2022

ZAKLJUČNA NALOGA (FINAL PROJECT PAPER)

KOBRIN

UNIVERZA NA PRIMORSKEM
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN INFORMACIJSKE TEHNOLOGIJE

ZAKLJUČNA NALOGA
(FINAL PROJECT PAPER)

ARHITEKTURA MIKROSTORITEV V PRODUKCIJSKEM OKOLJU
(MICROSERVICES-BASED ARCHITECTURE IN PRODUCTION ENVIRONMENT)

ALEKSANDR KOBRIN

UNIVERZA NA PRIMORSKEM
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN
INFORMACIJSKE TEHNOLOGIJE

Zaključna naloga
(Final project paper)

Arhitektura mikrostoritev v produkcijskem okolju

(Microservices-based architecture in production environment)

Ime in priimek: Aleksandr Kobrin
Študijski program: Računalništvo in informatika
Mentor: doc. dr. Peter Rogelj

Koper, avgust 2022

Ključna dokumentacijska informacija

Ime in PRIIMEK: Aleksandr KOBRIN

Naslov zaključne naloge: Arhitektura mikrostoritev v produkcijskem okolju

Kraj: Koper

Leto: 2022

Število listov: 47 Število slik: 13

Število referenc: 11

Mentor: doc. dr. Peter Rogelj

Ključne besede: Arhitektura programskih produktov, Mikrostoritve, Načrtovanje sistema

Izvleček: Arhitektura mikrostoritev (ang. microservices), je v zadnjem času pridobila na priljubljenosti zaradi prednosti, ki vključujejo razširljivost, zanesljivost, odpornost na napake in enostavnost vzdrževanja. Arhitektura mikrostoritev je znana že nekaj časa, vendar je svoj potencial izkazala šele v zadnjem obdobju zahvaljujoč dosežkom na področju strojne in programske opreme, z infrastrukturo v oblaku in virtualizacijo.
Glavni cilj arhitekture mikrostoritev je učinkovita uporaba vseh razpoložljivih virov, tako tehnično, kot poslovno. Zanimanje zanjo se povečuje z rastjo aplikacij v realnem svetu in potrebi po njihovem hitrem spreminjanju oziroma prilagajanju novim potrebam. Tradicionalne monolitne aplikacije je namreč težko vzdrževati, če se obseg nudenih storitev hitro povečuje. Veliko podjetjih po svetu zato pospešeno investira v razdelitev svojih obsežnih storitev na večje število mikrostoritev. To ne olajša le tehničnega razvoja storitev, ampak tudi projektno vodenje z vidikov oblikovanja ekip, porazdelitve dela, neodvisnosti ekip in podobno. Vendar pa arhitektura mikrostoritev lahko povzroča tudi večje težave kot monolitni pristop, če se mikrostoritve ne uporabljajo pravilno. Pomanjkljivostim se lahko izognemo s skrbnim načrtovanjem arhitekture aplikacije, to pa od načrtovalca zahteva predznanje in izkušnje, saj kompleksnost vsakega modula prispeva k kompleksnosti celotnega sistema. V tej zaključni nalogi predstavljamo arhitekturo mikrostoritev in predstavljamo različne vidike zasnove sistema, ki jih demonstriramo na praktičnem primeru aplikacije za vseživljenjsko učenje zdravstvenega osebja.

Key document information

Name and SURNAME: Aleksandr KOBRIN

Title of the final project paper: Microservices-based architecture in production environment

Place: Koper
Year: 2022
Number of pages: 47 Number of figures: 13
Number of references: 11
Mentor: Assist. Prof. Peter Rogelj, PhD
Keywords: Software architecture, Microservices, System design

Abstract: Microservices-based architecture has gained a lot of popularity recently. Microservices are a way to organize the system to be more scalable, reliable, fault-tolerant and maintainable by design. Professionals argue microservices to be a rebranded SOA, however it is only partially true. Despite the approach was invented quite some time ago, it has unleashed potential now thanks to achievements in hardware and software tooling as well. It includes cloud-based infrastructures, virtualisation. The main goal of microservice-based architecture is to utilise all available resources efficiently - both technical and business-related. The rise of the microservices hype curve was caused by real-world applications which began to grow quicker than ever before and change with pace. Monolithic applications are hard to maintain while they scale. So, a lot of companies around the world invest their resources in fragmentation of their huge applications into microservices. This approach helps not only to technical aspects of development, but also to project management: building the team, work distribution, independent teams and so on. However, microservices can cause bigger problems than monolithic approach if they are used improperly. In order to avoid drawbacks, the architecture of an application must be planned carefully and requires an architect to have prior knowledge and experience, as we transfer the complexity from each module to the system in general. In this thesis we intend to present the microservice architecture, describe different options regarding particular aspects of the system design and show a practical implementation example.

ACKNOWLEDGEMENTS

This thesis would not been possible without the support of many people. Many thanks and appreciation to my mentor and adviser, Assist. Prof. Peter Rogelj, for careful guidance on each step of producing this paper in very dark times the world faced. Thanks to my close ones for being mental support as well. I am also grateful to Marc Anthony Berends for conducting a proofreading.

Thanks to the University of Primorska for three years of in-depth learning. In combination with employment, it increased my knowledge and allowed to practice it immediately.

LIST OF CONTENTS

1 INTRODUCTION 1
1.1 Service definition 1
1.2 Advantages of microservices over monolithic architecture 2
1.3 Service-oriented architecture 2
1.4 Project description 3
2 DESIGNING MICROSERVICES 4
2.1 Decomposition factors 5
2.1.2 Decomposition by subdomain 6
2.2 Transactions across the system 8
2.2.1 Choreographic “Saga” 9
2.2.2 Orchestration-based “Saga” 9
2.3 Composition 10
3 DATA STORAGES 12
3.1 Shared database 12
3.2 Database per service 14
3.2.1 Drawbacks 16
3.3 Implementation of data storages 17
3.3.1 CQRS 18
3.3.2 Event sourcing 19
3.4 Types of databases 20
3.5 Cache 21
4 COMMUNICATION BETWEEN SERVICES 22
4.1 HTTP protocol 23
4.2 RPC 23
4.3 Message brokers and queues 24
4.4 Databases as queues 28
5 SERVER INFRASTRUCTURE FOR MICROSERVICES 29
5.1 Web Server 31
6 FINAL ARCHITECTURE 32
6.1 Services and communication between them 33
6.2 Data storage 35
6.3 High level description of a flow 36
6.4 Flexibility 36
7 CONCLUSIONS 37
8 REFERENCES 39

LIST OF FIGURES

Figure 1: Use case diagram for the defined system 4
Figure 2: Illustration of Strangler Fig pattern in steps [5] 5
Figure 3: Illustration of services being split into two domains after decomposition 7
Figure 4: Illustration of transaction for starting a webinar with Scheduler service being a director 10
Figure 5: Final functionality decomposition into isolated services 11
Figure 6: Illustration of multiple services using a single shared database [9] 13
Figure 7: Illustration of CAP Theorem and properties which can be achieved simultaneously 17
Figure 8: Communication between three service in the system using SQL View 19
Figure 9: Communication between three service in the system using events stored in the database 20
Figure 10: Message distribution between queues based on the routing key [10] 26
Figure 11: Final communication flow between services in the system 28
Figure 12: High-level view of the system and its interaction with real users 33
Figure 13: Illustration of the final architecture achieved after consideration of all aspects 37

LIST OF ABBREVATIONS

Abbreviation	**Definition**
ACID	Properties of Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
BASE	Basically Available, Soft state, Eventual Consistency
BFF	Backend-For-Frontend
CAP	Consistency, Availability, Partition tolerance
CPU	Central Processing Unit
CQRS	Command and Query Responsibility Segregation
CRUD	Combination of basic operations (Create/Read/Update/Delete)
DB	Database
DBMS	Database Management System
DDD	Domain Driven Development
DDOS	Distributed Denial of Service
FTP	File Transfer Protocol
GDRP	General Data Protection Regulation
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IT	Information Technology
JSON	JavaScript Object Notation
JWT	JSON Web Token
noSQL	Not only Structured Query Language
REST	Representational State Transfer
RPC	Remote Procedure Call
RPI	Remote Procedure Interface
SOA	Service-Oriented Architecture
SQL	Structured Query Language
SSR	Server-Side Rendering
STOMP	Streaming Text Oriented Message Protocol
MQTT	Message Queue Telemetry Transport
AMQP	Advanced Message Queuing Protocol
TTL	Time-To-Live
URL	Uniform Resource Locator

1 INTRODUCTION

Microservice-based architecture is a relatively new approach to system development. Despite the idea behind has been introduced a long time ago, the rise of the microservices hype curve was caused by real-world applications which began to grow quicker than ever before and change with pace. The approach aims to help developers with such issues as scalability, fault-tolerance, security and other problems which arise when an application starts to grow. This growth may be multi-dimensional and sometimes involves not only technical aspects, but represents business needs as well: bigger teams, larger codebase, server capacity overflow due to the traffic spikes and other scalability issues modern engineers face. The main idea behind such architecture is modularity as it helps to achieve low coupling between units of the system on the high-level view and scalability. Another reason for microservice-based architecture rise is efficient resources utilisation - both technical and business-related.

Considering their advantages, microservices can be used for most start-up applications, as well as long-term projects. In that sense, we can observe microservices as a way to scale technical resources at the same time as business scales up. Furthermore, big companies such as Netflix, Uber, and Amazon invest a lot of time and effort into microservice-based approach [8]. Companies are adopting microservices even if initially they had big monolithic applications, as microservices can be introduced into systems gradually taking part by part of functionality.

However, microservices can cause bigger problems than monolithic approach if they are used improperly. To avoid drawbacks, the architecture of an application must be planned carefully and requires an architect to have prior knowledge and experience, as we transfer the complexity from each module onto the system in general.

1.1 Service definition

A service is a unit, or sometimes a node, of a system built using microservices. Service represents a set of one or more features grouped by some factors based on either technical motives or business domain. As mentioned above, that approach decreases coupling and increases inner cohesion. Service may have its own dependencies, environment or even language it is written in.

Technology independence is one of the main reasons why microservices are popular among big companies, as they tend to have high-load systems and picking the right tool for

each need is a must. It's worth mentioning that services are much easier to test using both unit and integration testing.

1.2 Advantages of microservices over monolithic architecture

Historically, as machines had only one CPU core and networks had small speed and bandwidth, most programs were developed in a monolithic way. Obviously, as time passed and the codebase grew, projects faced problems described before: component design became too complex to handle or to test, only vertical scalability was available, onboarding of new employees was taking too much time. However, the biggest threat was one error which could fail the whole project. Despite we talk about the past, sometimes we still can find such applications over the Internet.

Hardware technologies developed over time as well as business in IT. Every minute of unavailability today might cost a lot. Microservices allow to isolate an error, keeping the rest of services up.

To sum up, key benefits microservices-based architectures include:

- Small, maintainable codebase for each service which is focused on one task
- Codebase of a service is independent from other services
- Loosely coupled services with fault isolation
- Better intuitive scalability
- Greater business ability
- Independently deployable

Those benefits are not achieved automatically. We will observe how to practically achieve each of them in the next chapters.

1.3 Service-oriented architecture

Some people strongly believe that microservices are just rebranded SOA [11], but in reality, these terms are different: microservices are one way to implement SOA. Service-oriented architecture represents a methodology of modular approach for software development, based on providing remote use of distributed, loosely coupled and easily replaceable components with standardised interfaces using standardised protocols. This definition perfectly aligns with microservice architecture. In general terms, microservices are an evolution from SOA inheriting the philosophy and raising it into more fine-grained services, since, unlike SOA, they force resource isolation.

1.4 Project description

The goal of the project is to develop and introduce a platform for a life-long learning of medical staff.

The core functionality is the following:

- providing access to webinars and articles with information which must be accessed only by approved worker of medical field
- tracking learning progress, rewarding users with points which are mandatory to collect during a year in order to keep their medical licence

The platform is built from the ground up. Microservice-based architecture was picked in advance. The goal is to create reliable, fault-tolerant infrastructure with almost no down-time, where features can be deployed independently. Time is very limited, so the architecture must be flexible enough to make particular shortcuts but remain manageable without technical debt prevailing.

Worth mentioning, the audience of the product is very targeted and narrow. The traffic is not expected to be high in the near future. However, architecture must be flexible enough to support such concepts as sharding or similar in case of such need.

Speaking of overall functionality, there are several key points to consider:

- Users should be able to register. During registration users should be able to upload confirmation of being a doctor or other medical worker.
- Users should be able to access and read research and news articles.
- Users should be able to leave comments for articles.
- Users should be able to access and watch webinars. During webinar, time which users spend on watching should be tracked. It will be a criterion of points reward.
- Users should be able to communicate between each other and with hosts during webinars.
- Administrators should be able to create articles and schedule webinars.
- Users should be able to receive notifications about upcoming webinars and new articles using different communication channels, such as Push notifications or email.
- Data analysts should be able to safely perform analysis over users’ activity

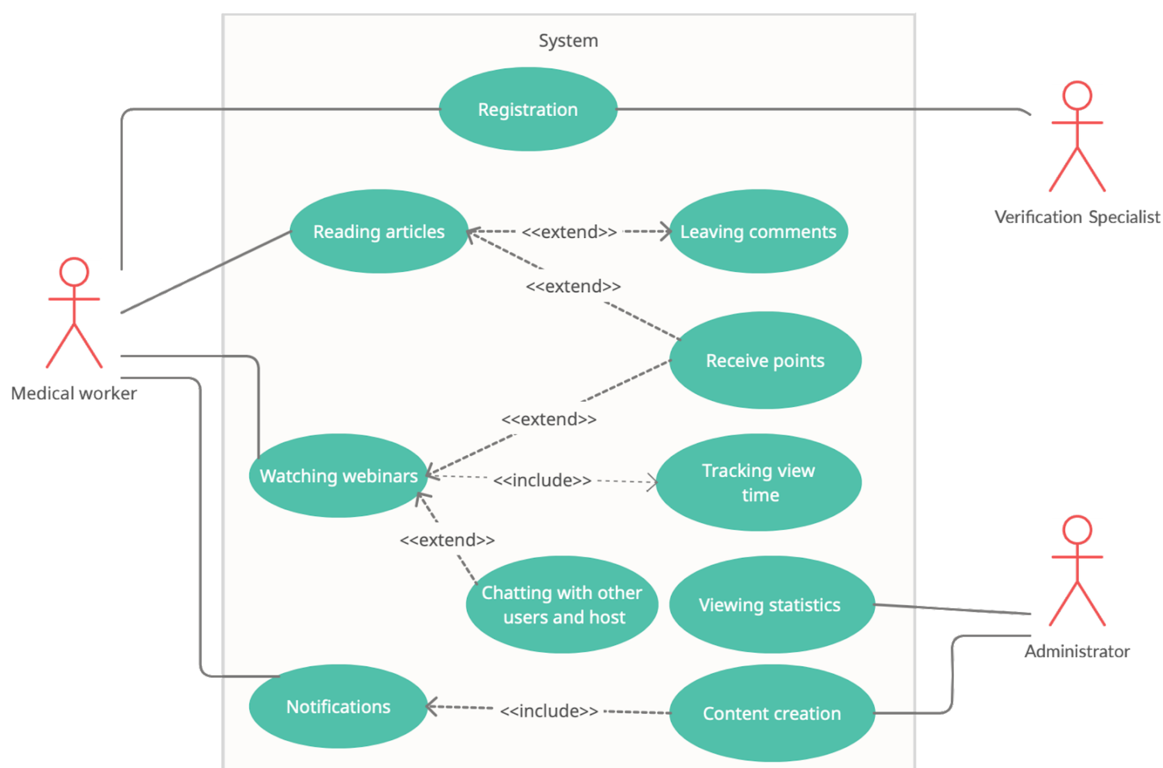

Figure 1: Use case diagram for the defined system

2 DESIGNING MICROSERVICES

The architect's most complex task is to design the distributed model for the system, keeping in mind scalability, data storages, future feature delivery speed, business processes and many more. In this chapter we will design such a model by splitting planned functionality into independent modules, however there will be references to next chapters such as data storages and communication between services, since it is impossible to design a system without consideration of these topics. It is worth mentioning that the transformation process of existing applications into microservices is even more difficult, as it provides a lot less manoeuvre and requires additional cautiousness while integrating. Usually, the biggest issue is a mix of eventual and strong consistencies of data. Therefore, there exist architectural patterns such as the Strangler Fig Pattern [7]. In general, access to the application which is being rewritten must be provided through an interface module. This module is meant for routing requests to legacy system or a new one depending on whether requested functionality has been rewritten already or not. Also, it can route requests to both in order to have a fallback if a new service fails. However, our application is built from the ground up, hence there is no need to deal with that kind of problem.

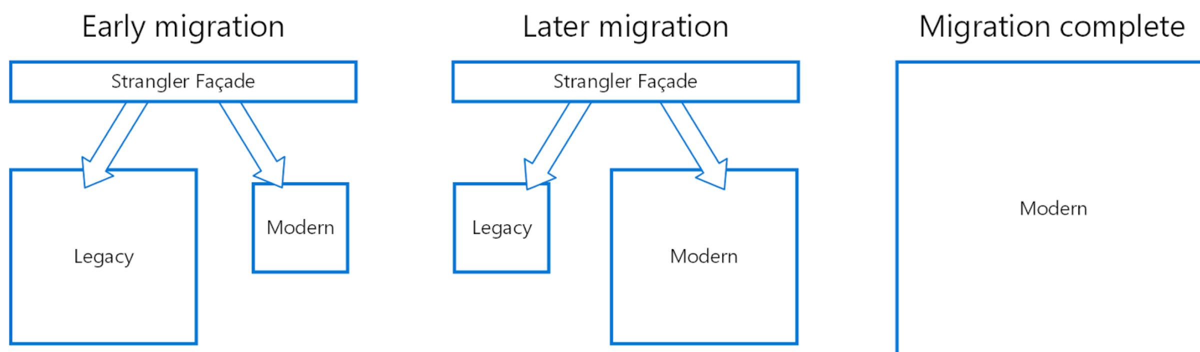

Figure 2: Illustration of Strangler Fig pattern in steps [5]

2.1 Decomposition factors

Decomposition factors usually include expected application load, business capabilities and fault isolation. To decompose the system, we need to define service categories, which services of our system will be labelled with. Basically, those are inherited from service-oriented methodology:

- Functional services (i.e., business services), which are critical for the business aspect of an application.
- Enterprise services, which serve to implement functionality.
- Application services, which are used to develop and deploy application.
- Infrastructure services, which are instrumental for backend processes like security and authentication.

We will group services in our system using this classification later in this chapter.

Apart from standard guidelines like Single Responsibility Principle, there are several strategies on system decomposition, although in this chapter we will focus on two of them which are widespread the most.

2.1.1 Decomposition by business capability

This approach relates to methodology of business-driven development which advises building systems around business processes inside of a company. A business capability is a concept from business architecture modelling. It is something that a business does in order to generate value. A business capability often corresponds to a business object. For instance, in the project this thesis is based on, main business objects are webinars and reward points. Even though they are related, management flows are different: points are requested from a government agency after the webinar has ended and the number of

required points is known along with the number of people who watched a particular webinar. Considering that, we can already fix two services: Webinar Management Service and Points Management Service.

One of the biggest benefits of this approach is fault isolation, which, together with perceived infrastructure, helps a business to remain functioning even if one module fails.

2.1.2 Decomposition by subdomain

In this philosophy services correspond to Domain-Driven Design (DDD) subdomains. DDD refers to the application's problem space - the business - as the domain. A domain consists of multiple subdomains. Each subdomain corresponds to a different part of the business.

The idea of subdomains is a higher view of business capabilities. Hence, decomposition by business capability is a special case of decomposition by subdomain where each domain or subdomain has one service. As a drawback, domain-oriented model would apply additional complexity since the system would involve more services. However, it is unavoidable in certain cases when, for example, additional fault isolation for the sake of availability is required.

Now, we can dive into designing the architecture for the application based on DDD subdomains. Since the best way for future maintenance is to focus on business capabilities, the list of functionalities would come handy.

According to the list of high-level functionalities mentioned in the introduction chapter, it is possible to identify following services and their types:

- Infrastructure services
    - Authentication and authorization service
- Enterprise services
    - Notification center (including emails, push notifications, notification on website)
    - Service for live chats and presence checks - Live-Webinar service
    - Points service
    - Webinar Management
    - Article Management
- Functional services
    - Statistics service

- SSR (Server-Side Rendering)

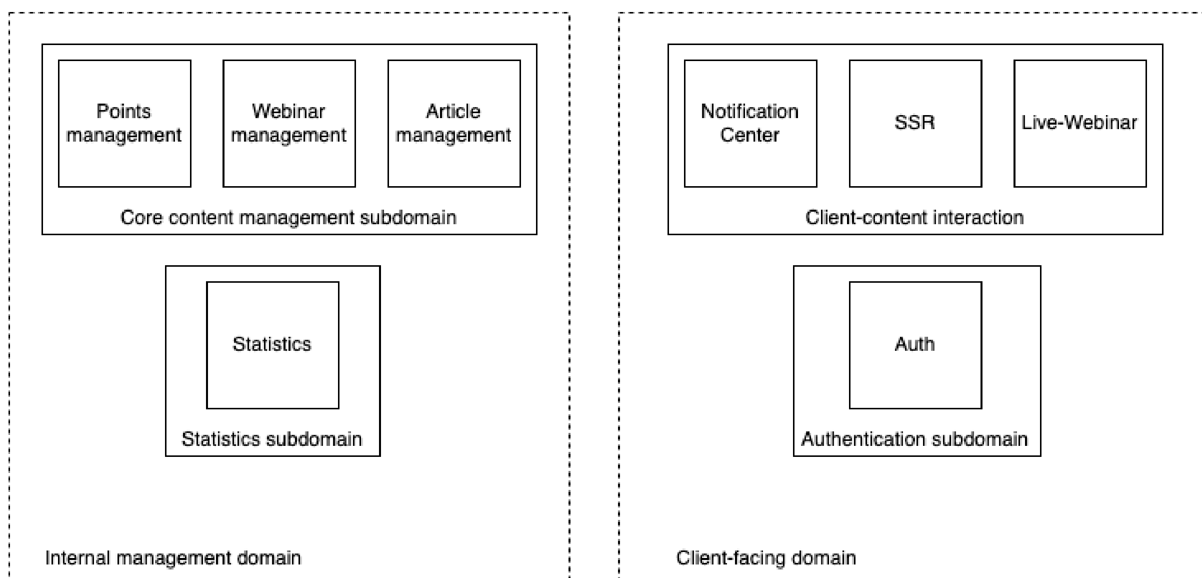

Figure 3: Illustration of services being split into two domains after decomposition

What we can do is to divide application functionality into two domains:

- Backoffice, or internal management, domain - a lot of work will be done by administrators such as arranging webinars, requesting points granted by state.
- Client-facing domain - there is no access to services from the inside, therefore all consumers of functionality are users - medical stuff.

However, since the audience is narrow and data load is not that high, we can combine some services into one to reduce the system complexity. This is a power side of DDD approach - ability to combine and divide functionality without losing isolation between subdomains. For example, management of webinars and articles is going to be supervised by the same employees. Points service can also be included. Since points are granted after webinar viewing, points and content are linked. We can call this new service by its subdomain name - Data Management service. Including all three functions into one service, data consistency will be guaranteed, unlike the case when this information is separated into different storages. Deeper discussion on data storages will be presented in the corresponding chapter.

On the other hand, services can be split due to one of the following reasons:

- too big codebase
- infrastructure reasons:
- resilience
- heavy load on part of functionality
- resource sharing between services

- service requires special knowledge

If we take a look at “Notification center” service, we can observe the following behaviour: for instance, users subscribed to a particular category of webinars should receive an email one hour before starting time. When the service gets a message with a request, which includes webinar information, it has to identify a group of users which will receive an email. Implementation of this procedure inside “Notification center” service would cause several problems. Firstly, we bound this service and data storage. Also, if an email distribution service fails in the process, it would mean a portion of emails won’t be delivered.

To avoid those problems, we leave only rendering and dispatching functionality to “Notification center” service. True “microservice”-way to solve this issue is to introduce an additional service for more task generation. So, in the final picture, the Task-generator service will receive a notification, prepare tasks with user information to pass to the email distribution service.

At this point a question arises: how should these services receive requests to execute their functionalities? It would be too inconvenient to invoke it manually. Usually, there is a service called Scheduler which notifies other services about certain events using different message approaches. Exact communication channels will be established in the Communication chapter.

The key observation: spinning a request across multiple services results in increased latency and creates greater dependability, which in its order reduces availability. Therefore, less services are involved into one transactional action - more stable the system is. Also, it is important to mention that microservices blossom only when each service is designed stateless. Stateless structure is a key to horizontal scaling. Otherwise, unneeded complexity for synchronisation of multiple instances of the same service will be applied.

2.2 Transactions across the system

Consider the following scenario: when it is time to start the webinar the system must react with more than one action. First, the status of the webinar must be changed in order for the client to be able to see that it went live. Secondly, room for chat and presence checks must be created in “Live-Webinar” service. There can be other instructions in the future for this action.

For adequate system behaviour on the client's side, it is mandatory to handle an error on one or more services. Usually, the issue is solved with a transaction-like approach. The biggest obstacle for transactions in distributed systems is the asynchronous nature of the architecture. A participant (a service) cannot immediately detect an error in the sequence of transactions as each instruction is run in different processes and may take some time.

To resolve the issue there is a pattern called Saga. Saga is nothing more but merely a sequence of local transactions grouped by logical operation. There are two main ways to implement transactions which spin several microservices with this pattern:

- Choreographic
- Orchestration

Which one to choose is heavily dependent on the use case.

2.2.1 Choreographic “Saga”

In general terms, the underlying principle of this approach is similar to the chain of functions which invoke the next function. Each local transaction in one service informs the next service to start its local transaction. As you can see, this way can lead to very complex solutions. Moreover, as mentioned above, in a distributed system it is better to keep operations across modules asynchronous to avoid bottlenecks. To roll back the state of the system, a service should get a notification of an error happening in the next service in the chain. Unfortunately, there is no way to get a notification if that next service shuts down. Therefore, a timeout should be planned for each step of the transaction. All of it applies additional complexity, increases coupling and is not suitable for transaction spinning more than 2-3 services.

However, choreographic Saga can be a good solution for short transactions, especially if the transaction is happening in one domain and all services are owned by one team.

2.2.2 Orchestration-based “Saga”

With an orchestration approach, a transaction is controlled by one service. This management can be done by a service participating in a transaction or by an external conductor. This conductor knows exactly what state the transaction is in, where it failed and how to roll back the system state.

One of the drawbacks this approach has is the following: if the conductor of a transaction fails, how do we roll back the state of services which have already processed their local

transactions? There are different ways to address this issue, such as the Event Sourcing pattern. It will be described in chapter on Data Management.

We can add logging, so in case of service failing, on its restart it will roll back the changes.

With those two approaches we can achieve consistency. For our case it is very useful to go with orchestration-based saga as we encapsulate all operations, which run on a specific time to one service - Scheduler - and, on the other hand, it directs transactions related to these events.

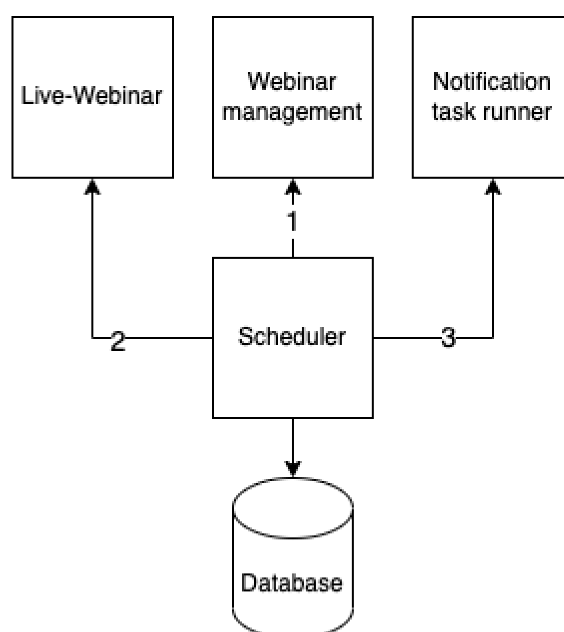

Figure 4: Illustration of transaction for starting a webinar with Scheduler service being a director

At this point there are no other cases in the system where we would need transactions. However, if there will be a need for them, the following rule should be applied: if transaction operation involves more than two services, then orchestration approach should be used - otherwise, choreography should be applied. Again, it is desirable to reduce the number of services in a transaction as much as possible.

2.3 Composition

As was mentioned previously, one drawback of microservices-based architecture is latency. It relates to requests which have to be processed by more than one service. An architect should model the system in a way that business and functional specifications will not take too much time or, in this case, involve many services. API composition pattern [6] can be applied. In general, this pattern helps to aggregate several operations into one, so

that the client does not need to perform these operations on its own. The main function of this pattern is to reduce coupling between services and encapsulate data retrieval.

In our system, there is an SSR service for rendering pages for clients. Usually, tools used for SSR can act as normal web servers, serving pages with injected information. By business requirements, users must be able to access their personal information, statistics on their activity on a portal, points they have earned and notifications.

By design of a graphical interface, there is a personal user page where they can access all four types of information: personal information, statistics, points and notifications which they missed. Even though a SSR service can make requests to each service which owns the information, or a database directly, it would increase its load. It may result in slower response time. The solution in this particular case is an additional service called Backend-For-Frontend (or shortly BFF).

Backend-For-Frontend - pattern which is designed to aggregate information needed for GUI. It is an adaptation of the Composition pattern. To simplify data collection, we can implement a query by defining an API Composer, which invokes the services that own the data and performs an in-memory join of the results. Also, it provides isolation between system layers - client-facing services will not have access to internal services, such as content management or statistics services, directly.

To sum up, what is the current structure of the system, see the picture below. You can notice 3 front services which must authenticate users, so the rest of the system may skip doing it.

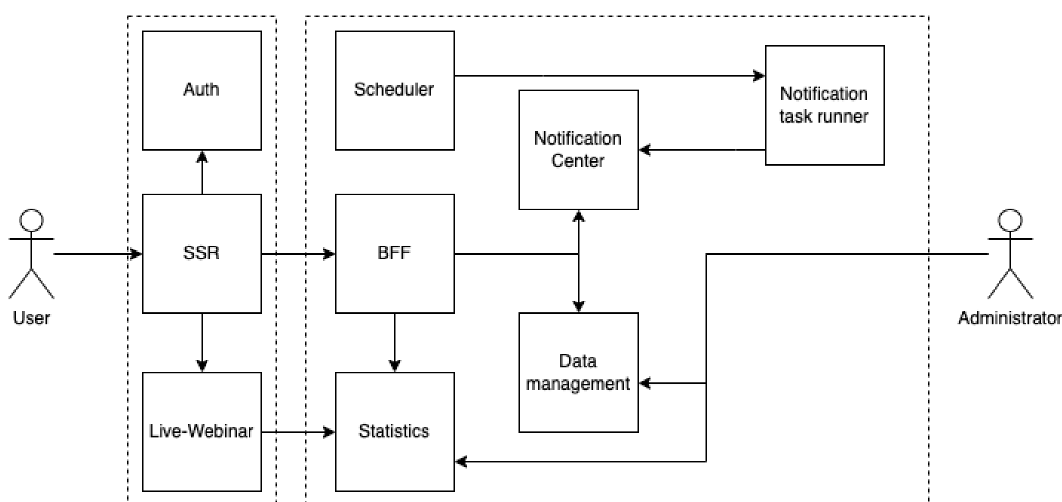

Figure 5: Final functionality decomposition into isolated services

3 DATA STORAGES

The way data is stored is not only crucial for business, but also determines the vector of scaling for the whole system. Picking the right solution is one the most important and difficult decisions.

There exist an enormous number of ways of storing data in microservice-based architecture. Decisions must be made almost on every layer of the system. Moreover, those decisions are not only vertical, but also horizontal, as there can be different approaches inside each domain. These issues vary from choosing a database for main storage, caches, session storage and queues. Even bigger concerns are external or internal location of database, sharding, replications and many others. As always, teams must find an optimal and reasoned solution on each layer of their systems, keeping in mind how it will affect other modules and services. Moreover, there are several patterns and all of them have huge advantages and drastic disadvantages. However, the chosen approach does not dictate to use it across the entire system. Microservices provide flexibility on a level of data storage as well, so developers can choose what to use for each separate requirement.

In this chapter two fundamental approaches will be presented, which will help to choose an appropriate data storage model for each aspect for a project this thesis is based on.

3.1 Shared database

Historically, a situation where having more than one SQL database was an appropriate structure, almost has never appeared, except sharding the information. Several databases cause additional business logic in the applications and cause latency as it always takes more time to concatenate, filter and prepare data outside of the database. Whenever companies invest resources into fragmentation of their hulking monoliths, they usually start with their old database, which is most likely run on in cluster mode with replicas set up.

Despite such approach may seem obsolete for microservices, it is still a valid pattern called Shared database and can be used for various types of applications.

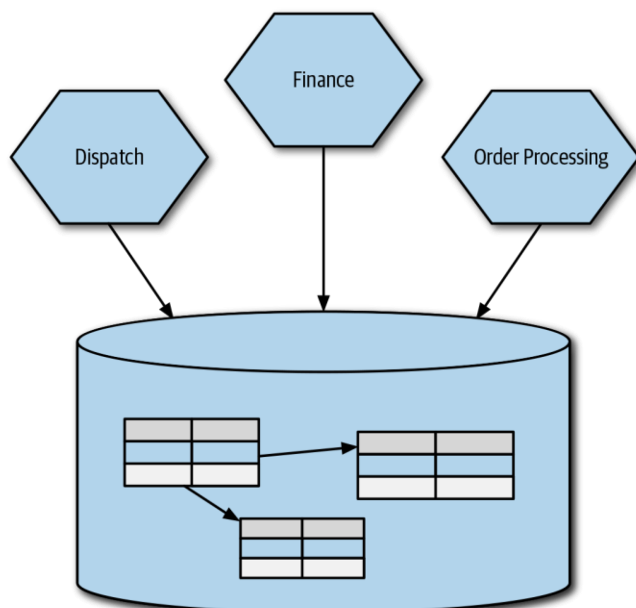

Figure 6: Illustration of multiple services using a single shared database [9]

This approach supports ACID [2] properties with local transactions as several services are allowed to access the single database with both writing and reading. “Shared database” pattern is easy to implement with SQL and NoSQL nowadays. However, it is important to keep in mind that NoSQL is not fully compliant with ACID without additional business logic.

To understand the reason why the “Shared database” pattern is used, the following example from the system we are building can be considered, where this pattern will be applied. There are several services which tend to access closely related data from the database. Apart from the functionalities of potentially several services (e.g., points management, webinars and articles management) which were united into one service, service called Auth also needs to access user data to compare password hashes in order to authenticate a person. Even though Auth service is placed into a different domain, there is no advantage to store user data across multiple locations. On the contrary, such an approach leads to additional logic to handle eventual consistency across services which in its order causes unnecessary errors.

Some queries must join data that is owned by multiple services. Establishing channels between services may be excessive if the amount of that information is not significant, as it would apply additional cognitive complexity to the system. Moreover, other business transactions sometimes must update data owned by multiple services, which makes it an even more challenging task. One can view the “Shared database” pattern as a pattern of non-reactive global store for all services, however with modern DBMS it is possible to

make them reactive. Another very beneficial use case of “Shared database” is compliance rules or any other enforcements that affect data globally. In the light of different innovations in personal data protection, such as GDPR, there are several restrictions for storing such data. One of restrictions is data retention time, which is very convenient to be implemented once on a database level.

Obviously, despite the convenience, which is offered by that approach, it has its own drawbacks:

- As is well known, microservices are small, maintainable, loosely coupled modules of the system. Their independence gives an opportunity to utilise resources in the most efficient way, giving teams more freedom and independence during the development process. In that light, a shared database can limit independence and be a bottleneck as the database scheme must be agreed on by all units. Moreover, in case of a need to remove a column, developers would end up spending time on a complex investigation on how to migrate the existing state of a system, which includes all services, to align with the new schema. NoSQL databases cover a set of such restrictions, but not all of them. Therefore, using the Shared database pattern, an architect must be cautious not to oversimplify data access.
- This approach works the best for a relatively small amount of data. With data amount growth, a database can primitively become a bottleneck for data if too many connections are opened, all threads remain busy all the time and locks being held on data accessed by multiple services simultaneously.
- “Shared database” pattern prevents developers from scaling that database horizontally. For instance, sharding needs to be thought through really carefully and will require business logic, because it is not possible to query and join data across distinct instances.

Luckily, these drawbacks do not affect our use case described above, so it is valid to use the “Shared database” pattern. Also, we can set up global rules for user related data.

3.2 Database per service

In order to achieve better scalability and flexibility, the general approach is always to split one big entity into smaller ones. Data storages for microservices are not an exception. Instead of having one big database, “Database per service” pattern can be applied, where each microservice will have a small, maintainable database. Although, there are several options on how to keep data, which belongs to a particular business logic, private:

- Private-tables-per-service – each service owns a set of tables that must be accessed only by that service.
- Schema-per-service – each service has a database schema that is private to that service.
- Database-server-per-service – each service has its own database server.

What should be chosen for our platform? Clearly, private tables have the lowest overhead and do not require additional logic, yet it does not ensure privacy of the data and does not force developers to stick to the pattern. Also, it does not separate schemes. Schema-per-service is a slightly better approach, because it gets clear who is the owner of a particular data segment. Nevertheless, as the purpose is to see the difference between the two approaches, we will focus on the third option with separate physical instances of databases. Also, private tables and schemas do not cure several drawbacks of "Shared database" and their usage lies in between two big patterns, which makes the architecting process vaguer and more unclear.

Database-server-per-service ensured on the physical layer is considered as a best practice and recommended for all systems as it continues the philosophy of loosely coupled modules. However, it is a much broader and more difficult way of organising data flow. Let us consider why "Database per service" is used and what can be a potential application for our case.

- As mentioned earlier, services must be loosely coupled in all possible ways, so developers would be able to create, modify and integrate them with ease. With one standalone database for each service, it comes by default as it inherits that property from the layer of services. Using this pattern, there is no need to synchronise database schemes or even database technologies across the system. While some services can use SQL data storage, others may have NoSQL or graph-based ones, for instance. If we look at Notification Center service, we can notice that there is no need to access a shared database. It should be a lightweight service with all related information being stored separately. That would decrease coupling and, in case of changing a schema, will not involve this service into the migration plan.
- As an addition to the previous point, smaller databases are easily scaled as database administrators should take care of only information which is owned by the single responsible service.
- Databases cannot become a bottleneck of the system. They work independently of each other. Moreover, locks are held merely by connections invoked by a single service. Furthermore, if monitoring says that database performance is poor due to overload, sharding and replications are easily implemented with no additional work

across different services. This is very handy for the statistics / analytics service that we introduced in the previous chapter. Statistics processing usually takes time and cannot be performed in real-time, so locking most of tables would drastically decrease performance of the entire system. Therefore, the database can be replicated into its copy with a mix of additional columns, which analysis will be performed on.

- This approach continues fault isolation philosophy as well, so if one service writes broken data, it does not affect other storages, which are now less coupled. For statistics service it is a crucial advantage since data analysis is complex by nature and may cause unwanted changes in the entire database.

Besides solving potential problems of monolith databases, it is worth mentioning other non-technical benefits brought by the "Database per service" approach, such as easy data understanding.

3.2.1 Drawbacks

Modular approach helps to avoid problems with scalability. As usual, it brings some drawbacks due to the arising complexity. Mistakes made during implementation of deep, underlying concepts are the costliest mistakes and each decision needs to be considered very carefully.

- One of the most common misjudgements in software engineering in general is an aspiration to implement all best practices proposed by the community. From time to time, best practice without any compromise would cost more resources than a working solution which would take a lot less effort. In the case of having dozens of databases, time spent on maintenance most likely prevails the value gained by introducing this approach. Situation may get worse if more than one or two different database types are used.

- Implementing business transactions that span multiple services is not straightforward. Distributed transactions are best to be avoided according to the CAP theorem. Moreover, many modern NoSQL databases don't support them, so they will require additional logic layers, which will be discussed shortly. Implementing queries that join data that is now in multiple databases is challenging as services form information barriers.

- Eventual consistency. Data may be inconsistent for a period of time while the transaction is spined across services. According to CAP theorem [1], this is

happening due to the impossibility of having more than two properties of data at the same time among its consistency, availability and partition tolerance. For most modern applications, data availability is a crucial property as it must be accessible at any point of time. Also, as we split data into several chunks, it is required to have a partition tolerance property. Therefore, consistency is not guaranteed by itself. However, with carefully planned design of architecture, eventual consistency can be achieved.

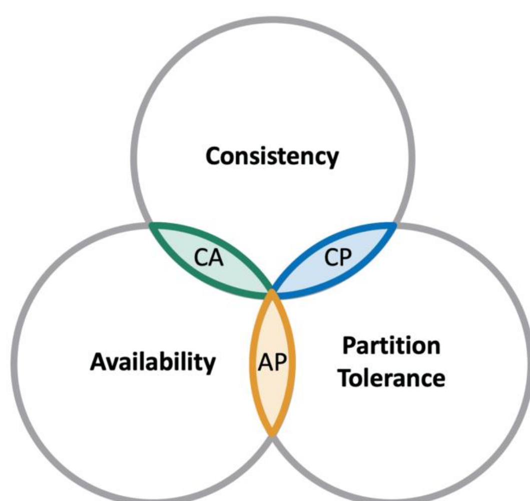

Figure 7: Illustration of CAP Theorem and properties which can be achieved simultaneously

To avoid drawbacks, there exist additional patterns to address them, such as “Saga” described in the previous chapter. Saga pattern can be a good measurement of microservice design: if the pattern is hard to implement, most likely bad choices were made in the designing phase.

3.3 Implementation of data storages

Summing up previous paragraphs, both approaches (Shared database and Database-per-service) have downsides and advantages which cannot be achieved by the opposite pattern. Each use case must be considered individually not only from a technical point of view, but also be dependent on business needs and opportunities. Even though it is difficult to find a perfect set of resolutions, such flexibility gives us a possibility to adapt to real life occasions.

While comparing, the conclusion derives that if databases are shared, it is a lot more convenient to query data which remains consistent all the time, but drawbacks are not

avoidable when application reaches a certain point of growth, while “Database per service” requires more effort. However, its issues are solvable with additional logic.

In reality, both “Shared database” and “Database per service” patterns are used in systems simultaneously. A key idea is to balance between modularity and convenience. Although perfect “Database per service” must be a reference point in big systems, it is practically impossible to reach. For instance, if modules (services) are defined by domain, several services are allowed to use one database, while other domains can access it only via API layer. This may help to ensure data consistency inside each domain.

“Shared database” pattern is a good approach for prototypes, small applications which are not going to grow or as a temporary solution to allow faster market entrance. The main goal is to utilise available resources most efficiently, which does not imply sticking to the only pattern.

As the traffic to the website, which this thesis is based on, cannot be called “High-load”, it would make sense to store most of the information which may be related into one physical database, as it will not create a bottleneck for the system. At the same time, it will allow to launch the platform for a public audience much quicker. Although, for part of information it makes sense to separate data into separate storages. For instance, statistics should also be separated as such unstructured data must be manually processed before revealing it to the client.

Additionally, there are other topics regarding data management to consider.

3.3.1 CQRS

CQRS (Command Query Responsibility Segregation) is a broad design pattern which aims to separate reading and writing into data storage. In the light of microservices, it helps to query related data from multiple locations. Moreover, it simplifies the complexity of business logic by separating concerns. To query data from multiple services, it is required to define a view database, which is a read-only replica that is designed to support that query. The application keeps the replica up to date by subscribing to domain events published by the service that owns the data. Developers need to be cautious about proper implementation of CQRS as it has several drawbacks such as increased complexity, potential code duplication and “eventually consistent” views.

In our system it is possible to use CQRS for several occasions [6]. Ideally, BFF service and potentially SSR service should be owned by one team, because both services’ goal is to

provide information and views to real users. At the same time, these services should not be responsible for any data manipulation, thus no database writing access should be provided. However, to decrease traffic between mentioned services and Data management service, it is possible to allow reading directly from the database. As was discussed before, sharing database schemas between services may lead to inflexibility, but database views can remain unchanged and be supported by owning service. Therefore, we can define a view in the database for different purposes client-side may have. These views will be accessed by BFF without requests to the Data management service directly. Since the flexibility microservice architecture provides, it can be a subject to change in the future if it ever becomes a bottleneck. Moreover, this approach is an application of the API Composition pattern defined in the previous chapter.

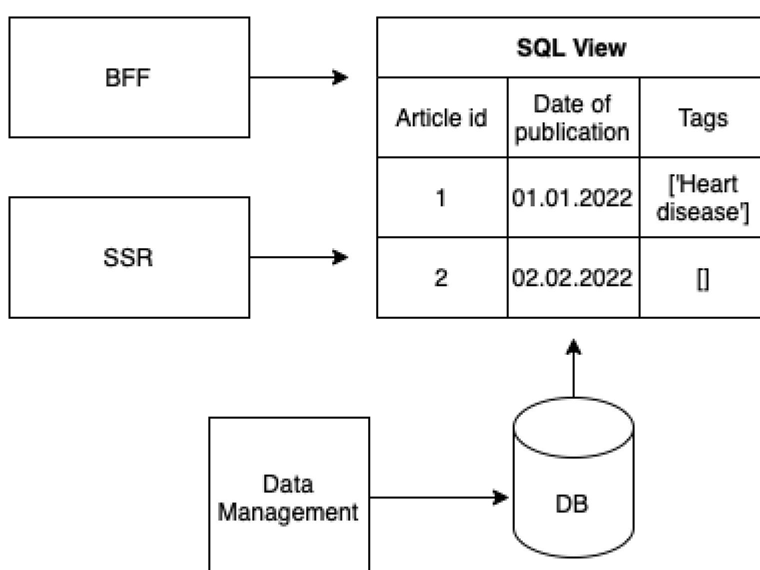

Figure 8: Communication between three service in the system using SQL View

3.3.2 Event sourcing

A lot of modern applications rely on events for various purposes. For example, as we mentioned earlier, a service in a Saga sequence atomically updates the database and publishes an event or message. Event Sourcing makes use of applications events.

Event Sourcing is a technique of representing the state by persisting state-changing events. Every time the business entity changes, the event is persisted in the event store.

As the name suggests, the event site is a database for events [6]. It can be SQL, NoSQL, or any other way that is suitable for the project. Moreover, the event store can act as a message broker. When an event is persisted, the event store delivers information to all

subscribers. Publishing an event is a single atomic operation. Therefore, it provides reliability and atomicity of database operation across microservices.

Furthermore, it creates a complete audit log. In case of any problem or bug, it is easy to research the state changes and eventually restore the valid state. Thus, debugging will be less complex. Additionally, event sourcing can avoid impedance mismatch between object-oriented and relational data. To sum up, event sourcing can be a great help in microservices architecture or any event-driven application.

In our system for webinar platform, there is a case which perfectly fits into this pattern. In Scheduler service, usage of Event Sourcing can be a great helper. Whenever the service generates an event about action which has to happen, it creates or updates a collection of records in data storage. Acting in such a way, the system will always know exactly at which stage the transaction is, even if the instance of service fails. If the instance shuts down, other instances can process a collection of events and rollback changes or continue the execution. This is achieved by atomicity of operation, as data storage acts as a single source of information.

On the other hand, services interested in events from Scheduler, namely Live-Webinar service or Task Generator service, may observe data storage and subscribe to those events as well. In that way, data storage would act as a message broker, which would also reduce infrastructural complexity.

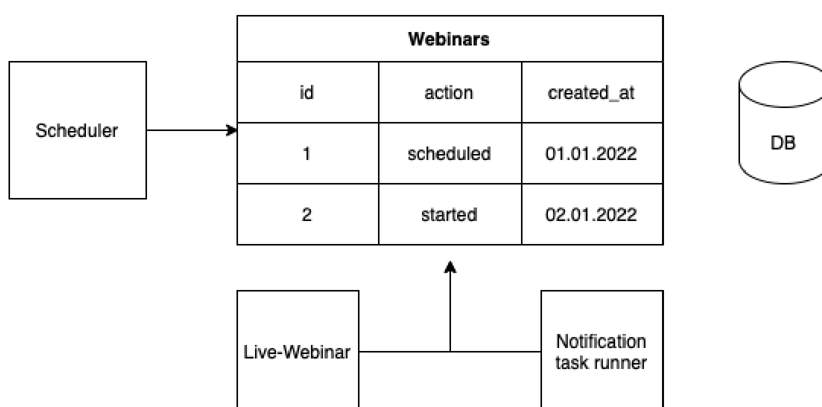

Figure 9: Communication between three service in the system using events stored in the database

3.4 Types of databases

Days when SQL was the only adequate option for any serious commercial project are over. NoSQL databases such as MongoDB and CouchDB, key-value databases like Redis, or even Graph oriented databases are used nowadays. Despite their differences, all of them

aim to solve SQL storage problems. This section will be focused on basic, ordinary NoSQL, as others have a particular use case, which are not applicable for the platform.

The biggest concern of choosing the appropriate type of database is scaling. In the real world, commercial applications must have an ability to adjust to business needs quickly. Therefore, despite SQL solutions are proven by time, they are not ideal for a huge pace of growth. Relational databases are a fantastic way to store and query data, which works best if data is placed together. By all benchmarks SQL DBMS such as PostgreSQL outstand any NoSQL solution, but they require a vertical scaling to function in the long-term. This contradicts the idea of small, maintainable storages which are more suitable for microservices.

We have mentioned CAP theorem before, which states that it is not possible to fulfil more than two guarantees out of the following: Consistency, Availability, Partition tolerance. That is a basis for choosing a database for the project. In case if data must always remain consistent whenever it is accessed, then SQL is still a preferable option. If real-time consistency is not crucial or data scheme is likely to be changed often (statistics as an example), then NoSQL is more suitable. Systems like MongoDB sacrifice consistency in favour of easier horizontal database scaling. Unlike SQL systems which implement ACID properties, NoSQL approach introduces BASE properties:

- Basic availability - every request is guaranteed to execute, either successfully or not.
- Soft state - state of the application may transform with time even without input to achieve consistency.
- Eventual consistency - data can be inconsistent for some time but will be consistent at some point of time.

So, in reality, both approaches can be used, as microservice-based architecture gives us such opportunity. Database type should be chosen not for the whole system, but for a portion of data inside it. For example, as main data storage for related data it was decided to use a single database as it is a convenient way to store and maintain it with low overhead. At the same time, it should not become an overhead because of several patterns and improvements stated in this chapter before. However, for unrelated services we may use any other solution.

3.5 Cache

All systems with time and growth experience an increase of latency due to inability to respond to all clients simultaneously, but scaling may be costly. For resourceful

calculations or database queries, it is possible to introduce caching solutions. Caching is usually viewed as a layered model.

First cache is stored on the client-side. It can be a browser or native application on any platform. This layer is appropriate for storing static images, scripts and metafiles. The second layer is Web Server which is almost unavoidable in production environments. This is more configurable and easier to invalidate storage. Usually, developers do not have to bother about implementation as modern Web Servers support caching automatically. The third and the last level (not taking DBMS into counting) is application layer or business logic. Developers must take care of caching themselves. Caching can be implemented inside instances, however with this approach each time a new version is deployed, or an instance is rerun after any error, cache would need to be "heated" from scratch. The best approach is to store calculations with in-memory databases such as Redis or Memcached. Both are NoSQL key-value solutions, which allow to easily store and retrieve data. It is fast, as data sits closer than on disk, and retrieving generates no additional logic. The only thing which must be taken care of is naming collisions. One way is to arrange a naming between developer teams which will prevent collisions.

4 COMMUNICATION BETWEEN SERVICES

After careful consideration on how physical and logical decomposition is implemented, new challenges arise: how do we connect all pieces into one distributed system, which will have reliable inner communication, secure endpoints for the outside world and which technologies to use to achieve it? We need to get back into the philosophy of microservices where the answer can be found: it depends on various factors, such as resource consumption growth, complexity and flexibility. Strategy for communication between services is not a separate topic to consider by itself, but it may influence component design of services.

To establish channels of communication, an architecture needs to adjust to several requirements:

- Reliable message exchange without data and event loss.
- A modern microservice-based application typically runs in a virtualized or containerized environment where the number of instances of a service and their locations changes dynamically. Required solution must target that issue.

In general, it is always a good idea to have the same approach and technologies throughout the system. However, since the power of microservices it is possible to deviate from this

assumption to pick a right tool for each feature. With that in mind, we can choose underlying technology with focus on logical decisions made about communication.

Let us consider what possibilities we have for actual implementation.

4.1 HTTP protocol

The most primitive, but still self-proven by time is HTTP protocol. It is a very well-known protocol which is used ubiquitously. Most web services use it to communicate with their clients (RESTful APIs, Web Sockets, FTP and others). There is nothing surprising about it since the protocol is reliable and extensible, supports caching, authentication, sessions and proxies. HTTP used with RESTful services may be a good solution for most of the cases, because there is a chance that its capabilities would be enough for particular needs.

As is said before, many applications across the internet use the HTTP protocol. Although it has several limitations in the sense of internal communication, there is nothing more suitable for external communication. Also, in the next chapter we will describe several components of typical server infrastructure. One of these components is a web server such as Nginx or Apache. Web server will allow to hide IP addresses and ports of services and will act as a proxy applying additional layers of security. Therefore, services must support HTTP, at least for those requests which are allowed to be made from the outside.

4.2 RPC

Main advantage of HTTP is that developers are really aware of it and its features. There are a lot of simple operations that may be performed using HTTP for inter-process communication. Despite its trustworthiness and credibility, HTTP may not be suitable for inter-process communication between microservices due to statelessness, dependability on microservice itself, need for constant serialisation and dynamic locations. Developers have to engineer a way around all downsides. One of potential solutions is usage of message brokers and queues, which we will get back to later. Also, as an alternative to RESTful APIs for message exchange inside the system, there exists Remote Procedure Invocation/Call (or RPI/RPC). The principle underneath is almost the same: module provides endpoints via which a client can invoke a function with or without response. The biggest difference is that the application does not have to keep any port open if services are deployed into the same cluster, since internal inter-process communication can be used. The system benefits from it in speed of message exchange and security. In case of services being run on distinct machines, custom protocols are usually used. The most famous representative of RPC libraries is gRPC - a multi-platform framework provided by Google.

According to widely cited tests [4] published by Ruwan Fernando, gRPC API connections are considerably faster than REST API connections. In fact, he reported that they are 7 to 10 times faster:

“gRPC is roughly 7 times faster than REST when receiving data & roughly 10 times faster than REST when sending data for this specific payload. This is mainly due to the tight packing of the Protocol Buffers and the use of HTTP/2 by gRPC.”

So, how do Protocol Buffers help to achieve high performance? Low-power low-bandwidth networks: gRPC’s use of serialised Protobuf messages offers light-weight messaging, greater efficiency, and speed for bandwidth-constrained, low-power networks (especially when compared to JSON). Moreover, gRPC offers bidirectional streams, which may come handy in terms of microservice-based architecture to synchronise their cooperation.

Unfortunately, gRPC is not widely adopted due to different reasons such as vendor dependency, necessity to learn and integrate new technology, but it is impossible to deny the potential of the technology. To sum up, communication can be implemented with HTTP, and it is a favourable tool for external endpoints, but HTTP can be used to make services collaborate between each other. However, gRPC is worthwhile with its required investment of time thankfully to transmission speed and other conveniences.

In the system for our Webinar platform, all external calls of API from a client will be made via HTTPS, however there is no need for loading a TLS certificate to each service, as Web Server will be responsible for security of communication. Although sometimes HTTP calls may suffice internal communication in certain cases, it cannot be used as a default communication model due to its drawbacks. Therefore, it would make sense to provide gRPC functionality to all services which will be involved in inner communication and will be required to serve data as soon as possible - in other words, queues will not fit their use case.

4.3 Message brokers and queues

Direct calls and usage of the “Server-client” pattern may be sufficient for simple applications as it covers communication in any infrastructure setup. Alas, in complex environments when each software error can cause financial losses businesses require fault-tolerance without data and event loss. With stateless remote signals it is impossible to ensure correctness of error handling. In case of service being down, we can lose an event or message due to the connectivity issues or internal errors in services themselves. Another

drawback is resource management since it is limited and hard to control. With growth, individual parts of the system will require additional resources, which will result in additional service instances. It indirectly involves another issue called Service Discovery. Whenever a request is made to a service, it should be routed to the best-suited instance. In our system this function will be taken care of by infrastructure tooling - namely, Kubernetes - which will be covered in the upcoming chapter. However, if no containerisation tool is involved, Service Discovery should have been implemented separately.

In the light of potential scaling demand, multiple destination services would need to be notified about certain events. So, the publisher service would have to notify them all manually with all related issues: higher network load, more complex transactional logic in case of partial failure which would need to be implemented in all participating listeners.

To solve those snags, Pattern "Messaging" was introduced as an opposite of all triggering procedures directly. There exists a whole family of MQ, which stands for Message Queue, protocols: STOMP, MQTT - MQ Telemetry Transport, AMQP - Advanced Message Queuing Protocol, etc. Messaging systems, or Message Brokers, are distributed systems aimed for providing transport level of communication between local and external processes. Their goal is to introduce an asynchronous model of inter process collaboration. Typically, the publisher-subscriber model is used, since it is easy to use, but more complex scenarios, such as RPC, can be also implemented with it. The most popular technologies are Apache Kafka and RabbitMQ, although there are several other options. Most of them are not suitable for all requirements and focus on particular use cases, leaving developer trade-offs to consider.

MQ systems address the following applications:

- Persistence - messages can be stored in-memory, on disk, be distributed and replicated across all instances connected to the network.
- Security - ability to set access policies and control rights for message consumption together with publishing.
- Deletion - determines the duration of message persistence.
- Filtration - some of the message brokers support message filtering to avoid consumption of unnecessary part messages, subscription only for partial topics by certain criteria.
- Guarantee of delivery - message brokers support setting up preferences for guarantee of message being delivered and/or processed.
- Routing - distribution of messages across services.

- Dosing of messages - number of messages being consumed at the same time: one by one or a portion with N messages.
- Order - user needs to have an ability to decide if order of incoming messages is important.

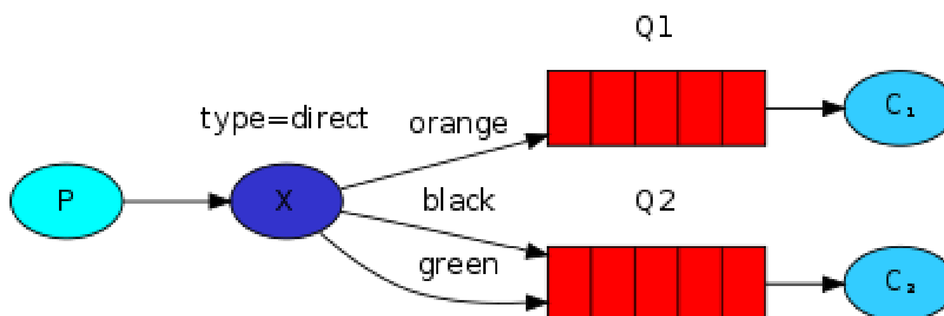

Figure 10: Message distribution between queues based on the routing key [10]

As we can observe, there are no direct channels of communication between services anymore. With such structure all goals of communication settings are achievable as now there is a strict, granular layer for message transportation. The system does not rely on services in terms of communication anymore and is able to handle exceptions on its own. For instance, if requested service is unavailable at the time of request and the event must be tracked by it, the externality of message queues allows us to store the message unless the service is restored. Yet another example of benefits which come out-of-the-box, in the case of an instance of service taking a task from the queue and failing to process it, lock on that task goes down and another instance is notified to grab it. Moreover, a separate reliable storage system gives an ability to monitor latency and load or to manage the state of the system.

It was decided to use one of Message Brokers in the system. The choice was to adapt RabbitMQ. Whenever there is a need to notify a set of services of an event, this communication channel should be used. For example, for email distribution there are 2 services involved: Notification task runner and Notification Center notified from Scheduler. This is a perfect scenario for message-based communication. One hour before any webinar, users should receive an email about it. Scheduler can put a message into a queue for Notification task runner, which will prepare separate messages for each user subscribed for a lecturer and pass these messages to Notification Center. This Center will send email by its means. However, even if any of these services are busy, data and commands are not lost and will be eventually sent out. Moreover, it is possible to configure Time-To-Live (TTL) of a message, so that if a message is in a queue for a specified amount of time, it should be destroyed. That way we can make sure email won't be sent out after the webinar has already started. Same communication can be used from Data

Management service, notifying about new articles, and to Statistics service from any service interested.

Also, as is mentioned before, RPC intercommunication style can be implemented as well with the help of two queues working together: the first queue is for the consumer, where client puts a message, the second queue is for client where he can retrieve a reply based on the ID of a message. However, there is no appropriate use case in our system.

As was said, the usage of message brokers automatically resolves the issue of service lookup as clients do not need to know the exact location of the requested service, as the broker takes care of it. Load balancing is done on the layer of communication as well (usually with a round-robin algorithm). It is possible to distribute a message to all instances without load balancing.

There are plenty of tools in the MQ family, but RabbitMQ was a choice. Every broker has its advantages, aiming at specific use cases. For instance, Apache Kafka's main purpose is to lower possible latency and increase throughput. However, Kafka does not provide a message delivery guarantee. Some may be dropped. Therefore, Kafka cannot be considered safe and be used as a default messaging solution. On the other side, brokers like RabbitMQ are not that efficient at processing stream, high-density data, but they provide more varied settings on different policies.

Message brokers are not only a reliable way to connect services, but also a resilient one as they are self-recovered and distributed. Data can be snapshotted to memory from time to time as well as replicated to all nodes connected to the cluster.

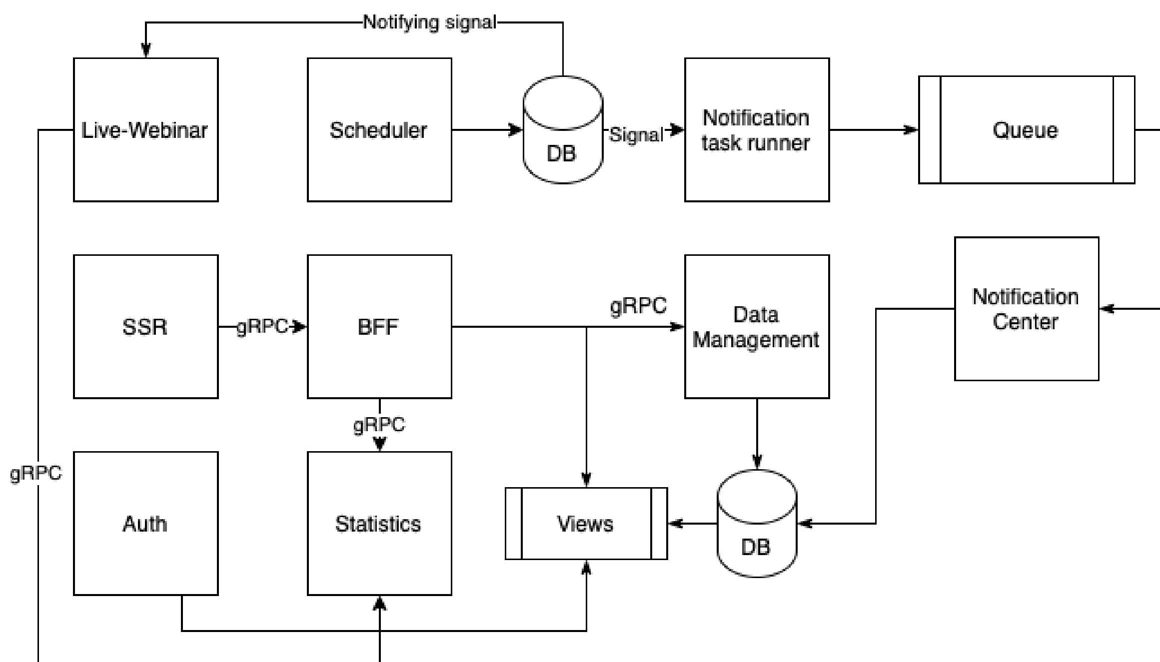

Figure 11: Final communication flow between services in the system

4.4 Databases as queues

As a compromise solution between direct API calls and message brokers, there is an option to construct a communication model using databases. Modern database systems have mechanisms of client notifications when an event occurs. Examples are a NOTIFY command in PostgreSQL or Tailable Cursors in MongoDB. Such an approach lacks an abstraction, so developers need to take care of some aspects themselves.

There is no particular pattern for this technique as actual implementation depends on the technology used: SQL, no-SQL or in-memory databases. However, usual procedure follows the steps:

- Service creates connection to the database.
- There are predefined physical locations in the database where each domain of events is isolated. Client puts an event to the database and Server is notified about data being inserted.
- Server retrieves the data and determines its state based on properties.

Unlike HTTP or RPC, “Databases as queues” pattern is event-based, so services do not know about each other. So, automatically it allows us to overcome all issues which are peculiar to HTTP and RPC. Also, there is no need for a Service Register. However, drawbacks are clear:

- Unnecessary constant open connection to the database.
- Routing is not flexible anymore. All services which are interested in part of events in any domain receive all of them, regardless of their status. Some of them will be automatically discarded if the event is in a state which is not appropriate for service which pulled the event.
- Need to take care of locking events, so several instances will not process the same event.

Despite all drawbacks, “Databases as queues” pattern can be a valid solution for various cases. That approach can be used with one or small group of services which pursue one goal.

In the chapter on System Design, it was mentioned that for the service called Scheduler the Orchestration-based “Saga” will be used in pair with Event Sourcing Pattern. This is a great example, where a database can act as a queue for an event, either between instances of one service or between several services.

5 SERVER INFRASTRUCTURE FOR MICROSERVICES

After a decision on architecture, databases and other important aspects of the system, the natural question may arise: how and where all these services will be run? In this chapter we will find the most suitable environment to run the system for the platform this thesis is based on.

Choosing an appropriate way of deployment steadily depends on the traffic generated by users and load upon each module of the system. It is crucial to remember that one of the main benefits of microservices is resource utilisation. Therefore, there is no need to scale one server vertically or blindly give out a dedicated server to each service. For heavily loaded components it is indeed possible to allocate resources, while small services which do not require a lot of resources can be glued together in some shared environment.

Due to the modular approach inherited by microservices, which allows us to build systems block by block and move them later, it is possible to start off with one approach keeping in mind the next one for a point of reaching scaling issues. So, the first starting point can be even a plain Linux dedicated server with services run directly on the operating system. It is a valid approach for small services. However, it is not suitable for services which are a subject of growth. It is totally fine to skip several steps and take a more sophisticated approach in advance.

Nowadays, the standard of running enterprise applications is containerisation. In order to utilise resources effectively, it is possible to create virtual environments. Plain virtualisation seems excessive and insufficient. Containers [3] can come in-handy. A container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another. One of the most popular container runtimes is Docker. Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. In larger application deployments, multiple containers may be deployed as one or more container clusters.

So, most of the time each service has an image, which can be replicated into several containers. These containers act as instances of the service. This approach strongly requires a service to be stateless or state must be taken care of, as was discussed in the previous chapters. What does containerisation give to us? At this point, it is easy to run a service in any environment without pre-installed dependencies, such as language interpreters. It is already solving our issue regarding creation of testing and production environments across hosts, as containers can be connected via virtual networks even though they are run on distinct hosts. For instance, in Docker it is feasible with an overlay network driver. So, it is possible to connect services into networks, isolated from each other. In practical terms, for us it means the following: running two instances of the same service in different environments do not interfere with each other's execution. Two identical databases can work at the same time. They still require two separate physical ports, however inside the network the port can stay the same, so other services inside this network will always know the exact location of a service.

Even though this approach resolves the issue of inefficient resource distribution, it seems tedious and barely automated. To address automation issues with growing numbers of services and containers, there exist various software frameworks called Container-Orchestration tools. One of them is Kubernetes. It serves as a higher level of container infrastructure architecture manager. It has its own API and schedulers, so that the solution is as configurable as needed. Kubernetes also acts as an auto load-balancer, not only rerouting requests into different containers, but also creating new ones if needed, without human intervention. Kubernetes solves for us really important problem, namely the need of Service Discovery, mentioned in Communication chapter, so that services are able to communicate without knowing the exact physical location of each other, as Kubernetes resolves all requests going inside the system automatically using just service names defined in config file, which usually has *.yml* extension. Also, Kubernetes, as a container manager, is able to recover from errors by restarting so-called pods - additional abstract environments for containers. Furthermore, these pods are a very convenient way to

configure both incoming and outcoming traffic. For example, some services may not require an open HTTP endpoint and communicate with the rest of the system via message queues or databases only. For such service, we can disable all incoming traffic making it secure by default. Another scenario is an opposite situation. Services, such as ones used to send web hooks, for instance, should be thoroughly and carefully designed, because it is not possible to predict what payload may be received. However, to reduce security concerns, we can forbid any traffic from a service to the rest of the system, so no matter what malicious response might be, the system is isolated.

Additionally, all services can be run exclusively in the cloud. Such services as AWS or Google Cloud provide a possibility to run services in any way possible, including so-called Lambda services - serverless computing. It helps to reduce real costs of infrastructure as such services are run only when the requests come. However, we do not have such features in our platform which would fit this approach, since it has a drawback, which is called “Cold start”. These services are natively compliant with Kubernetes, and it is easy to plug any system that a company has into a provider's infrastructure.

5.1 Web Server

Another standard tool that is very common nowadays is a standalone Web Server service. Web servers, such as NGinX or Apache, are almost always present in microservice-based systems. So far, it was not discussed how to route incoming requests from outside to our services. Services are run on different ports and different URLs. Hence, it is not possible to make direct requests from the client side as a location of a service can be changed dynamically. Web Servers can help a lot, working not only as a load balancer, but also as an API Gateway, catching all incoming requests and hiding real locations of services. Besides obvious benefits described before, Web Server also reduces complexity of services in terms of security. For example, instead of implementing DDOS mitigation inside every service, which will be inefficient, Web Server takes this functionality. Hence, each request is examined right before entering the system. Also, it is easy to serve static resources, such as media, using web servers.

For our system, we will not use Web Server as a separate standalone application. It is preferred to run a Web Server as a service inside the container. In that way, we can configure routing dynamically: whenever a request comes to a specific URL, Web Server will reroute it to a service, where this request is intended to get, by its container name. That setup gives us all benefits of using Web Server along with taking advantage of Kubernetes functionality.

It was decided to take NGinX as a Web Server because it is the most used and, therefore, maintained solution nowadays with a huge collection of different purpose plugins.

6 FINAL ARCHITECTURE

In previous chapters we managed to decompose the application into independent sets of functionalities using decomposition by subdomain. There are 9 isolated services, each of which has its own responsibility and communication channels with the outside world, even though most of outgoing communication is forwarded with Web Server built with NGinX. Besides that, communication between services inside the system was established as well.

As you could notice, each service has its own fundamental differences from the rest of the system. Each use case was transformed into a set of operations involving one or more services. However, on every step we were able to pick the right tool or pattern for a particular piece of functionality to achieve what microservices are bringing as benefits: better scalability, fault isolation and resource utilisation. Despite differences between ways each service is designed, we were able to connect them to act as a standalone application using different patterns and applying the most appropriate one for particular situations. Since there were a lot of decisions made on each layer of the development process from databases to infrastructure, let's reiterate over them to observe the whole system as one and verify its correctness from both engineering and management perspectives.

To start off, following core use cases were considered:

- Users should be able to register. During registration users should be able to upload confirmation of being a doctor or other medical worker.
- Users should be able to access and read research and news articles.
- Users should be able to leave comments for articles.
- Users should be able to access and watch webinars. During webinar, time which users spend on watching should be tracked. It will be a criterion of points reward.
- Users should be able to communicate between each other and with hosts during webinars.
- Administrators should be able to create articles and schedule webinars.
- Users should be able to receive notifications about upcoming webinars and new articles using different communication channels, such as Push notifications or email.
- Data analysts should be able to safely perform analysis over users' activity

There are more side use cases that potentially should be implemented in the future. Later in this chapter we will get back to them to see how easy it is to extend existing functionality with microservices flexibility.

6.1 Services and communication between them

After the designing phase there 9 services split into 2 layers: client-facing one, with services which clients directly interact with, and internal services, which no longer require to authenticate a user.

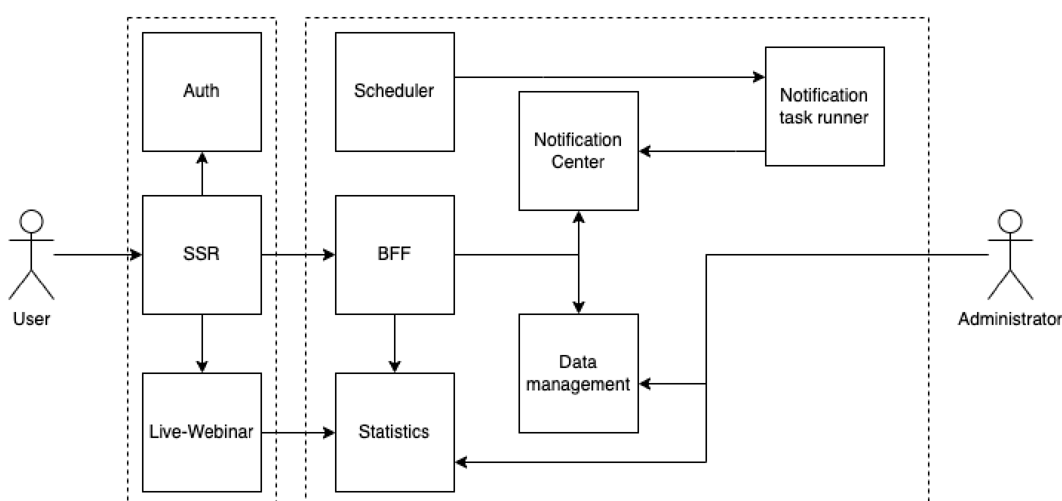

Figure 12: High-level view of the system and its interaction with real users

As you can see from the diagram, the set of services is:

“Auth”, or authentication service - a service responsible for authentication mechanism, where other services, which are for now represented only by SSR service, can access user permissions. This is an example of a rare service, which does not require any other service to function and can work fully independently from the rest of the system. The easiest and most scalable solution for authentication and authorisation in microservices-based architecture is JWT Tokens - an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA. With the private key shared between services, there is no need to always access “Auth” service to authorise a user, as a token can be decoded in any service.

SSR, or Server-Side Rendering service - a service responsible for partial page render to achieve better performance on the client side and to increase SEO metrics. For example, as webinars are a core of the platform, it is preferred to show the list of them to users as soon

as possible. Moreover, if titles and descriptions of webinars are rendered by a server, URLs are indexed higher in most search engines. As an additional advantage, SSR can perform several security measures such as permission checks or feature flag checks to hide information which should not be displayed for a particular group of users. SSR will be a client for:

- “Auth” service (mainly to retrieve users' permissions). As concluded in the Communication chapter, if operations must be performed immediately, the communication should be performed via HTTP, which is replaced by higher-level technology gRPC. Therefore, communication between SSR and Auth should be established using gRPC as if requests can’t be performed right after it arrives, there is no need to do it afterwards.
- Live-Webinar, to connect users to Web Sockets. Communication between SSR and Live-Webinar is mainly meant to check if a webinar is indeed active and, therefore, it is possible to reroute a client's request to that service. Hence, the same strategy - using gRPC to communicate - applies for this case as well.
- BFF, or Backend-For-Frontend, which acts as an API for client-side of the application. Same as before, if a client request fails, there is no need to perform a requested operation, hence no need to create a queue for that event. Therefore, again gRPC will be enough for communication.

BFF, or Backend-For-Frontend, is used to validate and proxy data from client-side to internal services and fetch aggregated data from internal services to the client side in order to lower the system load by reducing the number of requests for one render. Set of services BFF accesses includes:

- Data Management service via database level defined views in order to avoid coupling of schemas between services. With this implementation Data Management can store and modify schemas in any way unless a predefined view is updated properly.
- Statistics via gRPC
- Notification center via database view to retrieve unseen notifications

Data Management services acts and is used as a CRUD for main business critical information, such as webinars, articles and also points. In previous chapters it was decided to combine these functionalities to reduce infrastructural complexity which does not bring much benefit, such as fault isolation, resource consumption or domain separation. In case listed functionality will be expanding, it is possible to move it out to a separate service. Since it is mostly a CRUD service and the owner of entity schemas, this service’s outgoing communication will be conducted only with the database directly.

Scheduler - a service, which is responsible for spinning Cron jobs, which are related to different events: for example, notifying the system about webinar starting in one hour. This service has no communication besides database connection. Its job is to update the state of corresponding entities, and in its order, database notifies all interested consumers with a new state in a form of events (due to CQRS pattern). For now, Live-Webinar and Notification Task Runner services are the only signal consumers.

Live-Webinar - a service responsible for handling user activity during webinar - accept, validate, and distribute information such as comments between users, gather statistics passing it to internal statistics service, and check presence. This service will be using WebSocket protocol to handle this functionality. Apart from passing statistics to Statistics service, there will be only one communication channel which is database signal each time scheduler updates state of a webinar. In case the status of webinar is switched to “active”, Live-Webinar service has to prepare a virtual room where viewers could connect.

Notification Center and Notification Task runner - two services tightly working together, where Notification Task runner is necessary helper which was separated from main functionality of Notification Center to achieve fault-isolation, so that every email or notification of any notification distribution is a single message in the queue, which does not get lost in case of error during sending process. After the Notification Task Runner gets an update from DB, it generates a new message for each notification and puts it to the queue, which Notification Center consumes.

Statistics is a simple service which receives information about user activity and stores it in a separate database.

6.2 Data storage

In terms of data storages, it was decided to use the Shared Database pattern as it drastically reduces engineering complexity and, therefore, lowers time-to-market. Each service will have ownership over several entities which are related to its functionality. Other services should not be reading directly from not owned tables or schemas. However, for the Statistics service it is necessary to introduce a separate database. Data, this database stores, is not related to any other data in the system. Moreover, as it is stored separately, it’s really easy to replicate data to perform analysis. Such an approach ensures that data is not affected in any way. Nevertheless, in case of traffic going up and part of data is accessed a way more often, it is possible to move that data out to its own database to scale and shard independently.

From the perspective of team management, it is as flexible as the engineering perspective. Service ownership can be split by domains, layers or even each service on its own. The last option is meant for big services whose functionality cannot be split into smaller independent pieces. Live-Webinar service is a perfect example of that situation.

6.3 High level description of a flow

To verify correctness of the system, we can simulate real user average behaviour and see which actions are invoked in the system.

When a user's browser requests the URL of the system for the first time, it is served with a landing page provided by SSR services. After familiarisation with the website, users may decide to register. After filling required information, the browser performs an API call to BFF service, which will validate the data and invoke a call to Data Management to register the user. Now, when user information is present in the database, the browser is rerouted to "Auth" service to get authenticated by granting a JWT (JSON Web Token). Now SSR is able to perform authorisation by itself as JWT is sent with each request.

Registered medical workers are able to access the list of all articles available on the website as well as list of live and upcoming webinars. If users see an upcoming webinar they are interested in, they are able to click a button in the UI in order to subscribe for notifications. The process of a request is the same as registration: the client makes a request to BFF, which, in its turn, makes a request to the Data Management service adding a subscription to the database. One hour before the webinar starts, the Scheduler will ask Notification Task Runner to generate messages with user and webinar information and put them to the queue to trigger Notification Center to start email / notifications distributions. Subscribed users will receive reminders through preferred communication channels.

When the webinar is going live, it appears on the main page of the website for registered users. Opening it will ask the browser to connect with Live-Webinar via Web Sockets. Together with handling users' activity, Web Sockets allow us to calculate how much time users spent watching a particular webinar.

6.4 Flexibility

Last, but not the least, it is intuitively easy to add new features into the application. Let's consider two cases. Firstly, one can consider changing the code only in one service. It is the same as adding a feature into a small application not burdened with complex dependency cycles. For example, at some point it may become required to verify a person

using a registry of accredited medical staff. In this case, it would only take developers to add a small change in BFF service to perform that check via API call during a registration.

However, new features and functionality can have an impact for a whole application. It is possible that the platform will have several capabilities of accepting payments to give access to premium content. Instead of adding tables into the database and map it into objects in each service, building a new service responsible for billing is the best approach. Besides development of the service itself, integrating it into other services is as easy as adding a registry check in the first example. Services, like SSR, merely need to make a single request to a new billing service to get either permissions or a simple boolean identifying if a user is allowed to access the content.

Finally, this is the application's architecture from a higher perspective:

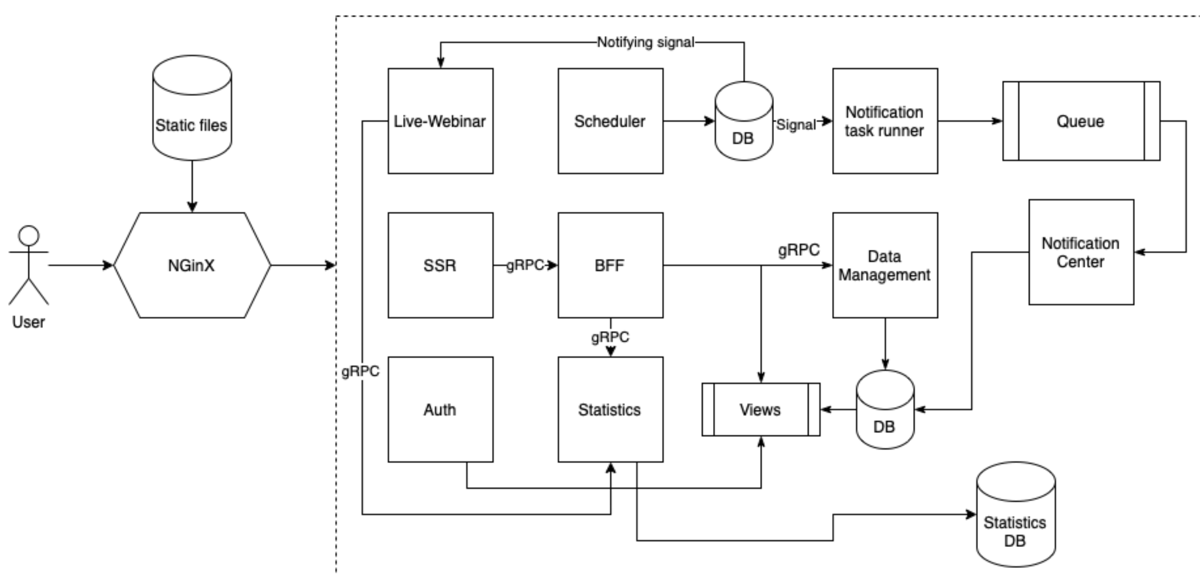

Figure 13: Illustration of the final architecture achieved after consideration of all aspects

Besides that, each service lifecycle will be controlled by Kubernetes, which will make sure all of them are in a healthy state with enough resources to handle any load.

7 CONCLUSIONS

In this thesis I have researched a way to organise an architecture of a project using microservices approach. It is demonstrated that microservices are a powerful tool to achieve modularity. During the designing phase, they added complexity which was not easy to handle. Each design pattern required another in order for the system to function properly. This "layered" structure of patterns, however, allows to make trade-offs. The key task of an architect is to find out what is really expected from the system and where it is

possible to come to stand. Therefore, using microservices requires full understanding of business needs, including current state and further scaling.

We have achieved critical fault isolation, so small errors will not shut the whole system down. An optimal resource utilisation plan was introduced as well together with ability and plan to scale the system out. The architecture is kept compact, but easily extended. Furthermore, due to high-grained modularity and loose coupling, it can be adjusted to business changes in straightforward fashion, as delivering new features either touches only a part of the entire system or nothing at all, only attaching new service to it.

I believe that microservices help the business side of projects to act more flexibly and increase its involvement into the development process. In today's market it is crucial to be able to adjust a product as well as perform scaling quickly.

Overall, microservices-based architecture adds an excessive complexity for certain applications, but solves issues, which cannot be solved in any other adequate way, for others. Together with containerisation and other modern virtualisation technologies, we have shown that any infrastructural setup can be used as an environment for a system. Before applying a microservices-based approach, developers must consider its benefits and complexity it brings and be prepared for demanding design decisions.

8 REFERENCES

[1] E. A. Brewer, A. Fox, *Harvest, yield, and scalable tolerant systems*, Proceedings of the Seventh Workshop on Hot Topics in Operating Systems, 1999, 174-178.

[2] J. Celko, Transactions and concurrency control, SQL for smarties: advanced SQL programming, third edition, Morgan Kaufmann, San Francisco, 2005, 720-722

[3] Docker, Inc. 2022, *Use containers to Build, Share and Run your applications*, accessed 11 July 2022, from https://www.docker.com/resources/what-container

[4] R. Fernando (2019, April 3*). Evaluating Performance of REST vs. gRPC*. Medium, accessed July 11, 2022, from https://medium.com/@EmperorRXF/evaluating-performance-of-rest-vs-grpc-1b8bdf0b22da

[5] Microsoft Corporation 2022, Strangler Fig pattern, accessed July 11, 2022, from https://docs.microsoft.com/en-us/azure/architecture/patterns/strangler-fig

[6] C. Richardson, Implementing queries in a microservice architecture, *Microservices Patterns*, Manning Publications, New York, 2019, 220-252.

[7] C. Richardson, Refactoring to microservices, *Microservices Patterns*, Manning Publications, New York, 2019, 428-472

[8] C. Richardson. Who is using microservices, accessed July 11, 2022, from https://microservices.io/articles/whoisusingmicroservices.html

[9] Sam Newman, Decomposing the Database, Monolith to Microservices, O'Reilly Media, Sebastopol, 2019, 125-206

[10] VMware, Inc., RabbitMQ tutorials: Routing, accessed July 11, 2022, from https://www.rabbitmq.com/tutorials/tutorial-four-python.html

[11] Wikipedia contributors, 2022, June 4, Service-oriented architecture, Wikipedia, accessed July 11, 2022, from https://en.wikipedia.org/wiki/Service-oriented_architecture