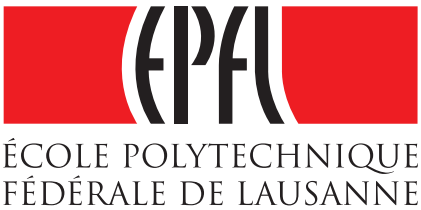


Master's Project

Cloud-Native Implementation of a Microservice Architecture

Tinh Di David Tang
david.tang@alumni.epfl.ch

Supervised by

Philippe Cuvecle
philippe.cuvecle@elca.ch

ELCA Informatique SA

Prof. Willy Zwaenepoel
willy.zwaenepoel@epfl.ch

Operating Systems Laboratory, EPFL

March 17, 2017

Abstract

Microservice is a very promising architecture by providing flexibility and resiliency for cloud-native applications. A microservice application is composed of many small, independent, scalable and fault-tolerant services. This project focuses on developing services with the Java Spring framework and deploying them on OpenShift Origin.
The goal is to explore which design patterns should be applied on such application and how to use the existing technology to implement them. The build and deployment processes should also be automated.
Some possible solutions are implemented and reviewed such as Docker for the virtualization, the Elastic stack for the monitoring, Hystrix for the circuit breaker and Zuul for the API gateway. Custom implementations are also developed when the existing products do not fit the needs.
In the final implementation, each service is packaged into a Docker container with some agents which send the logs and metrics to a centralized monitoring system. The services communicate with each other by using REST and authenticate with JWT.

Acknowledgements

First of all, I want to gracefully thank the following people for their various helps during my Master's project:

- **Philippe Cuvecle**: for his very helpful guidance and multiple supports during the whole project. Our multiple discussions about the architecture was also very interesting.
- **Willy Zwaenepoel**: for accepting to supervise this project and following my progress during these six months.
- **Mathieu Verbaere and Christian Gasser**: for proposing and defining the subject of this project.
- **Fabien Guillaume**: for proposing and managing the Libero project.
- **Nicolas Glayre**: he deploys a stable version of OpenShift Origin and helps me to fix the previous deployment.
- **Florian Bois, Jonathan Rohrbach, Laurent Valette and Pedro Dos Santos**: they helped me to understand Docker, the continuous integration process and OpenShift Origin.
- **Jennifer Veillard, Nicolas Plancherel and Guillaume Guenat**: for reviewing this report.

Contents

Part I

Concepts & Challenges

Chapter 1

Introduction

Nowadays, we are moving the resources on the cloud. Our devices become more portable and need a connection to the network in order to work. We store our documents on a remote backup service, we watch streaming movies, we listen to streamed music and we even play streamed games. Our local devices are becoming a simple interface to the cloud.
Mobility is one reason of this change: we want to be able to work and to be entertained everywhere without physically bringing everything with us. Durability might be another one: we don't want to lose our data if we lose our devices. Everything is synchronized on the cloud and the providers ensure the availability and durability for us.
To respond to this demand, we should rethink the way we develop applications. Our application should provide scalable, on demand, resilient and highly available services.

1.1 Cloud Computing

Cloud computing is the uses of some remote resources (compute or data) through the network instead of locally. These resources are usually on-demand and pay-as-you-go. You delegate the responsibility to the underlying system and this also gives you much more flexibility.
From a company's point of view, the cloud computing also provides:

- **Cost reduction**: the remote resources might be shared to optimize the utilization the hardware.
- **Faster time to market**: applications are delivered faster.

1.1.1 Cloud-Native

A cloud-native system is designed to take advantage of the cloud computing and should have the following properties:

- **Containerized**: applications of the system run in some containers (more information in section 4.1) to provide flexibility and isolation.
- **DevOps**: developers are deeply involved in the deployment of their applications (see chapter 5).
- **Microservices**: applications are independent and loosely-coupled together (see chapter 2).
- **Orchestrated**: applications are scheduled and managed by a central process to optimize the resources (see section 4.2).

1.1.2 Cloud Computing Service Models

From the Service Oriented Architecture (see section 1.2.2) philosophy, we want to have Everything as a Service. This allows us to abstract the underlying layers and to rely on the service-level agreement (SLA) of the provided service. Here are some service models:

- **Infrastructure as a Service (IaaS)**: provides self-provisioning (virtual) machines. No need to order and configure physical machines.
- **Platform as a Service (PaaS)**: runs and monitors applications in the appropriate environment (programming language, libraries, ...). The user gives the source code or the executable and the provider executes it.
- **Software as a Service (SaaS)**: configures and manages applications. The user logs in to use the software.
- **Database as a Service (DBaaS)**: a specific SaaS which provides database managements. No need to install and setup the database anymore.

Note that a Saas might runs on a PaaS and this latter might run on a IaaS.

1.2 Software design

When you begin to build your new software application, you will choose willingly or not a software design. Choosing the right design at the beginning can help you to save a lot of time. To choose the correct design according to your project, you should consider your needs, the organization of your team, your existing infrastructure, ...

1.2.1 Monolithic

A typical software project starts in a monolithic way by, for example, using the Model–view–controller or the hexagonal (also known as Ports and Adapters) architecture. Monolithic means that the whole application runs in one process or the system is not distributed.
This approach is very intuitive and easy to setup. This might be a good choice for a small project but if the project grows, you will be face to some issues, such as:

- **Maintainability**: many modules will be added over the time. The complexity of the project will grow exponentially and it would be very difficult for new developers to understand the project.
- **Scalability**: there are roughly two ways to scale an application: either vertically (deploy on a more powerful host) or horizontally (deploy on more hosts). A monolithic application might only be able to scale vertically (where the cost grows exponentially) because it would more likely be stateful. If it can be scaled horizontally, maybe behind a sticky session load balancer, you will have to scale the entire application even if only a tiny module was under pressure.
  It is also quite difficult to scale on demand because you have to buy hardware components and it is not easy to have more resources only for a short period.

1.2.2 Service-Oriented Architecture

When some companies merge together, they also need to merge their information systems without too much effort (too much changes in one or many of them). Service-oriented architecture (SOA) is a design which can solve this kind of problem, by roughly encapsulating the service provided by one system to the others. Over the time, companies also refactor their system following a service-oriented pattern to provide flexibility.
More specifically, a service-oriented architecture is a software design which is composed of autonomous, loosely-coupled, abstracted and reusable services which communicate to each other through the network. SOA is usually implemented with an enterprise service bus (ESB) which is the common communication bus between the producer (server) and consumer (client) services. A service also usually represents a business activity (sales, accounting, ...)

1.2.3 Microservice Architecture

Microservice architecture (MSA) is a specialization of SOA and reuse the notions of services (we will more specifically define what is a service in section 2.1), the services are more finely-grained

and they communicate to each other with less constraints (no shared data model and common communication channel). We will see more in details this architecture in chapter 2.
Note that if not specified, the term of service will refer to a microservice in the following chapters.

1.3 Related Work

The most referenced document about microservices is the book from Sam Newman [1]. This book covers a lot of things from the team organization to the deployment passing by the implementation and the infrastructure. This project does not cover as much topics (e.g. less on the human aspect and how to split an existing monolithic application), but we are more focusing on the practical aspect: which design patterns need to be applied and which currently available technologies can be used.
Martin Fowler has also published an article related to microservices [2]. It gives a good definition of the microservice architecture but this is also a quite theoretical view.
NGINX[1] has also published some white papers on this subject such as [3]. They are sometimes quite focused on how to use their products in a microservice application.

1.4 Goal & Scope

The main goal of this project is to collect and study the current practices in order to implement and deploy a cloud-native microservice application.
Given the flexibility that a microservice architecture offers (due to the decentralized governance), the implementation of such an architecture can be done in many ways depending on the needs and constraints.
This project is primarily going to focus on developing services in Java using the Spring framework. We are hence also focused on the backend services because the frontend is a quite different domain which has distinctive problematics, languages and frameworks.
The choice of Java for the backend services is not anodyne because it is one of most the popular language in the microservice world (e.g. Netflix has open-sourced some of their products for microservices in Java). Spring is probably the most popular Java framework and Spring Cloud is microservice-oriented.

[1] NGINX: `https://www.nginx.com/`

Chapter 2

Microservice Architecture

2.1 Microservice Design

We will use the following definitions from an article of James Lewis and Martin Fowler [2]:

- **Component**: an independently replaceable and upgradeable unit of software.
- **Library**: in-process component called using function calls.
- **Service**: out-of-process component which communicates with a mechanism such as web service request or remote procedure call.

Applications are usually split into components. In a monolithic application, the components are libraries, but in a microservice one, they are services. This allows modifying a component without redeploying the entire application.
Moreover, two characteristics of a service to keep in mind are:

- **Loose coupling** between services: the services are independent and see other services as black boxes.
- **High cohesion** within services: the change of a service should not impact the others.

We can also say that a microservice should follow the Unix concept of "Do One Thing and Do It Well" (DOTADIW) from Douglas McIlroy.

2.1.1 The Twelve-Factor App

A good starting point to define the properties of a service seems to be the twelve-factor app [4]. Each one of our services should hence follow this methodology which is defined by some developers from the popular PaaS Heroku[1]. The goal is to define the needed properties for a service which is scalable, portable, easy to deploy and agile.

1. **Codebase**: each app is stored in one codebase (tracked in a version control system) and each codebase contains only one app. Use the same codebase for many deploys (testing, production, staging, ...). Many codebases can form a distributed system. When apps have common code, it should be packaged into a shared library.
2. **Dependencies**: use a packaging system to manage the dependencies. They are declared in a manifest and they should be isolated when the app runs. This latter should be self-sufficient to install the needed dependency tools
3. **Config**: the configs are all variables which can vary between deploys (links to backing services, credentials, etc.). We can for example have some external config files to set these variables, but the best practice is to set them through environment variable, because this method does not depend on the language nor the operating system.

[1] Heroku: `https://www.heroku.com/`

4. **Backing services**: a backing service is another service (e.g. data store, message broker or another app) connected through the network. The links (URL, id) to these services should be stored in the config and we would be able to change them without changing the code.
5. **Build, release, run**: the codebase becomes a deploy after three distinct stages:
   (a) **build stage**: take the code at a specified commit, fetch the required dependencies and compile into an executable bundle (i.e. build)
   (b) **release stage**: take the build, combine it with the config and get the resulting release.
   (c) **run stage**: finally run the app by launching processes against the release.

   Each release should have an ID and is immutable. We should be able to roll back to a previous one thanks to the deployment tool.
6. **Processes**: the app is executed by one or many processes. Each of these process is stateless and share nothing directly between them. All the persisted data should be stored in some stateful backing service and sticky sessions should not be used. Each request can be served by a non-deterministic process and they all should be able to handle the request.
7. **Port binding**: a web app should be self-contained and is not executed in a webserver (e.g. Tomcat[2]). The web app binds to a defined port and waits for incoming requests. This can be generalized with other app than web apps.
8. **Concurrency**: the processes should be inspired from the Unix process model. The processes should never be deamonized but managed by a process manager which can handle the output streams and the states (crash, restart, shutdown).
9. **Disposability**: the processes should be disposable. The startup time should be take a few seconds. When receiving the SIGTERM signal, the process should shut down gracefully: stop accepting new requests and finish the current one or also discard it by sending back a NACK. Note that the process should also be able to handle sudden shut down by returning the job to the queue when the client does not respond.
10. **Dev/prod parity**: the app should be designed for continuous deployment by keeping the development and production as similar as possible. These three gaps should be as small as possible:
    (a) **time gap**: the freshly developed code should be deployable in production in hours.
    (b) **personnel gap**: the developers should also deploy their app or should be closely involved in the deployment.
    (c) **tools gap**: the environment should be the same. Nowadays, it's easier to have a real database, caching or queuing system (i.e. the same as in production) for the development thanks to packaging systems or virtual environments.
11. **Logs**: Logs should be treated as streams and not as files because it is a continuous flow of information. The app should not be concerned on the storage or routing of the logs, but simply write to the standard console output. This makes the development easier and in production, the process manager can route all these streams to a log router which processes and forwards them to a centralized logs storage.
12. **Admin processes**: we should be able to do punctual administrative or maintenance tasks on running app to migrate database, run a console (i.e. REPL shell) or scripts.

2.1.2 Granularity of a Microservice

While defining the services of our microservice system, we will wonder what is the right size of a service.
The number of lines of code is not a good indicator, because it depends on the programming language, framework, etc.
It is not possible to have a strict definition to know when a service should be split, but you should consider it when:

[2]Tomcat: `http://tomcat.apache.org/`

- the service is in two bounded contexts from the domain driven design (DDD).
- the service handles many representations of the same resource.
- a part of the service scales differently than another. For instance, a pump (which fills the database with some external data in a regular interval) is separated from the service (which expose the data and might need to be scaled).
- the service is developed by two distinct groups of people.

You should be aware of splitting too much your services because it will be difficult to manage many them. If the following cases happen, it might be better to merge them:

- modifying a service often implies modifying another.
- two services should be deployed together.
- the services share the same representation of a model.
- the services are developed by the same few people.

The team size should also be limited by the "two-pizza team" rule by Jeff Bezos (founder of Amazon.com) which states that it should always be possible to feed a team with two pizzas.
To conclude, Sam Newman says in his book [1] that a microservice should be "small enough and no smaller".

2.1.3 Data Management

2.1.3.1 Data Persistence

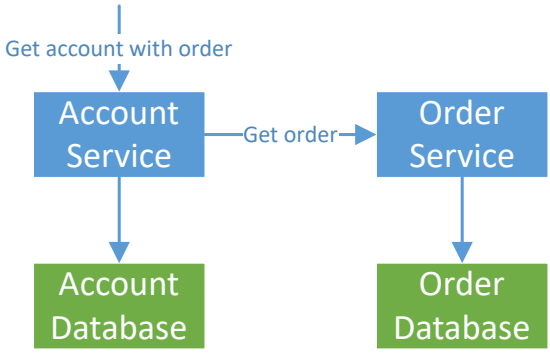
Applications typically need to persist data and the way of doing it is to store them in a database. A monolithic application usually uses only one logical database, but in the case of microservice, it is pointless to have a distributed system which connects to a unique database because the services will wait on the database.
So, we also have to distribute the storage of the resources among many independent databases. Each database should store only one kind of resource (i.e. has only one table) and are usually abstracted by a service.
Here are some ways to split a database (illustrated in figures 2.1):

- **Foreign Keys**: if an item contains a foreign key of another one, the mapping between the key and the item is no longer done in the database level, but in the service level.
- **Shared Static Data**: in the case of static data (e.g. list of countries), many services can read directly on the database.
- **Shared Data**: if the data is not read-only, the database should be abstracted by a service and other services perform the requests only to this service through an API instead of directly to the database.
- **Shared Table**: if a table has different fields which are used by different services, this table should be split and the data are linked with a foreign key. For instance, separate the price and the stock of a product.

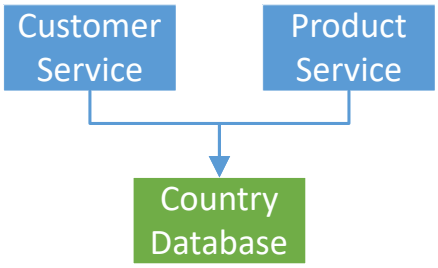
2.1.3.2 Data Representations & Access

A data might have different representations (i.e. views from different domain) between or even within applications.
The databases should only be accessible through an API exposed by a service (a.k.a. the database per service pattern). Here are two patterns to expose this API:

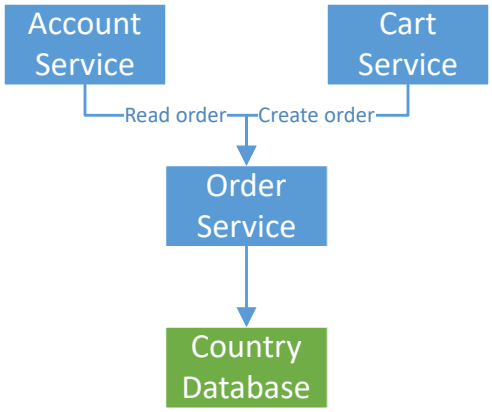
- **Create, Read, Update and Delete (CRUD)**: the service exposes a unique representation for the four operations.
- **Command Query Responsibility Segregation (CQRS)**: the read and write representations of the data are separated. You may have one command service which is dedicated to update the data and some other query services which exclusively read the data.



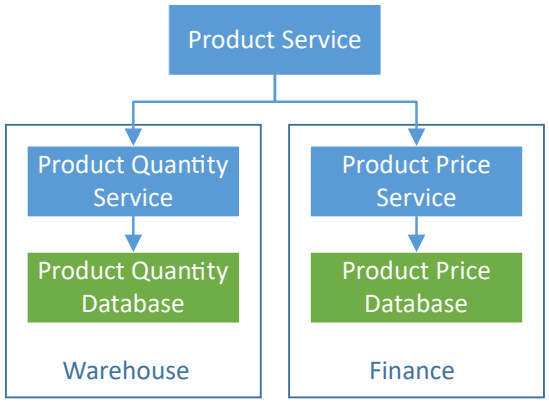
(a) Foreign key: the account service gets the orders from the order service.



(b) Shared static data: countries are read-only data and are hardly never updated.



(c) Shared data: both account and cart services read or modify orders.



(d) Shared table: a product is stored is two databases. One for the quantity in the warehouse and the other for the price.

Figure 2.1: Database Organization

In this case, each query service may expose different representations of the data.
The database can also be split into a read-write one and one or many read-only ones to spread the load.

2.2 Specification

Assuming that we have a description of the required features of our system (with some user stories for instance), the first step would be to roughly define the services with a dependency graph (see section 2.2.3.1). But we have to decide if the APIs is defined globally or each service defines its own.

Defining accurately the APIs ahead is very difficult for a complex application and the result will probably contain some mistakes. This method is then not really feasible in practice.

If each service defines their own API, there are two ways to define them:

- **Contract First**: the API is agreed between the service and its clients before its implementation.
- **Code First**: the service exposes an API and its clients must fit to this API.

To keep the APIs uniform, they should follow some globally defined guidelines. Another solution would be to roughly and globally define the APIs (only the methods but not the models), then each service should specify its API.

2.2.1 Contract First

In the specification first model, the developers implement the service following a strict API. To facilitate the enforcement of the specification, the interfaces should be generated from the interface description language (see section 3.5).

2.2.2 Code First

When we code first, the developers will define the API during the implementation and publish the it afterwards. It would be nice if the interface description language (see section 3.5) could be generated from the code.

2.2.3 Service Catalog

A service catalog lists the services to develop and defines their properties (e.g. organization or API).

2.2.3.1 Dependency Graph

The dependency graph shows which service depends on (i.e. calls) which one to know in which order the services should be developed.
PlantUML[3] is an awesome tool which can generate such a graph from a code (see listing 2.1). We were looking for a generator because further modification of the graph will be easier.

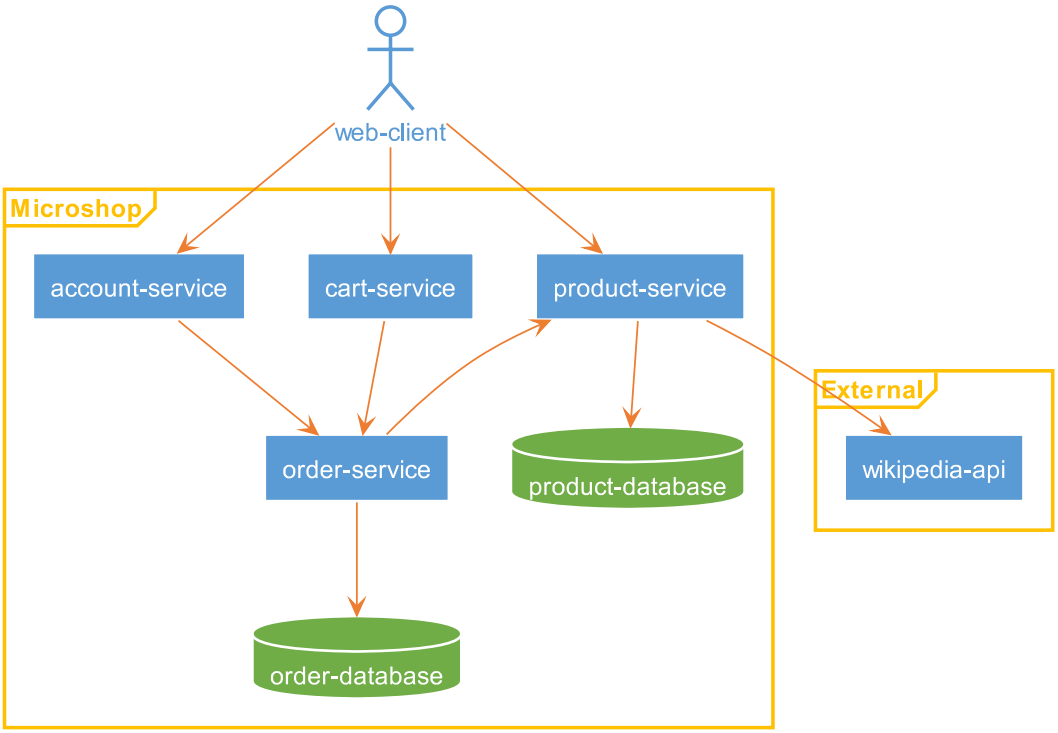


Figure 2.2: Microshop: dependency graph

2.2.3.2 Service Organization

Having many services also means that we should organize them to find and manage them easily. For that purpose, we will use a key-value pairs system.
After many iterations, we end up with the following keys and their non-exhaustive list of possible values:

[3]PlantUML: http://plantuml.com/

Listing 2.1: PlantUML description for Microshop. The result is figure 2.2

```
@startuml
skinparam shadowing false
skinparam actor {
    backgroundColor FFFFFF
    borderColor 5B9BD5
    fontColor 5B9BD5
}
skinparam component {
    ArrowColor ED7D31
    ArrowFontColor ED7D31
}
skinparam database {
    backgroundColor 70AD47
    borderColor FFFFFF
    fontColor FFFFFF
}
skinparam frame {
    backgroundColor FFFFFF
    borderColor FEC000
    fontColor FEC000
}
skinparam rectangle {
    backgroundColor 5B9BD5
    borderColor FFFFFF
    fontColor FFFFFF
}

actor "web-client"
frame "External" {
    rectangle "wikipedia-api"
}
frame "Microshop" {
    rectangle "product-service"
    database "product-database"
    rectangle "cart-service"
    rectangle "order-service"
    database "order-database"
    rectangle "account-service"
    [web-client] --> [product-service]
    [web-client] --> [cart-service]
    [web-client] --> [account-service]
    [product-service] --> [product-database]
    [product-service] --> [wikipedia-api]
    [product-service] <-- [order-service]
    [cart-service] --> [order-service]
    [order-service] --> [order-database]
    [account-service] --> [order-service]
}
@enduml
```

- layer: separate the services by layer or tier inspired from the multi-tier architecture.
  - business
  - data
  - infrastructure
- domain: orthogonally to the layers, we have the domain which separate vertically the services.
  - activity
  - monitoring
  - security
  - transport
  - shared (between many or all domains)
  - weather
- subdomain: an optional label to refine the domain separation.
- type: describe the functions of the service and refine the layer separation.
  - cache
  - client
  - broker
  - database
  - pump
  - service
- technology: indicate which framework or product is used.
  - hazelcast
  - mongodb
  - nodejs
  - postgresql
  - rabbitmq
  - spring
- name: a unique identifier of the service. Ideally, it is defined from the previous labels: "${domain}-${type}" or "${domain}-${subdomain}-${type}"

2.3 Evolution

Services usually update and upgrade over time. We should define a mechanism to handle these changes.
If you consider that a microservice is really disposable, when we need to modify a service, implement a new one instead of modifying the existing one. This avoid taking time to understand the existing code but it is a bit extremist.

2.3.1 Versioning

Each version of a service should be numbered and the Semantic Versioning[4] seems to be a good numbering logic. Roughly, the version is of the form *MAJOR.MINOR.PATCH* and we increment each number following these rules:

- **MAJOR**: breaking changes (the clients might be incompatible with the new version)
- **MINOR**: new backwards-compatible features
- **PATCH**: bug fixes (must be backwards-compatible)

2.3.2 Breaking Changes

In order to give the time to the clients to adapt their code, a solution is to expose two versions in parallel for a while and when the clients have switched to the new one, we can discontinue the old one. For instance, Python[5] keeps available its version 2 and 3 because version 3 introduces breaking changes and there are still people who use the version 2.

[4]Semantic Versioning: `http://semver.org/`
[5]Python: `https://www.python.org/`

The exposed API should be versioned only with the major version (`http://gateway/v1/products/`, `http://order-service/v2/orders/`, ...). There is no need to show the minor and patch versions.
There are two ways to handle two (or more) versions at the same time. They differ by giving the responsibility to:

1. **the services**: the service keeps the implementation of the old versions.
2. **the API gateway (see section 2.6.1)**: many versions of the same service are deployed and the gateway routes the requests to the correct version. For instance, it can use the name of the service (the request `http://gateway/v1/products/` will be routed to service `product-service-v1`).

The second solution is more in the microservices mindset because having many versions in the same implementation implies maintenance on the service (to remove the old version).

2.3.3 Database

If a database needs to be upgraded, one solution is quite similar to the first one described previously and is done in two phases:

- **expansion**: add fields or tables to make the database compatible with both versions.
- **contraction**: when the old version is discontinued, we remove the unused fields and tables.

2.4 Microservices Testing

To avoid regression and verify that our services work correctly, we should write different kinds of tests.

2.4.1 Unit Testing

Unit tests are likely written during a test-driven development (TDD) and each of them checks that a method works correctly. We do not start the service to run these tests and they should be environment agnostic (e.g. external files or network).
There should be a huge amount of this kind of tests in order to avoid regression during the development and these tests should be fast.
In summary, unit tests are fast, isolated, automatized (i.e. regularly run) and have a small scope.

2.4.2 Service Testing

The service tests are used to test each service individually in isolation. They test the service as a whole by doing requests on a running instance and optionally mocking the backing services.

2.4.2.1 Contract Testing

In the contract tests, we first write the contracts between a server and its clients. A contract defines a request and the expected response. Then, we verify that the server responds correctly against the set of contracts and that the clients make the correct requests by stubbing the server (see figure 2.3).

2.4.3 End-to-End Testing

End-to-end testing are done when all services are deployed. These tests are usually done through the frontend to verify the behavior of the whole system.
They are quite slower and harder to automatize than other kinds of tests, but have a bigger scope because the interactions between the services are verified.
We will not implement this kind of tests because we are mainly focusing on the backends.

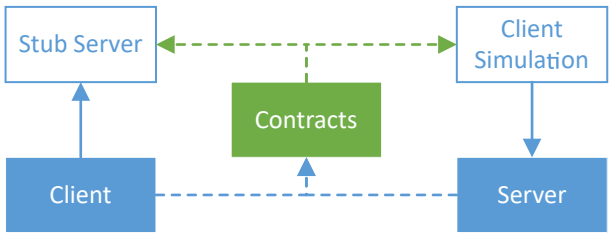


Figure 2.3: In contract testing, the client and the server agree on some contracts and these latter generate a stub server and tests which simulate the client.

2.4.4 Chaos Testing

An interesting way to test the fault tolerance of a system is to periodically simulate failures in the production environment.
Netflix proposes four principles of Chaos engineering:

1. **Build a Hypothesis around Steady State Behavior**: to verify that a system works, we need to define a steady state. This state should be defined by the measurable outputs of the system.
2. **Vary Real-world Events**: the events (i.e. failures) should be inspired from the real world and potentially modify the steady state.
3. **Run Experiments in Production**: a system might behave differently in different environment, so we should better test directly in production.
4. **Automate Experiments to Run Continuously**: test regularly in order to verify that the system is still working.

These principles are summarized in the following steps:

1. define a measurable steady state.
2. assume that the steady state is maintained in a control group and an experimental group.
3. simulate failures (i.e. introduce chaos) such as server crashes or unreliable network in the experimental group.
4. look for a difference between the control and experimental groups.

Unfortunately, we will also not implement this kind of testing because we are focusing on the development environment and not the production one.

2.5 Distributed System Challenges

A microservices system is composed of many independent services, hence also a distributed system with its benefits and drawbacks.

2.5.1 Fallacies of Distributed Computing

We usually make the following eight false assumptions while developing a distributed system (from the whitepaper of Arnon Rotem-Gal-Oz [5]):

1. **The network is reliable**: the service should assume that each packet can be lost and retry to send it after a timeout. It should also be able continue working when the network is restored.
2. **Latency is zero**: you should minimize the costly calls to other services
3. **Bandwidth is infinite**: but you should not send too much data in each call either
4. **The network is secure**: the system is secured in many levels (application, network, infrastructure)
5. **Topology doesn't change**: hosting machine and clients can be added or removed. The physical infrastructure should be abstracted or use a discovery service.

6. **There is one administrator**: upgrade of the whole system is difficult because the different parts of the system are under different responsibilities.
7. **Transport cost is zero**: going from the application layer to the network one and the network infrastructure (e.g. maintenance, hardware) are costly.
8. **The network is homogeneous**: interoperability between the services should be forecast.

In summary, a microservices system should be aware of these fallacies and be built as a fault tolerant system.

2.5.2 CAP Theorem

A distributed system communicates to each other by the network and this network can be subject to failure. When this happens, a network partition can appear and our system should be able to handle it (partition tolerance) by providing limited but consistent behavior (consistency) or by staying available but some data might be lost (availability).
This statement is from the well-known and proven CAP theorem which states that a distributed system can only simultaneously guarantee at most two of the three following properties:

- **Consistency**: the state of all nodes in the system is always the same.
- **Availability**: every node always respond to their clients and handle their requests (read and write).
- **Partition Tolerance**: the system is still working even if some nodes do not see each other's.

We assume that network partition occurs (the network is not reliable), therefore the partition tolerance is a must have. This left us the choice of choosing availability (AP) or consistency (CP). It depends on the use case, especially on the ephemerality of the data or its ability to recover.
For a monitoring system, the availability will be more important because we want to have the data as fast as possible and it doesn't matter if we lose some data. But for a product database, we need that the data are consistent.

2.5.2.1 Proof of Concept with RabbitMQ

We selected the message broker RabbitMQ[6] to see how can we use it to have an AP or CP system.

2.5.2.1.1 Setup I chose to use Docker Compose (see section 4.1.1.5) to deploy the RabbitMQ cluster and its clients (producers or consumers) because we can easily have a fresh install by recreating the containers and configure the virtual network on run-time (partition network). It is also more lightweight and easier to deploy than virtual machines.
We will work with a cluster of three nodes (see figure 2.4b) with all the queues mirrored to all other nodes. RabbitMQ can handle the cluster partitioning in four main ways:

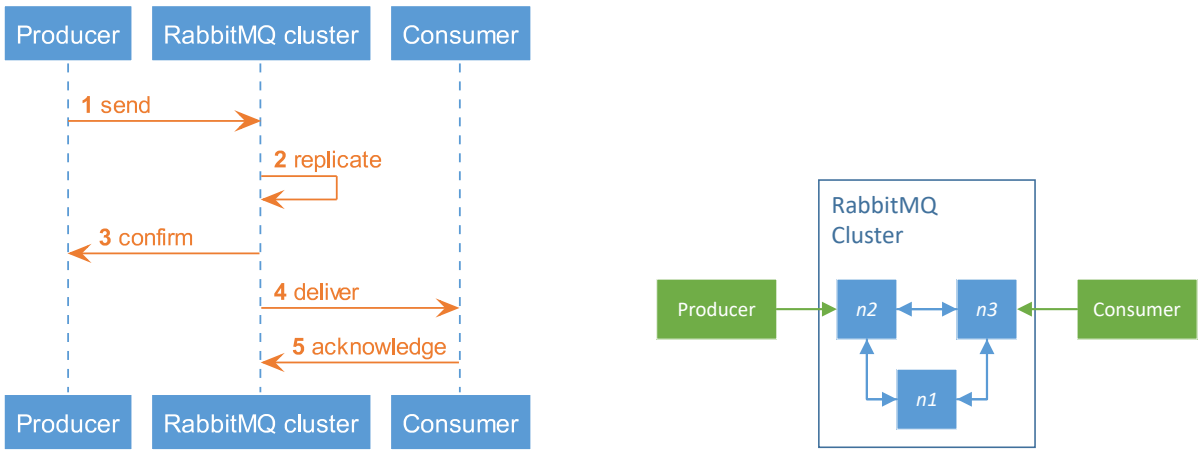
- **ignore**: continue as if there was no partition. Have to manually cluster the node again when the network recovers (we must choose which nodes will lose their data by recreating them). Select this if the network is hardly never partitioned and if you prefer to recover manually.
- **pause minority**: when the cluster is partitioned, all the nodes in a partition which contains less or equals to the half number of nodes will be paused (close every connection and wait for the network recovery to connect again to the main cluster). The recovery is done by keeping only the data in the winning partition which is computed as follows (from highest to lowest priority): more connected clients, more nodes and randomly. This seems to be a CP system.

[6]RabbitMQ: `https://www.rabbitmq.com/`

- **pause if all down**: a node is paused when it cannot reach any of the nodes in a list. Similar to the pause minority but we have more controls on the nodes.
- **autoheal**: all the nodes are still running during the partitioning and when the network is recovered, decide a winning partition as for pause minority to erase all the data in the losing partition. Provide a high availability but can lose a lot of messages.

2.5.2.1.2 Availability If you want to have an AP system, the autoheal mode seems to be the most appropriate. The cluster recovers automatically and the nodes are always running to accept requests. The drawback is you might lose every message which were send during the partitioning (if the producer was connected to a losing partition).

2.5.2.1.3 Consistency Pause minority seems to be the most appropriate mode to get a CP system, but only choosing this mode is not enough. You might still lose the messages which are send in a small window after the physical partitioning and before the time the node is paused. To avoid that, the producers should wait for the confirmation of the sent message and resend it after a timeout if no acknowledgment. In our case, the producer waits for the confirmation to send the next message. Note that RabbitMQ confirms a message when it is replicated to all nodes and not when it is delivered to the consumers.
We also ask the consumer to acknowledge the received messages in order that RabbitMQ considers the message as delivered. This avoid to lose messages which were currently processed by a crashed consumer. In our configuration, the consumers acknowledge the message before receiving the next one. You can see the communication between all participants in figure 2.4a.



(a) Message confirmation and acknowledgement in RabbitMQ

(b) Initial configuration of the proof of concept with RabbitMQ

Figure 2.4: RabbitMQ

With all these configurations, we can think that it is not possible to lose any messages, but a very interesting article from Balint Pato [6] shows that it is still possible to lose messages if the cluster is partitioned and recovers in a particular order:

1. Let's enumerate the three nodes: *n1*, *n2*, *n3* (see figure 2.4b).
2. *n1* is partitioned from *n2* and *n3*. The partition *p23* (composed of *n2* and *n3*) is the winning one and *n1* is paused. The producers and consumers continue to produce and consume message through the remaining nodes.
3. *n2* and *n3* are partitioned, so everything is paused. Note that there might be some accepted messages which are not yet delivered and are still in the queues.
4. The connection between *n1* and *n2* is back and RabbitMQ has to choose which one is the winning partition: none of them has any clients because they were previously paused, both partition is composed of one node so it is again a draw. The winning partition will therefore be selected randomly.

- If *n1* is the winning partition, the messages in the *n2* queues will be replaced by *n1* which are either already delivered by *p23* or nonexistent.
- If *n2* wins, we will not lose any messages because the erased messages have already been delivered.

5. When the network recovers completely, *n3* will certainly be the loosing partition because it is composed of fewer nodes. Therefore, the state of the cluster depends only on the previous step.

So, we see that there exists a very specific case where RabbitMQ might lose messages. A solution would be to have only two nodes instead of three. In that case, when a partition occurs, everything is frozen and will be recovered as is.

2.5.2.1.4 Conclusion It might be tempting to deploy only one cluster of all purposes, but we have seen that the configuration of an AP or CP cluster is very different. Thus, it would be better to have at least one cluster for the availability and another one for the consistency.

2.5.3 Consensus

Another challenge with distributed systems is how to synchronize some processes together in order for them to agree on a value. Each process proposes some values and they should agree on one.
A consensus protocol must have the following properties:

- **Termination**: all processes always end up by having one value.
- **Validity**: if every process proposes the same value, this value is chosen.
- **Integrity**: each process can only vote for a value which has been proposed by another one or itself.
- **Agreement**: every process must finally have the same value.

Note that the processes might be unreliable and the protocol should handle these failures. The protocol still works even if n processes fail, it is n-resilient.

2.5.3.1 Consensus Algorithm

We can cite two known consensus algorithms:

- **Paxos**[7]: the reference in this domain.
- **Raft**[8]: equivalent to Paxos in term of fault-tolerance and performance, but simpler.

Note that if a distributed system uses a consensus algorithm to agree between each other's, the system will be consistent due to the consensus between the components.

2.6 Design Patterns

In this section, we detailed some useful design patterns for a microservice system.
Other design patterns are also mentioned in this report, such as:

- **Bulkhead**: when a failure happens, it should be contained and not propagated to other parts of the system.
- **Timeout**: you should avoid waiting forever on a resource.

[7] Paxos: `https://www.microsoft.com/en-us/research/publication/paxos-made-simple/`
[8] Raft: `https://raft.github.io/`

2.6.1 API Gateway

A microservice system will likely end up with many backend services and a frontend web client which should connect to many of these backends. You will have, among others, the following issues:

1. Same-origin policy: the naive solution would be to enable Cross-origin resource sharing (CORS) to each of your backends.
2. Your web client should know the location of all services. It would be difficult to manage all the locations when the topology changes (see section 2.5.1).

To solve these problems, one solution is to use an API gateway which is roughly a reverse proxy on your backend services. The API gateway can solve the previous issues by:

1. Serving the web client through itself or by setting the CORS (only) in the gateway
2. Abstracting the backend services. The web client will hence only need to connect to the gateway.

2.6.1.1 Backends for Frontends

A variation of the API Gateway pattern is the backends for frontends. We have an API gateway dedicated to each type of frontends (e.g. web applications, mobile applications and third parties). These specific API gateways will route the requests to the same backend services but they might handle the routing a bit differently (per user authentication for the web application but key authentication for third party for instance) or a different data format.

2.6.2 Circuit Breaker

If your application connects frequently to an unavailable backing service (e.g. database, message broker, another service), you might prefer to handle this error more cleverly with a circuit breaker than retry until it works again.
Instead of calling directly the backing service, you do it through the circuit breaker which keeps the state of the connection (a.k.a. circuit). By analogy to the electric field, a circuit can be closed (everything goes well) or opened (the connection is broken). A connection check can be performed at each request or regularly to know the state of the circuit. Then, when we use the circuit (i.e. connect to the backing service), we check beforehand if the circuit is well closed and if not, a fallback can be triggered instead.
Thanks to this pattern, we will fail faster and we can also define a fallback when the circuit is opened. In return, the requests will perform a bit slower due to the middleware.

2.6.3 Service Discovery

In a cloud-native environment, the service instances can be added or removed quite often due to the on-demand philosophy. In order to allow our services to connect to each other's, they have to know the other service's locations.
With the service discovery mechanism, each service registers itself to a service registry with an identifier (ID) which is known by the other services. When a service need to calls another one, it gets the location of the instances from the registry. Each service is also able to know the list of available services.
More specifically, a service discovery should:

- Allow a service to register and unregister in a service registry
- Know when a service is not available anymore. This can be achieved in three ways:
  - The service discovery performs the health-check on every registered service (this solution does not scale).
  - Creating a service registrar when a service registers and this registrar do the health-check instead of the service discovery (a.k.a. the third-party registration pattern).

  - The services have to send the heartbeat themselves (a.k.a. the self-registration pattern).
- Distributed because it would be a single point of failure.
- Give the list of registered and/or available services by DNS and/or a REST API.

Note that for an event-driven architecture, the service discovery might be useless.

2.6.4 Load Balancing

A service might scale to many instances, so we should load balance the incoming requests into all available instances. Given that each instance is stateless, the load balancer does not need to always route a client to the same instance (a.k.a. sticky session load balancing). But a good one should know the load of each of its instances to keep the overall load as homogeneous as possible.
Following the Airbnb developers [7], an ideal load balancer has the following properties:

- An available service instance should receive requests.
- A new request routes to the least busy instance.
- No requests are routed to an instance which has an error.
- It is possible to unload a selected instance.
- The load of each instance is monitored.

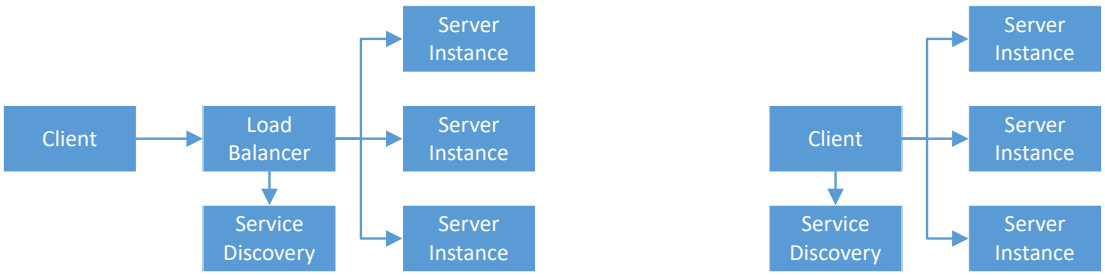
There are two kinds of load balancing: server-side or client-side.

2.6.4.1 Server-Side Load Balancing

One load balancer is deployed for each group of server instances and all requests to an instance is sent to the load balancer (see figure 2.5a) which gets the location of the instances from the discovery service. The client only knows the location of the load balancer and has no idea about how many server instances there are.

2.6.4.2 Client-Side Load Balancing

The client gets the location of all server instances from the service discovery and does the load balancing (see figure 2.5b).



(a) Server-Side (b) Client-Side

Figure 2.5: Load Balancing

2.6.5 Publish–Subscribe

If your microservice application is also event-driven, you would need asynchronous communication between your services.
One design pattern to send asynchronous messages is called publish-subscribe. The publisher put the messages in a message broker without knowing to whom they will be routed. On the other hand, the subscriber subscribes to one or many channels on the message broker and when there is a message routed to this channel, the message broker delivers it to the subscribed subscribers. There are two ways to route a message:

- **Topic**: the publisher set the topic of the sent message. Every message of a topic can be routed to a list of channel.
- **Content**: the content of the message defines the channel to which it will the routed.

The message brokers should store the messages until they are forwarded.
This design pattern allows the publishers and subscribers to be loosely-coupled. It also provides more scalability to the system.
The main drawback is the addition of a fallible component (i.e. the message broker) to an already complex system.

Chapter 3

Inter-Service Communication

The different services must communicate through the network instead of some function calls as for a monolithic application. the table 3.1 shows the kinds of interactions the services can have together.

Table 3.1: Kinds of interactions between services

		One-to-One	**One-to-Many**
Synchronous		request-response	
Asynchronous	**Response**	asynchronous request-response	publish-responses
	No response	notification	publish-subscribe

Note that it is possible to use synchronous interaction to simulate one-to-many interactions by performing one request for each server and also asynchronous ones if we do not wait for the response. A synchronous system will be more robust because the message flow is more predictable, but is less efficient due to the waits on the responses.

3.1 Synchronous

3.1.1 Hypertext Transfer Protocol

A well-known synchronous request-response protocol on the network is the hypertext transfer protocol (HTTP). It has the advantage to be widely in the world-wide web and is well known by the developers.
A request is composed of an HTTP method (e.g. GET, POST, PUT, PATCH, DELETE) and an optional body (note that some methods cannot have body). The client always expects a response from the server with a status code and optionally a body with extra information.

3.1.1.1 Representational State Transfer

The probably most known way to use HTTP to communicate between services is the representational state transfer (REST) style. A RESTful web service should satisfy the six following constraints [8]:

1. **Client-Server**: the responsibilities are divided between the client (user interface) and the server (data storage).
2. **Stateless**: each request contains all information needed to be understood by the server. The server has zero-knowledge about the previous requests.
3. **Cache**: the HTTP requests might be cached by an intermediary. So, the respond should indicate if we can cache it or not.
4. **Uniform Interface**: the server implementation is abstracted to the client with an interface which has the four following constraints:
   (a) **Identification of Resources**: each resource is identified (e.g. by an URL)

(b) **Manipulation of Resources through representations**: if a client has a representation of a resource, it is able to modify it.
(c) **Self-Descriptive Messages**: the message tells how it is formatted (using a media type[1] for instance).
(d) **Hypermedia as the Engine of Application State (HATEOAS)**: each response should contain the links to the next possible requests (see section 3.1.1.2).

5. **Layered system**: A client is not aware if the server forwards the request to another server.
6. **Code-On-demand (optional)**: the server can transfer executable code (applets or scripts) to the client.

3.1.1.2 HATEOAS

We usually use the term of REST even if not all constraints are fulfilled (especially the uniform interfaces with HATEOAS). The HATEOAS way to use HTTP is just an enforcement of the HATEOAS constraint.
When we get a representation of a resource from an HATEOASful service, it should also indicate how to get, modify or delete this resource by providing the URLs.

3.1.2 Apache Thrift

An alternative of using the HTTP protocol is Apache Thrift[2]. You should write a Thrift definition file which is the contract between the client and the server. Then, from this file, you can generate a client and a server in many available languages.

3.1.3 gRPC

gRPC[3] from Google is quite similar to Thrift, but uses Protocol Buffers (see section 3.3.3) to serialize data.

3.2 Asynchronous

3.2.1 Message-Oriented Middleware

A way to perform asynchronous communication between the services is to use a message-oriented middleware such as a message queue. The client put a message in a message queue, this message can be routed to the correct servers and the servers can consume the message when they have the time to process it.
We will discuss more about this topic in section 2.6.5.

3.3 Data Serialization Format

We have seen some protocols we can use to communicate between the services and we will see in which formats the data can be transferred.

3.3.1 Extensible Markup Language

Extensible Markup Language (XML)[4] is a mature and powerful format with some great tools such as:

- **XPath**: a language to query the nodes in an XML document.

[1] Media Types: `http://www.iana.org/assignments/media-types/media-types.xhtml`
[2] Apache Thrift: https://thrift.apache.org/
[3] gRPC: `http://www.grpc.io/`
[4] Extensible Markup Language: `http://www.w3schools.com/xml/`

- **XML Schema**: to define a pattern and validate XML document against it.
- **XSLT**: to transform a XML document to another or even to other types.

3.3.2 JavaScript Object Notation

A more recent format is JavaScript Object Notation (JSON)[5]. It is currently very trendy because it is more lightweight and easier to read than XML, but it is nothing more than a data format where XML is nearly a language (types and comments). There is JSON Schema which is the equivalent to XML Schema, but it is not very mature yet.

3.3.3 Protocol Buffers

Protocol Buffers[6] is a compact format developed by Google. It is less popular than the previous ones, but if you need to exchange very small packets, this can be a good choice.

3.4 Application Programming Interface

The application programming interface (API) is the contract between a service and its clients. Due to the lack of time, only the REST API has been explored in this project.

3.4.1 REST API Design

To facilitate the calls and the development of our services, the REST API of every services should be defined uniformly with the same schema. To define our guideline, we follow the APIs of web giants such as Google, Facebook or Twitter.

- **Use HTTP methods to describe the action**: avoid verbs in the path and use the HTTP method to describe the action. The table 3.2 shows the recommended use of the methods.
- **Prefer plurals in the URL**: always use plural nouns even to access a single resource makes the URL more uniform and intuitive.
- **Consistent case**: choose one case style to get some more readable paths. There exist three main case styles: CamelCase, snake_case and spinal-case.
  Even if the URL should be considered as case-sensitive, they usually don't and we are more used to see URL in lower case, therefore CamelCase will not be a good choice. The spinal-case seems better than the snake_case because '-' is more visible than '_' (for instance when the URL is underlined).
  We also have to choose a case style for the resources. If JSON is used as the data format, it is more natural to use CamelCase because it is used in JavaScript.
- **HTTP Status Codes**: giving the correct HTTP status code back to the client helps him/her to understand the response and/or error. Table 3.3 shows some frequent HTTP status code and their meanings.
- **List filter**: instead of getting the whole list each time, it would be preferable to get only the needed items by filtering them: `.../resources?field1=value1,value2&field2=value3`
- **Versioning**: the services evolve and their API will probably have breaking changes. So, the client should be able to know which version it currently uses in order to update its code and be compliant with the next one. A good practice is to put the version in the root of the path: `http://api.example.com/v1/...`
- **Partial response**: the client might need only some fields of the returned object, so to save the bandwidth, it would be nice to have a way to select the returned fields. For example, in a request parameter: `.../resources/resource-id?fields=attribute1,attribute2`

[5] JavaScript Object Notation: `http://www.json.org/`
[6] Protocol Buffers: `https://developers.google.com/protocol-buffers/`

Table 3.2: Recommended use of the HTTP methods and the body response.

HTTP method	CRUD	/resources	/resources/{resource-id}
GET	**Read**	200: list of resources	200: requested resource
POST	**Create**	201: created resource	
PUT	**Full update**		200: updated resource
PATCH	**Partial update**		200: updated resource
DELETE	**Delete**		204: no return

Table 3.3: Main HTTP status codes and their meanings.

HTTP Status Code	Message	Description
200	OK	Success with the requested resource
201	Created	Resource successfully created. Link to the resource in the `Location` header
202	Accepted	For asynchronous processing
204	No Content	Success with no response
400	Bad Request	Malformed syntax (missing a required property, ...)
401	Unauthorized	Authentication is required
403	Forbidden	Authentication is successful but no access right
404	Not Found	Requested resource not found
405	Method Not Allowed	Incorrect HTTP method
500	Internal Server Error	Server-side issue

3.5 Interface Description Language

An interface description language (IDL) allows to describe and document an API in order to publish the API. An ideal IDL and its tools should be able to:

- Describe an API (e.g. endpoints, data format).
- Generate a documentation (e.g. web page) of the API.
- Generate the interfaces for the producer to assure that implementation exposes the right API.
- Generate some stubs or a library for the consumers to facilitate the development.
- Generate tests to verify the implementation of the producer.
- Generate the IDL (and the documentation) from an implementation. This is useful if coding first (see section 2.2.2).

3.5.1 REST API Description Language

3.5.1.1 OpenAPI Specification (Swagger Specification)

The most popular and thus supported IDL for REST API is probably the OpenAPI Specification[7] (formerly known as Swagger specification[8]) developed by Swagger[9]. It uses JSON or YAML to describes the APIs. Swagger has some tools such as:

- **Swagger UI**[10]: generates a static web page which documents the API. We can also test the API directly on the web page.

[7]OpenAPI Specification: `https://www.openapis.org/specification/repo`
[8]Swagger specification: `http://swagger.io/specification/`
[9]Swagger: `http://swagger.io/`
[10]Swagger UI: `http://swagger.io/swagger-ui/`

- **Swagger Codegen**[11]: generates a stub for your server or clients. It already supports many languages and frameworks (e.g. Java, JavaScript, Python, Scala, Spring) but you can also create your own generator.
- **Swagger Editor**[12]: a web application where you can write the specifications and see on-the-fly the rendered Swagger UI. It can also generate the stubs with Codegen.
- **Springfox**[13]: developed by Springfox for Spring services. This dependency extracts the endpoints from the Spring MVC annotations and automatically generates and publish the corresponding Swagger file with Swagger UI.

The main drawbacks of Swagger are:

- the lack of tests generator.
- no polymorphism on the resources. The returned resource might have different sub-types with some additional specific fields.
- the Swagger Editor web application can only load one file at once. You might prefer to have shared descriptions for the resources.

While selecting an IDL for REST API, the default choice would probably be Swagger because it is widely supported due to its popularity. For instance, if you use an API gateway, it might publish the documentations only in a few IDL and there is a high probability that Swagger is supported.

3.5.1.1.1 Swagger Codegen The provided Swagger Codegen generates Spring project stubs with more classes than necessary (e.g. empty implementation of the methods). So, I customized Spring Codegen to only generate the interfaces and to already give the correct package name.

3.5.1.2 RAML

RESTful API Modeling Language[14] (RAML) is only based on YAML, but the language is more powerful than Swagger by allowing polymorphism of the resources. It also has some useful tools such as:

- **API Workbench**[15]: the RAML editor is an Atom[16] package. Even if it is still in beta at the time of writing, the editor is very complete and the main advantage is the possibility to include other files in the descriptions.
- **API Designer**[17]: a web editor similar to Swagger Editor.
- **Spring MVC-RAML Synchronizer**[18]: generates the RAML file from the Spring annotations, verifies the contracts and generates the stubs.

3.5.1.3 API Blueprint

Another competitor is API Blueprint[19] which is based on Markdown. It has some plugins for Sublime Text[20] and Atom as an editor.

[11] Swagger Codegen: `http://swagger.io/swagger-codegen/`
[12] Swagger Editor: `http://swagger.io/swagger-editor/`
[13] Springfox: `https://github.com/springfox/springfox`
[14] RESTful API Modeling Language: `http://raml.org/`
[15] API Workbench: `http://apiworkbench.com/`
[16] Atom: `https://atom.io/`
[17] API Designer: `https://www.mulesoft.com/platform/api/anypoint-designer`
[18] Spring MVC-RAML Synchronizer: `https://github.com/phoenixnap/springmvc-raml-plugin`
[19] API Blueprint: `https://apiblueprint.org/`
[20] Sublime Text: `https://www.sublimetext.com/`

3.5.1.4 Spring REST Docs

Spring has its own IDL: Spring REST Docs[21]. It uses Asciidoctor[22] to render HTML pages and has a better integration with the Spring ecosystem, especially for the testing part.

[21] Spring REST Docs: `https://projects.spring.io/spring-restdocs/`
[22] Asciidoctor: `http://asciidoctor.org/`

Part II

Environment

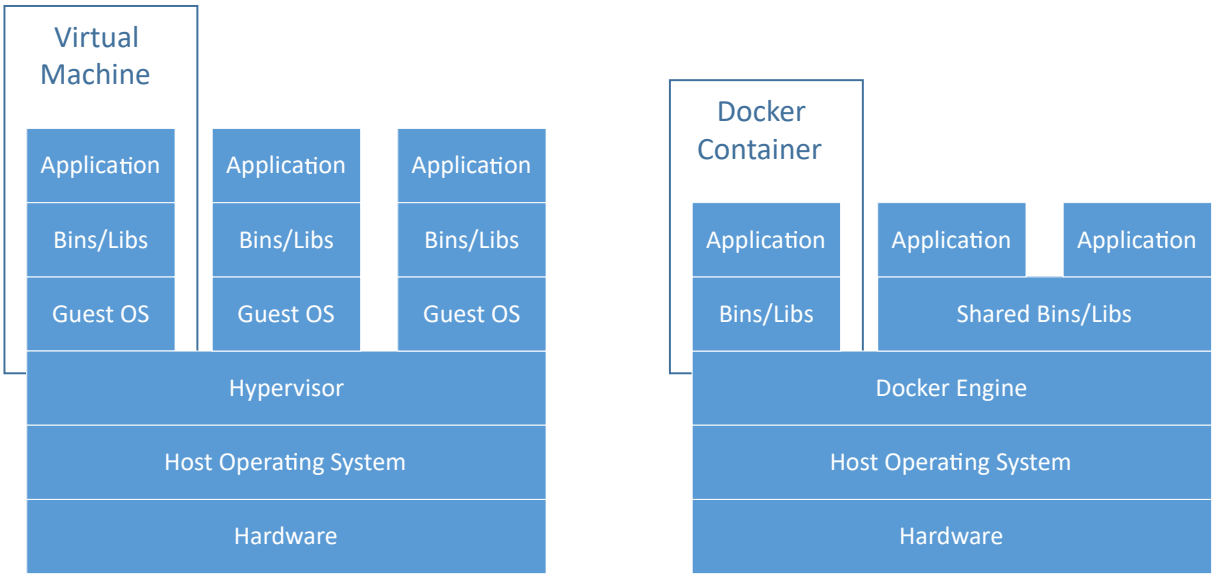
Chapter 4

Deployment

Deploying and managing a set of microservices which can scale individually on demand is quite a complex task which can hardly be done manually.
The simplest deployment would be to deploy each service on a virtual machine provided by an IaaS, but we are going to see for more manageable solutions.

4.1 Operating-System-Level Virtualization

We said that the instances of our microservices communicate with each other over the network. Hence, we should consider a service instance as a machine which might listens on a port for incoming request.
One solution is to use virtual machines to run them, but there exists a better solution called the operating-system-level virtualization (a.k.a. containerization).
Instead of virtualizing the operating system (OS) or even the hardware like a virtual machine (see figure 4.1a), we use the OS of the host and the virtualization is done through multiple isolated user space instances (see figure 4.1b). These virtual hosts are commonly called containers and each of them has its own isolated file system.



(a) Virtual Machines (b) Docker Containers

Figure 4.1: Comparison between virtual machines and Docker containers

The main limitation of this kind of virtualization is that the guest OS should use the same kernel as the host one. But a container starts very quickly (no need to start an OS) and the image is also slimmer because it contains only the diff of the file system but not the state of the machine.

4.1.1 Docker

Docker[1] is by far the most well-known and used operating-system-level virtualization. it uses some features of the Linux kernel to get a full virtualization, such as:

- **Namespaces**: isolate the resources
- **Control groups**: limit the usage of resources
- **Union file systems**: view file systems as some lightweight layers

4.1.1.1 Docker Engine

The Docker Engine is composed of a Docker daemon which manages the images and containers, and of a Docker CLI which provides an interface to the user. These two components communicate through a REST API.

4.1.1.2 Docker Image

To create a container, we need a Docker image which describes the file system and how to run the container. An image is composed of a sequence of layers (see figure 4.2).

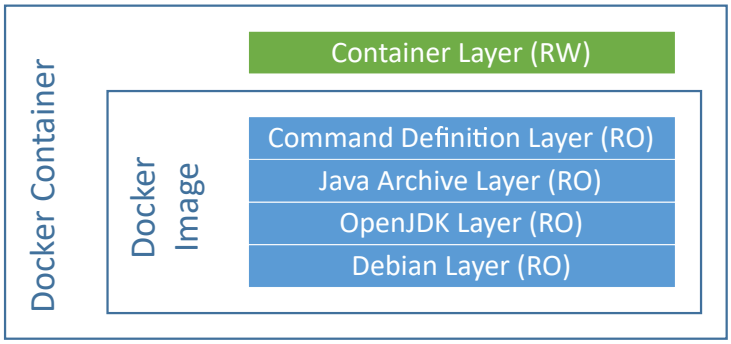


Figure 4.2: A Docker image is composed of many read-only (RO) layers. When this image runs as a Docker container, a thin read-write (RW) container layer is added on top of the image layers.

Each layer is roughly either a diff of the file system or a property (e.g. metadata, exposition of a port, definition of main process, environment variable, ...). Any images can be extended by appending some additional layers and this form a new image.
A Docker image is described by a *Dockerfile* (see listing 4.1) which defines the base image it is extended from and each instruction adds a new layer to the image.

Listing 4.1: Dockerfile sample

```
1 FROM debian:latest
2 RUN echo 'echo "Hello World !"' > /opt/entrypoint.sh
3 CMD /opt/entrypoint.sh
```

This hierarchy between the images allows us to create some base images and the images of our services would be extended from them. So, it is possible to share some common layers between our services (which contain some agents for instance).

4.1.1.2.1 Image Name The full name of a Docker image also contains its Docker Registry location:

$$\underbrace{\underbrace{\underbrace{localhost}_{repository\ host} : \underbrace{5000}_{repository\ port}}_{repository\ url} / \underbrace{debian}_{image\ name} : \underbrace{1.0}_{image\ tag}} \tag{4.1}$$

[1]Docker: `https://www.docker.com/`

The default repository URL is the Docker Hub (see 5.4.1) and the default tag is `latest`.
Note that the repository host can be a domain or subdomain but the URL cannot contains a path such as `localhost:5000/repository` because the Docker would not know where the image name starts.

4.1.1.3 Docker Container

The runnable and isolated instance created from a Docker image is called Docker container. A container can be running, stopped, moved or deleted by the Docker engine. The container adds a thin read-write layer on-top on the read-only image layers and all modifications are done in this thin layer. This allows to reuse the read-only layers for other containers (see figure 4.2).

4.1.1.4 Docker Registry

A Docker registry stores a set of images. More details in section 5.4
About the security, Docker connects by default to the registry in HTTPS and the registry should have a certificate signed by a trusted authority. If you use a self-signed certificate instead, you must add the certificate in each Docker engine in order to connect to the registry. It is also possible to enable the insecure mode in Docker to connect in HTTP.

4.1.1.5 Docker Compose

To manage a set of containers, it would be recommended to use Docker Compose[2], which allows to build, run, stop or remove a set of containers at once.
A file named *compose.yml* (see listing 4.2 and figure 4.3) defines a set of containers. The Docker images can be pulled from a registry or built from a *Dockerfile* and it is possible to configure the container directly in *compose.yml* (e.g. set the volumes, environment variables, commands).

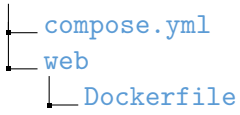


Figure 4.3: The directory tree of a Docker Compose project.

Listing 4.2: A *compose.yml* which describes a web application connected to a database. The directory tree is shown in figure 4.3

```
web:
  build: web/
  links:
  - database
  ports:
  - "80:9000"
database:
  image: mongo
```

4.2 Container Orchestration

Having a container per service instance leads to managing a lot of container. Automation of this management by a central process is required and is called orchestration.
Moreover, we would like to be able to run these containers on many hosts in order to scale horizontally.
Using a container orchestration also abstracts the hosting part to the developers. It doesn't matter for the developers how many hosts are available and on which host the container is running.

[2]Docker Compose: `https://www.docker.com/products/docker-compose`

Kubernetes is going to be mainly explained because ELCA has selected OpenShift Origin (more details in section 4.3.1) as a private PaaS. So, using this platform saves the time to evaluate and deploy another one. Moreover, the selection of the deployment platform is not part of the subject of this project.
But there are some other solutions such as:

- **Marathon**[3]: the solution for Apache Mesos[4], which is a cluster manager (virtualize a set of hosting machines). Note that Kubernetes can also be deployed on Mesos.
- **Nomad**[5] **by HashiCorp**: it does not only support Docker containers but also virtual machines.
- **Cloud solutions**: such as Amazon ECS[6] and Azure Container Service[7]

4.2.1 Docker Swarm

Docker Swarm[8] is the solution proposed by Docker. You can connect many Docker engines together and they will behave as a single one. It is quite easy to deploy and configure, but it does not provide any additional features than Docker.

4.2.2 Kubernetes

Google has open-sourced their container orchestration solution called Kubernetes[9]. It is probably one of the most known and is part of the Cloud Native Computing Foundation[10].
A Kubernetes cluster is composed of one or several masters, which host the web user interface, handle the API calls, assign the container's deployment and manage the minions (where the containers run).
Kubernetes uses the term of "service" to design one of its component, so in this section, the term "service" will refer to a Kubernetes service.

4.2.2.1 Organization

Kubernetes has a very specific organization for its containers (see figure 4.4):

- **Namespace**: Kubernetes can have many isolated projects and this isolation is called namespace. Only components which are in the same namespace can see each other.
- **Pod**: A pod is a logical host which can run one or more containers and they will run on the same "physical" host. It has its own IP address and volumes. The pod also has its own lifecycle and know the status of its containers (e.g. running, crashed). The pods are created and destroyed, but never stopped or saved.
- **Replication Controller**: The pods can scale horizontally and the number of replicas is ensured by the replication controller which can create or delete pods. You can scale manually the number of replicas or define an auto-scaling control agent which dynamically scale following the resources used by the pods.
- **Service**: The service is an abstraction of a set of pods. It has an IP address and a hostname which can be reached by any pods. A service load balances its requests to some pods which are determined by the label selector (the set of pods whose labels set is a subset of the label selector).
- **Label**: Each component (e.g. pod, replication controller, service) can have a set of labels (key-value pairs). These labels can be used to organize the components and to filter the selections.

[3]Marathon: `https://mesosphere.github.io/marathon/`
[4]Apache Mesos: `https://mesos.apache.org/`
[5]Nomad: `https://www.nomadproject.io/`
[6]Amazon ECS: `https://aws.amazon.com/ecs/`
[7]Azure Container Service: `https://azure.microsoft.com/en-us/services/container-service/`
[8]Docker Swarm: `https://www.docker.com/products/docker-swarm`
[9]Kubernetes: `https://kubernetes.io/`
[10]Cloud Native Computing Foundation: `https://www.cncf.io/`

The labels are used to orthogonally separate the components, which means that a deployment will have the same labels as the pods it controls, the service which expose these pods and the route attached to this service.
These labels are set following the service organization (see section 2.2.3.2).

4.2.2.2 Built-in Features

Kubernetes is more than a simple container manager. It also provides some additional features than Docker Swarm such as:

- Service Discovery: each pod can access every service by their name. Kubernetes uses SkyDNS (more details in section 7.4.1) as a DNS resolver.
- Load Balancing: given that the pods are behind a service which load balances the incoming requests, we don't have to implement a load balancing mechanism.
- Self-healing: the pods keep an eye on the health of their containers and can restart them if one of them dies.
- Storage orchestration: the provisioning of the volume storage is abstracted to the developers. The administrator can create a set of persistent volumes bound to a persistent storage (e.g. NFS). Then, each project can create a persistent volume claim which select an available persistent volume to bind to. Finally, a pod mounts its volume to a persistent volume claim. The way the volume is persisted is abstracted to the developer.
- Rollout: when upgrading the pods, the replication controller replaces one by one each pod by firstly creating the new one, ensures that it works properly and then delete the old one.
- Rollback: the history of the deployment is kept, so we can roll back to a previous deployment if something goes wrong.

4.2.2.3 Configuration File

The deployment to Kubernetes is done by creating a configuration file which describes what to deploy and how to configure it. These configuration files can be in JSON or YAML (we will prefer YAML for its readability).
To deploy, we can use the command-line interface to create every component listed in this file (see listing 4.3).
It is possible to deploy directly from the web console, but the main advantage of using configuration files is the ease to redeploy in another namespace and to delete all the described components.

4.3 Platform-as-a-Service

A Platform-as-a-Service (PaaS) is a platform where we can run and manage our applications by abstracting the underlying infrastructure. It can be private, public or even hybrid (multiple providers).
As mentioned previously, ELCA has selected OpenShift Origin as the private PaaS, but others exist such as:

- **Pivotal Cloud Foundry**[11]: probably one of the biggest competitor to OpenShift Origin
- **Apcera**[12]: oriented in the security of the containers.

4.3.1 OpenShift Origin

OpenShift Origin[13] is a private PaaS from Red Hat built on top of Kubernetes. It takes a lot of concepts from Kubernetes (see figure 4.4) and its command line interface is quite similar too.
OpenShift Origin adds some features to Kubernetes. For example:

[11] Pivotal Cloud Foundry: https://pivotal.io/platform
[12] Apcera: `https://www.apcera.com/`
[13] OpenShift Origin: `https://www.openshift.org/`

Listing 4.3: Example of a configuration file (with a service, a deploymentConfig (equivalent to the replication controller) and a route) where the pod exposes the port 8080 and the service the port 80.

```yaml
apiVersion: v1
kind: List
items:
  -
    apiVersion: v1
    kind: Service
    metadata:
      name: service-name
      labels:
        key: value
    spec:
      ports:
        -
          port: 80
          targetPort: 8080
      selector:
        deploymentconfig: selector-name
  -
    apiVersion: v1
    kind: DeploymentConfig
    metadata:
      name: deployment-name
      labels:
        key: value
    spec:
      replicas: 2
      selector:
        deploymentconfig: selector-name
      template:
        metadata:
          labels:
            deploymentconfig: selector-name
            key: value
        spec:
          containers:
            -
              name: container-name
              image: repository/image-name
              env:
                -
                  name: ENV_NAME
                  value: env-value
          serviceAccount: service-account
  -
    apiVersion: v1
    kind: Route
    metadata:
      name: route-name
      labels:
        key: value
    spec:
      to:
        kind: Service
        name: service-name
      port:
        targetPort: 80
```

- **Route**: to expose a Kubernetes service to the outside of OpenShift, we can create routes to expose a service as a hostname (i.e. url). The default hostname is on the form "http://<route-name>-<namespace>.<domain.ext>".
- **Integrated OpenShift Registry**: a Docker registry.
- **Build**: build the Docker image for us.
- **Image Stream**: roughly an observable Docker image, so when the image updates, it can trigger an action such that a redeployment.

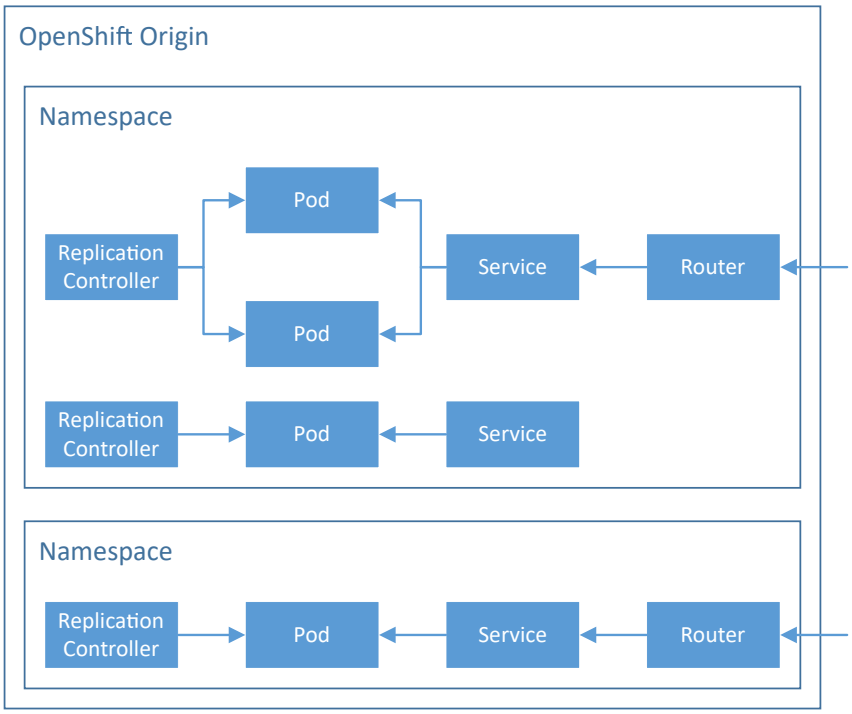


Figure 4.4: OpenShift Origin Overview

4.3.2 Rancher

At the beginning of my project, there was only a pre-production instance of OpenShift (an installation to evaluate the product) and it was not very stable. Given that we had many issues with it, we thought about deploying an alternative PaaS.
Rancher[14] is more a simple container orchestrator than a complete PaaS, but it is quite easy to install (run a Docker image in each host) and allows to visualize and manage (scale, move, create, delete) the containers from a web interface.

4.3.3 Database Deployment

The database is a very specific service which needs to persist its data over the redeployments. There are two ways to handle this persistence:

1. **Persistent Volume**: Docker proposes a volume mechanism which synchronize a directory on a container with another one on the host. This mechanism is usually used by the PaaS to bind this volume to a persistent storage (e.g. NFS). This way, we can envisage to deploy a database as any other services and persist the data directory.
2. **Database as a Service (DBaaS)**: instead of deploying our database, we can also use a DBaaS to persist our data.

Using a DBaaS seems to be a better approach because a database does not have the same properties (one persistent instance) than a typical service (scalable and disposable).

[14]Rancher: `http://rancher.com/rancher/`

Chapter 5

Build, Test & Release Automation

Deploying many microservices by hand is not very convenient and error prone, so we need to automate this process.
Continuous Integration and Continuous Delivery (CI/CD) tools are used to automate the builds, tests and releases of an application. They are usually composed of the following components:

- **Build tool**: we first need to automate the build process to be able to easily delegate it (it's easier to run one command line instead of writing a *how-to-build.txt* file).
- **Version Control System**: the source codes should be centralized in a code repository. The common practice is to use a version control system (VCS) which also supports the versioning.
- **Continuous Integration Software**: as its name suggests, it is the core component. The continuous integration software (CIS) fetches the source code from the VCS, builds it, run the tests and push the built artifacts (Docker images in our case) to a repository.
- **Repository Manager**: we need a place to store our artifacts in order to publish and deploy them later.

Continuous Deployment goes a step further and automate the deployment. For example, when our Docker image has been built and pushed to the registry, we can request a redeployment on the PaaS which will pull the new image and deploy it. Figure 5.1 shows the overview of the system.

5.1 Build Tool

The build tool needs a manifest placed in the project which defines how to build it (needed libraries, how to compile, test and deploy). In this project, we are mainly concerned about Java, so we will only speak about some Java Build Tools.

5.1.1 Apache Maven

The officially supported build tool at ELCA is Apache Maven[1], so it is the one I mainly used during my project. Maven uses a Project Object Model (POM) in XML as the manifest. It downloads the needed dependencies from the Maven Central Repository (or an internal repository) and stores them locally. Each POM can have a POM parent and inherits its configuration, so we can have a powerful hierarchy.
Moreover, the POM can defines a dependency management, which can be used to predefined the versions of the dependencies. For example, Spring Boot and Spring Cloud have their own POM "starter-parent" which already includes the needed plugins and libraries to quickly create a Spring Boot or Cloud project. These starter parents also include a dependency management which already contains the versions of the Spring libraries, so it will be easier to add a dependency during the development (no need to look for the version).

[1] Apache Maven: `https://maven.apache.org/`

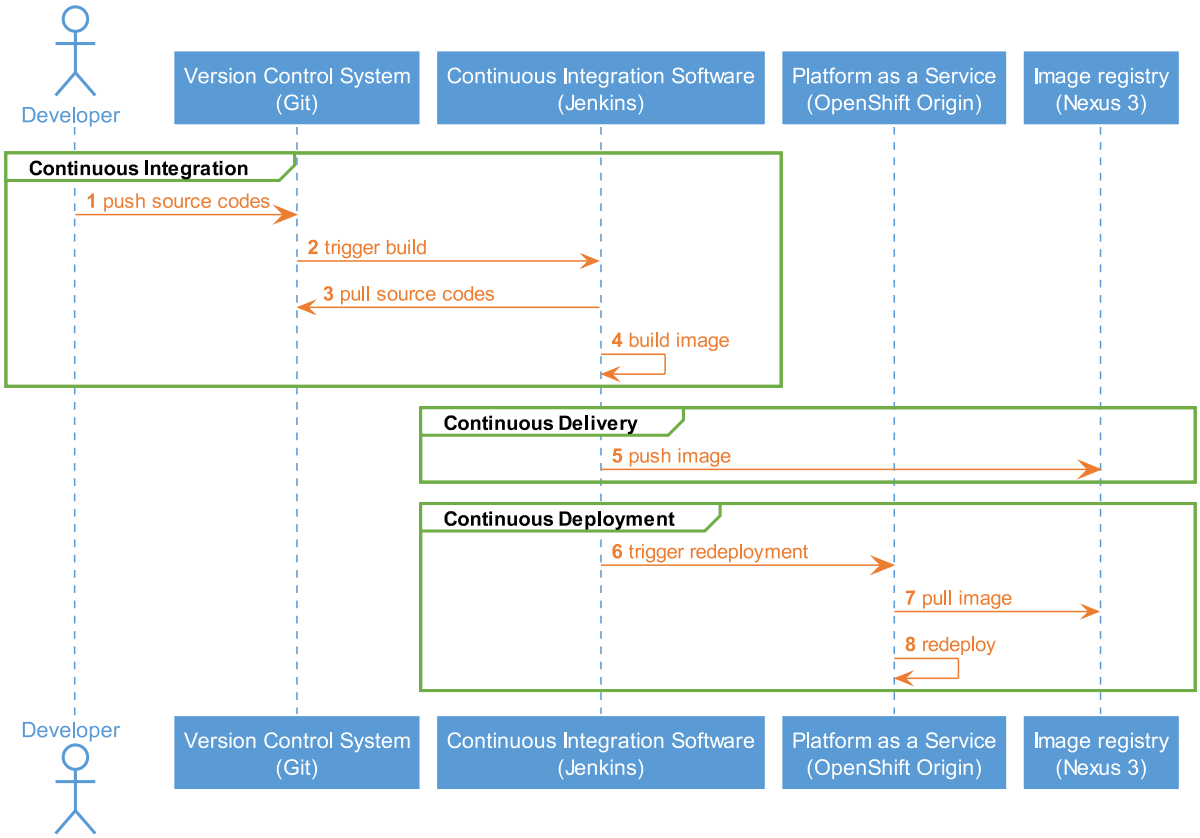


Figure 5.1: CI/CD & CD Overview

5.1.2 Gradle

Another interesting build tool is Gradle[2] which is based on Maven. Gradle's manifest *build.gradle* is more compact and readable than the Maven's *pom.xml*. One interesting feature of Gradle is the Gradle Wrapper. It is a small script which downloads and installs the specified version of Gradle if you haven't the correct version. This avoid some issues about the compatibility between the different versions. Note that a Maven Wrapper also exists, but this feature is built-in for Gradle.

5.2 Version Control System

5.2.1 Git

Nowadays, Git[3] is by far the most popular VCS thanks to GitHub[4]. Git is a distributed version control system, which means that each developer will have a full copy of the repository locally.

When you use Git in the context of a microservices architecture, you may wonder if it is better to create a repository per service or to group many services in one repository. Following the twelve-factors app (see section 2.1.1), you should put only one service per repository. But if you use Docker Compose (see section 4.1.1.5) to deploy your services, it might be simpler to put everything in one repository or to use Git Submodules.

[2]Gradle: `https://gradle.org/`
[3]Git: `https://git-scm.com/`
[4]GitHub: `https://github.com/`

5.2.2 Apache Subversion

Another VCS is Apache Subversion[5], which is a centralized one: each developer push his/her code to a central server which is the only one to contains the history of the repository. One interesting feature, which Git doesn't have, is the ability to clone only a folder of the repository and this folder acts like a sub-repository. This is very convenient for a microservice system because we can use only one repository and each service will have its own sub-repository inside and each developer can choose which sub-repository (s)he wants to clone.

5.2.3 Workflow

Having a VCS is nice, but how would you use it? The workflow is a set of rules which defines the organization and the utilization of the repository. Here are some possible workflows:

- **Centralized Workflow**: this is the most intuitive one. Every developer pushes his/her code into the master branch.
- **Feature Branch Workflow**: we use one branch for each feature and when a feature is finished, we create a pull request to the master branch.
- **Gitflow Workflow**: it is similar to feature branch, but divides the master branch into some more specific ones:
  - **master**: contains only the releases
  - **hotfix**: to quickly fix a release
  - **release**: to prepare a release
  - **develop**: the feature-branches merge into this one and wait for a release

In the context of microservices, I would select a simple workflow, such as the centralized one, because each service is quite small and developed by a small number of developers. But given the distributed governance, each repository can also have their own workflow.

5.3 Continuous Integration Software

5.3.1 Jenkins

One of the most known on premise continuous integration software (CIS) is probably Jenkins[6]. It is composed of a master and a set of slaves: the master contains the configuration and schedules the builds which occur on slave nodes. The builds can be triggered manually, when there is a commit on the VCS or regularly.
Given the fact that we use Jenkins to build some Docker images, we need some prerequisites:

- Install a plugin to build Docker images. The one tested is "CloudBees Docker Build and Publish plugin"[7]
- Have at least one slave with Docker installed.

For a Maven project, we configure Jenkins as for a common Maven project, but we add a "Post-build" step which "build and publish the Docker image" with the *Dockerfile* in the repository.
If we only have a simple Docker image to build, we can setup a "freestyle project" in Jenkins and set the "Build" step to "build and publish the Docker image".

5.3.2 Cloud

If you are looking for a CIS in the cloud, there is Travis CI[8] or CircleCI[9] for instance.

[5] Apache Subversion: `https://subversion.apache.org/`
[6] Jenkins: `https://jenkins.io/`
[7] CloudBees Docker Build and Publish plugin: `https://wiki.jenkins-ci.org/display/JENKINS/CloudBees+Docker+Build+and+Publish+plugin`
[8] Travis CI: `https://travis-ci.com/`
[9] CircleCI: `https://circleci.com/`

They are all based on the same principle: we put a configuration file in the root of our repository. Then, the CIS will pull our repository and build our project according to the given configuration.

5.4 Repository Manager

Even if we would probably need a repository which also supports other artifact types (jar, pom, etc.) following the languages used, we only focus on Docker images and we need a Docker registry (or a repository manager which supports the Docker registry) to store them.

5.4.1 Docker Hub

The default Docker registry is Docker Hub[10] and it contains the base images of many Linux distributions (e.g. Alpine, Debian) and also images with preinstalled packages or software (e.g. NGINX, Redis, MySQL, Node.js). Anyone can push publicly or privately their own Docker images on this registry.

5.4.2 Docker Registry

If you prefer to host yourself your own images, the easiest way is probably to use the Docker Registry[11] image. You just have to run the Docker image and you will be able to push your images into it.

5.4.3 Nexus Repository

ELCA uses Nexus Repository[12] as the repository manager and the third version supports Docker images (even in the OSS version).
There are three types of Docker registry in Nexus:

- **Proxy Repository for Docker**: used to proxy another repository. The common use is to proxy Docker Hub to cache the images and to control the used images.
- **Hosted Repository for Docker (Private Registry for Docker)**: a private Docker registry.
- **Repository Groups for Docker**: a set of repository which behaves like only one. Nexus will try to pull the requested image from each repository until the first success if any.

Nexus binds an HTTP and/or and HTTPS port(s) for each private Docker registry to isolate them with different access permissions. So, we need to expose at least one port per Docker repository. For instance, you can reserve all ports from 5000 to 5999 (included) of the machine where Nexus is installed and when we create a new Docker repository, it will select the first available port and binds on it (we can automatize this with to some Groovy scripts).

5.4.4 Artifactory

Another repository manager which can host Docker images is Artifactory[13], but only the pro version supports Docker.
Artifactory is very similar to Nexus (has also three types of Docker repositories), but it is possible to bind a subdomain to each Docker repository, which is more convenient than using ports.

[10] Docker Hub: `https://hub.docker.com/`
[11] Docker Registry: `https://docs.docker.com/registry/`
[12] Nexus Repository: https://www.sonatype.com/products-sonatype
[13] Artifactory: `https://www.jfrog.com/artifactory/`

Part III

Implementation

Chapter 6

Microservice Implementation

As said previously, we are mainly going to focus on the backend services developed in Java with the Spring framework.

6.1 Libero

To implement a microservice architecture, we need to have an application. Libero is an internal project intended to occupy the people which hasn't any assigned projects at ELCA. The goal is a web application which sells combined offers (transportation & leisure).
Libero has a high interest to be developed in a microservice approach because the developers are only on the project for a short time (i.e. a few weeks). Therefore, the on-boarding should be fast. Ideally, each person would have enough time to develop and deploy one service.

6.2 Spring

Spring[1] is an Open-Source Java Framework which facilitates the development of Java applications. Spring is mainly based on the inversion of control principle implemented by the dependency injection.

6.2.1 Dependency Injection

In the inversion of control (IoC) design principle, instead of calling the framework, we register to it and it will call us when needed.
Dependency injection (DI) is a type of IoC where a client does not have to instantiate and call the service, but an instance of this service is injected by the framework instead. This injection can be done through the constructor, a setter or an interface.

6.2.2 Spring Boot

Spring Boot allows to create standalone Spring application by, for instance, embedding a web server (Tomcat or Jetty).

6.2.2.1 Beans

The dependency injection mechanism is handled by beans. You can declare and instantiate a `@Bean` in a `@Configuration` class (see listing 6.1) and tell Spring to inject the instance with `@Autowired` (see listing 6.2).

[1] Spring: `https://spring.io/`

Listing 6.1: Example of a configuration class which declares a bean

```
@Configuration
public class CustomConfiguration {

    @Bean
    public Foo foo() {
        return new Foo();
    }

}
```

Listing 6.2: In this example, the variable `foo` is initialized with `null` during the construction and just after that, Spring injects the instance from listing 6.1 in this variable.

```
class Bar {
    @Autowired
    Foo foo = null; // null will be replaced by the instance
}
```

6.2.2.2 Auto-Configuration

In a Spring Boot application, all classes annotated with `@Configuration` will be loaded. But if you create a library (used as a dependency), you have to explicitly declare them in */src/main/resources/META-INF/spring.factories*.

6.2.2.3 Configuration Properties

The properties of a Spring application are defined in *application.properties* (or *application.yml*). Some custom properties can be defined in class annotated `@ConfigurationProperties` (and enabled in a configuration class with `@EnableConfigurationProperties`).

It is possible to set a property from another one or environment variable with a default value (see listing 6.3).

Listing 6.3: Set the name of the application as the name of the host or a random string.

```
spring.application.name=${spring.cloud.client.hostname:${random.value}}
```

6.2.2.3.1 Documentation The documentation of your properties can be written in */src/main/resources/META-INF/spring-configuration-metadata.json* (see listing 6.4).

6.2.2.3.2 Environment Variable Each property can also be set by the environment variables and this method has a higher priority than the configuration file method. For instance, to set the property `server.port`, use the variable `SERVER_PORT`.

6.2.2.3.3 Testing Properties If your tests need to run with other properties (e.g. deactivate libraries), you can use the annotation `@TestPropertySource` (see listing 6.5).

Note that the YAML extension is not supported for this feature.

Listing 6.4: Example of properties documentation

```
{
    "properties": [
        {
            "name": "kubernetes.discovery.enabled",
            "type": "java.lang.Boolean",
            "description": "Kubernetes Discovery is enabled. ",
            "defaultValue": "true"
        },
        {
            "name": "kubernetes.discovery.kubernetes-url",
            "type": "java.lang.String",
            "description": "The URL to access to the Kubernetes REST API. ",
            "defaultValue": "https://kubernetes.default.svc.cluster.local"
        }
    ]
}
```

Listing 6.5: Use testing properties defined in *test.properties* (located in *src/main/resources*).

```
@RunWith(SpringRunner.class)
@SpringBootTest
@TestPropertySource(locations = "classpath:test.properties")
public class ApplicationTests {

    @Test
    public void contextLoads() {
    }

}
```

6.2.3 Spring Web MVC

To develop web service following the MCV pattern, there is Spring Web model-view-controller (MVC)[2].

6.2.3.1 Model

A model is represented by a Java class with a constructor without any arguments, the setters and getters of all exposed fields. Thanks to Jackson JSON Processor[3], we can handle JSON document as Java class.

6.2.3.2 View

Our services are often RESTful ones, so the view usually just convert the model into JSON. But we can also render static web pages with a template engine, such as Thymeleaf[4].

6.2.3.3 Controller

A controller is a class annotated `@Controller` or `@RestController` (if the controller serves only REST content) and the methods of this class are able to handle requests. The annotation `@RequestMapping` defines the listened requests (e.g. path(s), HTTP method(s)). Listing 6.6 shows an example of a CRUD controller in Spring.

[2]Spring Web MVC: `https://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html`

[3]Jackson JSON Processor: `http://wiki.fasterxml.com/JacksonHome`

[4]Thymeleaf: `http://www.thymeleaf.org/`

Listing 6.6: Spring controller which exposes the REST API for a CRUD.

```
@RestController
@RequestMapping(path="/resources")
public class ResourceController {
    @Autowired
    private ResourceRepository repository;

    @GetMapping
    public List<Resource> getResources() {
        final List<Resource> resources = repository.findAll();
        return resources;
    }
    @PostMapping
    public ResponseEntity<?> createResource(@RequestBody Resource resource) {
        final Resource savedResource = repository.save(resource);
        final URI location = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")
                .buildAndExpand(savedResource.getId()).toUri();
        return ResponseEntity.created(location).body(savedResource);
    }
    @GetMapping(path = "/{id}")
    public ResponseEntity<?> readResource(@PathVariable(name = "id") String resourceId) {
        final Resource resource = repository.findOne(resourceId);
        if (resource == null) {
            return ResponseEntity.notFound().build();
        }
        return ResponseEntity.ok(resource);
    }
    @DeleteMapping(path = "/{id}")
    public ResponseEntity<?> deleteResource(@PathVariable(name = "id") String resourceId) {
        try {
            repository.delete(resourceId);
            return ResponseEntity.noContent().build();
        } catch (IllegalArgumentException e) {
            return ResponseEntity.notFound().build();
        }
    }
    @PutMapping(path = "/{id}")
    public ResponseEntity<?> fullyUpdateResource(@PathVariable(name = "id") String resourceId,
            @RequestBody Resource resource) {
        final Resource oldResource = repository.findOne(resourceId);
        if (oldResource == null) {
            return ResponseEntity.notFound().build();
        }
        resource.setId(oldResource.getId());
        final Resource newResource = repository.save(resource);
        return ResponseEntity.ok(newResource);
    }
    @PatchMapping(path = "/{id}")
    public ResponseEntity<?> partiallyUpdateResource(@PathVariable(name = "id") String
    resourceId,
            @RequestBody Resource resource) {
        final Resource oldResource = repository.findOne(resourceId);
        if (oldResource == null) {
            return ResponseEntity.notFound().build();
        }
        if (resource.getField() != null) {
            oldResource.setField(p.getField());
        }
        final Resource newResource = repository.save(oldResource);
        return ResponseEntity.ok(newResource);
    }
}
```

6.2.4 Discovery Client

Spring has an abstraction of the service discovery implementation (see section 2.6.3) called `DiscoveryClient`[5]. This interface allows to get:

- **The list of services**: get the service ID (`String`) of all known services.
- **The list of service instances**: given a service ID, give all its instances (a.k.a. pods in Kubernetes).

A service instance is abstracted by an interface which exposes the following properties:

- **Service ID**: the service ID of the instance
- **Host**: the hostname of the instance
- **Port**: the port exposed by the instance
- **URI**: the URL in order to connect to the instance
- **Metadata**: additional information stored as key-value pairs

The Spring version of Zuul (see section 7.1.2) uses the discovery client abstraction and there exists only an implementation for Eureka and Consul (respectively sections 7.4.3 and 7.4.2). So, we can use Zuul with one of them.

6.2.4.1 Kubernetes Discovery

Considering that we deploy on OpenShift Origin which is built on top of Kubernetes, we would prefer to use the service discovery provided by Kubernetes instead of deploying ours and having two layers of service discovery.
To have another discovery client implementation and to avoid deploying two independent service discoveries, a discovery client has been implemented for Kubernetes which calls the Kubernetes REST API[6] to get the available services and pods. Due to the lack of documentation, the implementation is inspired from the source code of Spring Cloud Consul Discovery[7]. The primary use case was to have automatic service discovery in Zuul (see section 7.1.2) by using the Kubernetes built-in service discovery.
To be able to call the REST API from the container, the requests should be authenticated with a token. This token is located inside the container because at the creation of the container, Kubernetes attach a volume which contains the token and the namespace of the service account[8]. So, we need to start the container with a service account which has the permission of making request to the REST API in order to get the list of services and pods from this container.
Getting the list of services is quite easy from the REST API. But getting the pods of a service requires to parse the label selector of the service and find all pods which match these labels.
Implementing the discovery client interface was not enough to make Zuul working, because it uses the discovery client to get the list of services and Ribbon (see section 7.5) to route the requests to the instances (see figure 7.1). So, a Kubernetes version of Ribbon has also been implemented.

6.2.5 Spring Cloud Stream

Spring Cloud Stream[9] abstracts the message broker (see section 2.6.5) by a binder and can be used for asynchronous communication between services (see section 3.2.1).
As seen in figure 6.1, the application sends and receives messages through a binder which connects to a middleware (i.e. a message broker).
At this time, there are only two official binder implementations:

[5]Spring Discovery Client: `https://spring.io/guides/gs/service-registration-and-discovery/`
[6]Kubernetes REST API: `https://kubernetes.io/docs/api-reference/v1/operations/`
[7]Spring Cloud Consul Discovery: `https://github.com/spring-cloud/spring-cloud-consul/tree/master/spring-cloud-consul-discovery`
[8]Kubernetes service account: `https://kubernetes.io/docs/user-guide/service-accounts/`
[9]Spring Cloud Stream: `https://cloud.spring.io/spring-cloud-stream/`

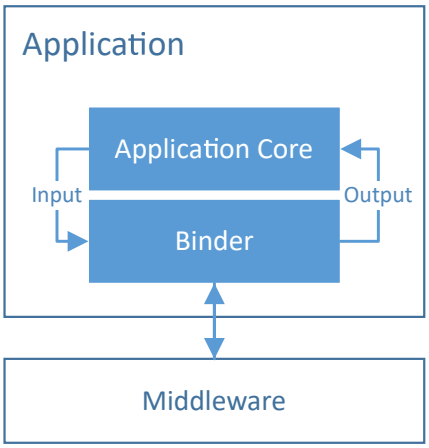


Figure 6.1: Spring Cloud Stream abstracts the middleware by a binder.

- RabbitMQ[10]
- Apache Kafka[11]

It would be nice to have other implementations, such as Redis or Hazelcast. An implementation for Hazelcast has been started, but due to the lack of time, the binder is not working yet.
Spring uses the concept of channels to route the messages to the right destination: the publishers send messages to a named channel and the subscribers will receive the messages if they bind to the same channel.
A service can also connect to many different message brokers. We can tell which channel goes to which binder and each binder can have a specific environment in order to connect to different message broker (see listing 6.7).

Listing 6.7: *application.yml* of a service which connects to two RabbitMQ servers. The application sends the Hystrix stream to `monitoring-broker` and the channel `customChannelName` to `custom-broker`.

```yaml
spring:
  cloud:
    stream:
      bindings:
        customChannelName:
          binder: custom-binder
        hystrixStreamOutput:
          binder: hystrix-stream-binder
      binders:
        custom-binder:
          type: rabbit
          environment:
            spring:
              rabbitmq:
                host: custom-broker
        hystrix-stream-binder:
          type: rabbit
          environment:
            spring:
              rabbitmq:
                host: monitoring-broker
```

We selected RabbitMQ over Kafka for the following reasons:

[10] RabbitMQ: `https://www.rabbitmq.com/`
[11] Apache Kafka: `https://kafka.apache.org/`

- **official Docker image**: there is an official Docker image (`https://hub.docker.com/_/rabbitmq/`).
- **simpler**: more complex deployment.
- **ephemeral messages**: we do not need to keep the messages after they are delivered.

6.3 Microservice Foundation

In a monolithic application, we build the foundation usually once, but we should implement many services in a microservice application. Therefore, the process of starting the development of a new service should be as easy as possible.

6.3.1 Spring Initializr

To quickly bootstrap a Spring project, there is Spring Initializr[12]. You can select the build tool (Maven or Gradle) or which dependencies will be already included in your application and it will create the stub for you.

6.3.2 Spring Tool Suite

If you use the Spring Tool Suite[13] (a Spring version of Eclipse[14]), the integrated development environment (IDE) integrates the Spring Initializr and the Spring Boot Dashboard which allows to start and manage many applications locally (i.e. run a distributed system locally).

6.3.3 POM Dependency Management

We can have our own dependency management (see section 5.1.1) which extends the Spring Cloud one and adds additional dependencies, such that SpringFox, Keycloak and homemade libraries.

6.3.4 POM Parent

After creating many services, it was noticed that the same properties (e.g. dependencies) were often added to get a complete skeleton (with Swagger for instance). So, we can have a POM parent (see section 5.1.1) which defines the common dependencies, such as SpringFox.

6.4 Dockerization

For a compiled language, such as Java or Scala, we more likely compile the project locally and only put the executable in the Docker image to have the most lightweight image possible. For an interpreted language, such as Python or JavaScript, we have no choice other than putting the source code in the image.

6.4.1 Maven Application

There are two ways to dockerize a Maven project:

1. **Docker Maven Plugin**: Maven will build the Docker image as an additional step. It is quite difficult to configure the plugin properly, but the build process is simplified afterward.
2. **Maven Dockerfile**: we keep Maven as it is and wrap the project with a *Dockerfile* (see figure 6.2). So, we have to successively built Maven and then Docker.

[12]Spring Initializr: `http://start.spring.io/`
[13]Spring Tool Suite: `https://spring.io/tools/sts`
[14]Eclipse: `https://eclipse.org/`

The second solution has been selected because it is much simpler to understand and configure. Moreover, the configuration in the CIS is more similar for other languages.

6.4.1.1 Docker Maven Plugin

There are two Docker Maven plugins available: one from Spotify[15] and the other one from fabric8[16]. They are quite similar and we can easily build and push our image directly with Maven with in one command line.

6.4.1.2 Maven Dockerfile

The *Dockerfile* of a Maven project (listing 6.8) copies the executable built by Maven, exposes the required ports and defines the entrypoint (i.e. run the application).

Figure 6.2: The directory tree of a dockerized Maven project. *Dockerfile* and *.dockerignore* are respectively shown in listing 6.8 and 6.10.

Note that we have limited the Java heap size to reduce the memory consumption of our services. We noticed that when the services are running on OpenShift, each of them uses more than 2 GB of memory because the physical host has a lot of RAM. So, after trying to reduce as much as possible, we end up with less than 500 MB with the configuration in listing 6.8.

Listing 6.8: Dockerfile for a Maven project

```
FROM openjdk:jre
MAINTAINER tdt
COPY target/*.jar /opt/app.jar
EXPOSE 8080
ENTRYPOINT ["java", \
    "-Xms2m", \
    "-Xmx64m", \
    "-jar", \
    "/opt/app.jar"]
```

6.4.2 NodeJS Application

Nowadays, developing in JavaScript usually means that we use a source-to-source compiler (from TypeScript[17] to JavaScript or just to bundle and compress the JavaScript for instance).
In this case, we wonder if the source code should be transpiled inside or outside of the container. The best practice seems to build inside the container and use the layer caching feature of Docker to cache dependencies. The trick is to firstly copy *package.json*, install the dependencies and then copy the rest of the project (see listing 6.9).

6.4.3 Dockerignore

When you build a Docker image, Docker firstly sends the context (the directory which contains *Dockerfile*) to the Docker daemon.
You should avoid sending unnecessary files by ignoring them with *.dockerignore* in the same directory as the *Dockerfile* (see figure 6.2). In the case of a Maven application, we only need to send the executable (see listing 6.10).

[15] Docker Maven Plugin Spotify: `https://github.com/spotify/docker-maven-plugin`
[16] Docker Maven Plugin fabric8: `https://dmp.fabric8.io/`
[17] TypeScript: `https://www.typescriptlang.org/`

Listing 6.9: Dockerfile for a NodeJS project

```
FROM node:latest
MAINTAINER tdt
COPY package.json .
RUN npm install --production
COPY . .
RUN npm run build
ENTRYPOINT ["node", "server.js"]
EXPOSE 9000
```

Listing 6.10: *.dockerignore* for a Maven application. We also exclude a *docker* folder which might contains additional files to build the image.

```
# exclude everything
*
# except
!target/*.jar # executable
!docker/ # additional files
```

6.5 Circuit Breaker

Every service which call another service or external API should implement the circuit breaker pattern (see section 2.6.2).

6.5.1 Hystrix

We selected Hystrix[18] from Netflix as an implementation of the circuit breaker because there is a Spring integration (`https://spring.io/guides/gs/circuit-breaker/`) and Netflix is quite known in the microservice world.
Hystrix is quite simple to use:

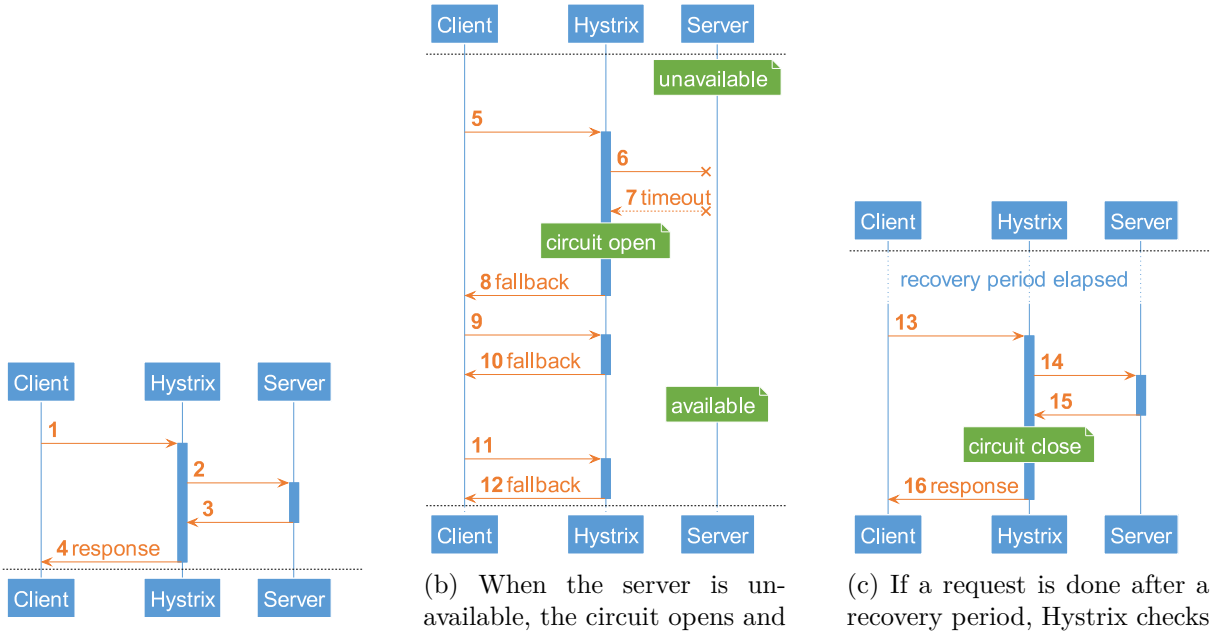
1. create a Spring `@Component` or `@Service` which represents a backing service (see listing 6.11). The behavior of the circuit breaker is configured with `@HystricCommand`.
2. inject this component with `@Autowired` in another class and call the methods which might fail.

Listing 6.11: BackingService.java

```
@Component
public class BackingService {

    @HystrixCommand(fallbackMethod = "getDefaultResource")
    public Resource getResource(String resourceId) {
        final Resource resource = ??? // might fail
        return resource;
    }

    public Resource getDefaultResource(String resourceId) {
        final Resource resource = ??? // define a fallback (e.g. default or cached resource)
        return resource;
    }

}
```

Figure 6.3 shows how Hystrix handles the requests.

[18] Hystrix: `https://github.com/Netflix/Hystrix/wiki`



(a) Requests are done through Hystrix

(b) When the server is unavailable, the circuit opens and every request will be short-circuited.

(c) If a request is done after a recovery period, Hystrix checks if the server is available. If it is, the circuit closes.

Figure 6.3: Circuit breaker with Hystrix

Hystrix also follows the bulkhead pattern and isolates the backing service requests. It executes each request on a separated thread. In this manner and if the response time is longer than expected (if any), the other requests are not affected by this wait.

6.6 Health Endpoint

The health of our services should be able to be checked and the common practice is to expose a `/health` endpoint (also known as the health endpoint monitoring pattern). This information should be in the same format for every service in order to easily monitor the services.
Spring Actuator exposes a health endpoint in JSON and we use this format (see listing 6.12) for all our services.
This format contains the overall status and some health indicators (see listing 6.13). A health indicator can be seen as a module of your service (e.g. backing service (MongoDB, RabbitMQ, ...), disk space, service discovery, configuration server) and each of them has a status.
By default, the possible statuses are (from the worst to the best):

1. **DOWN**: error
2. **OUT_OF_SERVICE**: less severe error
3. **UNKNOWN**: no information
4. **UP**: everything goes well

The overall status is equals to the status of the worst health indicator. In other terms, if one on the health indicator is DOWN, the overall status will also be DOWN.

6.6.1 Health Library

We may have some secondary health indicators (e.g. a service can work properly even if the configuration server (see section 7.6.2) is down because it only needs it while bootstrapping) which is not possible with the current computation of the overall status because if the configuration server is not available, the overall status of the service will also be DOWN.
I developed a health library which adds the feature of ignoring a set of health indicators. It overrides the default `OrderedHealthAggregator` by a custom `IgnoredOrderedHealthAggregator`.

Listing 6.12: JSON Schema for the health endpoint of Spring Actuator

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "Spring Boot Actuator Health Endpoint",
  "description": "The default health endpoint of Spring Boot Actuator",
  "$ref": "#/definitions/health",
  "patternProperties": {
    "^([a-zA-Z])+$": {
      "$ref": "#/definitions/health"
    }
  },
  "additionalProperties": false,
  "definitions": {
    "health": {
      "required": [
        "status"
      ],
      "properties": {
        "status": {
          "enum": [
            "DOWN",
            "OUT_OF_SERVICE",
            "UNKNOWN",
            "UP"
          ]
        },
        "description": {
          "type": "string"
        }
      }
    }
  }
}
```

To use this library, add it as a dependency and set the ignored health indicators in the application properties.

6.7 Project Lombok

There is repetitive code in the implementation of your services, such as getters and setters or declaration of the logger (see section 7.2.1.2.2.1) in each of your class.
Project Lombok[19] can generates these boilerplates by annotating your code and here are some examples:

- `@Data`: generate getters and setters for all fields
- `@AllArgsConstructor` & `@NoArgsConstructor`: generate constructors
- `@ToString`: override the `toString()` method with a new one which shows all the fields
- `@Slf4j`: create a SLF4J logger from the current class.

6.8 Microservice Testing

We have seen in section 2.4 that there are many kinds of tests but we will only focus on service testing because unit tests are very common and not specific to microservices.

6.8.1 Manual Testing Tools

Here are some tools to manually test a RESTful service.

[19] Project Lombok: `https://projectlombok.org/`

Listing 6.13: Example of the data returned by the `/health` endpoint of a service

```
{
  "description": "Kubernetes Discovery Client",
  "status": "UP",
  "discoveryComposite": {
    "description": "Kubernetes Discovery Client",
    "status": "UP",
    "discoveryClient": {
      "description": "Kubernetes Discovery Client",
      "status": "UP",
      "services": [
        "configuration-server",
        "elasticsearch",
        "gateway",
        "hazelcast",
        "keycloak-server",
        "kibana",
        "logstash",
        "monitoring-server",
        "test-service",
        "turbine-broker",
        "turbine-server",
      ]
    }
  },
  "diskSpace": {
    "status": "UP",
    "total": 10725883904,
    "free": 10258792448,
    "threshold": 10485760
  },
  "rabbit": {
    "status": "UP",
    "version": "3.6.6"
  },
  "refreshScope": {
    "status": "UP"
  },
  "hystrix": {
    "status": "UP"
  }
}
```

6.8.1.1 cURL

The simplest way to manually test the endpoints of a RESTful service is using cURL[20]. The command line tool is very powerful and you can perform any request.
The API documentations are commonly illustrated with request examples with cURL.

6.8.1.2 Postman

If you prefer a graphical user interface (GUI) over the command lines, there is Postman[21]. It has a nice GUI and keeps an history of your requests.

6.8.2 Contract Testing

The contract tests (see section 2.4.2.1) can be generated by the IDL (see section 3.5), but we will see how to define the contracts ourselves and how to generate these tests.

[20] cURL: `https://curl.haxx.se/`
[21] Postman: `https://www.getpostman.com/`

6.8.2.1 Spring Cloud Contract

Spring has a contract testing mechanism called Spring Cloud Contract[22]. It is not so easy to configure it properly, but it works well.

6.8.2.1.1 Contract The contracts are written in Groovy[23] (see listing 6.14) and each of them defines a request and the expected response.

Listing 6.14: A contract *createProduct.groovy* which describes a successful product creation.

```
org.springframework.cloud.contract.spec.Contract.make {
  request {
    method 'POST'
    url '/products'
    body("""
    {
      "name": "computer",
      "price": 1291.8
    }
    """)
    headers {
      header('Content-Type', 'application/json')
    }
  }
  response {
    status 201
  }
}
```

6.8.2.1.2 Server-Side We put the contracts in the source code of the server.
When we run the tests, an additional test class is created and run. This file is in */target/generated-test-sources/contracts/[...]/ContractVerifierTest.java* and contains the tests which perform the requests on the service.
Moreover, it pushes a stub version of your server in the configured repository. This stub only responds to the requests defined in the contracts.

6.8.2.1.3 Client-Side On the client side, we want to test the service in isolation from the server. For this purpose, we use the stub version of the server created previously.
When a test class requires the connection to a backing service, you should configure the stub runner which downloads and runs the stub version of the backing service at a specified port.

[22]Spring Cloud Contract: `https://cloud.spring.io/spring-cloud-contract/`
[23]Groovy: `http://groovy-lang.org/`

Chapter 7

Infrastructure

In addition to the infrastructure (called "environment" in part II) for building, deploying and managing our services, we also need to have an infrastructure deployed within the cluster. The term "infrastructure" refers to the inner one.
A similar infrastructure should be deployed beside each microservice project in order to apply some common design patterns (e.g. API gateway and service discovery) and to manage the services (e.g. configuration and monitoring).

7.1 API Gateway

An ideal implementation of the API gateway pattern (see section 2.6.1) should not only encapsulate the services but also provide some additional features such as:

- **Developer portal**: a place for the developers to browse the API documentation of all available services. It might also be a platform to define the APIs together.
  It should at least have a list of services and the links to their documentation.
- **Requests caching**: requests should be cached to improve the responsiveness of the system
- **Check tokens and authorization**: if the request should be authenticated, the gateway might check upfront if the request is authorized before forwarding it to the service.
  This adds a security layer and the unauthorized requests are rejected faster.
- **Header injection**: the possibility to inject values (e.g. a correlation identifier) in the header of all incoming requests.
- **Metrics and monitoring**: number of failed and succeed requests.
- **Requests throttling**: to prevent from a denial-of-service (DoS) attack or to limit the requests for each client, we should be able to limit the number of incoming requests.

7.1.1 Tyk

We firstly tried to use an open-source on premise product called Tyk[1].
It is mainly composed of three components:

- **the gateway**: routes every request. This component can be replicated behind a load balancer to handle huge traffic.
- **the dashboard**: manages the gateway(s), show the API usage, and control the portal
- **the portal**: allows the developers to publish and manage their APIs

The documentation of Tyk is quite minimalistic, so it was difficult to setup it properly.
Moreover, adding a new API is not very intuitive. We had to document the quite complex process and Tyk only supports documentation in Swagger and API Blueprint format.
It is not a bad product (it does its job and provides the common features), but we are limited to the provided features and it is not so easy to extend with more specific needs. Therefore, we preferred to not use it in a long term.

[1] Tyk: `https://tyk.io/`

7.1.2 Zuul

A more customizable solution is Zuul[2] which is developed and used by Netflix. The Spring Cloud version is deployed as a library of a Spring application.
Zuul already proposes some built-in features (see figure 7.1):

- Automatic service routing by using the discovery client (see section 6.2.4) and a custom mapping between the request path to the service identifier with a regular expression (called route locator).
- filters mechanism to handle the requests: each request pass through a set of filters and each of them can modify, reject or log the request. There are four filter types:
  - **pre-filter**: triggered before the forwarding to the service.
  - **routing filter**: forwards the request to the service and gets its response.
  - **post-filter**: called after getting the response from the service.
  - **error filter**: triggered when one of the other filters failed.

  We can easily create and add our own filters.
- Zuul uses Ribbon to load balance the requests to the instances of a service (more details in Ribbon in section 7.5).
- Ribbon uses Hystrix as circuit breaker to each of the service instance, so we can use the Hystrix Dashboard for monitoring (see section 7.2.3).

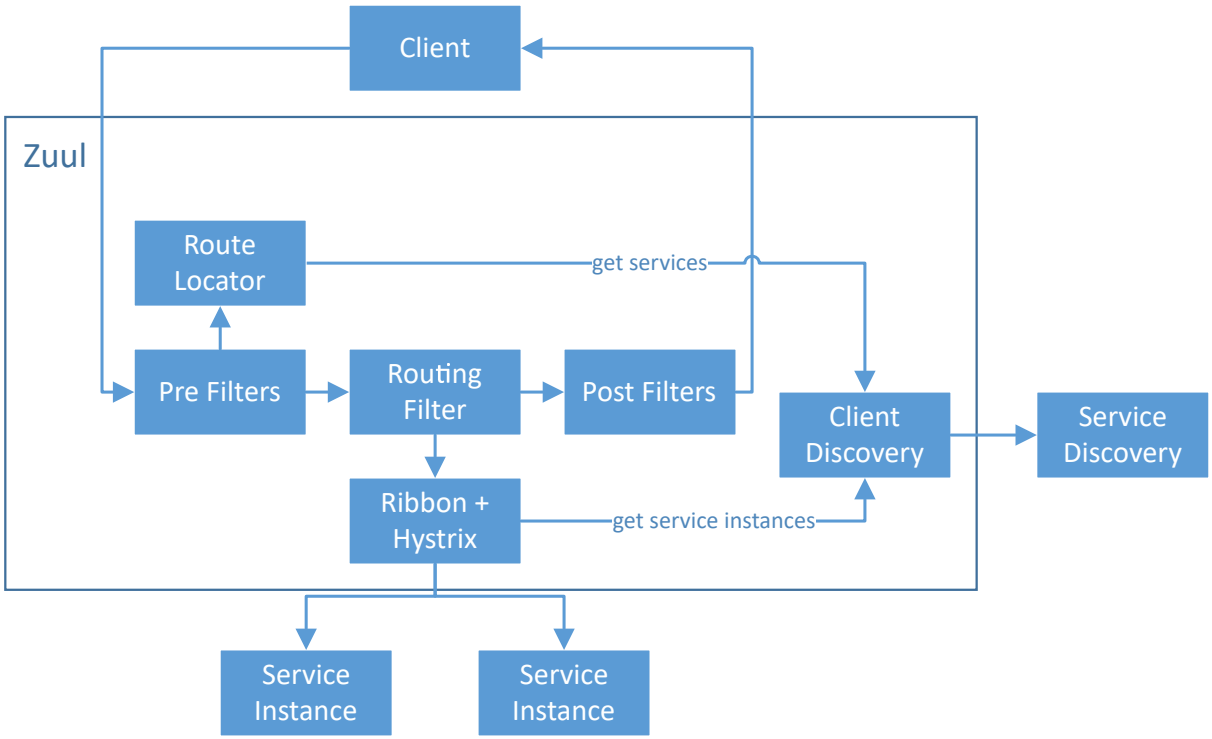


Figure 7.1: Zuul Overview. One interesting pre-filter is `PreDecorationFilter` which maps the request URL to a service ID using the route locator. The routing filter uses Ribbon to load balance the requests.

Given that Zuul is a library of a Spring application, we are able to customize and extend it as we wish. In counterpart, we have to implement nearly all features. Some of them has been implemented during this project:

- **Portal developer**: Thymeleaf is used to generate a web page which lists all available services with the links to their API documentation.

[2]Zuul: `https://github.com/Netflix/zuul/wiki`

The services do not have to use the same IDL, but they should expose their documentation at the same endpoint (such as `/doc`) in order to allow the gateway to know the location of the documentation.
This `/doc` endpoint can either redirect to the right path or contain the link to the documentation.

- **Header injection**: in a pre-filter, some key-value pairs (e.g. correlation ID) are injected in the request headers.
- **Metrics**: the logging is done in a post-filter to not slow the processing of the request.

7.2 Monitoring System

The monitoring of the service instances should be centralizing because we cannot manage to monitor each instance individually.
There are two kinds of monitoring data:

- Metrics: statistics about the host and the service (e.g. CPU and memory usage or health of the service).
- Logs: information about the service execution (e.g. stack traces or incoming requests).

7.2.1 Monitoring Data Centralization

The process to centralize our monitoring data (metrics and logs) has five distinct steps (see figure 7.2):

1. **Collect**: gets the data and aggregates them by sending to the next step. This is done in each service instance.
2. **Process**: the data need to be filtered, enhanced before storing them. This can be centralized or separated for each type of processing.
3. **Store**: keep the processed data in a (clustered) database.
4. **Visualize**: view the data to verify how things are going.
5. **Alert**: alert us when anomalies are detected.

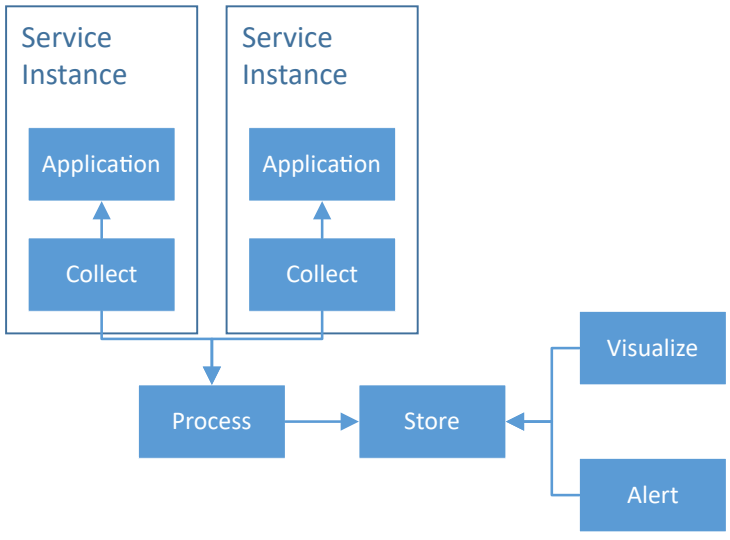


Figure 7.2: Overview of the Monitoring Data Centralization

7.2.1.1 Stack

We usually use a stack (set of products) which partially or fully covers the process. We choose to implement the Elastic stack with Beats due to its maturity and popularity.

7.2.1.1.1 Elastic Stack The Elastic stack (formerly known as ELK stack) from Elastic[3] is probably the most known one and the main products are:

- **Elasticsearch**: store
- **Logstash**: process
- **Kibana**: visualize

Elastic proposes Beats and X-Pack[4] to complete the Elastic stack. A variant is the EFK stack where Logstash is replaced by Fluentd.

7.2.1.1.2 TICK Stack The TICK Stack from InfluxData[5] is composed of:

- **Telegraf**: collect & process
- **InfluxDB**: store
- **Chronograf**: visualize
- **Kapacitor**: notify

7.2.1.2 Collect

7.2.1.2.1 Instrumentation The instrumentation of a program means adding more instructions in an existing program to be able to profile it or manage its logs. The instrumentation lets us monitor the program very deeply, but the service will be less exportable because we write code for a specific monitoring system.
At ELCA, we prefer to avoid this kind of specific code because we usually develop for our clients and they might prefer to monitor in a different way than ours in development.
For an internal project, instrumentation might be a good solution because it extends the monitoring possibilities.

7.2.1.2.1.1 Fluentd A good instrumented logging library is Fluentd (more details in section 7.2.1.3.2) because it is compatible with many languages and output components.

7.2.1.2.2 Agent Instead of instrumenting our application, we prefer writing logs into a file or into the standard console output and use an agent to forward this logs (and additional metrics) further. In this way, our services are be more standardized.
An agent is a program which runs continuously, autonomously and might performs repetitive tasks on another program. In our case, this agent is highly-coupled with the instance and we should have a set of agents for each instance.
An advantage of writing logs into a file is when the host crashes and the logs are not yet sent but are already on the disk, they are not lost and might be sent when the host reboots.

7.2.1.2.2.1 Logging Utility We should wonder how to easily write these logs in a standardized way. We can achieve that by using a logging utility, such that Logback[6] or even better with an abstraction such that SLF4J[7].
The logging utility can usually also handle the rotation of the log files. The log rotation mechanism allows you to have a limited size of your logs on the host by archiving the files and deleting the oldest ones. Which is great because we want to keep our containers as lightweight as possible and we don't need to keep the log files which were already shipped further.

[3]Elastic: `https://www.elastic.co/`
[4]X-Pack: `https://www.elastic.co/products/x-pack`
[5]InfluxData: `https://www.influxdata.com/open-source/`
[6]Logback: `http://logback.qos.ch/`
[7]SLF4J: `https://www.slf4j.org/`

7.2.1.2.2.2 Beats After logging into rotated files, we can use Beats[8] to collect and ship these data in a JSON form. Here are the relevant Beats in our case:

- Filebeat[9]: collects log files. It can lightly pre-process the data, such that aggregating multi-lines or adding fields.
- Metricbeat[10]: collects metrics about the host or external services, such that Mongo or Redis.
- Packetbeat[11]: collects metrics on the network.

7.2.1.2.2.3 Fluent Bit There is also Fluent Bit[12]. It is a lighter version of Fluentd (see section 7.2.1.3.2) written in C used to forward data without processing them. Fluent Bit can forward logs and metrics about the host.
It is also possible to use Fluentd as an agent if you prefer to process the data directly in each service instance.

7.2.1.2.2.4 Telegraf From the TICK stack, Telegraf[13] does the collection and the processing of the logs as an agent in each service instance.

7.2.1.3 Process

We often need to parse the data given by the collectors more heavily to get more insightful data. For instance, the default logs given by Spring Logback are composed by some lines of the form shown in listing 7.1. We can clearly recognize some information which can be parsed into different fields (e.g. date, time, log level or process identifier) in order to be able to filter on every ERROR logs for example.

Listing 7.1: A line of log given by Spring Logback

```
2016-10-26 13:13:45.520  INFO 68 --- [           main] s.b.c.e.t. TomcatEmbeddedServletContainer
    : Tomcat started on port(s): 8080 (http)
```

We usually use the processor to aggregate the data of all collectors. So even if you don't need to process the data, it is better to have a processor just for aggregation.

7.2.1.3.1 Logstash We can use Logstash[14] to process this data by defining some grok[15] patterns and route them with if-else statements (see figure 7.2).

7.2.1.3.2 Fluentd Fluentd[16] is an alternative to Logstash and it is hosted by the Cloud Native Computing Foundation. Instead of using algorithmic statements as Logstash, the logs are routed by using some tags and this approach gives cleaner configuration files.

7.2.1.4 Store

After collecting, aggregating and processing the monitoring data, we should store them in a database.

[8] Beats: `https://www.elastic.co/products/beats`
[9] Filebeat: `https://www.elastic.co/products/beats/filebeat`
[10] Metricbeat: `https://www.elastic.co/downloads/beats/metricbeat`
[11] Packetbeat: `https://www.elastic.co/products/beats/packetbeat`
[12] Fluent Bit: `http://fluentbit.io/`
[13] Telegraf: `https://www.influxdata.com/open-source/#telegraf`
[14] Logstash: `https://www.elastic.co/products/logstash`
[15] grok: `https://www.elastic.co/guide/en/logstash/master/plugins-filters-grok.html`
[16] Fluentd: `http://www.fluentd.org/`

Listing 7.2: *logstash.conf*

```
input {
  beats {
    port => 5044
  }
  rabbitmq {
    host => "${LOGSTASH_RABBITMQ_TURBINE_HOST:localhost}"
    user => "${LOGSTASH_RABBITMQ_TURBINE_USER:guest}"
    password => "${LOGSTASH_RABBITMQ_TURBINE_PASSWORD:guest}"
    exchange => "springCloudHystrixStream"
    key => "#"
    exclusive => true
    auto_delete => true
    type => "turbine"
  }
}
filter {
  if [@metadata][beat] {
    grok { match => { "[beat][hostname]" => "^%{NOTSPACE:[kubernetes][service]}-%{POSINT:[
    kubernetes][deployment]}-%{USERNAME:[kubernetes][pod]}$" } }
  }
  if [type] == "spring" {
    grok { match => { "message" => "^%{TIMESTAMP_ISO8601:[log][timestamp]}%{SPACE}%{LOGLEVEL:[
    log][level]} %{SPACE}%{NUMBER:[log][pid]}%{SPACE}---%{SPACE}%{SYSLOG5424SD:[log][threadname
    ]}%{SPACE}%{NOTSPACE:[log][javaclass]}%{SPACE}:%{SPACE}%{GREEDYDATA:[log][message]}$" } }
  } else if [type] == "turbine" {
    date { match => ["[data][currentTime]", "UNIX_MS"] }
    mutate {
      rename => {
        "[data][latencyExecute][99]" => "[data][latencyExecute][99.0]"
        "[data][latencyTotal][99]" => "[data][latencyTotal][99.0]"
      }
    }
  } else if [type] == "gateway-request" {
    date { match => ["[json][time]", "UNIX_MS"] }
  } else if [type] == "supervisor" {
    grok { match => { "message" => "^%{TIMESTAMP_ISO8601:[supervisor][timestamp]}%{SPACE}%{
    LOGLEVEL:[supervisor][loglevel]}%{SPACE}%{GREEDYDATA:[supervisor][message]}$" } }
  }
}
output {
  if [type] == "spring" {
    elasticsearch {
      hosts => "elasticsearch:9200"
      index => "application-%{+YYYY.MM.dd}"
      document_type => "%{type}"
    }
  } else if [type] in ["turbine", "monitoring-service-instance", "monitoring-agent", "supervisor
    "] {
    elasticsearch {
      hosts => "elasticsearch:9200"
      index => "monitoring-%{+YYYY.MM.dd}"
      document_type => "%{type}"
    }
  } else if [type] == "gateway-request" {
    elasticsearch {
      hosts => "elasticsearch:9200"
      index => "analytics-%{+YYYY.MM.dd}"
      document_type => "%{type}"
    }
  }
}
```

7.2.1.4.1 Elasticsearch The probably most known product to store monitoring data is Elasticsearch[17] which is a distributed search engine. It is also the heart of the Elastic stack.

7.2.1.4.2 InfluxDB An alternative is InfluxDB[18] from the TICK stack.

7.2.1.5 Visualize

Now that we have stored all our monitoring data, we want to visualize them in order to discover some correlations, for instance.

7.2.1.5.1 Kibana If you want to continue with the Elastic stack, there is Kibana[19] which can draw some graphs from the data given by Elasticsearch and show the aggregated logs as a table. To enhance the security of Kibana, you should install X-Pack to get "Security" which provides authentication and authorization.

7.2.1.5.2 Grafana A known alternative to Kibana is Grafana[20], which seems to be very good for drawing graph. But unfortunately, it doesn't seem to offer a log visualization.

7.2.1.5.3 Chronograf Chronograf[21] from the TICK stack also seems to be good for monitoring, but not logging.

7.2.1.6 Alert

To be able to see when issues occurred and investigate them is not enough because we want to be notified when they occur in order to fix them as fast as possible. To know when there is an issue, we should set some thresholds (e.g. minimum running instance of a service or maximum incoming requests).

7.2.1.6.1 Watcher X-Pack contains an alerting service called Watcher. An alert might be an email, a web hook, a Slack[22] message or a JIRA[23] issue, for instance.

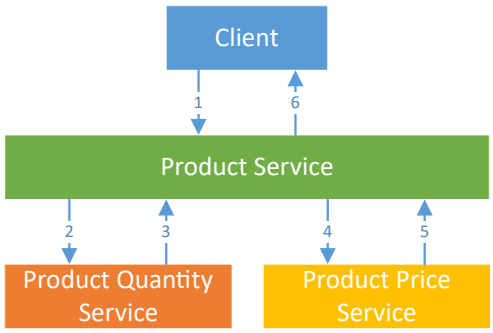
7.2.1.6.2 Kapacitor The alternative from TICK is Kapacitor[24]. It provides additional alerting options such as through TCP, the execution of a program or a Telegram[25] message.

7.2.2 Distributed Tracing

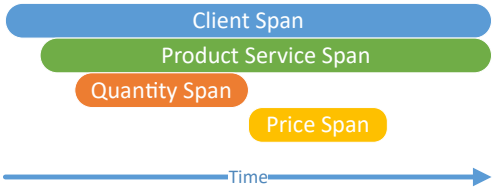
Casual monitoring with logs and metrics is not enough to debug a distributed system, because we don't have the correlation between the requests. To know the path taken by a request (the sequence of service instances called in order to respond to the request) and the time taken in each instance, we can use a distributed tracing system.
Most of them are based on Google's Dapper[26] and they use the same terminology:

- **Trace**: representation of a request handled by one or many processes (see figure 7.3a).
- **Span**: representation of the time taken by a process to handle its request (see figure 7.3b).

[17] Elasticsearch: `https://www.elastic.co/products/elasticsearch`
[18] InfluxDB: `https://www.influxdata.com/open-source/#influxdb`
[19] Kibana: `https://www.elastic.co/products/kibana`
[20] Grafana: `http://grafana.org/`
[21] Chronograf: `https://www.influxdata.com/open-source/#chronograf`
[22] Slack: `https://slack.com/`
[23] JIRA: `https://www.atlassian.com/software/jira`
[24] Kapacitor: `https://www.influxdata.com/open-source/#kapacitor`
[25] Telegram: `https://telegram.org/`
[26] Dapper: `https://research.google.com/archive/papers/dapper-2010-1.pdf`



(a) An example of the trace of a request.



(b) The spans produced by the request in figure 7.3a.

Figure 7.3: Distributed Tracing

7.2.2.1 Spring Cloud Sleuth & Zipkin

Spring Cloud Sleuth[27] instruments a Spring application to create the trace (by reading the incoming request headers from Spring MVC and adding the trace ID to the outgoing requests done by a `RestTemplate`).
The traces collected by Sleuth can be sent to Zipkin in order to visualize them. With the Spring version of Zipkin, we can send them through Spring Cloud Stream (see section 6.2.5).

7.2.2.2 OpenTracing

There exist many distributed tracing systems with different APIs and OpenTracing[28] is an effort to standardize these solutions. You still have to instrument your application, but it will be possible to change the tracer in a minimal effort.

7.2.3 Hystrix Dashboard & Turbine

As seen in section 6.5.1, we use Hystrix as a circuit breaker implementation. It usually exposes an Hystrix Stream which is a stream of the metrics of all circuits in the application.

7.2.3.1 Hystrix Dashboard

These Hystrix streams can be interpreted by the Hystrix Dashboard[29] which gives a visual interpretation of the circuit status with some useful data, such as the request rate, the health, the traffic volume and the error percentage.
Unfortunately, the dashboard can only listen on one stream at once and having a dashboard per service instance is not very convenient. To solve this problem, we can use Turbine.

7.2.3.2 Turbine

Turbine is an Hystrix Stream aggregator. The Turbine server aggregates the stream of many applications and exposes its own aggregated stream (called Turbine Stream). Usually, the Turbine server has a list of hosts (provided manually or by Eureka) and it fetches the stream of each of them.

7.2.3.2.1 Spring Cloud Stream The Spring version of Turbine provides a very interesting additional feature: we can also ask each host to send its Hystrix Stream to a message broker by using Spring Cloud Stream. On the other hand, the Turbine server gets all the stream elements also through Spring Cloud Stream. This offers the huge advantage to do not have to explicitly define the list of hosts (very useful for a microservice architecture where the containers might be

[27] Spring Cloud Sleuth: `https://cloud.spring.io/spring-cloud-sleuth/`
[28] OpenTracing: `http://opentracing.io/`
[29] Hystrix Dashboard: `https://github.com/Netflix/Hystrix/wiki/Dashboard`

very ephemeral). This list is inferred from the stream elements which go through the message broker. Moreover, this seems to be more scalable because the load is distributed among all hosts.

7.2.3.3 Send to the Monitoring Database

The Hystrix Dashboard allows only to monitor the circuit breakers over a short period. But it would be interesting to store these data in order to correlate them with other metrics or logs. Hence, we would like to redirect the Turbine stream to store the metrics into the monitoring database.
In our case (with the Elastic stack), Logstash gets a copy of the stream from RabbitMQ by adding a new queue to the Turbine stream exchange and consuming this new queue (see listing 7.2).

7.2.4 Health Server & Health Agent

The monitoring service embedded in Kubernetes only tells us if a component is available or not. The health endpoint of our services provides much more information about the state of the service by also giving the state of the submodules (e.g. Hystrix or database connection).
The main goal of the Health Agent and the Health Server is to collect and centralize these health information, so the clients and the administrator can have a general overview of the health of the whole application. For instance, a web client will be able to disable a component if the required services are unavailable.

7.2.4.1 Active Health Server

The simplest system is to have only a Health Server which periodically gets the list of available service instances from the service discovery, does a request on every `/health` endpoints to collect them and updates its internal database of health data (see figure 7.4a). The issue of this system is its scalability: the refresh rate lowers when the number of service instances increases.

7.2.4.2 Passive Health Server

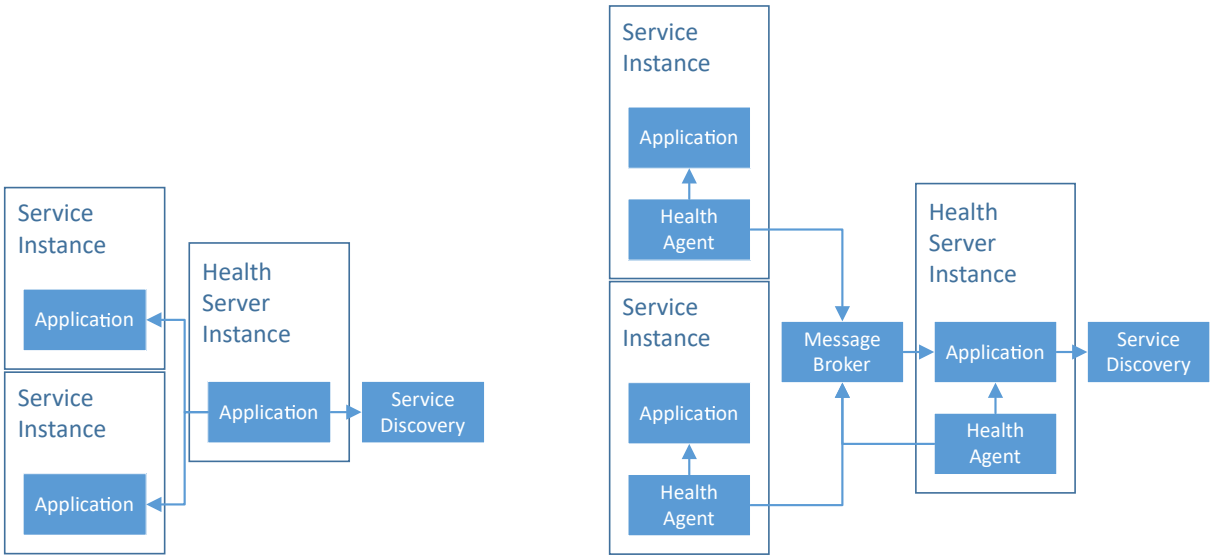
To make it more scalable, we inspire from Turbine and its stream. We have some agents which are containerized with each service instance and send the health information to the server. In this way, we prefer to use a message broker to buffer the messages (see figure 7.4b). Given the ephemeral aspect of these data, we can set a time-to-live of the messages in the broker to unload the server. In this mechanism, the server needs a way to update the database given a new health message.

7.2.4.3 Health Agent

The health agent is a small Spring application which gets the health information of its host by calling the `/health` endpoint, adds some information about the host (e.g. hostname, IP address) and finally sends these data to the Health Server through a message broker (e.g. RabbitMQ). We used the Spring Cloud Stream to be able to change the message broker afterwards.

7.2.5 Monitoring Server

We want to see the evolution of the number of service instances per service in the monitoring dashboard (e.g. Kibana). If the auto scaling is configured, we are able to easily see when a service has high traffic or when a service is completely down and has no available instance. For that purpose, a Monitoring Server was implemented. It regularly gets the list of available service instances and sends this information to the monitoring database (e.g. Elasticsearch).
To be able to create such graph in Kibana, the monitoring server creates an entry for each service instance. Each of these entries contains the service ID and the hostname of the service



(a) Active Health Server Overview

(b) Passive Health Server Overview

Figure 7.4

instance. In Kibana, in the Y-axis, there is the unique count of the instance hostname and the X-axis is the timestamp split per service ID.

7.2.6 Monitoring Agent

The monitoring agent replaces the health server and health agent (from section 7.2.4) because we find out that it is not necessary to have an overview of the health of all service instances. Each service should be able to deal with its own backing services thanks to the circuit breaker pattern.
But in case of a failure in one of our service instances (even if the service is still able to work properly), we want to know what happened in order to fix the problem.
The monitoring agent periodically checks the health endpoint of the instance it is attached to and whenever one of the health indicator or the overall health is not UP, it sends the content of the health to the monitoring database as a custom log (see section 7.2.7). In other words, it is a health agent which sends the data only when there might be an issue.
The agent is developed in Python because Java consumes too much memory and an agent should have a minimal impact on its host. Listing 7.3 shows the code of this agent.

7.2.7 Custom Logs

During the development of some infrastructure components, it happens quite often that we need to send additional data to the monitoring database (not just the logs and metrics of the service instances), which we will called custom logs. For instance, the monitoring server sends the available service instances and the gateway sends some information about incoming requests to the monitoring database (which is Elasticsearch in our case). So, we need to find an effective pattern to ship these custom logs from an application to the database.

7.2.7.1 Send to Elasticsearch

One way is to connect to Elasticsearch and create the documents. There is the Spring Data Elasticsearch[30] (not yet compatible with Elasticsearch 5) which is an abstraction to connect to Elasticsearch databases. The main drawback is the tight coupling between the applications and

[30]Spring Data Elasticsearch: `http://docs.spring.io/spring-data/elasticsearch/docs/current/reference/html/`

Listing 7.3: Code of the Monitoring Agent

```
import json, logging, os, sched, socket, time, urllib2
from logging.handlers import RotatingFileHandler

url = os.getenv('MONITORING_AGENT_URL', 'http://127.0.0.1:8080/health')
logFile = os.getenv('MONITORING_AGENT_LOG_FILE', '/var/log/container/monitoring-agent.log')
logMaxBytes = int(os.getenv('MONITORING_AGENT_LOG_MAX_BYTES', '10485760'))
logBackupCount = int(os.getenv('MONITORING_AGENT_LOG_BACKUP_COUNT', '4'))
interval = float(os.getenv('MONITORING_AGENT_INTERVAL', '60'))

request = urllib2.Request(url)
hostname = socket.gethostname()
ip = socket.gethostbyname(hostname)
scheduler = sched.scheduler(time.time, time.sleep)
logDirectory = os.path.dirname(logFile)
if not os.path.exists(logDirectory):
    os.makedirs(logDirectory)

logger = logging.getLogger("Monitoring Log")
logger.setLevel(logging.INFO)
handler = RotatingFileHandler(logFile, maxBytes=logMaxBytes, backupCount=logBackupCount)
logger.addHandler(handler)

def is_healthy(health):
    for key, value in health.iteritems():
        if type(value) is dict:
            if value['status'] != 'UP':
                return False
    return health['status'] == 'UP'

def data_to_json(data):
    return json.dumps(data, separators=(',', ':'))

def wrap_health(code, health):
    data = {
        'hostname': hostname,
        'ip': ip,
        'code': code,
        'health': health
    }
    return data_to_json(data)

def check():
    try:
        response = urllib2.urlopen(request)
    except urllib2.HTTPError as e:
        health = json.loads(e.read())
        logger.error(wrap_health(e.code, health))
    except urllib2.URLError as e:
        data = {
            'hostname': hostname,
            'ip': ip,
            'reason': str(e.reason)
        }
        logger.error(data_to_json(data))
    else:
        health = json.loads(response.read())
        if not is_healthy(health):
            logger.info(wrap_health(response.code, health))
    scheduler.enter(interval, 0, check, ())

check()
scheduler.run()
```

the database. Every application needs to know the credentials of the database and if we want to change the latter, we have to modify all applications.

7.2.7.2 Send to Logstash

Another solution is to send to Logstash which forwards to Elasticsearch. Note that we can also use Logstash to process the data before forwarding it.

7.2.7.2.1 HTTP Logstash has an input plugin which allows us to `POST` some logs as HTTP request.

7.2.7.2.2 Message Broker A better alternative than doing HTTP request to Logstash is to use a message broker as a middleware. Logstash has an input plugin for RabbitMQ, Kafka, ZeroMQ or even Redis. The logs are hence buffered and Logstash can process them as it can. The application also does not have to wait on Logstash if it is under pressure.

7.2.7.3 Send as Logs

If we step back again a bit, we see that we can also write these custom logs as standard logs and we should tell to the log shipper to also handle these additional logs. Do not forget also to rotate these logs and automatically delete the old files.
The best solution is to write as logs, because we use an existing log aggregation system. Using a message broker is also a good solution but we need to have an additional component in our already quite complex system. We can also use this alternative to have a second channel with a higher priority (if the application writes a huge number of logs and we want to be sure that we still get the application status nearly on real time).

7.2.8 Service Status

We might wonder how many different status a service should have. They are at least two (UP and DOWN) but there might exists some intermediary states where the service is still able to handle the requests (e.g. cannot connect to the configuration server or to the monitoring system) or the service can only handle some requests but not all.
We were thinking about a status system which tells the accurate availability of the service, but it was too complicated.
Finally, we find out that it is not necessary to have more than two states (UP and DOWN) for a service. Each service instance connects to its backing services through a circuit breaker and hence know when its backing services have some issues.

7.3 Multiple Processes

We have seen that we need to run some agents alongside with each service instance (i.e. run multiple processes per instance), but a Docker container can only run one process.

7.3.1 Kubernetes Pod

One solution is to use the Kubernetes Pod (which is a set of containers) to run one container with the service and some others for each agent (see listing 7.4). If the containers need to have a shared directory, they can use a volume.
The advantages of this method are:

- the Docker image of the service does not depend on the agents (i.e. monitoring system). We can use the same image for other monitoring systems.
- we fully use the monitoring system provided by OpenShift (see the console of each pod and execute some commands from the web interface).

But we depend too much on Kubernetes and it would be difficult to deploy your system on another platform.

Listing 7.4: `DeploymentConfig` of a multi-containers pod

```yaml
apiVersion: v1
kind: DeploymentConfig
metadata:
  name: example-service
spec:
  selector:
    deploymentconfig: example-service
  template:
    metadata:
      labels:
        deploymentconfig: example-service
    spec:
      volumes:
        -
          name: container-log
          emptyDir:
            medium:
      containers:
        -
          name: example-service
          image: example-service:latest
          volumeMounts:
            -
              name: container-log
              mountPath: /var/log/container
        -
          name: filebeat
          image: filebeat:latest
          volumeMounts:
            -
              name: container-log
              mountPath: /var/log/container
        -
          name: monitoring-agent
          image: monitoring-agent:latest
          volumeMounts:
            -
              name: container-log
              mountPath: /var/log/container
```

7.3.2 Supervisor

The solution documented by Docker to run many processes is to use Supervisor[31]. When Supervisor starts, it forks some child processes and monitors them (if a process stops, it may restarts it).

Instead of writing a configuration file which contains all configurations, we have one which includes all files in a directory (see listing 7.5). In this way, we can easily add a new or overwrite an existing configuration of an agent.

Listing 7.6 shows the configuration of the service.

[31] Supervisor: `http://supervisord.org/`

Listing 7.5: *supervisord.conf*

```
[supervisord]
nodaemon=true
logfile_backups=1

[include]
files=/etc/supervisor/conf.d/*.conf
```

Listing 7.6: *java.conf*

```
[program:java]
command=java -Xms2m -Xmx64m -jar /opt/app.jar
redirect_stderr=true
stdout_logfile=/dev/stdout
stdout_logfile_maxbytes=0
```

7.3.2.1 Base Image with Supervisor and Agents

Instead of installing and configuring Supervisor and the agents in each Docker images, we prefer to have a base image which already contains Supervisor, the common agents and their configurations (see listing 7.7). The Dockerfile for each service is hence simplified (see listing 7.8).

Listing 7.7: *Dockerfile* of the Java base image which includes Supervisor, Filebeat and the monitoring agent.

```
FROM openjdk:jre
MAINTAINER tdt

RUN apt-get update && apt-get install -y \
    apt-transport-https \
    supervisor \
 && curl https://packages.elasticsearch.org/GPG-KEY-elasticsearch | apt-key add - && echo "deb
    https://artifacts.elastic.co/packages/5.x/apt stable main" | tee -a /etc/apt/sources.list.d/
    elastic-5.x.list \
 && apt-get update && apt-get install -y \
    filebeat \
 && rm -rf /var/lib/apt/lists/*

COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf

COPY filebeat.yml /etc/filebeat/filebeat.yml
COPY filebeat.conf /etc/supervisor/conf.d/filebeat.conf

COPY monitoring-agent.py /opt/monitoring-agent.py
COPY monitoring-agent.conf /etc/supervisor/conf.d/monitoring-agent.conf

COPY java.conf /etc/supervisor/conf.d/java.conf

ENTRYPOINT [ "/usr/bin/supervisord" ]
```

7.3.3 Monit

An alternative to Supervisor is Monit[32]. It provides more advanced failures handling and a dashboard, but the main difference is that Monit runs the processes as daemon and checks them periodically instead of having child processes.

[32]Monit: `https://mmonit.com/monit/`

Listing 7.8: *Dockerfile* of a Java service extended from the base image defined in listing 7.7, add the executable and set the exposed port.

```
1 FROM java-base-image:latest
2 MAINTAINER tdt
3 COPY target/*.jar /opt/app.jar
4 EXPOSE 8080
```

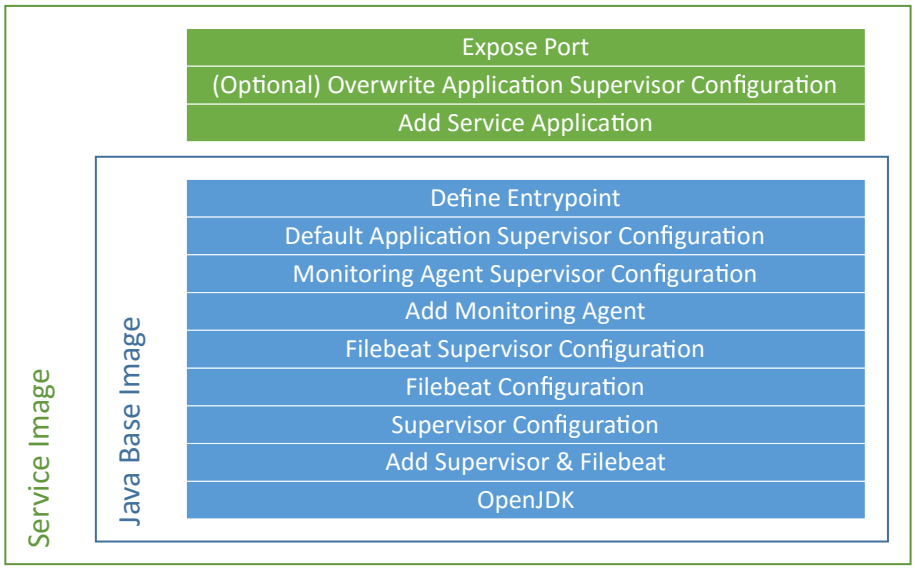


Figure 7.5: Layers of an Image with Supervisor built from listing 7.8

7.4 Service Discovery

In this section, we see how to implement the service discovery pattern (see section 2.6.3).

7.4.1 SkyDNS

SkyDNS is the service discovery used by Kubernetes and is built on-top of etcd[33] which is a distributed key-value store using the raft consensus algorithm. SkyDNS is just a translator between a DNS request and etcd. Since it uses etcd, which uses raft, it is a CP system.
There is no mechanism to register and unregister a service. We can only register a service with a given time-to-live and repeat the registering while the service is available.

7.4.2 Consul

Developed by Hashicorp, Consul is a distributed service discovery based on Serf[34] which can cluster nodes, maintain the cluster and communicate inside the cluster with the gossip protocol (each node spreads new information to his neighbors).
Consul is a distributed system composed of agents run in either one of the two following modes:

- **Server mode**: maintains the cluster state, elects leader and communicates with other datacenters.
- **Client mode**: stateless agent which forwards the data to a server agent

In addition to the service discovery, Consul also provide a key-value store and the possibility to define custom health checks.

[33] etcd: `https://coreos.com/etcd/`
[34] Serf: `https://www.serf.io/`

7.4.3 Eureka

Eureka is the solution developed by Netflix. Given that Netflix is deployed on the Amazon Web Services[35] (AWS), it was built to deploy on this platform (e.g. services are clustered into availability zones, which is AWS specific).
It is an AP system because the developers state that it is more important that every service gets a response and the staleness are handled by a load balancing and circuit breaking approach.

7.5 Load Balancing

A load balancer distributes incoming client requests to a set of server. It is quite similar to a reverse proxy, but the servers are all of the same type and it simply forwards the requests (without caching them for instance).
The load balancer should also health check the servers in order to know to which ones it can load balance.
It can be server-side, such as HAProxy[36]. This means that load balancing is transparent to the client which only see one server to connect to.
Or also in the client-side, such as Ribbon[37]. This time, the client knows the list of available servers and does the load balancing itself.

7.6 Configuration Management

As seen in the twelve-factor app (section 2.1.1), the configuration of a service is all variables which might change over the environment (backing service, credentials, ...) or for other reasons. A configuration is roughly a set of key-value pairs and we are going to see how we can efficiently manage these configurations for all services.

7.6.1 Configuration Externalization

If we want to move the service from the testing environment to the production one without rebuilding the service, we first need to externalize the configuration. For that purpose, two solutions would be to either put all the configuration on a file and load it at start-up or use the environment variables.
Spring proposes both solutions at the same time with the configuration properties (see section 6.2.2.3): the configuration can be defined in the *application.properties* file and all the key-value pairs can also be set in the environment variables (note that the environment variables have a higher priority).

7.6.2 Configuration Centralization

It can be annoying to see and/or change the configuration of all services by going on each container. We would prefer to centralize all configurations in order to easily manage them. Hence, each service fetches its configuration from this configuration server.
The best practice would be to externalize the location of the configuration server (to set this location trough the environment variables) and have all other configurations centralized.

7.6.2.1 Spring Cloud Config

Spring proposes Spring Cloud Config[38] to centralize our configurations on a Git repository. It exposes a REST API to get the configuration of a service with a profile and optionally a label.

[35] Amazon Web Services: `https://aws.amazon.com/`
[36] HAProxy: `http://www.haproxy.org/`
[37] Ribbon: `https://github.com/Netflix/ribbon/wiki`
[38] Spring Cloud Config: `https://cloud.spring.io/spring-cloud-config/spring-cloud-config.html`

Even if it is specialized for Spring configurations, we can also use it to store any other formats (e.g. NGINX) and use the REST API to get the configuration.
The configuration server clones the repository at the first request or when it starts. If the repository is not available, it still give the configurations from its local repository (see figure 7.6).

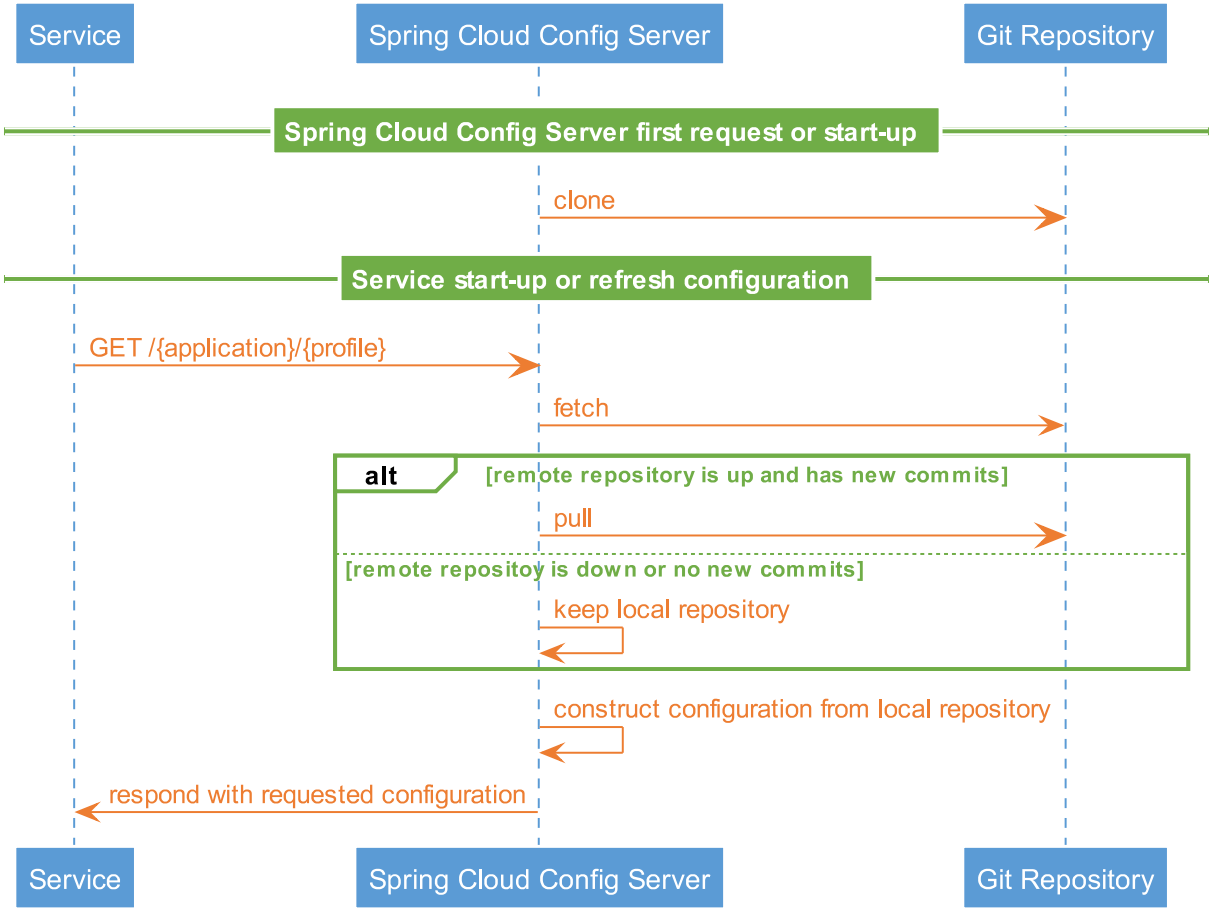


Figure 7.6: Spring Cloud Config Overview

It is possible to refresh a Spring service by doing a POST request on `/refresh`. This works well for casual variables, but not for the locations of the backing services which are connected at start-up.

7.6.2.1.1 Git Repository We can put all configurations in a repository organized with the directory structure or the labels. It is also possible to have many repositories, one for each environment (development, production, ...). The storage of the configurations is highly configurable and it depends on the use case.
In our case, we just have one repository and the root folder is the name of the profiles (i.e. environments). Figure 7.7 shows the file structure of the repository.
The name of each file should be the name of the application (i.e. service ID). The *application.yml* file is the shared configuration. If all your services connect to the same Consul server (for instance), you would prefer to configure this in this file once instead of in each specific configuration files.

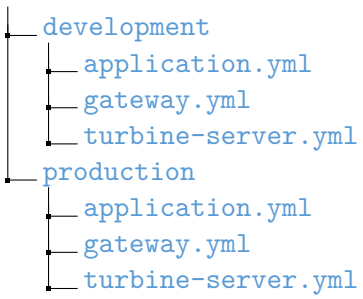


Figure 7.7: The directory tree of a repository for Spring Cloud Config.

Note that it is also possible to use Vault[39] instead of a Git repository.

7.6.2.1.2 Security You probably have some secret resources (e.g. backing service credentials) in your configurations and you prefer to hide them on the (potentially public) repository. With Spring Cloud Config, you can cipher symmetrically (AES-256) or asymmetrically (RSA) some values on the repository and the deciphering is done on-the-fly, either by the configuration server or by the services themselves:

- **Deciphering by the configuration server**: use this solution if the configuration server is in a very secure network because the secrets are sent in plain text to the services. The advantage is there is only one key to manage for all your services.
- **Deciphering by the services**: more secure because the value is never in plain text on the network, but you should pay attention of ciphering the same value with multiple keys, because this weakens the encryption.

The key is given to the service thanks to the property `encrypt.key` (or the environment variable `ENCRYPT_KEY`). Note that if you use Spring Boot Actuator, the `/env` endpoint shows the key and the deciphered values as `******`.

7.6.2.2 Consul

It is possible to use Consul (see section 7.4.2) as a configuration server because it has a key-value stores. It is roughly used in the same way as Spring Cloud Config, because the store can be seen as a file system (the key is the path and the value is the content of the file).

7.6.3 Embedded Configuration

A configuration server is built to serve long-running services which can reload the configuration without recreating the service and this server is also a point of failure because no service are able to start if it is not able to get its configuration.
A more container-oriented solution would be to have disposable and immutable containers where the configuration is already in the container and if it changes, we would have to rebuild and redeploy the container.
A possible implementation would be to use the continuous integration software (see section 5.3) to inject the configuration when it builds the Docker image by using the layering mechanism of Docker. In other words, an environment specific layer will be appended to an image which already contains the service (see figure 7.8). The point is to reuse the same image which contains the executable and not injecting directly the configuration into the executable, because we should avoid rebuilding the executable when we change the environment.

7.7 Summary

7.7.1 Infrastructure Iterations

The implemented infrastructure had several iterations before reaching an acceptable solution. For instance, about the containerization of our services on OpenShift:

1. **Simple Container**: we began to package each service in one container.
2. **Monitoring Agent**: our monitoring system requires to put some agents (i.e. Beats) in the container. We hence use Supervisor to run multiple processes in each container.
3. **Kubernetes Pods**: we find out that it is possible to use the multiple containers per pod concept to have environment agnostic containers. In other words, each service is packaged into a Docker image and we can use the same image for any monitoring system (i.e. the agents are not in the image).

[39] Vault: `https://www.vaultproject.io/`

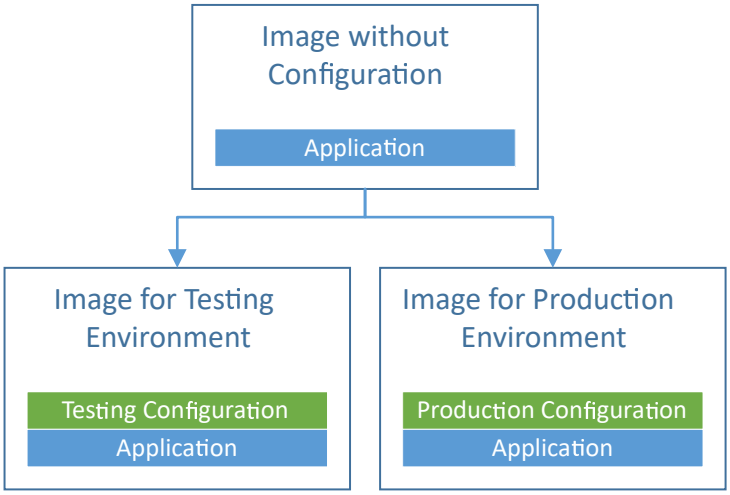


Figure 7.8: In the embedded configuration strategy, we firstly build an image without any configuration. Then, this image can be extended with the configuration of each environment.

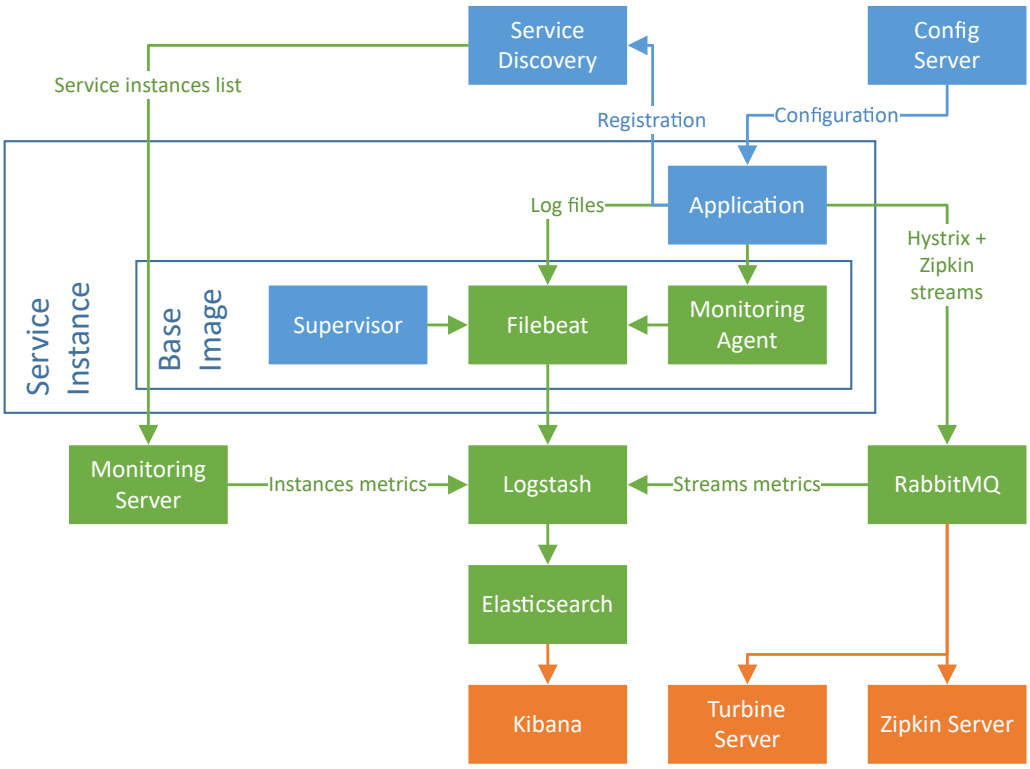
4. **Self-Monitored Container**: Finally, we find that having three (or more) containers per service instance might be a bit excessive. Therefore, we were moving back to containers with agents.

Another example is the use of the health endpoint:

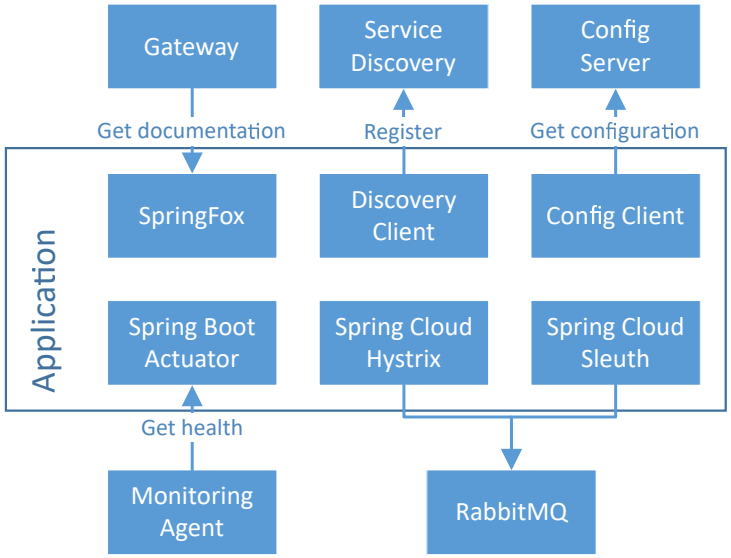
1. **Simple Health Check**: Kubernetes uses the health endpoint to know if the service instance is healthy (i.e. returns HTTP status code 2xx) or not. If it has an issue, Kubernetes may restart it.
2. **Fine-Grained Health Status**: the service instance should show accurately its status (e.g. "I can only handle a specific request"). Hence, a client is able to know in advance which requests it can perform on this service.
3. **Aggregated Health Status**: a health server collects and exposes the health of all service instances and the status aggregated per service.
4. **Monitored Binary Health**: finally, we may only need two possible status (i.e. UP and DOWN) because the platform can only run the instance or stops it. If the instance cannot handle some requests, the client handles it with the circuit breaker. We are still monitoring the instance with a monitoring agent which forwards the health when an error occurs.

7.7.2 Final Implementation

To summarize, we have a base image which contains Supervisor, Filebeat and the monitoring agent. The application and the monitoring agent write to some log files which are shipped by Filebeat to Logstash. When it starts, the service fetches its configuration from the config server and registers to the service discovery. Figure 7.9a illustrates this final implementation.



(a) Overview. The monitoring system is in green and some monitoring dashboards are published by the components in orange. Note that the direction of the arrows shows the data flow.



(b) Zoom on the Application. Note that the direction of the arrows shows who initiate the call. We also add the gateway which gets the API documentation of each service.

Figure 7.9: Final Implementation of the Infrastructure

Chapter 8

Security

8.1 Secure Communication

A microservice application does the inter-process communication on the network instead of using in-memory function calls as in a monolithic application. We should consider having an insecure network, so we should secure all requests done over the network.

8.1.1 Transport Layer Security

A popular solution to perform secure HTTP requests is using it over the transport layer security (TLS), also known as HTTPS. The handshake is done with one or two public key certificates (respectively for one or two ways authentication). Then the data is transmitted ciphered using a symmetric-key algorithm. This hence provides authentication, confidentiality and integrity on the data.
One solution would be that each service has its own certificate signed by a common authority. In this manner, the authentication of each service can be verified.
Managing all these certificates in each service might be difficult and the authentication per user is difficult. Therefore, we are using HTTPS only for confidentiality and integrity. For the user authentication and authorization, we are looking for a better solution.

8.1.2 Network Isolation

If you run your services on a PaaS, it probably provides a network isolation between the projects. Hence, if you only expose your gateway to the external network, it would be quite difficult to access your underlying services without passing through the gateway.
With Docker, you can also create virtual networks to isolate the services properly (i.e. a service only sees the other services it is connected to).

8.2 Authentication and Authorization

A monolithic application commonly manages its users and their authorization. In a distributed system, it is more complicated because the services are independent and we do not want them to log in to each service independently.

8.2.1 Single Sign-On

A user should log in once to the application even if it is distributed. The single sign-on authentication allows users to authenticate against multiple independent services with the same credentials.
Instead of authenticating locally in each service, we use an identity provider (see section 8.2.1.2). A user should first log in on this identity provider (IdP) and get an access token from it. Having a valid token, the user can perform authenticated requests to the services.

An access token contains information about the user and its permissions. For an HTTP request, the token is usually in the `Authentication` header.
There are two main implementations of single sign-on:

- **Security Assertion Markup Language (SAML)**: based on XML, this a mature and enterprise-oriented solution.
- **OpenID Connect**: a more recent solution built on top of OAuth 2.0[1]. It uses JWT (see section 8.2.1.1) as format for the access token and it is much lighter than SAML.

In this chapter, we are going to use the terminology of SAML:

- **Principal**: an entity (e.g. user) which wants to perform an authenticated request.
- **Identity Provider (IdP)**: a principal authenticate against the IdP and this latter gives an access token.
- **Service Provider (SP)**: a principal use the access token to send a request to the SP.

8.2.1.1 JSON Web Token

An access token format standard is the JSON Web Token[2] (JWT). This format provides authenticity and integrity, but not confidentiality on the data. A JWT is composed of three parts encoded in base64:

- **Header**: contains the algorithm (HMACSHA256 or RSASHA256) and the token type.
- **Payload**: the data (e.g. subject, username and roles)
- **Signature**: signature of the concatenation of the header and the payload encoded with the previously defined algorithm.

8.2.1.2 Keycloak

Keycloak[3] is an identity provider which can:

- Isolate projects by realms.
- Store a list of users or use an external one such that LDAP.
- Manage the users (e.g. sign up and set roles).
- Generate certificates to sign tokens.
- Manage the services (Keycloak calls them clients).
- Generate or refresh tokens for authenticated users and clients.
- Show the currently active sessions.

Here is how a user does an authenticated request through a web client and how a service provider (or service) verifies this authentication with the Keycloak server (see also figure 8.1):

1. **Setup**: each service contains *keycloak.json* which contains all the information needed to connect to the server (e.g. realm name, server URL, client ID and credentials). To authenticate the service against the server, one solution is to have a shared secret between the service and the server.
2. **Service Start-up**: when the service starts, it authenticates against the server to get the public key of this latter.
3. **User Authentication**: when the user authenticates through the web client, this latter redirects the user to the login page of the Keycloak server. If the authentication is successful, the server redirects the user back to the web client with an access token and a refresh token.

[1] OAuth 2.0: `https://oauth.net/2/`
[2] JSON Web Token: `https://jwt.io/`
[3] Keycloak: `http://www.keycloak.org/`

4. **Authenticated Request**: the web client ensures that the access token is still valid at each request to the server by refreshing it if needed thanks to the refresh token. The requests are authenticated by putting the access token in the `Authentication` header (the value of the header is of the form "`Bearer <access-token>`").
5. **Authentication Verification**: the service verifies the access token with the public key it gets from the server and if the token is indeed issued by the Keycloak server, it gets the roles from the token and sees if the user is authorized to perform the request.

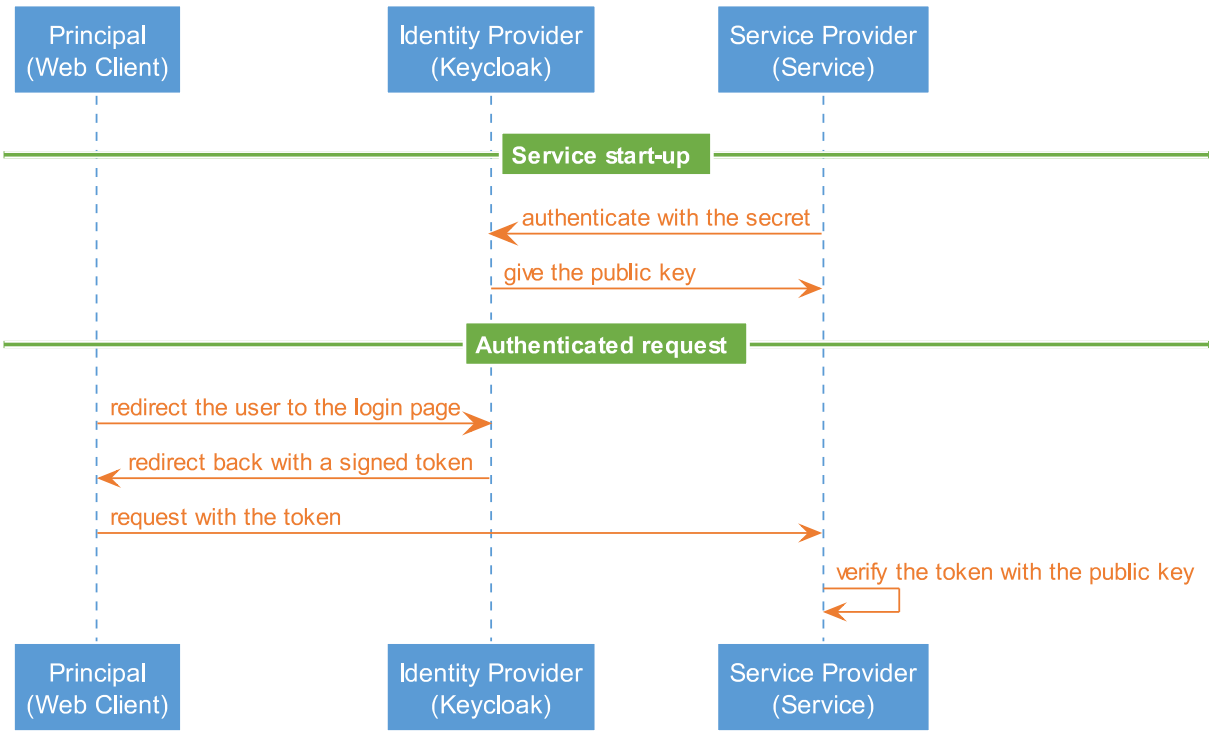


Figure 8.1: Authentication sequence with Keycloak

8.2.1.2.1 Keycloak Library There isn't any working Spring library allowing a service to authenticate as itself (see section 8.2.3). This library was therefore implemented and it calls the REST API of the Keycloak server to get the access and refresh tokens.
It also automatically refreshes the token when it expires. The library creates a `ClientHttpRequestFactory` which can be used to create a `RestTemplate` which transparently authenticates all requests.

8.2.2 Data Origin Authentication

Instead of using a single sign-on authentication, we can use the data origin authentication design pattern. Every request goes through the API gateway which verifies them. The services should only accept requests from the gateway. This makes the gateway a more critical point of failure and overloads it, which is not desired in a distributed system, such as a microservice one.
It would be better if each service verifies the token in every request it receives and checks if the request is authorized thanks to the roles contained in the token. The services should not assume that the incoming requests has already been verified (i.e. they do not rely on another service).
Even if the requests are already authenticated by the services, nothing prevent us to do a pre-verification in the gateway to reject forbidden request earlier. The configuration of the authentication should be externalized by the services in order to allow the gateway to get them (avoid duplicating the authorization definitions).

8.2.3 Service-to-Service Authentication and Authorization

A principal is not always a user and might also be a service when it has backing services.
This principal (i.e. service) can authenticate its requests by a token obtained in two ways: be one of the following types:

- **Creating a new token**: the service creates a new access token with its own permissions.
- **Forwarding the received token**: the service forwards the access token it received from its client.

As seen in section 8.1.1, it is also possible to authenticate the services with each other by using certificates.

Part IV

Illustration & Conclusion

Chapter 9

Implementation Illustration

To illustrate the cloud-native implementation of a microservice architecture, a shopping web application, named Microshop, has been developed. The goal of this demonstrator is to show as many cases as possible and have an example of how to handle them.

9.1 Microshop

Microshop is an application which allows a user to:

- See the available products with its name, price and description. A manager can also add and remove them.
- Add and remove products in a shopping cart.
- Buy the products in the shopping cart.
- Check his/her previous orders.

This is a microservice application, it is hence split in many components (see figure 9.1):

- **product-service**: CRUD service for the products. If the description is not specified in the creation of a product, it gets the description from Wikipedia[1]. Everyone can read the list of products but only managers can modify them.
- **cart-service**: manage the carts. To be scalable and stateless, an integrated Hazelcast[2] is present in each instance and each of them is a member of a cluster. Each cart is owned by a user and they can only be managed by their owner. The service checks out a cart by calling to the order service.
- **order-service**: CRUD service for the orders. To compute the total price of an order, it gets the price of each product from the product service. Only the cart-service can create orders and a user can only read his/her own orders.
- **account-service**: gives information (username, orders) about the account of the user. It forwards the token of the user to get the orders from the order-service.

The database is also separated into two instances (see figure 9.1):

- **product-database**: each product has an identifier, a name, a price and a description.
- **order-database**: each order has identifier, a map of the items (the product ID as key and number of items as value) and the total price.

9.1.1 Infrastructure

Here are the selected infrastructure components for Microshop (see figure 9.2):

- **Consul (service discovery)**: instead of using the Kubernetes service discovery, we prefer to deploy our own one in order to be able to deploy on another platform more easily.

[1] Wikipedia: `https://www.wikipedia.org/`
[2] Hazelcast: `https://hazelcast.org/`

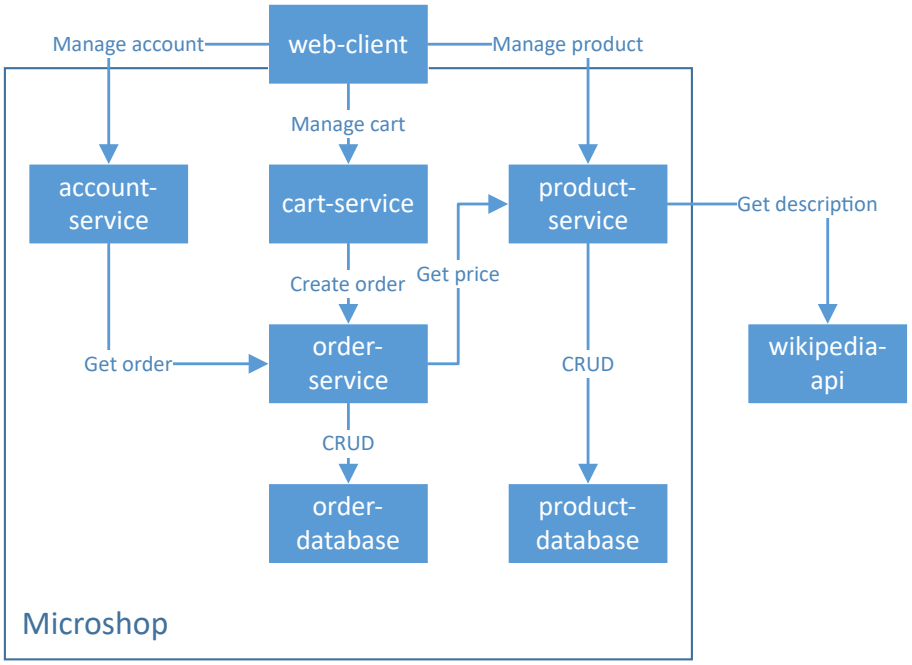


Figure 9.1: Microshop services

- **Elastic stack (monitoring system)**: the Elastic stack is the most mature one.
- **Keycloak (identity provider)**: open source and stable.
- **RabbitMQ (Spring Cloud Stream)**: easy to use and official Docker image available.
- **Spring Cloud Config (configuration management)**: integration with Spring.
- **Supervisor (agents)**: solution promoted by Docker.
- **Zuul (API gateway)**: to be able to customize the gateway as we wish.

9.1.2 Organization

Every component of the system and the libraries (e.g. Kubernetes Discovery Client) have their own repository. The reusable components of the infrastructure (i.e. the configuration, monitoring, turbine servers and the gateway) and the libraries should be able to be forked to ease the deployment of a new microservice application which can fork some repositories to bootstrap faster.

9.2 Libero

The infrastructure deployed for Libero is quite similar to the one for Microshop. The only difference is the use of the Kubernetes service discovery instead of Consul.
Another application (i.e. Microshop) has been developed beside Libero because the development is slower than expected. The on-boarding is theoretically short, but if the developers have to understand how to use the environment (e.g. Docker and OpenShift Origin), it might take some times.
For a microservice application, there should be a few persons which work on the long term (at least at the beginning). Because there are many works to do in order to have a stable foundation. This core team can also help to transfer the knowledge to the newcomers and be available for the support. Even if the development can be distributed, the groundwork should be done uniformly.

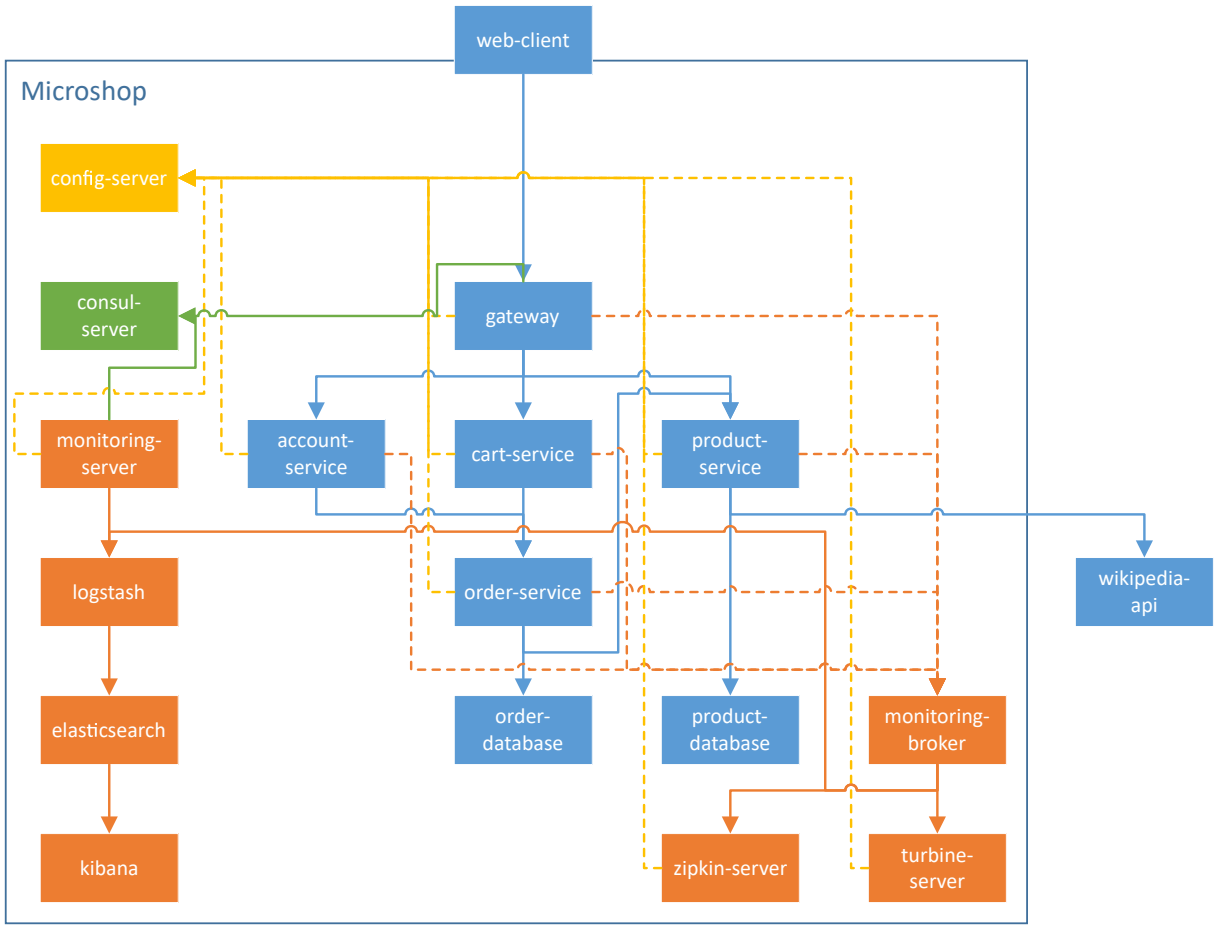


Figure 9.2: Microshop services with its infrastructure. For sake of visibility the registration of all services and servers (including the gateway) to the consul-server, and the aggregation of the logs by the agents to logstash.

Chapter 10

Conclusion

10.1 Lessons Learned

This was a very interesting project and I learnt a lot because I didn't know about most of the technologies tested.
However, there are some points which I would have done differently:

- I spent too much time on the installation and configuration of some products. Finally, the best solution of a problem is often the simplest one. Therefore, if we spent too much time to understanding a solution, it might not be a good one.
- I regret to have discovered the Cloud Native Computing Foundation a bit too late, so I didn't have the time to investigate properly all hosted projects.
- I wonder if selecting the Java language is a good choice for a microservice application. Even if Java is popular (hence widely supported), it is quite heavy and consumes more resources than other languages.
  Moreover, the portability of the Java through its Java virtual machine (JVM) becomes pointless when running on Docker containers because they all run on the same operating system.

10.2 Future Works

The implementation of an architecture is a continuous work because it is always possible to test a new solution or just to improve the existing one. Therefore, it was quite difficult to have a clear end of this project.
Here are some possible continuations of this project:

- **Cloud Native Computing Foundation**: implement and evaluate the solutions proposed by the foundation against the current implementation.
- **Deployment**: explore more in details the possibilities of the continuous integration and deployment (specially to build already configured containers).
- **Resilience**: test the resilience of a microservice application and how it can be improved. Implement also the chaos testing.
- **Frontend**: explore the problematics in a microservice frontend and end-to-end testing.
- **Event-driven architecture**: asynchronous communication between services.

10.3 Conclusion

Even if I was focusing only on some parts of the implementation of a microservice architecture (e.g. backend services and Java Spring framework), the subject is still very wide and the implementation is more difficult than expected because there is a lot of technologies which solves the same problem and implementing some of them is not always straightforward. This type of

architecture is quite new and seems to be very promising, therefore there are many frameworks and technologies which emerges and solves problems in their own ways.
This exploratory project was composed of many iterations and the final infrastructure has been implemented for illustration. We have seen that there are many solutions for the same problem and selecting one of them depends mostly on the context (e.g. on premise or cloud deployment, availability or constancy, centralized or distributed orientation).
While implementing a microservice application, we should think about automating the process of creating, developing, testing and deploying a service. Moreover, guidelines should be defined to have a coherent swarm of services.
Docker makes this kind of architecture possible by providing a lightweight and fast virtualization. It is not only effective for microservices but also has a very wide application fields, such as creating proof of concept or containerized continuous integration to always build and test in a clean environment.
The microservice architecture is very interesting, especially for cloud-native applications which need to be flexible and resilient. It needs a more complicated infrastructure than a casual architecture, but this workload at the beginning might be worth it in the long term when the application grows.

Bibliography

[1] Sam Newman. *Building Microservices: Designing Fine-Grained Systems.* O'Reilly Media, Inc., February 2015.

[2] Martin Fowler. Microservices, a definition of this new architectural term. `http://martinfowler.com/articles/microservices.html`.

[3] Chris Richardson and Floyd Smith. *Microservices From Design to Deployment.* NGINX, Inc., 2016.

[4] Adam Wiggins. The twelve-factor app, January 2012. `https://12factor.net/`.

[5] Arnon Rotem-Gal-Oz. Fallacies of distributed computing explained. `http://www.rgoarchitects.com/Files/fallacies.pdf`.

[6] Balint Pato. Rabbitmq reliability troubles and workarounds, June 2015. `http://www.refactorium.com/distributed_systems/messaging/rabbit/`.

[7] Igor Serebryany and Martin Rhoads. Smartstack: Service discovery in the cloud, October 2013. `http://nerds.airbnb.com/smartstack-service-discovery-cloud/`.

[8] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000.