POLISH-JAPANESE ACADEMY OF INFORMATION TECHNOLOGY

Chair of Software Engineering

Software and Database Engineering

Cezary Grabowski
Student no. 21853

Technical challenges of creating an IT system in microservices architectural style using cloud services

Master Thesis is written under the supervision of:

Mariusz Trzaska Ph. D.

Warsaw, February 2022

Abstract

Microservices architecture solves many problems. It gives flexibility in technology choices, scalability, or organising teams that work on given microservices. Nevertheless, like everything in technology, it comes at a cost. It creates many more technical and domain challenges compared to a monolithic system. As a result, the complexity and effort of development grow significantly. The thesis focuses on technical challenges one needs to face when creating a system in a microservices architecture. Based on the completed prototype, the author analyses the challenges like security, data sharing, distributed transactions, how to develop this kind of system locally, and how to prepare a production deployment on the Google Cloud using Kubernetes and Helm.

Keywords: Microservices architecture, technical challenges, message bus, distributed transactions, Kubernetes, Helm, Docker, Docker Compose, Google Cloud

POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Cezary Grabowski
Nr albumu 21853

Wyzwania technicznie w tworzeniu systemu IT w architekturze mikroserwisów przy użyciu chmury obliczeniowej

Praca magisterka napisana pod kierunkiem:

dr inż. Mariusz Trzaska

Warszawa, luty 2022

Streszczenie

Architektura mikroserwisów rozwiązuje wiele problemów. Jest elastyczna, jeżeli chodzi o wybór technologii, w jakich wykonane są poszczególne mikroserwisy. Ułatwia skalowalność i organizację zespołów wokół poszczególnych aplikacji. Jednakże, te zalety mają swoją cenę. Tworzą one wiele technicznych i domenowych wyzwań podczas tworzenia tej architektury. Jest ona znacznie bardziej skomplikowana niż system monolityczny. Praca skupia się na technicznych wyzwaniach, z jakimi trzeba się zmierzyć podczas tworzenia systemu wybierając tę architekturę. Autor pracy zaproponuje prototyp systemu oraz przeanalizuje takie zagadnienia jak bezpieczeństwo, współdzielenie danych i transakcje rozproszone. Zaprezentuje również sposób na efektywne rozwijanie takiego systemu w środowisku lokalnym oraz jak zarządzać systemem produkcyjnym z użyciem chmury Google Cloud oraz narzędzi Kubernetes i Helm.

Słowa kluczowe: Architektura mikroserwisów, wyzwania techniczne, szyna danych, transakcje rozproszone, Kubernetes, Helm, Docker, Docker Compose, Google Cloud

Table of contents

1. Introduction

This chapter presents the author's motivation for this thesis. It describes the used technologies and structure of this paper

1.1. Motivation

Microservices architecture has become more and more popular among start-ups and big companies [1]. Some of them decide to start with another architecture, i.e., using the monolith-first approach and migrating to microservices later. At the same time, the rest choose to build microservices from the beginning. Both ways raise many issues that the companies must deal with, but it is worth it in many cases. Not always, however.

This architectural style has clear benefits compared to monolithic systems, like technology heterogeneity, scaling, resilience, and ease of deployment, to name a few. For effective systems, those values are worth pursuing. Nevertheless, these characteristics come at a cost. Start with the complexity of development. Suddenly, one does not have to set up only one project but a few during development. It creates problems with efficiency, development pace, and high cognitive load.

Sometimes teams develop microservices too fast without knowing the domain too well. As a result, the system is incorrectly divided into bounded contexts [2]. It can cause a situation where a simple functionality can spread across many services. This results in a developer's frustration, low efficiency, and slow introduction of new changes. This is a consequence of introducing this architecture too fast. Even if the architecture is implemented correctly, it still provides inevitable complications.

The team implements a database per service pattern [3]. This can create a situation where implementing a functionality demands saving information to several databases. How to do it safely? The only way is to use distributed transactions [4]. In the case of a monolithic system, there would often be only one database, and saving information to many tables would only require a local transaction. The whole process would be much simpler.

The author's motivation for this thesis is that being unaware of the technical challenges of microservices architecture can cause many problems. This decision needs to be conscious and well thought out. The author describes these challenges and presents a prototype that exemplifies them. To name a few:

1. how to deal with distributed transactions using SAGA pattern, message bus and four databases in the architecture;
2. how to create a local development environment using docker-compose;
3. how to create a production environment using Kubernetes;
4. the process of deployment to Google Cloud;
5. what are the advantages and disadvantages of the tools that the author used?

1.2. Structure of this paper

Chapter 1 is an introduction to the paper. It describes the motivation, used technologies and tools.

Chapter 2 is about public cloud and containerization. It describes the brief history of evolving methods of production deployment, starting from physical servers to using containers. It contains both on-premises architecture and Cloud computing. It mentions the advantages and disadvantages of both choices.

Chapter 3 is about popular architectural styles. To have a big picture, the reader needs to know both monolithic architecture and microservices architecture. This chapter describes what a monolithic application is and its characteristics. Next, it mentions modular monolith, which is a good starting point for extracting the domain to microservices. In the end, it shortly describes microservices architecture.

Chapter 4 focuses on the characteristics of microservices architecture. It contains both benefits that we gain using it and disadvantages and challenges that we need to face.

Chapter 5 describes created prototype. It is an example that the author used to present the technical challenges of microservices architecture. First, the domain is explained. Then we explain the architecture using the C4 model [5]. It shows the architecture using different levels of abstractions which is helpful to understand it.

Chapter 6 is the author's proposition of handling the technical challenges. They start from the local environment. Developing a system created as a set of microservices is not trivial. Then we mention topics like production deployment, zero-downtime deployment, monitoring, scalability, distributed transactions and self-healing services.

Chapter 7 is a summary. It contains the author's opinions about microservices architecture and how the prototype can be expanded both in terms of the domain and architecture.

2. Technologies and tools

Creating a system in microservices architecture means that we need many technologies and tools. Chapter 2.1. contains information about technologies used inside microservices. Chapter 2.2. includes container management tools, such as Docker, Kubernetes and Helm.

2.1. Technologies and platform

Technologies used to develop backend microservices were PHP8 and Symfony. PHP is a general-purpose scripting language used mainly in web development. Symfony is a PHP framework [6] that simplifies creating websites and web applications. It offers many components that remove tedious and repetitive tasks. The main inspiration for Symfony is the Spring framework for Java.

For the frontend service, we used a JavaScript framework called Vue.js [7]. It is one of the three most popular ones. At least in 2021. Since the JavaScript environment is very dynamic, this can change very fast. This is an open-source solution that simplifies building user interfaces.

PostgreSQL is a database engine that we use for all backend services. It is an open-source relational database that fits well for the prototype. It suits well for the final solution as well.

For the message broker, we chose RabbitMQ. It is an open-source solution that proved to be a reliable choice for prototypes and enterprise systems.

Google Cloud is the cloud we chose for the prototype. Google offers a low price and works well with the Kubernetes cluster [8]. It has got a very intuitive user interface and provides many characteristics and monitoring options.

2.2. Containers management tools

We used docker [9] to build containers and container images. Docker documentation describes what is a container: “To summarize, a container:

- is a runnable instance of an image. Users can create, start, stop, move, or delete a container using the DockerAPI or CLI;
- can be run on local machines, virtual machines or deployed to the cloud;
- is portable (can be run on any OS);
- Containers are isolated from each other and run their own software, binaries, and configurations.” [10].

Container images define needed dependencies and software. This is a foundation that the container is built on.

In order to properly orchestrate those containers, we need appropriate tools. Handling them differs between production and local environment. To handle them locally, we use docker compose [11]. It is a tool to run and define multi-container docker applications. However, all of them are running on the same host, so this tool is not usable on production because only vertical scalability is possible. That is why we used Kubernetes and Helm for handling containers in a production environment.

In the official documentation, we can read: “Kubernetes is a portable, extensible, open-source platform for managing containerised workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.” [12].

Kubernetes platform offers a wide range of functionalities and fulfils much use in a microservices architecture. Just to name a few:

1. self-healing – the platform detects if a container does not work anymore and replaces it with a new one,
2. load-balancing – it can redirect the traffic among the selected applications,
3. autoscaling – when the application is experiencing massive traffic and exceeds the established threshold of resources, it automatically creates a new instance and balances the traffic between the two applications.

To read more about functionalities, head to the Kubernetes documentation [13]. However, how does it work in the first place? How is it built? To explain that, we need to introduce a few concepts used in this paper.

Kubernetes manages a cluster of nodes that are treated as a single unit. This is an abstraction that Kubernetes interacts with. Nodes can be physical servers or virtual machines.

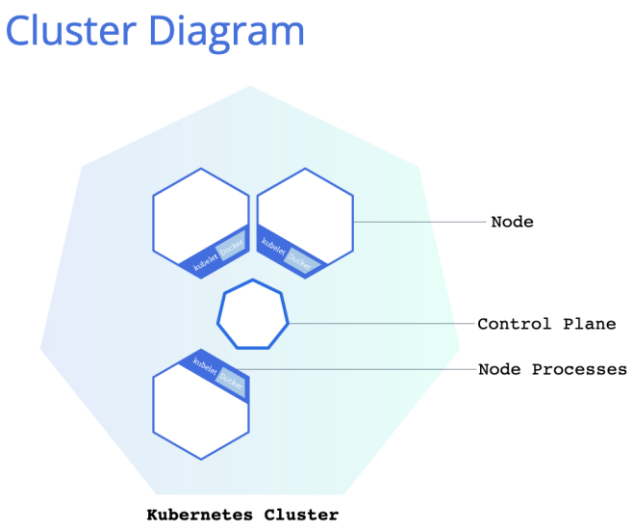

Figure 1 Kubernetes Cluster; source: [46]

In Figure 1, we can see the control plane. It manages the cluster, i.e., maintaining applications' desired state, deploying changes, and scaling. In addition, it exposes Kubernetes API, which is used for communication with nodes.

Let us look at nodes. A single node contains pods and the services necessary to run pods. A service is an object that redirects the network traffic to a pod. It is aware of how many pods of the same instance to load balance users. When the given pod is scaled, then the service is notified.

Kubernetes documentation has got a descriptive definition of pods: "Pods are the smallest deployable units of computing that you can create and manage in Kubernetes.

A Pod (as in a pod of whales or pea pod) is a group of one or more containers, with shared storage and network resources, and a specification for running the containers. (…)" [14].

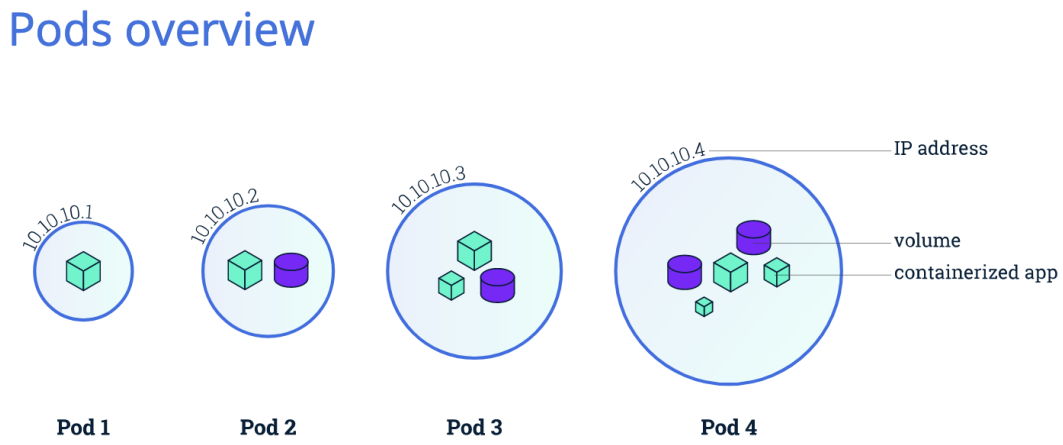

Figure 2 Pods overview; source: [47]

Figure 2 presents a set of pods, containerised applications and volumes inside them. A containerised application can be a python, JavaScript, or PHP application. A volume is something that needs to be saved on the disk. What that means is, in the case of pod failure, the data saved there can be available again when the pod is restored. Every pod has got an IP address attached to it. Interacting with a pod directly; one should use deployments to do that. This is a declarative template of pods' desired state. Listing 1 presents the Kubernetes deployment configuration example.

Listing 1 Kubernetes deployment configuration example; source: [15]

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

It contains container name, image name, how many replicas of this pod should be available, and many more [15]. Changing the configuration is as simple as changing the "YAML" file and running one command to update the config.

Figure 3 presents a big picture of a node and components inside it:

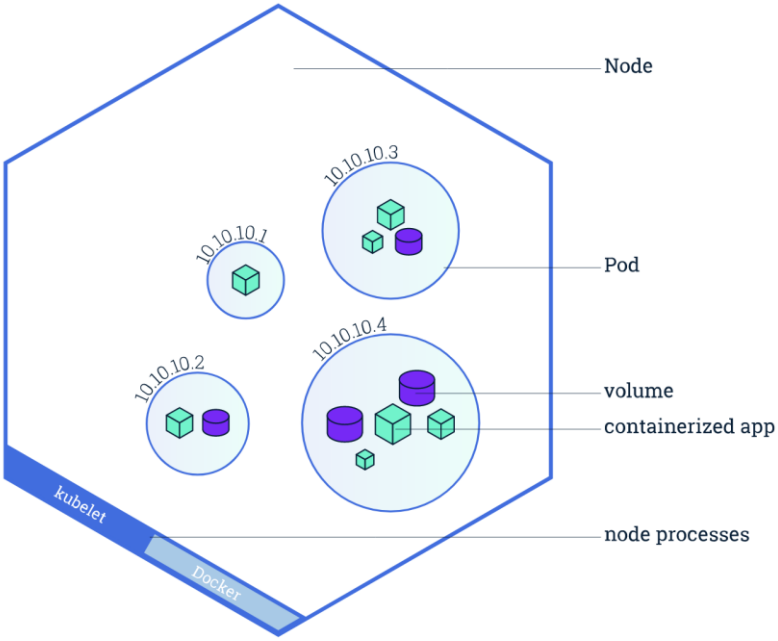

Figure 3 The big picture of Kubernetes components; source: [15]

Manipulating all these elements and gaining all Kubernetes' benefits demands a user specify many details and prepare a configuration. Besides the Kubernetes deployment, there are:

1. stateful sets – configuration needed for stateful applications,
2. config maps – a set of environment variables needed for each application,
3. secrets – a set of private keys and other secret data,
4. services – configuration responsible for correct internet traffic to a given application.

All elements above, and many more, are specified in YAML files. However, instead of creating YAML files separately, there is a tool that helps to manage that. It is called Helm [45].

Helm is a package manager for Kubernetes. It means that it bundles all necessary files that describe a set of Kubernetes resources. This bundle of files is called a Helm Chart. It provides a user with a working, default setup for a specific project, which is flexible enough to be tailored to individual needs. For example, depending on the technology, it could be a Chart for Elasticsearch or Kafka. The repository of ready charts is called ArtifactHUB.

3. Public cloud and containerization

The first approach to application deployments was to use physical servers. This was the easiest thing to do now. However, it has a significant drawback. There was no way to separate resources for each application working on the same server. A server that serves two applications from one server cannot divide resources between them. One of them is experiencing massive traffic, therefore using most resources that the machine provides. The second one uses only a chunk of them. In that case, there is a possibility that the server is noticeably slower or even unavailable for both applications. Of course, one could always serve two applications from two separate physical servers, but this solution would be inefficient. Scalability is also a problem.

In order to solve this issue, people started to use virtual machines. This approach helped with separating applications and specifying resources for each of them. Then applications served from one physical server were not affecting each other. Also, scaling is easier since there is a flexible option to add resources to a virtual machine if necessary. However, virtual machines do not share dependencies, including the operating system. This results in many resources needed to launch a VM, not even serving the application yet. Furthermore, the time necessary to create a new virtual machine is also unacceptable in many use cases. Therefore, companies needed something better, more flexible and faster.

The latest practice is to use containers. These work similarly to virtual machines but solve or at least reduce the issues presented by them. For example, containers have less isolation between them and share the operating system and basic libraries. Figure 4 presents infrastructural differences between all approaches:

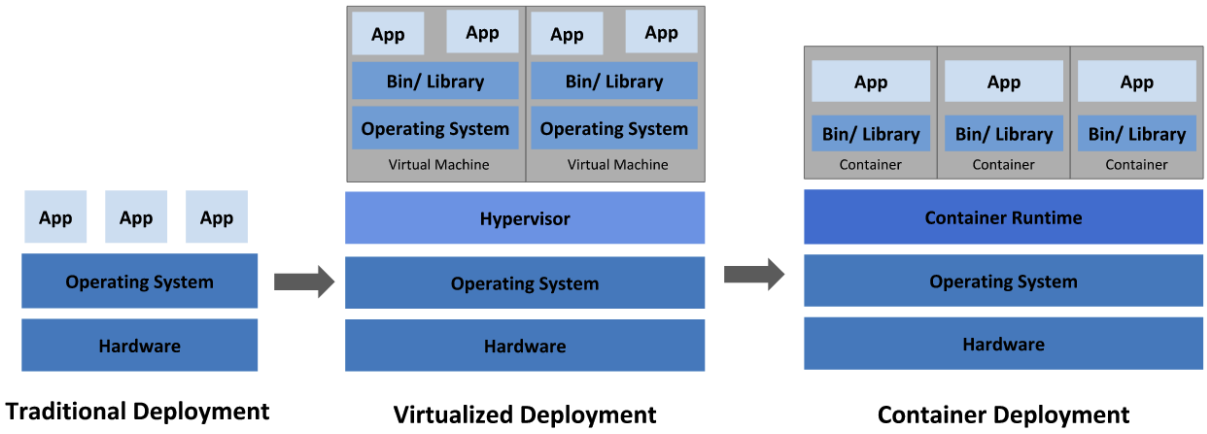

Figure 4 Comparison of traditional, virtualised and container developments.; source: [12]

Thanks to sharing the same operating system, they are lightweight, and it is fast to create new instances. There are a few more reasons why they are so popular:

1. observability – it is easy to observe the infrastructure metrics as well as application's logs,
2. isolation of resources,
3. high efficiency,
4. flexibility and pace in which new instances can be created,
5. ease of integration with continuous development, integration, and deployment.
6. IBM explains this topic in detail on their page [16].

Containerization offers much flexibility in terms of deployments. A system can still be deployed using On-Premises infrastructure or Cloud. There are advantages and disadvantages of both.

3.1. On-Premises

The On-Premises model is the in-house infrastructure. Many companies still use their data centres for many reasons. Having physical access to the infrastructure allows controlling the infrastructural setup completely. In addition, there are many customisation options available. If done correctly, the security can be a great strength of that solution, as everything can be tailored precisely for the business needs. What is more, if the business demands the company to have physical access to the data, then this solution is a must. In cloud services, it is not possible.

However, it comes at a cost. Having the On-Premises infrastructure requires several things. The first one is the extra IT support to take care of the servers. Those need to be maintained and managed correctly. Being able to customise everything means much work at the same time.

The second one is the ability to scale. As the company grows, the number of users grows with it. Data centres cannot handle the traffic, so they need an upgrade. The company suddenly needs to buy more memory, more disks, and better processors. It required installing, handling, and, more importantly, physical space. There is a moment where this is a vast blocker, and the ability to scale is hugely affected. The scaling process is prolonged and complicated compared to cloud computing.

The third one is money. The maintenance costs are high; the company must have significant capital to purchase servers and other needed hardware. It is often impossible for start-ups to have this kind of money, so they choose the second model.

3.2. Cloud computing

Cloud computing is taking over the tech world. Many great and successful companies use the cloud [17]. It offers many advantages, but a few disadvantages come 1with it.

The author believes that Microsoft introduced the best definition of cloud computing – "*Simply put, cloud computing is the delivery of computing services—including servers, storage, databases, networking, software, analytics, and intelligence—over the Internet ("the cloud") to offer faster innovation, flexible resources, and economies of scale. You typically pay only for cloud services you use, helping you lower your operating costs, run your infrastructure more efficiently, and scale as your business needs change. (…)*" [18]. While this definition is correct, it is too broad and should be explained in detail.

The first thing that should be explained is why it offers faster innovation and flexible resources. In many cases, the main goal of a start-up is to solve a problem that people have and are willing to pay for it. Very often, it is a service that is published on the web. It creates many challenges. First, the company must hire great people that can create the product. Their main goal is to focus on and deliver the business value. The rest of the things are obstacles. One of them is the infrastructure. In the case of the on-premises infrastructure, there is a need to hire additional staff to maintain the data centres, and in most cases, new companies do not have enough money to buy the hardware or hire new people. Indeed, they do not want to spend it on that if they do. Therefore, the cloud is a much more popular choice among them nowadays. The company that uses the cloud as their infrastructure is offered much flexibility. It can scale quickly and is not limited by the physical space of its offices. Employees do not have to maintain any hardware for their product to run. The cloud provider's job is to upgrade, maintain the hardware, and keep the software updated. Therefore, in the beginning, start-ups can focus on delivering business value.

The second thing is the costs. Expenses are saved on the hardware because there is no on-premises infrastructure, and the IT staff maintains it. Cloud offers a flexible pay-as-you-go model of payment. It means that the company that uses the cloud pays only for those resources being used. It is a very safe way to find out if the product has a chance to be successful. In case of failure, the company is

not left with the hardware to be sold. Switching off the used resources can be done in a few clicks on the provider's webpage.

Cloud computing offers much freedom and is a good choice in many cases. However, it needs to be a conscious one. Knowing both advantages and disadvantages can help with that. The other important aspect of creating a successful product is the architectural style.

4. Most popular architectural styles

Picking up the right architectural style can be the key to success or make the business go bankrupt. This chapter covers the two most popular architectural styles and describes their characteristics.

4.1. Monolithic application

4.1.1. Overview

A monolithic application is implemented as a single application (deployable unit), most often divided into three layers. This makes it easier to introduce new changes to the codebase. However, since it's a single deployable unit, a faulty change can affect the whole system. Figure 5 presents the most typical setup that consists of two elements:

- application code divided into three layers,
- database.

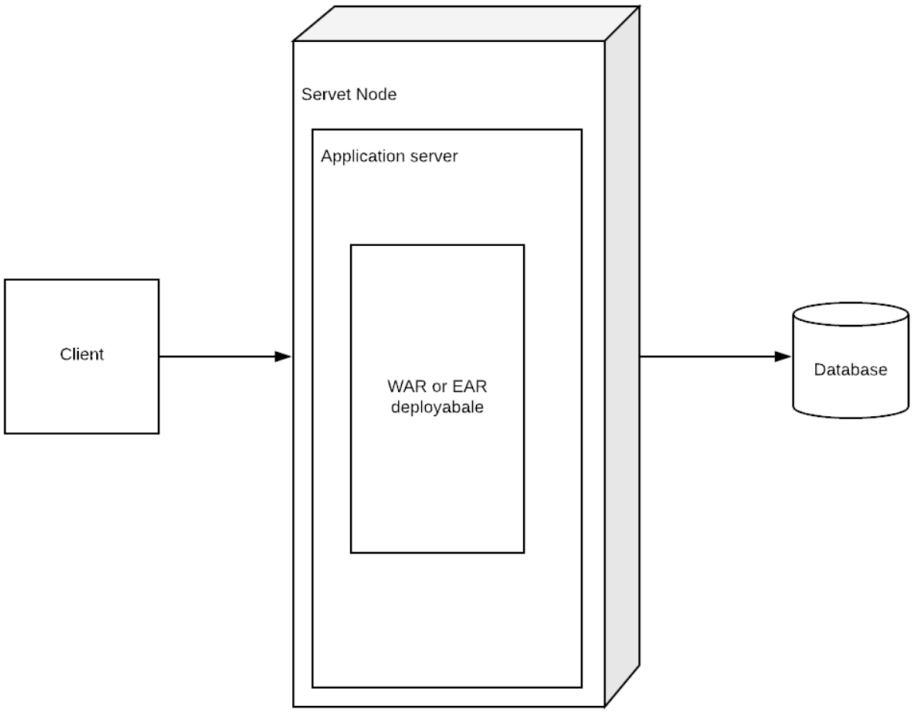

Figure 5 The architecture of a monolithic application; source: [19]

Application code is often divided into three layers. This is a popular approach to monolith application architecture which presents distinctive layers. While it does not change that the application is still a single deployment unit, these layers make introducing changes much more straightforward than in the unstructured codebase. Figure 6 shows a typical three-layered architecture:

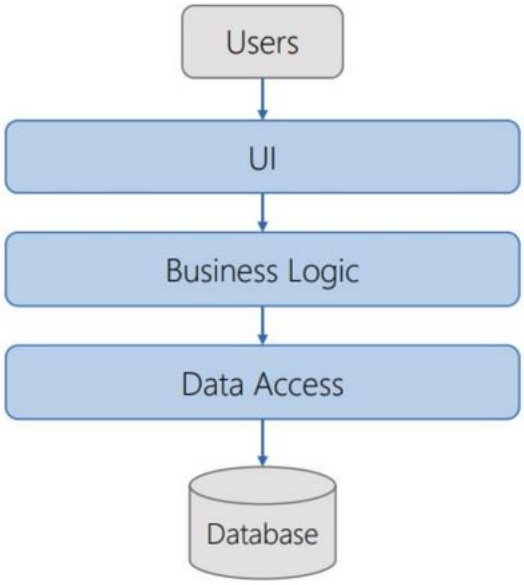

Figure 6 Traditional three-layered architecture; source: [20]

The codebase is separated into three layers:

1. UI,
2. Business Logic,
3. Data Access.

UI, also called a presentation layer, is responsible for presenting the data to the end-user. It is the most crucial part of the system to the end-user. It offers an interface to interact with. Therefore, it must be intuitive and eye appealing. If not, it does not matter if the business logic and data access layers are well organised. The user probably will not use the system in the end.

The business logic layer contains all the business rules of the working system. It provides the essence of the software and validation rules. It is the most valuable layer in terms of the business perspective. It stores all the business knowledge about the domain.

The data access layer provides simplified access to the database engine. It contains repositories that enable querying and updating records.

One of the advantages of this approach is the separation of concerns. Three distinct layers and a clear division of responsibilities makes developing the product less complicated. If the business rules change, the business logic layer needs to be changed. If the design team decides to change something, they must modify the UI layer. The data access layer will be modified if the database engine changes.

4.1.2. Characteristics

One of the benefits of this style is simplicity. Having only one deployable unit means the business logic is in one place. There is no situation where introducing a new feature affects a few microservices and demand a few teams' cooperation. It makes the whole process fast. Having this simple architecture means a fast pace of development and a much lower entry threshold for new employees than in the case of distributed systems. That is why this is a good solution for a new project. Since this is the most common way to develop applications now, there is a big chance that a new developer already knows this architecture.

The communication between each component is much faster than in distributed systems because it takes place within the same system process. Thanks to that, the data is secured because no other

process in the system can access it. In addition to security, the next significant aspect is transactions. Handling the ACID transactions [21] in distributed systems is complicated. It is not the case in monolithic applications. It is much easier to assert the ACID transactions on the operations as everything concerns the same database.

There are also drawbacks to this choice. Firstly, it is fragile. Every process is affected if there are memory leaks somewhere in the system. There is no isolation between this one faulty piece of code and the rest. What is more, if the error is introduced to the codebase and deployed to production, there is a significant risk that the whole application cannot run.

Secondly, it is worth mentioning scalability. In practice, this style allows an application to scale, but it comes at a cost. There is only one choice, either to scale everything or nothing. Even if only one chunk of code is heavily used, the system is upscaled unnecessarily. This is often the only choice, and companies choose this way, but it is a costly operation.

Lastly, maintaining the structure of the codebase is hard. It requires much discipline from developers. Furthermore, keeping the whole codebase in the same place allows breaking any boundaries the team has established quickly. As a result, it will be a spaghetti code instead of a well-structured code. One of the approaches for this architectural style that mitigates this issue is a modular monolith.

4.1.3. Modular Monolith

Modular monolith is still a single deployable unit; however, its structure consists of many autonomous modules. The code that they share is minimal. Therefore, it is easier to maintain them, the business boundaries are much clearer, and the potential extraction process to a microservice will be simpler. However, it still requires discipline from developers. Nevertheless, this structure offers a few benefits:

1. they are easily testable since they are autonomous from each other,
2. maintaining the code is much simpler,
3. migrating to the distributed system, i.e., microservices can be done quicker.

Figure 7 presents the example of modularisation in a monolith.

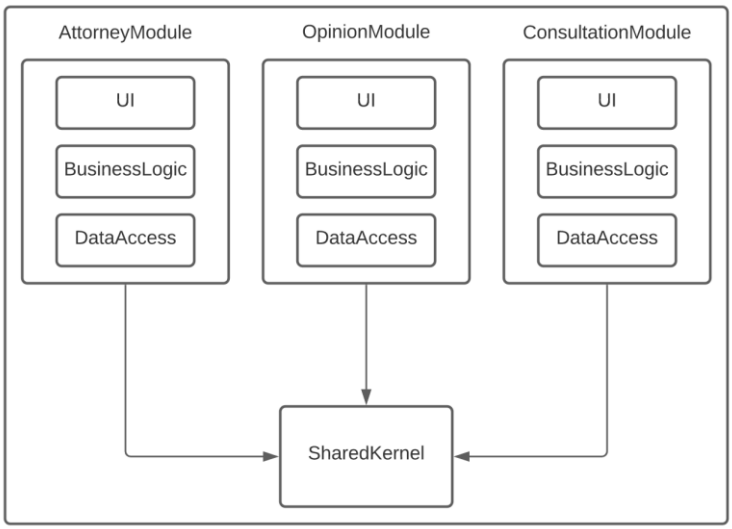

Figure 7 Example of modularisation in monolith; source: own elaboration

Every module has UI, business logic and data access layer. They are autonomous, but they need to share some code in many cases. It is acceptable that there is one module that contains libraries and tools that other modules depend on.

The migration process from this structure is more straightforward than from an unstructured monolith. In literature, this approach to creating the is called monolith first. Martin Fowler writes: "*This pattern has led many of my colleagues to argue that you should not start a new project with microservices, even if you are sure your application will be big enough to make it worthwhile.* " [22] and even though there are good reasons for it, there are people who disagree. Stefan Tilkov says: "*(…) I am firmly convinced that starting with a monolith is usually exactly the wrong thing to do. (…)*" [23]. That depends on various factors [24]. Even if the company starts with a modular monolith, it does not mean that the migration process is straightforward. It is complicated, complex, and broad. It can be easily covered in a separate book. In 2019, O'Reilly published one, written by Sam Newman [25]. He describes the pragmatic process of migration to microservices.

The question is, why is it worth migrating to microservices architecture and what this architecture is?

4.2. Microservices

The author thinks that the best definition of microservices architecture is provided by Sam Newman: "Microservices are independently releasable services that are modelled around a business domain. A service encapsulates functionality and makes it accessible to other services via networks (…)" [26]. It tackles every critical aspect, but there are terms that require additional explanation.

What does it mean that services are modelled around the business domain? The first step when starting a project is to define a domain. There are many tools and concepts to help with that. The approach that places the domain in the heart of software development is Domain-driven design. Martin Fowler explained the concept and the tools in detail and described how to divide the system into microservices [27]. This topic is broad and not trivial. That is why many people wrote books about it, i.e., Eric Evans [28] and Vaughn Vernon [29].

Consider a system with hundreds of microservices presented in Figure 8.

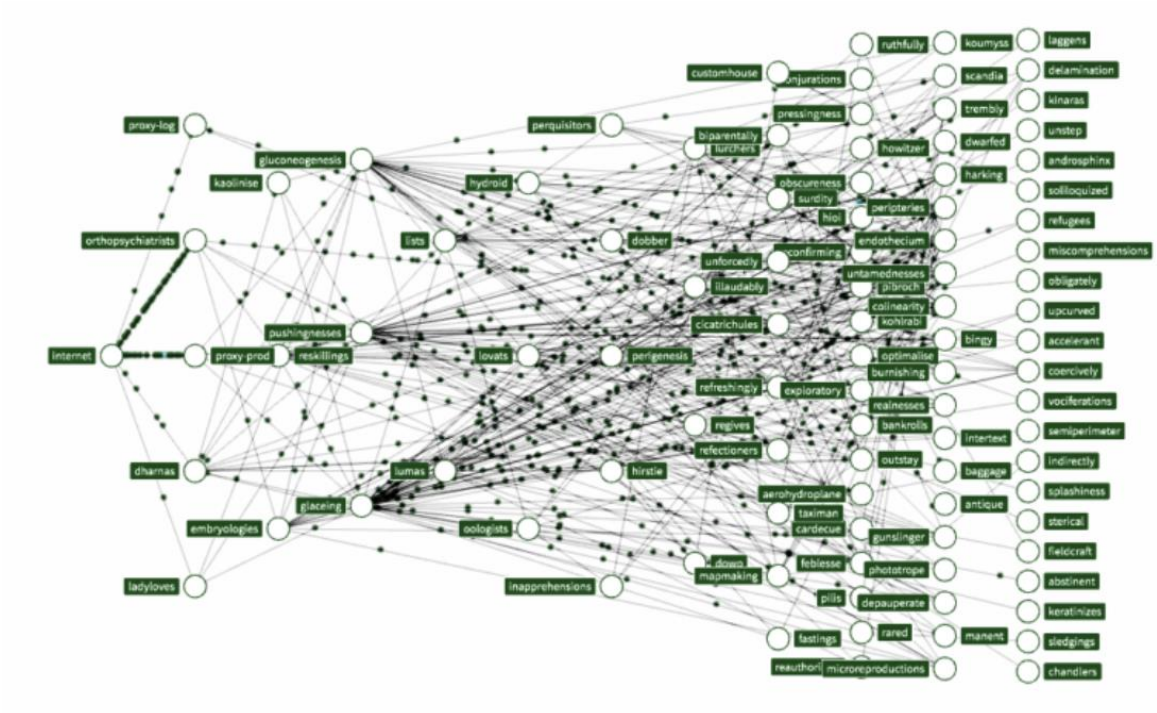

Figure 8 Example of services communication; source: [48]

There is a need to update only one of them. It does not make sense to release every service just to introduce changes in one of them. This situation would be very inefficient and costly. Therefore, services need to be independently releasable. This is a mandatory characteristic.

Encapsulating functionalities is another point worth explaining using the example. One of the services in the system is a pdf generator. Microservice responsible for handling the accounting in the system does not need to have any knowledge about how to create a pdf document. This is delegated to the pdf microservice. This is an example of a technical microservice. In this case, all that is needed is the published interface. There is no need to share the implementation details with others. Thanks to that and well-maintained documentation of the system, development can be very efficient. This gives much flexibility in terms of changing the implementation. Changes should not affect other services, therefore enabling independent releases of functionalities. What is more, it lowers the cognitive load for the employees. The newcomers most likely work on one microservice and do not have to learn the whole business in order to be efficient.

Microservices architecture is said to be one of the most complicated ones. It makes the process of development longer and more complex. Then why do companies decide to use it? What gains and problems does it bring? There are many characteristics to analyse.

5. Microservices characteristics

The author believes that microservices architecture is a double-edged sword. If used wisely, it can bring much value to the organisation but not knowing the challenges of this solution can be fatal. In order to make an informed decision, both advantages and disadvantages should be well understood.

5.1. Benefits

5.1.1. Deployments

When the system is significant, one of the common problems is the deployment process. In monolithic applications, even when one line is introduced to a codebase, the whole application is endangered. Any change requires the whole system to rebuild. It causes many threats, i.e., high-risk factor, a small change that can have a considerable impact, low ability to make frequent releases. The situation is complicated when a hundred developers are working on the same deployable unit. In that case, the pace of growth and developer experience would be questionable.

Choosing microservices architecture can be a remedy to these issues. This is precisely why microservices encapsulate functionalities. When a developer changes implementation details of a microservice, and the contract between services stays unchanged, there is no need to redeploy every part of a system. Service A does not care if the implementation details are changed.

What if this is a bug? What impact does it have? The only impacted microservices are the ones that we introduced a change in and every service that uses this part of a published interface. The rest of the system is working continuously.

Therefore, services need to be loosely coupled. If they were not, we could not gain those benefits. If the two microservices used the same database, then we needed to apply the changes on both units. Therefore, applying the Database per Service pattern is critical and highly connected to the next point.

5.1.2. Scaling

Very often, only one part of the system needs scaling. In a monolithic application, there is no way to do it only for the piece of code that causes performance issues. Therefore, the service needs to be scaled, which is not very efficient and can be costly. Having a set of microservices is the answer to this problem.

To demonstrate the scaling process, we will use the example with two microservices: attorney-service and opinion-service. Both services implement Database per Service pattern, so the architecture looks like in Figure 9.

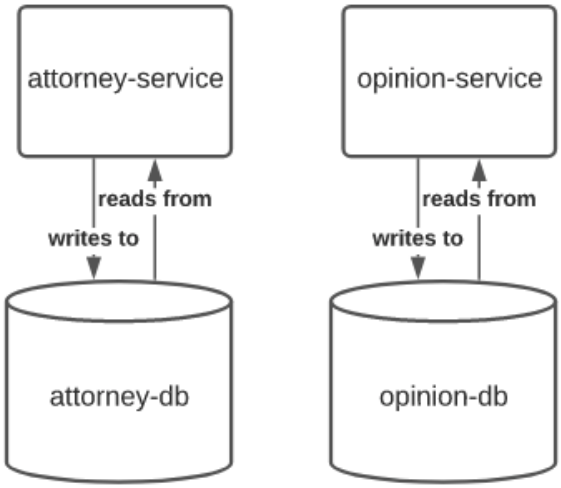

Figure 9 Scalable architecture; source: own elaboration

Given that there is significant traffic connected with opinions flow, there is a way to scale only part of the system that needs more resources. In that case, both the business code and the database should be scaled. This is possible because every service uses its own database. The example of the scaled infrastructure is presented in Figure 10.

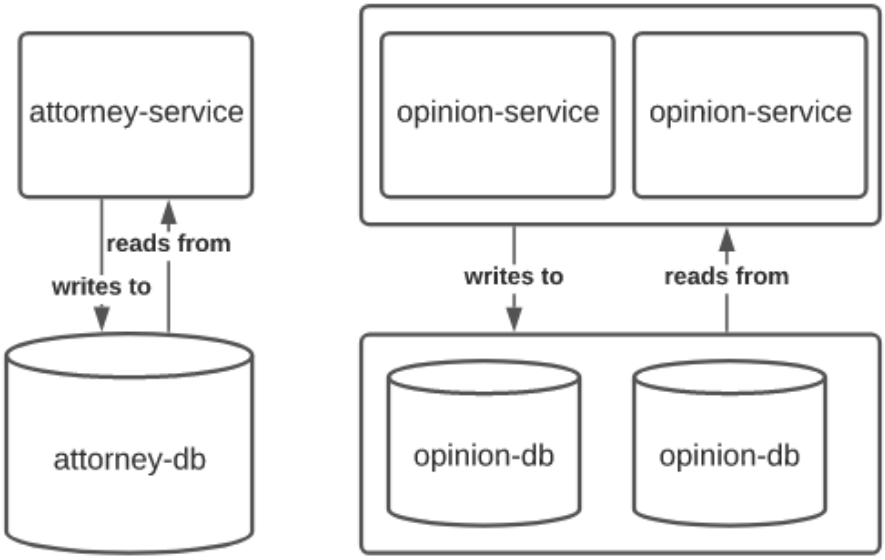

Figure 10 Scaled architecture; source: own elaboration

If microservices are deployed on the cloud, then on-demand provisioning systems can be leveraged to scale the system in minutes. The biggest providers, like AWS, offer autoscaling [30]. When the fixed memory or CPU usage is exceeded, the parts of the application are scaled automatically.

5.1.3. Heterogeneity

Microservices gives much flexibility in terms of choice of technology. In many cases, if there is communication between services, it is made using REST API using HTTP. Thanks to the encapsulation of the functionalities inside a microservice, there is much freedom in terms of choosing a proper

technology for a problem. Please consider scaled architecture from Figure 4. Both attorney-service and opinion-service are written in PHP. There is a need for text recognition of the opinion to automate the process of accepting them. PHP does not offer exciting libraries on that topic. However, there is the library called TensorFlow, which is written in python and is a big help in fulfilling this business requirement. Thanks to technology heterogeneity, it is possible to create a new service using python. Figure 11 presents the scaled architecture with an opinion analyser.

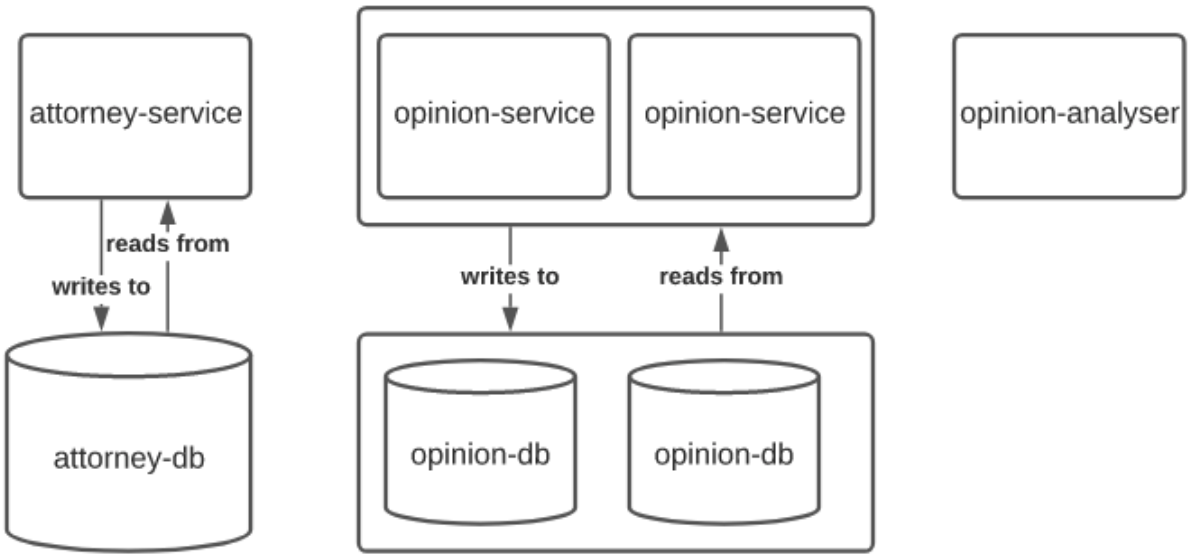

Figure 11 Scaled architecture with opinion analyser; source: own elaboration

Microservices architecture is a great chance to test new technologies. If the new service is written using a language that does not solve the problem, then it can be rewritten using any other technology. In the case of a monolithic application, that experiment is not possible. Thanks to that, the company that adopted this architecture is open to innovation and software developers can try new technologies.

5.1.4. Composability

Nowadays, IT systems very rarely provide capabilities for the web only. People can use a product on smartphones, tablets, or even wearables. That solution needs to be holistic and keep up with the customers. One of the things that can help with that is composability. It allows creating parts that can be used in many places, i.e., login functionality or sending emails. Those can be handled by two separated microservices. If there is a need for changing the email sender provider, only one microservice needs to be modified. Since the exposed API is not changed and only implementation details are modified, no other microservices need to be adapted.

5.2. Disadvantages and challenges

This chapter describes the price that a company must pay when it decides to use microservices architecture.

5.2.1. Developer experience and technology overload

In the perfect world, a software developer must take care of only one microservice. Almost always, it is much smaller than the monolith application. In addition, there is only one project running on the local machine. It is a much better scenario than having the whole monolithic application because of resources usage and an easier-to-setup environment. However, developing only one microservice is

rarely the case. Very often, there is a need to have more than one microservices, and this means that complexity is much higher. Suddenly a developer must set up projects that need to collaborate with each other. It may result in using external tools like docker-compose to orchestrate the local environment. This is another tool that would need introduced to the stack. The author of this paper thinks that the more tools are being used, the more bugs and problems relate to development environments. In that case, the business value would not be delivered efficiently. In the end, this is what really counts.

What is more, the technology heterogeneity can be a pain point and a danger for a project. Having the option to choose any technology for a microservice can be daunting and can end up in a big mess. Let us say that there is a new technology on the horizon, and a new microservice should be created. A software developer then decides to use this new technology to learn something new. It does not solve the business problem more efficiently than the well-tested technology in the project. Image this pattern occurring for a few years. After that time, the technology stack can consist of several languages, frameworks, and different databases. To keep the project maintainable, the company needs to keep many different people who know different technologies. There is no option for keeping only a few developers to take care of the whole system because they simply would not know all technologies used there, therefore being inefficient. This can cost a lot of money and may end up unable to efficiently develop the product.

5.2.2. Cost

As mentioned before, one of the company's costs is hiring people who have a wide range of skills. Alternatively, more people that know our technology stack. One way or another, this is a costly operation.

There are a few others costs that a company probably faces. Assuming that a product is successful, it is used by more people. This means scaling. The result of the scaling is more computing power, higher network traffic, and more significant storage. On-premises infrastructure would be a nightmare to maintain in the case of scaling, both logistically and in terms of money. Cloud computing is much better in logistics; however, it costs too. Until the economy of scale is achieved, scaling can be very costly.

Recruiting new people to the project can be expensive in the short term. A new person needs an introduction to a system and guidance. This causes a developer with experience in the project not to be productive in everyday work. A newcomer is not providing a business value because (s)he does not have any knowledge about the system.

Microservices architecture does not seem like a good choice if the company’s main target is to cut costs everywhere. Using this architecture can pay off, but it is not always the case. It gives the ability to develop multiple functionalities at once. This results in a high velocity of deployments and product development. However, the creators of the solutions have got many traps to avoid in order to get these benefits.

5.2.3. Security

In monolithic applications, security is more straightforward. Most of the communication takes place in the same instance, within the same process. In the case of microservices, the situation is more complex.

Firstly, communication is more complicated. Data transit between microservices, in most cases, happens over the internet. Choosing this way comes with a few threats. Let us look at communication between the two services presented in Figure 12.

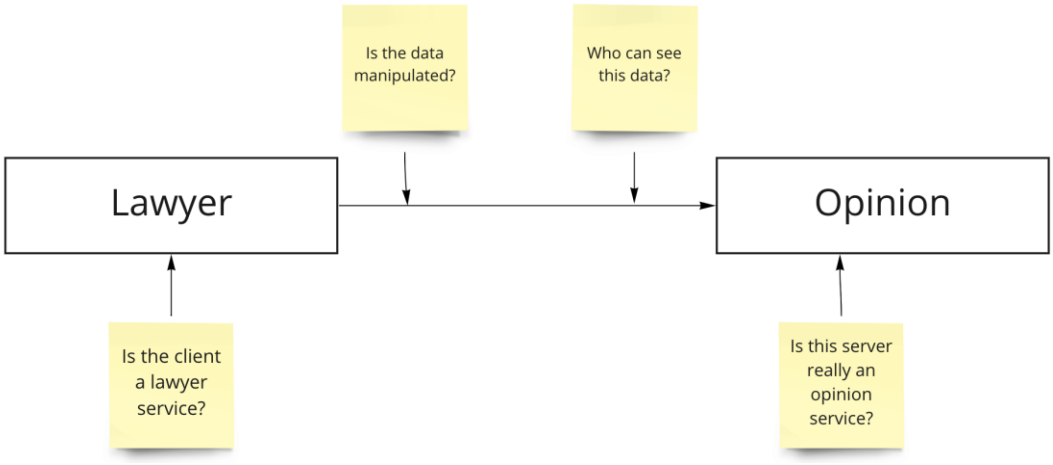

Figure 12 Threats microservices communication; source: own elaboration

What is the client's identity? What is the server's identity? Can data be manipulated? Is it visible for others? Who can see it? When communication takes place over the internet, these are real concerns, and they must be addressed. There are a lot of good practices to follow, and when it comes to security, one should stick to the popular solutions and not reinvent the wheel.

Secondly, credentials and secrets. Many microservices need sensitive data to operate. It should be available only in this service, and others should not have access to it. Kubernetes has a built-in mechanism for that. Another example can be a Vault. The most popular cloud providers offer built-in solutions as well. In the case of AWS, it is AWS secrets manager.

Thirdly, patching. Whenever a bug or a security issue is found, the tool's creators release a patch with notes. Notes contain things that were improved, including security. If the organisation is using that tool, it is critical to update the software as soon as possible because intruders can use that knowledge and create an exploit. Therefore, it is in the company's interest to update the software. In the case of one monolithic application, this is an easy task. One instance means one update. In the case of microservices architecture, the problem is more complex.

Moreover, the problem does not have to be in the used software. It can be an operating system too. One of the most popular solutions to handle the microservices architecture is the Kubernetes. Let us look at the example of the infrastructure underneath a typical Kubernetes cluster in Figure 13.

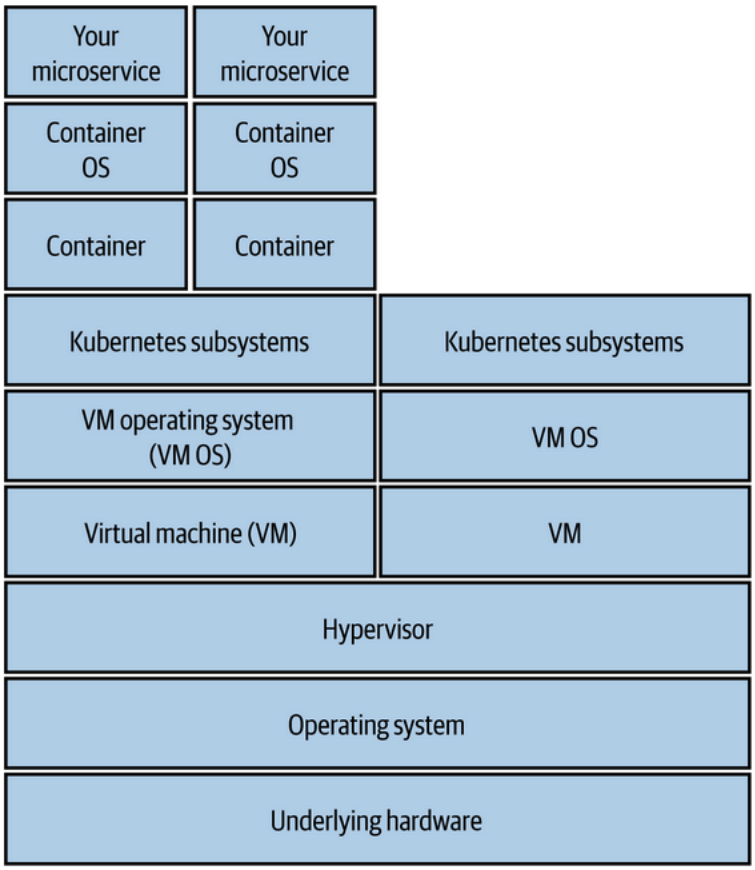

Figure 13 Layers that require maintenance and patching; source: [26]

Every layer needs maintenance and patching. It is a huge workload and stress on the team, which could create new things instead and move the product forward. Thankfully, using a cloud provider makes this situation more manageable. Most of them handle many of these layers by themselves. Figure 14 presents those layers.

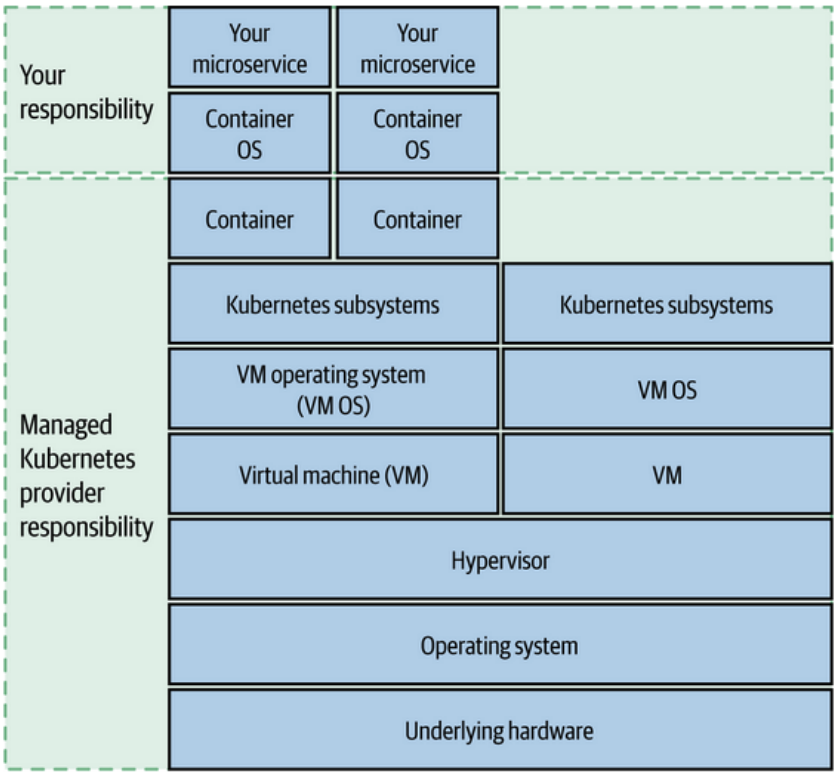

Figure 14 Layers maintained by the cloud provider; source: [26]

The only things that creators must keep an eye on are the libraries used to develop a product and a container operating system, which can be easily replaced. That simplifies the situation a lot.

This paper does not mention many aspects, i.e., authentication, authorisation, backups, and rebuilding. All security aspects are so broad that they could be a topic for the other paper. The author recommends the book Building Microservices 2nd edition, written by Sam Newman, to analyse the security aspects in depth.

5.2.4. Monitoring & Debugging

There is only one application to monitor, one CPU, memory, and database in the monolithic architecture. If 100% of the CPU is used, it is a problem. The same goes for memory and database stress. There is only one place to debug in case of service unavailability or slow performance.

In the microservices architecture, the situation is not as clear. It is much more complicated in the case of hundreds of microservices. For example, in the case of a production environment and opinion service that is not responsive. There can be many reasons for that. The opinion application may have crashed, the host may be down, or there may be network issues.

What is more interesting is that the cause of the problem can lay in the attorney service that the opinion service depends on. To debug the problem, there must be a solid monitoring solution implemented. The mandatory things to monitor are:

- application logs to catch any application-related problems;
- response time of the application;
- performance of the host;
- infrastructure information - memory, CPU on machines to detect potential problems early on;

- alerts – the most important thing. If the product consists of a hundred microservices, the one place aggregating all this information is a must. It must implement alerts to notify the team when the end-user witnesses a bug or outage. In the perfect world, it would be even earlier.

Monitoring microservices architecture is more complicated and more complex than in the case of the monolithic application.

5.2.5. Data consistency

Keeping data consistency in the system is essential. In monolithic style, it is relatively easy to do. Having one SQL database and one application means that local ACID transactions are used. Nowadays, every noticeably SQL database engine provides ACID transactions. This acronym consists of:

- atomicity – every transaction either succeeds all its operations entirely or fails completely,
- consistency – the transaction changes the state of the database. All data is valid after the transaction is done,
- isolation – two transactions that take place in the exact moment does not interfere with each other – one transaction does not change the result of the other,
- durability – once the transaction is executed, the data changes are saved in the database.

In the case of one database, all aspects are met. This is more complicated in distributed systems. For example, say that there is an application attorney-service and uses the cluster of two instances of attorney-DB like in Figure 15.

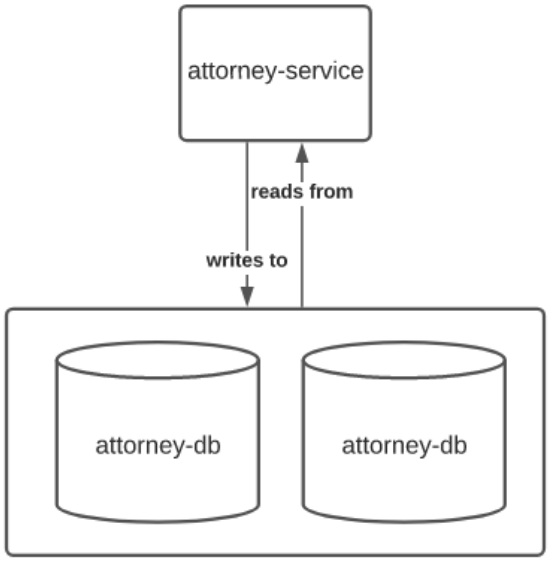

Figure 15 Scaled attorney database cluster; source: own elaboration

When attorney-service interacts with the database [Figure 15], it interacts with either attorney-db1 or attorney-db2. What is more, a query to attorney-db1 may give a different result than querying attorney-db2 because the data differs between those two. Eventually, the data is consistent because it is propagated to every node in a cluster. This is called eventual consistency. We agree that queries may return different results simultaneously, but eventually, they will be consistent in every node. In many cases, this approach is good enough for the business domain because it provides a fast response time, and the data does not have to be consistent in an instance. That is why the BASE model [31] was created as an alternative for ACID:

1. basic availability – database is highly available,

2. soft state – Replicas do not have to be consistent all the time,
3. eventual consistency – Replicas are consistent at some point at a time (not specified).

The above describes data consistency between the nodes in a cluster. However, that is not the only problem there is. To have a scalable architecture, we need to apply a database per service pattern, which assures that no two services access the same database. This creates an interesting problem of consistent writes to the database. Sometimes business functionalities demand saving information to a few databases in a single flow. What to do then?

To ensure data consistency, one needs to implement transactions that span services – distributed transactions. The best current practice is to implement the SAGA pattern. In the example above, we have choreography-based SAGA. This pattern is an approach on how to deal with many local transactions. After each local transaction is done, an event is published, which can indicate two things:

- the transaction ended correctly – this is the signal that the next transaction can be started,
- transaction failed – which means that the previous local transaction must be reversed.

Figure 16 presents the example.

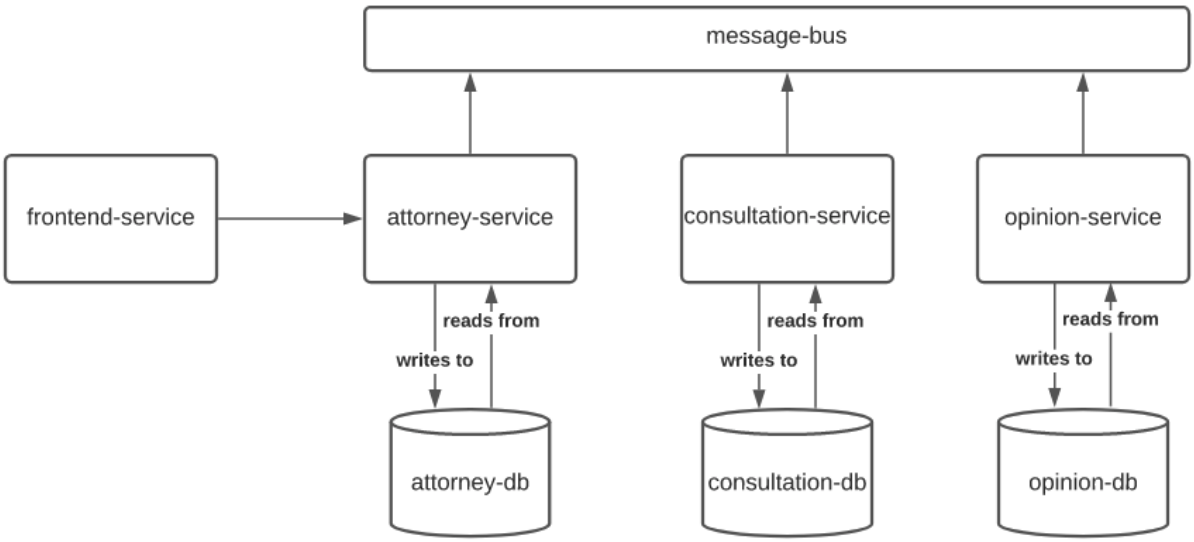

Figure 16 “Schedule consultation” operation in a microservices architecture.; source: own elaboration

The system has got four microservices: frontend-service, consultation-service, place-service and attorney-service. Since the database-per-service pattern was implemented, all services that are not stateless need separate databases. This is a reason why the SAGA pattern was applied. Since we need to publish messages – we need a message bus. Thanks to that, we can have asynchronous communication. This allows us to have the system available as fast as possible, and we can be sure that it is consistent eventually. However, when the user schedules consultation, we need to ensure a few things:

- the user can schedule a consultation – consultation-service,
- the place needs to be available in the chosen date – place-service,
- the attorney needs to be available on the chosen date – attorney-service.

Table 1 Microservices and events they listen to; source: own elaboration

Service name	Listens to
consultation-service	`PlaceUnavailableForConsultationEvent`, `AttorneyUnavailablePlaceRemovedEvent`, `AttorneyAvailableForConsultationEvent`
place-service	`ConsultationScheduledEvent`, `AttorneyUnavailableForConsultationEvent`
attorney-service	`PlaceAvailableForConsultationEvent`

Consider a few scenarios which present why we need those events to ensure data consistency. Those scenarios are presented in Table 2, Table 3 and Table 4.

Table 2 Scheduling consultation succeeded; source: own elaboration

Service	Trigger	Operation	Event published
consultation-service	User interaction	Saves a consultation in pending state	`ConsultationScheduledEvent`
place-service	`ConsultationScheduledEvent`	Reserves a place for a given date	`PlaceAvailableForConsultationEvent`
attorney-service	`PlaceAvailableForConsultationEvent`	Reserves attorney for a given date	`AttorneyAvailableForConsultationEvent`
consultation-service	`AttorneyAvailableForConsultationEvent`	Change a consultation state to accepted.	-

Table 3 Scheduling consultation failed – place unavailable; source: own elaboration

Service	Trigger	Operation	Event published
consultation-service	User interaction	Saves a consultation in pending state	`ConsultationScheduledEvent`
place-service	`ConsultationScheduledEvent`	Reserves a place for a given date but is already taken on that date.	`PlaceUnavailableForConsultationEvent`
consultation-service	`PlaceUnavailableForConsultationEvent`	Changes a consultation state to “place unavailable”.	-

Table 4 Scheduling consultation failed – attorney unavailable; source: own elaboration

Service	Trigger	Operation	Event published
consultation-service	User interaction	Saves a consultation in pending state	`ConsultationScheduledEvent`
place-service	`ConsultationScheduledEvent`	Reserves a place for a given date	`PlaceAvailableForConsultationEvent`
attorney-service	`PlaceAvailableForConsultationEvent`	Reserves attorney for the given date but is not available on that date	`AttorneyUnvailableForConsultationEvent`
place-service	`AttorneyUnavailableForConsultationEvent`	Remove a place’s reservation for a given date since the consultation cannot occur because the attorney is unavailable.	`AttorneyUnavailablePlaceRemovedEvent`
Consultation-service	`AttorneyUnavailablePlaceRemovedEvent`	Changes a consultation state to “attorney unavailable”.	-

The three examples above prove that distributed transactions are complex and how many things can go wrong in a simple operation. They need much work to implement and a substantial cognitive load for new people to understand what is going on. The example presents a choreography-based saga. There is also an orchestration-based saga described by Chris Richardson [32].

5.2.6. Communication between services

Building responsive, resilient and scalable systems in a microservices architecture are not trivial. In order to do that, we need asynchronous communication between microservices. To achieve that, we can use the concept of Event-driven architecture.

It is an architecture where services communicate primarily with events, less frequently with APIs. We distinguish two types of messaging models: point-to-point and publish/subscribe.

In the point-to-point messaging model, a service sends a message to the message queue. When this message is consumed by one service, it is gone. No other service can access it. Figure 17 presents this behaviour:

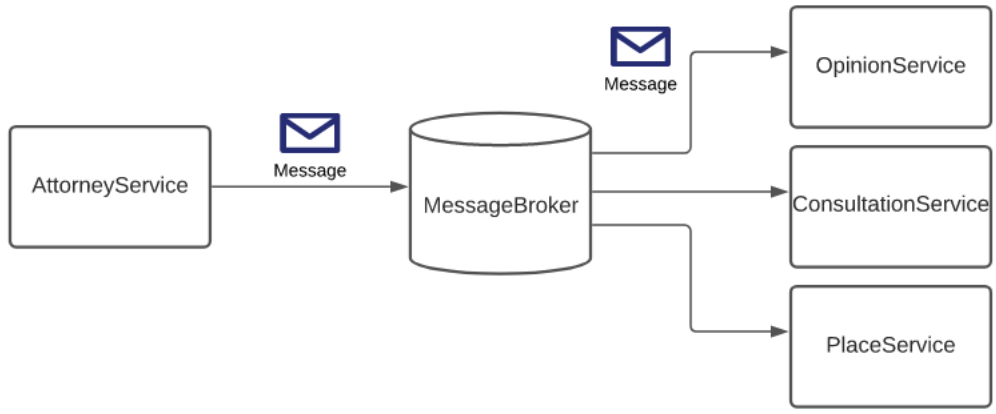

Figure 17 Point-to-point Messaging Model; source: own elaboration

When we have two services that are interested in the sent message, only the faster one will consume it. However, there is the second pattern we can use: Publish/Subscribe Messaging model.

The difference between those two is how many receivers/subscribers will get the message. Very often, more than one services are interested in one message or event. Then, the message should be sent to all subscribed services. Figure 18 is an example of such a solution:

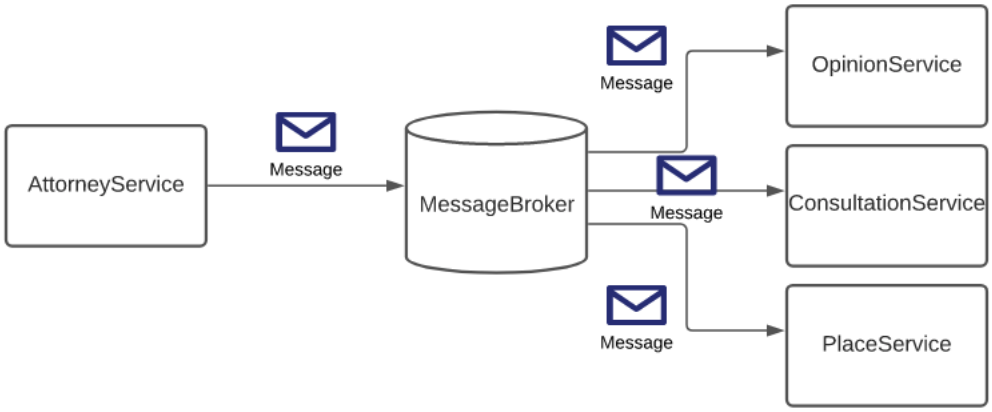

Figure 18 Publish/Subscribe Messaging Model; source: own elaboration

Thanks to that, our system is resilient. When the message broker cannot send the message due to network issues, it collects them. When the network is available again, then all messages are sent to subscribers. An end-user may not even notice that. The business rules will be met as well because the data will be consistent at some point in time.

It is also easily scalable. When the message queue gets bigger, then we can scale services separately in order to consume all messages that are waiting. That way, we are flexible and can adjust the infrastructure depending on the user traffic.

What is more, we have logs of every single operation in our system. If the user comes to us and reports a bug, we can track exactly what happened and why did the bug occur. If we are interested in the history of change in an object during scheduling consultation, for example, then we can do it. This is called Event Sourcing [33]. This approach tells us not only what state our data are in. It tells us how we ended up there.

5.3. Summary

This chapter presented the characteristics of a microservices architecture. Considering both advantages and disadvantages can help with deciding whether to use this architecture or not.

6. Created prototype

In this chapter, the author presents the prototype. It is an attorney's marketplace, where a user can schedule meetings and send opinions about attorneys. This example shows that even a simple domain showcases the complexity of microservices architecture. Technical challenges are the same in a system of a larger scale.

6.1. Domain

The system is an attorney's marketplace. Created prototype enables a user to interact with attorneys, consultations and opinions. Below are a few use cases:

1. user lists attorneys,
2. user schedules a consultation with an attorney,
3. user gives an opinion to a specific attorney,
4. user lists scheduled consultations,
5. user reschedules consultation.

In addition, the system offers an administrator's panel with CRUD operations for all available entities in the system. However, even a relatively simple prototype caused a few nontrivial problems regarding the architecture and distributed transactions.

Microservices architecture offers much complexity in terms of the future growth of a system. There are not many technological boundaries. For example, we can implement graph databases or a microservice responsible for handling machine learning models. Since our system stores opinions, then before publishing, they must be accepted by the moderator. After some time, we can automate this process by rejecting opinions that are not fulfilling predefined criteria.

6.2. Architecture

To present the software architecture, the author will use the C4 Model [34]. The model consists of four types of software architecture diagrams. Each of them presents a different level of abstraction. The diagrams are as follows:

1) level 1: System context diagram,
2) level 2: Container diagram,
3) level 3: Component diagram,
4) level 4: Code (optional).

Level 4 of the diagram is optional. It provides low-level details of the classes and methods in them. Since the authors of this model recommend this diagram for the most complex and essential components [35], we will omit this one mainly because it is a prototype.

6.2.1. Level 1: System context diagram

This diagram aims to present the "big picture" of the solution. It shows how our system interacts with others, whether an actor or an external software system. Figure 19 contains such a diagram for our prototype.

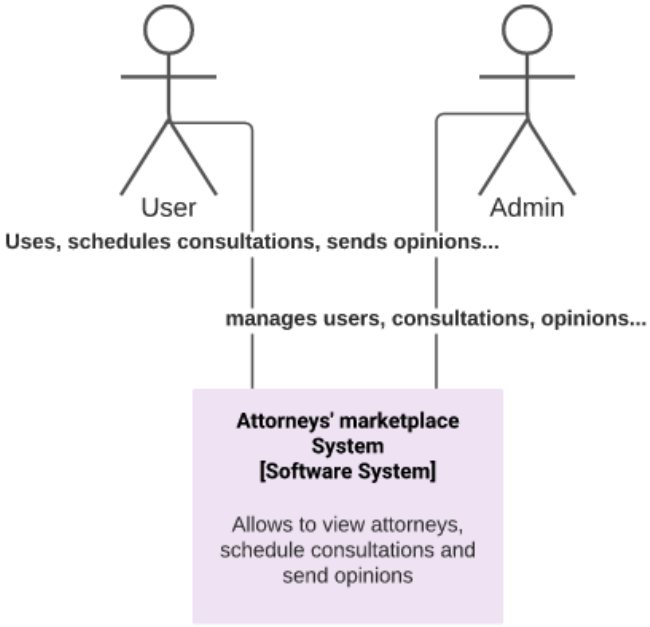

Figure 19 System context diagram; source: own elaboration

In this case, we have three elements of a diagram. Two actors and one software system. The software system is an Attorneys' marketplace. It allows to view attorneys, schedule consultations and send opinions. Two types of users interact with it:

- User – uses, schedules consultations, sends opinions,
- Admin – manages users, consultations, opinions, places.

This diagram does not go into detail. It offers the highest level of abstraction without technical details. Thanks to that, people without technical knowledge can understand it.

6.2.2. Level 2: Container diagram

The next step is to present containers of a system. A container is a separably deployable/runnable unit that executes code or stores data [36]. It can be a single-page application, datastore, message broker or microservice. This diagram provides one level deeper understanding of architecture. In addition, it gives information about used technologies and communication protocols. It is intended for technical people that work on a solution since it contains technical detail.

Figure 20 presents the Container diagram for the prototype:

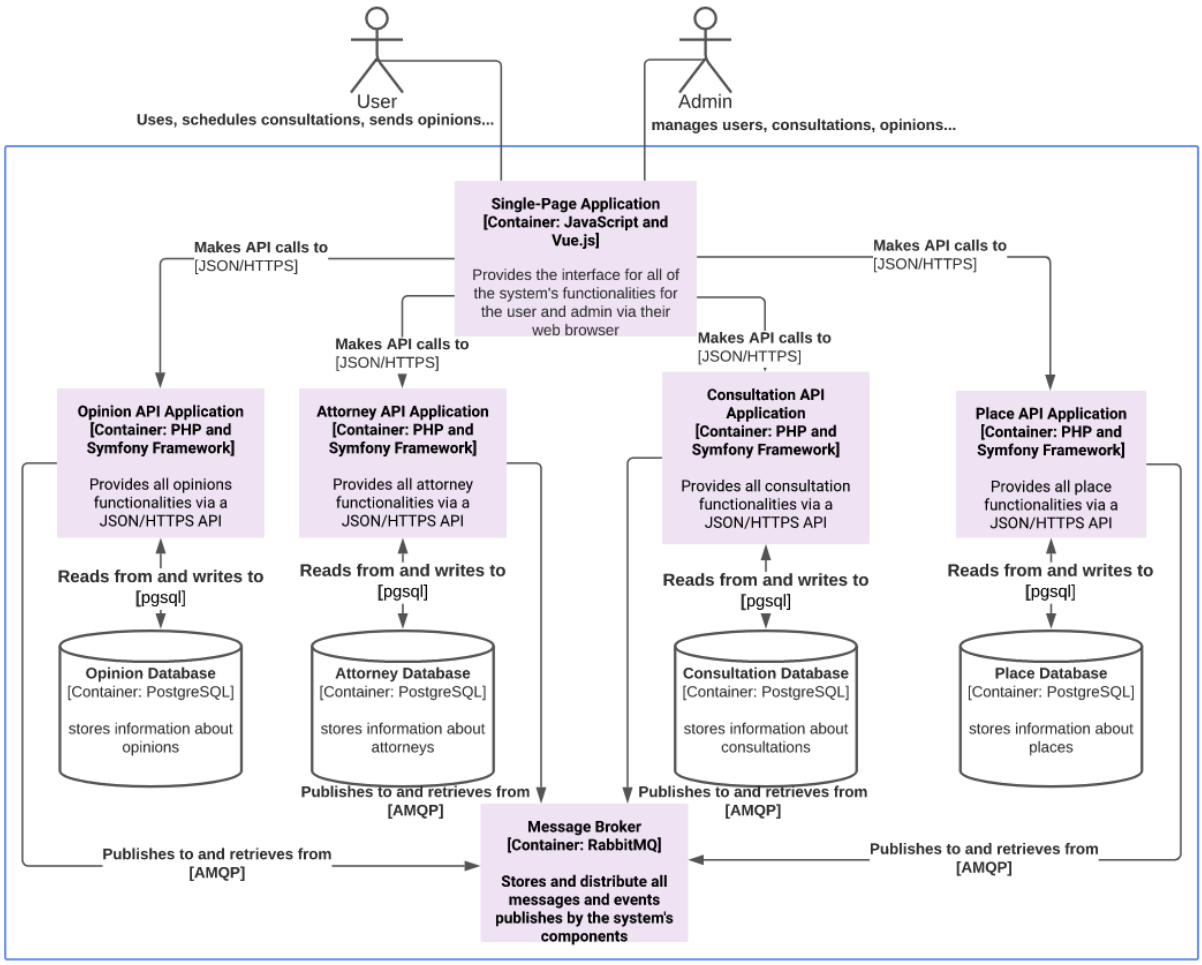

Figure 20 Container Diagram; source: own elaboration

The Single-Page application is a container created with JavaScript and Vue.js. It provides the interface for all functionalities in the system. It is available for users and admins. They access it via a web browser. Depending on the user role, the application displays a different UI.

This application makes API calls with backend microservices via JSON/HTTPS. We have four of them in our system:

- opinion API Application – provides opinions functionalities via JSON/HTTPS API,
- attorney API Application – provides attorneys functionalities via JSON/HTTPS API,
- consultation API Application – provides consultations functionalities via JSON/HTTPS API,
- place API Application – provides places functionalities via JSON/HTTPS API.

Every backend microservice is created using PHP and Symfony Framework. Those technologies offer components that simplify creating secure REST APIs.

Each of them uses its own database. Databases use the PostgreSQL engine. It is an open-source relational database that serves the business purpose well.

The last container is a message broker. In that case, we used RabbitMQ, one of the most popular open-source message brokers. It proved to be robust, secure and works with PHP well.

6.2.3. Level 3: Component diagram

The component diagram zooms into the single container and displays its components. Diagram presents the details of a container and displays communication between those and other components. Figure 21 shows the Opinion API Application details:

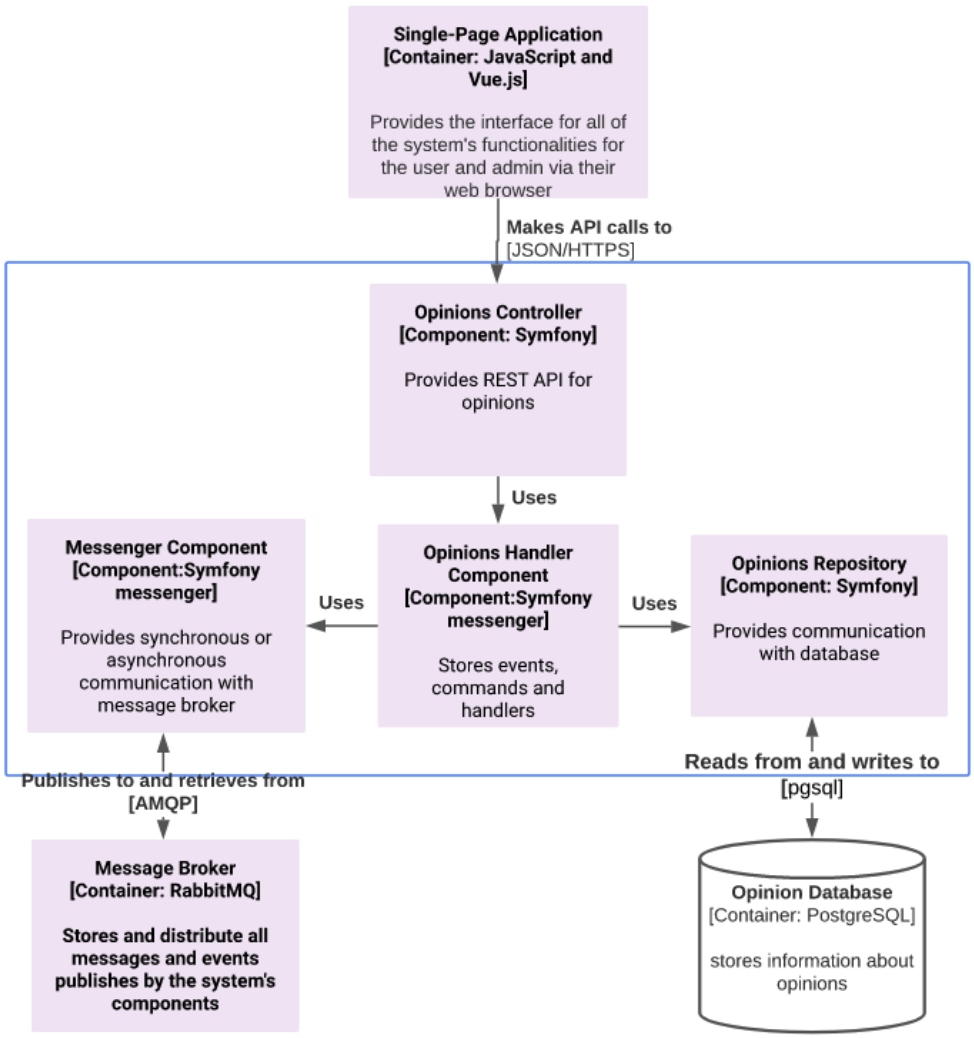

Figure 21 Opinions API Application component diagram; source: own elaboration

Opinion API Applications consists of four components:

- opinions Controller – responsible for providing REST API for opinions,
- messenger Component – provides synchronous or asynchronous communication with message broker,
- opinions Handler Component – it stores events, commands, handlers and business logic,
- opinions repository – provides communication with the database.

The author created all components except one. Messenger Component is an external library created by the community. The code is available on GitHub [37].

Figure 22, Figure 23 and Figure 24 present diagrams for Consultation API Application, Attorney API Application, Place API Application.

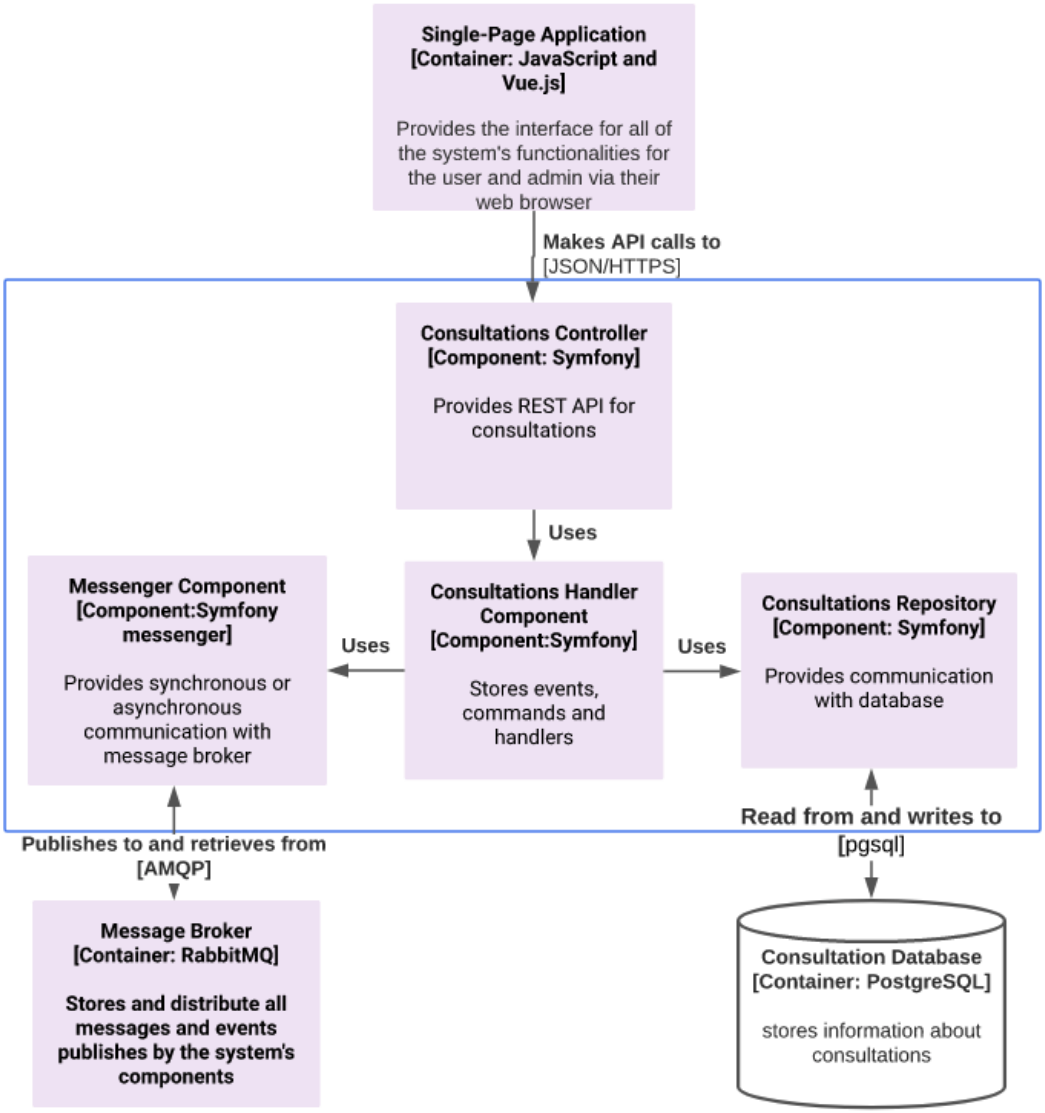

Figure 22 Consultation API Application component diagram; source: own elaboration

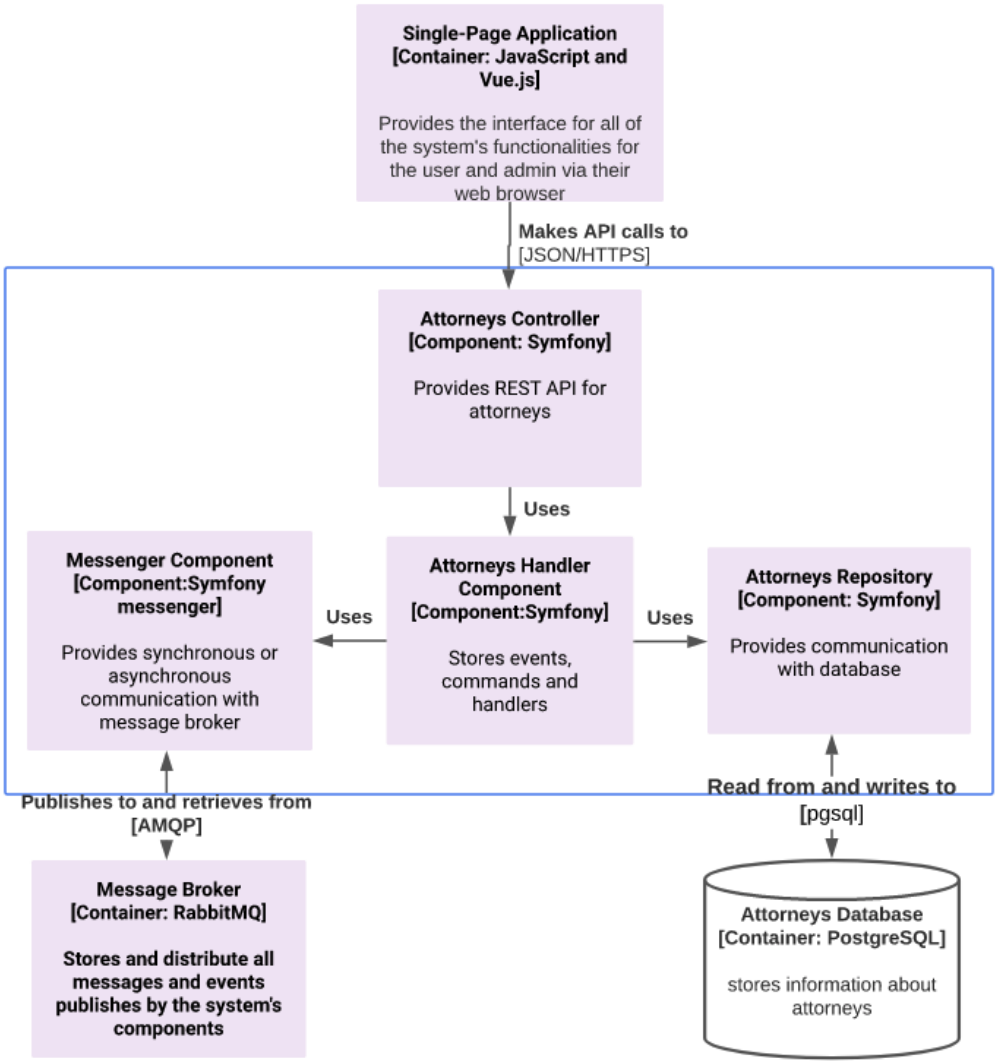

Figure 23 Attorney API Application component diagram; source: own elaboration

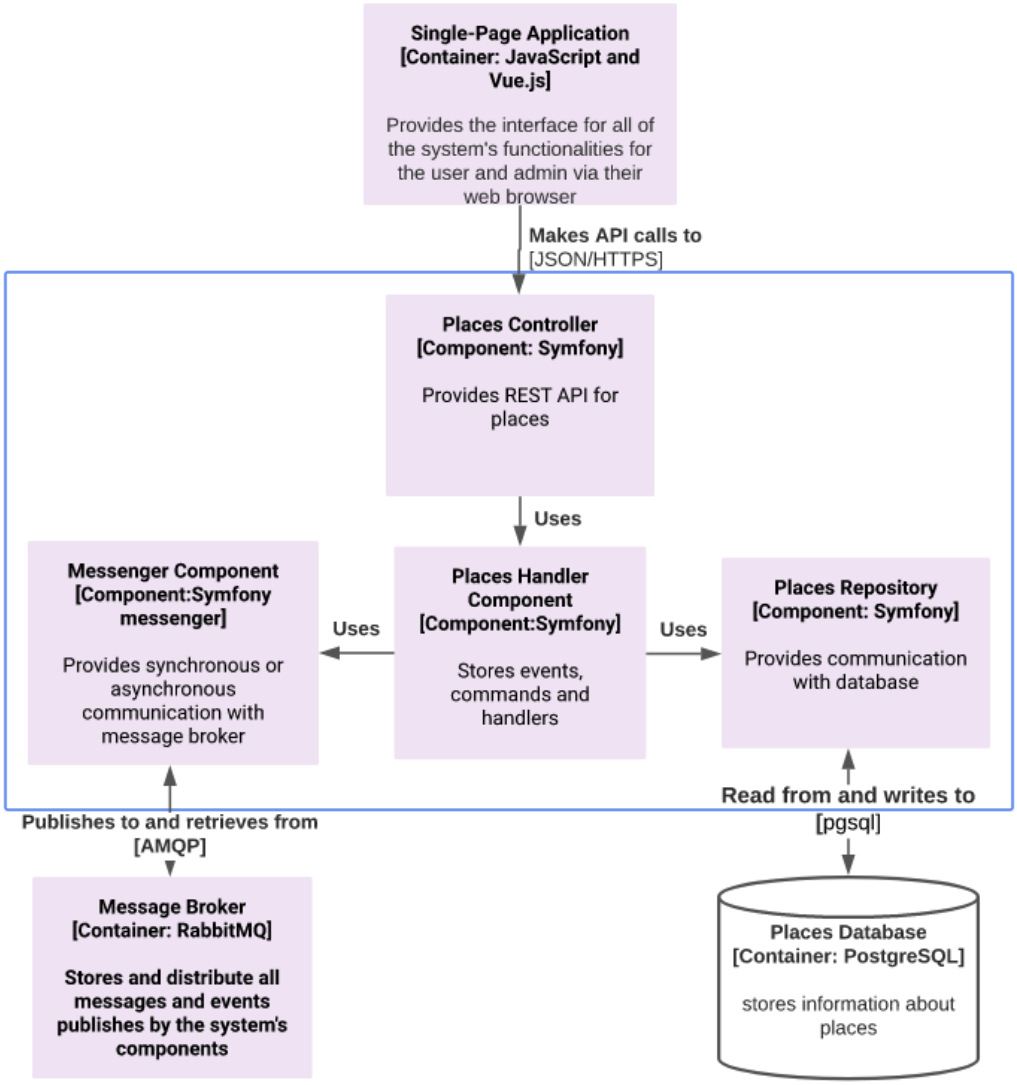

Figure 24 Place API Application component diagram; source: own elaboration

All backend services have similar components inside them and are created using the same technology. On the other hand, the Single-Page Application contains a different set of components. Figure 25 shows the details:

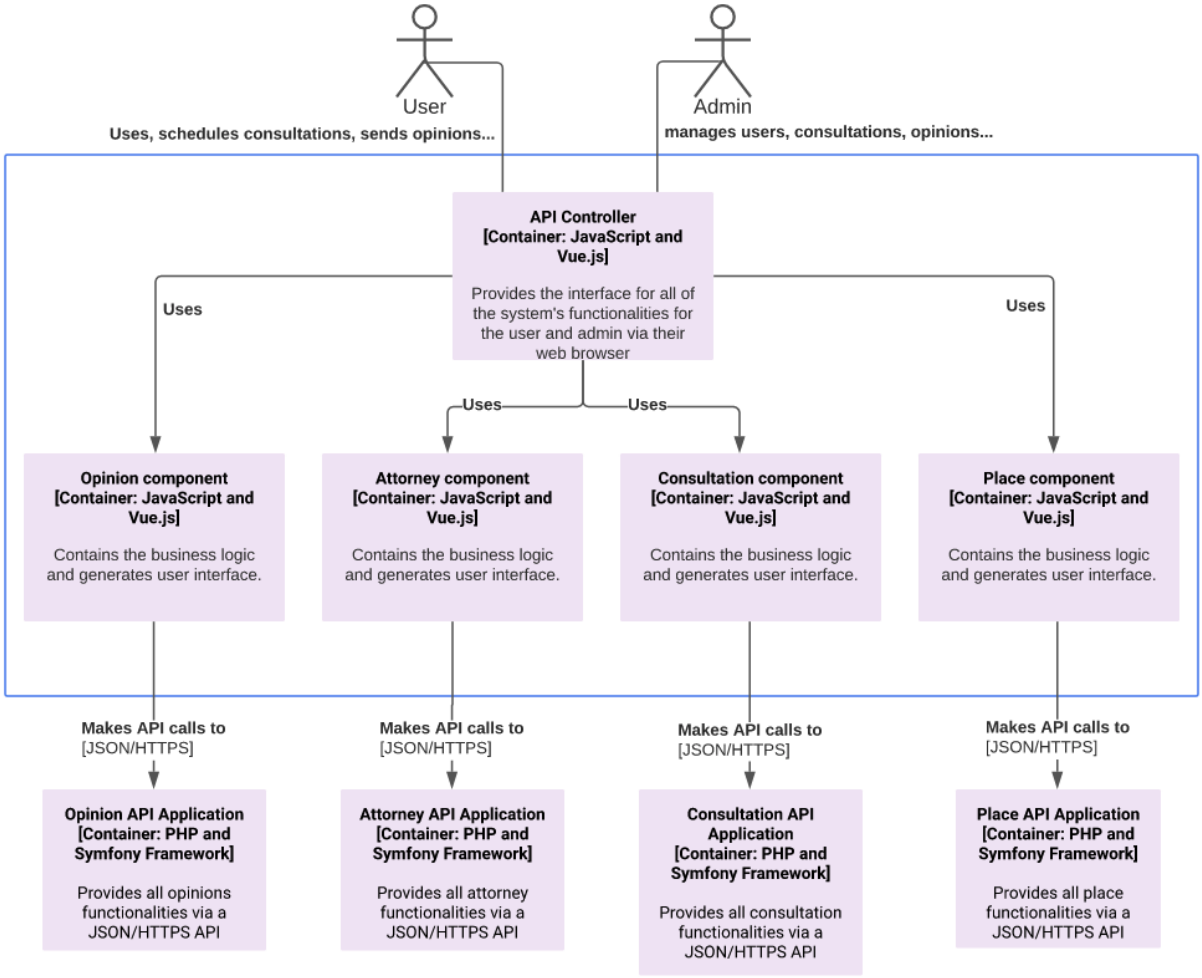

Figure 25 Single-Page Application component diagram; source: own elaboration

The single-Page application consists of:

1. API Controller – it distributes routing to specific components depending on the provided URL;
2. Opinion component – contains the business logic and generates the user interface. Make request calls to Opinion API Application;
3. Attorney component – contains the business logic and generates the user interface. Make request calls to Attorney API Application;
4. Consultation component – contains the business logic and generates the user interface. Make request calls to Consultation API Application;
5. Place component – contains the business logic and generates the user interface. Make request calls to Place API Application.

The database containers and a message broker are open-source containers. RabbitMQ has got its internal components explained on GitHub [38]. PostgreSQL offers extensive documentation on architecture [39].

6.3. Summary

This chapter presented implemented prototype. It described the domain of the solution, used technologies and new directions of development. Thanks to the chosen architecture and division of the

domain, there are many approaches to expand the system using new technologies, frameworks and databases. C4 Model presented the architecture from different levels of abstractions.

7. How to handle technical challenges

Using Kubernetes, Helm, and cloud services presents many opportunities to handle the technical challenges of microservices. In this chapter, the author proposed the approach to each of them.

7.1. Local development

Local development of a product should be a fast and pleasant experience. Choosing a correct approach is one of the deciding factors in whether the product is going to be finished on time or not. It is essential that people working in the codebase focus on providing the business value and not waste time on their local environment and its problems. It is even more complicated in the case of microservices architecture. There would be only one team working on one microservice in the perfect world, but this is rarely the case. A single developer often needs to have a few microservices working on the local machine. What is essential locally and what is worth a compromise, we talk about it in this chapter.

When working locally on a product, there are a few things worth mentioning. Firstly, the pace of development. This is crucial for the developer and the business owner to deliver the business functionalities fast and work well on production. If this is not the case, then a developer often cannot focus entirely on providing a business value because of the technical challenges with the environment. This can lead to frustration in both developer and business owner and can make the product fail. That is why working locally needs to be efficient, fast, and rewarding. At the same time, business functionalities must behave the same locally and in production. That is why local and production environments need to behave similarly. In other words, implementation details can vary, but in general, microservice opinion-service communicates with attorney-service over HTTP REST API in both cases.

Secondly, some aspects can be mitigated locally. Generally, scaling is a crucial aspect of microservices, but locally it does not matter. When working on a product, one instance of a microservice is enough. What is more, the security aspect is not that important. This point is a bit controversial and depends on a developed product. However, in a local environment, people should work on anonymised or fake data. Then, even if a developer leaks it, it will not hurt the business. All the traffic between microservices should not be encrypted. This enables finding bugs much faster.

Thirdly, the monitoring. This aspect is as important locally as on production. In both cases, we want to know what is going on in our architecture and where the potential source of our problems is. Without proper monitoring, this would not be possible.

Keeping in mind the paragraphs above, the author decided to choose a docker-compose to develop the prototype locally. Docker-compose provides a fast way of developing multi-container Docker applications. It uses a single YAML file that defines all these containers separately and provides an easy way of customisation. The author recommends documentation [11] that provides a step-by-step guide of creating this kind of environment and gives many examples for different technologies. Let us look at a docker-compose.yml example and explain the elements defined there. After that, we provide the docker-compose.yml used for the prototype. The example is presented on Listing 2:

Listing 2 Example of docker-compose.yml; source: [11]

```
version: "3.9"
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
      - logvolume01:/var/log
    links:
      - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

The config above contains three main elements:

1. version – top-level element, defining the version of docker-compose;
2. services – it is used to define all containers that can be run, in our case web and redis;
3. volumes – it defines volumes or mount paths that must be accessible by containers. These are the resources that hold data. It can be log files or databases.

Every service can have its own configuration. Let us look at the 'web' service:

1. build – it specifies the path to the Dockerfile that is used to build the container image;
2. ports – we can expose a port from the container to the host. In this case, we can access the web service with port 5000;
3. links – defines a network connection between the web and redis;
4. volumes - specifies the mapping between the host and the container. The first argument concerns the application code, the second one the logs. In that case, every time the container is destroyed, the logs are preserved.

The example above consists of two services. Then the config does not contain much information, which is a good thing because the cognitive load is low. However, working with many services means that this file can grow very fast. Let us look at the docker-compose config file used for preparing the prototype. It is presented on Listing 3:

Listing 3 Docker-compose used for prototype; source: own elaboration

```
version: "3.7"
services:
frontend-service:
  build: frontend-service
  container_name: findyoulawyer_frontend
  volumes:
    - "./frontend-service:/usr/src/app"
    - /usr/src/app/node_modules/
  ports:
    - "8080:8080"

consultation-db:
  image: 'postgres'
  container_name: consultation-db
  env_file:
    - .env
  volumes:
    - consultation-db-data:/var/lib/postgresql/data/
opinion-db:
  image: 'postgres'
  container_name: opinion-db
  env_file:
    - .env
  volumes:
    - opinion-db-data:/var/lib/postgresql/data/
  lawyer-db:
    ...
  place-db:
    ...
  consultation-api:
    ...
  opinion-api:
    ...
  lawyer-api:
    ...
  place-api:
    ...
  consultation-nginx:
    ...
  opinion-nginx:
    ...
  lawyer-nginx:
    ...
  place-nginx:
    ...
  rabbitmq:
    ...
volumes:
  db-data:
  rabbitmq-data:
  rabbitmq-logs:
  consultation-db-data:
  lawyer-db-data:
  opinion-db-data:
  place-db-data:
```

The details of the configuration for each service are encapsulated because of the amount of information. The whole file is attached to this paper. This configuration gives much flexibility in terms of adding new services or removing existing ones. What is more important, it synchronises the codebase changes in an instance. Thanks to that, the process of development is efficient and pleasant. It does not solve a lot of microservices challenges like scalability, but again, it is not essential when working locally, so it is a compromise we must take.

As we know how to work locally, we need to prepare a solution that works on production. In that case, the rules of the game change. Aspects that were not important locally need to be handled now. What are these aspects, how to leverage Kubernetes and Helm to our advantage and finally, how to deploy the application using Google Cloud?

7.2. Production deployment process

One of the tools that can be used to manage containers in a production environment is Helm. Using it can bring a lot of value and benefits for the organisation. It has got a high entry threshold but, with time, it turns out to be a helpful tool. Let us look at Figure 26 which presents a set of files used for the prototype:

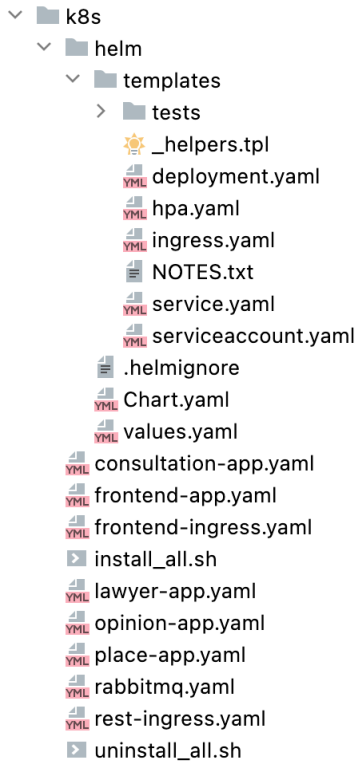

Figure 26 Helm Chart for the prototype; source: own elaboration

There are many possibilities for customisation. A few of them are listed below:

- network traffic inside a cluster,
- security policies,
- secret and configuration management,

- service discovery and load balancing,
- storage orchestration.

Implementation examples and technical challenges are presented in the next chapter.

The solution prepared for the prototype may look a bit different than usual. The author leveraged the situation where charts look very similar for every application. Thanks to that, there is only one helm directory that contains a generic, customisable configuration. It means that it uses variables that are assigned during runtime. There are a few YAML files that contain the application-specific values used in the helm chart configuration: *consultation-app.yaml*, *frontend-app.yaml*, *rabbitmq.yaml*, *opinion-app.yaml*, *place-app.yaml*, *lawyer-app.yaml*. They are included in Listing 4 and Listing 5.

Listing 4 consultation-app.yaml; source: own elaboration

```yaml
name: consultation-app
group: consultation
container:
  image: cezarygrabowski/find-your-lawyer-consultation
  imageTag: 1.3.0
  port: 3002
readinessProbeScheme: HTTPS
livenessProbeScheme: HTTPS
service:
  initialDelaySeconds: 10
  port: 3002
  name: consultation-app
  type: NodePort
  targetPort: 3002
  nodePort: 30002
  portName: consultation-port
imagePullSecrets:
  name: regcred2
replicaCount: 2
```

Listing 5 frontend-app.yaml; source: own elaboration

```yaml
name: frontend-app
group: frontend
container:
  image: cezarygrabowski/find-your-lawyer-frontend
  imageTag: 1.6.0
  port: 8080
  imagePullSecrets:
    - name: regcred
readinessProbeScheme: HTTPS
livenessProbeScheme: HTTPS
service:
  initialDelaySeconds: 100
  port: 8080
  name: frontend-app
  type: NodePort
  targetPort: 8080
  nodePort: 30001
```

```
  portName: frontend-port
```

These two listings above present the implementation details for each application. The frontend-service used in the prototype has similar needs to every other service, i.e., consultation service, despite having completely different technologies applied inside of them. Thanks to containerisation, the process of deployment is the same for all of them. The implementation details that vary are:

- container images and tags – every application contains a different container image that contains the source code and all software dependencies. They need to be hosted on the web since they are downloaded during deployment to the cluster. The most popular tool to build images and containers is Docker. That is what was used for the prototype. All applications' images were hosted on Dockerhub, a repository with images for most of the technologies. Using a tags system gives much flexibility in terms of deploying changes. If there is a new version of the source code, then it needs to be tagged. Both Gitlab and Github have this option. After that, the tag in the YAML file needs to be updated and changes deployed. Now the new version is on the cluster. What if there is a problem during deployment? Then simply use the old version: change the tag in the YAML file to the old one and deploy changes. This way, it is clean, efficient, and elegant;
- names of the applications – these are used as labels to differentiate Kubernetes resources. It can be used to listen to network traffic on selected pods or display logs from them;
- network traffic and ports – a few ports are specified for each app. They represent the ports published from the container to the other pods in the cluster and finally ports that can be used to access the pods directly from the external world. However, this is not mandatory and, very often, not recommended. In the used prototype, only frontend service needs to be available from the external network since it is an entry point to our system.

The process of deployment is straightforward. The author created a bash script that holds all necessary commands to deploy the project to the cluster. They are presented on Listing 6

Listing 6 Bash script that installs a system on a cluster; source: own elaboration

```
#!/bin/bash

helm install -f frontend-app.yaml frontend-app ./helm
helm install -f lawyer-app.yaml lawyer-app ./helm
helm install -f opinion-app.yaml opinion-app ./helm
helm install -f consultation-app.yaml consultation-app ./helm
helm install -f place-app.yaml place-app ./helm
helm install -f rabbitmq.yaml rabbitmq-app ./helm
```

Every line handles different applications. Every time there is a helm directory pointed out, a different group of variables is used. This is a script that is run only once, at the beginning. Every time the applications' config changes, we need to upgrade the cluster definition. The set of commands for that are presented on Listing 7:

Listing 7 Bash script that upgrades the chart; source: own elaboration

```
#!/bin/bash

helm upgrade -f frontend-app.yaml frontend-app ./helm
helm upgrade -f lawyer-app.yaml lawyer-app ./helm
helm upgrade -f opinion-app.yaml opinion-app ./helm
helm upgrade -f consultation-app.yaml consultation-app ./helm
helm upgrade -f place-app.yaml place-app ./helm
```

```
helm upgrade -f rabbitmq.yaml rabbitmq-app ./helm
```

Of course, there is no need to upgrade every chart in most cases. Please upgrade only those who need it. That is the whole deployment process. It can be integrated into a CI/CD pipeline to automate things. It often is. Nevertheless, there is only one thing left to decide on, a Kubernetes cluster.

The most popular solution is cloud services. It uses these tools' potential fully. Most of the biggest providers offer a service that prepares a cluster ready to use:

1. Amazon Elastic Kubernetes Service (EKS) – Amazon Web Services,
2. Google Kubernetes Engine – Google Cloud,
3. Azure Kubernetes Service – Azure.

For the prototype, we used Google Cloud since the interface is very intuitive, and the company allows us to use it for free for some time. Other providers offer similar conditions, and their services work as well as Google's. There are many options nowadays.

Google Cloud's documentation is very descriptive and provides all necessary information about creating and managing a cluster [8].

Since all the implementation details and the Helm charts are attached to this paper, there is no need to go into more details. More important is how to face the technical challenges using those tools.

7.3. Event-driven architecture

In order to enable event-driven architecture and make event-sourcing possible, we use RabbitMQ as a message broker and Symfony messenger. Figure 27 presents the messaging that exists in the created prototype:

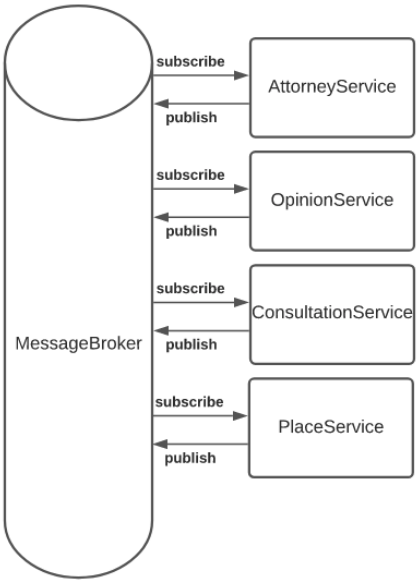

Figure 27 Messaging in the prototype; source: own elaboration

We have got a message broker and all backend services communicating with it. They both subscribe and publish to it to send and consume messages and events. Thanks to the user-friendly

RabbitMQ's panel presented in Figure 28, we can easily observe what messages are published. In the real-world application, we will log everything published and consumed.

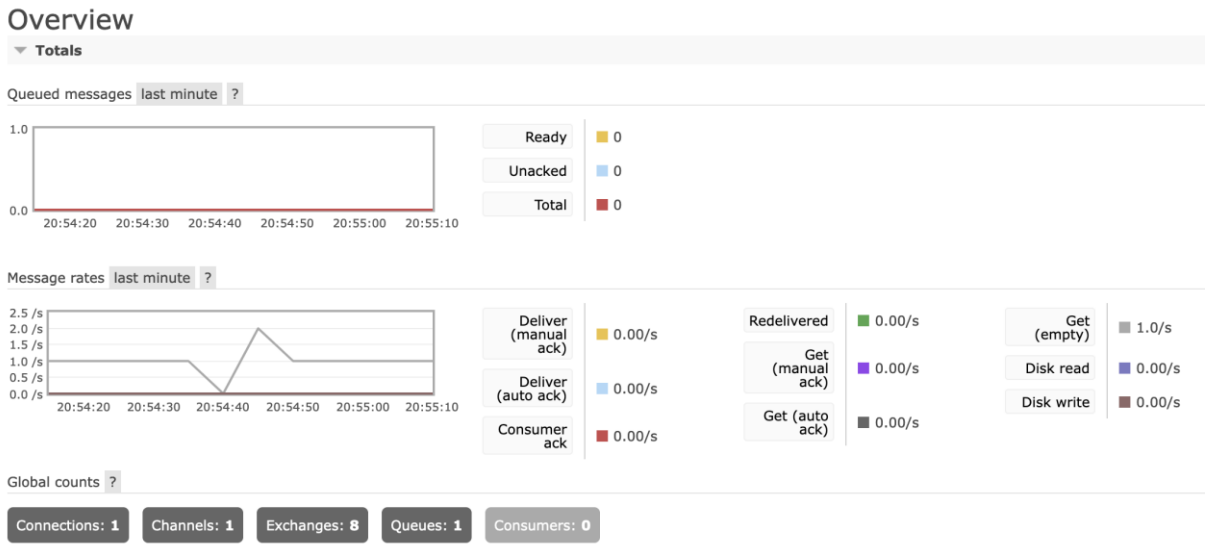

Figure 28 RabbitMQ dashboard; source: own elaboration

In order to create a connection between the service and RabbitMQ, we need to configure Symfony Messenger.

It is a library that enables synchronous and asynchronous queued message handling [40]. The configuration used is included on Listing 8:

Listing 8 Symfony messenger configuration; source: own elaboration

```
framework:
    messenger:
        transports:
            async: amqp://guest:guest@rabbitmq:5672/%2f/messages
        routing:
            App\MessageBus\Consultation\ConsultationScheduledEvent: async

App\MessageBus\Attorney\AttorneyUnavailableForConsultationEvent: async
            App\MessageBus\Attorney\AttorneyAvailableForConsultationEvent:
async

            App\MessageBus\Place\PlaceAvailableForConsultationEvent: async
            App\MessageBus\Place\PlaceUnavailableForConsultationEvent:
async

App\MessageBus\Place\AttorneyUnavailablePlaceMeetingRemovedEvent: async
```

The only thing we need to do is specify the address of transport for our asynchronous communication. Then we specify published events and the type of transport that we want. We can also define synchronous communication in the chase of some events.

7.4. Zero-downtime deployments

Look at the deployment process of the consultation-app. There is a Kubernetes service that controls the network traffic and a pod with the container that holds the old version of the application, as presented in Figure 29

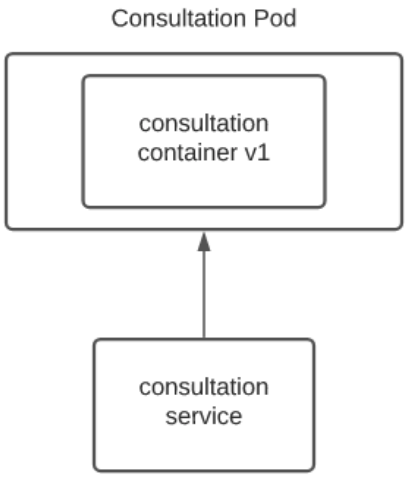

Figure 29 Kubernetes service and pod for consultation; source: own elaboration

A new pod is created with a new version of the application container during the deployment. When the pod is created correctly, the service is notified. Then the service redirects the traffic to the new pod, as presented in Figure 30:

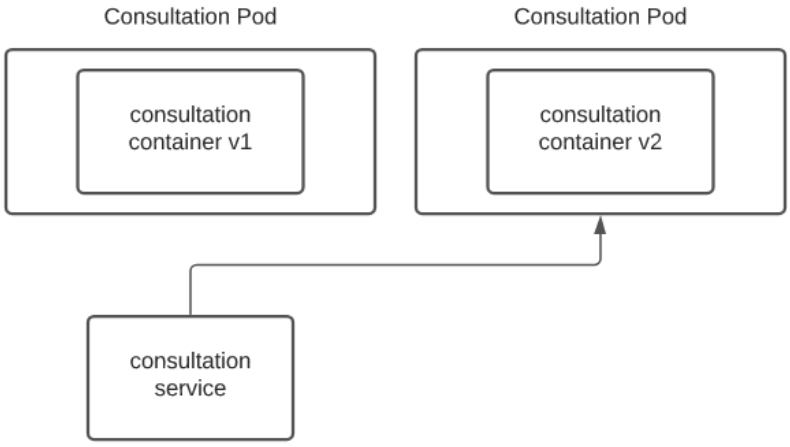

Figure 30 Traffic redirected to the new pod; source: own elaboration

Since the old one is not needed anymore, it is destroyed. This way, there is no downtime during deployment. While the new pod is being built, the old one handles the traffic. What in case of failure? The new pod is in an invalid state, and we need to examine logs to debug the error. Since there was no traffic redirection, the end-user is unaware of the failure during deployment.

7.5. Monitoring

This is one of the most important aspects of having a distributed system. In order to react to failures or errors, we need to capture them first. Monitoring must be holistic, which means that both applications and infrastructure should be observable:

1. metrics - CPU, memory, disk utilisation;
2. logs - Application logs and critical errors;
3. traces – interactions between Kubernetes resources.

Kubernetes provides the mechanism to monitor all the above. However, many tools are built on top of that to make it easier and provide configurable alerts.

7.6. Scalability

To handle scalability, Kubernetes provides a built-in mechanism. Say that the traffic on our website is enormous. We know the bottleneck is the consultation process thanks to the implemented monitoring. Thanks to our architecture, we do not need to scale all the applications. We can scale only the ones that need it; in this case – consultation-app.

The process of scaling is simple. Every deployment, which interacts with pods, provides a declarative way to update many configuration options, including the number of pods. This is where *ReplicaSet* needs to be introduced.

ReplicaSet is the Kubernetes object that ensures the correct number of pods running at any given time. Deployments use this object in the background and give the user a simple interface to use. The scaling configuration is presented on Listing 9:

Listing 9 Scaling configuration; source: own elaboration

```
spec:
  {{- if not .Values.autoscaling.enabled }}
  replicas: {{ .Values.replicaCount }}
  {{ else }}
  {{- toYaml .Values.autoscaling.config | nindent 2 }}
  {{- end }}
```

The “replicas” key describes the number of pods running simultaneously. By default, this number is one. What to do to scale it? Update the YAML file for consultation-app with `replicaCount` value set to two:

Listing 10 Updated consultation-app.yaml file; source: own elaboration

```
name: consultation-app
group: consultation
container:
  image: cezarygrabowski/find-your-lawyer-consultation
  imageTag: 1.3.0
  port: 3002
readinessProbeScheme: HTTPS
livenessProbeScheme: HTTPS
service:
  initialDelaySeconds: 10
  port: 3002
  name: consultation-app
```

```
  type: NodePort
  targetPort: 3002
  nodePort: 30002
  portName: consultation-port
imagePullSecrets:
  name: regcred2
replicaCount: 2
```

Figure 31 shows how the service balances the traffic between two pods:

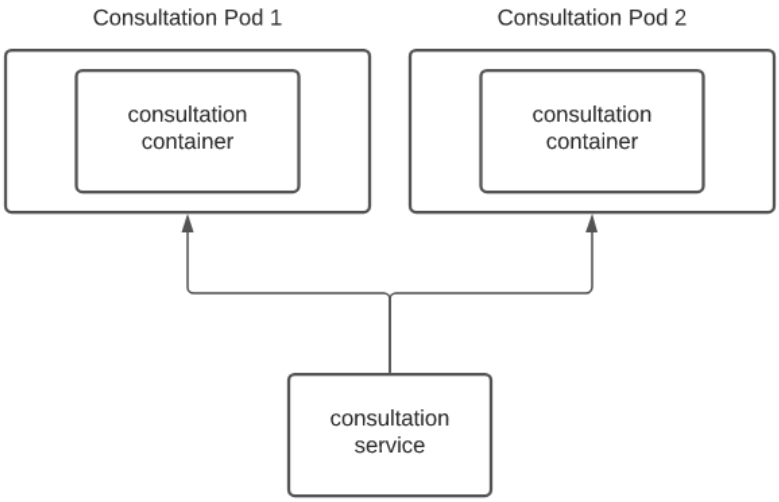

Figure 31 Service serving the traffic between two pods; source: own elaboration

Configuration on Listing 10 is not ready to auto-scale. It lacks a few options. Listing 11 provides additional configuration:

Listing 11 Autoscaling configuration; source: own elaboration

```
autoscaling:
  enabled: true
  config:
    minReplicas: 1
    maxReplicas: 10
    metrics:
      - type: Resource
        resource:
          name: cpu
          target:
            type: Utilization
            averageUtilization: 50
```

If the current number of pods working exceeds ten, the configuration above forces a scaling. Every new pod is added after the average utilisation is exceeded on the CPU. The current limit is 50%. Other metrics can be taken to account, such as memory or packets per second. Kubernetes provides a descriptive walkthrough [41].

7.7. Distributed transactions

When the database per service pattern is applied, we need to use distributed transactions. The most popular approach is the SAGA pattern. There are two approaches to implementing this pattern: choreography-based and orchestration-based.

The orchestration-based saga demands a single place in the codebase to integrate all participants. The advantage of this approach is that the logic is in one place. It is easier to introduce new changes, and participants do not have to know about commands published by other services. This approach is suitable for complex workflows. However, there are two drawbacks: a single point of failure and additional design complexity because we need to create an orchestrator. A platform worth mentioning that manages distributed transactions is Eventuate [42]. It was created by Chris Richardson, the author of https://microservices.io.

The platform provides two frameworks:

1. Eventuate Tram – for services that use a traditional persistence.
2. Eventuate Local – an event sourcing framework.

These remove the complexity of creating a new orchestrator and are worth becoming familiar with. The creator provides a few examples of using it on GitHub [43].

In the prototype, the author decided to use the choreography-based saga and implement it from scratch. This approach is more straightforward than the orchestration-based and can be a good choice for distributed transactions that do not span many services. However, the drawback of this solution is that no single place gathers all the logic. Furthermore, we need to maintain external documentation to keep track of the system.

Use case “Schedule consultation” described in used events and handlers that interacted with the Message Bus. Figure 32 presents all the events and handlers used for this flow:

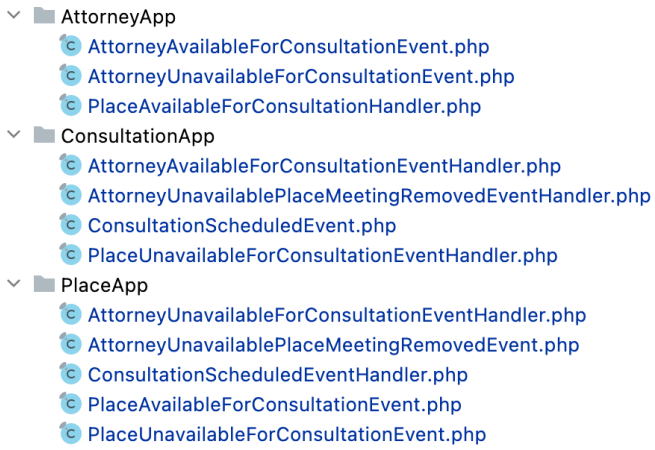

Figure 32 Handlers and events for scheduling consultation; source: own elaboration

Events are the data containers for payloads. Listing 12 presents one of them.

Listing 12 `AttorneyAvailableForConsultationEvent` source code; source: own elaboration

```
<?php
declare(strict_types=1);

namespace App\MessageBus\Attorney;

final class AttorneyAvailableForConsultationEvent
{
    public \DateTimeInterface $dateTime;
    public string $placeId;
    public string $consultationId;
    public string $attorneyId;

    public function __construct(
        \DateTimeInterface $dateTime,
        string $placeId,
        string $consultationId,
        string $attorneyId
    ) {
        $this->dateTime = $dateTime;
        $this->placeId = $placeId;
        $this->consultationId = $consultationId;
        $this->attorneyId = $attorneyId;
    }
}
```

Handlers, on the other hand, are much more interesting. First one is presented on Listing 13:

Listing 13 `PlaceAvailableForConsultationHandler` source code; source: own elaboration

```
public function __invoke(
        PlaceAvailableForConsultationEvent
placeAvailableForConsultationEvent
     ) {
        try {
            $attorneyMeeting = new AttorneyMeeting(
                $this->attorneyRepository->find(
                        $placeAvailableForConsultationEvent->attorneyId
                ),
                $this->placeRepository-
>find($placeAvailableForConsultationEvent->placeId),
                $placeAvailableForConsultationEvent->dateTime
            );
            $this->attorneyMeetingRepository->save($attorneyMeeting);
            $this->logger->info("Attorney available");

            $this->messageBus->publish(
                new AttorneyAvailableForConsultationEvent(
                    $placeAvailableForConsultationEvent->dateTime,
                    $placeAvailableForConsultationEvent->placeId,
                    $placeAvailableForConsultationEvent->consultationId,
                    $placeAvailableForConsultationEvent->attorneyId
                )
            );
        } catch (Exception $e) {
```

```
            $this->logger->info(
                "Attorney unavailable, consId: " .
                $placeAvailableForConsultationEvent->consultationId .
                ", placeId: " . $placeAvailableForConsultationEvent-
>placeId
            );
            $this->attorneyRepository->resetEntityManager();

            $this->messageBus->publish(
                new AttorneyUnavailableForConsultationEvent(
                    $placeAvailableForConsultationEvent->dateTime,
                    $placeAvailableForConsultationEvent->placeId,
                    $placeAvailableForConsultationEvent->consultationId,
                    $placeAvailableForConsultationEvent->attorneyId
                )
            );
        }
    }
}
```

The handler is invoked whenever the `PlaceAvailableForConsultationEvent` is consumed from the message bus. It creates an `attorneyMeeting` object and publishes the event called `AttorneyAvailableForConsultationEvent`. If the attorney is unavailable at the given date, the exception is thrown, and the handler publishes `AttorneyUnavailableForConsultationEvent`. Source code for the rest of the handlers and events is included in the attachments to this paper.

Figure 33 contains logs generated when the place is unavailable:

```
consumer_1  | 14:02:40 INFO      [messenger] Received message App\MessageBus\Consultation\Consu
ltationScheduledEvent ["message" => App\MessageBus\Consultation\ConsultationScheduledEvent { …}
,"class" => "App\MessageBus\Consultation\ConsultationScheduledEvent"]
consumer_1  | 14:02:40 INFO      [app] Place unavailable, consId: d6a18cf7-cd4c-4fc0-97d5-ee5d3
d102fd7, placeId: 3af83a07-a7a1-49ea-bb3d-1cd4337e9c9c
consumer_1  | 14:02:40 INFO      [messenger] Sending message App\MessageBus\Place\PlaceUnavaila
bleForConsultationEvent with async sender using Symfony\Component\Messenger\Bridge\Amqp\Transpo
rt\AmqpTransport ["message" => App\MessageBus\Place\PlaceUnavailableForConsultationEvent { …},"
class" => "App\MessageBus\Place\PlaceUnavailableForConsultationEvent","alias" => "async","sende
r" => "Symfony\Component\Messenger\Bridge\Amqp\Transport\AmqpTransport"]
consumer_1  | 14:02:40 INFO      [messenger] Message App\MessageBus\Consultation\ConsultationSc
heduledEvent handled by App\MessageBus\Place\ConsultationScheduledEventHandler::__invoke ["mess
age" => App\MessageBus\Consultation\ConsultationScheduledEvent { …},"class" => "App\MessageBus\
Consultation\ConsultationScheduledEvent","handler" => "App\MessageBus\Place\ConsultationSchedul
edEventHandler::__invoke"]
consumer_1  | 14:02:40 INFO      [messenger] App\MessageBus\Consultation\ConsultationScheduledE
vent was handled successfully (acknowledging to transport). ["message" => App\MessageBus\Consul
tation\ConsultationScheduledEvent { …},"class" => "App\MessageBus\Consultation\ConsultationSche
duledEvent"]
consumer_1  | 14:02:40 INFO      [messenger] Received message App\MessageBus\Place\PlaceUnavail
ableForConsultationEvent ["message" => App\MessageBus\Place\PlaceUnavailableForConsultationEven
t { …},"class" => "App\MessageBus\Place\PlaceUnavailableForConsultationEvent"]
consumer_1  | 14:02:40 INFO      [messenger] Message App\MessageBus\Place\PlaceUnavailableForCo
nsultationEvent handled by App\MessageBus\Consultation\PlaceUnavailableForConsultationEventHand
ler::__invoke ["message" => App\MessageBus\Place\PlaceUnavailableForConsultationEvent { …},"cla
ss" => "App\MessageBus\Place\PlaceUnavailableForConsultationEvent","handler" => "App\MessageBus
\Consultation\PlaceUnavailableForConsultationEventHandler::__invoke"]
consumer_1  | 14:02:40 INFO      [messenger] App\MessageBus\Place\PlaceUnavailableForConsultati
onEvent was handled successfully (acknowledging to transport). ["message" => App\MessageBus\Pla
ce\PlaceUnavailableForConsultationEvent { …},"class" => "App\MessageBus\Place\PlaceUnavailableF
orConsultationEvent"]
```

Figure 33 Place unavailable for visit logs; source: own elaboration

These is proof that even not complicated operations generate a significant portion of traffic and logs. This fact must be noted when using microservices architecture and database per service pattern.

The second handler worth mentioning is presented on Listing 14:

Listing 14 ConsultationScheduledEventHandler source code; source: own elaboration

```
public function __invoke(ConsultationScheduledEvent
$consultationScheduledEvent)
{
    $this->checkPlaceAvailability($consultationScheduledEvent->dateTime,
$consultationScheduledEvent->placeId, $consultationScheduledEvent-
>consultationId, $consultationScheduledEvent->attorneyId);
}

private function checkPlaceAvailability(\DateTimeInterface $dateTime,
string $placeId, string $consultationId, string $attorneyId)
{
    try {
        $placeMeeting = new PlaceMeeting();
        $placeMeeting->setPlace($this->placeRepository->find($placeId));
        $placeMeeting->setDateTime($dateTime);
        $this->placeMeetingRepository->save($placeMeeting);
        $this->logger->info("Place available, consId: " . $consultationId .
", placeId: " . $placeId);
```

```
        $this->messageBus->publish(new
PlaceAvailableForConsultationEvent($dateTime, $placeId, $consultationId,
$attorneyId));
    } catch (\Throwable $e) {
        $this->placeMeetingRepository->resetEntityManager();
        $this->logger->info("Place unavailable, consId: " . $consultationId
. ", placeId: " . $placeId);
        $this->messageBus->publish(new
PlaceUnavailableForConsultationEvent($dateTime, $placeId, $consultationId,
$attorneyId));
    }
}
```

Handler `ConsultationScheduledEventHandler` is called on `ConsultationScheduledEvent`. It checks the place availability using `placeRepository`. If the place is free, then the `PlaceAvailableForConsultationEvent` is published, in other case `PlaceUnavailableForConsultationEvent`.

7.8. Self-healing services

Kubernetes is aware of all pods that are running on the cluster. When one of them crashes, it is notified. Then, it tries to create a new one and replace them. Nevertheless, it is crucial to know what the process looks like and what mechanisms the Kubernetes use to find out the current state of the given pod.

Kubernetes uses two mechanisms:

- liveness probe,
- readiness probe.

The liveness probe indicates whether the application is running or not. Sometimes applications crash, and the liveness probe notifies the system about that event. However, when the application starts, it needs some time to warm up. During this time, it is not able to handle any traffic. That is why the Kubernetes service should not redirect the network traffic to the pod just yet. That is why the readiness probe exists. When those two indicators are fulfilled, service serves traffic to the given pod.

The configuration applied in the prototype is presented on Listing 15:

Listing 15 Liveness and readiness probe configuration; source: own elaboration

```
livenessProbe:
  initialDelaySeconds: {{ .Values.service.initialDelaySeconds }}
  httpGet:
    path: /
    port: {{ .Values.container.port }}
    scheme: {{ .Values.readinessProbeScheme }}
readinessProbe:
  initialDelaySeconds: {{ .Values.service.initialDelaySeconds }}
  httpGet:
    path: /
    port: {{ .Values.container.port }}
    scheme: {{ .Values.readinessProbeScheme }}
```

For every probe, there is an option to delay checking the pod's health and application readiness. It is useful, especially when the application is extensive and deployment time grows noticeably. Not correctly specifying this option can lead to false results. For example, say that deployment time lasts for two minutes. The liveness probe starts working after ten seconds. By default, it checks the pod every ten seconds for three times. It means that after thirty seconds, the pod is treated as failed. Then the new pod is created, and the same thing happens again. The solution is to define the delay.

The other option to define is the way that probes are executed. We use the `httpGet` method, which is an HTTP request to a specific port and path. If we do not provide a host, then Kubernetes uses the IP of the pod. Configurations for consultation-app and frontend-app are presented on Listing 16 and Listing 17.

Listing 16 Consultation-app probes configuration; source: own elaboration

```
readinessProbeScheme: HTTPS
livenessProbeScheme: HTTPS
service:
  initialDelaySeconds: 10
  port: 3002
  name: consultation-app
```

Listing 17 Frontend-app probes configuration; source: own elaboration

```
readinessProbeScheme: HTTPS
livenessProbeScheme: HTTPS
service:
  initialDelaySeconds: 100
  port: 8080
  name: frontend-app
```

The flexibility of defining those values for every application pays off because, for example, the frontend-app needs a lot more time to build than the consultation-app and other backend services. That is why the `initialDelaySeconds` is set to one hundred. Fads

8. Conclusions

The following chapter describes possible extensions to the crafted prototype. Chapter 8.1 proposes how the solution can be expanded. The domain is easily extended, thanks to the flexible architecture. Because of technology heterogeneity, we are not limited to the tools we have chosen until now. Chapter 8.2 summarizes author's thoughts about technical challenges in microservices architecture.

8.1. Further development

There are many possible extensions to the created prototype – starting with the domain.

The next step can be introducing law firms into our system. They consist of attorneys working there. Thanks to that, users can look at the law firm with positive opinions and then choose an attorney only from a specific company. After that, a payment system can be introduced, ending up as another microservices. Attorneys and law firms could pay for being promoted on the listings.

There is flexibility in terms of technology that can be used. We have the flow where the new opinion is created. With time, we will have it automated as much as possible. That is where we can use machine learning and TensorFlow, a python library. With that, we can create a new microservice that will be responsible for checking if the opinion is trustworthy or not.

This architecture allows for rapid growth and hiring more people. They will not be working on one application but many. Thanks to the correct division to microservices initially, the pace of growth is rapid and constant. In the case of monolithic applications, the codebase would grow much faster. After some time, every developer would work on the same application. This could lead to a situation where, at some point, the system would not grow as fast as it could because of constant deployments, codebase conflicts between developers and slow runtime on local machines. Correctly implemented microservices architecture is the answer to those problems.

The prototype allows applying the CQRS pattern in the future [44]. Right now, we have four backend services, and their addresses need to be defined in the frontend service. Because of that, the frontend service needs to be aware of the implementation details of the system. However, it should send a query command to, for example, query attorneys, and it should not be aware of what element of the system exactly is handling this query. The only communication point would be a message broker and a proper query bus. Updated container diagram of C4 model would look like on Figure 34

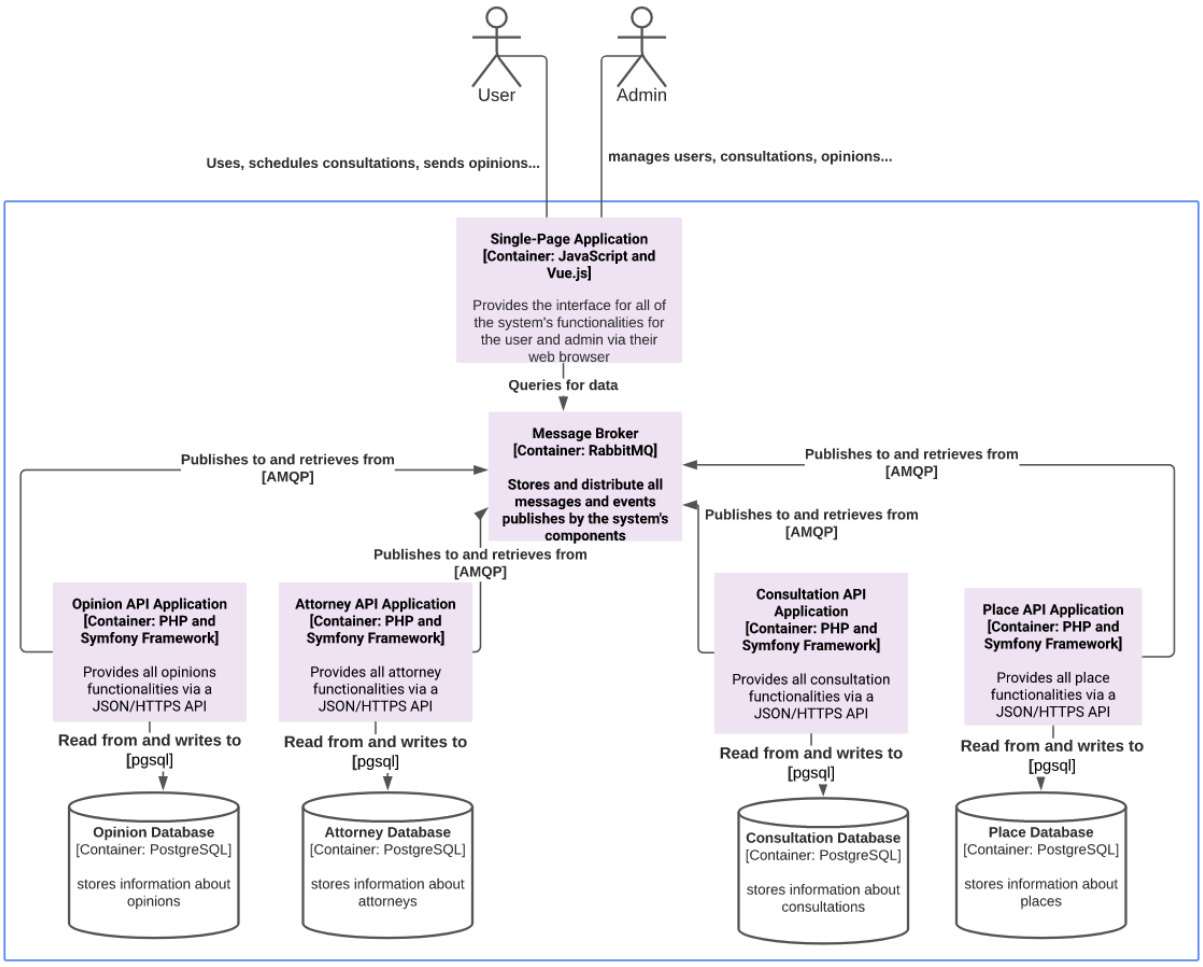

Figure 34 Updated container component of C4 model; source: own elaboration

8.2. Summary

Microservices architecture has become more and more popular. It can offer a lot for large-scale systems and those demanding technology heterogeneity. Nevertheless, developing this kind of system is challenging. It requires experience and expertise. Many companies want the system that is created using that architectural style. Its advantages are undeniable. However, while it provides many benefits, it creates many threats. The company that wants to apply this architecture must be aware of the presented challenges. The team developing the product needs to be able to handle those. The business use cases need to justify this choice. This architecture can be a double-edged sword. If not applied correctly, it can do more harm than good.

Technical challenges are inevitable while working with the microservices architecture. There are many things that the team developing the product needs to deal with. Only then we can use the full potential of this architecture. One example is scaling. This architectural style creates the opportunity to scale the chosen parts of the system independently. This is easier and cheaper to do than scaling the monolithic application. However, we need to have a relevant tool that offers an easy way to do it. In this paper, Kubernetes was used.

In many cases, the database is the bottleneck and needs to be scalable. However, scaling applications with the business code is not enough. To do that, we must apply the database per service

pattern. As a result, we need to use distributed transactions sooner or later. However, that introduces more complexity and slows down development significantly.

Everything said above is true, given that the domain is correctly divided into microservices. If not, then we have more problems. A company may need to scale two instead of one microservices and their databases to scale a given process. This is not optimal. Incorrect domain division creates many other problems. The implications and how to do it correctly are topics for another paper. Methodologies like Domain-Driven Design are helpful in this case.

When the company works on a new product, it is worth thinking of microservices as the right approach in their case. What is essential in the beginning is the pace of the product development. Microservices architecture does not speed it up - it slows it down significantly. The slower the development, the less chance of success. The company's CEO does not care if the application architecture is the best if the product does not earn money. Therefore, starting with the microservices architecture is rarely a good idea. Especially if the domain is not fully discovered. That is why there is an alternative way.

Very often, it is a good idea to start with a modular monolith. This is a monolith application that consists of business modules. It is easy to change the modules' boundaries since they are in the same codebase and the development pace is much faster than in microservices architecture. Having separate modules offers an easier way to extract the chosen domain as a microservice than the typical monolith application, which can easily be the spaghetti code. Martin Fowler writes about this approach as Monolith First [22]. However, the process of extracting the domain is not trivial as well. This is a topic for a whole book. Sam Newman wrote one, called "Monolith to Microservices". It is very pragmatic and comprehensive.

The author believes that this paper gave a comprehensive look at the microservices architecture from a technical perspective. It offered examples of handling the technical challenges, introduced necessary concepts and tools that get the job done. It is important not to follow the popular trends blindly but choosing the correct architecture for a specific system.

9. Bibliography

[1]. S. S. Mike Loukides, https://www.oreilly.com/radar/microservices-adoption-in-2020/, 15 6 2020. [Online]. Available: https://www.oreilly.com/radar/microservices-adoption-in-2020. [Accessed 8 1 2022].

[2]. M. Fowler, "https://martinfowler.com/bliki/BoundedContext.html", 15 01 2014. [Online]. [Accessed 24 11 2021].

[3]. C. Richardson, "https://microservices.io/patterns/data/database-per-service.html", [Online]. [Accessed 24 11 2021].

[4]. K. Xiang, "https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture", 01 10 2018. [Online]. [Accessed 24 11 2021].

[5]. S. Brown, "https://c4model.com/," [Online]. [Accessed 9 1 2022].

[6]. SensioLabs, "https://symfony.com/doc/current/index.html", [Online]. [Accessed 9 1 2022].

[7]. E. You, "https://vuejs.org/", [Online]. [Accessed 9 1 2022].

[8]. G. C. Team, "https://cloud.google.com/kubernetes-engine/docs/how-to/creating-a-zonal-cluster", [Online]. [Accessed 27 12 2021].

[9]. Docker, "https://docs.docker.com", [Online]. Available: https://docs.docker.com/ [Accessed 26 12 2021].

[10]. Docker, "https://docs.docker.com/get-started/#what-is-a-container", [Online]. [Accessed 22 12 2021].

[11]. Docker, "https://docs.docker.com/compose/", [Online]. [Accessed 22 12 2021].

[12]. T. K. Authors, "https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/", 23 7 2021. [Online]. [Accessed 22 12 2021].

[13]. T. K. Authors, "https://kubernetes.io/docs/home/", [Online]. [Accessed 27 12 2021].

[14]. T. K. Authors, "https://kubernetes.io/docs/concepts/workloads/pods/", 17 10 2021. [Online]. [Accessed 23 12 2021].

[15]. T. K. Authors, https://kubernetes.io/docs/concepts/workloads/controllers/ deployment, 17 9 2021. [Online]. [Accessed 26 12 2021].

[16]. C. Team, "https://www.ibm.com/cloud/blog/containers-vs-vms", 9 4 2021. [Online]. [Accessed 22 12 2021].

[17]. V. Synenka, "https://customerthink.com/top-10-companies-using-cloud-and-why", 31 8 2021. [Online]. [Accessed 26 11 2021].

[18]. Microsoft, "https://azure.microsoft.com/en-us/overview/what-is-cloud-computing", [Online]. [Accessed 26 11 2021].

[19]. K. Singh, M. Çalışkan, O. Mihályi and P. Pscheidl, Java EE 8 Microservices, Packt, 2018.

[20]. D. Pinte, "http://www.pinte.ro/Blog/DesignPatterns/Clean-Architecture-An-alternative-to-traditional-three-layer-database-centric-applications/37", [Online]. [Accessed 10 1 2022].

[21]. IBM, "https://www.ibm.com/docs/en/cics-ts/5topic=processing-acid-properties-transactions”, 2021. [Online]. [Accessed 27 11 2021].

[22]. M. Fowler, "https://martinfowler.com/bliki/MonolithFirst.html”, 3 6 2015. [Online]. [Accessed 27 11 2021].

[23]. S. Tilkov, "https://martinfowler.com/articles/dont-start-monolith.html”, 9 6 2015. [Online]. [Accessed 27 11 2021].

[24]. J. Lumetta, "https://buttercms.com/books/microservices-for-startups/should-you-always-start-with-a-monolith”, [Online]. [Accessed 28 11 2021].

[25]. S. Newman, Monolith to Microservices, vol. 1st edition, O'Reilly Media, 2019.

[26]. S. Newman, Building microservices, vol. 2nd edition, O'Reilly Media, 2019.

[27]. M. Fowler, "https://martinfowler.com/tags/domain%20driven%20design.html”, [Online]. [Accessed 28 11 2021].

[28]. E. Evans, Domain-Driven Design: Tackling complexity in the Heart of Software, vol. 1st edition, Pearson Education, 2004.

[29]. V. Vernon, Implementing Domain-Driven Design, Addison-Wesley, 2011.

[30]. AWS, "https://aws.amazon.com/autoscaling/”, [Online]. [Accessed 30 11 2011].

[31]. B. M. Sasaki, "https://neo4j.com/blog/acid-vs-base-consistency-models-explained/”, 13 11 2018. [Online]. [Accessed 9 1 2022].

[32]. C. Richardson, "https://microservices.io/patterns/data/saga.html”, [Online]. [Accessed 20 12 2021].

[33]. M. Fowler, "https://martinfowler.com/eaaDev/EventSourcing.html”, [Online]. [Accessed 20 12 2021].

[34]. S. Brown, "https://www.infoq.com/articles/C4-architecture-model/”, 25 6 2018. [Online]. [Accessed 09 01 2022].

[35]. S. Brown, "https://c4model.com/#CodeDiagram”, [Online]. [Accessed 9 1 2022].

[36]. S. Brown, "https://c4model.com/#ContainerDiagram”, [Online]. [Accessed 9 1 2022].

[37]. SensioLabs, "https://github.com/symfony/messenger”, [Online]. [Accessed 9 1 2022].

[38]. VMware, "https://github.com/rabbitmq/internals”, [Online]. [Accessed 9 1 2022].

[39]. The PostgreSQL Global Development Group, "https://www.postgresql.org/docs/9.1/tutorial-arch.html”, [Online]. [Accessed 9 1 2022].

[40]. SensioLabs, "https://symfony.com/doc/current/messenger.html”, [Online]. [Accessed 9 1 2022].

[41]. T. K. Authors, "https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/”, [Online]. [Accessed 8 1 2022].

[42]. C. Richardson, "https://eventuate.io/post/eventuate/2020/02/24/why-eventuate.html”, [Online]. Available: https://eventuate.io/post/eventuate/2020/02/24/why-eventuate.html. [Accessed 29 12 2021].

[43]. C. Richardson, "https://github.com/eventuate-tram/eventuate-tram-sagas”, [Online]. [Accessed 29 12 2021].

[44]. M. Fowler, "https://martinfowler.com/bliki/CQRS.html", [Online] , [Accessed 05 01 2022].

[45]. Helm authors, https://helm.sh/docs/, [Online], [Accessed 21 01 2022].

[46]. Kubernetes Authors, https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/cluster-intro/, [Online]. [Accessed: 22 01 2022]

[47]. Kubernetes Authors, https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/, [Online]. [Accessed: 22 01 2022]

[48]. Hound Technology, Inc., https://www.honeycomb.io, [Online]. [Accessed 23 12 2021]

Attachment A - List of figures, listings and tables

Figures

Tables

Listings

Attachment B – Glossary of used terminology and abbreviations

Spaghetti code - is a pejorative phrase for unstructured and difficult-to-maintain source code. (“Spaghetti code - Wikipedia”)

Vault – a tool created by Hashicorp for handling secrets. https://www.vaultproject.io/

ArtifactHUB – a repository with ready-to-use helm charts

Attachment C – Docker compose configuration

```
version: "3.7"
services:
  frontend-service:
      build: frontend-service
      container_name: findyourlawyer_frontend
      volumes:
        - "./frontend-service:/usr/src/app"
        - /usr/src/app/node_modules/
      ports:
        - "8080:8080"
  consultation-db:
    image: 'postgres'
    container_name: consultation-db
    env_file:
      - .env
    volumes:
      - consultation-db-data:/var/lib/postgresql/data/
  opinion-db:
    image: 'postgres'
    container_name: opinion-db
    env_file:
      - .env
    volumes:
      - opinion-db-data:/var/lib/postgresql/data/
  lawyer-db:
    image: 'postgres'
    container_name: lawyer-db
    env_file:
      - .env
    volumes:
      - lawyer-db-data:/var/lib/postgresql/data/
  place-db:
    image: 'postgres'
    container_name: place-db
    env_file:
      - .env
    volumes:
      - place-db-data:/var/lib/postgresql/data/
  consultation-api:
    container_name: consultation-api
    build:
      context: ./docker/php
```

```
  depends_on:
    - consultation-db
  volumes:
    - ./consultation-api:/var/www/api:cached
opinion-api:
  container_name: opinion-api
  build:
    context: ./docker/php
  depends_on:
    - opinion-db
  volumes:
    - ./opinion-api:/var/www/api:cached
lawyer-api:
  container_name: lawyer-api
  build:
    context: ./docker/php
  depends_on:
    - lawyer-db
  volumes:
    - ./lawyer-api:/var/www/api:cached
place-api:
  container_name: place-api
  build:
    context: ./docker/php
  depends_on:
    - place-db
  volumes:
    - ./place-api:/var/www/api:cached
consultation-nginx:
  image: nginx:stable-alpine
  container_name: consultation_nginx
  expose:
    - "80"
  ports:
    - '8081:80'
  depends_on:
    - consultation-api
    - consultation-db
  volumes:
    - ./consultation-api:/var/www/api
    - ./docker/consultation-nginx/default.conf:/etc/nginx/conf.d/default.conf
opinion-nginx:
  image: nginx:stable-alpine
  container_name: opinion_nginx
```

```
  expose:
    - "80"
  ports:
    - '8082:80'
  depends_on:
    - opinion-api
    - opinion-db
  volumes:
    - ./opinion-api:/var/www/api
    - ./docker/opinion-nginx/default.conf:/etc/nginx/conf.d/default.conf
lawyer-nginx:
  image: nginx:stable-alpine
  container_name: lawyer_nginx
  expose:
    - "80"
  ports:
    - '8083:80'
  depends_on:
    - lawyer-api
    - lawyer-db
  volumes:
    - ./lawyer-api:/var/www/api
    - ./docker/lawyer-nginx/default.conf:/etc/nginx/conf.d/default.conf
place-nginx:
  image: nginx:stable-alpine
  container_name: place_nginx
  expose:
    - "80"
  ports:
    - '8083:80'
  depends_on:
    - place-api
    - place-db
  volumes:
    - ./place-api:/var/www/api
    - ./docker/place-nginx/default.conf:/etc/nginx/conf.d/default.conf
rabbitmq:
  image: rabbitmq:3-management-alpine
  container_name: 'findyoulawyer_rabbitmq'
  ports:
    - 5672:5672
    - 15672:15672
  volumes:
    - rabbitmq-data:/var/lib/rabbitmq/
```

```
      - rabbitmq-logs:/var/log/rabbitmq/
volumes:
  db-data:
  rabbitmq-data:
  rabbitmq-logs:
  consultation-db-data:
  lawyer-db-data:
  opinion-db-data:
  place-db-data:
```