertDistance($distance, $unit);
    }
```

Now that we have our algorithm ready, we can add it to our `getDistance` function, as follows:

```
    public function getDistance($pointA, $pointB, $unit = 'km')
    {
      return $this->getEuclideanDistance($pointA, $pointB, $unit);
    }
```

At this point, you have everything in place and you can start the tests. Enter the location container and run PHPUnit in `/var/www/html`. This is our first approach; the PHPUnit result will be a fail, and the output of this application will tell you where the problem is. In our case, the main reason for the fail is that the algorithm we used is not accurate enough for our application. We can't deploy this version of our application because it has failed tests and we have to change our test or the code that implements our tests.

As we mentioned before, there are multiple ways of calculating the distance between two points and each one can be more or less accurate. The first implementation we tried failed because it is used for plains and our world is a sphere.

Open your `LocationController` again and create a new distance implementation using a haversine calculation:

```
public function getHaversineDistance($pointA, $pointB, $unit = 'km')
{
  $distance = rad2deg(
       acos(
           (sin(deg2rad($pointA['latitude'])) *
           sin(deg2rad($pointB['latitude']))) +
           (cos(deg2rad($pointA['latitude'])) *
           cos(deg2rad($pointB['latitude'])) *
           cos(deg2rad($pointA['longitude'] -
           $pointB['longitude'])))
       )
   ) * 60;

  return $this->convertDistance($distance, $unit);
}
```

As you can see, this distance calculation function is a little more complex and considers the spherical form of our world. Change the `getDistance` function to use our new algorithm:

```
return $this->getHaversineDistance($pointA, $pointB, $unit);
```

Now run PHPUnit again and everything should be okay; the test will pass and our code is ready for production.

With unit testing and TDD, the process is always the same:

1. Create the tests.
2. Make your code to pass the tests.
3. Run the tests and if they fail, start again from step 2.

Another feature we want to have in our location microservice is to get the closest secrets we have near our current position. Open the `LocationControllerTest` file and add the following code:

```
public function testClosestSecrets()
{
  $currentLocation = [
       'latitude'  => 40.730610,
       'longitude' => -73.935242
  ];

  $location = new App\Http\Controllers\LocationController();
```

```
    $closestSecrets = $location->getClosestSecrets($currentLocation);
    $this->assertClassHasStaticAttribute('conversionRates',
    App\Http\Controllers\LocationController::class);
    $this->assertContainsOnly('array', $closestSecrets);
    $this->assertCount(3, $closestSecrets);

     // Checking the first element
     $currentElement = array_shift($closestSecrets);
     $this->assertArraySubset(['name' => 'amber'], $currentElement);

     // Second
     $currentElement = array_shift($closestSecrets);
     $this->assertArraySubset(['name' => 'ruby'], $currentElement);

     // Third
     $currentElement = array_shift($closestSecrets);
     $this->assertArraySubset(['name' => 'diamond'], $currentElement);
}
```

In the preceding piece of code, we defined our current location (New York) and asked our implementation to give us a list of the closest secrets. Our location implementation will have a cache list of secrets, and we know where they are located (this will help us to know the correct order).

Open the `LocationController.php` and first add a cache list of secrets. In the real world, we don't have hardcoded values, but it's good enough for testing purposes:

```
public static $cacheSecrets = [
   [
       'id' => 100,
       'name' => 'amber',
       'location' => ['latitude'  => 42.8805, 'longitude' => -8.54569,
       'name'      => 'Santiago de Compostela']
   ],
   [
       'id' => 100,
       'name' => 'diamond',
       'location' => ['latitude'  => 38.2622, 'longitude' => -0.70107,
       'name'      => 'Elche']
   ],
   [
       'id' => 100,
       'name' => 'pearl',
       'location' => ['latitude'  => 41.8919, 'longitude' => 12.5113,
       'name'      => 'Rome']
   ],
   [
```

```
            'id' => 100,
            'name' => 'ruby',
            'location' => ['latitude'  => 53.4106, 'longitude' => -2.9779,
            'name'      => 'Liverpool']
        ],
        [
            'id' => 100,
            'name' => 'sapphire',
            'location' => ['latitude'  => 50.08804, 'longitude' => 14.42076,
            'name'      => 'Prague']
        ],
    ];
```

Once we have our list of secrets ready, we can add our `getClosestSecrets` function as follows:

```
    public function getClosestSecrets($originPoint)
    {
      $closestSecrets    = [];
      $preprocessClosure = function($item) use($originPoint) {
        return $this->getDistance($item['location'], $originPoint);
      };

       $distances = array_map($preprocessClosure, self::$cacheSecrets);

       asort($distances);

       $distances = array_slice($distances, 0,
         self::MAX_CLOSEST_SECRETS, true);

       foreach ($distances as $key => $distance) {
         $closestSecrets[] = self::$cacheSecrets[$key];
       }

       return $closestSecrets;
    }
```

In this code, we used our cache list of secrets to calculate the distance between the origin point and each of our secret points. As soon as we had the distance, we sorted the result and returned the closest three.

Running PHPUnit in our location container will show us that all the tests are being passed, giving us the confidence to deploy our code to production.

Future commits can make changes to the distance calculation or to the closest function, and they can break our tests. Luckily for you, there is a unit testing covering them and PHPUnit will throw an alert, so you can start rethinking your code implementation.

Let your imagination fly and test everything–from the simple and small case to any odd and obscure case you can imagine. The idea is that your application will break and break very badly, in the middle of the night or during your holidays. There is nothing you can do about this except adding as many tests as you can so that you can ensure that the release you have in production is stable enough to reduce the risks of breaking.

Behat

Behat is an open source behavior-driven development framework. All the Behat tests are written in plain English and wrapped into readable scenarios. This framework uses the Gherkin syntax and was inspired by Cucumber, a Ruby tool. The main advantage of Behat is that most of the test scenarios can be understood by anyone.

Installation

Behat can be easily installed using Composer. You only need to edit the `composer.json` of each microservice and drop a new line with `"behat/behat" : "3.*"`. Your `require-dev` definition will be as follows:

```
"require-dev": {
  "fzaninotto/faker": "~1.4",
  "phpunit/phpunit": "~4.0",
  "behat/behat": "3.*"
},
```

Once you have updated the `dev` requirements, you need to enter each of your PHP-FPM containers and run Composer:

```
# cd /var/www/html && composer update
```

Test execution

Running Behat is as easy as running PHPUnit. You only need to enter your PHP-FPM container, go to the `/var/www/html` folder, and run the following command:

```
# vendor/bin/behat
```

Behat example from scratch

One of the key features of our microservices application is the ability to find secrets. Users should be able to save the secrets and, for this, they need a wallet. So, let's write our user-story in our user microservice:

```
Feature: Secrets wallet
 In order to play the game
 As a user
 I need to be able to put found secrets into a wallet

 Scenario: Finding a single secret
    Given there is an "amber"
    When I add the "amber" to the wallet
    Then I should have 1 secret in the wallet

 Scenario: Finding two secrets
    Given there is an "amber"
    And there is a "diamond"
    When I add the "amber" to the wallet
    And I add the "diamond" to the wallet
    Then I should have 2 secrets in the wallet
```

As you can see, the test can be understood by anyone in the project–from the developers to the stakeholders. Each test scenario always has the same format:

```
Scenario: Some description of the scenario
 Given some context
 When some event
 Then the outcome
```

You can add some modifiers to the preceding basic template (and or but) to empower the scenario description. At this point, with your scenario ready, you can save it as a `features/wallet.feature` file.

The first time you start writing a Behat test on your project, you need to initialize the suite with the following command:

```
# vendor/bin/behat --init
```

The preceding command will create the files needed by Behat to run the scenario tests. The main file we will use is `features/bootstrap/FeatureContext.php`; this file will be our test environment.

Once we have our `FeatureContext` file in place, it is time to start creating our scenario steps. For example, place the following method in your `FeatureContext`:

```
/**
* @Given there is a(n) :arg1
*/
public function thereIsA($arg1)
{
   throw new PendingException();
}
```

Behat uses doc-blocks for step definitions, step transformations, and hooks.

In the preceding piece of code, we are telling Behat that the `thereIsA()` function matches each `Given there is a(n)` step. In our example, that definition will match the steps in the following cases:

- Given that there is an amber
- There is a diamond

We need to map each of our scenario steps so that our `FeatureContext` will end up as follows:

```
<?php

 use Behat\Behat\Context\Context;
 use Behat\Behat\Tester\Exception\PendingException;
 use Behat\Gherkin\Node\PyStringNode;
 use Behat\Gherkin\Node\TableNode;

/**
* Defines application features from the specific context.
*/
class FeatureContext implements Context
{
  private $secretsCache;
  private $wallet;

  public function __construct()
  {
    $this->secretsCache = new SecretsCache();
    $this->wallet = new Wallet($this->secretsCache);
  }
```

```
    /**
    * @Given there is a(n) :secret
    */
    public function thereIsA($secret)
    {
      $this->secretsCache->setSecret($secret);
    }

    /**
    * @When I add the :secret to the wallet
    */
    public function iAddTheToTheWallet($secret)
    {
      $this->wallet->addSecret($secret);
    }

    /**
    * @Then I should have :count secret(s) in the wallet
    */
    public function iShouldHaveSecretInTheWallet($count)
    {
       PHPUnit_Framework_Assert::assertCount(
         intval($count),
         $this->wallet
       );
    }
  }
```

Our tests use external classes that we need to define. These classes implement our logic and, for example, purposefully create the `features/bootstrap/SecretsCache.php` with the following content:

```
<?php
final class SecretsCache
{
  private $secretsMap = [];

  public function setSecret($secret)
  {
     $this->secretsMap[$secret] = $secret;
  }

  public function getSecret($secret)
  {
    return $this->secretsMap[$secret];
  }
}
```

You also need to create `features/bootstrap/Wallet.php` with the following sample code:

```
<?php
final class Wallet implements \Countable
{
  private $secretsCache;
  private $secrets;

  public function __construct(SecretsCache $secretsCache)
  {
    $this->secretsCache = $secretsCache;
  }

  public function addSecret($secret)
  {
    $this->secrets[] = $secret;
  }

  public function count()
  {
    return count($this->secrets);
  }
}
```

The preceding two classes are the implementation of our tests and, as you can see, they have the logic of storing secrets in a wallet. Now, if you run `vendor/bin/behat` on your console, the tool will check all our test scenarios and give us confidence that our code will behave the way we want it to.

This was a simple example of testing an application with Behat. In our GitHub repository, you can find more specific examples. Also, feel free to explore the Behat ecosystem; you can find multiple tools and extensions that can help you test your application.

Selenium

Selenium is a suite of tools to automate web browsers across many platforms and can be used as a browser extension, or it can be installed on a server to run our browser tests. The main advantage of Selenium is that you can easily record a full user journey and create a test from the record. This test can be later added to your pipeline to be executed on each commit to discover regressions.

Selenium WebDriver

The WebDriver is the API that you can use to run your browser testing from other tools. It is a powerful testing environment normally placed in a dedicated server where it sits waiting to run browser tests.

Selenium IDE

The Selenium IDE is a Firefox extension that allows you to record, edit, and debug browser tests. This plugin is not only a recording tool, but a full IDE with autocomplete functionalities. You can even record and create tests with the IDE and later run them with the WebDriver.

Most of the time, Selenium is used as a complimentary testing tool executed from another testing framework. For instance, you can improve your Behat tests with Selenium thanks to the Mink project (`http://mink.behat.org/en/latest/`). This project is a wrapper for different browser drivers, so you can use them in your BDD workflow.

We will talk about the deployment of our application in `Chapter 7`, *Security*. We will learn how to automate all these tests and integrate them in our CI/CD workflow.

Summary

In this chapter, you studied the importance of using tests on your application, tools such as Behat and Selenium, and also about implementing driven development. In the next chapter, you will study error handling, dependency management, and the microservices framework.

5

Microservices Development

In the last chapters, we explained how to install Docker, Composer, and Lumen, which will be necessary for each microservice. In this chapter, we will develop some parts of the *Finding secrets* application.

In this chapter, we will develop some of the more crucial parts, such as the routing, middleware, connection with a database, queues, and the communication between microservices of the Finding secrets application so that you will be able to develop the rest of the application in the future.

The structure of our application will have the following four microservices:

- **User:** It manages the registration and account actions. It is also responsible for storing and managing our secrets wallet.
- **Secrets:** It generates random secrets around the world and also allows us to get information about each secret.
- **Location**: It checks the closest secrets and users.
- **Battle:** It manages the battle between users. It also modifies the wallets to add and remove secrets after the battle.

Dependency management

Dependency management is a methodology that allows you to declare the libraries required for your project and makes it easier to install or update them. The most well-known tool for PHP is called **Composer**. In previous chapters, we gave a little overview about this tool.

For our project, we will need to use a single Composer setup for each microservice. When we installed Lumen, Composer did the work for us and created the configuration file, but now we will explain how it works in detail.

Once we have Docker installed and we are in the PHP-FPM container we want to work on, it is necessary to generate the `composer.json` file. This a configuration file for Composer where we define our project and all the dependencies:

```
{
  "name": "php-microservices/user",
  "description": "Finding Secrets, User microservice",
  "keywords": ["finding secrets", "user", "microservice", "Lumen" ],
  "license": "MIT",
  "type": "project",
  "require": {
    "php": ">=5.5.9",
    "laravel/lumen-framework": "5.2.*",
    "vlucas/phpdotenv": "~2.2"
  },
  "require-dev": {
    "fzaninotto/faker": "~1.4",
    "phpunit/phpunit": "~4.0",
    "behat/behat": "3.*"
  },
  "autoload": {
    "psr-4": {
        "App": "app/"
    }
  },
  "autoload-dev": {
    "classmap": [
        "tests/",
        "database/"
    ]
  }
}
```

The first 6 lines (name, description, keywords, license, and type) of the `composer.json` file are used to identify the project. It will be public if you share the project in any repository.

The `"require"` section defines the required libraries needed in our project and the version for each one. The `"require-dev"` is very similar, but they are the libraries that need to be installed on the development machines (for example, any test library).

The `"autoload"` and `"autoload-dev"` define the way that our classes will be loaded and the folders to be mapped on the project for different uses.

Once we have this file created, we can execute the following command in our machine:

```
composer install
```

At this point, composer will check our settings, and it will download all the required libraries, including Lumen.

There are other tools out there, but they aren't used as much and they are less flexible.

Routing

Routing is a mapping between the entry points of your application (requests) and a specific class and method in your source that executes your logic. For example, you can have defined in your application a mapping between the `/users` route and the method `list()` which is inside your `Users` class. Once you have this mapping in place, as soon as your application receives a request for the route `/users`, it will execute all the logic you have inside the `list()` method (located in the `Users` class). Routing allows the API consumers to interact with your application. In microservices, the RESTful convention is the most used and we will follow it.

- **HTTP Methods**:
    - **GET:** It is used to retrieve information about a specified entity or collection of entities. The amount of data does not matter; we will use GET for one or many results, and also we can use filters in order to filter the results.
    - **POST:** It is used to enter information in the application. It is also used to send new information in order to create new things or send a message.
    - **PUT:** It is used to update an entire entity already stored in the application.
    - **PATCH:** It is used to partially update an entity already stored in the application.
    - **DELETE:** It is used to remove an entity from the application.

The routes file in Lumen is located at `app/Http/routes.php`, so we will have one routes file for each microservice. For the `User` microservice, we will have the following endpoints:

```
$app->group([
    'prefix' => 'api/v1',
    'namespace' => 'App\Http\Controllers'],
    function ($app) {
        $app->get('user', 'UserController@index');
        $app->get('user/{id}', 'UserController@get');
        $app->post('user', 'UserController@create');
        $app->put('user/{id}', 'UserController@update');
```

```
            $app->delete('user/{id}', 'UserController@delete');
            $app->get('user/{id}/location',
            'UserController@getCurrentLocation');
            $app->post('user/{id}/location/latitude/{latitude}
            /longitude/{longitude}',
            'UserController@setCurrentLocation');
        }
    );
```

In the earlier piece of code, we defined our routes for the `User` microservice.

In Lumen, the versioning for the API can be specified on the routes file by including `'prefix'`. This framework also allows us to have different API versions for the same microservice, so we do not need to modify an existing method to be used in a different version.

The `'namespace'` defines the same namespace for all the methods included in the same group. The following lines define every entry point:

```
$app->get('user/{id}', 'UserController@get');
```

For example, the preceding method is included in the group with the prefix `'api/v1'`; the verb is GET, and the entry point is `user/{id}`, so it could be retrieved executing an HTTP GET call to `http://localhost:8080/api/v1/user/123`.
The `UserController@get` parameter defines where we need to develop the logic for this call–in this case, it is on the controller `UserController` and the method called `get`.

In Lumen, the standard folder to store all your controllers is `app/Http/Controllers`, so you only need to create the `app/Http/Controllers/UserController.php` file with your IDE with the following content:

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
class UserController extends Controller
{
  public function index(Request $request)
  {
    return response()->json(['method' => 'index']);
  }
  public function get($id)
  {
    return response()->json(['method' => 'get', 'id' => $id]);
  }
  public function create(Request $request)
  {
```

```
        return response()->json(['method' => 'create']);
      }
      public function update(Request $request, $id)
      {
        return response()->json(['method' => 'update', 'id' => $id]);
      }
      public function delete($id)
      {
        return response()->json(['method' => 'delete', 'id' => $id]);
      }
      public function getCurrentLocation($id)
      {
        return response()->json(['method' => 'getCurrentLocation',
                                'id' => $id]);
      }
      public function setCurrentLocation(Request $request, $id,
                                         $latitude, $longitude)
      {
        return response()->json(['method' => 'setCurrentLocation',
                                 'id' => $id, 'latitude' => $latitude,
                                 'longitude' => $longitude]);
      }
    }
```

The preceding code defined all the methods we have specified in our `app/Http/routes.php` file. For example, we return a simple JSON to test whether each route works fine.

Remember that the main language used in the communication between your microservices is JSON, so all our responses need to be in JSON.

In Lumen, it is very easy to return a JSON response; you only need to use the `json()` method of the response instance, as follows:

```
return response()->json(['method' => 'update', 'id' => $id]);
```

If the value stored in our `$id` variable is `123`, the preceding sentence will return a well-formed JSON response:

```
{
  "method" : "update",
  "id" : 123
}
```

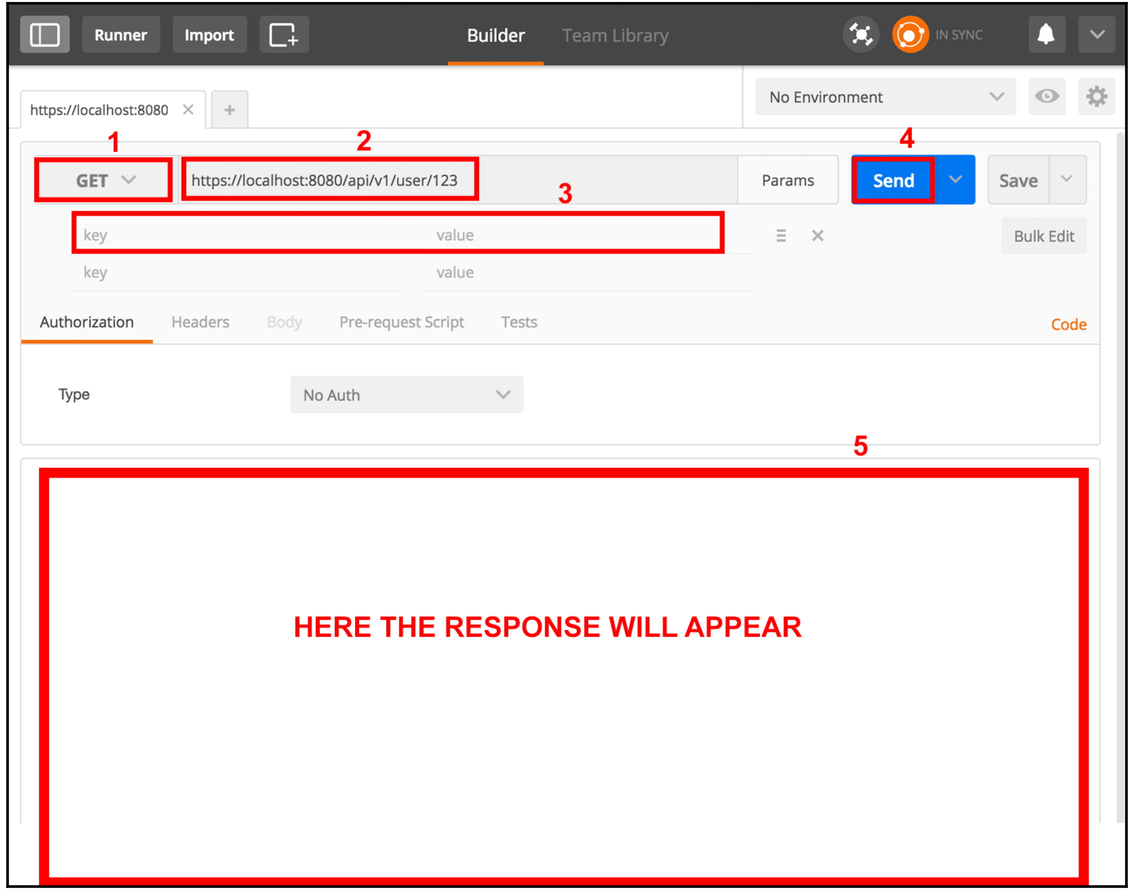
Now, we have everything in place for our `User` microservice.

Perhaps, you are wondering what is the URI assigned to our `get()` method of the `User` microservice in our container environment. It is very easy to find it–simply open the `docker-compose.yml` file, and you can find the port mapping for the `microservice_user_nginx` container. The port mapping we have set up indicates that our localhost 8084 port will redirect the petitions to port 80 of the container. In summary, our URI will be `http://localhost:8084/api/v1/get/123`.

Postman

Our application based on microservices will not have a frontend part; the goal of the API Rest is to create microservices that can be consumed by different platforms (Web, iOS, or Android) just by calling the available methods in the routes file; so in order to execute the calls to our microservices, we will use Postman. It is a tool that allows you to execute the different calls including the parameters you need. Using Postman, it is possible to save the methods to use them in the future.

You can download or install Postman from `https://www.getpostman.com`, as follows:



Overview of the Postman tool

As you can see in the preceding Postman tool screenshot, it has many features, such as saving the requests or setting up different environments; but for now, we only need to know the basic functions to execute calls to our application, which are as follows:

1. Set the verb–GET, POST, PUT, PATCH, or DELETE. Some frameworks cannot reproduce PUT or PATCH calls, so you will need to set the verb POST instead, and include a parameter with key `_method` and value PUT or PATCH.
2. Set the request URL. It is the desired entry point for our application.
3. Add more parameters, if necessary–for example, a parameter to filter results. For POST calls, the Body button will be enabled so that you can send parameters in the body request instead of the URL.

4. Click on SEND to execute the call.
5. The response will appear providing the status code and time in seconds.

Middleware

As we explained in the previous chapters, middleware are very useful in applications based on microservices. Let's explain how you can use them using Lumen.

Lumen has a directory to place all the middleware, so we will create a middleware on the `User` microservice to check whether the consumer has the provided `API_KEY` to communicate with our application.

To identify our consumers, we recommend that you use an `API_KEY`. This practice will avoid unwelcome consumers using our application.

Imagine that we provided our customer with an `API_KEY` with the value `RSAy430_a3eGR`, and it is necessary to send this value in every single request to our application. We can use a middleware to check whether this `API_KEY` was provided. Create a file called `App\Http\Middleware\ApiKeyMiddleware.php`, and place this piece of code in it:

```
<?php
namespace App\Http\Middleware;
use Closure;

class ApiKeyMiddleware
{
  const API_KEY = 'RSAy430_a3eGR';
  public function handle($request, Closure $next)
  {
    if ($request->input('api_key') !== self::API_KEY) {
        die('API_KEY invalid');
    }
    return $next($request);
  }
}
```

Once we have created our middleware, we have to add it to the application. To do this, include the following lines in the `bootstrap/app.php` file:

```
$app->middleware([App\Http\Middleware\ApiKeyMiddleware::class]);
$app->routeMiddleware(['api_key' => App\Http\Middleware
\ApiKeyMiddleware::class]);
```

Now, we can add the middleware to the `routes.php` file. It can be put in different places; you can put it in a single request or even in the entire group, as follows:

```
$app->group([
    'middleware' => 'api_key',
    'prefix' => 'api/v1',
    'namespace' => 'App\Http\Controllers'],
    function($app) {
        $app->get('user', 'UserController@index');
        $app->get('user/{id}', 'UserController@get');
        $app->post('user', 'UserController@create');
```

Give it a try on Postman; make an HTTP POST call to `http://localhost:8084/api/v1/user` . You will see a message that says `API_KEY invalid`. Now make the same call but add a parameter called `API_KEY` with the value `RSAy430_a3eGR`; the request passes the middleware and arrives at the function.

Implementing a microservice call

Now that we know how to make a call, let's create a more complex example. We will build our battle system. As mentioned in the previous chapters, a battle is a fight between two players in order to get secrets from the loser. Our battle system will consist of three rounds, and in each round, there will be a dice throw; the user that wins the most rounds will be the winner and will get a secret from the loser's wallet.

We suggest using some testing development practices (TDD, BDD, or ATDD) as we explained before; you can see some examples in the preceding chapter. We will not include more tests in this chapter.

In the Battle microservice, we can create a function for the battle in the `BattleController.php` file; let's look at a valid structure method:

```
public function duel(Request $request)
{
    return response()->json([]);
}
```

Do not forget to add the endpoint on the `routes.php` file to link the URI to our method:

```
$app->post('battle/duel', 'BattleController@duel');
```

At this point, the `duel` method for the Battle microservice will be available; give it a try with Postman.

Now, we will implement the duel. We need to create a new class for the dice. To store a new class, we will create a new folder called `Algorithm` in the root, and the file `Dice.php` will include the dice methods:

```
<?php
namespace App\Algorithm;
class Dice
{
    const TOTAL_ROUNDS   = 3;
    const MIN_DICE_VALUE = 1;
    const MAX_DICE_VALUE = 6;
    public function fight()
    {
        $totalRoundsWin = [
            'player1' => 0,
            'player2' => 0
        ];

        for ($i = 0; $i < self::TOTAL_ROUNDS; $i++) {
            $player1Result = rand(
                self::MIN_DICE_VALUE,
                self::MAX_DICE_VALUE
            );
            $player2Result = rand(
                self::MIN_DICE_VALUE,
                self::MAX_DICE_VALUE
            );
            $roundResult = ($player1Result <=> $player2Result);
            if ($roundResult === 1) {
                $totalRoundsWin['player1'] =
                $totalRoundsWin['player1'] + 1;
            } else if ($roundResult === -1) {
                $totalRoundsWin['player2'] =
                $totalRoundsWin['player2'] + 1;
            }
        }

        return $totalRoundsWin;
    }
}
```

Once we have developed the `Dice` class, we will call it from the `BattleController` to see who wins a battle. The first thing is to include the `Dice` class on the `BattleController.php` file at the top, and then we can create an instance of the algorithm we will use for the duels (this is a good practice in order to change the duel system in the future; for example, if we want to use a duel based on energy points or card games, we would only need to change the `Dice` class for the new one).

The duel function will return a JSON with the battle results. Please look at the new highlighted code included on the `BattleController.php`:

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Algorithm\Dice;

class BattleController extends Controller
{
    protected $battleAlgorithm = null;
    protected function setBattleAlgorithm()
    {
        $this->battleAlgorithm = new Dice();
    }

    /** ... Code omitted ... **/

    public function duel(Request $request)
    {
        $this->setBattleAlgorithm();
        $duelResult = $this->battleAlgorithm->fight();

        return response()->json(
            [
                'player1'     => $request->input('userA'),
                 'player2'     => $request->input('userB'),
                 'duelResults' => $duelResult
             ]
        );
    }
}
```

Give it a try using Postman; remember that it is an HTTP POST request to the URI `http://localhost:8081/api/v1/battle/duel` (note that we set up the port 8081 for the battle microservice on Docker), and it is necessary to send the parameters `userA` and `userB` with the usernames you want. If everything is correct, you should get a response similar to this:

```
{
    "player1": "John",
    "player2": "Joe",
    "duelResults": {
        "player1": 2,
        "player2": 1
    }
}
```

Request life cycle

The request life cycle is the map of a request until it is returned to the consumer as a response. It is interesting to understand this process in order to avoid issues during the request. Every framework has its own way of doing the request, but all of them are quite similar and follow some basic steps like Lumen does:

1. Every request is managed by `public/index.php`. It does not have much code, it just loads the Composer-generated autoloader definition and creates the instance of the application from `bootstrap/app.php`.
2. The request is sent to HTTP Kernel, which defines some necessary things such as error handling, logging, application environment, and other necessary tasks that should be added before the request is executed. HTTP Kernel also defines a list of middleware that the request must pass before retrieving the application.
3. Once the request passes the HTTP Kernel and arrives at the application, it reaches the routes and tries to match it with the correct one.
4. It executes the controller and the code that correspond to the route and creates and returns a response object.
5. The HTTP headers and the response object content is returned to the client.

This is just a basic example of the request-response workflow; the real process is more complex. You should take into account that the HTTP Kernel is like a big black box that does things that are not visible to the developer in a first instance, so understanding this example is enough for this chapter.

Communication between microservices with Guzzle

One of the most important things in microservices is the communication between them. Most of the time a single microservice doesn't have all the information requested by the consumer, so the microservice needs to call to a different microservice to get the desired data.

For example, adhering to the last example for the duel between two users, if we want to give all the information about the users included in the battle in the same call and we don't have a specific method to get the user information in the Battle microservice, we can request the user information from the user microservice. To achieve this, we can use the PHP core feature cURL or use an external library that wraps cURL, giving an easier interface as `GuzzleHttp`.

To include `GuzzleHttp` in our project, we only need to add the following line in the `composer.json` file of the Battle microservice:

```
{
    // Code omitted
    "require": {
        "php": ">=5.5.9",
        "laravel/lumen-framework": "5.2.*",
        "vlucas/phpdotenv": "~2.2",
        "guzzlehttp/guzzle": "~6.0"
    },
    // Code omitted
}
```

Once we save the changes, we can enter our PHP-FPM container and run the following command:

```
cd /var/www/html && compose update
```

`GuzzleHttp` will be installed and ready to use on the project.

In order to get the user information from the `User` microservice, we will build a method that will give the information to the `Battle` microservice. For now, we will store the user information in a database, so we have an array to store it. In the `app/Http/Controllers/UserController.php` of the `User` microservice, we will add the following lines:

```
<?php

namespace App\Http\Controllers;
use Illuminate\Http\Request;
class UserController extends Controller
{
    protected $userCache = [
        1 => [
            'name' => 'John',
            'city' => 'Barcelona'
        ],
        2 => [
            'name' => 'Joe',
            'city' => 'Paris'
        ]
    ];

    /** ... Code omitted ... **/

    public function get($id)
    {
        return response()->json(
            $this->userCache[$id]
        );
    }

    /** ... Code omitted ... **/
}
```

You can test this new method on Postman by doing a GET call to `http://localhost:8084/api/v1/user/2`; you should get something like this:

```
{
    "name": "Joe",
    "city": "Paris"
}
```

Once we know that the method to get the user information is working, we will call it from the `Battle` microservice. For security reasons, each container on Docker is isolated from the other containers, unless you specify that you want a connection in the links section of the `docker-composer.yml` file. To do so, use the following method:

- Stop Docker containers:

```
docker-compose stop
```

- Edit `docker-compose.yml` file by adding the following line:

```
microservice_battle_fpm:
    build: ./microservices/battle/php-fpm/
    volumes_from:
    - source_battle
    links:
        - autodiscovery
        - microservice_user_nginx
    expose:
        - 9000
    environment:
        - BACKEND=microservice-battle-nginx
        - CONSUL=autodiscovery
```

- Start Docker containers:

```
docker-compose start
```

From now on, the `Battle` microservice should be able to see the `User` microservice, so let's call the `User` microservice in order to get the user information from the Battle microservice. To do this, we need to include the `GuzzleHttp\Client` in the `BattleController.php` file and create a Guzzle instance in the duel function to use it:

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Algorithm\Dice;
use GuzzleHttp\Client;
class BattleController extends Controller
{
    const USER_ENDPOINT = 'http://microservice_user_nginx/api
    /v1/user/';
    /** ... Code omitted ... **/

    public function duel(Request $request)
    {
        $this->setBattleAlgorithm();
```

```
            $duelResult = $this->battleAlgorithm->fight();
            $client = new Client(['verify' => false]);
            $player1Data = $client->get(
            self::USER_ENDPOINT . $request->input('userA'));
            $player2Data = $client->get(
            self::USER_ENDPOINT . $request->input('userB'));
            return response()->json(
                [
                    'player1' => json_decode($player1Data->getBody()),
                    'player2' => json_decode($player2Data->getBody()),
                    'duelResults' => $duelResult
                ]
            );
        }
    }
```

Once we have finished the modifications, we can test it on Postman again by executing the same call as before–`http://localhost:8081/api/v1/battle/duel` (remember to make an HTTP POST call and send the parameters `userA` with value 1 and `userB` with value 2). The response should be similar to this (note that this time the user information is coming from the `User` microservice, although we are calling the `Battle` microservice):

```
    {
        "player1": {
            "name": "John",
            "city": "Barcelona"
        },
        "player2": {
            "name": "Joe",
            "city": "Paris"
        },
        "duelResults": {
            "player1": 0,
            "player2": 3
        }
    }
```

Database operations

In the previous chapters, we explained that you can have single or multiple databases for your application. This is one of the advantages of microservices; you can scale a single microservice when you realize that it is getting a big load by dividing the database into a single database for a specific microservice.

For our example, we will create a single database for the secrets microservice. For storage software, we decided to use **Percona** (a MySQL fork), but feel free to use any database you like.

To create a database container in Docker is very easy. We only need to edit our `docker-compose.yml` file, and change the links section of the `microservice_secret_fpm` service with the following:

```
links:
    - autodiscovery
    - microservice_secret_database
```

In the changes we did, we are telling Docker that now our `microservice_secret_fpm` can communicate with our `microservice_secret_database` container. Let's create our database container. To do this, we only need to define the service in the `docker-compose.yml` file, as follows:

```
microservice_secret_database:
    build: ./microservices/secret/database/
    environment:
        - CONSUL=autodiscovery
        - MYSQL_ROOT_PASSWORD=mysecret
        - MYSQL_DATABASE=finding_secrets
        - MYSQL_USER=secret
        - MYSQL_PASSWORD=mysecret
    ports:
        - 6666:3306
```

In the preceding code, we tell the application where Docker can find the `Dockerfile` where we set up some environment variables and also that we are mapping the port 6666 of our machine to the default Percona port on the container. One important thing that you need to know about Docker and the Percona official image is that using some special environment variables, the container will create a database and some users for you.

You can find all the files you need in our Docker GitHub repository under the `chapter-05-basic-database` tag.

Now that we have our container ready, it's time to set up our database. Lumen provides us with a tool to make migrations and manage them, so we can know if we have our database up to date if we are working with a team. A **migration** is a script to create and roll back operations in our database.

To make a migration in Lumen, first you need to enter your secrets PHP-FPM container. To do this, you only need to open your terminal and execute the following command:

```
docker exec -it docker_microservice_secret_fpm_1 /bin/bash
```

The above command creates an interactive terminal in the container and runs the bash console so that you can start typing your commands. Please ensure that you are on the project root:

```
cd /var/www/html
```

Once you are on the project root, you need to create a migration; this can be done with the following command:

```
php artisan make:migration create_secrets_table
```

The above command will create an empty migration template in the `database/migrations/2016_11_09_200645_create_secrets_table.php` file, as follows:

```
<?php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;
class CreateSecretsTable extends Migration
{
    public function up()
    {
    }
    public function down()
    {
    }
}
```

The preceding piece of code is the sample generated by the artisan command. As you can see, there are two methods in the migration script. Everything you write inside the `up()` method will be used when a migration is executed. In the case of doing a rollback, everything inside the `down()` method will be used to undo your changes. Let's fill our migration script with the following content:

```
<?php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;
class CreateSecretsTable extends Migration
{
    public function up()
    {
        Schema::create(
```

```
                'secrets',
                function (Blueprint $table) {
                    $table->increments('id');
                    $table->string('name', 255);
                    $table->double('latitude');
                    $table->double('longitude')
                        ->nullable();
                    $table->string('location_name', 255);
                    $table->timestamps();
                }
            );
        }
        public function down()
        {
            Schema::drop('secrets');
        }
    }
```

The preceding sample is very easy to understand. In the `up()` method, we are creating a secrets table with some columns. It's a fast and easy way to create a table, which is similar to using a `CREATE TABLE` SQL statement. Our `down()` method will revert all our changes, and in our case, the way of reverting the changes is by removing the secrets table from our database.

Now, you are ready to execute the migration from your terminal with the following command:

```
php artisan migrate
Migrated: 2016_11_09_200645_create_secrets_table
```

The migrate command will run the `up()` method of our migration script and create our secrets table.

If you need to know the status of the execution of your migration scripts, you can perform a `php artisan migrate:status`, and the output will tell you the current status:

```
+------+----------------------------------------+
| Ran? | Migration                              |
+------+----------------------------------------+
| Y    | 2016_11_09_200645_create_secrets_table |
+------+----------------------------------------+
```

At this point, you can connect with your favorite database client to the 6666 port of your machine; our database will be there, ready to be used in our application.

Imagine now that you need to do a rollback of the changes done to your database; it is very easy to do it in Lumen, you only need to run the following command:

```
php artisan migrate:rollback
```

Once we have created the table, we can populate our table by doing seed on Lumen or manually. Our recommendation is that you use seeds, as this is an easy way to keep track of any changes. To populate our new table, we only need to create a new file `database/seeds/SecretsTableSeeder.php` with the following content:

```
<?php
use Illuminate\DatabaseSeeder;
class SecretsTableSeeder extends Seeder
{
    public function run()
    {
        DB::table('secrets')->delete();
        DB::table('secrets')->insert([
            [
                'name' => 'amber',
                'latitude' => 42.8805,
                'longitude' => -8.54569,
                'location_name' => 'Santiago de Compostela'
            ],
            [
                'name' => 'diamond',
                'latitude' => 38.2622,
                'longitude' => -0.70107,
                'location_name' => 'Elche'
            ],
            [
                'name' => 'pearl',
                'latitude' => 41.8919,
                'longitude' => 2.5113,
                'location_name' => 'Rome'
            ],
            [
                'name' => 'ruby',
                'latitude' => 53.4106,
                'longitude' => -2.9779,
                'location_name' => 'Liverpool'
            ],
            [
                'name' => 'sapphire',
                'latitude' => 50.08804,
                'longitude' => 14.42076,
                'location_name' => 'Prague'
```

```
                ]
            ]);
        }
    }
```

In the preceding class, we defined a `run()` method, which will be executed every time we want to fill our database with some data. In our example, we added the different secrets we had hardcoded in our application. Now that we have our `SecretsTableSeeder` class ready, we need to edit the `database/seeds/DatabaseSeeder.php` to add a call to our custom seeder. If you change the `run()` method to match the following piece of code, your microservice will have some data:

```
    public function run()
    {
        $this->call('SecretsTableSeeder');
    }
```

Once you have everything in place, it's time to execute the seeder, so go to the secrets PHP-FPM container again and run the following command:

```
php artisan db:seed
```

If `artisan` throws an error telling you that it can't find the table, it is due to the composer autoloading system. Executing a `composer dump-autoload` will fix your problem, and you can run the `artisan` command again without any problems.

At this point, you will have your secrets table created and populated with some example records.

Working with databases in Lumen is out of the box and uses Eloquent as an ORM.

An **Object-Relational mapping** (**ORM**) is a programming model that transforms the database tables into entities that make the developer's work easier, allowing them to make basic queries faster and use less code.

We recommend using an ORM in order to avoid syntax problems if you want to migrate the database to a different system in the future. As you know, the SQL languages have few differences among them–for example, the way to get a determinate number of rows:

```
SELECT * FROM secrets LIMIT 10 //MySQL
SELECT TOP 10 * FROM secrets //SqlServer
SELECT * FROM secrets WHERE rownum<=10; //Oracle
```

So, if you use an ORM, you do not need to remember the SQL syntax, because it abstracts the developer from the database operations and the developer only needs to think about the development.

The following are advantages of the ORM:

- The security in the data access layer against attacks
- It is easy and fast to work with
- It does not matter what database you use

Using an ORM is recommended when we are developing a public API in order to avoid security problems and make the queries easier and clearer for the rest of the team.

To set up your Lumen project to work with Eloquent, you only need to open the `bootstrap/app.php` file and uncomment the following line (around line 28):

```
$app->withEloquent();
```

Also, you will need to set up the database parameters located in the `.env.example` file, you can find it in the root folder of each microservice. Once you finish editing the file, you need to rename it `.env` (removing `.example` from the file name):

```
DB_CONNECTION=mysql
DB_HOST=microservice_secret_database
DB_PORT=3306
DB_DATABASE=finding_secrets
DB_USERNAME=secret
DB_PASSWORD=mysecret
```

As you can see, we are using the database, user, and password we set up in Docker in the beginning of the Database operations section.

In order to work with our database, we need to create our models, and since we have a `finding_secrets` database, it makes sense to have a secret model in the `app/Models/Secret.php` file:

```
<?php
namespace App\Model;
use Illuminate\Database\Eloquent\Model;
class Secret extends Model
{
    protected $table    = 'secrets';
    protected $fillable = [
        'name',
        'latitude',
        'longitude',
```

```
            'location_name'
        ];
    }
```

The preceding piece of code is very easy to understand; we only had to define the relationship between our model class and the database `$table` and the list of `$fillable` fields. This is the minimum you need for your model.

Fractal is a library that provides a presentation and transformation layer for our RESTful API. Using this library will keep our responses consistent, nice, and clean. To install this library, we only need to open the `composer.json` of our PHP-FPM containers, add `"league/fractal": "^0.14.0"` to the required list of elements and perform a `composer update`.

Another way of installing fractal is by running the following command on your PHP-FMP terminal: `composer require league/fractal`. Note that this command will use the latest version and might not work with our examples.

With fractal installed, now it's time to define our secret transformer. You can think about the transformers as an easy way of having the transformation from your model to a consistent response in one place. Create the `app/Transformers/SecretTransformer.php` file with your IDE with the following content:

```
<?php
namespace App\Transformers;
use App\Model\Secret;
use League\Fractal\Transformer\Abstract;
class SecretTransformer extends TransformerAbstract
{
    public function transform(Secret $secret)
    {
        return [
            'id'        => $secret->id,
            'name'      => $secret->name,
            'location'  => [
                'latitude'  => $secret->latitude,
                'longitude' => $secret->longitude,
                'name'      => $secret->location_name
            ]
        ];
    }
}
```

As you can see from the preceding code, we are specifying the transformation of the secrets model, and because we want all the locations grouped, we added all the location information of the secret inside the location key. In the future, if you need to add new fields or modify the structure, now that everything is in one place, it will make your life as a developer easy.

For our example purpose, we will modify the index method of our secrets controller to return a response from the database using fractal. Open your `app/Http/Controllers/SecretController.php` and insert the following uses:

```
use App\Model\Secret;
use App\Transformers\SecretTransformer;
use League\Fractal\Manager;
use League\Fractal\Resource\Collection;
```

Now, you need to change the `index()`, as follows:

```
public function index(
    Manager $fractal,
    SecretTransformer $secretTransformer,
    Request $request)
{
    $records = Secret::all();
    $collection = new Collection(
        $records,
        $secretTransformer
    );
    $data = $fractal->createData($collection)
        ->toArray();

    return response()->json($data);
}
```

Firstly, we added some object instances that we will need to the method signature, and thanks to the dependency injection built in Lumen, we don't need to do any more work. They will be ready to use inside our method. The definition of our method does the following things:

- Gets all the secret records from the database
- Creates a collection of secrets using our transformer
- The fractal library creates a data array from our collection
- Our controller returns our transformed collection as JSON

If you give it a try with Postman, the response will be similar to this:

```
{
    "data": [
        {
            "id": 1,
            "name": "amber",
            "location": {
                "latitude": 42.8805,
                "longitude": -8.54569,
                "name": "Santiago de Compostela"
            }
        },

        /** Code omitted ** /
        {
            "id": 5,
            "name": "sapphire",
            "location": {
                "latitude": 50.08804,
                "longitude": 14.42076,
                "name": "Prague"
            }
        }
    ]
}
```

All our records are now returned from the database in a consistent way, all with the same structure inside our `"data"` response key.

Error handling

In the following section, we will explain how to validate the input data in our microservice and how to manage the possible errors. It is important to filter the request we are receiving–not only to notify the consumer that the request was not valid, but also to avoid security problems or parameters that we are not expecting.

Validation

Lumen has a fantastic validation system, so we do not need to install anything to validate our data. Note that the following validation rules can be placed on the `routes.php` or on the controller inside every function. We will use it inside the function to be clearer.

To use our database for the validation system, we need to configure it. This is very simple; we just need to create a `config/database.php` file (and the folder) in our root with the following code:

```
<?php
return [
    'default'     => 'mysql',
    'connections' => [
        'mysql' => [
            'driver'    => 'mysql',
            'host'      => env('DB_HOST'),
            'database'  => env('DB_DATABASE'),
            'username'  => env('DB_USERNAME'),
            'password'  => env('DB_PASSWORD'),
            'collation' => 'utf8_unicode_ci'
        ]
    ]
];
```

Then, you have to add the database line in the `bootstrap/app.php` file:

```
$app->withFacades();
$app->withEloquent();
$app->configure('database');
```

Once you have done this, the Lumen validation system is ready. So, let's write the rules to validate the POST method to create a new secret on the `Secrets` microservice:

```
public function create(Request $request)
{
    $this->validate(
      $request,
      [
        'name'          => 'required|string|unique:secrets,name',
        'latitude'      => 'required|numeric',
        'longitude'     => 'required|numeric',
        'location_name' => 'required|string'
      ]
    );
```

In the preceding code, we confirm that the parameters should pass the rules. The field `'name'` is required; it is a string, and also it should be unique in the `secrets` table. The field's `'latitude'` and `'longitude'` are numeric and required too. Also, the `'location_name'` field is required and it is a string.

In the Lumen documentation (`https://lumen.laravel.com/docs`), you can check out all the available options to validate your inputs.

You can try it out in your Postman; create a POST request with the following `application/json` parameters to check a failed insertion (note that you can also send it like form-data key values):

```
{
    "name": "amber",
    "latitude":"1.23",
    "longitude":"-1.23",
    "location_name": "test"
}
```

The preceding request will try to validate a new secret with the same name as other previous records. From our validation rules, we are not allowing the consumers to create new secrets with the same name, so our microservice will respond with a `422` error with the following body:

```
{
    "name": [
        "The name has already been taken."
    ]
}
```

Note that the status codes (or error codes) are very important to inform your consumers what happened with their requests; if everything is fine, it should respond with a `200` status code. Lumen returns a `200` status code by default.

In `Chapter 11`, *Best Practices and Conventions*, we will see a full list of all the available codes that you can use in your application.

Once the validation rules pass, we should save the data on the database. This is very simple in Lumen, just do this:

```
$secret = Secret::create($request->all());
if ($secret->save() === false) {
    // Manage Error
}
```

After this, we will have our new record available in the database. Lumen provides other methods to create for other tasks such as fill, update, or delete.

Manage exceptions

It is necessary to know that we have to manage the possible errors that happen in our application. To do this, Lumen provides us with a list of exceptions that can be used.

So, now we will try to get an exception when trying to call another microservice. To do this, we will call the secret microservice from the user microservice.

Please remember that for security reasons if you didn't link a container with another container, they can't see each other. Edit your `docker-compose.yml` and add the link from the `microservice_user_fpm` to the `microservice_secret_nginx`, as follows:

```
microservice_user_fpm:
    build: ./microservices/user/php-fpm/
    volumes_from:
        - source_user
    links:
        - autodiscovery
        - microservice_secret_nginx
    expose:
        - 9000
    environment:
        - BACKEND=microservice-user-nginx
        - CONSUL=autodiscovery
```

Now, you should start your containers again:

```
docker-compose start
```

Also, remember that we need to install `GuzzleHttp` as we did before on the `Battle` microservice and on the `User` microservice in order to call the `Secret` microservice.

We will create a new function on the `User` microservice to show the secrets kept in a `user` wallet.

Add this to `app/Http/routes.php`:

```
$app->get(
    'user/{id}/wallet',
    'UserController@getWallet'
);
```

Then, we will create the method to get a secret from the `user` wallet–for example, look at this:

```
public function getWallet($id)
{
    /* ... Code ommited ... */
    $client = new Client(['verify' => false]);
    try {
        $remoteCall = $client->get(
            'http://microservice_secret_nginx
        /api/v1/secret/1');
    } catch (ConnectException $e) {
        /* ... Code ommited ... */
        throw $e;
    } catch (ServerException $e) {
        /* ... Code ommited ... */
    } catch (Exception $e) {
        /* ... Code ommited ... */
    }
      /* ... Code ommited ... */
}
```

We are calling the `Secret` microservice, but we will modify the URI in order to get a `ConnectException`, so please modify it:

```
$remoteCall = $client->get(
    'http://this_uri_is_not_going_to_work'
);
```

Give it a try on Postman; you will receive a `ConnectException` error.

Now, set the URI correctly again and put some wrong code on the secret microservice side:

```
public function get($id)
{
  this_function_does_not_exist();
}
```

The preceding code will return an error **500** for the secret microservice; but we are calling it from the `User` microservice, so now we will receive a `ServerException` error.

There are hundreds of kinds of errors that you can manage in your microservice by catching them. In Lumen, all the exceptions are handled by the `Handler` class (located at `app/Exceptions/Handler.php`). This class has two defined methods:

- `report()`: This allows us to send our exceptions to external services–for example, a centralized logging system.
- `render()`: This transforms our exception into an HTTP response.

We will be updating the `render()` method to return custom error messages and error codes. Imagine that we want to catch a Guzzle `ConnectException` and return a more friendly and easy-to-manage error. Take a look at the following code:

```
/** Code omitted **/
use SymfonyComponentHttpFoundationResponse;
use GuzzleHttpExceptionConnectException;

/** Code omitted **/

public function render($request, Exception $e)
{
    switch ($e) {
        case ($e instanceof ConnectException) :
            return response()->json(
                [
                    'error' => 'connection_error',
                    'code'  => '123'
                ],
                Response::HTTP_SERVICE_UNAVAILABLE
            );
            break;
        default :
            return parent::render($request, $e);
           break;
    }
}
```

Here, we are detecting the Guzzle `ConnectException` and giving a custom error message and code. Using this strategy helps us to know what is failing and allows us to act according to the error we are dealing with. For example, we can assign the code `123` to all our connection errors; so, when we detect this issue, we can avoid a cascade failure in other services or notify the developer.

Async and queue

In microservices, the queues are one of the most important things that help increase the performance and reduce the execution time.

For instance, if you have to send an e-mail to a customer when the customer finishes the registration process of your application, the application does not need to send it at that moment; it can be put in a queue to be sent a few seconds later when the server is not as busy. Also, it is async because the customer does not need to wait for the e-mail. The application will display the message *Registration finished* and the e-mail will be put in the queue and processed at the same time.

Another example is when you need to do very heavy workloads, so you can have a dedicated machine with better hardware to do the tasks.

One of the most well-known in-memory data structure stores is **Redis**. You can use it as a database, a cache layer, as a message broker, or even as a queue storage. One of the key points of Redis is its support for different structure types, such as strings, hashes, lists, and sorted sets among others. This software was designed to be easy to manage and to have a high performance and, for these reasons, it is a de facto standard in the Web industry.

One of the main uses of Redis is as a cache storage. You can store your data forever or add an expiration time, without having to worry about how and when you need to remove the data; Redis will do it for you. Due to the easy use, great support and libraries available, Redis fits any project of any scale.

We will build an example on the `User` microservice to send e-mails using a queue built on top of Redis.

Lumen provides us with a queue system using the database; but there are other options available using external services. In our example, we will use Redis, so let's see the steps to install it on Docker.

Open the `docker-compose.yml` and add the following container description:

```
microservice_user_redis:
    build: ./microservices/user/redis/
    links:
        - autodiscovery
    expose:
        - 6379
    ports:
        - 6379:6379
```

You also need to update the links section of the `microservice_user_fpm` container to match the following:

```
links:
    - autodiscovery
    - microservice_secret_nginx
    - microservice_user_redis
```

In the preceding pieces of code, we defined a new container for Redis, and we linked it to the `microservice_user_fpm` container. Now open the `microservices/user/redis/Dockerfile` file and add the following code to have the latest Redis version available:

```
FROM redis:latest
```

To use Redis in our Lumen project, we need to install a few dependencies through composer. So, open your `composer.json` and add the following lines to the required section and do a composer update inside the user PHP-FPM container:

```
"predis/predis": "~1.0",
"illuminate/redis": "5.2.*"
```

For email support, you only need to add the following line to the require section of your composer.json file:

```
"illuminate/mail": "5.2.*"
```

Once Redis is installed, we need to set up the environment. Firstly, we need to add the following lines on the `.env` file:

```
QUEUE_DRIVER=redis
CACHE_REDIS_CONNECTION=default
REDIS_HOST=microservice_user_redis
REDIS_PORT=6379
REDIS_DATABASE=0
```

Now, we need to add the Redis configuration on the `config/database.php` file; if you have any other databases added (for example, MySQL), put this after that, but inside the return array:

```
<?php
return [
    'redis' => [
        'client'  => 'predis',
        'cluster' => false,
        'default' => [
            'host'     => env('REDIS_HOST', 'localhost'),
```

```
                'password' => env('REDIS_PASSWORD', null),
                'port'     => env('REDIS_PORT', 6379),
                'database' => env('REDIS_DATABASE', 0),
            ],
        ]
    ];
```

Also, it is necessary to copy the `vendor/laravel/lumen-framework/config/queue.php` file to `config/queue.php`.

Finally, do not forget to register everything on the `bootstrap/app.php` file and add the following lines, so our application is able to read the configuration we just set up:

```
$app->register(
    Illuminate\Redis\RedisServiceProvider::class
);
$app->configure('database');
$app->configure('queue');
```

So now, we will explain how to build a queue in our `User` microservice. Imagine that in the application, when new users are created, we want to give them the first secret as a gift; so after the user creation, we will call the secret microservice in order to get the first secret for the user. This is not a priority, so that is the reason why we will use a queue to do this task.

Create a new file on `app/Jobs/GiftJob.php` with the following code:

```
<?php
namespace AppJobs;
use GuzzleHttpClient;
class GiftJob extends Job
{
    public function __construct()
    {
    }

    public function handle()
    {
        $client = new Client(['verify' => false]);
        $remoteCall = $client->get(
            'http://microservice_secret_nginx
            /api/v1/secret/1'
        );
        /* Do stuff with the return from a remote service, for
        example save it in the wallet */
    }
}
```

You can modify the class construction to pass data to your job, for example, an object instance with all the user information.

Now, we need to instance the job from our `app/Http/Controllers/UserController.php` controller:

```
use AppJobsGiftJob;
public function create(Request $request)
{
    /* ... Code omitted (validate & save data) ... */
    $this->dispatch(new GiftJob());
    /* ... Code omitted ... */
}
```

Once the queue job is done, we have to start the queue worker in the background. The following code will do the work for you, and it will keep running until the thread dies, you can add a supervisor to ensure that the queue continues working:

```
php artisan queue:work
```

You can give this a try on Postman by calling `http://localhost:8084/api/v1/user`. Once you call this, Lumen will put the work on Redis, and it will be available for the queue worker. Once the worker gets and processes the task from Redis, you will see the following next message in the terminal:

```
[2016-11-13 17:59:23] Processed: AppJobsGiftJob
```

Lumen provides us with more possibilities for the queues. For example, you can set priorities for the queue, specify a time-out for a job or even set a delay for the task. You can find this information in the Lumen documentation.

Caching

Many times consumers request the same things, and the application returns the same information. In this scenario, caching is the solution to avoid constantly processing the same request and returning the required data faster.

Caching is used for data that does not change frequently in order to have a precalculated response without processing the request. The workflow is as follows:

1. The first time that the consumer requests some information, the application processes the request and gets the required data.

2. It saves the required data for that request in the cache with an expiration time that we define.
3. It returns the data to the consumer.

Next time that a consumer requests something you need to do the following:

1. Check whether the request is in the application cache and it has not expired.
2. Return the data located in the cache.

So, in our example, we will use caching in the location microservice in order to avoid requesting the closest secrets multiple times.

The first thing that we need to use a cache layer for in our application is a container with Redis (you can find other cache software out there, but we decided to use Redis as it is very easy to install and use). Open the `docker-compose.yml` file and add the new container definition, as follows:

```
microservice_location_redis:
    build: ./microservices/location/redis/
    links:
        - autodiscovery
    expose:
        - 6379
    ports:
        - 6380:6379
```

Once we add the container, you need to update the links section for the `microservice_location_fpm` definition to connect to our new Redis container as follows:

```
links:
    - autodiscovery
    - microservice_location_redis
```

In this case, our `docker/microservices/location/redis/Dockerfile` file will only contain the following (feel free to add more things to the container if you want):

```
FROM redis:latest
```

Do not forget to execute `docker-compose stop` to successfully kill all the containers and start them again with our changes with a `docker-compose up -d`. You can check whether the new container is running by executing `docker ps` in your terminal.

Now is time to make some changes in the source of our location microservice to use our new cache layer. The first change we need to make is on the `composer.json`; add the following required libraries to the `"require"` section:

```
"predis/predis": "~1.0",
"illuminate/redis": "5.2.*"
```

Once you make the changes to the `composer.json` file, remember to do a composer update to get the libraries.

Now, open the location microservice `.env` file to add the Redis setup, as follows:

```
CACHE_DRIVER=redis
CACHE_REDIS_CONNECTION=default
REDIS_HOST=microservice_location_redis
REDIS_PORT=6379
REDIS_DATABASE=0
```

Since our environment variables are now set up, we need to create the `config/database.php` with the following content:

```
<?php
return [
    'redis' => [
        'client'  => 'predis',
        'cluster' => false,
        'default' => [
            'host'     => env('REDIS_HOST', 'localhost'),
            'password' => env('REDIS_PASSWORD', null),
            'port'     => env('REDIS_PORT', 6379),
            'database' => env('REDIS_DATABASE', 0),
        ],
    ]
];
```

In the preceding code, we defined how to connect to our Redis container.

Lumen comes without the cache configuration, so you can copy the `vendor/laravel/lumen-framework/config/cache.php` file into `config/cache.php`.

We will need to make some small adjustments to our `bootstrap/app.php`–uncomment `$app->withFacades();` and add the following lines:

```
$app->configure('database');
$app->configure('cache');
$app->register(
```

```
    Illuminate\Redis\RedisServiceProvider::class
);
```

We will change our `getClosestSecrets()` method to use cache instead of calculating the closest secrets every time. Open the `app/Http/Controllers/LocationController.php` file and add the required use for cache:

```
use Illuminate\Support\FacadesCache;

    /* ... Omitted code ... */
const DEFAULT_CACHE_TIME = 1;
public function getClosestSecrets($originPoint)
{
    $cacheKey = 'L' . $originPoint['latitude'] .
    $originPoint['longitude'];
    $closestSecrets = Cache::remember(
        $cacheKey,
        self::DEFAULT_CACHE_TIME,
        function () use($originPoint) {
            $calculatedClosestSecrets = [];
            $distances = array_map(
                function($item) use($originPoint) {
                    return $this->getDistance(
                        $item['location'],
                        $originPoint
                    );
                },
                self::$cacheSecrets
            );
            asort($distances);
            $distances = array_slice(
                $distances,
                0,
                self::MAX_CLOSEST_SECRETS,
                true
            );
            foreach ($distances as $key => $distance) {
                $calculatedClosestSecrets[] =
                self::$cacheSecrets[$key];
            }

            return $calculatedClosestSecrets;
     });

    return $closestSecrets;
}
/* ... Omitted code ... */
```

In the preceding code, we changed the implementation of the method by adding the layer cache; so instead of always calculating the closest points, we first check on our cache with `remember()`. If nothing is returned from the cache, we make the calculation and store the result.

Another option for saving data in Lumen cache is to use `Cache::put('key', 'value', $expirationTime);` where the `$expirationTime` can be the time in minutes (integer) or a date object.

The key is defined by you, so a good practice is generating a key that you can remember in order to regenerate in the future. In our example, we defined the key using `L` (for location) and then the `latitude` and `longitude`. However, if you are going to save an ID, it should be included as part of the key.

Working with our cache layer is easy in Lumen.

To obtain elements from the cache, you can use "`get`". It allows two parameters–the first one is to specify the key you want (and is required), and the second one is the value to use if the key is not stored in cache (and obviously is optional):

```
$value = Cache::get('key', 'default');
```

A similar method to store data available is `Cache::forever($cacheKey, $cacheValue);` this call will store the `$cacheValue` identified by `$cacheKey` in our cache layer, forever, until you delete or update it.

If you don't specify the expiration time for your stored elements, it is important to know how to remove them. In Lumen, you can do it with `Cache::forget($cacheKey);` if you know the `$cacheKey` assigned to an element. In the case of needing to remove all the elements stored in the cache, we can do this with a simple `Cache::flush();`.

Summary

In this chapter, you have learned how to develop the different parts of an application based on microservices. Now, you have the required knowledge to deal with database storage, cache, communication between microservices, queues, and the request workflow from the point of entry to the application (routes) and the validation of the data until the time it is given to the consumer. In the next chapter, you will learn how to monitor your application in order to avoid and fix issues that happen during the application execution process.

6
Monitoring

In the last chapter, we spent some time developing our example application. Now it is time to start with more advanced topics. In this chapter, we will show you how to monitor your microservices application. Keeping a track of everything that is happening in your application will help you know the overall performance at any time, and you can even find issues and bottlenecks.

Debugging and profiling

Debugging and profiling is very necessary in the development of a complex and large application, so let's explain what they are and how we can take advantage of these kinds of tools.

What is debugging?

Debugging is the process of identifying and fixing errors in programming. It is mainly a manual task in which developers need to use their imagination, intuition, and have a lot of patience.

Most of the time, it is necessary to include new instructions in the code to read the value of variables at a concrete point of execution or code to stop the execution in order to know whether it is passing through a function.

However, this process can be managed by the debugger. This is a tool or application that allows us to control the execution of our application in order to follow each executed instruction and find the bugs or errors, avoiding having to add code instructions in our code.

The debugger uses an instruction called breakpoint. A **breakpoint** is, as its name suggests, a point at which the application stops in order to be driven by the developer to decide what to do. At that point, the debugger gives different information about the current status of the application.

We will see more about the debugger and breakpoint later on.

What is profiling?

Like debugging, profiling is a process to identify if our application is working properly in terms of performance. Profiling investigates the application's behavior in order to know the dedicated time to execute different parts of code to find bottlenecks or optimize them in terms of speed or consumed resources.

Profiling is usually used during the development process as part of debugging, and it is necessary to measure it in proper environments by specialists to get real data from it.

There are four different kinds of profilers: based on events, statistics, tools to support code, and simulation.

Debugging and profiling in PHP with Xdebug

Now we will install and set up Xdebug in our project. This must be installed on our IDE, so depending on which one you use, this process will be different, but the steps to follow are quite similar. In our case, we will install it on PHPStorm. Even if you use a different IDE, after installing Xdebug, the workflow for debugging your code in any IDE will largely be the same.

Debugging installation

To install Xdebug on our Docker, we should modify the proper `Dockerfile` file. We will install it on the user microservices, so open the `docker/microservices/user/php-fpm/Dockerfile` file and add the following highlighted lines:

```
FROM php:7-fpm

RUN apt-get update && apt-get -y install
git g++ libcurl4-gnutls-dev libicu-dev libmcrypt-dev libpq-dev libxml2-dev
unzip zlib1g-dev
&& git clone -b php7 https://github.com/phpredis/phpredis.git
/usr/src/php/ext/redis
```

```
&& docker-php-ext-install curl intl json mbstring mcrypt pdo pdo_mysql
redis xml
&& apt-get autoremove && apt-get autoclean
&& rm -rf /var/lib/apt/lists/*

RUN apt-get update && apt-get upgrade -y && apt-get autoremove -y
&& apt-get install -y git libmcrypt-dev libpng12-dev libjpeg-dev libpq-dev
mysql-client curl
&& rm -rf /var/lib/apt/lists/*
&& docker-php-ext-configure gd --with-png-dir=/usr --with-jpeg-dir=/usr
&& docker-php-ext-install mcrypt gd mbstring pdo pdo_mysql zip
&& pecl install xdebug
&& rm -rf /tmp/pear
&& echo "zend_extension=$(find /usr/local/lib/php/extensions/ -name
xdebug.so)n" >> /usr/local/etc/php/conf.d/xdebug.ini
&& echo "xdebug.remote_enable=on" >> /usr/local/etc/php/conf.d/xdebug.ini
&& echo "xdebug.remote_autostart=off" >>
/usr/local/etc/php/conf.d/xdebug.ini
&& echo "xdebug.remote_port=9000" >> /usr/local/etc/php/conf.d/xdebug.ini
&& curl -sS https://getcomposer.org/installer | php -- --install-
dir=/usr/local/bin --filename=composer

RUN echo 'date.timezone="Europe/Madrid"' >>
/usr/local/etc/php/conf.d/date.ini
RUN echo 'session.save_path = "/tmp"' >>
/usr/local/etc/php/conf.d/session.ini

{{ Omited code }}

RUN curl -sSL https://phar.phpunit.de/phpunit.phar -o /usr/bin/phpunit &&
chmod +x /usr/bin/phpunit

ADD ./config/php.ini /usr/local/etc/php/
CMD [ "/usr/local/bin/containerpilot",
"php-fpm",
"--nodaemonize"]
```

The first highlighted block is necessary to `install xdebug`. The `&& pecl install xdebug` line is used to install Xdebug using PECL, and the rest of the lines set the parameters on the `xdebug.ini` file. The second one is to copy the `php.ini` file from our local machine to Docker.

It is also necessary to set some values on the `php.ini` file, so open it, it is located on `docker/microservices/user/php-fpm/config/php.ini`, and add the following lines:

```
memory_limit = 128M
post_max_size = 100M
upload_max_filesize = 200M

[Xdebug]
xdebug.remote_host=YOUR_LOCAL_IP_ADDRESS
```

You should enter your local IP address instead of `YOUR_LOCAL_IP_ADDRESS` in order to be visible in Docker, so Xdebug will be able to read our code.

Your local IP address is your IP inside your network, not the public one.

Now, you can make the build in order to install everything necessary to debug by executing the following command:

```
docker-compose build microservice_user_fpm
```

This can take a few minutes. Xdebug will be installed once this is finished.

Debugging setup

Now it is time to set up Xdebug on our favorite IDE. As we said before, we will use PHPStorm, but feel free to use any other IDE.

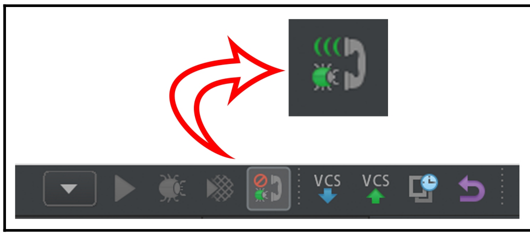
We have to create a server on the IDE, in PHPStorm this is done by navigating to **Preferences** | **Languages & Frameworks** | **PHP**. So, add a new one and set the `name` to `users`, for example, `host` to `localhost`, `port` to `8084`, and `debugger` to `xdebug`. It is also necessary to enable **Use path mappings** in order to map our routes.

Now, we need to navigate to **Tools** | **DBGp proxy – configuration** and ensure that the IDE key field is set to `PHPSTORM`, `Host` to `users` (this name must be the same one you entered on the servers section), and `Port` to `9000`.

Stop and start Docker by executing the following commands:

```
docker-compose stop
docker-compose up -d
```

Set PHPStorm to be able to listen to the debugger:



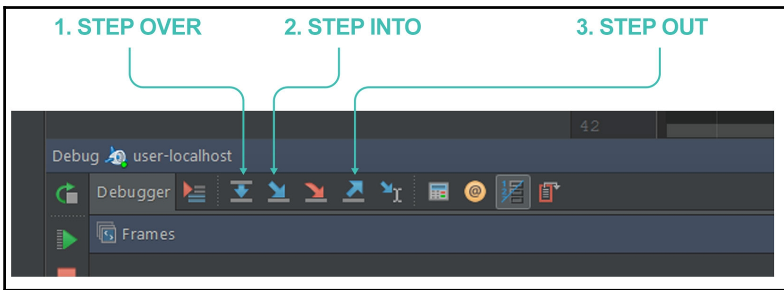
Xdebug button to listen to connections in PHPStorm

Debugging the output

Now you are ready to view the debugger results. You just have to set the breakpoints in your code and the execution will stop at that point, giving you all the data values. To do this, go to your code, for example, on the `UserController.php` file, and click on the left side of a line. It will create a red point; this is a breakpoint:

Breakpoint in PHPStorm

Now, you have the breakpoint set and the debugger running, so it is time to make a call with Postman to try the debugger. Give the breakpoint a try by executing a POST call to `http://localhost:8084/api/v1/user` with the `api_key = RSAy430_a3eGR and XDEBUG_SESSION_START = PHPSTORM` parameters. The execution will stop at the breakpoint and, from there, you have the execution control:



Debugger console in PHPStorm

Note that you have all the current values for the parameters on the variables side. In this case, you can see the `test` parameter set to `"this is a test"`; we assigned this value two lines before the breakpoint.

As we said, now we have control of the execution; the three basic functions are as follows:

1. **Step over:** This continues the execution with the following line.
2. **Step into:** This continues the execution inside a function.
3. **Step out:** This continues the execution outside a function.

All these basic functions are executed step by step, so it will stop in the next line, it does not need any other breakpoints.

As you can see, this is very useful to find errors in your code.

Profiling installation

Once we have Xdebug installed, we just need to add the following lines on the `docker/microservices/user/php-fpm/Dockerfile` file to enable the profiling:

```
RUN apt-get update && apt-get upgrade -y && apt-get autoremove -y
&& apt-get install -y git libmcrypt-dev libpng12-dev libjpeg-dev libpq-dev
mysql-client curl
&& rm -rf /var/lib/apt/lists/*
&& docker-php-ext-configure gd --with-png-dir=/usr --with-jpeg-dir=/usr
```

```
&& docker-php-ext-install mcrypt gd mbstring pdo pdo_mysql zip
&& pecl install xdebug
&& rm -rf /tmp/pear
&& echo "zend_extension=$(find /usr/local/lib/php/extensions/ -name
xdebug.so)n" >> /usr/local/etc/php/conf.d/xdebug.ini
&& echo "xdebug.remote_enable=onn" >> /usr/local/etc/php/conf.d/xdebug.ini
&& echo "xdebug.remote_autostart=offn" >>
/usr/local/etc/php/conf.d/xdebug.ini
&& echo "xdebug.remote_port=9000n" >> /usr/local/etc/php/conf.d/xdebug.ini
&& echo "xdebug.profiler_enable=onn" >>
/usr/local/etc/php/conf.d/xdebug.ini
&& echo "xdebug.profiler_output_dir=/var/www/html/tmpn" >>
/usr/local/etc/php/conf.d/xdebug.ini
&& echo "xdebug.profiler_enable_trigger=onn" >>
/usr/local/etc/php/conf.d/xdebug.ini
&& curl -sS https://getcomposer.org/installer | php -- --install-
dir=/usr/local/bin --filename=composer
```

With `profiler_enable`, we enable the profiler and the output directory is set by `profiler_output_dir`. This directory should exist on our user microservice in order to get the profiler output files. So, if it is not yet created, do it now on `/source/user/tmp`.

Now, you can make the build in order to install everything necessary to debug by executing the following command:

```
docker-compose build microservice_user_fpm
```

This can take a few minutes. Xdebug will be installed once this is finished.

Profiling setup

It does not need to be set up, so just stop and start Docker by executing the following commands:

```
docker-compose stop
docker-compose up -d
```

Set PHPStorm to be able to listen to the debugger as we did with the debugger.

Analyzing the output file

To generate the profiling file, we need to execute a call as we did before with Postman, so feel free to execute the method you want. It will generate a file located on the folder we made before with name `cachegrind.out.XX`.

If you open this file, you will note that it is difficult to understand, but there are some tools to read this type of content. PHPStorm has a tool located on **Tools | Analyze Xdebug Profiler Snapshot**. Once you open it, you can select the file to analyze and then the tool will show you a detailed analysis of all the files and functions executed in the call. Displaying the time spent, times called, and other interesting things are very useful to optimize your code and find bottlenecks.

Error handling

Error handling is the way we manage the errors and exceptions in our application. This is very important in order to have all the possible errors that can happen in our development detected and organized.

What is error handling?

The term *error handling* is used in development to refer to the process of responding to the occurrence of an exception during the execution.

Usually, the appearance of exceptions breaks the normal workflow of an application execution and executes a registered exception handler, giving us more information about what is happening and, sometimes, how we can avoid the exception.

The way that PHP handles the errors is very basic. A default error message is composed of the filename, line, and a little description about the error that the browser receives. In this chapter, we will see three different ways to handle errors.

Why is error handling important?

The majority of applications are very complex and large, and they are also developed by different people or even different teams if we are working on an application based on microservices as we are doing here. Can you imagine all the potential bugs that would appear in a project if we mix all these things? It is impossible to be aware of all the possible issues that an application can have or the users will find in it.

So, error handling helps in the following two ways:

- **Users or consumers**: In microservices, error handling is very useful because it allows the consumers to know the possible problems that an API has and maybe they can figure out if it is a problem related to the introduced parameters in the API call, or the file size of an image. Also, in microservices, it is very useful to use different status codes for the errors in order to let the consumer know what is happening. You can find these codes in the `Chapter 11`, *Best Practices and Conventions*.

  On a commercial website, the error handling avoids showing strange messages like `PHP Fatal error: Cannot access empty property` to the users or customers. Instead of this, it can say something simple, such as `There is an error. Please contact us.`

- **Developers or yourself**: It allows the rest of the team and even yourself to be aware of any errors in your application, helping you to debug possible issues. There are many tools to get these kinds of bugs and send them to you by e-mail, written in a log file, or put on an event log, detailing the error tracking, function parameters, database calls, and more interesting things.

Challenges when managing error handling with microservices

As mentioned earlier, we will explain three different ways to handle errors. When we are working with microservices, we have to monitor all the possible errors in order to let the microservices know what the problem is.

Basic die() function

In PHP, the basic way to handle errors is using the die() command. Let's look at an example. Imagine that we want to open a file:

```
<?php
  $file=fopen("test.txt","r");
?>
```

When the execution arrives at that point and tries to open the file called `test.txt`, if the file does not exist PHP will throw an error like this:

```
Warning: fopen(test.txt) [function.fopen]: failed to open stream:
No such file or directory in /var/www/html/die_example.php on line 2
```

To avoid the error message, we can use the `die()` function with the reason written in it so that the execution does not continue:

```
<?php
  if(!file_exists("test.txt")) {
    die("The file does not exist");
  } else {
    $file=fopen("test.txt","r");
  }
?>
```

This is just an example of the basic error handling on PHP. Obviously, there are better ways to manage this, but this is the minimum required to manage errors. In other words, avoiding the automatic error message from the PHP application is more efficient than stopping the execution manually and giving a human-language reason to the user.

Let's look at an alternate way to manage this.

Custom error handling

Creating a system to manage the errors in your application is a better practice than using the `die()` function. Lumen provides us with this system and it is already configured when it is installed.

We can configure it by setting other parameters. The first thing is the error detail. It is possible to get more information about the errors by setting it to `true`. To do this, it is necessary to add the `APP_DEBUG` value on your `.env` file and set it to `true`. This is the recommended way to work on the development environment so that the developers can know more about the issues of the application, but once the application is on the production server, this value should be set to `false` in order to avoid giving more information to the users.

This system manages all the exceptions through the `AppExceptionsHandler` class. This class contains two methods: `report` and `render`. Let's explain them.

Report method

The `Report` method is used to log the exceptions that happen in your microservice or it is even possible to send them to an external tool, such as Sentry. We will go through this in the chapter.

As mentioned, this method only logs the problem on the base class, but you can manage the different exceptions however you want. Check how you can do this in the following example:

```
public function report(Exception $e)
{
  if ($e instanceof CustomException) {
    //
  } else if ($e instanceof OtherCustomException) {
    //
  }
  return parent::report($e);
}
```

The way to manage the different errors is `instanceof`. As you can see, in the preceding example, you can have different responses for each exception type.

It is also possible to ignore some exception types by adding a variable to the `$dontReport` class. It is an array of different exceptions that you do not want to report. If we do not use this variable on the `Handle` class, only the `404` errors will be ignored by default.

```
protected $dontReport = [
  HttpException::class,
  ValidationException::class
];
```

Render method

If the `report` method is used to help developers or yourself, the render method is to help users or consumers. This method gives an exception in an HTTP response that will be sent back to the user if it is a website and to a consumer if it is an API.

By default, the exception is sent to the base class to generate a response, but it can be modified. Take a look at this code:

```
public function render($request, Exception $e)
{
  if ($e instanceof CustomException) {
    return response('Custom Message');
  }
  return parent::render($request, $e);
}
```

As you can see, the `render` method receives two parameters: the request and the exception. With these parameters, you can make a proper response for your users or consumers, giving the information you want to give for each exception. For example, by giving an error code to the consumers of your API, they can check it on the API documentation. Take a look at the following example:

```
public function render($request, Exception $e)
{
  if ($e instanceof CustomException) {
    return response()->json([
        'error' => $e->getMessage(),
        'code' => 44 ,
      ],
    Response::HTTP_UNPROCESSABLE_ENTITY);
  }
  return parent::render($request, $e);
}
```

The consumer will receive an error message with the `code 44`; this should be on our API documentation, and the proper status code. Obviously, this can be different in order to match your consumer's needs.

Error handling with Sentry

It is even better having a system to monitor the errors. There are a lot of error-tracking systems on the market but one that stands out is Sentry, a real-time cross-platform error tracking system that provides us with clues to understand what is happening in our microservices. An interesting feature is its support of notifications by e-mail or other medium.

Using a well-known system benefits your application, you are using a trusted and well-known tool, which in our case has an easy integration with our framework, Lumen.

The first thing we need to do is install Sentry in our Docker environment; so, as always, stop all your containers with `docker-compose stop`. Once all the containers are stopped, open the `docker-compose.yml` file and add the following containers:

```
sentry_redis:
  image: redis
expose:
  - 6379

sentry_postgres:
  image: postgres
  environment:
    - POSTGRES_PASSWORD=sentry
    - POSTGRES_USER=sentry
  volumes:
    - /var/lib/postgresql/data
  expose:
    - 5432

  sentry:
    image: sentry
  links:
    - sentry_redis
    - sentry_postgres
  ports:
    - 9876:9000
  environment:
    SENTRY_SECRET_KEY: mymicrosecret
    SENTRY_POSTGRES_HOST: sentry_postgres
    SENTRY_REDIS_HOST: sentry_redis
    SENTRY_DB_USER: sentry
    SENTRY_DB_PASSWORD: sentry

  sentry_celery-beat:
    image: sentry
  links:
    - sentry_redis
    - sentry_postgres
    command: sentry celery beat
  environment:
    SENTRY_SECRET_KEY: mymicrosecret
    SENTRY_POSTGRES_HOST: sentry_postgres
    SENTRY_REDIS_HOST: sentry_redis
    SENTRY_DB_USER: sentry
    SENTRY_DB_PASSWORD: sentry

  sentry_celery-worker:
    image: sentry
```

```
    links:
      - sentry_redis
      - sentry_postgres
      command: sentry celery worker
    environment:
      SENTRY_SECRET_KEY: mymicrosecret
      SENTRY_POSTGRES_HOST: sentry_postgres
      SENTRY_REDIS_HOST: sentry_redis
      SENTRY_DB_USER: sentry
      SENTRY_DB_PASSWORD: sentry
```

In the preceding code, we firstly created a specific `redis` and `postgresql` container that will be used by Sentry. Once we had the required data storage containers, we added and linked the different containers that are the core of Sentry.

```
docker-compose up -d sentry_redis sentry_postgres sentry
```

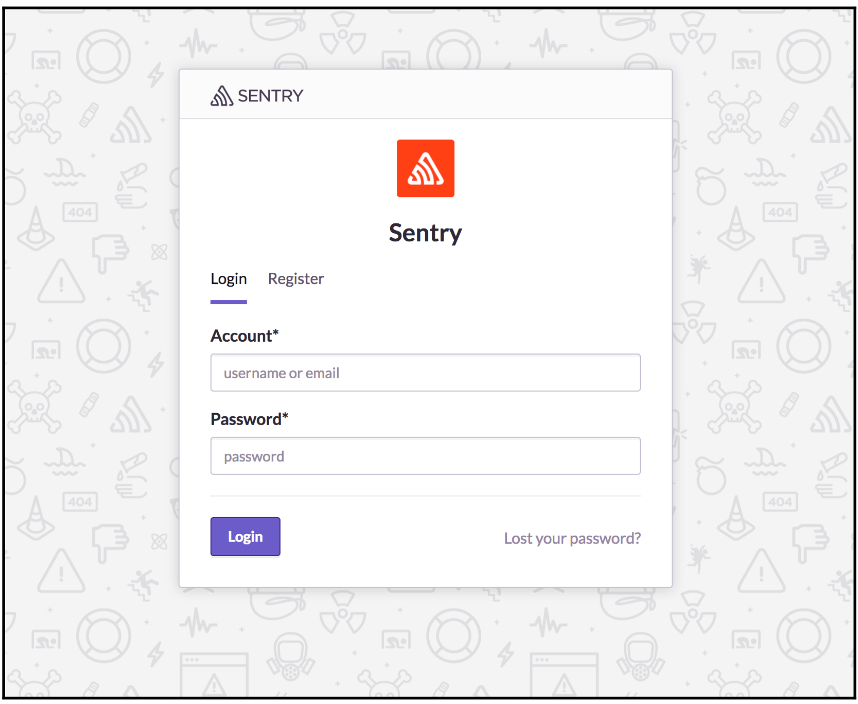
The preceding command will spin up the minimum containers we need for the Sentry setup. Once we have them up for the first time, we need to configure and fill the database and users. We can do it by running just one command on the container we have available for Sentry:

```
docker exec -it docker_sentry_1 sentry upgrade
```

The preceding command will do all the setup needed for Sentry to run and will ask you to create an account to have access to the UI as admin; do this to save it and use it later. As soon as it finishes and returns you to the command path, you can spin up the remaining containers of our project:

```
docker-compose up -d
```

As soon as everything is up, you can open `http://localhost:9876` in your browser and you will see a screen similar to the following one:

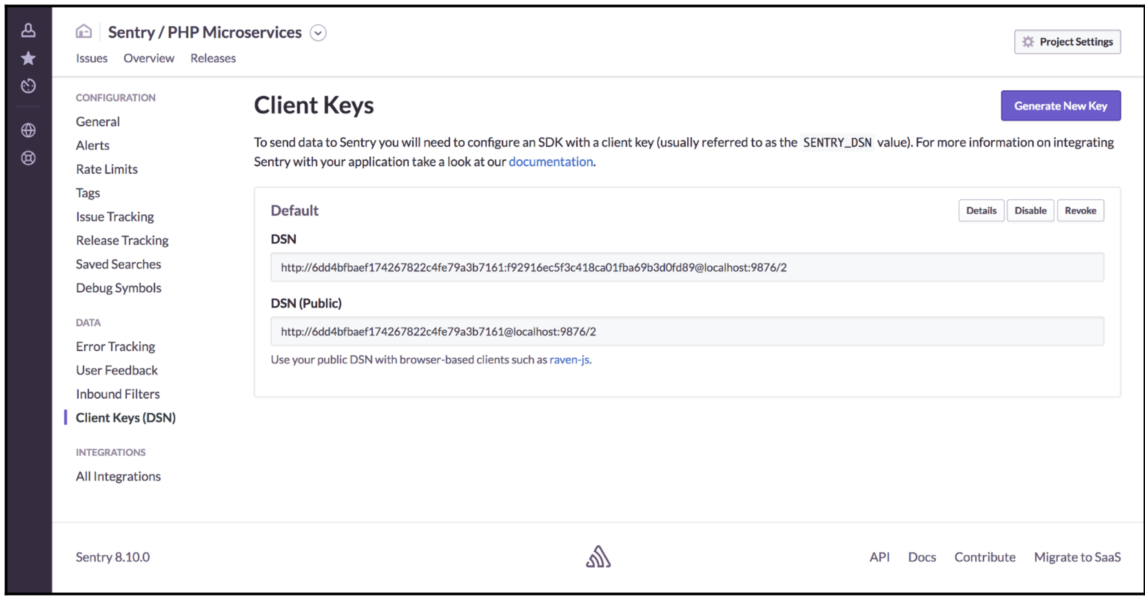


Sentry login page

Log in with the user you created in the previous step and create a new project to start tracking our errors/logs.

Instead of using a single Sentry project to store all your debug information, it is better if you split them into logical groups, for example, one for the user microservice API, and so on.

Once you have created your project, you will need the DSN assigned to this project; open your project settings and select the **Client Keys** option. In this section, you can find the **DSN** keys assigned to the project; you will be using these keys in your code so that the library knows where it needs to send all the debug information:



Sentry DSN keys

Congratulations! At this point, you have Sentry ready to be used in your project. Now it is time to install the `sentry/sentry-laravel` package using composer. To install this library, you can edit your `composer.json` file or enter your user microservice container with the following command:

```
docker exec -it docker_microservice_user_fpm_1 /bin/bash
```

Once you are inside the container, give the following command to use composer to update your `composer.json` and install it for you:

```
composer require sentry/sentry-laravel
```

Once it is installed, we need to configure it on our microservice, so open the `bootstrap/app.php` file and add the following lines:

```
$app->register('SentrySentryLaravelSentryLumenServiceProvider');
# Sentry must be registered before routes are included
require __DIR__ . '/../app/Http/routes.php';
```

Now, we have to configure the report method as we saw earlier, so go to the `app/Exceptions/Handler.php` file and add the following lines in the report function:

```
public function report(Exception $e)
{
   if ($this->shouldReport($e)) {
     app('sentry')->captureException($e);
   }
   parent::report($e);
}
```

These lines will report the exception to Sentry, so let's create the `config/sentry.php` file with the following configuration:

```
<?php
  return array(
    'dsn' => '___DSN___',
    'breadcrumbs.sql_bindings' => true,
  );
```

Application logs

A log is a record of debug information that can be important in the future to see the performance of your application or to see how your application is doing or even to get some stats. Practically, all known applications produce some kind of log information. For example, by default, all the requests to NGINX are recorded in the `/var/log/nginx/error.log` and `/var/log/nginx/access.log`. The first one, `error.log`, stores any errors generated by your application, for example, PHP exceptions. The second one, `access.log`, is created by each request that hits your NGINX server.

As an experienced developer, you already know that keeping some logs in your application is very important and you are not alone in this task, you can find a lot of libraries that can make your life easier. You may be wondering where the important places are and where you can place log calls and the information you need to save. There is no rule of thumb you can always follow, you only need to think about the future, and about what information you will need in the worst case scenario (a broken app).

Let's focus on our example application; in the user service, we will be dealing with user registrations. An interesting point where you can place a log call is just before you save a new user registration. By doing this, you can keep track of your logs and know which information we are trying to save and when. Now, imagine that there is a bug in the registration process and it breaks with special characters but you were not aware of this, the only thing you know is that there are some users reporting issues with the registration. What will you do now? Check the logs! You can easily check which information the users are trying to store and detect that users with special characters are not being registered.

For example, if you are not using a log system, you can use `error_log()` to store messages in the default log file:

```
error_log('Your log message!', 0);
```

The `0` parameter indicates that we want to store our message in the default log file. This function allows us to send the message by e-mail, changing the `0` parameter with `1` and adding an extra parameter with the e-mail address.

All the log systems allow you to define different levels; the most common ones are (note that they can be named differently in different log systems, but the concept is the same):

- **INFO**: This refers to non-critical information. You usually store debug information with this level, for example, you can store a new record each time a specific page is rendered.
- **WARNING**: These are errors that are not very important or where the system can recover itself. For example, the lack of some information that can generate an inconsistent state of your application.
- **ERROR**: This is critical information and, of course, all of these are errors that take place in your application. This is the first level you will check every time you find an error.

Challenges in microservices

When you are working with a monolithic application, your logs will be stored in the same location by default or at least in only a few servers. If you have any problems and you need to check your logs, you can get all the information in a few minutes. The challenge is when you are dealing with a microservice architecture where each microservice generates log information. It is even worse if you have multiple instances of a microservice, each instance creating its own log data.

What will you do in this case? The answer is to store all the log records in the same place using log systems like Sentry. Having a log service allows you to scale your infrastructure without worrying about your logs. They will all be stored in the same place, allowing you to easily find information about different microservices/instances.

Logs in Lumen

Lumen comes out of the box with **Monolog** (PSR-3 interface) integrated; this log library allows you to use different log handlers.

In the Lumen framework, you can set up the amount of error detail of your application in the `.env` file. The `APP_DEBUG` setting defines how much debug information will be generated. The main recommendation is to set this flag to `true` in development environments but always to `false` in production.

To use logging facilities in your code, you only need to ensure that you have uncommented the `$app->withFacades();` line of your `bootstrap/app.php` file. Once you have the facades enabled, you can start using the Log class in any place of your code.

By default, without any extra configuration, Lumen will store the logs in the `storage/logs` folder.

Our logger provides the eight logging levels defined in RFC 5424:

- `Log::emergency($error);`
- `Log::alert($error);`
- `Log::critical($error);`
- `Log::error($error);`
- `Log::warning($error);`
- `Log::notice($error);`
- `Log::info($error);`
- `Log::debug($error);`

An interesting feature is the option where you have to add an array of contextual data. Imagine that you want to log a record of a failed user login. You can do something similar to the following code:

```
Log::info('User unable to login.', ['id' => $user->id]);
```

In the preceding piece of code, we are adding extra information to our log message–the ID of the user who had problems trying to log into our application.

The setup of Monolog with a custom handler like Sentry (we explained how to install it in your project earlier) is very easy, you only need to add the following piece to the `bootstrap/app.php` file:

```
$app->configureMonologUsing(function($monolog) {
  $client = new Raven_Client('sentry-dsn');
  $monolog->pushHandler(
    new MonologHandlerRavenHandler($client,
                                MonologLogger::WARNING)
  );
  return $monolog;
});
```

The preceding code changes how Monolog will work; in our case, instead of storing all our debug information in the `storage/logs` folder, it will use our Sentry installation and the `WARNING` level.

We showed you two different ways of storing your logs in Lumen: in local files like a monolithic application or with an external service. Both of them are fine but our recommendation for microservices development is to use an external tool like Sentry.

Application monitoring

In software development, application monitoring can be defined as the process which ensures that our application performs in an expected manner. This process allows us to measure and evaluate the performance of our application and can be helpful to find bottlenecks or hidden issues.

Application monitoring is usually made through a specialized software that gathers metrics from the application or the infrastructure that runs your software. These metrics can include CPU load, transaction times, or average response times among others. Anything you can measure can be stored in your telemetry system so you can analyze it later.

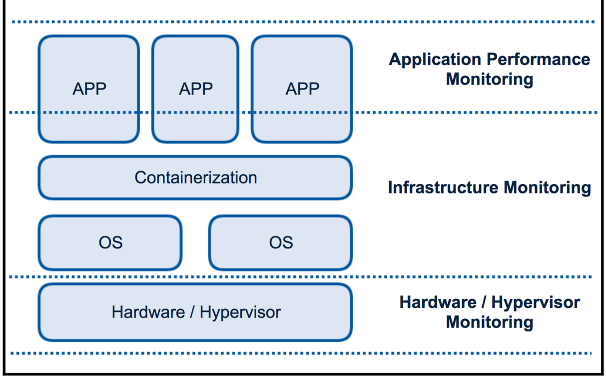
Monitoring a monolithic application is easy; you have everything in one place, all logs are stored in the same place, all metrics can be gathered from the same host, you can know if your PHP thread is killing your server. The main difficulty you may find is finding the part of your application that is underperforming, for example, which part of your PHP code is wasting your resources.

When you work with microservices, you have your code split into logical parts, allowing you to know which part of the application is underperforming, but at a big cost. You have all your metrics segregated between different containers or servers, making it difficult to have a big picture of the overall performance. By having a telemetry system in place, you can send all your metrics to the same location, making it easier to check and debug your application.

Monitoring by levels

As a developer, you need to know how your application is performing at all the levels, from the top level that is your application to the bottom that is the hardware or hypervisor level. In an ideal world, we will have control over all the levels, but the most probable scenario is that you will only be able to monitor up to the infrastructure level.

The following image shows you the different layers and the relation with the server stack:



Monitoring layers

Application level

Application level lives inside your application; all the metrics are generated by your code, in our case by PHP. Unfortunately, you can't find free or open source tools for **Application Performance Monitoring** (**APM**) exclusively for PHP. In any case, you can find interesting third-party services with free plans to give it a try.

Two of the most well-known APM services for PHP are New Relic and Datadog. In both the cases, the installation follows the same path–you install an agent (or library) on your container/host and this small piece of software will start sending the metrics to its service, giving you a dashboard where you can manipulate your data. The main disadvantage of using third-party services is that you don't have any control over this agent or metric system, but this disadvantage can be transformed into a plus point–you will have a reliable system that you don't need to manage, you only need to worry about your application.

Datadog

The installation of the Datadog client could not be easier. Open the `composer.json` of one of your microservices and drop the following line inside the `required` definitions:

```
"datadog/php-datadogstatsd": "0.3.*"
```

As soon as you save your changes and make a composer update, you will be able to use the `Datadogstatsd` class in your code and start sending metrics.

Imagine that you want to monitor the time your secret microservice spends getting all the servers you have in your database. Open the `app/Http/Controllers/SecretController.php` file of your secret microservice and modify your class, as follows:

```
use Datadogstatsd;

/** … Code omitted ... **/
const APM_API_KEY = 'api-key-from-your-account';
const APM_APP_KEY = 'app-key-from-your-account';

public function index(Manager $fractal, SecretTransformer
$secretTransformer, Request $request)
{
  Datadogstatsd::configure(self::APM_API_KEY, self::APM_APP_KEY);
  $startTime = microtime(true);
  $records = Secret::all();

  $collection = new Collection($records, $secretTransformer);
  $data = $fractal->createData($collection)->toArray();
  Datadogstatsd::timing('secrets.loading.time', microtime(true) -
  $startTime, ['service' => 'secret']);

  return response()->json($data);
}
```

The preceding piece of code defines your app and API keys of your Datadog account and we used them to set up our `Datadogstatsd` interface. The example logs the time spent retrieving all our secret records. The `Datadogstatsd::timing()` method will send the metric to our external telemetry service. Doing the monitoring inside your application allows you to decide the places of your code you want to generate metrics in. There is no rule of thumb when you are monitoring this level, but you need to remember that it is important to know where your application spends most of its time, so add metrics in each place of your code that you think could be a bottleneck (like getting data from another service or from a database).

With this library, you can even increment and decrement custom metric points with the following method:

```
Datadogstatsd::increment('another.data.point');
Datadogstatsd::increment('my.data.point.with.custom.increment', .5);
Datadogstatsd::increment('your.data.point', 1, ['mytag'' => 'value']);
```

The three of them increase a point: the first increments `another.data.point` in one unit, the second one increments our point by `0.5`, and the third one increments the point and also adds a custom tag to the metric record.

You can also decrement points with `Datadogstatsd::decrement()`, which has the same syntax as `::increment()`.

Infrastructure level

This level controls everything between the OS and your application. Adding a monitoring system to this layer allows you to know if your container is using too much memory, or if the load of a specific container is too high. You can even track some basic metrics of your application.

There are multiple options to monitor this layer in the high street, but we will give you a sneak peek at two interesting projects. Both of them are open source and even though they use different approaches, you can combine them.

Prometheus

Prometheus is an open source monitoring and alerting toolkit that was created at SoundCloud and is under the umbrella of the **Cloud Native Computing Foundation**. Being the new kid on the block doesn't mean that it doesn't have powerful features. Among others, we can highlight the following main features:

- Time series collection through pull over HTTP
- Target discovery via service discovery (kubernetes, consul, and so on) or static config
- Web UI with simple graphing support
- Powerful query language that allows you to extract all the information you need from your data

Installing Prometheus is very easy with Docker, we only need to add a new container for our telemetry system and link it with our autodiscovery service (Consul). Add the following lines to the `docker-compose.yml` file:

```
telemetry:
  build: ./telemetry/
  links:
    - autodiscovery
  expose:
    - 9090
  ports:
    - 9090:9090
```

In the preceding code, we only tell Docker where the `Dockerfile` is located, link the container without autodiscovery container, and expose and map some ports. Now, it's time to create the `telemetry/Dockerfile` file with the following content:

```
FROM prom/prometheus:latest
ADD ./etc/prometheus.yml /etc/prometheus/
```

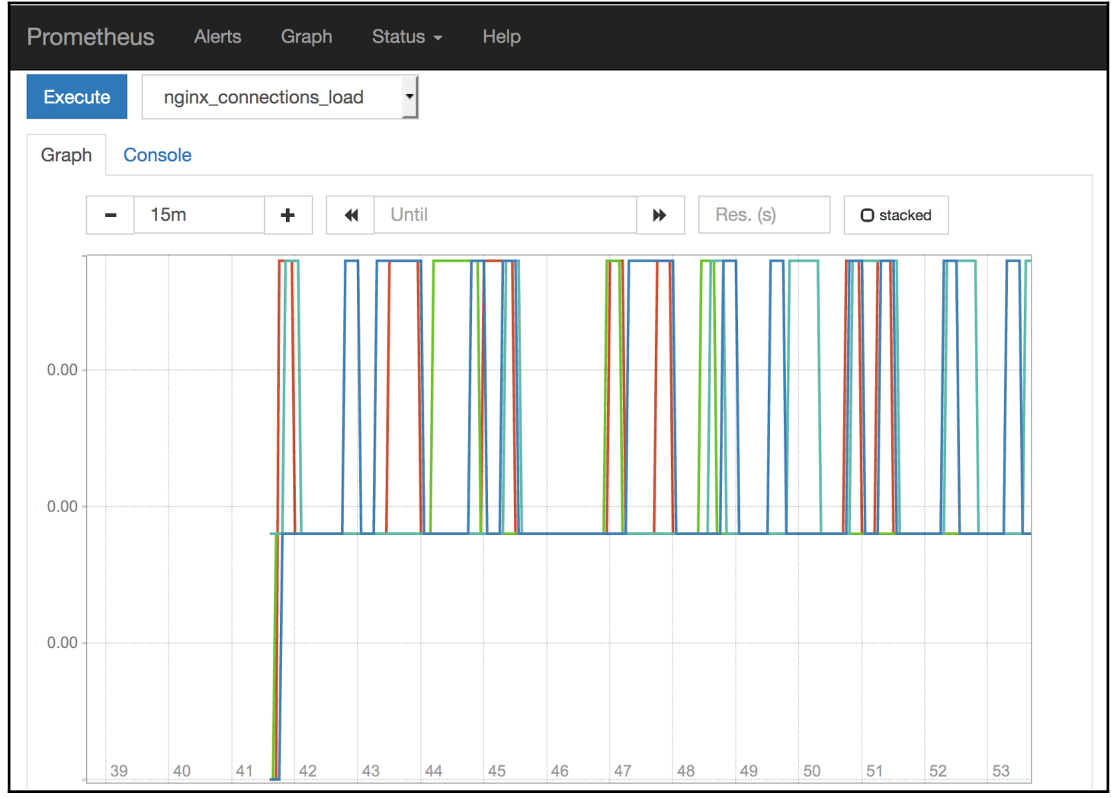
As you can see, it does not take a lot to create our telemetry container; we are using the official image and adding our Prometheus configuration. Create the `etc/prometheus.yml` configuration with the following content:

```
global:
  scrape_interval: 15s
  evaluation_interval: 15s
  external_labels:
  monitor: 'codelab-monitor'

scrape_configs:
```

```
    - job_name: 'containerpilot-telemetry'

  consul_sd_configs:
    - server: 'autodiscovery:8500'
    services: ['containerpilot']
```

Again, the setup is very easy as we are defining some global scrapping intervals and one job called `containerpilot-telemetry` that will use our autodiscovery container and monitor all the services stored in consul announced under the `containerpilot` name.

Prometheus has a simple and powerful web UI. Open your `localhost:9090` and you have access to all the metrics gathered by this tool. Creating a graph is very easy, choose a metric and Prometheus will do all the work for you:



Prometheus graph UI

At this point, you will probably be wondering how you can declare metrics. In the earlier chapters, we introduced `containerpilot`, a tool we will use as a PID in our containers to manage our autodiscovery. The `containerpilot` has the ability to declare metrics to be available for the supported telemetry systems, in our case Prometheus. If you open, for example,
the `docker/microservices/battle/nginx/config/containerpilot.json` file, you can find something similar to the following code:

```
"telemetry": {
  "port": 9090,
  "sensors": [
    {
      "name": "nginx_connections_unhandled_total",
      "help": "Number of accepted connnections that were not
               handled",
      "type": "gauge",
      "poll": 5,
      "check": ["/usr/local/bin/sensor.sh", "unhandled"]
    },
    {
      "name": "nginx_connections_load",
      "help": "Ratio of active connections (less waiting) to the
               maximum worker connections",
      "type": "gauge",
      "poll": 5,
      "check": ["/usr/local/bin/sensor.sh", "connections_load"]
    }
  ]
}
```

In the preceding piece of code, we are declaring two metrics:
`"nginx_connections_unhandled_total"` and `"nginx_connections_load"`.
`ContainerPilot` will run the command defined in the `"check"` parameter inside the container and the result will be scrapped by Prometheus.

You can monitor anything in your infrastructure with Prometheus, even Prometheus itself. Feel free to change our basic installation and setup and adapt it to use the autopilot pattern. If the Prometheus' web UI is not enough for your graphics and you need more power and control, you can easily link our telemetry system with Grafana, one of the most powerful tools out there to create dashboards with all kinds of metrics.

Weave Scope

Weave Scope is a tool used for monitoring your containers, it works well with Docker and Kubernetes and has some interesting features that will make your life easier. Scope gives you a deep top-down view of your app and your entire infrastructure. With this tool, you can diagnose any problems in your distributed containerized application, and everything in real time.

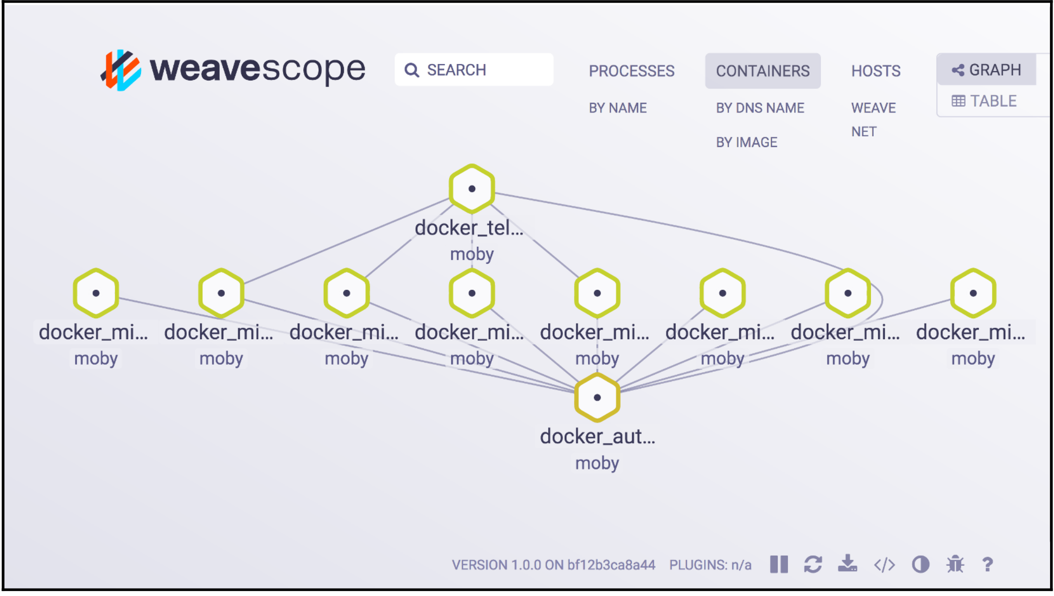
Forget about complex configurations, Scope automatically detects and starts monitoring every host, Docker container, and any process running in your infrastructure. As soon as it gets all this information, it creates a nice map showing the inter-communications between all your containers in real time. You can use this tool to find memory issues, bottlenecks, or any other problems. You can even check different metrics of a process, container, service, or host. A hidden feature you can find in Scope is the ability to manage containers, view logs, or attach a terminal, all from the browser UI.

There are two options to deploy Weave Scope: in a standalone mode where all the components run locally or as a paid cloud service where you don't need to worry about anything. The standalone mode runs as a privileged container inside each one of your infrastructure servers and has the ability to correlate all the information from your cluster or servers and display it in the UI.

The installation is very easy–you only need to run the following commands on each one of your infrastructure servers:

```
sudo curl -L git.io/scope -o /usr/local/bin/scope
sudo chmod a+x /usr/local/bin/scope
scope launch
```

As soon as you have Scope launched, open the IP address of your server (localhost if you are working locally like us) `http://localhost:4040`, and you will see something similar to the following screenshot:



Weave Scope containers graph visualization

The preceding image is a snapshot of the application we are building; here, you can see all our containers and the connections between them in a specific moment in time. Give it a try and while you make some calls to our different API endpoints, you will be able to see the connections between containers changing.

Can you find any problems in our microservices infrastructure? If so, you are correct. As you can see, we didn't connect some of our containers to the autodiscovery service. Scope helped us find a possible future problem, now feel free to fix it.

As you can see, you can use Scope to monitor your app from a browser. You only need to be careful with who has access to the privileged Scope container; if you have plans to use Scope in production, ensure that you limit the access to this tool.

Hardware/hypervisor monitoring

This layer matches our hardware or hypervisor level and is the lowest place where you can place your metrics. The maintenance and monitoring of this layer is usually done by sysadmins and they can use very well-known tools, such as **Zabbix** or **Nagios**. As a developer, you will probably not be worried about this layer. If you deploy the application in a cloud environment, you will not have any kind of access to the metrics generated by this layer.

Summary

In this chapter, we explained how you can debug and do a profiling of a microservice application, an important process in any software development. On a day-to-day basis, you will not spend all your time debugging or profiling your application; in some scenarios, you will spend a lot of time trying to fix bugs. For this reason, it is important to have a place where you can store all your errors and debug information, this information will give you a deeper understanding of what is happening with your app. Finally, as a full-stack developer, we showed you how to monitor the top two layers of your application stack.

7
Security

When we are developing an application, we should always be thinking about how we can make our microservices more secure. There are some techniques and methods that every developer should know in order to avoid security problems. In this chapter, you will discover the ways to use authentication and authorization to use in your microservices, and how to manage the permissions of each functionality once your users log in. You will also discover the different methods you can use to encrypt your data.

Encryption in microservices

We can define encryption as the process of transforming information in such a way that only authorized parties are able to read it. This process can be done practically at any level of your application. For example, you can encrypt your whole database, or you can add the encryption in the transport layer with SSL/TSL or with **JSON Web Token** (**JWT**).

These days, the encryption/decryption process is done through modern algorithms and the highest level where the encryption is added is on the transport layer. All the algorithms used in this layer provide at least the following features:

- **Authentication**: This feature allows you to verify the origin of a message
- **Integrity**: This feature gives you proof that the content of the message wasn't changed on its way from the origin

The final mission of the encryption algorithms is to provide you with a security layer so that you can interchange or store sensitive data without having to worry about someone stealing your information, but it is not free of cost. Your environment will use some resources dealing with encryption, decryption, or handshakes among other related things.

As a developer, you need to think that you will be deployed to a hostile environment–production is a war zone. If you start thinking this way, you will probably start asking yourself the following questions:

- Will we deploy to hardware or to a virtualized environment? Will we share resources?
- Can we trust all the possible neighbors of our application?
- Will we split our application into different and separated zones? How will we connect our zones?
- Will our application be PCI compliant or will it need a very high degree of security due to the data we store/manage?

When you start answering all these questions (among others), you will start figuring out the level of security needed for your application.

In this section, we will show you the most common ways to encrypt the data in your application so that you can later choose which one to implement.

Note that we are not considering full disk encryption because it is considered to be the weakest method to protect your data.

Database encryption

When you are dealing with sensitive data, the most flexible and with lower overhead method of protecting your data is using encryption in your application layer. However, what happens if, for some reason, you cannot change your application? The next most powerful solution is to encrypt your database.

For our application, we have chosen a *relational database*; specifically, we are using Percona, a MySQL fork. Currently, you have two different options to encrypt your data in this database:

- Enable the encryption through MariaDB patch (another MySQL form that is pretty similar to Percona). This patch is available in 10.1.3 and the later versions.
- The InnoDB tablespace level encryption method is available from Percona Server 5.7.11 or MySQL 5.7.11.

Perhaps you are wondering why we are talking about MariaDB and MySQL when we have chosen Percona. This is because the three of them have the same core, sharing most of their core functionalities.

All the major database softwares allow you to encrypt your data. If you are not using Percona, check the official documentation of your database to find the required steps needed to allow encryption.

As a developer, you need to know the weakness of using a database level encryption in your application. Among others, we can highlight the following ones:

- Privileged database users have access to the key ring file, so be strict with user permissions in your database.
- Data is not encrypted while is stored in the RAM of your server, it is only encrypted when the data is written in the hard drive. A privileged and malicious user can use some tools to read the server memory and as a consequence, your application data too.
- Some tools like GDB can be used to change the root user password structure, allowing you to copy data without any issues.

Encryption in MariaDB

Imagine that instead of using Percona, you want to use MariaDB; database encryption is available thanks to the `file_key_management` plugin. In our application example, we are using Percona as data storage for the secrets microservice, so let's add a new container for MariaDB only so that you can later give it a try and interchange the two RDBMS.

First, create a `mariadb` folder in your Docker repository inside the secrets microservice on the same level as the database folder. Here, you can add a `Dockerfile` with the following content:

```
FROM mariadb:latest

RUN apt-get update \
&& apt-get autoremove && apt-get autoclean \
&& rm -rf /var/lib/apt/lists/*

RUN mkdir -p /volumes/keys/
RUN echo
"1;
C472621BA1708682BEDC9816D677A4FDC51456B78659F183555A9A895EAC9218" >
/volumes/keys/keys.txt
RUN openssl enc -aes-256-cbc -md sha1 -k secret -in
/volumes/keys/keys.txt -out /volumes/keys/keys.enc
COPY etc/ /etc/mysql/conf.d/
```

In the preceding code, we are pulling the latest official MariaDB image, updating it, and creating some certificates that we will need for our encryption. The long string saved in the `keys.txt` file is a key we generated ourselves with the following command:

```
openssl enc -aes-256-ctr -k secret@phpmicroservices.com -P -md sha1
```

The last command of our `Dockerfile` will copy our bespoke database configurations inside the container. Create our custom database configuration in `etc/encryption.cnf` with the following content:

```
[mysqld]
plugin-load-add=file_key_management.so
file_key_management_filekey = FILE:/mount/keys/server-key.pem
file-key-management-filename = /mount/keys/mysql.enc
innodb-encrypt-tables = ON
innodb-encrypt-log = 1
innodb-encryption-threads=1
encrypt-tmp-disk-tables=1
encrypt-tmp-files=0
encrypt-binlog=1
file_key_management_encryption_algorithm = AES_CTR
```

In the preceding code, we are telling our database engine where we store our certs and we enable the encryption. Now, you can edit our `docker-compose.yml` file and add the following container definition:

```
microservice_secret_database_mariadb:
  build: ./microservices/secret/mariadb/
  environment:
    - MYSQL_ROOT_PASSWORD=mysecret
    - MYSQL_DATABASE=finding_secrets
    - MYSQL_USER=secret
    - MYSQL_PASSWORD=mysecret
  ports:
    - 7777:3306
```

As you can see from the preceding code, we are not defining anything new; you now probably have enough experience with Docker to understand that we are defining where our `Dockerfile` is located. We set up some environment variables and mapped the `7777` local port to the container `3306` port. As soon as you have made all your changes, a simple `docker-compose build microservice_secret_database` command will generate the new container.

After building the container, it's time to check whether everything is working. Spin up the new container with `docker-compose up microservice_secret_database` and try to connect it to our local `7777` port. Now, you can start using encryption in this container. Consider the following example:

```
CREATE TABLE `test_encryption` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `text_field` varchar(255) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB `ENCRYPTED`=YES `ENCRYPTION_KEY_ID`=1;
```

In the preceding code, we added some extra tags to our SQL; they enable the encryption in the table and use the encryption key with the ID `1` we stored in `keys.txt` (the file we used to start our database). Give it a try and, if everything runs smoothly, feel free to make the necessary changes to use this new database container instead of the other one we have in our project.

InnoDB encryption

Percona and MySQL 5.7.11+ versions come with a new feature out of the box–support for **InnoDB** tablespace level encryption. With this feature, you can encrypt all your InnoDB tables without too much fuss or configuration. In our example application, we are using Percona 5.7 on the secrets microservice. Let's look at how to encrypt our tables.

First, we need to make some small amendments to our Docker environment; first of all, open `microservices/secret/database/Dockerfile` and replace all the content with the following the lines of code:

```
FROM percona:5.7
RUN mkdir -p /mount/mysql-keyring/ \
  && touch /mount/mysql-keyring/keyring \
  && chown -R mysql:mysql /mount/mysql-keyring
  COPY etc/ /etc/mysql/conf.d/
```

At this point in the book, you probably don't need an explanation of what we did in our `Dockerfile`, so let's create a new `config` file that we will later copy to our container. Inside the `secret microservice` folder, create an `etc` folder and generate a new `encryption.cnf` file with the following content:

```
[mysqld]
early-plugin-load=keyring_file.so
keyring_file_data=/mount/mysql-keyring/keyring
```

In the configuration file we created earlier, we are loading the `keyring` lib, where our database can find and store the generated keyrings used to encrypt our data.

At this point, you have everything you need to enable the encryption, so rebuild the container with `docker-compose build microservice_secret_database` and spin all your containers up again with `docker-compose up -d`.

If everything is fine, you should be able to open your database without any problems and you can alter the tables we stored with the following SQL command:

```
ALTER TABLE `secrets` ENCRYPTION='Y'
```

You may be wondering why we altered our `secrets` table if we already enabled the encryption in the database. The reason behind this is because the encryption doesn't come enabled by default, so you need to explicitly tell the engine which tables you want to encrypt.

Performance overhead

Using encryption in your database will reduce the performance of your application. Your machines/containers will use some resources dealing with the encrypt/decrypt process. In some tests, this overhead can be over 20% when you are not using the tablespace level encryption (MySQL/Percona +5.7). Our recommendation is to measure the average performance of your application with and without the encryption enabled. This way, you can ensure that the encryption will not have a high impact on your application.

In this section, we showed you two quick ways of adding an extra layer of security to your application. The final decision for using these features depends on you and the specifications of your application.

TSL/SSL protocols

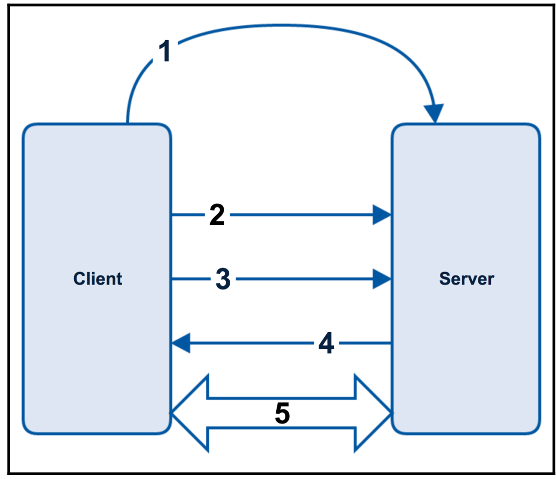
Transport Layer Security (**TSL**) and **Secure Sockets Layer** (**SSL**) are cryptographic protocols used to secure communication in an untrusted network, for example, the Internet or LAN of your ISP. SSL is the predecessor of TSL and both of them are often used interchangeably or in conjunction with TLS/SSL. These days, SSL and TSL are practically the same thing and it makes no difference if you choose to use one or the other, you will be using the same level of encryption, dictated by the server. If an application, for example, an e-mail client, gives you the option to choose between SSL or TSL, you are only selecting how the secure connection is initiated, nothing else.

All the power and security of these protocols rely on what we know as certificates. TSL/SSL certificates can be defined as small data files that digitally bind a cryptographic key to an organization or a person's details. You can find all kinds of companies that sell TSL/SSL certificates, but if you don't want to spend money (or you are in the development phase), you can create self-signed certificates. These kinds of certificates can be used to encrypt data, but the clients will not trust them unless you skip the validation.

How the TSL/SSL protocol works

Before you start using TSL/SSL in your application, you need to know how it works. There are many other books dedicated to explaining how these protocols work, so we will only give you a sneak peek.

The following diagram is a summary of how the TSL/SSL protocol works; first, you need to know that TSL/SSL is a TCP client-server protocol and the encryption starts after a few steps:



TSL/SSL protocol

The following are the steps of the TSL/SSL protocol:

1. Our client wants to start a connection to a server/service secured with TSL/SSL, so it asks the server to identify itself.
2. The server attends to the petition and sends the client a copy of its TSL/SSL certificate.

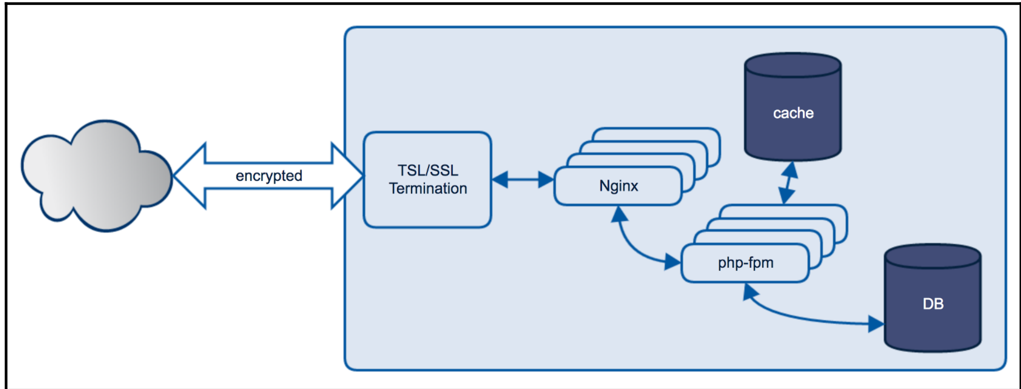
3. The client checks if the TSL/SSL certificate is a trusted one and if so, sends a message to the server.
4. The server returns a digitally signed acknowledgement to start a session.
5. After all the previous steps (handshake), the encrypted data is shared between the client and the server.

As you can imagine, the terms *client* and *server* are ambiguous; a client can be a browser trying to reach your page or the client can be a microservice trying to communicate with another microservice.

TSL/SSL termination

As you learned before, adding a TSL/SSL layer to your application adds a little overhead to the overall performance of your app. To mitigate this problem, we have what we call TSL/SSL termination, a form of TSL/SSL offloading, which moves the responsibility of encryption/decryption from the server to a different part of your application.

TSL/SSL termination relies on the fact that once all the data is decrypted, you trust all the communication channels you are using to move this decrypted data. Let's see an example with a microservice; take a look at the following image:



TSL/SSL termination in a microservice

In the preceding image, all the in/out communications are encrypted using a specific component of our microservice architecture. This component will be acting as a proxy and it will be dealing with all the TSL/SSL stuff. As soon as a request from a client comes, it manages all the handshake and decrypts the request. Once the request is decrypted, it is proxied to the specific microservice component (in our case, it is NGINX) and our microservice does what is needed, for example, getting some data from the database. Once the microservice needs to return a response, it uses the proxy where all our response is encrypted. If you have multiple microservices, you can scale out this small example and do the same–encrypt all the communications between the different microservices and use encrypted data inside the microservice.

TSL/SSL with NGINX

You can find multiple softwares that you can use to do TSL/SSL termination. Among others, the following list flags the most well known:

- **Load balancer**: Amazon ELB and HaProxy
- **Proxies**: NGINX, Traefik, and Fabio

In our case, we will use NGINX to manage all the TSL/SSL termination, but feel free to try other options.

As you probably know already, NGINX is one of the most versatile softwares in the market. You can use it as a reverse proxy or a web server with a high performance level and stability.

We will explain how to do a TSL/SSL termination in NGINX, for example, for the battle microservice. First, open the `microservices/battle/nginx/Dockerfile` file and add the following command just before the CMD command:

```
RUN echo 01 > ca.srl \
&& openssl genrsa -out ca-key.pem 2048 \
&& openssl req -new -x509 -days 365 -subj "/CN=*" -key ca-key.pem -out
ca.pem \
&& openssl genrsa -out server-key.pem 2048 \
&& openssl req -subj "/CN=*" -new -key server-key.pem -out server.csr \
&& openssl x509 -req -days 365 -in server.csr -CA ca.pem -CAkey ca-key.pem
-out server-cert.pem \
&& openssl rsa -in server-key.pem -out server-key.pem \
&& cp *.pem /etc/nginx/ \
&& cp *.csr /etc/nginx/
```

Here, we created some self-signed certificates and stored them inside the `/etc/nginx` folder of the `nginx` container.

Once we have our certificates, it's time to change the NGINX configuration file. Open the `microservices/battle/nginx/config/nginx/nginx.conf.ctmpl` file and add the following server definition:

```
server {
  listen 443 ssl;
  server_name _;
  root /var/www/html/public;
  index index.php index.html;
  ssl on;
  ssl_certificate /etc/nginx/server-cert.pem;
  ssl_certificate_key /etc/nginx/server-key.pem;
  location = /favicon.ico { access_log off; log_not_found off; }
  location = /robots.txt { access_log off; log_not_found off; }
  access_log /var/log/nginx/access.log;
  error_log /var/log/nginx/error.log error;
  sendfile off;
  client_max_body_size 100m;
  location / {
    try_files $uri $uri/ /index.php?_url=$uri&$args;
  }
  location ~ /\.ht {
    deny all;
  }
  {{ if service $backend }}
  location ~ \.php$ {
    try_files $uri =404;
    fastcgi_split_path_info ^(.+\.php)(/.+)$;
    fastcgi_pass {{ $backend }};
    fastcgi_index /index.php;
    include fastcgi_params;
    fastcgi_param SCRIPT_FILENAME
    $document_root$fastcgi_script_name;
    fastcgi_intercept_errors off;
    fastcgi_buffer_size 16k;
    fastcgi_buffers 4 16k;
  }
  {{ end }}
}
```

The preceding piece of code sets up a new listener in the `nginx` server, in the `443` port. As you can see, it is very similar to the default server settings; the difference lies in the ports and the location of the certificates we created in the previous step.

To use this new TSL/SSL endpoint, we need to make some small changes to the `docker-compose.yml` file and map the `443` NGINX port. To do this, you only need to go to the `microservice_battle_nginx` definition and add a new line in the ports declaration, as follows:

```
    - 8443:443
```

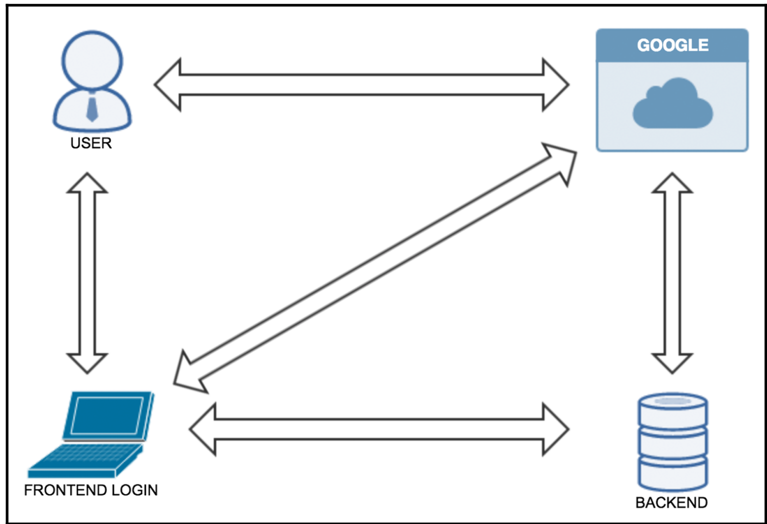
The new line will map our `8443` port to the `nginx` container `443` port, allowing us to connect through TSL/SSL. You can give it a try now with Postman but, due to the fact that it is a self-signed certificate, by default it is not accepted. Open **Preferences** and disable **SSL certificate verification**. As homework, you can change all our example services to only use the TSL/SSL layer to communicate with each other.

In this section of the chapter, we have shown you the main ways in which you can add an extra layer of security to your application, encrypting your data and the communication channel used to interchange messages. Now that we are sure that our application has at least some level of encryption, let's continue with another important aspect of any application–authentication.

Authentication

The starting point of every project is the authentication system, in which it is possible to identify the users or customers who will use our application or API. There are many libraries to implement the different ways to authenticate users; in this book, we will see two of the most important ways: **OAuth 2** and **JWT**.

As we already know, microservices are *stateless*, which means that they should communicate with each other and users using an *access token* instead of cookies and sessions. So, let's look at what the workflow of the authentication is like using it:



Authentication by token workflow

As you can see in the preceding image, this should be the process of getting a list of secrets required by a customer or user:

1. **USER** asks **FRONTEND LOGIN** for a list of secrets.
2. **FRONTEND LOGIN** asks **BACKEND** for the list of secrets.
3. **BACKEND** asks **FRONTEND LOGIN** for the user access token.
4. **FRONTEND LOGIN** asks **GOOGLE** (or any other provider) for the access token.
5. **GOOGLE** asks **USER** for their credentials.
6. **USER** provides credentials to **GOOGLE**.
7. **GOOGLE** provides user access token to **FRONTEND LOGIN**.
8. **FRONTEND LOGIN** provides **BACKEND** the user access token.
9. **BACKEND** checks with **GOOGLE** who the user of that access token is.
10. **GOOGLE** tells **BACKEND** who the user is.
11. **BACKEND** checks the user and tells **FRONTEND LOGIN** the list of secrets.
12. **FRONTEND LOGIN** shows **USER** the list of secrets.

Obviously, in this process, everything happens without the user knowing it. The user only has to provide his/her credentials to the proper service. In the preceding example, the service is **GOOGLE**, but it can even be our own application.

Now, we will build a new docker container in order to create and set up a database to authenticate users using OAuth 2 and JWT.

Create a `Dockerfile` in the docker user microservice under the `docker/microservices/user/database/Dockerfile` database folder with the following line. We will use Percona as we did for the secret microservice:

```
FROM percona:5.7
```

Once you have created the `Dockerfile`, open the `docker-composer.yml` file and add the user database microservice configuration at the end of the User microservice section (just before the source containers). Also, add `microservice_user_database` to the `microservice_user_fpm` links section to make the database visible:

```
microservice_user_fpm:
{{omitted code}}
links:
{{omitted code}}
 - microservice_user_database
microservice_user_database:
 build: ./microservices/user/database/
 environment:
 - CONSUL=autodiscovery
 - MYSQL_ROOT_PASSWORD=mysecret
 - MYSQL_DATABASE=finding_users
 - MYSQL_USER=secret
 - MYSQL_PASSWORD=mysecret
 ports:
 - 6667:3306
```

Once we have set the configuration, it is time to build it, so run the following command on your terminal to create the new container we have just set up:

```
docker-compose build microservice_user_database
```

It can take some time; when it finishes, we have to start the containers again by running the following command:

```
docker-compose up -d
```

You can check whether the user database microservice was created correctly by executing `docker ps`, so check to see the new `microservice_user_database` on it.

It is time to set up the user microservice to be able to use the database container we have just created, so add the following line to the `bootstrap/app.php` file:

```
$app->configure('database');
```

Also, create the `config/database.php` file with the following configuration:

```
<?php
  return [
    'default'     => 'mysql',
    'migrations'  => 'migrations',
    'fetch'       => PDO::FETCH_CLASS,
    'connections' => [
      'mysql' => [
        'driver'    => 'mysql',
        'host'      => env('DB_HOST','microservice_user_database'),
        'database'  => env('DB_DATABASE','finding_users'),
        'username'  => env('DB_USERNAME','secret'),
        'password'  => env('DB_PASSWORD','mysecret'),
        'collation' => 'utf8_unicode_ci',
      ]
    ]
  ];
```

Note that, in the preceding code, we have used the same credentials we used on the `docker-compose.yml` file in order to connect to the database container.

That is everything. We now have a new database container connected to user microservice and it is ready to be used. Add a new users table by creating a migration or executing the following query in your favorite SQL client:

```
CREATE TABLE `users` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `email` varchar(255) NOT NULL,
  `password` varchar(255) NOT NULL,
  `api_token` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=latin1;
```

OAuth 2

Let's introduce a secure and especially useful in microservices authentication system based on access tokens.

OAuth 2 is a standard protocol that allows us to limit some methods of our API REST to specific users, avoiding having to ask the users for their usernames and passwords.

This protocol is very common because it is more secure as it avoids sharing passwords or delicate credentials when communicating between APIs.

OAuth 2 uses an access token that needs to be obtained by the user in order to use the application. The token will have an expiration time and it can be refreshed without having to give the user credentials again.

How to use OAuth 2 on Lumen

Now, we will explain how to install, set up, and try OAuth 2 authentication on Lumen. The goal of this is to have a microservice using OAuth 2 for your methods; in other words, the consumer will need to provide a token before using the methods that require authentication.

OAuth 2 installation

Go to the user microservice by executing the following commands on the docker folder:

```
docker-compose up -d
docker exec -it  docker_microservice_user_fpm_1 /bin/bash
```

Once we are in the User microservice, it is necessary to install OAuth 2 by adding the following line in the `composer.json` file in the `require` section:

```
"lucadegasperi/oauth2-server-laravel": "^5.0"
```

Then, execute `composer update` and the package will install OAuth 2 on your microservice.

Setup

Once the package is installed, we have to set up some important things in order to run OAuth 2. Firstly, we need to copy the OAuth 2 config file located at `/vendor/lucadegasperi/oauth2-server-laravel/config/oauth2.php` to `/config/oauth2.php`; if the `config` folder does not exist, create it. Also, we need to copy the migration files included in the `/vendor/lucadegasperi/oauth2-server-laravel/database/migrations to /database/migrations` folder.

Do not forget to register OAuth 2 by adding the following lines to `/bootstrap/app.php`:

```
$app-
>register(\LucaDegasperi\OAuth2Server\Storage\
FluentStorageServiceProvider::class);
$app-    >register(\LucaDegasperi\OAuth2Server\
OAuth2ServerServiceProvider::class);
$app->middleware([
  \LucaDegasperi\OAuth2Server\Middleware\
  OAuthExceptionHandlerMiddleware::class
]);
```

At the top of the file, before the `app->withFacades();` line (if it is not uncommented, do it), add the following lines:

```
class_alias('Illuminate\Support\Facades\Config', 'Config');
class_alias(\LucaDegasperi\OAuth2Server\Facades\Authorizer::class,
'Authorizer');
```

Now, we will execute the migrations in order to create the necessary tables in the database:

```
composer dumpautoload
php artisan migrate
```

If you have issues in executing the migrations, try adding the `'migrations' => 'migrations', 'fetch' => PDO::FETCH_CLASS,` line to the `config/database.php` file and then, execute `php artisan migrate:install --database=mysql`.

Once we have created all the necessary tables, insert a register in the `oauth_clients` table using Lumen seeders or by executing the following query on your favorite SQL client:

```
INSERT INTO `finding_users`.`oauth_clients`
(`id`, `secret`, `name`, `created_at`, `updated_at`)
VALUES
('1', 'YouAreTheBestDeveloper007', 'PHPMICROSERVICES', NULL, NULL);
```

Now, we have to add a new route on `/app/Http/routes.php` in order to get a valid token for the user we have just created. For example, the route can be `oauth/access_token`:

```
$app->post('oauth/access_token', function() {
  return response()->json(Authorizer::issueAccessToken());
});
```

Finally, modify the `grant_types` value on the `/config/oauth2.php` file, changing it to the following lines of code:

```
'grant_types' => [
  'client_credentials' => [
    'class'            => '\League\OAuth2\Server\Grant\
    ClientCredentialsGrant',
    'access_token_ttl' => 0
  ]
],
```

Let's try OAuth2

We are now ready to get our token by doing a POST call on Postman to `http://localhost:8084/api/v1/oauth/access_token`, including the following parameters on the body:

```
grant_type: client_credentials
client_id: 1
client_secret: YouAreTheBestDeveloper007
```

If we enter the wrong credentials, it will give the following response:

```
{
  "error": "invalid_client",
  "error_description": "Client authentication failed."
}
```

If the credentials are correct, we will get the `access_token` in JSON:

```
{
  "access_token": "anU2e6xgXiLm7UARSSV7M4Wa7u86k4JryKWrIQhu",
  "token_type": "Bearer",
  "expires_in": 3600
}
```

Once we have a valid access token, we can restrict some methods for unregistered users. This is very easy on Lumen. We just have to enable route middlewares on `/bootstrap/app.php`, so add the following code in that file:

```
$app->routeMiddleware(
  [
    'check-authorization-params' =>
    \LucaDegasperi\OAuth2Server\Middleware\
    CheckAuthCodeRequestMiddleware::class,
    'csrf' => \Laravel\Lumen\Http\Middleware\
    VerifyCsrfToken::class,
    'oauth' =>
    \LucaDegasperi\OAuth2Server\Middleware\
    OAuthMiddleware::class,
    'oauth-client' => \LucaDegasperi\OAuth2Server\Middleware\
    OAuthClientOwnerMiddleware::class,
    'oauth-user' => \LucaDegasperi\OAuth2Server\Middleware\
    OAuthUserOwnerMiddleware::class,
  ]
);
```

Go to the controller `UserController.php` file and add a `__construct()` function with the following code:

```
public function __construct(){
  $this->middleware('oauth');
}
```

This will affect all the methods on the controller, but we can exclude some of them with the following code:

```
public function __construct(){
  $this->middleware('oauth', ['except' => 'index']);
}
public function index()
{
  return response()->json(['method' => 'index']);
}
```

Now we can test the index function by doing a GET call to `http://localhost:8084/api/v1/user`. Do not forget to include a header called `Authorization` with the `Bearer anU2e6xgXiLm7UARSSV7M4Wa7u86k4JryKWrIQhu` value.

If we excluded the index function or if we enter the token properly, we will get the JSON response with status code 200:

```
{"method":"index"}
```

If we did not exclude the index method and we enter a wrong token, we will get an error code 401 and the following message:

```
{"error":"access_denied","error_description":"The resource owner or
authorization server denied the request."}
```

Now you have a secure and better application. Remember that you can add the error handling you learned in the last chapter to your authorization methods.

JSON Web Token

JSON Web Token (**JWT**) is group of security methods to be used in HTTP requests and to be transferred between the client and the server. A JWT token is a JSON object that is digitally signed using JSON web signature.

In order to create a token with JWT, we need the user credentials, a secret key, and the encryption type to be used; it can be HS256, HS384, or HS512.

How to use JWT on Lumen

It is possible to install JWT on Lumen using composer. So, once you are in the user microservice container, execute the following command in your terminal:

```
composer require tymon/jwt-auth:"^1.0@dev"
```

Another way to install the library is to open your `composer.json` file and add `"tymon/jwt-auth": "^1.0@dev"` to the list of required libraries. Once it is installed, we need to register JWT on register service providers as we did with OAuth 2. On Lumen, it is possible to do this by adding the following line in the `bootstrap/app.php` file:

```
$app->register('Tymon\JWTAuth\Providers\JWTAuthServiceProvider');
```

Also, uncomment the following line:

```
$app->register(App\Providers\AuthServiceProvider::class);
```

Your `bootstrap/app.php` file should look as follows:

```
<?php
  require_once __DIR__.'/../vendor/autoload.php';
  try {
    (new Dotenv\Dotenv(__DIR__.'/../'))->load();
  } catch (Dotenv\Exception\InvalidPathException $e) {
    //
  }
  $app = new Laravel\Lumen\Application(
    realpath(__DIR__.'/../')
  );
  // $app->withFacades();
  $app->withEloquent();
  $app->singleton(
    Illuminate\Contracts\Debug\ExceptionHandler::class,
    App\Exceptions\Handler::class
  );
  $app->singleton(
    Illuminate\Contracts\Console\Kernel::class,
    App\Console\Kernel::class
  );
  $app->routeMiddleware([
    'auth' => App\Http\Middleware\Authenticate::class,
  ]);
  $app->register(App\Providers\AuthServiceProvider::class);
  $app->register
  (Tymon\JWTAuth\Providers\LumenServiceProvider::class);
  $app->group(['namespace' => 'App\Http\Controllers'],
  function ($app)
  {
       require __DIR__.'/../app/Http/routes.php';
  });
  return $app;
```

Setting up JWT

Now we need a secret key, so run the following command in order to generate and place it on the JWT config file:

```
php artisan jwt:secret
```

Once it is generated, you can see the secret key placed in the `.env` file (your secret key will be different). Check this and ensure that your `.env` looks as shown:

```
APP_DEBUG=true
APP_ENV=local
SESSION_DRIVER=file
DB_HOST=microservice_user_database
DB_DATABASE=finding_users
DB_USERNAME=secret
DB_PASSWORD=mysecret
JWT_SECRET=wPB1mQ6ADZrc0ouxMCYJfiBbMC14IAV0
CACHE_DRIVER=file
```

Now, go to the `config/jwt.php` file; this is the JWT `config` file, ensure that your file is as follows:

```
<?php
  return [
    'secret' => env('JWT_SECRET'),
    'keys' => [
      'public' => env('JWT_PUBLIC_KEY'),
      'private' => env('JWT_PRIVATE_KEY'),
      'passphrase' => env('JWT_PASSPHRASE'),
    ],
    'ttl' => env('JWT_TTL', 60),
    'refresh_ttl' => env('JWT_REFRESH_TTL', 20160),
    'algo' => env('JWT_ALGO', 'HS256'),
    'required_claims' => ['iss', 'iat', 'exp', 'nbf', 'sub',
    'jti'],
    'blacklist_enabled' => env('JWT_BLACKLIST_ENABLED', true),
    'blacklist_grace_period' => env('JWT_BLACKLIST_GRACE_PERIOD',
    0),
    'providers' => [
      'jwt' => Tymon\JWTAuth\Providers\JWT\Namshi::class,
      'auth' => Tymon\JWTAuth\Providers\Auth\Illuminate::class,
      'storage' =>
      Tymon\JWTAuth\Providers\Storage\Illuminate::class,
    ],
  ];
```

It is also necessary to set up `config/app.php` properly. Ensure that you entered the user model correctly, it will define the table where JWT should search for the user and password provided:

```
<?php
  return [
    'defaults' => [
      'guard' => env('AUTH_GUARD', 'api'),
      'passwords' => 'users',
    ],
    'guards' => [
      'api' => [
        'driver' => 'jwt',
        'provider' => 'users',
      ],
    ],
    'providers' => [
      'users' => [
        'driver' => 'eloquent',
        'model' => \App\Model\User::class,
      ],
    ],
    'passwords' => [
      'users' => [
        'provider' => 'users',
        'table' => 'password_resets',
        'expire' => 60,
      ],
    ],
  ];
```

Now we are ready to define the methods that require authentication by editing `/app/Http/routes.php`:

```
<?php
  $app->get('/', function () use ($app) {
    return $app->version();
  });
  use Illuminate\Http\Request;
  use Tymon\JWTAuth\JWTAuth;
  $app->post('login', function(Request $request, JWTAuth $jwt) {
    $this->validate($request, [
      'email' => 'required|email|exists:users',
      'password' => 'required|string'
    ]);
    if (! $token = $jwt->attempt($request->only(['email',
    'password']))) {
      return response()->json(['user_not_found'], 404);
```

```
    }
    return response()->json(compact('token'));
  });
  $app->group(['middleware' => 'auth'], function () use ($app) {
    $app->post('user', function (JWTAuth $jwt) {
      $user = $jwt->parseToken()->toUser();
      return $user;
    });
  });
```

You can see in the preceding code that our middleware only affects the methods included in the group that we defined the middleware in. We can create all the groups we want in order to pass the methods through the middleware(s) that we select.

And finally, edit the `/app/Providers/AuthServiceProvider.php` file and add the following highlighted code:

```
<?php
  namespace App\Providers;
  use App\User;
  use Illuminate\Support\ServiceProvider;
  class AuthServiceProvider extends ServiceProvider
  {
    public function register()
    {
      //
    }
    public function boot()
    {
      $this->app['auth']->viaRequest('api', function ($request) {
        if ($request->input('email')) {
          return User::where('email', $request->input('email'))-
          >first();
        }
      });
    }
  }
```

Finally, we need to make some changes on your user model file, so go to `/app/Model/User.php` and add the following lines of `JWTSubject` to the class implements list:

```
<?php
  namespace App\Model;
  use Illuminate\Contracts\Auth\Access\Authorizable as
  AuthorizableContract;
  use Illuminate\Database\Eloquent\Model;
```

```
use Illuminate\Auth\Authenticatable;
use Laravel\Lumen\Auth\Authorizable;
use Illuminate\Contracts\Auth\Authenticatable as
AuthenticatableContract;
use Tymon\JWTAuth\Contracts\JWTSubject;
class User extends Model implements JWTSubject,
AuthorizableContract,
AuthenticatableContract {
  use Authenticatable, Authorizable;
  protected $table = 'users';
  protected $fillable = ['email', 'api_token'];
  protected $hidden = ['password'];
  public function getJWTIdentifier()
  {
    return $this->getKey();
  }
  public function getJWTCustomClaims()
  {
    return [];
  }
}
```

Do not forget to add the `getJWTIdentifier()` and `getJWTCustomClaims()` functions, as you can see in the preceding code. These functions are necessary to implement `JWTSubject`.

Let's try JWT

In order to test this, we have to create a new user in the users table of the database. So, add it by making a migration or executing the following query in your favorite SQL client:

```
INSERT INTO `finding_users`.`users`
(`id`, `email`, `password`, `api_token`)
VALUES
(1,'john@phpmicroservices.com',
'$2y$10$m5339OpNKEh5bL6Erbu9r..sjhaf2jDAT2nYueUqxnsR752g9xEFy',
NULL,);
```

The hashed password inserted manually corresponds to '123456'. Lumen will save your user passwords hashed for security reasons.

Open Postman and give it a try by making a POST call to `http://localhost:8084/user`. You should receive the following response:

```
Unauthorized.
```

This is happening because the `http://localhost:8084/user` method is protected by an authentication middleware. You can check this on your `routes.php` file. In order to get the user, it is necessary to provide a valid access token.

The method to get a valid access token is `http://localhost:8084/login`, so make a POST call with the parameters that correspond to the user we added, `email = john@phpmicroservices.com`, and password, `123456`. If they are correct, we will get a valid access token:

```
{"token":"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
eyJpc3MiOiJodHRwOi8vbG9jYWxob3N0OjgwODQvbG9naW4iLCJ
pYXQiOjE0ODA4ODI4NTMsImV4cCI6MTQ4MDg4NjQ1MywibmJmIjox
NDgwODgyODUzLCJqdGkiOiJVVnRpTExZTFRWcEtyWnhsIiwic3
ViIjoxfQ.jjgZO_Lf4dlfwYiOYAOhzvcTQ4EGxJUTgRSPyMXJ1wg"}
```

So now, we can use the preceding access token to make the POST call to `http://localhost:8084/user` as we did before. This time, we will get the user info:

```
{"id":1,"email":"john@phpmicroservices.com","api_token":null}
```

As you can see, it is very simple to protect your methods using a valid access token. It will make your application more secure.

Access Control List

This is a very common system in all applications regardless of their size. **Access Control List** (**ACL**) provides us with an easy way to manage and filter the permissions of every user. Let's look at this in a little more detail.

What is ACL?

The method that an application uses to identify every single user of the application is ACL. This is the system that informs the application what access rights or permissions a user or group of users have for a specific task or action.

Every task (function or action) has an attribute to identify which users can use it, and ACL is a list that links every task with every action, such as read, write, or execute.

ACL has the following two featured advantages for the applications that use it:

- **Management**: Using ACL in our application allows us to add users to groups and manage the permissions for each group. Also, it is easier to add, modify, or remove permissions to many users or groups.
- **Security**: Having different permissions for each user is better for the application's security. It avoids fake users or exploits breaking the application just by giving different permissions to normal users and administrators.

For our application based on microservices, we recommend having a different ACL for each microservice; this avoids having a single point of entry for the entire application. Remember that we are building microservices, and one of the requirements was that microservices should be isolated and independent; so, having a microservice to control the rest of them is not a good practice.

This is not a difficult task, it makes sense because every microservice should have different tasks and every task is different for each user in terms of permissions. Imagine that we have a user who has the following permissions. This user can create secrets and check the nearby secrets, but is not allowed to create battles or new users. Managing the ACL globally will be a problem in terms of scalability when a new microservice is added to the system or even when new developers join the team and they have to understand the complex system of global ACL. As you can see, it is better to have an ACL system for each microservice, so when you add a new one, it is not necessary to modify the ACL for the rest.

How to use ACL

Lumen provides us with an authentication process in order to get a user to sign up, log in, log out, and reset the password, and it also provides an ACL system whose classes are called `Gate`.

`Gate` allows us to know whether a specific user has permissions to do a specific action. This is very easy and it can be used in every method of your API.

To set up ACL on Lumen, you have to enable facades on your `bootstrap/app.php` by removing the semicolon from the `app->withFacades();` line; if this line does not appear on your file, add it.

It is also necessary to create a new file on `config/Auth.php` with the following code:

```
<?php
  return [
    'defaults' => [
      'guard' => env('AUTH_GUARD', 'api'),
    ],
    'guards' => [
      'api' => [
        'driver' => 'token',
        'provider' => 'users'
      ],
    ],
    'providers' => [
      'users' => [
        'driver' => 'eloquent',
        // We should get model name from JWT configuration
        'model' => app('config')->get('jwt.user'),
      ],
    ],
  ];
```

The preceding code is necessary to use the `Gate` class on our controller in order to check the user permissions.

Once this is set up, we have to define the different actions or situations available for a specific user. To do this, open the `app/Providers/AuthServiceProvider.php` file; on the `boot()` function, we can define every action or situation by writing the following code:

```
<?php
  /* Code Omitted */
  use Illuminate\Contracts\Auth\Access\Gate;
  class AuthServiceProvider extends ServiceProvider
  {
    /* Code Omitted */
    public function boot()
    {
      Gate::define('update-profile', function ($user, $profile) {
        return $user->id === $profile->user_id;
      });
    }
```

Once we have defined the situation, we can put it into our function. There are three different ways to use it: *allows*, *checks* and *denies*. The first two are the same, they return true when the defined situation returns true, and the last one returns true when the defined situation returns false:

```
if (Gate::allows('update-profile', $profile)) {
  // The current user can update their profile...
}
if (Gate::denies('update-profile', $profile)) {
  // The current user can't update their profile...
}
```

As you can see, it is not necessary to send the `$user` variable, it will get the current user automatically.

Security of the source code

The most likely situation is that your project will connect to an external service using some credentials, for example, a database. Where will you store all this information? The most common way is to have a configuration file inside your source where you place all your credentials. The main problem with this approach is that you will commit the credentials, and any person with access to the source will have access to them. It doesn't matter that you trust the people who have access to the repo; it is not a good idea to store credentials.

If you can't store credentials in your source code, you are probably wondering how you will store them. You have two main options:

- Environment variables
- External services

Let's take a look at each one so that you can choose which option is better for your project.

Environment variables

This way of storing credentials is very easy to implement–you only define the variables you want to store in the environment and later, you can get them in your source.

The framework we have chosen for our project is Lumen and with this framework, it is very easy to define your environment variables and later use them in the code. The most important file is the `.env` file, located in the root of your source. By default, this file is in `gitignore` to avoid being committed, but the framework comes with an `.env.example` example so that you can check how to define the variables. In this file, you can find definitions, such as the following ones:

```
DB_CONNECTION=mysql
DB_HOST=localhost
DB_PORT=3306
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret
```

The preceding definitions will create the environment variables and you can get the values in your code with a simple `env('DB_DATABASE');` or `env('DB_DATABASE', 'default_value');`. The `env()` function supports two parameters, so you can define a default value in case the variable you are trying to get is not defined.

The main benefit of using environment variables is that you can have different environments without needing to change anything in your source; you can even change the values without making any changes to your code.

External services

This way of storing credentials uses an external service to store all the credentials and they work more or less like the environment variables. When you need any credentials, you have to ask this service.

One of the mainstream credential storage systems these days is the HashiCorp Vault project, an open source tool that allows you to create a secure place where you can store your credentials. It has multiple benefits and we highlight, among them, the following ones:

- HTTP API
- Key rolling
- Audit logs
- Support for multiple secret backends

The main disadvantage of using an external service is the extra complexity you are adding to your application; you will add a new component to manage and keep up to date.

Tracking and monitoring

When you are dealing with security in your application, it is important to keep track and monitor what is happening in it. In `Chapter 6`, *Monitoring*, we implemented Sentry as a log and monitoring system and we also added Datadog as our APM, so you can use these tools to keep track of what is happening and to send you alerts.

However, what do you want to track? Let's imagine that you have a login system, this component is a good place to add your tracking. If you track each failed login for a user, you can know if somebody is trying to attack your login system.

Does your application allow users to add, modify, and delete content? Track any changes to the content so that you can detect untrusted users.

In security, there are no standards about what to track and what not to track, simply use your common sense. Our main recommendation is to create a list of sensitive points in your application that cover at least where the users can login, create content, or delete it and later use these lists as a starting point to add tracking and monitoring.

Best practices

As with any other part of the application, when you are dealing with security, there are some well-known best practices you need to follow or at least be aware of to avoid future issues. Here, you can find the most common ones related to web development.

File permissions and ownership

One of the most basic security mechanisms is file/folder permissions and ownership. Assuming that you are working on a Linux/Unix system, the main recommendation is to assign the ownership of your source code to the web server or PHP engine user. Regarding file permissions, you should be using the following setting:

- **500 permissions for directories (dr-x——)**: This setting prevents any accidental deletion or modification of files in the directory.
- **400 permissions for files (-r———)**: This setting prevents any users from overwriting files.

- **700 permissions (drwx——)**: This is for any writable directories. It gives full control to the owner and is used in upload folders.
- **600 permissions (-rw——-)** : This setting is for any writable files. It avoids any modification of your files by any user who is not an owner.

PHP execution locations

Avoid any future problems by allowing the execution of PHP scripts only on selected paths and deny any kind of execution in sensitive (writable) directories, for example, any upload directories.

Never trust users

As a rule of thumb, never trust the user. Filter any input that comes from anybody, you never know the dark intentions behind a form submit. Of course, never rely only on frontend filtering and validation. If you added filtering and validation to the frontend, do it again in the backend.

SQL injection

Nobody wants their data to be exposed or to be accessed by someone who does not have permission and this type of attack against your application is due to bad filtering or validation of the inputs. Imagine that you use a field to store the name of the user that is not correctly filtered, a malicious user can use this field to execute SQL queries. To help you to avoid this issue, use the ORM filtering methods or any filtering method available in your favorite framework when you are dealing with databases.

Cross-site scripting XSS

This is another type of attack against your application and is due to bad filtering. If you allow your users to post any kind of content on your page, it may be possible for some malicious users to add scripts to the page without your permission. Imagine that you have a comments section on your page and your input filtering is not the best, a malicious user can add a script as a comment that opens a spam pop up. Remember what we told you before–never trust your users–filter and validate everything.

Session hijacking

In this attack, the malicious user steals another user's session keys, giving the malicious user the opportunity to be like the other user. Imagine that your application deals with financial information and a malicious user can steal an admin session key, now this user can get all the information they need. Most of the time, sessions are stolen using an XSS attack, so first, try to avoid any XSS attacks. Another way of mitigating this issue is preventing JavaScript from having access to the session ID; you can do this in your `php.ini` with the `session.cookie.httponly` setting.

Remote files

Including remote files from your application can be very dangerous, you will never be 100% sure that the remote file you are including can be trusted. If at some point, the included remote file is compromised, attackers can do what they want, for example, remove all the data from your application.

An easy way to avoid this is to disable the remote files in your `php.ini`. Open it and disable the following settings:

- `allow_url_fopen`: This is enabled by default
- `allow_url_include`: This is disabled by default; if you disable the `allow_url_fopen` setting, it forces this to be disabled too

Password storage

Never store any passwords in plain text. When we say never, we mean never. If you think that you will need to check a user's password you are wrong, any kind of restoring or resupplying a missing password needs to go through a recovery system. When you store a password, you store the password hash that is mixed with some random salt.

Password policies

If you keep sensitive data and you don't want your application to be exposed by the passwords of your users, put a very strict password policy in place. For example, you can create the following password policy to reduce cracking and dictionary attacks:

- At least 18 characters
- At least 1 uppercase
- At least 1 number
- At least 1 special character
- Not been used before
- Not being a concatenation of the user data, changing vowels to numbers
- Expires every 3 months

Source code revelation

Keep the source code out of sight of curious eyes, if for some reason your server is broken, all the source code will be exposed as plain text. The only way to avoid this is to only keep the required files in the web server root folder. As an addition, be careful with special files, such as `composer.json`. If we expose our `composer.json`, everybody will know the different versions of each of our libraries, giving them an easy way of knowing any possible bugs.

Directory traversal

This kind of an attack tries to access files that are stored outside the web root folder. Most of the time, this is due to bugs in the code, so the malicious user can manipulate variables that reference files. There is no easy way to avoid this; however, if you use external frameworks or libraries, keeping them up to date will help you.

These are the most obvious security concerns you need to be aware of, but this is not an exhaustive list. Subscribe to security newsletters and keep all your code up to date to reduce risks to the minimum.

Summary

In this chapter, we talked about security and authentication. We showed you how you can encrypt your data and communication layers; we even showed you how to build a robust login system, and how you can deal with the secrets of your application. Security is a very important aspect in any project, so we gave you a small list of common security risks you need to be aware of and, of course, the main recommendation–never trust your users.

8
Deployment

Throughout the previous chapters, you have learned how to develop an application based on microservices. Now, it is time to learn about the deployment of your application, learning the best strategies to automate and roll back your application and also, doing back ups and restores if needed.

Dependency management

As we mentioned in `Chapter 5`, *Microservice Development*, **Composer** is the most-used dependency management tool; it can help us move a project from the development environment to production in the deployment process.

There are some different opinions about what the best workflow for the deployment process is, so let's look at the advantages and disadvantages of every case.

Composer require-dev

To be used on the development environment, Composer provides a section on their `composer.json`, called `require-dev`, and when we need to install some libraries on our application that do not need to be on production, we have to use it.

As we already know, the command to install a new library using Composer is `composer require library-name`, but if we want to install a new library, such as testing libraries, debugging libraries, or any others that do not make sense on production, we can use `composer require-dev library-name` instead. It will add the library to the `require-dev` section and when we deploy the project to production, we should use the `--no-dev` parameter when executing `composer install --no-dev` or `composer update --no-dev` in order to avoid installing development libraries.

The .gitignore file

With the `.gitignore` file, it is possible to ignore files or folders that you do not want to track. Even though Git is a versioning tool, many developers use this in the deployment process. The `.gitignore` file contains a list of files and folders that will not be tracked on your repository when they change. This is usually used to upload folders that contain images or any other file uploaded by users and also, it is used for the vendor folder, the folder that contains all the libraries used on the project.

Vendor folder

The `vendor` folder contains all the libraries used in our application. As previously mentioned, there are two different ways of thinking about how to use the `vendor` folder. There are advantages and disadvantages of including Composer in production in order to get the `vendor` folder from the repository once the application is deployed or when it is downloading the libraries used on development into production.

Deployment workflows

The deployment workflow can be different in every application depending on the project needs. For example, if you want to keep the whole project, including the `vendor` folder, in the repository or if you prefer to get the libraries from Composer once the project is deployed. We will look at a couple of the most common workflows in this chapter.

Vendor folder on repository

The first deployment workflow has the entire application on the repository. This is when we use Composer for the first time in our development environment and we push the `vendor` folder to our repository, so all the libraries will be kept on the repository.

Therefore, on production we will get the entire project from the repository without needing to do a Composer update because our libraries were put in production with the deployment. So, you do not need Composer in production.

Advantages of including the `vendor` folder in the repository are as follows:

- You know that the same code (including libraries) was working on development.
- Minor risk of breaking updated libraries in production.

- You do not depend on external services in the deployment process. Sometimes, the libraries are not available at a specific moment.

Disadvantages of including the `vendor` folder on the repository are as follows:

- Your repository has to store libraries already stored on Composer. The space needed can be a problem if you need many or large libraries.
- You are storing code that is not yours.

Composer in production

The second deployment workflow has two different ways of proceeding, but both of them do not need to store the `vendor` folder in the repository; they will get the libraries from Composer once the code is deployed to production.

Once the code is deployed to production, the `composer update` command will be executed either **manually** or **automatically** in the deployment process.

Advantages of running Composer in production are as follows:

- You are saving space in your repository
- You can execute–optimize-autoload in production in order to map the libraries added

Disadvantages of running Composer in production are as follows:

- The deployment process will depend on external services.
- Major risk in some situations when updating packages. For example, if a library is suddenly modified or corrupted, your application will break.

Frontend dependencies

It is necessary to know that it is possible to have management dependencies on the frontend side too, so it is possible to choose if it is better to put it on the repository or not. Grunt and Gulp are two of the most used tools in order to automatize tasks in your application. Also, if your application based on microservices has a frontend part you should use the following tools in order to manage styles and assets.

Grunt

Grunt is a tool to automatize tasks on your application. Grunt can help you to concat or minify JS and CSS files, optimize images, or even help you with unit testing.

Every task is implemented by a Grunt plugin developed on Javascript. Also, they use Node.js, so it makes Grunt a multiplatform tool. You can check all the available plugins at `http://gruntjs.com/plugins`.

It is not necessary to learn Node.js, just install Node.js and you will have Node Packaged Modules available to install Grunt (and many other packages). Once Node.js is installed, run the following command:

```
npm install grunt-cli -g
```

Now, you can create a `package.json` that will be read by the NPM command:

```
{
  "name": "finding-secrets",
  "version": "0.1.0",
  "devDependencies": {
    "grunt": "~0.4.1"
  }
}
```

Then, `npm install` will install the dependencies contained in the `package.json` file. Grunt will be stored in the `node_modules` folder. Once Grunt is installed, it is necessary to create a `Gruntfile.js` to define the automated tasks, as shown in the following code:

```
'use strict';
module.exports = function (grunt) {
grunt.initConfig({
  pkg: grunt.file.readJSON('package.json'),
});
//grunt.loadNpmTasks('grunt-contrib-xxxx');
//grunt.registerTask('default', ['xxxx']);
};
```

There are three sections to define the automated tasks:

- **InitConfig**: This refers to tasks that will be executed by Grunt
- **LoadNpmTask**: This is used to load the required plugin to make the tasks
- **RegisterTask**: This registers the tasks that will run

Once we decide what plugin to install and define all the necessary tasks, run grunt on your terminal to execute them.

Gulp

Like Grunt, **Gulp** is also a tool to automatize tasks and it is also developed on NodeJS, so it is necessary to install Node.js in order to have NPM available to install it.
Once we have installed Node.js, we can install Gulp globally by running the following command:

```
npm install -g gulp
```

Another way of installing gulp, and is the recommended option, is locally and you can do it with the following command:

```
npm install --save-dev gulp
```

All the tasks should be included in a `gulpfile.js` located on the root project to be automated:

```
var gulp = require('gulp');
gulp.task('default', function () {
});
```

The preceding code is very simple. As you can see, the code is `gulp.task`, the task name, and then the `function` defined for that task name.

Once you have the functions defined, you can run `gulp`.

SASS

CSS is complex, large, and hard to maintain. Can you imagine maintaining a file with thousands and thousands of lines? This is where Sass can be used. This is a preprocessor that adds features, such as variables, nesting, mixins, inheritance, and others to CSS that makes CSS a real development language.

Syntactically Awesome Stylesheets (**SASS**) is a metalanguage of CSS. It is a script language that is translated to CSS. SassScript is the Sass language and it has two different syntaxes:

- **Indented syntax:** This uses the indent to separate block codes and the new line character to separate rules
- **SCSS**: This one is an extension of the CSS syntax, it uses braces for code blocks and semicolons to separate lines within a block

The indented syntax has `.sass` extensions, and SCSS has `.scss` extensions.

Sass is very simple to run. Once it is installed, just run `sass input.scss output.css` on your terminal.

Bower

Bower is a dependency management like Composer, but it works for the frontend side. It is also based on Node.js, so once Node.js is installed, you can install Bower using NPM.
Using Bower, it is possible to have all the frontend libraries updated, without needing to update them manually.
The command to install Bower once Node.js is installed is as follows:

```
npm install -g bower
```

Then, you can execute `bower init` in order to create the `bower.json` file on your project.

The `bower.json` file will remind you of `composer.json`:

```
{
  "name": “bower-test”,
  "version": "0.0.0",
  "authors": [
    "Carlos Perez and Pablo Solar"
  ],
  "dependencies": {
    "jquery": "~2.1.4",
    "bootstrap": "~3.3.5"
    "angular": "1.4.7",
    "angular-route": "1.4.7",
  }
}
```

In the preceding code, you can you can see the dependencies added to the project. They can be modified in order to have these dependencies installed on your application like Composer works. Also, the commands to work with Bower are very similar to Composer:

- **bower install:** This is to install all the dependencies on `bower.json`
- **bower update:** This is to update the dependencies contained on `bower.json`
- **bower install package-name:** This installs a package on Bower

Deploy automation

At some point, your application will be deployed to production. If your application is small and you only use a few containers/servers, everything will be fine, you can easily manage all your resources (containers, VMs, servers, and so on) by hand in each deployment. However, what happens if you have hundreds of resources you need to update on each deployment? In this case, you need some kind of deployment mechanism; even if you have a small project and only one container/server, we recommend automating your deployment.

The main benefits of using an automatic deployment process are as listed:

- **Easy to maintain:** Most of the time, the steps needed by the deployment can be stored in files so that you can edit them.
- **Repeatable:** You can execute the deployment again and again and it will follow the same steps each time.
- **Less error-prone:** We are humans and, as humans, we make mistakes multitasking.
- **Easy to track:** There are multiple tools you can use to keep a log of everything that happens in every commit. These tools can also be used to create groups of users who can make deploys. The most common tools you can use are **Jenkins**, **Ansible Tower**, and **Atlassian Bamboo**.
- **Easy to release more often:** Having a deployment pipeline in place will help you develop and deploy faster because you will not spend time dealing with the push of your code to production.

Let's look at some ways to automate your deployments, starting with the simplest options and increasing the complexity and features more and more. We will analyze the pros and cons of each one so that, at the end of the chapter, you will be available to choose the perfect deployment system for your project.

Simple PHP script

This is the most simple way you can automate your deployments–you can add a script to your code (in a public location), as shown in the following code:

```
<?php
define('MY_KEY', 'this-is-my-secret-key');
if ($_SERVER['REQUEST_METHOD'] ===
'POST' && $_REQUEST['key']  === MY_KEY) {
    echo shell_exec('git fetch && git pull origin master');
}
```

In the preceding script, we only do a pull from master if the script is reached with the correct key. As you can see, it is very easy and it can be fired by anyone who knows the secret key, for example, by a browser. If your code repository allows the set up of webhooks, you can use them to fire your script each time a push or commit is done in your project.

Here are the pros of this deployment method:

- It's easy to create if the work required is small, for example, a `git pull`
- It's easy to keep track of changes to the script
- It's easy to be fired by you or any external tools

Here are the disadvantages of this deployment method:

- The web server user needs to be able to use the repo
- It increases in complexity when you need to deal with, for example, branches or tags
- It is not easy to use when you need to deploy to multiple instances, you will need external tools like rsync
- Not very secure. If your key gets found out by a third party, they can deploy on your server whatever they want

In an ideal world, all your commits to production will be perfect and pristine, but you know the truth–at some point in the future, you will need to roll back all your changes. If you have this deployment method in place and you want to create a rollback strategy, you have to increase the complexity of your PHP script so that it can manage tags. Another not-recommended option is, instead of adding a rollback to your scripts, you can do, for example, a `git undo` and push all the changes again.

Ansible and Ansistrano

Ansible is an IT automation engine that can be used to automate cloud provisioning, manage configurations, deploy applications, or orchestrate services among other uses. This engine does not use an agent, so there is no need for additional security infrastructure, it was designed to be used through SSH. The main language used to describe your automation jobs (also called **playbooks**)is YAML and its syntax is similar to English. Due to the fact that all your playbooks are simple text files, you can store them easily in your repository. An interesting feature that you can find in Ansible is its Galaxy, a hub of add-ons you can use in your playbooks.

Ansible requirements

Ansible uses the SSH protocol to manage all the hosts, and you only need to install this tool on one machine–the machine you will use to manage your fleet of hosts. The main requisite for the control machine is Python 2.6 or 2.7 (from Ansible 2.2 it has support for Python 3), and you can use any OS except Microsoft Windows.

The only requirement on the managed hosts is Python 2.4+, which comes installed by default by most of the UNIX-like operating systems.

Ansible installation

Assuming that you have the correct Python version on your control machine, installing Ansible is very easy with the help of the package managers.

On RHEL, CentOS and similar linux distributions execute the following command to install Ansible:

```
sudo yum install ansible
```

The Ubuntu command is as follows:

```
sudo apt-get install software-properties-common \
&& sudo apt-add-repository ppa:ansible/ansible \
&& sudo apt-get update \
&& sudo apt-get install ansible
```

The FreeBSD command is as follows:

```
sudo pkg install ansible
```

The Mac OS command is as follows:

```
sudo easy_install pip \
&& sudo pip install ansible
```

What is Ansistrano?

Ansistrano is an open source project composed with `ansistrano.deploy` and `ansistrano.rollback`, two Ansible Galaxy roles used to easily manage your deployments. It's considered to be the Ansible port for Capistrano.

Once we have Ansible available on our machine, it is very easy to install the Ansistrano roles with the following command:

```
ansible-galaxy install carlosbuenosvinos.ansistrano-deploy \
carlosbuenosvinos.ansistrano-rollback
```

After the execution of this command, you will be able to use Ansistrano in your playbooks.

How does Ansistrano work?

Ansistrano deploys your application following the Capistrano flow:

1. **Setup phase**: In this phase, Ansistrano creates the folder structure that will hold the application releases.
2. **Code update phase**: In this phase, Ansistrano puts your release in your hosts; it can use rsync, Git, or SVN among other methods.
3. **Symlink phase** (see below): After the new release is deployed, it changes the current softlink that points the available release to the new release location.
4. **Cleanup phase**: In this phase Ansistrano removes old releases stored in your hosts. You can configure the number of releases in your playbooks through the `ansistrano_keep_releases` parameter. In following examples you will se how this parameter works

With Ansistrano, you can hook custom tasks to be executed before and after each task.

Let's look at a simple example to explain how it works. Imagine that your application is deployed to `/var/www/my-application`; the contents of this folder will be similar to the following example after your first deployment:

```
-- /var/www/my-application
 |-- current -> /var/www/my-application/releases/20161208145325
 |-- releases
 |   |-- 20161208145325
 |-- shared
```

As you can see from the preceding example, the current symlink points to the first release we have in our host. Your application will always be available in the same path, `/var/www/my-application/current`, so you can use this path in any place you need, for example, NGINX or PHP-FPM.

As your deployments continue, Ansistrano will deal with the deploys for you. The next example will show you what your application folder will look like after a second deployment:

```
-- /var/www/my-application
 |-- current -> /var/www/my-application/releases/20161208182323
 |-- releases
 |   |-- 20161208145325
 |   |-- 20161208182323
 |-- shared
```

As you can see from the preceding example, now we have two releases in our hosts and symlink was updated to point to the new version of your code. What happens if you do a rollback with Ansistrano? Easy, this tool will remove the latest release you have in your hosts and update the symlink. In our example, your application folder content will be similar to this:

```
-- /var/www/my-application
 |-- current -> /var/www/my-application/releases/20161208145325
 |-- releases
 |   |-- 20161208145325
 |-- shared
```

To avoid problems, if you try to roll back and Ansistrano can't find a previous version to move to, it will do nothing, keeping your hosts without changes.

Deploying with Ansistrano

Now, let's create a small automation system with Ansible and Ansistrano. We are assuming that you have a known and persistent infrastructure available where you will push your app or microservice. Create a folder in your development environment to keep all your deployment scripts.

In our case, we previously created three VMs in our local environment with SSH enabled. Note that we are not covering the provisioning of those VMs but if you want, you can even use Ansible to do it for you.

The first thing you need to create is a `hosts` file. In this file, you can store and group all your servers/hosts so that you can later use them in the deployment:

```
[servers:children]
 production
 staging

[production]
192.168.99.200
192.168.99.201

[stageing]
192.168.99.100
```

In the preceding configuration, we created two groups of hosts--`production` and `staging`. On each one of them, we have a few hosts available; in our case, we set up the IP address of our local VM for testing purposes, but you can use URIs if you want. One of the advantages of grouping your hosts is the ability you have to even create bigger groups; for example, you can create a group formed by other groups. For example, we have a `servers` group that wraps all the production and staging hosts. If you are wondering what happens if you have a dynamic environment, no problem; Ansible has your back and comes with multiple connectors that you can use to get your dynamic infrastructure, for example, from AWS or Digital Ocean, among others.

Once you have your `hosts` file ready, it is time to create our `deploy.yml` file where we will store all the tasks we want to execute in our deployment. Create a `deploy.yml` file with the following content:

```
---
- name: Deploying a specific branch to the servers
 hosts: servers
 vars:
     ansistrano_allow_anonymous_stats: no
     ansistrano_current_dir: "current"
```

```
        ansistrano_current_via: "symlink"
        ansistrano_deploy_to: "/var/www/my-application"
        ansistrano_deploy_via: "git"
        ansistrano_keep_releases: 5
        ansistrano_version_dir: "releases"

        ansistrano_git_repo: "git@github.com:myuser/myproject.git"
        ansistrano_git_branch: "{{ GIT_BRANCH|default('master') }}"

    roles:
        - { role: carlosbuenosvinos.ansistrano-deploy }
```

Thanks to Ansistrano, our deployment tasks are very easy to define, as you can see from the preceding example. What we did is create a new task that will be executed in all the hosts wrapped under the tag servers, and define a few variables available for the Ansistrano role. Here, we defined where we will deploy our application on each host, the method we will use for the deploy (Git), how many releases we will keep in the hosts (5), and the branch we want to deploy.

An interesting feature of Ansible is that you can pass variables from the command line to your generic deployment process. This is what we do in the following line:

```
ansistrano_git_branch: "{{ GIT_BRANCH|default('master') }}"
```

Here, we are using a `GIT_BRANCH` variable to define which branch we want to deploy; if Ansible can't find this defined variable, it will use master.

Are you ready to test what we have done? Open a terminal and go to the location where you have stored the deployment tasks. Imagine that you want to deploy the latest versions of your code to your production hosts; you can do it with the following command:

```
ansible-playbook deploy.yml --extra-vars "GIT_BRANCH=master" --limit
production -i hosts
```

In the preceding command, we are telling Ansible to use our `deploy.yml` playbook and we also defined our `GIT_BRANCH` to be master so that this branch will be deployed. As we have all our hosts in the hosts file and we only want to make the deployment to the `production` hosts, we limited the execution to the desired hosts with `--limitproduction`.

Now, imagine that you have a new version ready, all your code was committed and tagged under the `v1.0.4` tag, and you want to push this release to your staging environment. You can do it with a very simple command:

```
ansible-playbook deploy.yml --extra-vars "GIT_BRANCH=v1.0.4" --limit
staging -i hosts
```

As you can see, deploying your application is very easy with Ansible/Ansistrano and it is even easier to roll back to a previously deployed release. To manage the rollbacks, you only need to create a new playbook. Create a `rollback.yml` file with the following content:

```
---
- name: Rollback
 hosts: servers
 vars:
     ansistrano_deploy_to: "/var/www/my-application"
     ansistrano_version_dir: "releases"
     ansistrano_current_dir: "current"
 roles:
     - { role: carlosbuenosvinos.ansistrano-rollback }
```

In the preceding piece of code, we are using the Ansistrano rollback role to move to the previous deployed release. If you only have one release in your hosts, Ansible will not undo the changes because it is not possible. Do you remember the variable we set up in the `deploy.yml` file, called `ansistrano_keep_releases`? This variable is very important to know how many rollbacks you can do in your hosts, so adjust it to your needs. To roll back your production servers to the previous release, you can use the following command:

```
ansible-playbook rollback.yml --limit production -i hosts
```

As you can see, Ansible is a very powerful tool that you can use for your deployments, but it is not used only for deployments; you can even use it for orchestration, for example. With a vibrant community and with RedHat supporting the project, Ansible is a necessary tool.

Ansible has an enterprise version of a web tool that you can use to manage all your Ansible playbooks. Even though it needs a paid subscription, if you manage less than ten nodes, you can use it for free.

Other deployment tools

As you can imagine, there are multiple and different tools that you can use to do your deployments and we cannot cover all of them in this book. We wanted to show you a simple one (PHP scripts) and a more complex and powerful one (Ansible), but we don't want you to finish this chapter without knowing the other tools that you can use:

- **Chef**: This is an interesting open source tool you can use to manage your infrastructure as code.
- **Puppet**: This is an open source configuration management tool with a paid enterprise version.
- **Bamboo**: This is a continuous integration server from Atlassian and, of course, you need to pay to use this tool. This is the most complete tool you can use combine with the Atlassian catalog of products.
- **Codeship**: This is a cloud continuous deployment solution that aims to be a tool focused on being an end-to-end solution for running tests and deploying apps
- **Travis CI**: This is a similar tool to Jenkins used for continuous integration; you can also use it to make your deployments.
- **Packer, Nomad, and Terraform**: These are different tools from HashiCorp that you can use to write your infrastructure as code.
- **Capistrano**: This is a well-known remote server automation and deployment tool, which is easy to understand and easy to use.

Advanced deployment techniques

In the previous section, we showed you some ways you can deploy your application. Now, it's time to increase the level of complexity with some advanced techniques used on big deployments.

Continuous integration with Jenkins

Jenkins is the most known continuous integration application; being an open source project allows you to create your own pipeline with high flexibility. It was built in Java, so this is the main requirement you have if you want to install this tool. With Jenkins, everything is easier, even the installation. For example, you can spin up a Docker container with the last version with only a few commands:

```
docker pull jenkins \
&& docker run -d -p 49001:8080 -t jenkins
```

The preceding command will download and create a new container with the latest Jenkins version, ready to use.

The main idea behind Jenkins is the concept of a job. A job is a sequence of commands or steps you can execute automatically or by hand. With jobs and the use of plugins (available to download from the web UI), you can create your custom workflow. For example, you can create a workflow similar to the next one that is fired by your repository as soon as a commit/push happens:

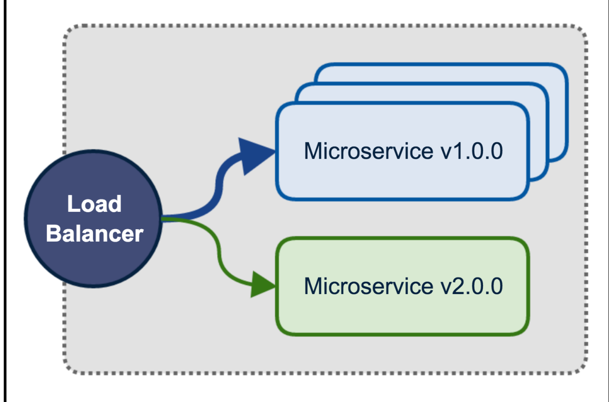
1. A unit test plugin starts testing your application.
2. As soon as it passes, a code sniffer plugin checks your code.
3. If the previous steps are okay, Jenkins connects through SSH to a remote host.
4. Jenkins pulls all the changes in the remote host.

The preceding example is an easy one; you can improve and complicate the example more, firing an Ansible playbook instead of using SSH.

This application is so versatile that you can use it in any you want. For example, you can use it to check the replication status of your Master-Slave database. In our opinion, this application is worth a try and you can find examples of any kinds of tasks adapted to this software.

Blue/Green deployment

This deployment technique relies on having a duplicate of your infrastructure so that you can have a new version of your application installed in parallel with the current version. In front of your application, you have a router or **Load Balancer** (LB) that is used to redirect the traffic to the desired version. As soon as you have your new version ready, you only need to change your router/LB to redirect all the traffic to the new version. Having two sets of releases gives you the flexibility and advantage of easy rollbacks and also gives you time to ensure that the new version works fine. Refer to the following diagram:



Blue/Green deployment on microservices

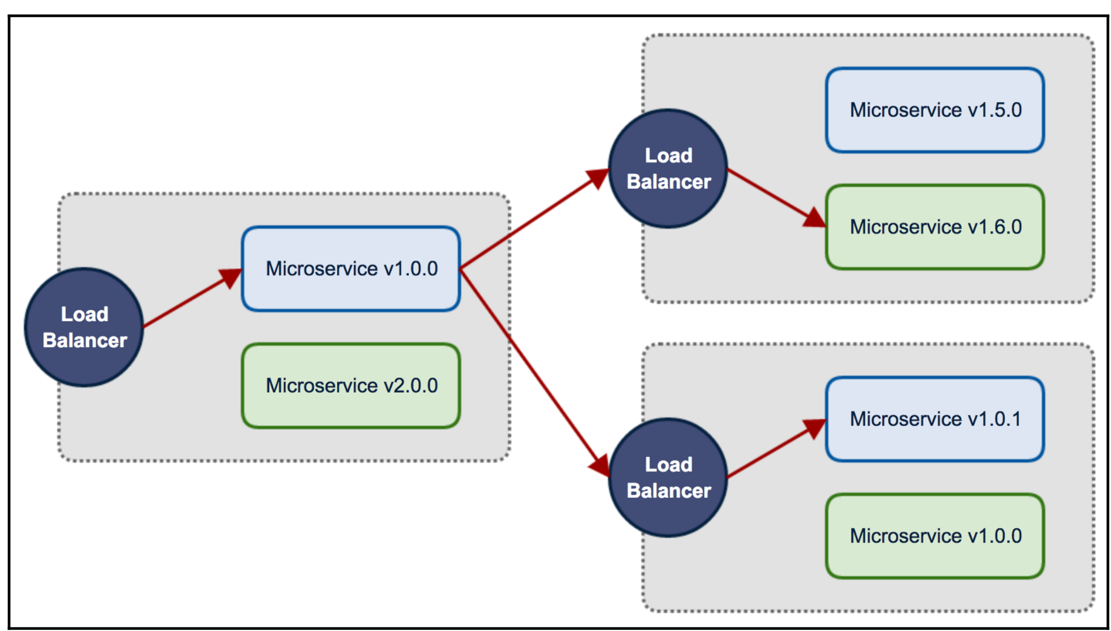
As you can see from the preceding image, the Blue/Green deployment can be done at any level of your application. In our example image, you can spot a microservice that is getting ready to deploy a new version but has not been released yet, and you can see that some microservices released the latest version of their code, keeping the previous one for rollback.

This technique is widely used by big tech companies without any kinds of problems; the main disadvantage is the increased amount of resources you need to run your application–more resources means more money to be spent on your infrastructure. If you want to give it a try, the most-used load balancers on this kind of deployments are **ELB**, **Fabio**, and **Traefik**, among others.

Canary releases

Canary releases is a similar deployment technique to the Blue/Green one with a subtle difference–only a small amount of hosts are upgraded at the same time. Once you have a portion of your hosts with the release you want, using a cookie, a lb, or a proxy, a fraction of the traffic is redirected to the new version.

This technique allows you to test your changes with a small portion of your traffic; if the application behaves as expected, we continue migrating more hosts to the new version until all the traffic is redirected to the new version of your application. Take a look at the following diagram:



Canary releases with microservices

As you can see from the preceding image, there are four instances of some microservices; three of them keep the old version of the application and only one has the latest version. The LB is used to split the traffic between the different versions, sending the majority of the traffic to the **v1.0.0** and only a small portion of the traffic to the **v2.0.0**. If everything is fine, the next step will be increasing the number of **v2.0.0** instances, reducing the number of **v1.0.0** instances, and redirecting more traffic to the new versions.

This deployment technique adds a little bit of complexity to your current infrastructure, but allows you to start testing your changes with small portions of users/traffic. Another benefit is the reuse of your existing infrastructure; you don't need to have duplicated set of hosts to make your deployments.

Immutable infrastructure

These days, a trend in the tech industry is to use immutable infrastructure. When we say immutable infrastructure, we mean that what you have in your development environment is later deployed to production without any changes. You can achieve this thanks to the containerization technology and some tools, such as Packer.

With Packer, you can create an image of your application and later distribute this image through your infrastructure. The main benefit of this technique is that you ensure that your production environment will behave like your development. Another important aspect is security; imagine that there is a security breach in your NGINX container, a new release with the base image update will solve the issue and it will be propagated with your application without the need of external intervention.

Backup strategies

In any project, the backup is one of the most important ways to avoid losing data. In this chapter, we will learn the backup strategies to be used in your application.

What is backup?

Backup is the process of saving code or data in a different place to where the code or data is usually stored. This process can be done using different strategies, but all of them have the same goal–not to lose data in order for it to be accessed in the future.

Why is it important?

A backup can be done for two reasons. The first one is the loss of data due to a hack attack, corrupted data, or any mistakes executing queries on the production server. This backup will help restore the lost or corrupted data.

The second reason is policy. The law says that it is required to store user data for years. Sometimes, this functionality is done by a system, but a backup is another way to store this data.

To sum up, backups allow us to be calm. We ensure that we are doing things properly and in case any disasters happen, we have a solution to fix them fast and without (significant) data loss.

What and where we need to back up

If we are using some repositories, such as Git, in our application, this can be our backup place for the files. The assets or any other files uploaded by users should be backed up too.

A good practice to see if we are backing up all the necessary files is reviewing the `.gitignore` file and ensuring that we have backed up all the files and folders included in that folder.

Also, the most important and precious thing to back up is the database. This should be backed up more frequently.

Do not store a backup in the same place where the application is working. Try to have different locations for the backup copies.

Backup types

The backups can be full, incremental, or differential. We will look at the difference between them and how they work. Applications usually combine different backup types: a full backup with incremental or differential backups.

Full backup

The full backup is the basic one; it consists of generating a full copy of the current application. This option is used by large applications periodically, and small applications can use it daily.

The advantages are as follows:

- The full application is backed up in a single file
- It always generates a full copy

The disadvantages are as follows:

- The time to generate it can be very long
- The backup will need a lot of disk space

Please note that it's generally good practice to include date/time in the backup file name so you can know when was created only looking to the file name.

Incremental and differential backups

The incremental backup copies the data that has changed since the last backup. The `datetime` should be included in this kind of backup in order to be checked by the backup tools the next time a new backup is generated.

The advantages are as follows:

- It is faster than a full backup
- It takes up less disk space

The disadvantages are as follows:

- The entire application is not stored in a single generated backup

There is another type, called **differential**. This is like the incremental backup (copying all the data that has changed since the last backup); it is executed the first time and then it will continue copying all the modified data from the last full backup.

So, it will generate more data than the incremental one but less than the full one after the first time. This type is an intermediate one between full and incremental. It needs more space and time than incremental and less than full.

Backup tools

It is possible to find many backup tools. The most common tool in large projects is Bacula. There are other similar ones for small projects like a custom script that will be run frequently.

Bacula

Bacula is a backup management tool. This application manages and automates the backup tasks and it is perfect for large applications. This tool is a little complex to set up, but once you have it ready, it does not need any other changes and it will work without any problems.

Three different parts exist on Bacula, and every single part needs to be installed in a different package:

- **Director**: This manages all the backup processes
- **Storage**: This is where the backup is stored
- **File**: This is the client machine where we have our application running

In our application based on microservices, we will have many files (one for each microservice) and optionally, many storage locations (in order to have different locations for the backups) and directors.

This tool works with daemons. Each part has its own daemon and each daemon works following its own config files. The config files are set up in the installation process, and it is only necessary to change some little things, such as the remote IP address, certificates, or the plan to automate backups.

The security of Bacula is really amazing–every part (director, storage, and file) has its own key and depending on the connection, it is encrypted using it. Also, Bacula allows TLS connections for more security.

Bacula allows you to do full, incremental, or differential backups, and they can be automated on the director's part.

Percona xtrabackup

XtraBackup is an open source tool to do hot database backups on your application without blocking the database. This is possibly the most important feature of this application.

This tool allows MySQL databases, such as MariaDB and Percona, to perform streaming and compression and do incremental backups.

The advantages are as follows:

- Fast backups and restores
- Uninterrupted transaction processing during the backups
- Saves disk space and network bandwidth
- Automatic backup verification

Custom scripts

The fastest way to create backups is to use a custom script in production. This is a script that when it runs, it creates a backup by doing a `mysqldump` (if we are using a MySQL database), compressing the desired files, and putting them in the desired location (ideally, remotely on a different machine).

These scripts should be executed by a cronjob that can be set up to run them every day or week.

Validating backups

As part of the backup strategies, it is a good practice to have techniques to validate the data stored on the backups. If you have backups with errors, it is like not having any backups at all.

In order to check that our backups are valid, are not corrupted, and they work as expected, it is necessary to frequently do a mock restore to avoid failure if we need to restore them in the future.

Be ready for the apocalypse

No one wants to have to restore a backup, but it is necessary to be ready in case any of your microservices are broken or corrupted and we have to react fast.

The first step is knowing where the most recent backup of your application is in order to restore it as soon as possible.

If the problem is related to the database, we have to put the application under maintenance, restore the database backup, check that it works properly, and then, make the application live again.

If the problem is related to something like assets or files, it is possible to restore without putting the application under maintenance.

Keep calm and get your backup.

Summary

Now you know how to deploy your application to production and automate the deployment process. Also, you learned what you have to deploy and that you can get it from any dependency management, how to do a rollback if necessary, and the different strategies to back up your application.

9
From Monolithic to Microservices

In this chapter, we will look at some possible strategies to follow when we have to transform a monolithic application into microservices, along with some examples. This process can be a little difficult if we already have a big and complex application, but fortunately, there are some well-known strategies that we can follow in order to avoid problems throughout this process.

Refactor strategies

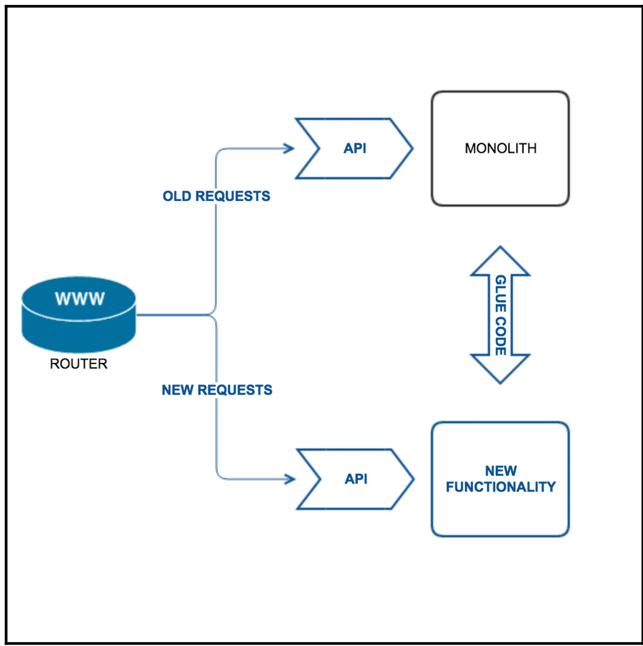
The process of transforming a monolithic application into microservices is a refactor code in order to modernize your application. This should be done incrementally. Trying to transform the entire application into microservices in one step could cause problems. Little by little, it will create a new application based on microservices and finally, your current application will disappear because it will be transformed into little microservices leaving the original application empty or maybe it will also be a microservice.

Stop diving

When your application is already a hole, you have to stop diving in that hole. In other words, stop making your monolithic application bigger. This is when you have to implement a new functionality, it is necessary to create a new microservice and connect it to the monolithic application instead of continuing developing the new functionalities in the monolith.

To do this, when a new functionality is implemented, we will have the current monolith, the new functionality, and also two more things:

- **Router:** This is responsible for the HTTP requests; in other words, this is like a gateway that knows where it needs to send every request, either to the old monolith or to the new functionality.
- **Glue code:** This is responsible for connecting the monolithic application to the new functionality. It is very common for the new functionality to need to access the monolithic application in order to get data or any necessary functions from it:



Stop diving strategy

Regarding glue code, there are 3 different possibilities to access the application from the new functionality to the monolith:

- Create an API on the monolith side to be consumed by the new function
- Connect directly with the monolith database
- Have a synchronized monolith copy of the database on the functionality side

As you can see, this strategy is a pretty way to start developing microservices in your current monolithic application. In addition, the new functionality can scale, deploy, and develop in an isolated way from the monolith, improving your application. However, this does not solve the problem, it just avoids making the current problem any bigger, so let's take a look at two more strategies.

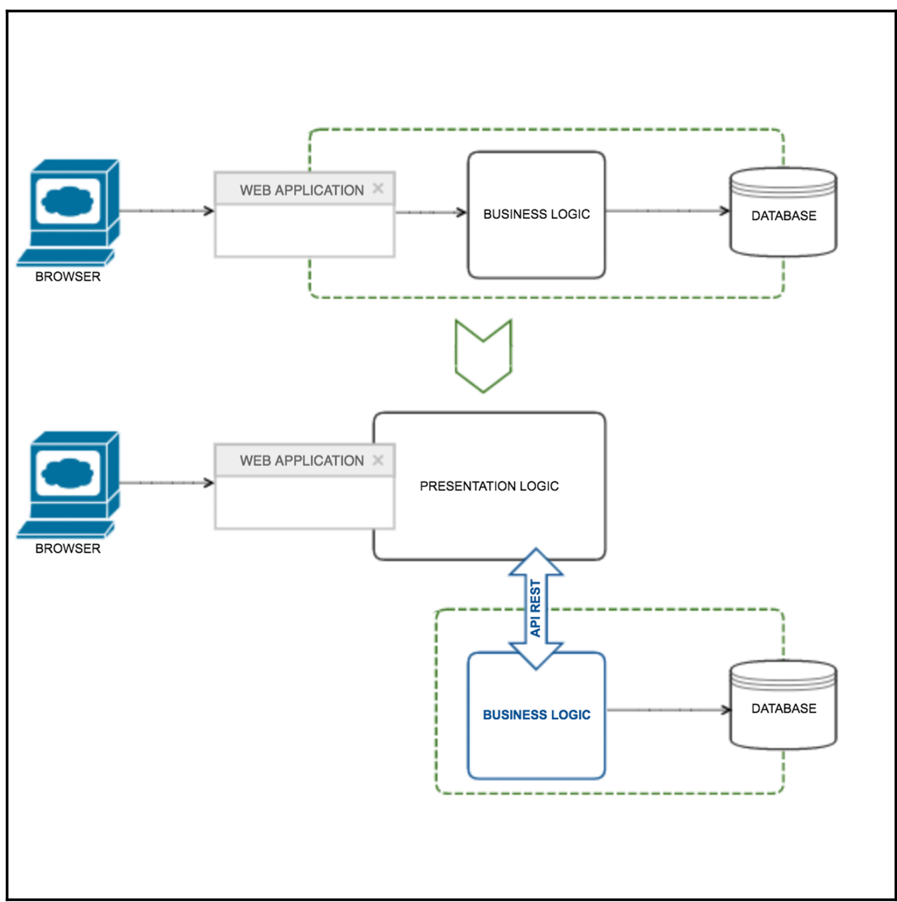
Divide frontend and backend

Another strategy is to divide the logic presentation part from the data access layer. An application usually has at least 3 different parts:

- **Presentation layer:** This is the user interface, in other words, the HTML language of a website
- **Business logic layer:** This consists of the components used to implement the business rules
- **Access data layer:** This has components that have access to the database

There is usually a separation between the presentation layer and the business logic and access data layers. The business layer has an API that has one or more facades that encapsulate the business logic components. From this API, it is possible to divide the monolith into 2 smaller applications.

After the division, the presentation layer makes calls to the business logic. Take a look at the following example:



Divide frontend and backend strategy

This division has 2 different advantages:

- It allows you to scale and develop two different and isolated applications
- It provides you with an API that can be consumed for future microservices

The problem with this strategy is that it is only a temporary solution, it can be transformed into one or two monolithic applications, so let's look at the next strategy in order to remove the rest of the monolith.

Extraction services

The last strategy is about isolating modules from the resultant or resultants monoliths. Little by little, we can extract modules from it and make a microservice from every module. Once we have all the important modules extracted, the resultant monolith will also be a microservice or it will even disappear. The general idea is to create logical groups of features that will be your future microservices.

A monolithic application usually has many potential modules to be extracted. The priority must be set by selecting the easier ones first and then the most beneficial ones. The easier ones will give you necessary experience in extracting modules into microservices to do it with the important ones later.

Here are some tips to help you choose the most beneficial ones:

- Modules that change frequently
- Modules that require different resources to the monolith
- Modules that require expensive hardware

It is useful to look for the existing coarse-grained boundaries, they are easier and cheaper to convert to microservices.

How to extract a module

Now, let's look at how to extract a module, we will use an example to make the explanation a little easier to understand. Imagine that your monolithic application is a blog system. As you can imagine, the core functionality is the posts that are created by users and each post supports comments. As you can see from our small description, you can define the different modules of your application and decide which is the most important.

Once you have the description and features of your application clear, you can continue with the general steps used to extract a module from your monolithic application:

1. Create an interface between the module and the monolith code. The best solution is a bidirectional API, because the monolith will need data from the module and the module will need data from the monolith. This process is not easy, you will probably have to change code from the monolithic application in order to make the API work.
2. Once the coarse-grained interface is implemented, convert the module into an isolated microservice.

For example, imagine that the `POST` module is the candidate to be extracted, their components are used by the `Users` and `Comments` modules. As the first step says, it is necessary to implement a coarse-grained API. The first interface is an entry API used by `Users` to invoke the `POST` module and the second one is used by `POST` in order to invoke `Comments`.

In the second step of the extraction, convert the module to an isolated microservice. Once this is done, the resulting microservice will be scalable and independent, so it will be possible to make it grow or even to write it from scratch.

Little by little, the monolith will be smaller and your application will have more microservices.

Tutorial: From monolithic to microservices

In this chapter's examples, we will not use a framework and the code will be written without using a MVC architecture in order to focus on the subject of this chapter and learn how to transform a monolithic application into microservices.

There is no better way to learn something than by practicing, so let's look at an entire example of a blog platform that we defined in the previous section.

The blog platform example can be downloaded from our PHP microservices repository, so if you want to follow our steps, it is possible to do so by downloading it and following this guide.

Our example is a basic blog platform with the minimum functionalities to go through this tutorial. It is a blog system that allows the following things:

- Registering new users
- Logging in users
- Admins can post new articles
- Registered users can post new comments
- Admins can create new categories
- Admins can create new articles
- Admins can manage the comments
- All the users can see the articles

So, the first step in transforming a monolithic application into microservices is to familiarize yourself with the current application. In our imaginary schema, the current application can be divided into the following microservices:

- Users
- Articles
- Comments
- Categories

It is pretty clear in this example, but in a real example it should be studied deeply in order to divide the project into little microservices that will do specific functions by following the priorities we explained earlier in the chapter.

Stop diving

Now that we know how to follow the strategy that we explained before, imagine that we want to add a new functionality to send private messages between users in our blog platform.

To figure this out, we need to know which functionalities will have the new sending private messages feature in order to find where the glue code and the request to get information (routes) from the new microservice should be.

So, the functionalities of the new microservices can be as follows:

- Sending a message to a user
- Reading your messages

As you can see, these functionalities are very basic, but remember that this is only to familiarize yourself with the process of creating a new microservice in a monolithic application.

We will create the private messages microservices and, of course, we will use Lumen again. To quickly create the skeleton, run the following command on your terminal:

```
composer create-project --prefer-dist laravel/lumen private_messages
```

The preceding command will create a folder with Lumen installed.

In the `Chapter 2`, *Development Environment*, we explained how to create Docker containers. Now, you have the chance to use everything you have learned and implement the monolithic and the different new microservices in the Docker environment. Based on the previous chapters, you should be able to do this on your own.

Our new feature needs a place to store the private messages, so we will now create the table to be used by the private messages microservices. This can be created in a separate database or even in the same application's database. Remember that microservices can share the same database if the situation allows it, but imagine that this microservice will have a lot of traffic, so it is a better solution for us to have it in a separate database.

Create a new database or connect with the application database and execute the following query:

```
CREATE TABLE `messages` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `sender_id` INT NULL,
  `recipient_id` INT NULL,
  `message` TEXT NULL,
  PRIMARY KEY (`id`));
```

Once we have created the table, it is necessary to connect the new microservice to it, so open the `.env.example` file and modify the following lines:

```
DB_CONNECTION=mysql
DB_HOST=localhost
DB_PORT=3306
DB_DATABASE=private_messages
DB_USERNAME=root
DB_PASSWORD=root
```

If your database is different, change it in the preceding code.

Rename the `.env.example` file to `.env` and change the `$app->run();` code to the following one on the `public/index.php` file; this will allow you to make calls to this microservice:

```
$app->run(
  $app->make('request')
);
```

Now, you can check that your microservice is working properly on Postman by making a GET call to `http://localhost/private_messages/public/`. Remember to make all the required changes to match your development infrastructure.

You will receive a 200 status code with the Lumen version installed.

In our microservice, we will need to include at least the following calls:

- GET `/messages/user/id`: This is required to get the messages that a user has
- POST `/message/sender/id/recipient/id`: This is required to send a message to a user

So, now we will create the routes on `/private_messages/app/Http/routes.php` by adding the following lines at the end of the `routes.php` file:

```
$app->get('messages/user/{userId}',
          'MessageController@getUserMessages');
$app->post('messages/sender/{senderId}/recipient/{recipientId}',
           'MessageController@sendMessage');
```

The next step is to create a controller, called `MessageController`, on `/app/Http/Controllers/MessageController.php` with the following content:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class MessageController extends Controller
{
  public function getUserMessages(Request $request, $userId) {
    // getUserMessages code
  }

  public function sendMessage(Request $request, $senderId,
                              $recipientId) {
    // sendMessage code
  }
}
```

Now, we have to tell Lumen that it is necessary to use the database, so uncomment the following lines on `/bootstrap/app.php`:

```
$app->withFacades();
$app->withEloquent();
```

Now, we can create both the functionalities:

```
public function getUserMessages(Request $request, $userId)
{
  $messages = Message::where('recipient_id',$userId)->get();
  return response()->json($messages);
}

public function sendMessage(Request $request, $senderId,
                            $recipientId)
{
  $message = new Message();

  $message->fill([
    'sender_id'    => $senderId,
    'recipient_id' => $recipientId,
    'message'      => $request->input('message')]);

  $message->save();
  return response()->json([]);
}
```

Once our methods are complete, the microservice is finished. So, now we have to connect the monolithic application to the private messages microservice.

We need to create a new button for the registered users on the `header.php` file of the monolithic application:

```
<?php if (empty($arrUser['username'])) : ?>
  <li role="presentation"><a href="login.php">Log in</a></li>
  <li role="presentation"><a href="signup.php">Sign up</a></li>
<?php else : ?>
  <?php if ($arrUser['type'] === 'admin') : ?>
    <li role="presentation">
      <a href="admin/index.php">Admin Panel</a>
    </li>
  <?php endif; ?>
  <li role="presentation">
    <a href="messages.php">Messages</a>
  </li>
  <li role="presentation">
    <a href="index.php?logout=true">Log out</a>
  </li>
<?php endif; ?>
```

Then, we need to create a new file, called `messages.php`, in the `root` folder with the following code:

```
<?
  include_once 'libraries.php';

  $url = "http://localhost/private_messages/public/messages
          /user/".$arrUser['id'];
  $ch = curl_init();
  curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, false);
  curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
  curl_setopt($ch, CURLOPT_URL,$url);
  $result=curl_exec($ch);
  curl_close($ch);
  $messages = json_decode($result, true);
```

As you can see, we are making a `curl` call to the microservice in order to get the user message list. Also, we need to get the user list to fill the user selector for sending messages. This piece of code can be considered as glue code because it is necessary to match the microservice data with the monolith data. We will use the following glue code:

```
$arrUsers = array();
$query = "SELECT id, username FROM `users` ORDER BY username ASC";
$result = mysql_query ($query, $dbConn);
while ( $row = mysql_fetch_assoc ($result)) {
  array_push( $arrUsers,$row );
}
```

Now we can build the html code to display the user messages and the necessary form for sending messages:

```
  include_once 'header.php';
?>
<p class="bg-success">
  <?php if ($_GET['sent']) { ?>
    The message was sent!
  <?php } ?>
</p>
<h1>Messages</h1>
<?php foreach($messages as $message) { ?>
  <div class="panel panel-primary">
    <div class="panel-heading">
      <h3 class="panel-title">
        Message from <?php echo $arrUsers[$message['sender_id']]
                       ['username'];?>
      </h3>
    </div>
```

```
      <div class="panel-body">
        <?php echo $message['message']; ?>
      </div>
    </div>
  <?php } ?>
  <h1>Send message</h1>
    <div>
      <form action="messages.php" method="post">
        <div class="form-group">
          <label for="category_id">Recipient</label>
          <select class="form-control" name="recipient">
            <option value="">Select User</option>
            <option value="">-----------------------</option>
            <?php foreach($arrUsers as $user) { ?>
              <option value="<?php echo $user['id']; ?>">
                <?php echo $user['username']; ?>
              </option>
            <?php } ?>
          </select>
        </div>
        <div class="form-group">
          <label for="name">
            Message
          </label>
          <br />
          <input name="message" type="text" value=""
           class="form-control" />
        </div>
        <input name="sender" type="hidden"
         value="<?php echo $arrUser['id']; ?>" />

        <div class="form-group">
          <input name="submit" type="submit" value="Send message"
           class="btn btn-primary" />
        </div>

      </form>
    </div>
  <?php include_once 'footer.php'; ?>
```

Note that there is a form for sending messages, so we have to add some code to make a call to the microservice in order to send the message. Add the following code after the `$messages = json_decode($result, true);` line:

```
  if (!empty($_POST['submit'])) {
    if (!empty($_POST['sender'])) {
      $sender = $_POST['sender'];
    }
```

```
    if (!empty($_POST['recipient'])) {
      $recipient = $_POST['recipient'];
    }
    if (!empty($_POST['message'])) {
      $message = $_POST['message'];
    }

    if (empty($sender)) {
      $error['sender'] = 'Sender not found';
    }
    if (empty($recipient)) {
      $error['recipient'] = 'Please select a recipient';
    }
    if (empty($message)) {
      $error['message'] = 'Please complete the message';
    }

    if (empty($error)) {
      $url = 'http://localhost/private_messages/public/messages
              /sender/'.$sender.'/recipient/'.$recipient;

      $handler = curl_init();
      curl_setopt($handler, CURLOPT_URL, $url);
      curl_setopt($handler, CURLOPT_POST,true);
      curl_setopt($handler, CURLOPT_RETURNTRANSFER,true);
      curl_setopt($handler, CURLOPT_POSTFIELDS, "message=".$message);
      $response = curl_exec ($handler);

      curl_close($handler);

      header( 'Location: messages.php?sent=true' );
      die;
    }
  }
```

That's it. We have our first microservice included in the monolithic application. This is how to proceed when we have to add a new functionality in a current monolithic application.

Divide frontend and backend

The second strategy, as we said before, consists of isolating the presentation layer from the business logic. This can be done by creating an entire microservice that includes all the business logic and data access or simply by isolating the presentation layer from the business layer, like a **Model-View-Controller** (**MVC**) structure does.

This is not a complete solution for the problem of using monolithic applications because it results in us having two monolithic applications instead of one.

To do this, we should start by creating a new `Controller.php` file in the `root` folder. We can call this class `Controller` and it will contain all the methods that the views need. For example, the `Article` view needs `getArticle`, `postComment`, and `getArticleComments`:

```
<?php

class Controller
{
  public function connect () {
    $db_con = mysql_pconnect(DB_SERVER,DB_USER,DB_PASS);
    if (!$db_con) {
      return false;
    }
    if (!mysql_select_db(DB_NAME, $db_con)) {
      return false;
    }
    return $db_con;
  }

  public function getArticle($id)
  {
    $dbConn = $this->connect();

    $query = "SELECT articles.id, articles.title, articles.extract,
                     articles.text, articles.updated_at,
                     categories.value as category,
                     users.username FROM `articles`
              INNER JOIN
              `categories` ON categories.id = articles.category_id
              INNER JOIN
              `users` ON users.id = articles.user_id
              WHERE articles.id = " . $id . " LIMIT 1";
    $result = mysql_query ($query, $dbConn);
    return mysql_fetch_assoc ($result);
  }

  public function getArticleComments($id)
  {
    $dbConn = $this->connect();

    $arrComments = array();
```

```
    $query = "SELECT comments.id, comments.comment, users.username
              FROM `comments` INNER JOIN `users`
              ON comments.user_id = users.id
              WHERE comments.status = 'valid'
              AND comments.article_id = " . $id . "
              ORDER BY comments.id DESC";

    $result = mysql_query ($query, $dbConn);
    while ($row = mysql_fetch_assoc($result)) {
      array_push($arrComments,$row);
    }

    return $arrComments;
  }

  public function postComment($comment,$user_id,$article_id)
  {
    $dbConn = $this->connect();

    $query = "INSERT INTO `comments` (comment, user_id, article_id)
              VALUES ('$comment','$user_id','$article_id')";
    mysql_query($query, $dbConn);
  }
}
```

The article view should include the methods included in the `Controller.php` file. Take a look at the following code:

```
<?
  include_once 'libraries.php';
  include_once 'Controller.php';

  $controller = new Controller();

  if ( !empty($_POST['submit']) ) {

    // Validation
    if (!empty($_POST['comment'])) {
      $comment = $_POST['comment'];
    }
    if (!empty($_GET['id'])) {
      $article_id = $_GET['id'];
    }
    if (!empty($arrUser['id'])) {
      $user_id = $arrUser['id'];
    }

    if (empty($comment)) {
```

```
        $error['comment'] = true;
      }
      if (empty($article_id)) {
        $error['article_id'] = true;
      }
      if (empty($user_id)) {
        $error['user_id'] = true;
      }
      if ( empty($error) ) {

        $controller->postComment($comment,$user_id,$article_id);
        header ( 'Location: article.php?id='.$article_id);
        die;
      }
    }

    $article = $controller->getArticle($_GET['id']);
    $comments = $controller->getArticleComments($_GET['id']);

    include_once 'header.php';
?>

<h1>Article</h1>

<div class="panel panel-primary">
  <div class="panel-heading">
    <h3 class="panel-title">
      <?php echo $article['title']; ?>
    </h3>
  </div>
  <div class="panel-body">
    <span class="label label-primary">Published</span> by
    <b><?php echo $article['username']; ?></b> in
    <i><?php echo $article['category']; ?></i>
    <b><?php echo date_format(date_create($article
                                  ['fModificacion']),
                            'd/m/y h:m'); ?>
    </b>
    <hr/>
    <?php echo $article['text']; ?>
  </div>
</div>

<h2>Comments</h2>
<?php foreach ($comments as $comment) { ?>

  <div class="panel panel-warning">
    <div class="panel-heading">
```

```
        <h3 class="panel-title"><?php echo $comment['username']; ?>
                                said</h3>
      </div>
      <div class="panel-body">
        <?php echo $comment['comment']; ?>
      </div>
    </div>
  <?php } ?>

  <div>
    <?php if ( !empty( $arrUser ) ) { ?>

      <form action="article.php?id=<?php echo $_GET['id']; ?>"
            method="post">
        <div class="form-group">
          <label for="user">Post a comment</label>
          <textarea class="form-control" rows="3" cols="50"
                    name="comment" id="comment">
          </textarea>
        </div>

        <div class="form-group">
          <input name="submit" type="submit" value="Send"
                 class="btn btn-primary" />
        </div>
      </form>

    <?php } else { ?>
      <p>
        Please sign up to leave a comment on this article.
        <a href="signup.php">Sign up</a> or
        <a href="login.php">Log in</a>
      </p>
    <?php } ?>
  </div>

  <?php include_once 'footer.php'; ?>
```

These are the steps that we should follow in order to isolate the business logic layer from the views.

If you want, you can put all the view files (`header.php`, `footer.php`, `index.php`, `article.php`, and so on) into a folder called `views` in order to have them organized in the same folder.

Once we have all the views isolated from the business logic, we will have all the methods included in the controller instead of having them in the presentation layer. As we said before, this is only a temporary solution, so we will look at the real solution in order to extract the modules into microservices.

Extraction services

In this strategy, we have to select the first module that we want to isolate in order to make a microservice from it. In this case, we will start doing it on the `Categories` module.

The categories are most used on the admin panel. It is possible to create, modify, and delete categories from it and then, it is possible to select them when creating a new article and they are displayed in the articles to indicate the article category.

The extraction process is not easy; we have to ensure that we are aware of all the places the module is being used at. To do this, we will create a bidirectional API or create all the category methods in the controller and then, we can isolate them in a microservice.

Open the `admin/categories.php` file, we have to do the same as we did with the divide frontend and backend strategy–find all the places where the categories are referenced and create a new method on the controller. Take a look at this:

```
<?

  session_start ();

  require_once 'config.php';
  require_once 'connection.php';
  require_once 'isUser.php';

  include_once '../Controller.php';

  $controller = new Controller();
  $dbConn = connect();

  /* Omitted code */

  if (!empty($_GET['del'])) {
    $controller->deleteCategory($_GET['del']);

    header( 'Location: categories.php?dele=true' );
    die;
  }
```

```
    if (!empty($_POST['submit'])) {
      if (!empty($_POST['name'])) {
        $name = $_POST['name'];
      }
      if (empty($name)) {
        $error['name'] = 'Please enter category name';
      }
      if (empty($error)) {
        $controller->createCategory($name);
        header( 'Location: categories.php?add=true' );
        die;
      }
    }

    if (!empty($_POST['submitEdit'])) {

      /* Ommited code */

      if (empty($error)) {
        $controller->updateCategory($id, $name);
        header( 'Location: categories.php?edit=true' );
        die;
      }
    }

    $arrCategories = $controller->getCategories();

    if (!empty($_GET['id'])) {
      $row = $controller->getCategory($_GET['id']);
    }

    include_once 'header.php';

  ?>
  <!-- Omitted HTML code -->
```

The `controller.php` file has to contain the category methods:

```
  public function createCategory($name)
  {
    $dbConn = $this->connect();

    $query = "INSERT INTO `categories` (value) VALUES ('$name')";
    mysql_query($query, $dbConn);
  }

  public function updateCategory($id,$name)
  {
```

```
    $dbConn = $this->connect();

    $query = "UPDATE `categories` set value = '$name'
              WHERE id = $id";
    mysql_query($query, $dbConn);
  }

  public function deleteCategory($id)
  {
    $dbConn = $this->connect();

    $query = "DELETE FROM `categories` WHERE id = ".$id;
    mysql_query($query, $dbConn);
  }

  public function getCategories()
  {
    $dbConn = $this->connect();

    $arrCategories = array();

    $query = "SELECT id, value FROM `categories` ORDER BY value ASC";
    $resultado = mysql_query ($query, $dbConn);

    while ($row = mysql_fetch_assoc ($resultado)) {
      array_push( $arrCategories,$row );
    }

    return $arrCategories;
  }

  public function getCategory($id)
  {
    $dbConn = $this->connect();

    $query = "SELECT id, value FROM `categories` WHERE id = ".$id;
    $resultado = mysql_query ($query, $dbConn);
    return mysql_fetch_assoc ($resultado);
  }
```

There are more references to categories in the `admin/articles.php` file, so open it and add the following lines after the `require_once` lines:

```
include_once '../Controller.php';
$controller = new Controller();
```

These lines will allow you to use the category methods included in the `controller.php` file in the `articles.php` file. Modify the code used to get the categories to this one:

```
$arrCategories = $controller->getCategories();
```

Finally, it is necessary to make some changes on the article view. This is the view to display an article, and it contains the category selected when creating the article.

To get an article, the executed query is as follows:

```
SELECT articles.id, articles.title, articles.extract,
       articles.text, articles.updated_at,
       categories.value as category,
       users.username FROM `articles`
INNER JOIN `categories` ON categories.id = articles.category_id
INNER JOIN `users` ON users.id = articles.user_id
WHERE articles.id = " . $id . " LIMIT 1;
```

As you can see, the query requires the categories table. If you want to use a different database for the categories microservice, you will have to remove the highlighted line from the query, select `articles.category_id` in the query and then, get the category name with the method created to provide it. So, the query will look like this:

```
SELECT articles.id, articles.title, articles.extract,
       articles.text, articles.updated_at, articles.category_id,
       users.username FROM `articles`
INNER JOIN `users` ON users.id = articles.user_id
WHERE articles.id = " . $id . " LIMIT 1;
```

The following is the code to get the category name from the provided category ID:

```
public function getArticle($id)
{
  $dbConn = $this->connect();

  $query = "SELECT articles.id, articles.title, articles.extract,
                   articles.text, articles.updated_at,
                   articles.category_id,
                   users.username FROM `articles`
            INNER JOIN `users` ON users.id = articles.user_id
            WHERE articles.id = " . $id . " LIMIT 1;";
```

```
    $result = mysql_query ($query, $dbConn);

    $data = mysql_fetch_assoc($result);
    $data['category'] = $this->getCategory($data['category_id']);
    $data['category'] = $data['category']['value'];
    return $data;
}
```

Once we have made all of these changes, we are ready to isolate the category table in a different database, so we can create a categories microservice from the created methods in the `controller.php` file:

- `public function createCategory($name)`
- `public function updateCategory($id,$name)`
- `public function deleteCategory($id)`
- `public function getCategories()`
- `public function getCategory($id)`

As you would imagine, these functions are used to create the `routes.php` file of the categories microservice. So, let's create a new microservice like we did with the stop diving strategy.

Create the new categories microservice by executing the following command:

```
composer create-project --prefer-dist laravel/lumen categories
```

The preceding command will install Lumen in a folder called categories, so we can start creating the code for our new categories microservice.

Now we have two options–the first one is to use the same table located on the current database–we can point the new microservice to the current database. The second option is to create a new table in a new database, so the new microservice will use its own database.

If we want to create a new table in a new database, we can proceed as follows:

- Export the current categories table in a SQL file. It will keep the current data stored.
- Import the SQL file to the new database. It will create the exported table and data in the new database.

The export/import process can be performed using a SQL client or by executing `mysqldump` in console.

Once the new table is imported to a new database or you decide to use the current database, it is necessary to set up the `.env.example` file in order to connect the new microservice to the correct database, so open it and put the correct parameters on it:

```
DB_CONNECTION=mysql
DB_HOST=localhost
DB_PORT=3306
DB_DATABASE=categories
DB_USERNAME=root
DB_PASSWORD=root
```

Do not forget to rename the `.env.example` file to `.env` and change the `$app->run()` line on `public/index.php`, like we did earlier to the following code:

```
$app->run(
  $app->make('request')
);
```

Also, uncomment the following lines on `/bootstrap/app.php`:

```
$app->withFacades();
$app->withEloquent();
```

Now we are ready to add the necessary methods to the `routes.php` file. We have to add the category methods that we have on the `Controller.php` of the monolithic application and translate them to routes:

- `public function createCategory($name)`: This is a POST method for creating a new category. So, it can be something like `$app->post('category', 'CategoryController@createCategory');`.
- `public function updateCategory($id,$name)`: This is a PUT method to edit an existing category. So, it can be something like `app->put('category/{id}', 'CategoryController@updateCategory');`.
- `public function deleteCategory($id)`: This is a DELETE method for deleting an existing category. So, it can be something like `app->delete('category/{id}', 'CategoryController@deleteCategory');`.

- `public function getCategories()`: This is a GET method for getting all the existing categories. So, it can be something like `app->get('categories', 'CategoryController@getCategories');`.
- `public function getCategory($id)`: This is a GET method too, but this only gets a single category. So, it can be something like `app->get('category/{id}', 'CategoryController@getCategory');`.

So, once we have all our routes added to the `routes.php` file, it is time to create the category model. To do this, create a new folder on `/app/Model` and a file on `/app/Model/Category.php`, like this one:

```
<?php

  namespace App\Model;

  use Illuminate\Database\Eloquent\Model;

  class Category extends Model {
    protected $table = 'categories';
    protected $fillable = ['value'];
    public $timestamps = false;
  }
```

Once we have created the model, create a `/app/Http/Controllers/CategoryController.php` file with the necessary methods:

```
<?php

  namespace App\Http\Controllers;

  use App\Model\Category;
  use Illuminate\Http\Request;

  class CategoryController extends Controller
  {
    public function createCategory(Request $request) {
      $category = new Category();
      $category->fill(['value' => $request->input('value')]);
      $category->save();

      return response()->json([]);
    }

    public function updateCategory(Request $request, $id) {
      $category = Category::find($id);
      $category->value = $request->input('value');
```

```
    $category->save();

    return response()->json([]);
  }

  public function deleteCategory(Request $request, $id) {
    $category = Category::find($id);
    $category->delete();

    return response()->json([]);
  }

  public function getCategories(Request $request) {
    $categories = Category::get();

    return $categories;
  }

  public function getCategory(Request $request, $id) {
    $category = Category::find($id);

    return $category;
  }
}
```

Now, we have finished our categories microservice. You can give it a try on Postman in order to check that all the methods work. For example, the `getCategories` method can be called by Postman with the `http://localhost/categories/public/categories` URL.

Once we have the new categories microservice created and working properly, it is time to disconnect the categories module and connect the monolithic application to the microservice.

Go back to the monolithic application and find all the references to the category methods. We have to replace them by making calls to the new microservice. We will make these calls using native curl calls, but you should consider using Guzzle or a similar package instead, as we did in the previous chapters.

To do this, firstly we should create a function to make the calls in the `Controller.php` file. It can be something like this:

```
function call($url, $method, $field = null)
{
  $ch = curl_init();
  if(($method == 'DELETE') || ($method == 'PUT')) {
    curl_setopt($ch, CURLOPT_CUSTOMREQUEST, $method);
  }
```

```
    curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, false);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    curl_setopt($ch, CURLOPT_URL,$url);
    if ($method == 'POST') {
      curl_setopt($ch, CURLOPT_POST,true);
    }
    if($field) {
      curl_setopt($ch, CURLOPT_POSTFIELDS, 'value='.$field);
    }
    $result=curl_exec($ch);
    curl_close($ch);
    return json_decode($result, true);
  }
```

The preceding code will be used in order to reuse code in every single call to the categories microservice.

Go to the `/admin/categories.php` file and replace the `controller->createCategory($name);` line with the following code:

```
$controller->call('http://localhost/categories/public/category',
                  'POST', $name);
```

In the preceding code, you can check that we are making a POST call to the create category method with the value parameter set to the `$name` variable.

In the `/admin/categories.php` file, find and replace the `controller->updateCategory($id, $name);` line with the following code:

```
$controller->call('http://localhost/categories/public/category/'.$id,
                  'PUT', $name);
```

In the same file, find and replace `$controller->deleteCategory($_GET['del']);` with the following code:

```
$controller->call('http://localhost/categories/public
                  /category/'.$_GET['del'], 'DELETE');
```

In the same file again and also in the `/admin/articles.php` file, find and replace `$arrCategories = $controller->getCategories();` with the following code:

```
$arrCategories = $controller->call('http://localhost/categories
                                   /public/categories', 'GET');
```

The last one is located in the `/admin/categories.php` file again. Find and replace the `row = $controller->getCategory($_GET['id']);` line with the following code:

```
$row = $controller->call('http://localhost/categories
                         /public/category/'.$_GET['id'], 'GET');
```

Once we have finished replacing all the category methods on the monolithic application with calls to the new categories microservice, we can delete all the references to the monolithic category module.

Go to the `Controller.php` file and delete the following functions, you do not need them any more because they reference to the monolithic category module:

- `public function deleteCategory($id)`
- `public function createCategory($name)`
- `public function updateCategory($id,$name)`
- `public function getCategories()`
- `public function getCategory($id)`

And finally, if you created a new database for the categories microservice, you can drop the categories table located in the monolithic application database by executing the following query:

```
DROP TABLE categories;
```

We have finished extracting the categories service from the monolithic application. The next step would be to select another module, follow the same steps again, and repeat this process until the monolithic application disappears or becomes a microservice too.

In `Chapter 7`, *Security* we talked about security in microservices. To practice what you have learned, review all the code in this chapter and find the weakness in our examples.

Summary

In this chapter, you learned the strategies to follow in order to transform a monolithic application into microservices using sample codes for every single step. From now on, you are ready to say goodbye to monolithic applications and transform them in order to start working with microservices.

10
Strategies for Scalability

You have your application ready. Now it is time to plan for the future. In this chapter, we will give you a global overview of how you can check the possible bottlenecks of your application and how you can calculate the capacity of your application. At the end of the chapter, you will have the basic knowledge to create your own scalability plan.

Capacity planning

Capacity planning is the process of determining the infrastructure resources required by your application to meet future workload demands for your app. This process ensures that you have adequate resources available only when they are needed, reducing costs to the minimum. If you know how your application is used and the limits of your current resources, you can extrapolate the data and know, more or less, the future requirements. Creating a capacity plan for your application has some benefits, among which we can highlight the following ones:

- **Minimize costs and avoid waste from over-provisioning**: Your application will use only the required resources, so it makes no sense, for example, to have a 64 GB RAM server for your database when you are only using 8 GB.
- **Prevent bottlenecks and save time**: Your capacity plan highlights when each element of your infrastructure reaches its peak, giving you a hint about where the bottleneck can be.
- **Increase business productivity**: If you have a detailed plan that indicates the limits of each element of your infrastructure and knows when each element will reach its limits, it gives you spare room to dedicate your time to other business tasks. You will have a set of instructions to follow on the precise moment you need to increase the capacity of your application. No more crazy moments when you are suffering bottlenecks and don't know what to do.

- **Used as a mapping of business objectives**: If your application is critical for your business, this document can be used to highlight some business objectives. For example, if the business wants to reach 1,000 users, your infrastructure needs to be allowed to support them, flagging some investments needed to fulfill this requirement.

Knowing the limits of your application

The main purpose of knowing the limits of your application is to know how much capacity we still have at any given point of time before we start having issues. The first thing we need to do is create an inventory of the components of our application. Make the inventory as detailed as possible; it will help you know all the tools you have in your project. In our example application, the list of different components could be something similar to this:

- Autodiscovery service:
    - Hashicorp Consul
- Telemetry service:
    - Prometheus
- Battle microservice:
    - Proxy: NGINX
    - App engine: PHP 7 FPM
    - Data storage: Percona
    - Cache storage: Redis
- Location microservice:
    - Proxy: NGINX
    - App engine: PHP 7 FPM
    - Data storage: Percona
    - Cache storage: Redis
- Secret microservice:
    - Proxy: NGINX
    - App engine: PHP 7 FPM
    - Data storage: Percona
    - Cache storage: Redis
- User microservice:
    - Proxy: NGINX
    - App engine: PHP 7 FPM

- Data storage: Percona
- Cache storage: Redis

Once we have reduced our application to the base components, we need to analyze and determine the usage of each component over time and its maximum capacity in an appropriate measurement.

Some components can have multiple measurements associated, for example, the data storage layer (Percona in our case). For this component, we can measure the number of SQL transactions, amount of storage used, CPU load, and so on. In the previous chapters, we added a Telemetry service; you can use this service to gather basic stats from each component.

Some of the basic stats you can log for each component of your application are as listed:

- CPU load
- Memory usage
- Network usage
- IOPS
- Disk utilization

In some software, you need to gather some specific measurements. For instance, on a database you can check the following things:

- Transactions per second
- Cache hit ratio (if you have enabled query cache)
- User connections

The next step is determining the natural growth of the application. This measurement can be defined as the growth performed by your application if nothing special has been done (such as PPC campaigns and new features). This measurement can be the number of new users or the amount of active users, for example. Imagine that you deploy your application to production and stop adding new features or doing marketing campaigns. If the number of new users increased 7% over the last month, that amount is the natural growth of your application.

Some businesses/projects have seasonal trends, which means that at specific dates, the usage of your application increases. Imagine that you are a gifts' retailer, most of your sales probably are done around Valentine’s day or at the end of the year (Black Friday, Xmas). If this is your case, analyze all the data you have to establish the seasonality data.

Now that you have some basic stats of your application, it is time to calculate what is known as the headroom. The headroom can be defined as the amount of resources you have until you are out of resources. It can be calculated with the following formula:

$$Headroom = (IdealUsage * MaxCapacity) - CurrentUsage - \sum_{t=1}^{12} (Growth(t) - Optimizations(t))$$

Headroom formula

As you can see from the preceding formula, calculating the headroom for a specific component is very easy. Let's explain each variable before we take an example:

- **IdealUsage:** This is a percentage that describes the amount of capacity for a specific component of your application that we are planning to use. This ideal usage should never be 100%, as strange behaviors, such as not being able to save data in your database, can start appearing when you approach the resource limits. Our recommendation is to set this amount to be between 60% and 75%, giving you enough extra space for peak moments.
- **MaxCapacity:** This is the amount that indicates the maximum capacity of the component subject of our study. For example, a web server that can manage up to 250 concurrent connections.
- **CurrentUsage:** This is the amount that indicates the current usage of the component that we are studying.
- **Growth:** This is the percentage that indicates the natural growth of our application.
- **Optimizations:** This is the optional variable that describes the amount of optimizations we can achieve in a specific amount of time. For instance, if your current database can manage 35 queries per second, you can achieve 50 queries per second after a few optimizations. In this case, the amount of optimization is 15.

Imagine that you are calculating the headroom of requests per second that can be managed by one of our NGINX. For our application, we have decided to set the ideal usage to 60% (0.6). From our measurements and from the data extracted from our load testing (explained later in the chapter), we know that the maximum number of **requests per second** (**RPS**) is 215. In our current stats, our NGINX server served a peak of 193 RPS today and we have calculated the growth for the next year to be at least 11 RPS.

The time period we want to measure is 1 year, and we think that we can achieve the 250 RPS of maximum capacity in this time, so our Headroom value will be as follows:

*Headroom = 0.6 * 215 – 123 – (11 – 35) = 30 RPS*

What does this calculation mean? Since the result is positive, it means that we have enough spare room for our predictions. If we divide the result by the sum of growth and optimizations, we can calculate how much time we have until we reach our limits.

Since our time period is 1 year, we can calculate how much time we have until we reach our limits, as follows:

Headroom Time = 30 rpms / 24 = 1.25 years

As you have probably already deduced, we have 1.25 years until our NGINX server reaches the limit of RPS. In this section, we showed you how to calculate the headroom of a specific component; now, it is your turn to make the calculations for each one of your components and for the different metrics available for each component.

Availability math

Availability can be defined as how often the site is available in a specific period of time, for example, a week, a day, a year, and so on. Depending on how critical your application is for you or your business, a downtime can equal lost revenue. As you can suppose, the availability can become the most important metric on scenarios where your application is used by customers/users and they need your service at any time.

We have the theoretical concept about what availability is. It is time to do some math, so take your calculator. As from the earlier general definition, the availability can be calculated as the amount of time your application can be used by your users/customers for divided by the time frame (the specific period of time we are measuring).

Let's imagine that we want to measure the availability of our application over one week. In one week, we have `10,080` minutes:

*7 days x 24 hours per day x 60 minutes per hour = 7 * 24 * 60 =* **10,080 minutes**

Now, imagine that your application had a few outages that week and the amount of available minutes of your application was reduced to `10,000`. To calculate the availability for our example, we only need to do some simple math:

10,000 / 10,080 = 0.9920634921

The availability was usually measured as a percentage (%), so we need to transform our result to a percentage:

*0.9920634921 * 100 = 99.20634921% ~* **99.21**

The availability of our application over the week was `99.21%`. Not too bad, but far from the result we aim for, that is, the closest percentage to `100%` that we can get. Most of the time, the availability percentage is referred as the **number of nines** and, the closer they are to `100%`, the more difficult it is to maintain the availability of your application. To give you an overview of how difficult it will be to reach the `100%` availability, here are some examples of availabilities and the possible downtime:

- 99.21% (our example):
    - Weekly: 1 h 19 m 37.9 s
    - Monthly: 5 h 46 m 15.0 s
    - Yearly: 69 h 14 m 59.9 s
- 99.5%:
    - Weekly: 50 m 24.0 s
    - Monthly: 3 h 39 m 8.7 s
    - Yearly: 43 h 49 m 44.8 s
- 99.9%:
    - Weekly: 10 m 4.8 s
    - Monthly: 43 m 49.7 s
    - Yearly: 8 h 45 m 57.0 s
- 99.99%:
    - Weekly: 1 m 0.5 s
    - Monthly: 4 m 23.0 s
    - Yearly: 52 m 35.7 s

- 99.999%:
    - Weekly: 6.0 s
    - Monthly: 26.3 s
    - Yearly: 5m 15.6 s
- 99.9999%:
    - Weekly: 0.6 s
    - Monthly: 2.6 s
    - Yearly: 31.6 s
- 99.99999%:
    - Weekly: 0.1 s
    - Monthly: 0.3 s
    - Yearly: 3.2 s

As you can see, as getting close to `100%` availability becomes harder and harder, the downtimes are tighter. However, how can you reduce your downtime or at least ensure that you are doing your best to keep it low? There is no easy answer to this question, but we can give you a few hints of the different things you can do:

- The worst possible scenario will happen, so you should simulate failures frequently to be ready to deal with the apocalypse of your application.
- Find the possible bottlenecks of your application.
- Tests, tests, and more tests everywhere, and, of course, keep them updated.
- Log any event, any metric, anything you can measure or save as a log, and keep it for future references.
- Know the limits of your application.
- Have some good development practices in place, at least to share the knowledge of how your application was built. Among all of them, you can do the following ones:
    - Second approval for any hotfix or feature
    - Programming in pairs

- Create a continuous delivery pipeline.
- Have a backup plan and keep your backups safe and ready to be used.
- Document everything, any small change or design, and always keep the documentation up to date.

You now have a global overview about what **availability** means and the maximum downtime expected for each rate. Be careful if you give your users/customers a SLA (Service Level Agreement), as you will be creating a promise about the availability of your application that you will have to fulfill.

Load testing

Load testing can be defined as the process of putting a demand (load) in your application to measure its response. This process helps you identify the maximum capacity of your application or infrastructure and it can highlight bottlenecks or problematic elements of your application or infrastructure. The normal way of doing load testing is first doing a test on “normal” conditions, that is, with a normal load in your application. Having measured the response of your system under normal conditions allows you to have a baseline that you will use to compare with in future testings.

Let’s see some of the most common tools you can use for your load testing. Some are simple and easy to use and others are more complex and powerful.

Apache JMeter

The Apache JMeter application is an open source software built in Java and designed to do load testing and measure performance. At first, it was designed for web applications, but it was expanded to test other functions in time.

Some of the most interesting features of Apache JMeter are as follows:

- Support for different applications/servers/protocols: HTTP(S), SOAP/Rest, FTP, LDAP, TCP, and Java objects.
- Easy integration with third-party Continuous Integration tools: It has libraries for Maven, Gradle, and Jenkins.
- Command-line mode (non GUI/headless mode): This enables you to do your test from any OS with Java installed.

- Multi-threading framework: This allows you to make concurrent samples by many threads and simultaneous sampling of different functions by separate thread groups.
- Highly extensible: It is extensible through libraries or plugins, among others.
- Full featured test IDE: It allows you to create, record, and debug your test plans.

As you can see, this project is an interesting tool you can use in your loading tests. In the following sections, we will show you how to build a simple test scenario. Unfortunately, we don't have enough space in the book to cover all the features, but at least you will know the basics that will be the foundations of more complex tests in the future.

Installing Apache JMeter

Being developed in Java allows this application to be portable to any OS with Java installed. Let's install it in our development machine.

The first step is to fulfill the main requirement–you need a JVM 6 or later version to make the application work. You probably have Java in your machine already, but if this is not the case, you can download the latest JDK from the Oracle page.

To check the version of your Java runtime, you only need to open a terminal in your OS and execute the following command:

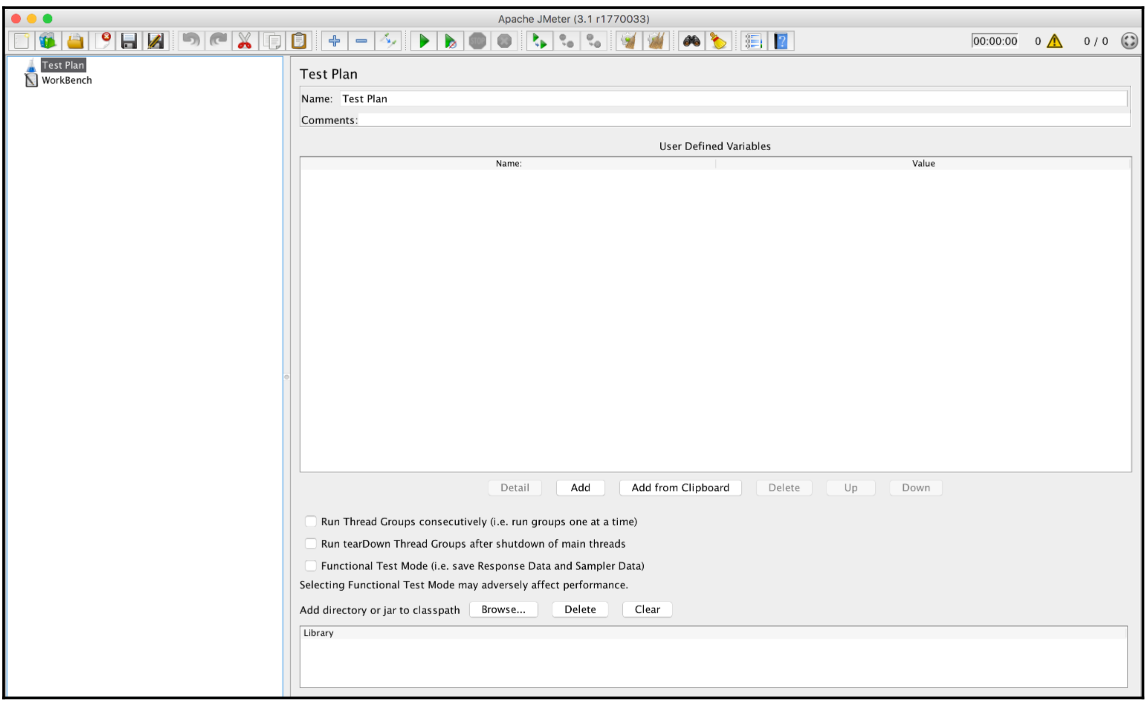
```
java -version
```

The preceding command will tell you the version available in your machine.

As soon as we are sure that we have the correct version, we only need to go to the official Apache JMeter page (`http://jmeter.apache.org`) and download the latest binaries in ZIP or TGZ formats. Once the binaries are fully downloaded to your machine, you only need to uncompress the downloaded ZIP or TGZ, and Apache JMeter is ready to be used.

Executing load tests with Apache JMeter

Open the folder where you have uncompressed the Apache JMeter binaries. There, you can find a `bin` folder and some scripts for different OSes in it. If you are using Linux/UNIX or Mac OS, you can execute the `jmeter.sh` script to open the application GUI. If you are running Windows, there is a `jmeter.bat` executable that you can use to open the GUI:



Apache JMeter GUI

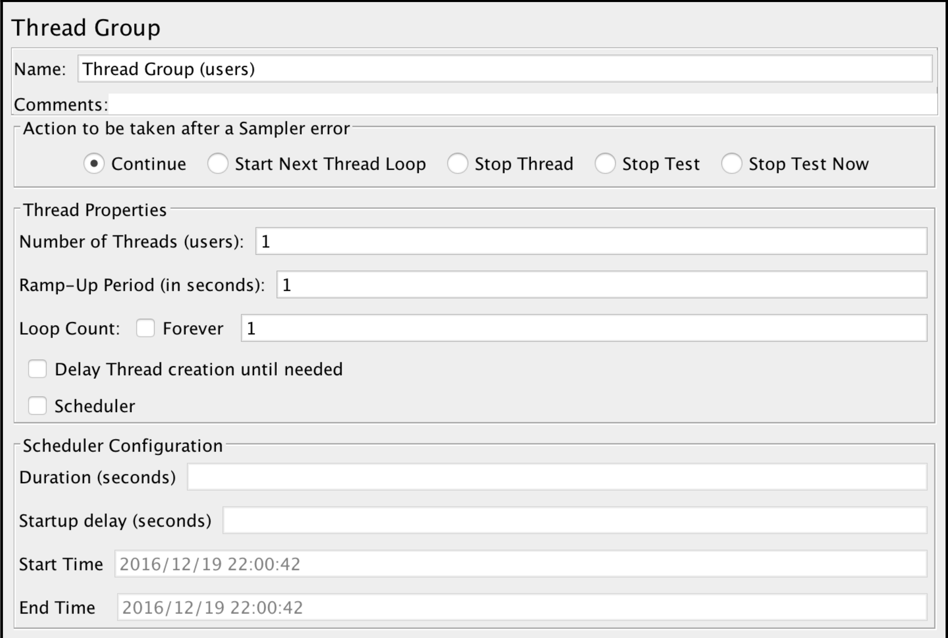
The Apache JMeter GUI allows you to build your different testing plans and, as you can see in the preceding screenshot, the interface is very easy to understand even without reading the manual. Let's build a test plan with the GUI.

A test plan can be described as a series of steps that Apache JMeter will run in a specific order.

Our first step in order to create our test plan is adding a **Thread Group** under the **Test Plan** node. In Apache JMeter, a thread group can be defined as a simulation of concurrent users. Follow the given steps to create a new group:

1. Right-click on the **Test Plan** node.
2. In the context menu, select **Add** | **Threads (Users)** | **Thread Group**.

The preceding steps will create a child element in our **Test Plan** node. Select it so that we can make some adjustments to our group. Refer to the following screenshot:



Thread group settings

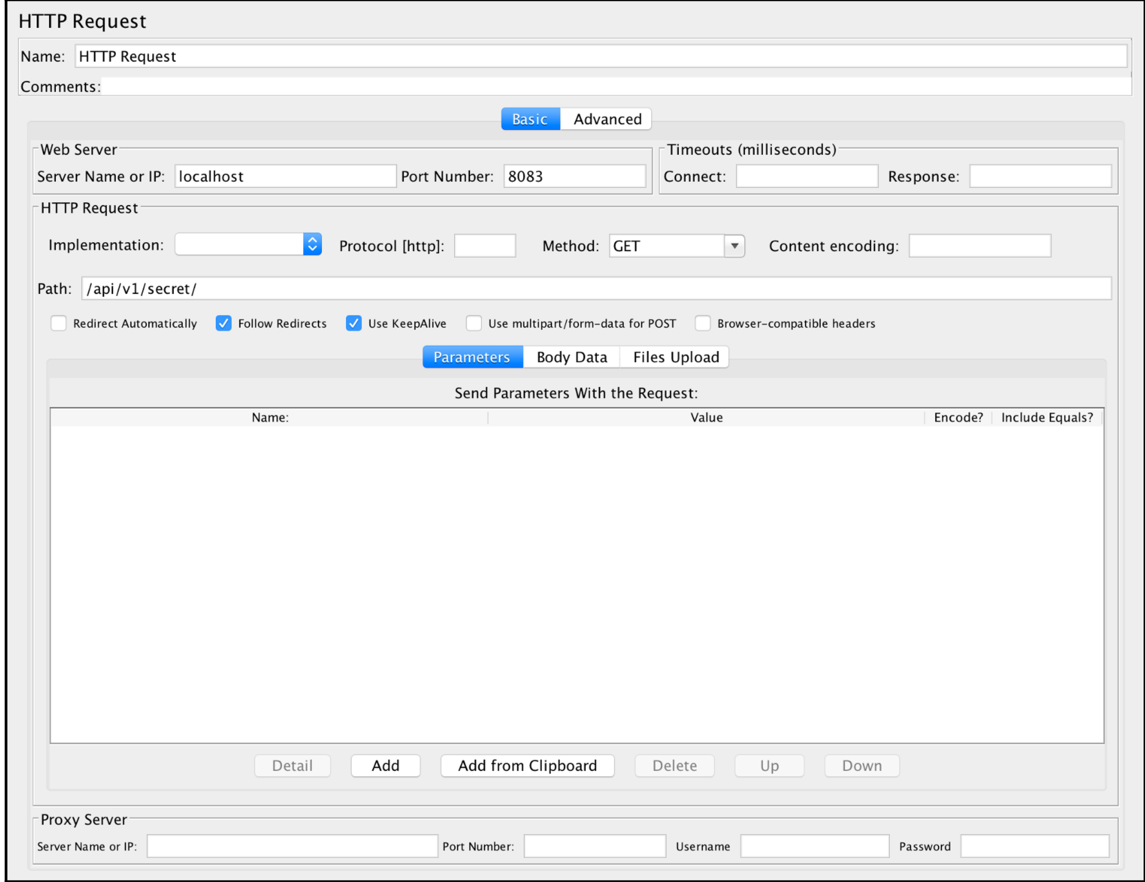
As you can see in the preceding screenshot, each thread group allows you to specify the amount of users for your tests and the duration of the test. The main options available are as listed:

- **Actions to be taken after a Sample error:** This option allows you to control the behavior of the test as soon as a sample error is thrown. The most used option is the **Continue** behavior.
- **Number of Threads (users):** This field allows you to specify the number of concurrent users you will be using to hit your application.
- **Ramp-Up Period (in seconds):** This field is used to tell Apache JMeter how much time can be used to create all the threads you specified in the previous field. For example, if you set this field to 60 seconds and the **Number of Threads (users)** was set to 6, Apache JMeter will take 60 seconds to spin up all the 6 threads, one each 10 seconds.
- **Loop count and Forever:** These fields allow you to stop the test after a specific number of executions.

The remaining options are self-explanatory and in our example, we will only use the mentioned fields.

Imagine that you want to use 25 threads (like users) and you set up the ramp-up to 100 seconds. The math will tell you that a new thread will be created every 4 seconds until you have the 25 threads running (100/25 = 4). These two fields allow you to design your tests to start slowly and increase the amount of users hitting your application at the right time.

Once we have our threads/users defined, it is time to add a request because our test will do nothing without a request. To add a request, you only need to select the Thread Group node, right-click on the context menu, and choose **Add | Sampler | HTTP Request**. The previous action will add a new children node to our Thread Group. Selecting the new node, Apache JMeter, will show you a form similar to the following screenshot:



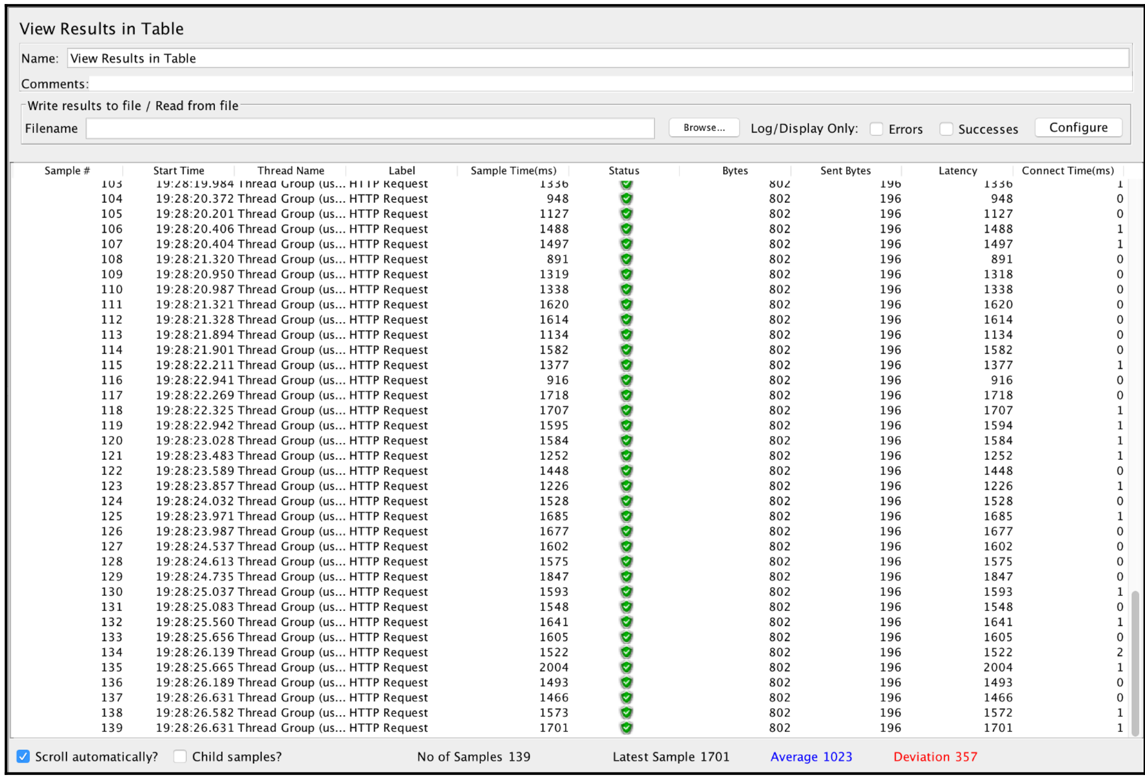
HTTP Request options

As you can see in the preceding screenshot, we can set up the host we want to hit our test with. In our case, we decide to hit `localhost` on port `8083` with a `GET` request to the `/api/v1/secret/` path. Feel free to explore the advanced options or add custom parameters. Apache JMeter is very flexible and covers practically every possible scenario.

At this point, we have set up a basic test, now it's time to see the results. Let's explore a few interesting ways to gather the information from the test. To see and analyze the results of each iteration of our test, we need to add a **Listener**. To do this, as in the previous steps, right-click on the **Thread Group** and navigate to **Add | Listener | View Results in Table**. This action will add a new node to our tests and, as soon as we start the test, the results will appear in the application.

If you had selected the **Forever** option in the **Thread Group**, you need to stop your test by hand. You can do it with the red cross icon displayed next to the green play. This button will stop the tests waiting for each thread to end their actions. If you click on the stop icon, Apache JMeter will kill all the threads immediately.

Let's give it a try and click on the green play icon to start your test. Click on your **View Results in Table** node and you will see all the results of the test appearing:

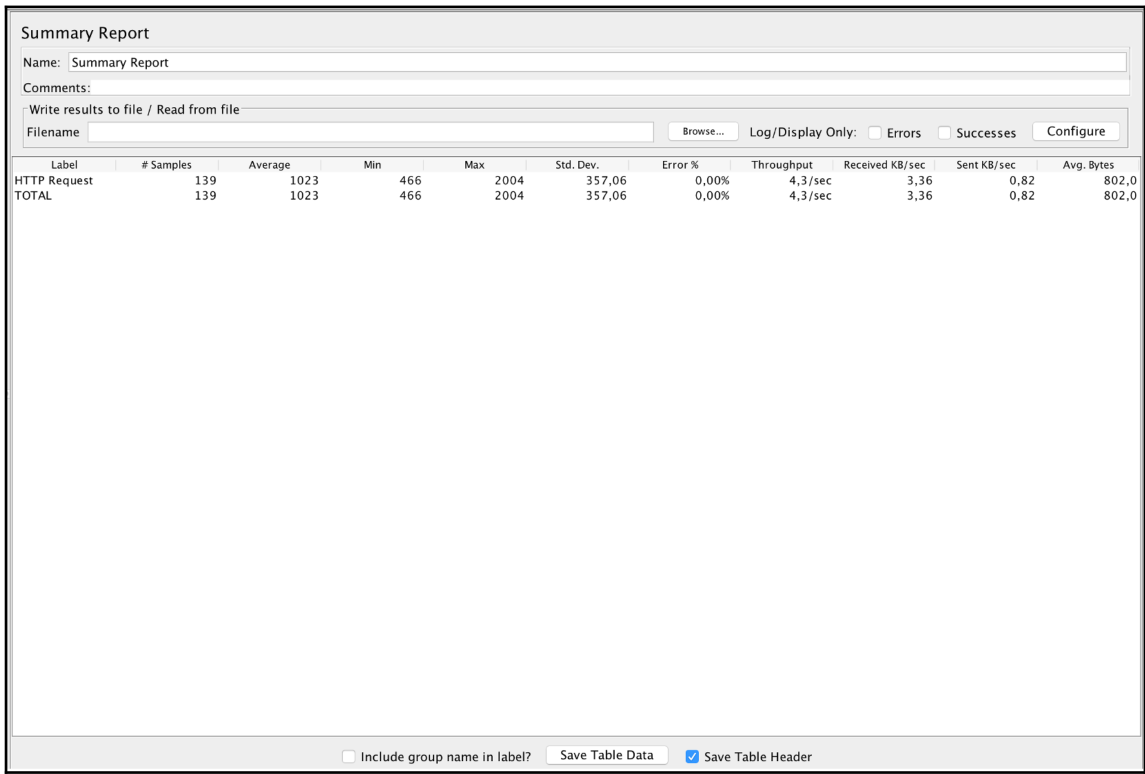


Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
103	19:28:19.984	Thread Group (us...	HTTP Request	1336		802	196	1336	1
104	19:28:20.372	Thread Group (us...	HTTP Request	948		802	196	948	0
105	19:28:20.201	Thread Group (us...	HTTP Request	1127		802	196	1127	0
106	19:28:20.406	Thread Group (us...	HTTP Request	1488		802	196	1488	1
107	19:28:20.404	Thread Group (us...	HTTP Request	1497		802	196	1497	1
108	19:28:21.320	Thread Group (us...	HTTP Request	891		802	196	891	0
109	19:28:20.950	Thread Group (us...	HTTP Request	1319		802	196	1318	0
110	19:28:20.987	Thread Group (us...	HTTP Request	1338		802	196	1338	0
111	19:28:21.321	Thread Group (us...	HTTP Request	1620		802	196	1620	0
112	19:28:21.328	Thread Group (us...	HTTP Request	1614		802	196	1614	0
113	19:28:21.894	Thread Group (us...	HTTP Request	1134		802	196	1134	0
114	19:28:21.901	Thread Group (us...	HTTP Request	1582		802	196	1582	0
115	19:28:22.211	Thread Group (us...	HTTP Request	1377		802	196	1377	1
116	19:28:22.941	Thread Group (us...	HTTP Request	916		802	196	916	0
117	19:28:22.269	Thread Group (us...	HTTP Request	1718		802	196	1718	0
118	19:28:22.325	Thread Group (us...	HTTP Request	1707		802	196	1707	1
119	19:28:22.942	Thread Group (us...	HTTP Request	1595		802	196	1594	1
120	19:28:23.028	Thread Group (us...	HTTP Request	1584		802	196	1584	1
121	19:28:23.483	Thread Group (us...	HTTP Request	1252		802	196	1252	1
122	19:28:23.589	Thread Group (us...	HTTP Request	1448		802	196	1448	0
123	19:28:23.857	Thread Group (us...	HTTP Request	1226		802	196	1226	1
124	19:28:24.032	Thread Group (us...	HTTP Request	1528		802	196	1528	0
125	19:28:23.971	Thread Group (us...	HTTP Request	1685		802	196	1685	1
126	19:28:23.987	Thread Group (us...	HTTP Request	1677		802	196	1677	0
127	19:28:24.537	Thread Group (us...	HTTP Request	1602		802	196	1602	0
128	19:28:24.613	Thread Group (us...	HTTP Request	1575		802	196	1575	0
129	19:28:24.735	Thread Group (us...	HTTP Request	1847		802	196	1847	0
130	19:28:25.037	Thread Group (us...	HTTP Request	1593		802	196	1593	1
131	19:28:25.083	Thread Group (us...	HTTP Request	1548		802	196	1548	0
132	19:28:25.560	Thread Group (us...	HTTP Request	1641		802	196	1641	1
133	19:28:25.656	Thread Group (us...	HTTP Request	1605		802	196	1605	0
134	19:28:26.139	Thread Group (us...	HTTP Request	1522		802	196	1522	2
135	19:28:25.665	Thread Group (us...	HTTP Request	2004		802	196	2004	1
136	19:28:26.189	Thread Group (us...	HTTP Request	1493		802	196	1493	0
137	19:28:26.631	Thread Group (us...	HTTP Request	1466		802	196	1466	0
138	19:28:26.582	Thread Group (us...	HTTP Request	1573		802	196	1572	1
139	19:28:26.631	Thread Group (us...	HTTP Request	1701		802	196	1701	1

Apache JMeter Results in Table

As you can see in the preceding screenshot, Apache JMeter records different data for each request, such as the amount of bytes sent/returned, the status, or the request latency, among others. All this data is interesting to analyze the behaviour of your application when you change the amount of load and with this Listener, you can even export the data so that you can use external tools to analyze the results.

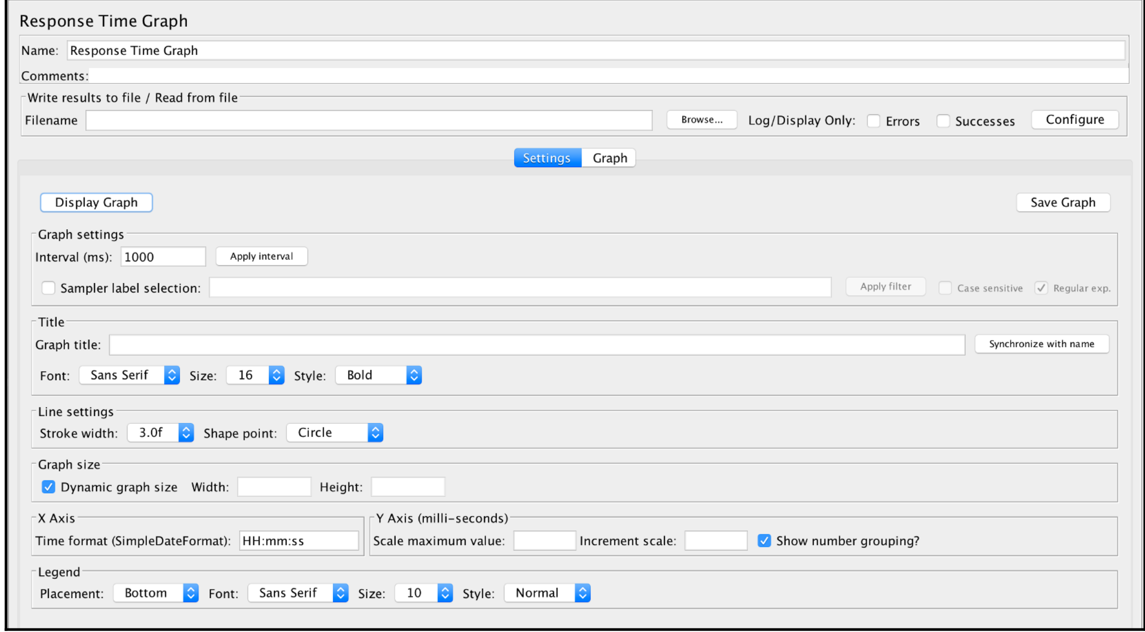
If you don't have external tools to analyze your data with, but you want to have some basic stats to compare with the different loads you are exposing your application to, you can add another interesting Listener. As we did before, open the right-click context menu of the **Thread Group** and navigate to `Add` | **Listener** | `Summary Report`. This listener will give you some basic stats that you can use to compare results:



Apache JMeter Summary Report

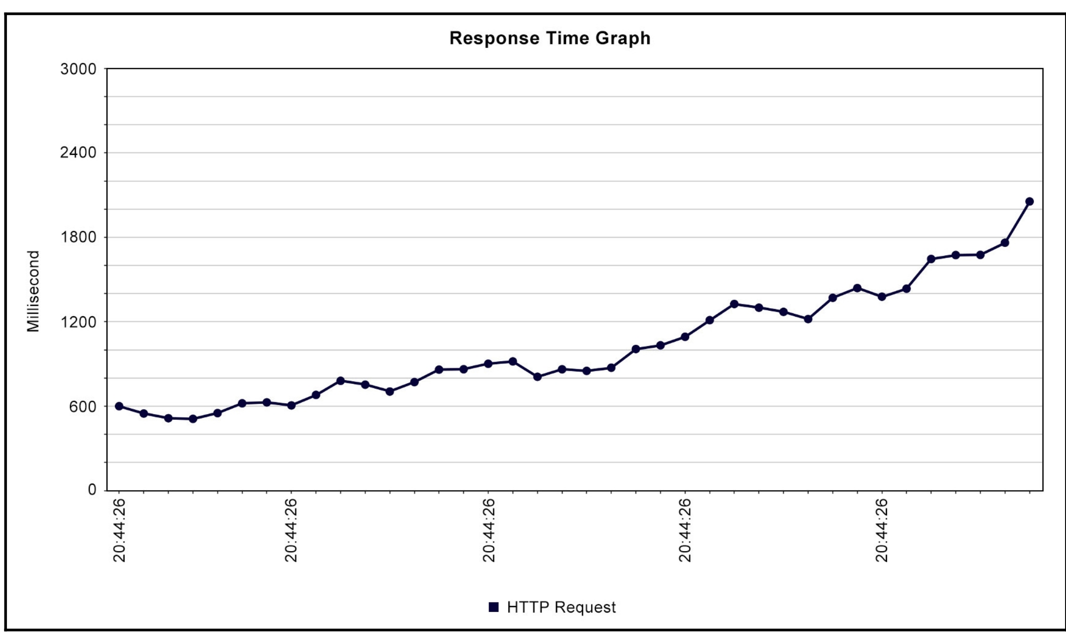
As you can see from the preceding screenshot, this listener has given us some averages from our measurements.

Using a table to show the results is fine. However, as we all know, a picture is worth a thousand words, so let's add a few graph listeners so that you can complete your load testing report. Right-click on the **Thread Group** and, in the context menu, go to **Add** | **Listener** | **Response Time Graph**. You will see a screen similar to the following one:



Apache JMeter Response Time Graph

Feel free to make some changes to the default settings. For instance, you can reduce the **Interval (ms)**. If you run your tests again, all the data generated by the test will be used to generate a nice graph, such as the following one:



Response time graph

As you can see from the graph generated from our test results, the increase of threads (users) leads to an increase in the response times. What do you think this means? If you said that our testing infrastructure needs to scale up to accommodate the increase in the loading, your response is correct.

Apache JMeter comes with multiple options to create your loading test. We have only showed you how to create a basic test and how to see the results. Now, it's your turn to explore all the different options available to create advanced tests and discover which features are more suitable for your project. Let's see some other tools that you can use for your load tests.

Load testing with Artillery

Artillery is an open source toolkit that you can use to do load testing to your application, and it is similar to Apache JMeter. Among other features, we can highlight the following advantages of this tool:

- Support for multiple protocols, and HTTP(S) or WebSockets come out of the box
- Easy to integrate with real-time reporting software or services like DataDog an InfluxDB
- High performance, so it can be used on commodity hardware/servers
- Very easy to extend, so it can be adapted to your needs
- Different report options with detailed performance metrics
- Very flexible, so you can test practically any possible scenario

Installing Artillery

Artillery was built on node.js, so the main requirement is to have this runtime installed on the machine you will use to fire the tests.

We love the containerization technology but unfortunately there is no easy way of using artillery on Docker without a dirty hack. In any case we recommend you to use a dedicated VM or server were you can fire your load testings.

To use Artillery you need node.js in your VM/server and this software is very easy to install. We are not going to explain how to create a local VM (you can use VirtualBox or VMWare to create one), we are only going to show you how you can install it on RHEL/CentOS. For other OSes and options, you can find detailed information on the node.js documentation (`https://nodejs.org/en/download/`).

Open the terminal of your RHEL/CentOS VM or server and download the setup script for the LTS version:

```
curl --silent -location  https://rpm.nodesource.com/setup_6.x | bash -
```

As soon as the previous command finishes, you need to execute the next command as root, as shown in the following command:

```
yum -y install nodejs
```

After executing the previous commands you will have Node.js installed and ready in your VM/server. It is time to install Artillery with the Node.js package manager, the `npm` command. In your terminal, execute the following command to install globally Artillery:

```
npm install -g artillery
```

As soon as the previous command finishes, you will have Artillery ready to use it.

The first thing we can do is to check that Artillery was correctly installed and that it is available. Enter the following command to do it:

```
artillery dino
```

The preceding command will show you a lovely dinosaur, which means that Artillery is ready to be used.

Executing loading tests with Artillery

Artillery is a very flexible toolkit. You can run your tests from the console or you can have YAML or JSON files, which describe your testing scenarios, and run them. Please note that in our following examples we are using `microservice_secret_nginx` as the host we are going to test, you need to adjust this host to the IP address of your local environment. Let's give you a sneak peek of this tool; run the following command in our load testing VM/server:

```
artillery quick --duration 30 --rate 5 -n 1
http://microservice_secret_nginx/api/v1/secret/
```

The preceding command will do a quick test in a duration of 30 seconds. In this time, Artillery will create five virtual users; each one of them will do one GET request to the provided URL. As soon as the preceding command is executed, Artillery will start the tests and print some stats every 10 seconds. At the end of the tests (30 seconds), this tool will show you a small report similar to the following one:

```
Complete report @ 2016-12-17T16:09:34.140Z
  Scenarios launched:  150
  Scenarios completed: 150
  Requests completed:  150
  RPS sent: 4.87
  Request latency:
    min: 578.1
    max: 1223.7
    median: 781.5
    p95: 1146.5
```

```
    p99: 1191.1
  Scenario duration:
    min: 583.2
    max: 1226.8
    median: 786.1
    p95: 1150
    p99: 1203.8
  Scenario counts:
    0: 150 (100%)
  Codes:
    200: 150
```

The preceding report is very easy to understand and gives you an overview of how your infrastructure and app are performing.

A basic concept you need to understand before we start analyzing the Artillery report is the concept of Scenarios. In a few words, a **Scenario** is a sequence of tasks or actions you want to test, and they are related. Imagine that you have an e-commerce application; a testing scenario can be all the steps a user performs before they complete a purchase. Consider the following example:

1. The user loads the home.
2. The user searches for a product.
3. The user adds a product to the basket.
4. The user goes to the checkout.
5. The user makes the purchase.

All the mentioned actions can be transformed into a request to your application that simulates the user action, which means that a scenario is a group of requests.

Now that we have this concept clear, we can start analyzing the report output from Artillery. In our sample, we only have one scenario with only one request (`GET`) to `http://microservice_secret_nginx/api/v1/secret/`. This testing scenario is executed by five virtual users who fire only one `GET` request for 30 seconds. A simple math calculation, `5 * 1 * 30`, gives us the total of scenarios tested (`150`), which is the same amount of requests in our case. The `RPS sent` field gives you the average requests per second our test server has made during our tests. It is not a very important field, but it gives you an idea about how the test is performed.

Let's check the `Request latency` and `Scenario duration` stats given by Artillery. The first thing you need to know is that all the measures from these groups are measured in milliseconds.

In the case of `Request latency`, the data is showing us the time used by our application to process the requests we have sent. Two important stats are the 95% (`p95`) and the 99% (`p99`). As you probably already know, the percentile is a measure used in statistics that indicates the value under which a given percentage of observations fall. From our example, we can see that 95% of the requests were processed in 1,146.5 milliseconds or less or that 99% of them were processed in 1,191.1 milliseconds or less.

The stats shown in the `Scenario duration` in our example are pretty much the same as the `Request latency`, because each scenario is formed by only one request. If you create more complex scenarios with multiple requests on each one, the data of both the groups will differ.

Creating Artillery scripts

As we have told you before, Artillery allows you to create YAML or JSON files with the load testing scenarios. Let's transform our quick example into a YAML file so that you can keep it in a repository for future executions.

To do this, you only need to create a file in our testing container called, for example, `test-secret.yml`, with the following content:

```
config:
 target: 'http://microservice_secret_nginx/'
 phases:
 - duration: 30
 arrivalRate: 5

scenarios:
 - flow:
 - get:
 url: "api/v1/secret/"
```

As you can see in the preceding code, it is similar to our `artillery quick` command, but now you can store them in your code repository to run it again and again against your application.

You can run your test with the `artillery run test-secret.yml` command and the results should be similar to those generated by the quick command.

The Docker container images come with the minimum software needed, so you probably can't find a text editor in our load testing image. At this point of the book, you will be able to create a Docker volume and attach it to our testing container so that you can share files.

Advanced scripting

One of the highlighted features of this toolkit is the ability to create custom scripts, but you are not only attached to fire static requests. This tool allows you to randomize the requests using external CSV files, parsing JSON responses, or inline values from the script.

Imagine that you want to test your API endpoint responsible for the creation of new accounts in your application and instead of using YAML files, you are using a JSON script. You can use an external CSV file with the user data to be used in the tests with the following adjustments:

```
"config": {
  "payload": {
    "path": "./relative/path/to/test-data.csv",
    "fields": ["name", "surname", "email"]
  }
}
// ... omitted config ...//
"scenarios": [
  {
    "flow": [
      {
        "post": {
          "url": "/api/v1/user",
          "json": {
            "name": {{ name }},
            "surname": {{ surname }},
            "email": {{ email }}
          }
        }
      }
    ]
  }
]
```

The preceding `config` fields will tell Artillery where our CSV file is located and the different columns used in the CSV. Once we have set up our external file, we can use this data in our scenarios. In our example, Artillery will pick random rows from `test-data.csv` and generate post request to `/api/v1/user` with that data. The fields field from `payload` will create variables we can use, such as `{{ variableName }}`.

Creating this kind of scripts seems easy but, at some point of the creation of your scripts, you will need some debug information to know what your script is doing. If you want to see the details of every request, you can run your script as follows:

```
DEBUG=http artillery run test-secret.yml
```

In the case of you wanting to also see the responses, you can run the load testing scripts as follows:

```
DEBUG=http:response artillery run myscript.yaml
```

Unfortunately, there is no space in the book to cover, in great detail, all the options available in Artillery. However, we wanted to show you an interesting tool that you can use to do load testing. If you need more information or even if you want to contribute to the project, you only need to go to the project's page (`https://artillery.io`).

Load testing with siege

Siege is an interesting multithread HTTP(s) load testing and benchmarking tool. It seems small and simple compared with other tools, but it is efficient and easy to use, for example, to do a quick test to your latest changes. This tool allows you to hit an HTTP(S) endpoint with a configurable number of concurrent virtual users and it can be used in three different modes: regression, Internet simulation, and brute force.

Siege was developed for GNU/Linux, but it has been successfully ported to AIX, BSD, HP-UX, and Solaris. If you want to compile it, you shouldn't have any problem on most System V UNIX variants and on most newer BSD systems.

Installing siege on RHEL, CentOS, and similar operating systems

If you use CentOS with the extras repository enabled, you can install the EPEL repo with a simple command:

```
sudo yum install epel-release
```

As soon as you have the EPEL repo available, you only need to do a `sudo yum install siege` to have this tool available in your OS.

Sometimes, the `sudo yum install epel-release` command does not work, for example, when you are not using Centos, your distribution is RHEL or a similar one. In these cases, you can install the EPEL repository by hand with the following commands:

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
sudo rpm -Uvh epel-release-latest-7*.rpm
```

Once the EPEL repository is available in your OS, you can `install siege` with the following command:

```
sudo yum install siege
```

Installing siege on Debian or Ubuntu

Siege is very easy to install on Debian or Ubuntu with the official repositories. If you have one of the latest version of these OSs, you only need to execute the following commands:

```
sudo apt-get update
sudo apt-get install siege
```

The preceding commands will update your system and install the `siege` package.

Installing siege on other OS

If your OS wasn't covered in the previous steps, you can do it by compiling the sources, there are plenty of how-tos on the Internet explaining what you need to do.

Quick siege example

Let's create a quick text to one of our endpoints. In this case, we will test our endpoint in 30 seconds with 50 concurrent users. Open the terminal of the machine you have siege installed in and type in the following command. Feel free to change the command to the correct host or endpoint:

```
siege -c50 -d10 -t30s http://localhost:8083/api/v1/secret/
```

The preceding command is explained in the following points:

- `-c50` : Create 50 concurrent users
- `-d10` : Delay each simulated user for a random number of seconds between 1 and 10
- `-t30s` : Time to run the test; 30 seconds in our case
- `http://localhost:8083/api/v1/secret/` : Endpoint to be tested

As soon as you hit *Enter*, the `siege` command will start firing requests to your server and you will have an output similar to the following one:

```
filloa:~ psolar$ siege -c50 -d10 -t30s
http://localhost:8083/api/v1/secret/
** SIEGE 3.1.3
** Preparing 50 concurrent users for battle.
The server is now under siege...
HTTP/1.1 200   0.50 secs:     577 bytes ==> GET  /api/v1/secret/
/** ... omitted lines ... **/
Lifting the server siege...      done.
```

After around 30 seconds, siege will stop the requests and show you some stats, such as the following ones:

```
Transactions:                 149 hits
Availability:                 100.00 %
Elapsed time:                 29.91 secs
Data transferred:             0.08 MB
Response time:                3.33 secs
Transaction rate:             4.98 trans/sec
Throughput:                   0.00 MB/sec
Concurrency:                  16.57
Successful transactions:      149
Failed transactions:          0
Longest transaction:          5.89
Shortest transaction:         0.50
```

From the preceding results, we can conclude that all our requests were okay, none of our requests failed, and the average response time was 3.33 seconds. As you can see, this tool is simpler and it can be used on a day-to-day basis to check at which level of concurrent users your application starts throwing errors or to put the app under stress while you check other metrics.

Scalability plan

A **scalability plan** is a document that describes all the different components of your application and the necessary steps to scale the application as soon as it is needed. The scalability plan is a live document, so you need to review it and keep it updated frequently.

There is no master template ready to fill as the scalability plan is more an internal document with all the information you need to make the right decision about the scalability of your app. Our recommendation is to use the scalability plan as your guide, including all the contents of your capacity plan, you can even add how to hire new workers to this document.

Some of the sections you can have in your scalability plan can be the following ones:

- An overview of the application and its components
- Comparison of cloud providers or places where you will deploy your application
- Resume of your capacity plan and theoretical limits of the application
- Scalability phases or steps
- Provisioning times and costs
- Organization scalability steps

The preceding sections are only a suggestion, feel free to add or remove any section to fit in your business plan.

Here's an overview of some sections of the capacity plan. Imagine that we have our example microservices application ready and want to start scaling from the minimum resources available. First, we can describe the different elements we have in our application as a basic inventory from which we evolve our application:

- Battle microservice
    - NGINX
    - PHP 7 fpm
- Location microservice
    - NGINX
    - PHP 7 fpm
- Secret microservice
    - NGNIX
    - PHP 7 fpm
- User microservice
    - NGINX
    - PHP 7 fpm
- Data storage layer
    - Database: Percona

As you can see, we have described each component needed for our application, and we have started sharing the data layer between all the microservices. We didn't add any cache layer; also, we didn't add any autodiscovery and telemetry service (we will add extra features in the following steps).

Once we have our minimum requirements, let's take a look at the different steps we can have in our scalability plan.

Step #0

In this step, we will have all our requirements in one machine even though it is not production ready yet because your application cannot survive an issue in your machine. A single server with the following characteristics will be enough:

- 8 GB RAM
- 500 GB disk

The base OS will be RHEL or CentOS, with the following software installed:

- NGINX with multiple vhosts setup
- Git
- Percona
- PHP 7 fpm

In this step, the provisioning time can be a few hours. We not only need to spin up the server, but also need to set up each one of the required services (NGINX, Percona, and others). Using tools such as Ansible can help us with the quick and repeatable provisioning.

Step #1

At this point you are getting the application ready for production, choosing between VM or containers (in our case, we have decided to use containers for flexibility and performance), splitting the single server configuration into multiple servers dedicated to each service required, like our previous requirements, and adding autodiscovery and telemetry services.

You can find a small description of the architecture of our application at this step:

- Autodiscovery
    - Hashicorp Consul container with ContainerPilot
- Telemetry
    - Prometheus container with ContainerPilot
- Battle microservice
    - NGINX container with ContainerPilot
    - PHP 7 fpm container with ContainerPilot
- Location microservice
    - NGINX container with ContainerPilot
    - PHP 7 fpm container with ContainerPilot
- Secret microservice
    - NGINX container with ContainerPilot
    - PHP 7 fpm container with ContainerPilot
- User microservice
    - NGINX container with ContainerPilot
    - PHP 7 fpm container with ContainerPilot
- Data storage layer
    - Database: Percona container with ContainerPilot

The provisioning time in this step will be reduced from hours, like the preceding step, to minutes. We have an autodiscovery service (HashiCorp Consul) in place and, thanks to ContainerPilot, each of our different components will register itself in the autodiscovery register and it will auto setup. In a few minutes, we can have all the containers provisioned and set up.

Step #2

In this step of your scalability planning, you will be adding cache layers to all the application microservices to reduce the number of requests and improve the overall performance. To improve the performance, we decided to use Redis as our cache engine, so you need to create a Redis container on each microservice. The provisioning time for this step will be like the previous one but measured in minutes.

Step #3

In this step, you will be moving the storage layer to each microservice, adding three Percona containers in Master-Slave mode with automatic setup with ContainerPilot and Consul.

The provisioning time for this step will be like the previous one, measured in minutes.

Step #4

In this step of the scalability plan you will be studying the load and usage patterns of the application. You will add load balancers in front of the NGINX containers to have more flexibility. Thanks to this new layer, we can do A/B testing or Blue/Green deployments, among other features. Some interesting and open source tools you can use in this case are Fabio Proxy and Traefik.

The provisioning time for this step will be like the previous one, measured in minutes.

Step #5

At this final step you will be checking again the application infrastructure to keep it up to date and scaling up and horizontally when necessary.

The provisioning time for this step will be like the previous one, measured in minutes.

As we told you before, the scalability plan is a live document, so you need to revise it frequently. Imagine that a new database software is created in a few months and it is optimal for high loads; you can review your scalability plan and introduce this new database in your infrastructure. Feel free to add all the information you consider important for the scalability of your application.

Summary

In this chapter, we showed you how to check the limits of your application, which gives you an idea about the possible bottlenecks you will be fighting with. We also showed you the basic concepts you need to have to create your capacity and scalability plans. We also showed you a few options to do some load testing to your application. You should have enough knowledge to have your application ready for heavy use, or at least you know the weak points of your app.

11
Best Practices and Conventions

This chapter will teach you to stand out among the rest of the developers. This is possible by developing and executing the strategies learned in this book with style, and by following concrete standards.

Code versioning best practices

Over time, your application will evolve and, at some point, you will be at the point where you are wondering what you will do with the API of any of your microservices. You can keep the changes to a minimum and be transparent to the users of your API, or you can create different versions of your code. The best solution is versioning your code (API).

The well-known and commonly used ways for code versioning are as listed:

- **URL**: In this method, you add the version of your API inside the URL of the requests. For example, the `https://phpmicroservices.com/api/v2/user` URL indicates that we are using the `v2` of our API. We used this method in our examples throughout the book.
- **Custom request header**: In this method, we do not specify the version in our URL. Instead, we use HTTP headers to specify the version we want to use. For instance, we can do a HTTP call to `https://phpmicroservices.com/api/user`, but with an extra header, `"api-version: 2"`. In this case, our server will check the HTTP header and use the `v2` of our API.

- **Accept header**: This method is very similar to the previous one, but instead of using a custom header, we will use the `Accept` header. For our example, we will make a call to `https://phpmicroservices.com/api/user` but our Accept header will be `"Accept: application/vnd.phpmicroservices.v2+json"`. In this case, we are indicating that we want version 2 and the data will be on JSON.

As you can imagine, the easiest way to implement versioning in your code is with the version code inside your URL but, unfortunately, that is not considered the best option. What most developers consider the best way of versioning your code is to use HTTP headers to specify the version you want to use. Our recommendation is to use the method that suits your project best. Analyze who will use your APIs and how, and you will discover the versioning method you need to use.

Caching best practices

A cache is a place where you can store temporal data; it is used to increase the performance of the applications. Here, you can find some small tips to help you with your cache.

Performance impact

Adding a cache layer to your application always has a performance impact that you need to measure. It does not matter where you are adding the cache layer in your application. You need to measure the impact to know if the new cache layer is a good choice.
First, make some metrics without the cache layer and, as soon as you have some stats, enable the cache layer and compare the result. Sometimes you can find that the benefit of a cache layer becomes a hell of management to keep the cache running. You can use some of the monitoring services we talked about in the previous chapters to monitor the performance impact.

Handle cache misses

A cache miss is when a request is not saved in your cache and the application needs to get the data from your service/application. Ensure that your code can handle cache misses and the consequent updates. To keep track of the rate of missing cache hits, you can use any monitoring software or even a logging system.

Group requests

Whenever possible, try to group your cache requests as much as possible. Imagine that your frontend needs five different elements from your cache server to render a page. Instead of doing five calls, you can try to group the requests, saving you some time.

Imagine that you are using Redis as your cache layer and want to save some values in the `foo` and `bar` variables. Take a look at the following code:

```
$redis->set('foo', 'my_value');
/** Some code **/
$redis->set('bar', 'another_value');
```

Instead of doing that, you can do both the sets in a single transaction:

```
$redis->mSet(['foo' => 'my_value', 'bar' => 'another_value']);
```

The preceding example will do both the sets in one commit, saving you some time and improving the performance of your application.

Size of elements to be stored in cache

It is more efficient to store large items in your cache than storing small items. If you start caching loads of small items, the overall performance will decrease. In this case, the serialization size, time, cache commit time, and capacity usage will grow.

Monitor your cache

If you have decided to add a cache layer, at least keep it monitorized. Keeping some stats of your cache will help you know how well it is doing (the cache hit ratio) or if it is reaching its capacity limit. Most of the cache software is stable and robust, but it does not mean that you will not have any problems if you keep it unmanaged.

Choose your cache algorithm carefully

Most of the cache engines support different algorithms. Each algorithm has its benefits and its own problems. Our recommendation is to analyze your requirements deeply and not use the default mode of the cache engine of your choice until you are sure that it's the correct algorithm for your use case.

Performance best practices

If you are reading this book, it is probably because you are interested in web development and, in the last few years, the importance of the performance of web applications (such as APIs) is becoming more and more relevant. Here are some stats to give you an idea:

- Amazon reported years ago that for every 100 ms of increase in the loading time, their sales decreased by 1%.
- Google found that reducing the size of the page from 100 KB to 80 KB diminished their traffic by 25%.
- 57% of online consumers will abandon a site after waiting for 3 seconds for a page to load.
- 80% of the people who abandoned the site will not return. About a 50% of these people will tell others their negative experience.

As you can see, the performance of your app can impact your users and even your revenues. In this section, we will give you some tips to improve the overall performance of your web application.

Minimize HTTP requests

Each HTTP request has a payload. For this reason, an easy way to increase the performance is to reduce the number of HTTP requests. You need to have this idea in mind in every aspect of your development. Try to do the minimum external calls to other services in your APIs/backends. In your frontends, you can combine files to attend to only one request. You only need to have a balance between the number of requests and the size of each request.

Imagine that your frontend has its CSS split in several different files; instead of loading each one every time, you can combine them into a single or a few files.

Another quick and small change you can do with HTTP requests is to try to avoid `@import` functions in your CSS files. Using link tags instead of `@import` functions will allow your browser to download the CSS file in parallel.

Minimize HTML, CSS, and JavaScript

As developers, we try to code in a format that is easier for us to read–a human-friendly format. By developing this way, we are increasing the size of our plain text files with unnecessary characters. Unnecessary characters can include white space, comments, and newline characters.

We are not saying that you need to write obfuscated code, but as soon as you have everything ready, why don't you remove the inessential characters?

Imagine that the content of one of your JavaScript file (`myapp.js`) is as follows:

```
/**
* This is my JS APP
*/
var myApp = {
  // My app variable
  myVariable : 'value1',

  // Main action of my JS app
  doSomething : function () {
    alert('Doing stuff');
  }
};
```

After the minimization, your code can be saved to a different file (`myapp.min.js`) and it can look as follows:

```
var myApp={myVariable:"value1",doSomething:function()
  {alert("Doing stuff")}};
```

In the new code, we reduced the size of the file around 60%, a huge saving. Note that our repository will have both versions of the file: the human-friendly to make our changes, and the minimized one, which we will load in our frontend.

You can do the minimization with online tools, or you can integrate tools such as `gulp` or `grunt` in your pipeline. When you have set up these tools, they will track the changes of some specific files (CSS, JS, and others) and as soon as you save your changes to any of these files, the tool will minimize the content. Another hidden benefit of using tools for the minification is that most of them also check the code or rename your variables to keep them even smaller.

Image optimization

One of the most used assets in web development can be the images. They make your website look amazing, but they can make your site painfully slow. The main recommendation is to keep the number of images to the minimum, but if you need to keep images, at least try to optimize them before they are sent to your users. In this section, we will show you some things you can do to optimize your images and, as a consequence, the performance of your application.

Use sprites

A sprite is an image composed by multiple images; later, you can use this image and show only the portion you are interested in. Imagine that you have a nice website and, on each page, you have some social icons (Facebook, Twitter, Instagram, and so on). Instead of having one image for each social icon, you can combine them in one and use CSS to show only the part you want for each icon. Doing this, you will have all the social icons with only one load, reducing the number of requests.

Our recommendation is to keep your sprites small and include only the most used and shared images in them.

Use lossless image compression

Not all image formats are suitable for the Web as some formats are either too big or do not support compression. The three most used image types on the Web nowadays are as listed:

- **JPG**: This is one of the most commonly used method of lossless compression
- **PNG**: This is the best format with lossless data compression
- **GIF**: This is an old-school format that supports up to 8 bits per pixel for each image and is well known for its animated effects

The recommended format for the web at this moment is **PNG**. It is well supported by the browsers, easy to create, supports compression, and gives you all the power you need to improve the performance of your site to the maximum.

Scale your images

If you are using images and not data URIs, they should be sent at their original size. You should avoid resizing your images using CSS and send the image with the correct size to the browser. The only case when it is recommended to scale images by CSS is with fluid images (responsive design).

You can scale your images easily with PHP libraries like Imagick or GD. With these libraries and a few lines of code, you can scale your images in seconds. Usually, you do not scale the images on the fly. Most of the time, as soon as an image is uploaded to your application, a batch process takes care of the image, creating the different sizes needed by your application.

Imagine that you can upload images of any size to your application and you only show images with a maximum width of `350`px in your frontend. You can easily scale the previously stored image with Imagick:

```
$imagePath = '/tmp/my_uploaded_image.png';
$myImage   = new Imagick($imagePath);

$myImage->resizeImage(350, 0, Imagick::FILTER_LANCZOS, 1);

$myImage->writeImage('/tmp/my_uploaded_image_350.png');
```

The preceding code will load the `my_uploaded_image.png` file and resize the image to a width of `350`px using the Lanczos filter (refer to the PHP Imagick documentation to see all the available filters you can use).

That's one way of doing it, another (and maybe even more effective) common way is to resize images on demand (that is, when first requested from client side), and then store the resized image either in cache or permanent storage.

Use data URIs

Another quick way of reducing the number of HTTP requests is embedding images as data URIs. This way, you will have the image as a string inside your code avoiding the request of the image, this method is the best fit for static pages. The best way to generate this kind of URIs is with external or online tools.

The following example will show you how it will look in your HTML:

```
<img src="data:image/jpeg;base64,/9j/4AAQSkZJRgABAQEAYABgA..."
 alt="My Image">
```

Cache, cache, and more cache

Performance on the Web is all about serving data as fast as possible and, if our application already sent data that continues to be valid, why send it again? By default, modern browsers try to reduce the number of requests they make to the same site, so they keep a copy of some assets/resources in their internal cache for their future use. Thanks to this behavior, if you are browsing a website, we are not trying to load all the assets again and again as soon as you move between sections.

You can help your browser in specifying each request response with the following `Cache-Control` HTTP headers:

- **max-age=[seconds]**: This sets the maximum amount of time that a response will be considered fresh. This directive is relative to the time of the request.
- **s-maxage=[seconds]**: This is similar to max-age, but for shared caches.
- **public**: This tag marks the response as cacheable.
- **private**: This tag allows you to store the response to one user. Shared caches are not allowed to store the response.
- **no-cache**: This tag instructs caches to submit the request to the original server for validation.
- **no-store**: This tag instructs caches not to keep a copy of the response.
- **must-revalidate**: This tag tells the caches that they must follow any fresh information you give them about a response.
- **proxy-revalidate**: This is similar to must-revalidate, but for proxy caches.

The main recommendation is to tag static assets with an expiration of at least one week or one day on long-life assets. In the case of assets that change frequently, the recommendation is to set the expiration to a couple of days or less. Adjust the cache expiration dates of your assets according to their life.
Imagine that you have one image that changes every 6 hours; in this case, you should not set the expiration date to a week, the best choice will be around 6 hours.

Avoid bad requests

There is nothing more annoying than bad requests as this kind of requests can reduce the performance of your application drastically. As you probably know, the browsers have a limited number of concurrent connections that they can manage at the same time for the same host. If your website makes a lot of requests, this list of available slots for connections can be full and the remaining requests are queued.

Imagine that your browser can manage up to 10 concurrent connections and your web application makes 20 requests. Not all the requests can be attended to at the same time and some of them are queued. What happens now if your application is trying to get an asset and it does not exist? In this case, the browser will waste its time (and slot) waiting for the non-existent asset to be served, but it will never happen.

As a recommendation, keep an eye on your browser developer tools (a set of web debugging tools built into the browsers you can use to debug and profile your site). These tools allow you to spot problematic requests and you can even check the amount of time used on each request. In most of the browsers, you can open the embedded developer tools pressing the *F12* key but, if your browser do not open the tools on pressing this key, check the browser's documentation.

Use Content Delivery Networks (CDNs)

A content delivery network hosts a copy of your assets in servers designed to respond quickly and from the closest server possible. That way, if you are moving the requests from your servers to the CDN servers, your web servers will be dealing with fewer requests, improving the performance of your application.

Imagine that you use jQuery in your frontend; if you change your code to load the library from an official CDN, the probability of an user having the library in their browser cache increases.

Our main recommendation is to use CDNs at least for your CSS, JavaScript, and images.

Dependency management

You have multiple PHP libraries, frameworks, components, and tools available to use in your project. Until a few years ago, PHP did not have a modern way of managing project dependencies. At this moment we have Composer, a flexible project that was converted into the de facto standard of dependency management.

You are probably familiar with Composer as we were using this tool all over the book to install new libraries in the `vendor` folder. At this point, you will be wondering whether you should commit the dependencies of your `vendor` folder. There is no quick response, but the general recommendation is no, you should not commit the `vendor` folder to your repository.

The main disadvantages of committing the vendor folder can be summarized as follows:

- Increases the size of your repository
- Duplicates the history of your dependencies

As we told you before, not committing the vendor is the main recommendation, but if you really need to do it, here are some suggestions:

- Use tagged releases (no dev versions) so that Composer fetches zipped sources
- Use the `--prefer-dist` flag or set `preferred-install` to `dist` in your config file
- Add the `/vendor/**/.git` rule to your `.gitignore` file

Semantic versioning

In any project you start, you should use semantic versioning on your master branch. Semantic versioning is a set of rules you can follow to tag the code of your application in your versioning control software. By following these rules, you will know the current status of your production environment at any moment. Another benefit of using tags in your code is that it allow us to move between versions or do roll backs in an easy and quick way.

Another advantage of having your source code with release tags is that it allows you to work with release branches, allowing you to have better planification and control of the changes you are making to your code.

How semantic versioning works

On semantic versioning, your code is marked with tags with the `vX.Y.Z` form, which means the version of your code. Each piece of your tag means something:

- **X (major)**: An increase in this version number indicates that there are big changes in place; they are important enough to be incompatible with the current version
- **Y (minor)**: An increment in this version number indicates that we are adding new features to our project
- **Z (patch)**: An increment in this version number indicates that we added a patch to your source code

The actualization of the release tag is usually made by the developer who will push code to the production environment. Please remember to update the release tag before the deployment of your code.

Semantic versioning in action

Imagine that you start in someone else's project and the master branch is tagged as `v1.2.3`. Let's look at some examples.

We have been told to add a new feature to the project

Working on a live project leads to receiving petitions for new features. In this case, we are clearly dealing with an increment of the minor version number because we are adding new code that is not incompatible with the actual base code. In our case, if our master branch is on `v1.2.3`, the new version tag will be `v1.3.0`. We have increased the minor version number and also, we reset the patch number because we are adding new code.

We have been told that there is a bug in our project

On a day-to-day basis you will be fixing bugs in your code. In this case, we are dealing with a small change that the main function is to solve our issue, so we need to increase the patch version. In our example, if the current production version is `v1.2.3`, the new version tag will be `v1.2.4`. We only increased the patch number because our fix does not imply other, bigger changes.

We have been asked for a big change

Imagine now that we have been asked for a big change to our source code; once we apply our change, some parts of our source will be incompatible with the previous versions. For example, imagine that you are using `library_a` and we changed to using `library_b` and they are mutually exclusive. In this case, we are dealing with a very big change that indicates that we need to increase our major version number and also, we need to reset the minor and patch numbers. For example, if our production code is tagged as `v1.2.3`, the new version code after you apply your changes will be `v2.0.0`.

As you can see, doing semantic versioning will help you keep your source clean and makes it easier to know which kind of code changes are being done only by looking at the version number.

Error handling

When we throw an exception because something happened during the execution of our application, we should give more information to our users or consumers about what happened. This is possible by adding describable standard codes, also known as status codes. Using these standard codes in your responses will help you (and your colleagues) to know quickly if something is going wrong in your application. Check the following list to know what are the correct and most common HTTP status codes to use them in your API.

Client request successful

If your application needs to inform the API client that the request was successful, you would usually reply with one of the following HTTP status codes:

- **200 – OK**: The request was done successfully
- **201 – Created**: Successfully created the URI specified by the client
- **202 – Accepted**: Accepted for processing but the server has not finished processing it
- **204 – No Content**: Request is complete without any information being sent back in the response

Request redirected

When your application needs to reply to a request telling that the request was redirected, you will be using one of the following HTTP status codes:

- **301 – Moved Permanently**: Requested resource does not exist on the server. A Location header is sent to the client to redirect it to the new URL. Client continues to use the new URL in the future requests.
- **302 – Moved Temporarily**: Requested resource has temporarily moved. A Location header is sent to the client to redirect it to the new URL. The client continues to use the old URL in the future requests.
- **304 – Not Modified**: Used to respond to the `If-Modified-Since` request header. It indicates that the requested document has not been modified since the specified date, and the client should use a cached copy.

Client request incomplete

If the information you need to send to the API client is about an incomplete or wrong request, you will be returning one of the following HTTP codes:

- **400 – Bad Request**: The server detected a syntax error in the client's request.
- **401 – Unauthorized**: The request requires user authentication. The server sends the WWW-Authenticate header to indicate the authentication type and realm for the requested resource.
- **402 – Payment Required**: This is reserved for the future.
- **403 – Forbidden**: Access to the requested resource is forbidden. The request should not be repeated by the client.
- **404 – Not Found**: The requested document does not exist on the server.
- **405 – Method Not Allowed**: The request method used by the client is unacceptable. The server sends the `Allow` header stating what methods are acceptable to access the requested resource.
- **408 – Request Time-Out**: The client has failed to complete its request within the request timeout period used by the server. However, the client can re-request.
- **410 – Gone**: The requested resource is permanently gone from the server.
- **413 – Request Entity Too Large**: The server refuses to process the request because its message body is too large. The server can close connection to stop the client from continuing the request.

- **414 – Request-URI Too Long**: The server refuses to process the request because the specified URI is too long.
- **415 – Unsupported Media Type**: The server refuses to process the request because it does not support the message body's format.

Server errors

On the unfortunate event of your application needing to inform the API client that there is some problem, you will be returning one of the following HTTP codes:

- **500 – Internal server error**: A server configuration setting or an external program has caused an error.
- **501 – Not Implemented**: The server does not support the functionality required to fulfill the request.
- **502 – Bad gateway**: The server encountered an invalid response from an upstream server or proxy.
- **503 – Service unavailable**: The service is temporarily unavailable. The server can send a `Retry-After` header to indicate when the service may become available again.
- **504 – Gateway Time-Out**: The gateway or proxy has timed out.

Coding practices

Your code is the heart of your application; therefore, you want to write it properly, cleanly, and in an efficient manner. In this section, we will give you some hints to improve your code.

Dealing with strings

One of the standards of the industry is to use the UTF-8 format in all your application levels. If you skip this recommendation, you will be dealing with encoding problems all your project's life. At the moment of writing this book, PHP does not support Unicode at low level, so you need to be careful when dealing with strings, specially with UTF-8. The following recommendations are only if you are working with UTF-8.

In PHP, the basic string operations such as assignation or concatenation don't need anything special in UTF-8; in other situations, you can use the core functions to deal with your strings. Most of the time, these functions have a counterpart (prefixed as `mb_*`) to deal with the Unicode. For example, in the PHP core, you can find the `substr()` and `mb_substr()` functions. You must use the multibyte functions whenever you operate with Unicode string. Imagine that you need to get a part of a UTF-8 string; if you use `substr()` instead of `mb_substr()`, there are good chances to get the result you are not expecting.

Single quotes versus double quotes

Single quoted strings are not parsed by PHP, so it doesn't matter what you have in your string, PHP will return the string unchanged. In the case of double quoted strings, they are parsed by the PHP engine and any variable in the string will be evaluated. With double quoted strings, the escaped characters (t or n, for example) will also be evaluated.

In real-life applications, the difference of performance of using one or another can pass unnoticed but, on high-load applications, the performance can be different. Our recommendation is to be consistent and use only double quotes if you need variables and escaped characters to be evaluated. In any other case, use single quotes.

Spaces versus tabs

There is a war between developers who use spaces and developers who use tabs to tabulate their code. Each approach has its own benefits and inconveniences, but the PHP FIG recommendation is to use four spaces. Using only spaces avoids problems with diffs, patches, history, and annotations.

Regular expressions

In PHP, you have two options to write your regular expressions: the PCRE and the POSIX functions. The main recommendation is to use the PCRE functions (prefixed by `preg_*`) because the POSIX family of functions have been deprecated in PHP 5.3.

Connection and queries to a database

You have multiple ways of connecting to a database in PHP but, among all of them, the recommended way of connecting is using PDO. One of the benefits of using PDO is that it has a standard interface to connect to multiple and different databases, allowing you to change your data storage without too much problem. When you are making queries against your database, and if you don't want any problem, ensure that you always use prepared statements. This way, you will avoid most of the SQL injection attacks.

Using the === operator

PHP is a loose type programming language, and this flexibility comes with some caveats when you are comparing your variables. If you use the === operator, PHP ensures that you are doing a strict comparison and avoiding false positives. Note that === is slightly faster than the `is_null()` and `is_bool()` functions.

Working with release branches

Once we have our project following semantic versioning, we can start working with releases and release branches in our version control system, for example, Git. Using releases and release branches allows us to plan and organize the changes we will make in our code better.

The work with releases is based on the semantic versioning due to the fact that each release branch will be created from master, usually from the latest master version that has a tag (for example, v1.2.3).

The main benefits of using release branches are as listed:

- Helps you follow a strict methodology for pushing your code to production
- Helps you easily plan and control the changes you make to your code
- Tries to avoid the common issue of dragging unwanted code to production
- Allows you to block special branches, such as dev or stage, to avoid commits without pull requests

Note that it is only a recommendation; each project is different, and this workflow can be unsuitable for your project.

Quick example

To use releases in your project, you need to work with a release branch and another temporal branch where you make the changes to the code. For the following example, imagine that our project has the master branch tagged as v1.2.3.

The first step is to check whether we already have a release branch against which we will be working. If it is not the case, you need to create a new one from master:

- First, we need to decide which will be our following version number; we will use all we have learned from the semantic versioning.
- Once we know our following version number, we will create a release branch from master. The next command will show you how to get the latest master branch and create and push a new release branch:

```
git checkout master
git fetch
git pull origin master
git checkout -b release/v1.3.0
git push origin release/v1.3.0
```

- After the preceding steps, our repository will have our release branch clean and ready to be used.

At this point, we have our release branch ready. This means that any code modification will be done in a temporal branch created from our release branch:

- Imagine that we need to add a new feature to our project, so we need to create a temporal branch from our release branch:

```
git checkout release/v1.3.0
git fetch
git pull origin release/v1.3.0
git checkout -b feature/my_new_feature
```

- As soon as we have our `feature/my_new_feature`, we can commit all our changes to this new branch. Once all our changes are committed and ready, we can merge our `feature/my_new_feature` with the release branch.

The preceding steps can be repeated any number of times until all the tasks you have scheduled for the release are done.

Once you have finished all the release tasks and all your changes have been approved, you can merge the release branch with master. As soon as you finish the merge with master, remember to update the release tag.

We can summarize our example with the following reminder notes:

- New release branches are always created from master
- Temporal branches are always created from the release branches
- Try to avoid the merge of other temporal branches with the current temporal branch
- Try to avoid the merge of non-scheduled branches with your release branch

In the preceding workflow, we recommend using the following branch prefixes to know the type of change associated with the branch:

- `release/*`: This prefix indicates that all the included changes will be deployed in the future release with the same version number
- `feature/*`: This prefix indicates that any change added to the branch is a new feature
- `hotfix/*`: This prefix indicates that the included changes are committed to fix bugs/issues

Working this way, it will be more difficult to push unwanted code to production. Feel free to adapt the preceding workflow to your needs.

Summary

In this chapter, we gave you some bits and bobs about the common best practices and conventions you can use in your projects. They are all recommendations, but they make the difference to stand out from the rest of the projects.

12
Cloud and DevOps

We did not want to end the book without talking about Cloud and DevOps functions. Having a server in house is not a good solution when Cloud platforms exist; so, in this chapter, you will understand why you should use Cloud for your application and which provider is the best for your requirements. Also, you will learn how to deploy your application into these Cloud platforms using automated tools.

The DevOps' role is closely related to the Cloud, so we will go through this subject and what the DevOps tasks are.

What is Cloud?

The fastest way to explain what we know as Cloud is by saying that the Cloud is the delivery of online services hosted on the Internet, but we can also say that Cloud allows us to consume digital resources in a very easy way. Some common Cloud services used these days are disk storage, virtual machines or TV services among others. As you can imagine, the main benefit of the Cloud is that we do not need to build and maintain these infrastructures at home.

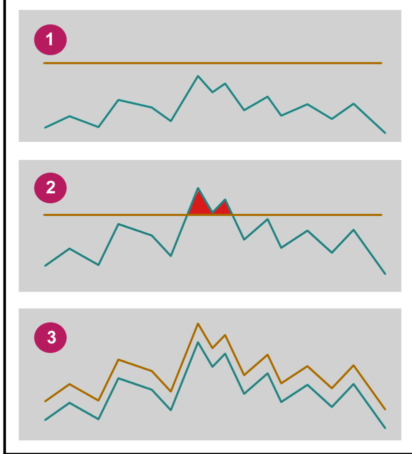
As developers, you will know that Cloud is a good approach for our applications. Let's take a look at some advantages.

Autoscalable and elastic

When your application is online, it is impossible to predict whether the traffic will be very high in a few months or even a few days. Cloud allows us to have an autoscalable infrastructure that matches the traffic or consumption resources of our application. It can grow if your traffic is higher or decrease if your application does not have the traffic you hoped for.

Usually, there are three options when you want to resize the servers. In the next picture we will be showing you graphically the different options you have to resize your servers. The yellow line is the maximum load your application can manage and the blue line the current load of your site:

- **Picture 1**: Use more servers than you need in order to avoid traffic problems when peak traffic occurs.
- **Picture 2**: Use enough servers for normal traffic. You should know that it is possible to have problems on specific days. For example, if your application is an online shop, problems may arise on days like Black Friday.
- **Picture 3**: Use an elastic Cloud; it increases and decreases by adding or removing servers automatically depending on the peak traffic so that you always have the infrastructure you need.



Resizing ways

Lower management efforts

If you lose time setting up your server and performing maintaining tasks, you are losing time that you could use on improving your application. Cloud allows us to just focus on our application because it provides us with a new way to develop applications, providing preconfigured resources that allow us to develop applications without worrying about the infrastructure we are working on.

In addition, Cloud usually provides us with a complete and useful dashboard to manage the machines, so we do not need to use an SSH console anymore, making our tasks easier. It even provides us with better ways to manage our databases and load balancers or certificates.

Cheaper

Using Cloud is cheaper than having the servers at home. These savings are because of the following reasons:

- You are only paying for the infrastructure you need all the time, so you do not need to change your machines when the traffic on your application grows
- The IT guys (in case you need them) will be more productive because they will only focus on problems that Cloud cannot help with

Be aware when you pay for a Cloud server that you are not only paying for the servers; this server also includes the storage, an operating system, virtualization, the physical space, updates, a cooling system, and many other things, such as energy or data center operations.

Grow faster

This point is closely linked to the last one. If your application is new and you do not know if it will be successful, it is not a good idea to buy physical servers and all the related things in order to put your application online.

Using Cloud, you can pay monthly for the server you need and if the application does not go as expected, you can reduce the plan or even close it and you will spend less money.

Also, saving money at the beginning will allow you to grow faster, paying attention and putting money into the application instead of spending money on hardware.

Time to market

If you want to test new ideas and put them online, it will be faster. This is the main Cloud advantage and it is very precious on the internet. For big companies, it is very difficult to go as fast as small companies and Cloud allows them to include changes online in an easy and fast way, making this a very competitive advantage.

Select your Cloud provider

Choosing the best Cloud provider is not an easy thing, but you can check your application needs in order to select the provider that suits it best.

Consider the following things:

- **Ensure that your provider knows your needs**: The communication between your team and your Cloud provider is essential. It is very important that your provider knows things such as read/write number per second, where the users are from if there are concurrent users, how your deploy scripts work, or what your development, staging, and production environments are like.
- **Where is my data**: The Cloud servers are located somewhere, so it is important to know where they are because if you store customers' data, the law may not allow you to store data in some countries.
- **Security**: If your application is not secure, you are at risk all the time, so it is good to know what protection systems your Cloud provider has, such as firewalls or how they isolate the hardware, in order to avoid intrusions and whether they offer 24-hour support.
- **Test it before moving your application**: Your Cloud provider may allow you to test the service before moving the entire application, so use this option in order to check whether the servers will be enough for your traffic and resources.

Amazon Web Services (AWS)

The internet giant Amazon.com has its own Cloud. It provides web hosting and also, many other services to help companies. The most important feature on **Amazon Web Services** (**AWS**) is the load balancer and the possibility of sending some of your application's tasks (such as processing data or web hosting) to Amazon.

To sum up, AWS is not just a simple Cloud, it includes many services (more than 50) for web experts and other people that require specific features that Amazon offers.

Amazon gives us some different price plan options depending on the time we use it for–price per hour, per year, or even 3 years are the possibilities of AWS.

An important thing on the Cloud is the **Service Level Agreements** (**SLA**). For Amazon, these include a 99.95% of uptime guarantee with monthly cycles.

The customization on AWS works like templates; in other words, at the moment a full configuration of CPU, RAM, and space for your application is not possible; you should choose between some template options, so if you only need more RAM, you can not just upgrade the RAM, you should choose a different template that can upgrade the CPU and hard disk too.

Usually, the Cloud servers are in many countries around the world. Amazon EC2 (the servers of AWS) are currently located in North America (16), South America (3), Europe (7), Asia (14).

It may be worth noting that, at the moment and according to general consensus, AWS gives us the worst cost-benefit ratio in bandwidth and processing power among others. It's still the most feature complete solution out in the market, but it might not be the best fit for you.

Microsoft Azure

Azure is the Microsoft operating system that provides us with an environment to execute and deploy applications and services on the Cloud. It provides us with a custom environment and servers located on the Microsoft data centers.

The applications we store on Azure should work on Windows Server 2008 R2, and they can be developed on .NET, PHP, C++, Ruby, or Java. Also, Azure provides us with some database mechanisms, such as NoSQL, blobs, message queues, and NTFS drives to read/write disk operations.

The main advantages of Windows Azure are as listed:

- It reduces operation costs and provisioning on the applications
- Fast response to customer need changes
- Scalability

Azure gives us different ways of paying for the services, you can pay for hour fractions or you can make yearly payments. The SLA of Azure is the same as Amazon and includes a 99.95% of up time guarantee with monthly cycles; the customization works with templates too.

Azure Cloud servers are currently located in North America (9), South America (1), Europe (6), Asia (9), and Australia (2).

In conclusion, Amazon and Azure are very similar; the main difference between them is the operating system used. If your application is developed on .NET or requires Windows servers, Azure is the best option.

Rackspace

Rackspace is not one of the biggest ones (such as Amazon or Microsoft), but it is considered as one that we should mention when we talk about Cloud services.

When we hire Rackspace, we are paying to use the service, for example, when we need to increment the capacity of our application at a specific moment. The Rackspace servers are administered by them and it is even possible to hire only the support system, having our servers outside Rackspace.

Rackspace gives us the option of paying for less than an hour of time, yearly, or even for 3 years. The SLA is 99.90% uptime guarantee with monthly cycles, and the customization works using templates, like Amazon and Azure do.

The servers are currently located in North America (3), Europe (1), Asia (1), and Australia (1).

In conclusion, Rackspace is cheaper than Amazon or Azure, and it is a very good solution to start working with Cloud. Also, it has a very good distributed DNS and they are the creators of **OpenStack**, an open source stack of different software components used to implement the Cloud servers through virtualization. This offers a new dashboard, add-on services with databases, server monitoring, block storages, and creation of virtual networks.

DigitalOcean

DigitalOcean is the second biggest hosting company in the world. Their plans are the cheapest ones and the DigitalOcean community is really good–they have forums for developers and a lot of tutorials about administration servers.

It is possible to choose the option of paying hourly or monthly. Also, the SLA is 99.99% uptime guarantee, even better than Amazon or Azure, and the customization works using templates, just like Amazon, Azure, and Rackspace do.

They currently have five servers in North America, five in Europe, and one in Asia.

DigitalOcean is a good solution for experts because they do not administer the servers. The servers are always Linux, so this is not a solution for projects that require Windows. Also, an advantage of using DigitalOcean is that if your project grows, it is possible to easily and quickly scale your server.

Joyent

Samsung bought **Joyent**. This Cloud has great potential. It was created to compete with Amazon EC2, and it had some important customers, such as Twitter and LinkedIn.

Joyent created Node.js and they have the best technology for containers; it was inherited from Solaris and implemented on their own operating system, **SmartOS**, (an operating system designed for the Cloud). If you are looking for the best performance and you do not care about the price, Joyent is your best friend.

You can choose the option of paying hourly, yearly, or even every 3 years. Also, the SLA is 100% uptime guarantee with 30 minutes cycles, the best one. The customization works using templates, just like Amazon, Azure, Rackspace, or DigitalOcean do.

They have three servers in North America and one in Europe at the moment.

Rackspace and Joyent have an open source infrastructure, so it is possible to download and use it on your own machine.

Google Compute Engine

Google Compute Engine is a complete product that includes infrastructure and service, allowing us to execute virtual machines with Linux on the same infrastructure as Google works.

The Google Compute Engine dashboard could not be better. It is clean and easy to navigate through. Also, it is very fast during the deployment and the scalability process, and the tools included on this Cloud make Google Compute Engine a good solution for analytics and big data.

For Google Compute Engine, the SLA is 99.95% uptime guarantee with monthly cycles. It allows us to store up to seven snapshots for free.

They have nine servers in North America, three in Europe, and six in Asia at the moment.

Deploying your application to the Cloud

Throughout the book, we were working with containers; we already told you how beneficial they are for your projects. Now, it's time to deploy your application to the Cloud. There are different providers out there, and we gave you some hints on how to choose the best provider for your project. In this section, we will show you some interesting options you have to orchestrate and manage your containers in production.

Docker Swarm

We were playing with Docker and their **Docker Engine** throughout the book. With the Docker engine, we are able to spin up and down the containers we use in our application. As you imagine, you can install the Docker Engine in your production server and use it like our development environment, but do you think that this approach is fault tolerant? Obviously the response is no. You can try to do some magic having multiple Docker Engines in different servers, but it will be hard to set up and maintain. Fortunately, Docker created Docker Swarm; this software provides you with native clustering capabilities and turns your group of Docker Engines into a single virtual Docker Engine. It will be like working in your development machine; the Swarm will be dealing with all the hard stuff for you, you only need to take care of your application.

As we want you to have a global overview of Docker Swarm, listed are the main features of this tool:

- **Compatible with Docker tools**: Docker Swarm uses and provides the standard Docker API, so any tool that already uses the Docker API can use Docker Swarm to transparently scale to multiple hosts.
- **High scalability and performance**: Like all the Docker software, Swarm is production ready and it was tested to scale up to one thousand (1,000) nodes and fifty thousand (50,000) containers. The results of those tests showed that you can achieve these high deploy numbers without performance degradation in the node cluster.

- **Failover and high availability**: Docker Swarm is ready to manage failover and give you high availability. You can create multiple Swarm masters and specify your policy for leader election on master failures. At the time of writing this book, there is an experimental support for container rescheduling from failed nodes.
- **Flexible container scheduling**: Swarm comes with a built-in scheduler that will be responsible for maximizing the performance and resource utilization of your infrastructure.
- **Pluggable schedulers and node discovery**: If the built-in scheduler that comes with Swarm does not fit well with your requirements, you can plug in external schedulers, such as **Apache Mesos** or **Kubernetes**. To fulfill all the autodiscovery requirements of your application, you can choose between the different available methods in Swarm: a hosted discovery service, a static file, Consul, or Zookeeper.

Installing Docker Swarm

From Docker 1.12 and higher, the Swarm mode comes out of the box, so installing it is very easy. You only need to follow the steps we showed you in the `Chapter 2`, *Development Environment*, but instead of performing them on your development machine, you need to perform them on your production servers. We recommend using Linux/Unix in your production nodes, so all the steps we are describing here are for a Linux/Unix system.

We will build a Swarm cluster so, before you continue reading, ensure that you have the following requirements ready:

- A minimum of three host machines with Docker Engine 1.12 (or higher) installed
- One of the machines will be the manager machine, so ensure that you have all the IP addresses of your hosts
- Open ports between your hosts
    - **TCP port 2377**: This port is used for our cluster management messages
    - **TCP and UDP port 7946**: These ports are used for the communication between our nodes
    - **TCP and UDP port 4789**: These ports are used for overlaying network traffic

In summary, our production environment will be composed of the following hosts:

- **Manager node**: This is responsible for all the heavy work of orchestration and scheduling. We will call this machine `manager_01`.
- **Two workers**: These are dummy nodes we will use to host our containers. We will call these hosts `worker_01` and `worker_02`.

As we mentioned before, you need to know the IP addresses of your different Docker hosts and the most important one is the IP address of the manager machine. The workers will be connecting to this IP address to know what they need to do. For example, imagine that our manager host has the `192.168.99.100` IP.

At this point, you are ready to set up your Swarm cluster. First of all, ensure that the Docker Engine is running in all your nodes. Once you have checked that the engine is running in your hosts, you need to enter your `manager_01` node through SSH or console. In your `manager_01` node, run the following command to start the Swarm:

```
docker swarm init --advertise-addr 192.168.99.100
```

The preceding command will initialize Swarm and establish the current node as the manager. It also gives you the commands you need to run to add more managers or to add workers to the cluster. The output of your `init` command should be similar to the following output:

```
Swarm initialized: current node (b33ldnwlqda735xirme3rmq7t) is now a
manager.
```

Once you have your Swarm initialized, you need to get a token to be used to join other machines to the cluster. To get this token, you only need to run the following command at any moment:

```
docker swarm join-token worker
```

Imagine that you need to add a new `worker` to the swarm; in this case, you only need to get the token from the preceding command and run the following one:

```
docker swarm join \

--token SWMTKN-1-12m5gw74y2eo8llzj9oi2b2ij3z3fwwjg830svx3go5pig1stl-
ev5x8obsajn4yhzy774lvhhfu \

192.168.99.100:2377
```

The preceding command will add a `worker` to the Swarm, and if you need to add another manager to this cluster, you can run `docker swarm join-token manager` and follow the instructions.

In theory, the init command started the Swarm; if you want to check the correct cluster initialization, you only need to execute the following command:

```
docker info
```

The preceding `info` command will give you some output similar to the following one; note that we removed some information to fit it in the book:

```
Containers: 44
Running: 0
Paused: 0
Stopped: 44
Images: 171
Server Version: 1.12.5
Swarm: active
NodeID: b33ldnwlqda735xirme3rmq7t
Is Manager: true
ClusterID: 705ic3oomoadlhocvcluszfwz
Managers: 1
Nodes: 1
Orchestration:
Task History Retention Limit: 5
Node Address: 192.168.99.100
```

As you can see, we have our Swarm active and ready; if you want to get more information about the nodes, you only need to execute the following command:

```
docker node ls
```

The preceding command will give you an output similar to the following one.

```
ID                 HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS

b33ldnwlqda... *  moby    Ready   Active     Leader
```

At this point, you have one host, which is your manager node, but you do not have any workers where you can spin up your containers. Let's add some worker nodes to our cluster.

Adding a worker node to our cluster is very easy with Swarm–you only need to access the host through SSH or console and run the command output by your swarm init on the manager node:

```
docker swarm join \

--token SWMTKN-1-12m5gw74y2eo8llzj9oi2b2ij3z3fwwjg830svx3go5pig1stl-
ev5x8obsajn4yhzy774lvhhfu \

192.168.99.100:2377
```

If everything went fine, the preceding command will give you the following output that indicates that the current node was added to the cluster:

```
This node joined a swarm as a worker.
```

Imagine that you didn't save the join token; you can obtain it again from your manager node. You only need to login in the manager node and execute the following command:

```
docker swarm join-token worker
```

The preceding command will give you the command you need to run in your workers nodes again.

As you can see, it is very easy to add nodes to your cluster; add all the remaining workers. To check the status of your cluster nodes, you can execute the following command in your manager node:

```
docker node ls
```

Adding services to our Swarm

At this point, you will have your production environment ready for deployments; let's deploy something to test the new environment. We will deploy a very simple image that does a ping to `google.com`. As soon as you feel comfortable deploying services to the Swarm, you can give it a try and deploy our example application.

Open a connection or login in your manager node and run the following command:

```
docker service create --replicas 1 --name pingtest alpine ping google.com
```

The preceding command creates a new service in the cluster, with the `--name` flag we are assigning a pretty name for our service, in this case `pingtest`. The `--replicas` parameter indicates the number of instances we want for our service; in our example we specified only one instance. With `alpine ping google.com`, we are telling our Swarm which image we want to use (`alpine`) and the command we want to execute in this image (`ping google.com`).

As you can see, it was very easy to deploy the new testing service. If you want to see which services are running in your cluster, login in your manager node and execute `docker service ls`, the output will be similar to the following one:

```
ID            NAME     REPLICAS  IMAGE   COMMAND
3ojsox6wioz4  pingtest  1/1      alpine  ping google.com
```

Once you have your service running in production, at some point you will need to have more information about the service. It is very easy with Docker, you only need to execute the following command in your manager node:

```
docker service inspect --pretty pingtest
```

Your output can be similar to the following one:

```
ID:     3ojsox6wioz45d37xkcgn1xwv
Name:     pingtest
Mode:     Replicated
Replicas:     1
Placement:
UpdateConfig:
Parallelism:     1
On failure:     pause
ContainerSpec:
Image:     alpine
Args:     ping google.com
Resources:
```

If you want the output to be returned as a JSON, you only need to remove the `--pretty` parameter from the command:

```
docker service inspect pingtest
```

The output of the preceding command will be similar to the next one:

```
[
{
    "ID": "3ojsox6wioz45d37xkcgn1xwv",
    "Version": {
        "Index": 23
```

```
        },
        "CreatedAt": "2017-01-07T12:53:18.132921602Z",
        "UpdatedAt": "2017-01-07T12:53:18.132921602Z",
        "Spec": {
            "Name": "pingtest",
            "TaskTemplate": {
                "ContainerSpec": {
                    "Image": "alpine",
                    "Args": [
                        "ping",
                        "google.com"
                    ]
                },
                "Resources": {
                    "Limits": {},
                    "Reservations": {}
                },
                "RestartPolicy": {
                    "Condition": "any",
                    "MaxAttempts": 0
                },
                "Placement": {}
            },
            "Mode": {
                "Replicated": {
                    "Replicas": 1
                }
            },
            "UpdateConfig": {
                "Parallelism": 1,
                "FailureAction": "pause"
            },
            "EndpointSpec": {
                "Mode": "vip"
            }
        },
        "Endpoint": {
            "Spec": {}
        },
        "UpdateStatus": {
            "StartedAt": "0001-01-01T00:00:00Z",
            "CompletedAt": "0001-01-01T00:00:00Z"
        }
    }
    ]
```

As you can see, the JSON format has more information; feel free to use the option that is more suitable for you. In case you need to know where your service is running, you can do a `docker service ps pingtest`, as always, in your manager node.

Scaling your services in Swarm

We showed you how easy it is to create new services in your Swarm cluster, now it's time to let you know how you can scale your services up/down. Go to your manager node and run the following command:

```
docker service scale pingtest=10
```

The preceding command will create (or destroy) the required amount of containers to adjust it to your desire; in our case, we want 10 containers for our `pingtest` service. You can check the correct execution of the command with `docker service ps pingtest`, giving you the following output:

```
ID                         NAME         IMAGE   NODE  DESIRED STATE   CURRENT STATE       ERROR
c8q1as1itf2gqcfs28uvsdsve  pingtest.1   alpine  moby  Running         Running about an hour ago
3b7mt3zdjni1sah6d5g6awitg  pingtest.2   alpine  moby  Running         Running 52 minutes ago
2oenxtgjurdcarwdqg2z5jpwt  pingtest.3   alpine  moby  Running         Running 52 minutes ago
c31uyfod9q7ln5rmb6xo9chnq  pingtest.4   alpine  moby  Running         Running 52 minutes ago
807nc71sqtf1l2pgqo6gzaotm  pingtest.5   alpine  moby  Running         Running 52 minutes ago
191gxqlo1bp05v93ssciju1wq  pingtest.6   alpine  moby  Running         Running 52 minutes ago
1nfein6zumk1vy4074vvhiava  pingtest.7   alpine  moby  Running         Running 53 minutes ago
459cr9tw0nk5xr6rj1g7apq3y  pingtest.8   alpine  moby  Running         Running 52 minutes ago
0fsogsguit2ffueclv6b7spjb  pingtest.9   alpine  moby  Running         Running 52 minutes ago
4jw31cff8umzrayaxksjwlkwm  pingtest.10  alpine  moby  Running         Running 52 minutes ago
```

From the preceding output, you can check that the service is running in 10 containers and also, you can check in which node it is running. In our case, they are all running in the same host as we only added one node to our cluster.

You now know how to create your Swarm cluster and how you can easily start new services, now it's time to show you how you can stop any running service.

Connect to your manager node and execute the following command:

```
docker service rm pingtest
```

The preceding command will remove the `pingtest` service from your Swarm cluster. As always, you can check if the service was stopped with the `docker service inspect pingtest` command or by checking the running containers with the usual `docker ps`.

At this point in the chapter, you will be able to create a Swarm cluster and spin up any service; give it a try and move our example application to use Swarm.

As you can imagine, we like how easy Docker makes your development cycle and how simple the deployment can be, but there are other projects out there that you can use in your production environment. Let's look at the most commonly used ones these days so that you can choose which option is better for your project.

Apache Mesos and DC/OS

Apache Mesos abstracts all the compute resources from machines, enabling fault-tolerant and elastic distributed systems. Creating an Apache Mesos distributed system can be complicated, so Mesosphere created DC/OS, an OS built on top of Apache Mesos. Thanks to DC/OS, you can have all the power of Mesos but it's easier to install or manage.

Some of the features available in Apache Mesos and, of course, in DC/OS are as follows:

- **Linear scalability**: You can scale up to 10,000 hosts without any problems
- **High availability**: Mesos uses Zookeeper to provide a fault-tolerant replicated master and agents
- **Containers support**: Native support for launching containers with Docker and AppC images
- **Pluggable isolation**: Isolation support for CPU, memory, disk, ports, GPU, and modules for custom isolation
- **APIs and Web UI**: The built-in APIs and Web UI allow you to easily manage any aspect of Mesos
- **Cross Platform**: You can run Mesos in Linux, OSX, and even Windows

As you can see, Apache Mesos and DC/OS are an interesting alternative to Docker or Kubernetes. Those projects unify all your resources segregated between all your nodes and transform them into one distributed system. It gives you the impression that you are only managing a single machine.

Kubernetes

Kubernetes is one of the mainstream open source systems used for automating deployment, scaling, and management of your containers. It was created by Google and it has a vibrant community.

It is a full orchestration service and among all the features it has, we can highlight the following ones:

- **Self-healing**: This is an interesting feature that restarts failed containers and replaces and reschedules containers when any of your nodes die. It is responsible for killing unhealthy containers and also avoids advertising containers that are not ready.
- **Service discovery and load balancing**: This feature allows you to forget about creating and managing your own discovery service; you can use what comes out of the box. Kubernetes also gives each container its own IP address and a DSN name for a set of containers. Thanks to all of this, you can do load-balancing easily.
- **Automated rollouts and rollbacks**: There is nothing more critical than rollouts and rollbacks. Kubernetes can manage these actions for you; it will monitor your application to ensure that it continues running smoothly while you are doing a rollout/rollback.
- **Horizontal scaling**: You can scale your application up or down with a single command, using a user interface or even define some rules to do it automatically.
- **Automatic binpacking**: Kubernetes takes care of how to place your containers based on their resource requirements and other constraints. The decision is taken by trying to maximize the availability.

As you can see, Kubernetes comes with most of the features needed in big projects out of the box. For this reason, it is one of the most used systems for containers orchestration. We recommend that you investigate more about this project. You can find all the information you need on the official page. If you have a specific question that you can't find the response in the official documentation for, the big community behind this project can help you.

Deploying to Joyent Triton

Earlier, we showed you how you can build your Swarm cluster. It is an interesting way of managing your infrastructure, but what happens if you need all the power of the Cloud but without the inconvenience of dealing with the orchestration? In the following example, we assume that you don't have the budget or time to set up your Cloud servers with the orchestration software of your choice.

At the beginning of the chapter, we talked about the major Cloud providers and, among all of them, we talked about Joyent. This company has a hosting solution called **Triton**; you can use this solution to create VMs or containers with a single click or an API call.

The first thing you need to do if you want to use their hosting services is to create an account on their `https://www.joyent.com` page. Once you have your account ready, you will have full access to their environment.

Once your account is ready, add an SSH key to your account. This key will be used to authenticate you against your containers and the Joyent's API. If you do not have an SSH key to use, you can create one manually. It is very easy to create a SSH key, for example, in Mac OS you only need to execute the following command:

```
ssh-keygen -t rsa
```

This command will ask you some questions about where the key will be stored or the passphrase you want to use to secure your key. Once you answer all the questions, usually the key will be stored in the `~/.ssh/id_rsa.pub` file. You only need to copy the content of this file to your Joyent's account. If you are using Linux, the process of creating a SSH key is pretty similar.

Once you have your account ready, you can start creating containers; you can do it using their web UI, but in our case we will show you how to do it from your terminal.

We were using Docker and Joyent with the Docker API implemented in Triton, so you will see how easy it is to make your deployments. The first thing you need to do is install the Triton CLI tool; this application was built on Node.js, so you need Node.js (`https://nodejs.org`) installed on your development machine. Once you have node, you only need to execute the following command to install the Triton CLI:

```
sudo npm install -g triton
```

The preceding command will install Triton as a global application on your machine. As soon as Triton is available on your computer, you need to configure the tool; enter the following command and answer all the questions asked:

```
triton profile create
```

At this point, your Triton CLI tool will be ready to be used. Now, it is time to configure Docker to use Triton. Open your terminal and execute the following command:

```
eval $(triton env)
```

The preceding command will configure your local Docker to use Triton. From now on, all your Docker commands will be sent to Triton. Let's try to deploy our example application–go to the location of your `docker-compose.yml` file and execute the next command:

```
docker-compose up -d
```

The preceding command will work like always but, instead of using our development machine engine, it will spin up our containers in the Cloud. One of the advantages of Triton is that they assign at least one IP address to each container, so if you need to get the IP address of a specific container you only need to execute `docker ps` to get all the containers running (in Triton) and their names. Once you have the name of the container, you only need to execute the following command to obtain the IP address from the Triton CLI:

```
docker inspect --format '{{ .NetworkSettings.IPAddress }}' container_name
```

The preceding command will give you the IP address of the container you have chosen. Another way of obtaining the IP is from the web UI.

A docker-compose stop will kill all the containers deployed to the Cloud.

Now you can use everything you have learned in the book and deploy to your Joyent Triton Cloud without too many problems. This is probably the easiest way of deploying your Docker containers in the market right now. Working with Joyent Triton is like working in local.

Be aware that this is not the only option you have to deploy Docker in production; we only showed a simple and easy way. You can try other options, such as CoreOS and Mesosphere (DC/OS), among others.

What is DevOps?

DevOps is a set of practices that emphasizes the collaboration and communication between development and operations (IT). The main goal is to establish a culture of a rapid, frequent, and more reliable way of releasing software. To achieve this goal, the DevOps usually try to automate as much as they can. If your project (or company) grows, at some point, you will put some DevOps principles in place to secure the future of your application.

Some of the technical benefits of adopting this culture in your organization can be as follows:

- Aims to maximize the use of **Continuous Integration** and **Continuous Delivery**
- Reduces the complexity of the issues to fix
- Reduces the number of failures
- Provides a faster resolution of problems

The main pillar of DevOps is the culture of communication between all parts involved in the development of your application, especially between development and operations guys. It doesn't matter how nice or amazing your application is, a lack of communication inside your organization can shut down your entire project.

Once your organization has a really good communication channel between all the parts involved in the development of your software, you can analyze and put the next pillar of the DevOps culture–the automation–in place. To create a reliable system, you need to invest in the automation of repetitive manual tasks and processes; this is where continuous integration and continuous delivery come in. Creating your CI/CD pipeline will help you automate repetitive tasks like unit tests or the deploy, improving the overall quality of your application. It will even help you save some time in your day-to-day tasks. Imagine that deploying your application manually to your production environment takes an average of 8 minutes each time; if you deploy at least once a day, you will be wasting more than 30 hours in a whole year only making deploys.

DevOps is all about your application and the process around the development and deployment, and the core principles are as follows:

- Agile software development
- Continuous integration
- Continuous delivery pipelines

- Automated and continuous testing
- Proactive monitoring
- Improved communication and collaboration

As you can see, the DevOps culture is not something you can implement in a few hours in your organization, it is a long process in which you need to analyze how the development process in your organization works and the required changes you need to make to find a more flexible and agile way. We covered most of the core principles in this book; it is now up to you to fill the gaps and implement a DevOps culture in your organization.

Summary

In this chapter, we talked about what a Cloud is and what you need to know to choose your hosting provider. We also told you about the different options you have to orchestrate your application into the Cloud. Now, it is your turn to analyze and choose the best option for your project.

Index

.

.gitignore file 235

=

=== operator
  using 329

A

Acceptance Test-Driven Development (ATDD)
  about 106
  advantages 106
  ATDD algorithm 108
  user stories 107
Access Control List (ACL)
  about 225
  advantages 226
  using 226, 227, 228
Amazon 13
Amazon Web Services (AWS) 335, 336
Ansible
  about 242
  installation 242
  requisites 242
Ansistrano
  about 242, 243
  application, deploying with 245, 247
  workflow 243
Apache JMeter
  features 292
  installing 293
  URL 293
  used, for load testing 292, 294, 295, 296, 297, 299, 300, 301
Apache Mesos
  about 340
  features 347
API gateway
  about 71
  benefits 72
API tests 98
application deployment
  DC/OS 347
  to Cloud 339
  to Joyent Triton 349, 350
  with Apache Mesos 347
  with Docker Swarm 339, 340
  with Kubernetes 348
application development
  environments, building 16
  lightweight communication protocol, using 15
  network latency 15
  on microservices 14
  queues, using 15, 16
  repositories, handling 16
  scalability 15
  small logical black boxes, creating 14
  worst case scenario, handling 16
application logs
  about 188
  challenges, in microservices 189
  ERROR 189
  in Lumen 190
  INFO 189
  WARNING 189
application monitoring
  about 191
  at application level 192
  at infrastructure level 194
  by levels 192
  hardware/hypervisor, monitoring 200
  with Datadog 193
  with Prometheus 195, 197
  with Weave Scope 198, 199

Application Performance Monitoring (APM) 192
application structure
  Battle 134
  Location 134
  Secrets 134
  User 134
application
  Access data layer 260
  Business logic layer 260
  Presentation layer 260
Artillery
  advanced scripting 306
  installing 302
  load tests, executing 303, 304
  scripts, creating 305
  URL 307
  used, for load testing 302
assertion 114
async 164, 165
ATDD algorithm
  about 108
  Demo 109
  Develop 109
  Discuss 108
  Distill 109
authentication
  about 211, 212, 213, 214
  JSON Web Token (JWT) 211, 219
  OAuth 2 211, 215
autodiscovery service 47
availability math 289, 291

B

backup
  about 252
  differential backup 254
  for files 253
  full backup 253
  importance 252
  incremental backup 254
  location, selecting 253
  restoring 256
  strategies 252
  types 253
  validating 256
backups, tools
  about 254
  Bacula 254
  custom script 256
  Percona xtrabackup 255
Bacula
  about 254
  director 255
  file 255
  storage 255
Bamboo 93
Behat
  about 95, 128
  example 129
  installation 128
  test execution 128
behavior-driven development (BDD)
  about 21, 104
  Cucumber 105
  features 104
Berkeley Software Distribution (BSD) 20
beta channel 26
BitBucket
  about 43
  URL 43
blog platform
  backend, dividing 271, 275
  extraction services 275, 278, 280, 281, 284
  frontend, dividing 270, 271, 275
  new microservices functionality, adding 264, 265, 266, 268
  transforming, from monolithic to microservices 263, 264
Blue/Green deployment 249
Bower
  about 239
  bower install command 240
  bower install package-name command 240
  bower update command 240
breakpoint 173
BUS 90
Business process management (BPM) 10

C

cache 77
CACHE layer 24
caching, best practices
  about 315
  cache algorithm, selecting 316
  cache miss, handling 315
  cache, monitoring 316
  elements size, considering 316
  performance impact 315
  requests, grouping 316
caching
  about 77, 167, 168, 169, 171
  HTTP caching 80, 81
  Least Frequently Used (LFU) 77
  Least Recently Used (LRU) 77
  Most Recently Used (MRU) 77
  static files, caching 82
  strategy 77, 78, 80
Canary releases 250
capacity planning
  about 285
  application limits, identifying 286, 287, 289
  availability math 289, 291
  benefits 285
CentOS
  siege, installing 307
Cloud Native Computing Foundation 195
Cloud provider
  Amazon Web Services (AWS) 335, 336
  considerations 335
  DigitalOcean 337
  Google Compute Engine 338, 339
  Joyent 338
  Microsoft Azure 336, 337
  Rackspace 337
  selecting 335
Cloud, advantages
  autoscalable 333
  cheaper 334
  elastic 333
  growth 334
  lower management efforts 334
  time to market 335
Cloud
  about 332
  application, deploying 339
code versioning
  accept header 315
  best practices 314
  custom request header 314
  URL 314
code, best practices
  === operator, using 329
  database, connecting 329
  database, querying 329
  regular expressions, writing 328
  single quotes, versus double quotes 328
  spaces, versus tabs 328
  strings, dealing with 327
Codeception 95
Common Gateway Interface (CGI) 17, 60
Community ENTerprise Operating System (CentOS) 29
Composer require-dev 234
Composer
  about 110, 134, 234
  advantages 236
  disadvantages 236
  executing, in production 236
Consul 73
ContainerPilot
  about 73
  URL 52
Content Delivery Network (CDN)
  about 82
  Pull CDNs 82
  Push CDNs 82
  using 322
continuous delivery (CD), tools
  Behat 95
  Codeception 95
  PHPSec 95
  PHPUnit 95
  Selenium 95
continuous delivery (CD)
  about 93
  benefits 94
  workflow 94

continuous integration (CI), tools
Bamboo 93
Jenkins 93
PHP CI 93
Travis 93
continuous integration (CI)
about 91, 92
benefits 92
tools 92, 93
Continuous Integration/Continuous Delivery (CI/CD) 20
continuous integration
with Jenkins 248
Cucumber 105
custom error handling
about 181
render method 182, 183
report method 182
custom script
using 256

D

DATA STORE 25
data URIs
using 320
database encryption
about 202
InnoDB encryption 205, 206
performance overhead, reducing 206
database operations
performing 149, 150, 151, 153, 155, 156, 157, 158
database per service
about 73
benefits 73
drawbacks 74
Datadog 193
DC/OS 347
Debian
siege, installing 308
debugging
about 172
in PHP, with Xdebug 173
dependency management
.gitignore file 235
about 134, 234, 322
Composer require-dev 234
deployment workflow 235
disadvantages 323
frontend dependencies 236
vendor folder 235
deployment tools
Bamboo 248
Capistrano 248
Chef 247
Codeship 248
Nomad 248
Packer 248
Puppet 248
Terraform 248
Travis CI 248
deployment workflow
about 235
Composer, executing in production 236
vendor folder, on repository 235
deployment, advanced techniques
about 248
Blue/Green deployment 249
Canary releases 250
continuous integration, with Jenkins 248
immutable infrastructure 252
deployment, automation
about 240
Ansible 242
Ansistrano 242
benefits 240
PHP script, adding 241
development environment
autodiscovery service 47, 48
NGINX 48, 49, 51, 54
PHP-FPM 48, 49, 51, 54
setting up, for microservices 45, 46
DevOps
about 351, 352
benefits 351
core principles 351
die() function 180, 181
differential backup 254
DigitalOcean 337
Distributed Version Control Systems (DVCS) 39

Docker Engine 339
Docker files
  URL 46
Docker Swarm
  about 339
  features 339, 340
  installing 340, 341, 342, 343
  services, adding 343, 344
  services, scaling 346, 347
Docker Toolbox
  versus Docker for Mac 27
Docker, for Linux
  Docker group, creating 30
  installing, with yum 29
  on CentOS/RHEL 29
  requisites 29
Docker, for Mac
  installation process 27, 28
  requisites 27
  URL 27
  versus Docker Toolbox 27
Docker, for Ubuntu
  common issues 32
  DNS server 33, 34
  Docker group, creating 34
  installing, with apt 31, 32
  requisites 31
  starting, on boot 34, 35
  UFW, forwarding 33
Docker, for Windows
  Docker compose 36
  Docker tools, installing 35, 36
  Git 36
  Kitematic 36
  requisites 35
Docker
  beta channel 26
  compose version, checking 37
  Docker engine version, checking 37
  installation, checking 38
  installing, on Linux 29
  installing, on macOS 27
  installing, on Ubuntu 30
  installing, on Windows 35
  machine version, checking 37
  management tasks 38
  stable channel 26
domain-driven design (DDD)
  about 83
  context 83
  domain 83
  model 83
  process 84, 85, 86
  ubiquitous language 83
  using, in microservices 86
double quotes
  versus single quotes 328

E

eBay 12
ELB 250
encryption
  about 201, 202
  database encryption 202
  in MariaDB 203, 204, 205
  TSL/SSL protocols 206
End Of Life (EOL) 20
end-to-end tests 98
Enterprise Service Bus (ESB) 10
environment variables
  using 228, 229
error handling
  about 158, 179, 325
  challenges 180
  custom error handling 181
  exceptions, managing 161, 162, 163
  HTTP status codes 325
  need for 179, 180
  validation 158, 159, 160
  with die() function 180, 181
  with Sentry 183, 187
Etag header tag 80
event consumers 88
event-driven architecture (EDA)
  about 87
  advantages 88, 91
  in microservices 88, 89, 90
EVENTS SERVICE API 90
exceptions
  about 162

managing 161, 163
Extended Page Tables (EPT) 27

F

Fabio 73, 250
feature branch workflow 44
forking workflow 44
framework
about 56
characteristics 65
Lumen 66, 67
middleware 61, 62, 63
Phalcon 65
PHP Framework Interoperability Group (PHP-FIG) 56
PHP Standard Recommendation 7 (PSR-7) 57, 58
Silex 67
Slim 66
Zend Expressive 67
frontend dependencies
about 236
Bower 239
Grunt 237
Gulp 238
SASS 238
full backup
about 253
advantages 253
disadvantages 253

G

Gherkin 105
Git
about 39
advantages 40
using 41, 42
versus SVN 39
workflow 40
Gitflow workflow 44
GitHub
about 42
URL 43
glue code 259
GOOGLE 213
Google Compute Engine 338, 339
Grunt
about 237
InitConfig 237
LoadNpmTask 237
RegisterTask 237
URL 237
Gulp 238
Guzzle
used, for microservices call 146, 147, 148

H

hardware/hypervisor
monitoring 200
headroom
about 288
CurrentUsage 288
Growth 288
IdealUsage 288
MaxCapacity 288
Optimizations 288
hosting
about 42
with BitBucket 43
with GitHub 42
HTTP caching 80, 81
HTTP headers, options
max-age 81
must-revalidate 81
no-cache 81
no-store 81
no-transform 81
private 81
proxy-revalidate 81
public 81
s-maxage 81
HTTP headers
Cache-control 80
Expires 80
Last-modified 80
Pragma 80
HTTP methods
DELETE 136
GET 136
PATCH 136

POST 136
PUT 136
HTTP status codes
client request incomplete 326
client request successful 325
request redirected 326
server errors 327
Hypertext Preprocessor (PHP) 17
Hypervisor Framework 27

I

image optimization
about 319
bad requests, avoiding 322
Content Delivery Networks (CDNs), using 322
data URIs, using 320
data, caching 321
images, scaling 320
lossless image compression, using 319
sprites, using 319
immutable infrastructure
using 252
incremental backup
advantages 254
disadvantages 254
InnoDB encryption 205, 206
integration testing 97

J

Jenkins 93
continuous integration 248
Joyent 338
Joyent Triton
about 349, 350
URL 349
JSON Web Token (JWT)
about 201, 219
implementing 224, 225
setting up 220, 221, 222
using, on Lumen 219, 220

K

Kubernetes
about 340, 348
features 348

L

Linux
Docker, installing 29
Load Balancer (LB) 249
load testing
about 292
with Apache JMeter 292, 294, 295, 296, 297, 299, 300, 301
with Artillery 302, 303, 304
with siege 307
logs 188
lossless image compression
GIF 319
JPG 319
PNG 319
using 319
Lumen
about 66, 67
application logs 190
JSON Web Token (JWT), using 219, 220
OAuth 2, using 215
URL 66, 67
URL, for documentation 159

M

macOS
Docker, installing 27
MariaDB
encryption 203, 204, 205
memcached 78
Memory Management Unit (MMU) 27
microservice call
implementing 142, 145
request life cycle 145
with Guzzle 146, 147, 148
microservices, use cases
Amazon 13
eBay 12
Netflix 12
Spotify 13
Uber 13
microservices
API gateway 71, 72
application development 14

application logs, challenges 189
architecture 24, 25
Battle service 70
characteristics 11, 12
design 24, 25
design pattern 71
disadvantages 13, 14
Docker, installing 26
domain-driver design (DDD), using 86
event-driven architecture (EDA) 88, 89, 90, 91
Location service 70
PHP, using 16
requisites 25, 26
Secret service 70
structure 69, 70
testing 97
User service 70
versus monolithic application 7, 8, 9, 10
versus Service Oriented Architecture (SOA) 10, 11
Microsoft Azure
about 336, 337
advantages 336
middleware
about 61
example 62, 63
using 141, 142
migration 150
Mink project
URL 133
model domain 83
Model-View-Controller (MVC) 6, 270
monitoring 230
monolithic application
versus microservices 7, 8, 9, 10
Monolog 190

N

Nagios 200
Netflix 12
NGINX 73
about 24, 48, 49, 51, 54
TSL/SSL protocols, using 209, 210, 211
Node.js
URL 302, 349
number of nines 290

O

OAuth 2
about 215
implementing 217, 218, 219
installation 215
setting up 216, 217
using, on Lumen 215
object-oriented (OO) programming 18
Object-Relational mapping (ORM) 154
OpenStack 337
Operating System (OS) 25

P

Percona 150
Percona xtrabackup
about 255
advantages 255
performance optimization, best practices
about 317
CSS, minimizing 318
HTML, minimizing 318
HTTP requests, minimizing 317
image optimization 319
JavaScript, minimizing 318
Phalcon
about 65
URL 66
PHP CI 93
PHP Data Objects (PDO) 18
PHP Framework Interoperability Group (PHP-FIG)
about 56
URL 57
PHP Next-Gen (PHPNG) 20
PHP Standard Recommendation 7 (PSR-7)
about 57, 58
headers 58
host header 59
messages 58
request target 59, 60
server-side requests 60
streams 59
uploaded files 60, 61
URIs 59, 60

URL 61
PHP Standards Recommendation (PSR) 57
PHP, versions
about 18
version 4.x 18
version 5.x 18
version 6.x 18, 19
version 7.x 19
PHP-FPM (FastCGI Process Manager) 48, 49, 51, 54, 110
PHP
advantages 19, 20, 21
debugging, with Xdebug 173
disadvantages 21
history 17, 18
profiling, with Xdebug 173
URL 19
URL, for documentation 19
using, on microservices 16
PHPSec 95
PHPUnit
about 95, 110
assertArrayHasKey 114
assertArraySubset 116
assertClassHasAttribute 115
assertClassHasStaticAttribute 116
assertContains() 117
assertDirectory() 117
assertFile() 117
assertions 114
assertJson() 119
assertRegExp() 119
assertString() 118
boolean assertions 120
other assertions 120, 121
tests, executing 113
type assertions 120
unit testing 112, 121, 122, 124
playbooks 242
Postman
URL 139
using 139, 141
profiling
about 172, 173
in PHP, with Xdebug 173
Prometheus
about 195, 197
features 195

Q

QUEUE system 25
queue
using 164, 165

R

Rackspace 337
Red Hat Enterprise Linux (RHEL) 29
Redis 78, 164
refactor strategies
about 258
backend, dividing 260, 261
extraction services 262
frontend, dividing 260, 261
microservices, creating with new functionality 258, 259, 260
module, extracting 262, 263
registry 72
regular expressions
writing 328
release branches
about 329
benefits 329
example 330
render method 182, 183
report method 182
Representational State Transfer (REST) 74
request life cycle 145
requests per second (RPS) 288
RESTful API
consumer amenities 76
conventions 74, 75
standards 75, 76
REVERSE PROXY 24
RHEL
siege, installing 307
router 259
routing
about 136, 139
middleware, using 141, 142
Postman, using 139, 141

S

SassScript
  indented syntax 239
scalability plan
  about 309
  application infrastructure, checking 313
  application, developing for production 312
  caching layers, adding 312
  load pattern, reviewing 313
  requisites, gathering 311
  storage layer, moving to microservice 313
  usage pattern, reviewing 313
Secure Sockets Layer (SSL) 206
security, best practices
  about 230
  cross-site scripting XSS 231
  directory traversal 233
  file permissions, setting 230, 231
  ownership, assigning 230, 231
  password policies, creating 232, 233
  passwords, storing 232
  PHP execution locations, specifying 231
  remote files, using 232
  session, hijacking 232
  source code revelation 233
  SQL injection 231
  user trust, avoiding 231
Selenium
  about 95, 132
  Selenium IDE 133
  Selenium WebDriver 133
semantic versioning, examples
  bugs, fixing 324
  features, adding to project 324
  source code, modifying 325
semantic versioning
  about 45, 323
  URL 45
  X (major) tag 324
  Y (minor) tag 324
  Z (patch) tag 324
Sentry
  used, for error handling 183, 187
service discovery, tools
  Consul 73
  ContainerPilot 73
  Fabio 73
  NGINX 73
service discovery
  about 72
  shared database 73, 74
Service Level Agreements (SLA) 336
Service Oriented Architecture (SOA)
  versus microservices 10, 11
shared database
  about 73, 74
  drawbacks 74
siege
  example 308
  installing, on CentOS 307
  installing, on Debian 308
  installing, on other OS 308
  installing, on RHEL 307
  installing, on similar operating systems 307
  installing, on Ubuntu 308
  used, for load testing 307
Silex
  about 67
  URL 67
single quotes
  versus double quotes 328
Slim
  about 66
  URL 66
SmartOS 338
source code
  environment variables, using 228, 229
  external service, using 229
  security 228
spaces
  versus tabs 328
Spotify 13
sprites
  using 319
stable channel 26
static files
  caching 82
Story Test-Driven Development (STDD) 106
subversion (SVN)

about 39
versus Git 39
Syntactically Awesome Stylesheets (SASS) 238

T

tabs
versus spaces 328
TDD algorithm
code, executing 103
following 102
test, refactoring 103
unit tests, writing 102
test-driven development (TDD)
about 21, 98
advantages 100, 101
performing 99, 100
TDD algorithm 102
test, improving 99
tests, executing 99
unit test, writing 99
testing
API tests 98
end-to-end tests 98
importance 96
in microservices 97
integration testing 97
unit testing 97
Thread Group
options 296
tools
Behat 128
Composer 110
PHPUnit 110
Selenium 132
using 110
tracking 230
Traefik 250
transport layer algorithms
Authentication feature 201
Integrity feature 201
Transport Layer Security (TSL) 206
Travis 93
Triton 349
TSL/SSL protocols
about 206
termination 208, 209
using 207, 208
with NGINX 209, 210, 211

U

Uber 13
ubiquitous language 85
Ubuntu
Docker, installing 30
siege, installing 308
Uncomplicated Firewall (UFW) 33
unit testing 97

V

vendor folder
about 235
advantages 235
disadvantages 236
version control, strategy
about 43
centralized work 43
feature branch workflow 44
forking workflow 44
Gitflow workflow 44
semantic versioning 45
version control
about 39
centralized version 39
distributed version 39
Git 39, 41, 42
Git, versus SVN 39
hosting 42
Virtual Machine (VM) 27

W

Weave Scope 198, 199
Windows
Docker, installing 35

X

Xdebug
installing 173, 175
output file, analyzing 178
output, debugging 176, 177

profiling, installation 177
profiling, setting up 178
setting up 175, 176
used, for debugging in PHP 173
used, for profiling in PHP 173

yum
used, for installing Docker on Linux 29

Z

Zabbix 200
Zend Expressive
about 67
URL 67