Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Scaling databases in microservices-based applications

Mikhail Nalutka

Supervisor: Ing. Jiří Šebek
Field of study: Otevřená informatika
Subfield: Software
May 2024

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Nalutka**	Jméno: **Mikhail**	Osobní číslo: **499053**
Fakulta/ústav:	**Fakulta elektrotechnická**		
Zadávající katedra/ústav:	**Katedra počítačů**		
Studijní program:	**Otevřená informatika**		
Specializace:	**Software**		

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Škálování databází v aplikaci založených na microservice architektuře

Název bakalářské práce anglicky:

Scaling databases in an application based on a microservice architecture

Pokyny pro vypracování:

V současné době jsou mikroslužby skloňované jako nejlepšípřístup pro psaní komplexních aplikací a vůbec realizaci podnikových
architektur. Je důležité, aby aplikace byla dostupná i při vysokém zatížení.
Jedním ze způsobů, jak zajistit větší odolnost aplikace vůči zátěži, je škálování databáze.
Cíle této práce jsou:
1) provést rešerši používaných patternů a principů pro škálování databází
2) provést analýzu, v jakých situacích je vhodné použít konkretní patterny
3) vytvořit knihovnu, která obsahuje vzorové implementace patternů mikroslužeb pro škálování databází
4) vytvořit vzorovou aplikaci, která demonstruje účinnost zvolených design patternů
5) provést výkonnostní testy aplikaci

Seznam doporučené literatury:

1) Antonio Messina, Riccardo Rizzo, Pietro Storniolo, Alfonso Urso. "A Simplified Database Pattern for the Microservice Architecture"
2) Gastón Márquez, Mónica M. Villegas, Hernán Astudillo. "A pattern language for scalable microservices-based systems"
3) Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, Marcos Kalinowski. "Data Management in Microservices: State of the Practice, Challenges, and Research Directions"

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Jiří Šebek kabinet výuky informatiky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **15.02.2024** Termín odevzdání bakalářské práce: **24.05.2024**

Platnost zadání bakalářské práce: **21.09.2025**

Ing. Jiří Šebek podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.	
Datum převzetí zadání	Podpis studenta

Acknowledgements

I would like to thank my supervisor, Ing. Jiří Šebek for his guidance and consultations throughout the creation of this bachelor thesis.

Declaration

I hereby declare that I have completed this thesis independently. All sources of information used in this work have been cited and are included in the list of used literature.

In Prague, 23. May 2024

Abstract

The goal of this bachelor thesis is to research methods for database scaling in microservice applications and demonstrate some of them on a demo application.

For this purpose, different methods for database scaling were presented and compared. After that, a demo microservice application was created in order to demonstrate the effects of the chosen scaling techniques. Finally, a performance test was done between the application which was not scaled and the scaled application to see if database scaling helped the application to better handle the load.

Keywords: Database scaling, microservice application, Docker, Kubernetes, Locust

Supervisor: Ing. Jiří Šebek

Abstrakt

Cílem této bakalářské práci je prozkoumat metody pro škálování databází v mikroservisních aplikacích a některé z nich pak demonstrovat na vzorové aplikaci.

Za tímto účelem byly představeny a porovnány různé metody škálování databáze. Poté byla vytvořena ukázková mikroservisní aplikace, na které byly demonstrovany účinky zvolených technik škálování. Nakonec byl proveden zatěžový test mezi aplikací, která nebyla škálována, a škálovanou aplikací, aby se zjistilo, zda škálování databáze pomohlo aplikaci lépe zvládat zátěž.

Klíčová slova: Škálování databází, mikroservisní aplikace, Docker, Kubernetes, Locust

Překlad názvu: Škálování databází v aplikaci založených na microservice architektuře

Contents

Figures

Tables

Chapter 1

Introduction

When the business is growing and the number of users accessing its products is increasing, it is very important to make sure that the service used by customers can handle this increased load. Not being able to serve the increased amount of requests leads to the application being unstable and even unavailable, which means a loss in reputation and profit. Even a couple of minutes of downtime can cost companies hundreds of thousands of dollars[8]. This is why it is important to promptly scale an application. Notably, the database scalability deserves special attention, as a database can become a bottleneck for the whole application.

Lately, more and more companies migrate their applications from monolithic architecture to microservices. [9] It is understandable, considering all the benefits, for example the ability to split the load between smaller services. Such approach allows to apply different scaling methods to different services. This leads to an overall more stable application, which can handle more user requests.

The goal of this thesis is to compare different database scaling methods and apply them on a microservice-based application. To achieve this, we will research existing methods and techniques used for scaling databases. After that, an example microservice application will be created in order to demonstrate the use of chosen scaling methods on it. After that, performance test will be done to see if the database scaling improved application's performance and throughput.

Chapter 2
Research

This chapter gives theoretical introduction to database scaling methods, microservice architecture, containerization and orchestration of applications.

2.1 Database scalability

Database scalability is the ability to expand or contract the capacity of system resources in order to support the changing usage of your application[10]. By increasing its availability, scaling allows the application to handle the increasing demand.

2.1.1 Types of database scaling

Where are two main ways how to scale a database. These are vertical and horizontal scaling.

Vertical scaling

Vertical scaling refers to increasing the processing power of a single server or cluster[10]. By having a more powerful server, application can handle more load. However, it is not possible to keep increasing server's resources forever. Eventually, a limit of maximum resources will be met. At that point, one needs to consider another scaling method.

Horizontal scaling

Horizontal scaling entails adding more machines to further distribute the load of the database[11]. Unlike vertical scaling, horizontal scaling can be performed over and over again, since we are not limited by a single server's limits. One problem of a horizontal scaling is that it is often more complicated to manage it.

2.2 Database horizontal scaling methods

There are two main techniques to scale a database: replication and data partitioning

2.2.1 Replication

Replication refers to creating copies of a database or database node[10]. Such a copy is called replica. Replicas contain data from the primary node. Data updates in the primary node are automatically reflected in all its replicas. This approach increases fault tolerance of a system. Even if one of the nodes becomes unavailable, other nodes can still respond to the requests. Replication can also be used for scaling. By having multiple database nodes, client requests can be distributed among them. That way, you can process much more requests faster.

There are two main methods how to replicate databases. That is Master-Slave replication and Master-Master replication.

Master-Slave replication

Here the primary node (master) is the only node that can process write requests, while the remaining replica nodes (slaves) are read-only. When the primary node goes down, one of the replica nodes is promoted to master.

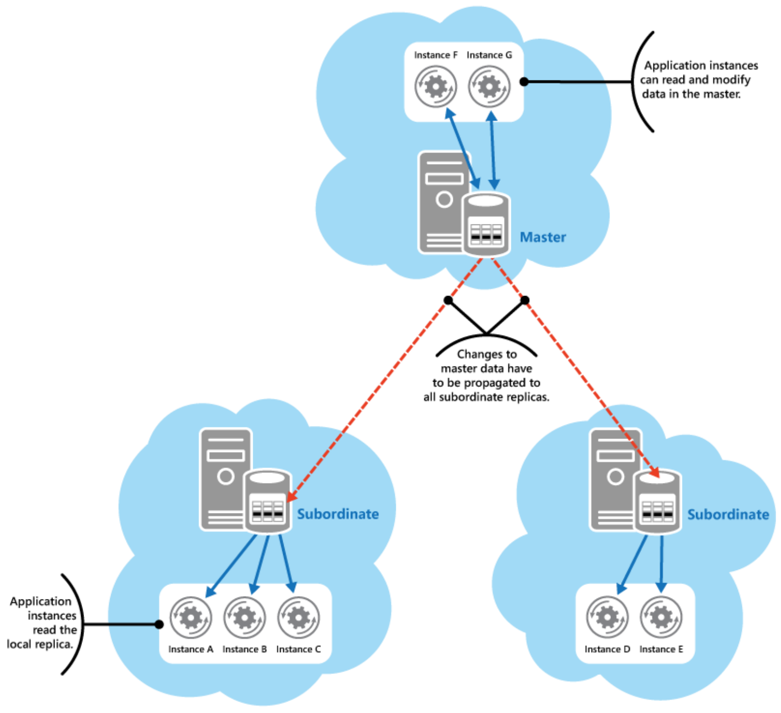

Instance F
Instance G
Application instances can read and modify data in the master.
Master
Changes to master data have to be propagated to all subordinate replicas.
Subordinate
Subordinate
Application instances read the local replica.
Instance A
Instance B
Instance C
Instance D
Instance E

Figure 2.1: Master-Slave replication (source: [1])

Master-Slave replication works best for read-focused data workloads, since we can distribute read requests among replicas. However, it is not suitable for write-heavy workloads, as each write must be copied to every replica.

Master-Master replication

In this replication, each replica node is considered to be master, meaning that each replica is responsible for write and read requests. Since multiple nodes can process write operations, concurrent writes to the same data can lead to conflicts.

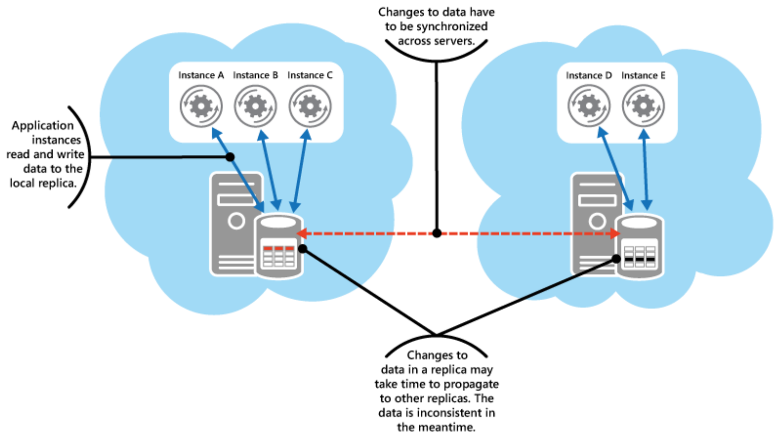

Changes to data have to be synchronized across servers.
Instance A
Instance B
Instance C
Instance D
Instance E
Application instances read and write data to the local replica.
Changes to data in a replica may take time to propagate to other replicas. The data is inconsistent in the meantime.

Figure 2.2: Master-Master replication (source: [1])

Unlike Master-Slave replication, Master-Master replication is great for write-heavy data workloads, since write requests are not processed by just one node.

2.2.2 Partitions

Partitioning is a method of dividing a large amount of data in one table into smaller unique data stores called partitions. Data is distributed among partitions based on a chosen column or a set of columns known as a partition key. Partitioning can be used for scaling since we can make each database node hold a unique subset of data.

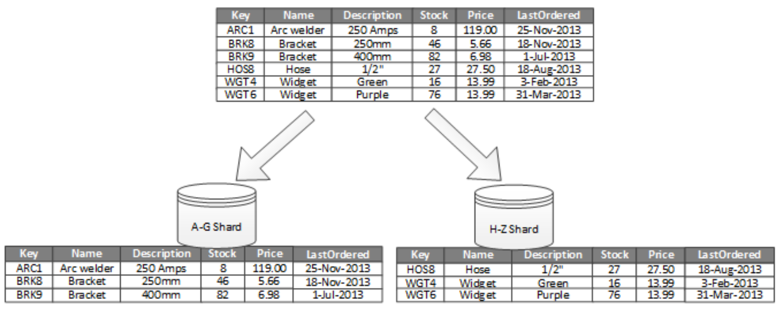
Moreover, partitioned tables get access to partition pruning, a query optimization technique that can be used for improving performance on such tables. When we search data belonging to the specific partition, the query planner is able to ignore other partitions and scan only the relevant one, thus speeding up the data retrieval.

Partitioning is also useful for managing historical data. Let's say that your application has to save user's data for one year after their accounts are deleted. For this purpose, you could store this data in the separate partition. After than, you can easily dispose of the partition by dropping it, which is much more efficient than having to bulk delete data from the main table.

There are two main approaches to data partitioning: horizontal partitioning and vertical partitioning.

Horizontal partitioning

Horizontal partitioning is distinguished by each partition having the same table schema but holding a unique subset of rows from the partitioned table. In other words, we split the table by rows, where different partitions contain different records.



Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013

Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013

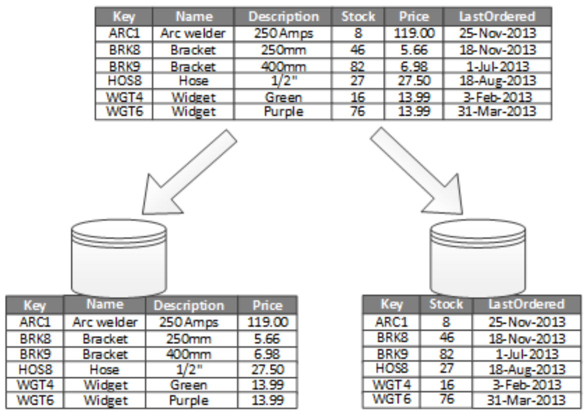
Key	Name	Description	Stock	Price	LastOrdered
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013

Figure 2.3: Horizontally partitioning data based on a sharding key (source: [2])

These partitions can be placed not only on a single server, but also on multiple servers, thus distributing the load among multiple machines. This approach is called sharding, where each individual partition is a shard.

Vertical partitioning

Vertical partitioning is different in a way that each partition holds a subset of columns from the partitioned table. This means that partitions have different schemas but the records are the same. The most common strategy to distribute the data is to put the most frequently accessed columns in a single partition or a small number of partitions.



Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013

Key	Name	Description	Price
ARC1	Arc welder	250 Amps	119.00
BRK8	Bracket	250mm	5.66
BRK9	Bracket	400mm	6.98
HOS8	Hose	1/2"	27.50
WGT4	Widget	Green	13.99
WGT6	Widget	Purple	13.99

Key	Stock	LastOrdered
ARC1	8	25-Nov-2013
BRK8	46	18-Nov-2013
BRK9	82	1-Jul-2013
HOS8	27	18-Aug-2013
WGT4	16	3-Feb-2013
WGT6	76	31-Mar-2013

Figure 2.4: Vertically partitioning data (source: [2])

2.2.3 Database optimization techniques

Besides scaling, one can also significantly improve the database's performance by performing database optimization. Common methods for this are indices and materialized views.

Indices

Usage of indices is a common technique to increase database performance. An index on a table can considerably speed up the data retrieval for the frequent queries at the cost of increasing system's overhead. Proper indices can also be used to significantly speed up table JOINs if the index is defined on the JOIN column.

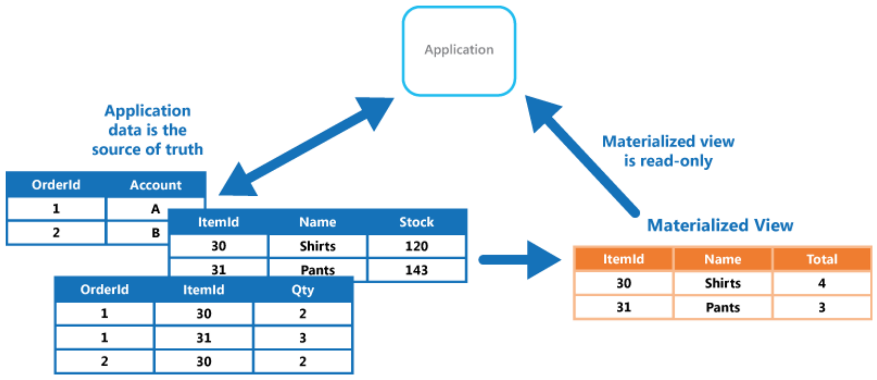
However, indices should be used cautiously, since poorly chosen indices can even decrease performance. When designing an index, one must carefully analyze application's frequent read operations and consider the following trade offs:

- Indices take up storage space in the database, since the database server has to maintain each index's structure alongside the table data. The amount of space that is required depends on the table's size and the number of columns in the index.

- Indices can slow down write operations, as each modification to the indexed column requires the update of an index itself. This is why write-heavy tables usually receive a performance hit from indices.

Materialized view

Materialized view is a type of view that saves query data to a table instead of having to compute data each time the view is queried. Since materialized view does not calculate data each time it is queried, the data it contains can quickly become stale. This is why it is important to periodically refresh materialized views with fresh data.


Application
Application data is the source of truth
Materialized view is read-only
Materialized View

OrderId	Account
1	A
2	B

ItemId	Name	Stock
30	Shirts	120
31	Pants	143

OrderId	ItemId	Qty
1	30	2
1	31	3
2	30	2

ItemId	Name	Total
30	Shirts	4
31	Pants	3

Figure 2.5: The Materialized view pattern (source: [3])

Materialized views are used to boost read performance for complex queries that are executed often because the query cost is paid only when the materialized view is created or refreshed. Because materialized views are represented as physical tables, they can also benefit from the use of indices, making data retrieval even faster. And since the materialized views are not updated often, the indices will not hinder write performance as much.

2.3 Microservice architecture

A microservices architecture is a software architecture which consists of a collection of small, autonomous services. Each service is self-contained and should implement a single business capability within a bounded context. A bounded context is a natural division within a business and provides an explicit boundary within which a domain model exists. [4]

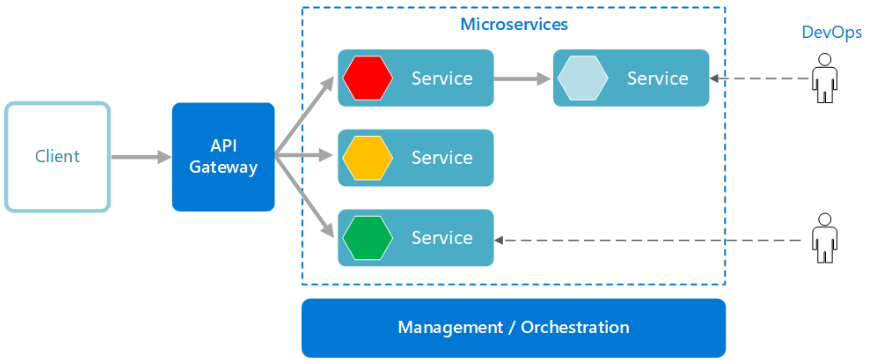


Figure 2.6: Example of a microservice architecture (source: [4])

Each microservice should have its own data store and other services don't have access to it. This leads to more independent and loosely coupled services, which allows for independent development and scaling.

2.3.1 Advantages of microservice architecture over monolithic architecture

Splitting a monolithic application into a set of small services brings a lot of benefits. Such small services are better suited for scaling. Often, only some parts of the application experience high load. With microservices, we can scale these parts independently according to the load they are getting.

Another benefit of the microservice architecture is the increased availability. Even if some service becomes unavailable, other services will still work and the application will still be able to serve most of the requests.

2.3.2 Disadvantages of microservice architecture over monolithic architecture

Microservice architecture also has some drawbacks. Such architecture on the whole is more complex, as it introduces a lot of moving parts. As the applications contains a lot of smaller services, they also have to communicate with each other. This can lead to increased latency and network load.

Moreover, with each microservice having its own data store, it can be a challenge to maintain a data consistency across the application.

2.3.3 Communication between microservices

In order to complete a user request, a lot of small services often have to work together. This is why implementing efficient and robust interservice communication is key. Communication can be either synchronous or asynchronous.

- Synchronous communication In this type of communication, a service calls an API that another service exposes, using a protocol such as HTTP or gRPC[12]. It is called synchronous because the caller needs to get a response to continue executing its task.
- Asynchronous communication In this type of communication, a service sends message to the message queue without waiting for a response, and one or more services process the message asynchronously. [12]

Both types of communication have their benefits and drawbacks. Synchronous communication is easier to implement, but it can create bottlenecks since services can not process another requests while waiting for a response.

Asynchronous communication, on the other hand, provides better decoupling and increased performance, since the service that sends a message does not wait for an immediate response. However, asynchronous communication is much more complex to manage.

2.4 Containerization

Containerization is the process of packaging the application together with all the dependencies which are required to run the application. Such package is called a container. One of the main benefit of containerizing an application is that they are highly portable, meaning that these containerized applications can run without modifications on any environment that supports containers.

Since containers provide an isolated environment for application to run on, they fit well with a microservice architecture. Each service can run inside it's own container, supporting loose coupling and resilience.

2.4.1 Docker

Docker is an open-source platform that is used for creating and managing application containers. It is by the far the most popular and widely used containerization tool.[13]

Docker architecture

Users can interact with Docker by using commands from the Docker API to the Docker client. Docker client then sends these commands through the REST API to the Docker daemon, which is responsible for managing Docker objects such as images and containers.[5]

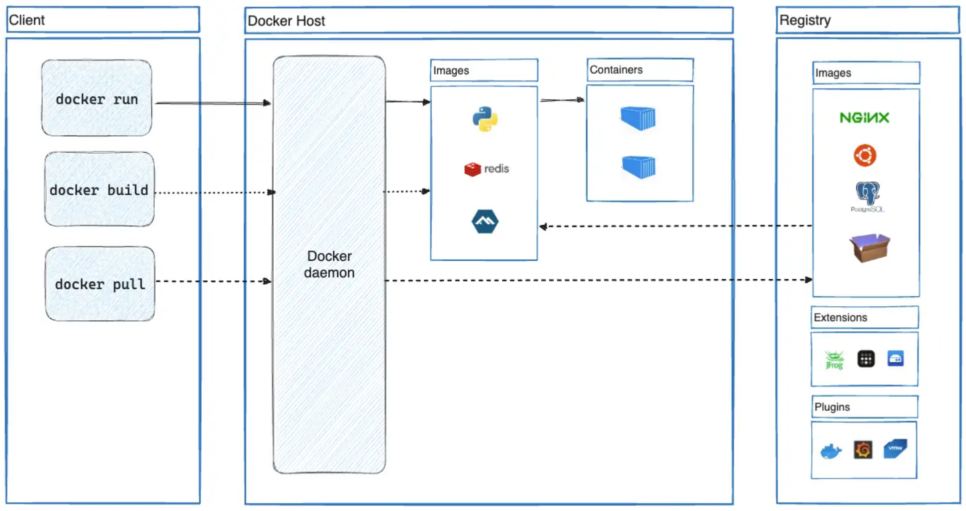


Figure 2.7: Docker architecture (source: [5])

Docker objects

- **Docker Images** - An image is a read-only template with instructions for creating a container.[5] Images are created from a Dockerfile, which describes the steps needed to create the image. Another option is to use the already existing image from the Docker registry. Docker offers it's own registry called Docker Hub, which is the world's largest repository of Docker images. [14] It is free to use for all public repositories, but you have to pay if you want to set up a private repository.
- **Docker Containers** - A container is the runnable instance of an Docker Image. Users can manage their containers through the Docker API.[5]

2.5 Orchestration

Even though the containers itself are lightweight, managing them at the larger scale can be challenging. This is where orchestration tools come into play. Orchestration automates the provisioning, deployment, networking, scaling, availability, and lifecycle management of containers.[15] The only thing developer needs to do is write a configuration file, and the orchestration tool does the rest. This configuration file usually defines container images and their locations, declares how the containers are connected to each other and the amount of resources that each container can use.

2.5.1 Kubernetes

Kubernetes is an open source platform for automating deployment, scaling, and management of containerized applications. It can also provide some features, such as automatic load balancing and distributing the network traffic between containers.[16]

Kubernetes architecture

When you deploy Kubernetes, you get an environment called cluster. These clusters consist of nodes, which are essentially computing units that deploy, run and manage containers. Each cluster has a master node known as a control plane, which manages the worker nodes. Every working node has a software agent called Kubelet that receives and executes orders from the master node. [17]

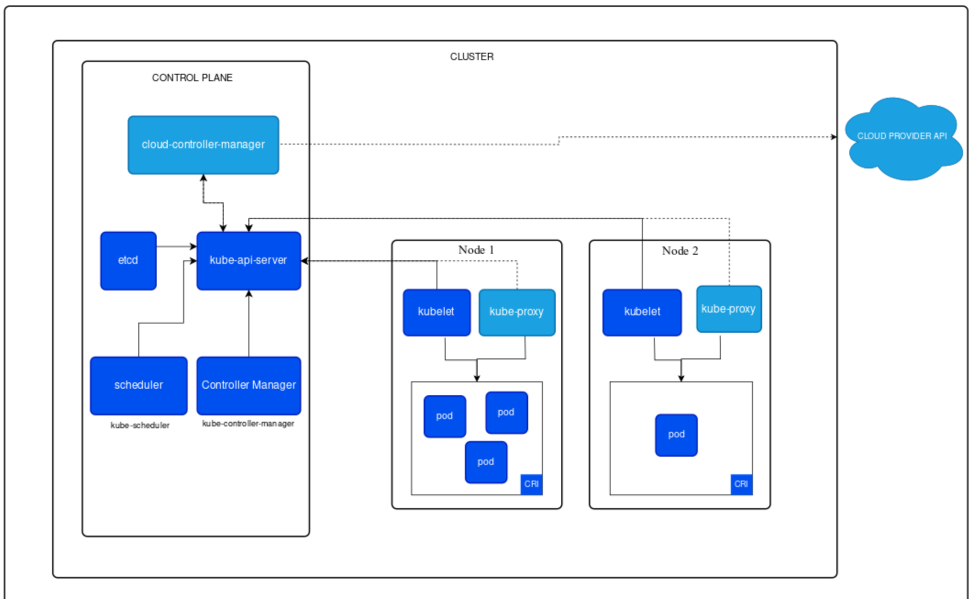


Figure 2.8: Kubernetes cluster architecture (source: [6])

A group of one or more containers makes a pod. In Kubernetes, it is the smallest deployable unit of computing. All containers that are a part of the same pod always share storage and resources. Pods are created by specifying Kubernetes workload resources.[18]

Kubernetes workload resources

- **Deployment** - A deployment is used to manage the deployment and scaling of a set of pods. It defines a desired state for a set of identical pods, known as replicas.[19]

- **StatefulSet** - Like deployment, StatefulSet manages the deployment and scaling of a set of pods. The difference is that StatefulSet does this in a predictable and ordered manner. It also provides stable network identities and stable storage for each pod.[20]
- **ReplicaSet** - ReplicaSet is responsible for maintaining a defined number of pod replicas. For this, it can create and delete pods to reach the required replica count.[21]
- **Service** - A service defines a set of pods and the way how to access them.[22]
- **Ingress** - Ingres is an API object that is used to provide external access to the services. In order to function, it needs an Ingress controller.[23]
- **Secret** - In order to store sensitive data like passwords outside of the application's code, a secret object is used. Other pods can then access the keys stored in the secret object.[24]

2.6 Conclusion

In this chapter we looked at different database scaling methods, described the microservice architecture and explained containerization and orchestration. In the next chapters, a demo microservice application will be designed, implemented and then deployed. This application will demonstrate the effects of database replication, partitioning, materialized views and indices.

Chapter 3
Analysis

This chapter marks the beginning of the thesis's practical part by describing the demo application and identifying its functional requirements.

3.1 Demo application

To showcase database scaling we need a demo application. For this purpose, a university information system was chosen. Such a system for a typical university can be used by tens of thousands of people [25], so it is a good candidate for optimization.

University Information System allows teachers and students to digitize and manage their university related data more easily. It will allow students to manage their study plan, create timetables, register for courses and so on. For teachers it makes it easier to manage the courses they teach.

3.2 Functional requirements

1. System will automatically create accounts for all users
2. System will automatically delete accounts for users who left the university
3. System will automatically create timetables for all users each semester
4. System will allow users to modify their account's information
5. System will allow users to browse available courses
6. System will allow users to browse available exam dates
7. System will allow users to browse available thesis topics
8. System will allow users to create a course
9. System will allow users to modify a course
10. System will allow users to delete a course
11. System will allow users to create their timetables

12. System will allow users to create a class
13. System will allow users to modify a class
14. System will allow users to delete a class
15. System will allow users to register for a class
16. System will allow users to deregister from a class
17. System will allow users to register for the exam date
18. System will allow users to deregister from the exam date
19. System will allow users to create the exam date
20. System will allow users to modify the exam date
21. System will allow users to delete the exam date
22. System will allow users to register for a thesis topic
23. System will allow users to deregister from a thesis topic
24. System will allow users to create a thesis topic
25. System will allow users to delete a thesis topic

In the figure 3.1 we can see a use case diagram for the university information system. There are two main roles, teacher and student. For the automatic creation and deletion of user accounts, a system actor was introduced.

Both teacher and student have some shared functionality, like creating and modifying timetables, modifying accounts and browsing courses. Users will be able to modify their personal data such as mobile number, address and bank account number.

Students will be able to choose a thesis topic from the list of all available topics, which are added to the system by the teachers. Exam dates and courses follow the same logic. After passing an exam, the teacher records the grade for that subject in the system and the course is marked as completed.

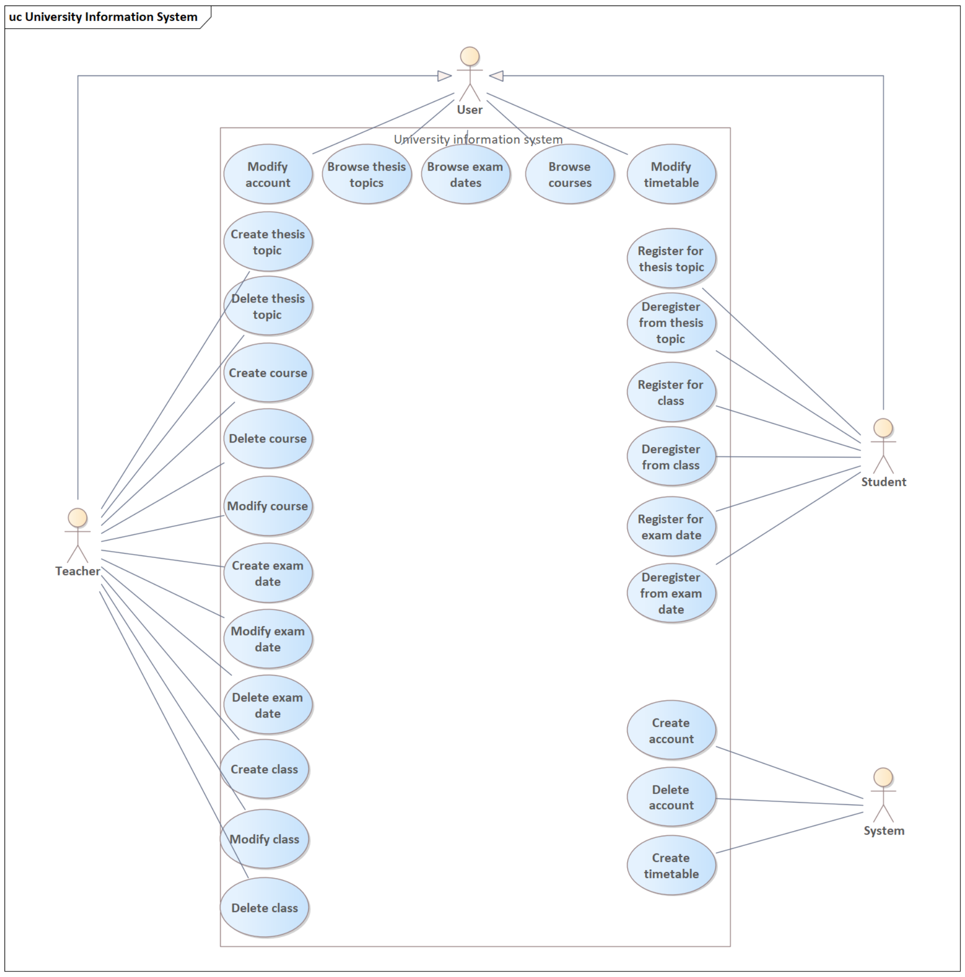


Figure 3.1: Use case diagram

Chapter 4
Design

In this chapter, we will take a closer look at the architecture of our system. We will discuss entities and the relationships between them, different components that make up the system and how they interact with each other. Next, we will explain which database scaling methods will be used on the application and why. Lastly, sequence diagrams will showcase the system's more complex operations.

4.1 System's data design

The application has two main roles - Student and Teacher. These roles are inherited from the User entity because they have some shared functionality such as creating, modifying and accessing their timetables. User accounts are created automatically. After a user is no longer associated with the university, all their data is erased from the database completely.

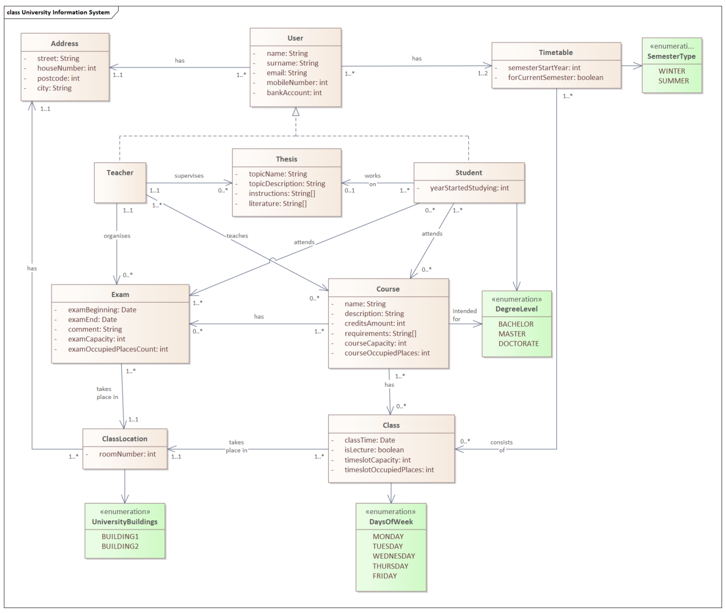


Figure 4.1: Class diagram

Each semester, a new empty timetable is created for all users. Only previous and current semester's timetables are persisted for each user. In order to differentiate between them, we need to record the semester's start year and it's period. For this purpose we could have created an enumeration class which would contain all semester codes, but it is not an ideal solution since the enum would always increase in size. Instead, we have introduced the semesterStartYear attribute and the Semester type enumeration class.

Timetable entity is used to record all classes that student is attending in a given semester. Individual classes are represented by the Timeslot entity, which records the course that the class belongs to and the time when the class begins each week. Each class has a location that is represented by the helper entity ClassLocation.

4.2 System's architecture

Component diagram for our system consists of 3 different parts. The first is the API gateway, which represents the entry point of our application. It receives a request from the user, which is then routed and sent to the responsible microservice.

System's functionalities are divided among four different microservices:

- **User service** - User service manages user's information such as their name, degree level and email. It allows users to access and modify their personal information.

- **Exam service** - Exam service records available exam dates for all courses. Through it, teachers have the ability to create and modify exam dates for the courses that they are teaching, while students can register for these exams.

- **Thesis service** - Thesis service keeps track of all thesis topics available to the students. Teachers can add new thesis topics that are supervised by them, and then students can register for theses.

- **Course service** - Course service holds information about all offered courses and their available classes. Teachers create and modify courses and the classes that belong to them. After that, students can register for different classes. Course service is also responsible for managing user's timetables.

All our microservices share the same structure. Controller layer receives requests from the API gateway, which is then passed to the service layer. Service layer implements all business logic and is connected to the repository layer, which communicates with the database. Our system's architecture uses the database per service pattern, meaning that each service has its own private database that the other services can not access.

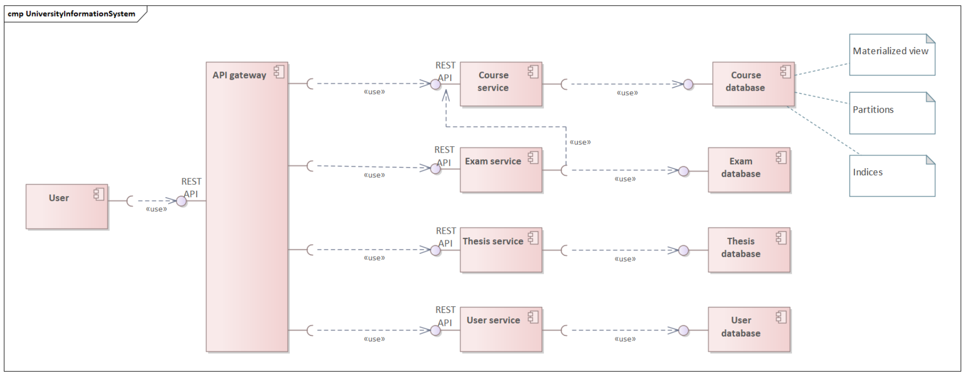


Figure 4.2: Component diagram

4.3 Persistence layer design

In order to further optimize our database, we created a materialized view with indices and partitioned certain tables. Optimizations were mainly done on course service's database since it contains the most data and is also most often accessed

4.3.1 Materialized views

Generally, the most frequently requested data in our application is the user's timetables. Therefore, it is very important to optimize the data retrieval for this use case.

The problem is that to build a user's timetable with all the classes and their locations, application needs to combine data from five tables, making it a quite expensive operation due to all the table JOINs. When you also consider the fact that it is also one of the most used operation, you quickly realise that it would be a massive performance hit for our application.

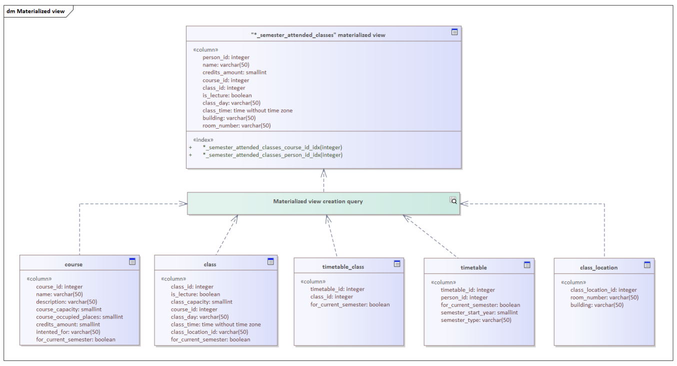


Figure 4.3: Materialized view created from five tables

In order to solve this problem, materialized view was used. Since the main benefit of the materialized view is the precomputation of data, we can take advantage of that and have the required data in one place. It also means that we pay the price of joining tables only when the materialized view is refreshed, instead of doing it on every timetable retrieval request. Moreover, the problem of data in the materialized view getting stale does not affect us as much because the underlying tables are updated only during the course registration period each semester. After this period is over and the new semester begins, we create our materialized view that does not require refreshes until the next semester.

4.3.2 Partitions

Our application has to store all information about courses, their classes and timetables for two semesters: the current one and the previous one. When the new semester comes, information about timetables and individual classes for the old semester needs to be erased, but information about courses is retained for the future use. In order to better manage historic data without compromising on performance, table partitioning was introduced.

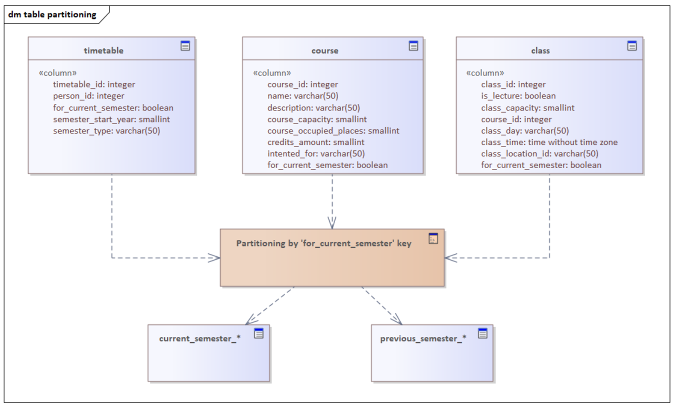

dm table partitioning
timetable
«column»
timetable_id: integer
person_id: integer
for_current_semester: boolean
semester_start_year: smallint
semester_type: varchar(50)
course
«column»
course_id: integer
name: varchar(50)
description: varchar(50)
course_capacity: smallint
course_occupied_places: smallint
credits_amount: smallint
intented_for: varchar(50)
for_current_semester: boolean
class
«column»
class_id: integer
is_lecture: boolean
class_capacity: smallint
course_id: integer
class_day: varchar(50)
class_time: time without time zone
class_location_id: varchar(50)
for_current_semester: boolean
Partitioning by 'for_current_semester' key
current_semester_*
previous_semester_*

Figure 4.4: Table partitioning for three tables

As you can see in the figure above, three tables were partitioned: ’course’, ’class’, ’timetable’. Column ’for_current_semester’ was chosen as a partition key. Since data from the current semester will be accessed much more often, query performance will increase thanks to the partition pruning, as in this case the data from the previous semester can be skipped.

Another benefit of keeping timetables and classes from the previous semester in separate partitions is the efficient disposal of old data. Instead of having to do a bulk delete of data on the main table, we can just drop a partition which is far more faster.

4.3.3 Indices

Since our materialized view can contain thousands of rows, it is a good idea to introduce indices. This will let us greatly speed up the data retrieval.

When querying our materialized view, which we described earlier, most of the time we search through either ’person_id’ or ’course_id’ columns. This is why we have set up an index on each of these columns. With these indices in place, time complexity for the queries that are using one of them improved from linear to logarithmic. This increase should be noticeable in the case of large datasets.

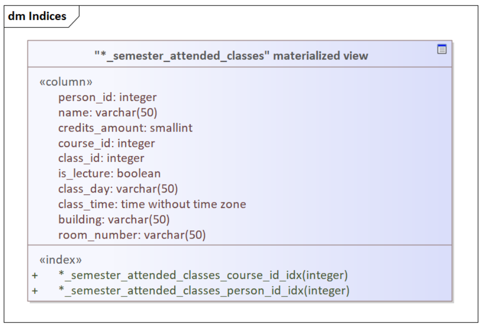


Figure 4.5: Indices on materialized view

4.4 Examples of system's functionalities

To showcase some important functionalities of the system, three sequence diagrams were created.

4.4.1 Registering student for a class

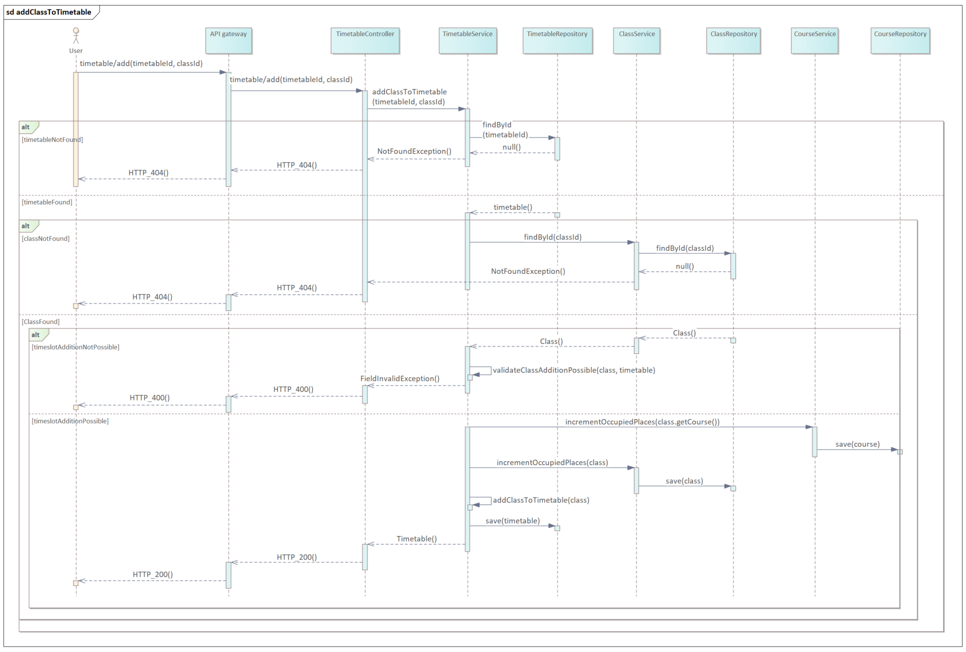


Figure 4.6: Sequence diagram addClassToTimetable

The class registration request first comes to the API gateway. After that, it is routed to the timetable controller, which then forwards it to the timetable service. If the required timetable and class can not be found in the repositories, then user gets a 404 not found error. If timetable and class are found, then it is validated if the timetable has space and if the class has free places left. If successful, class is added to the timetable and the amount of occupied places is incremented for both the class and the course that it belongs to. On validation failure, user receives a 400 bad request error.

4.4.2 Registering student for a thesis

The thesis registration request is forwarded from API gateway to thesis controller. If the required thesis can not be found in the repository, then user receives a 404 not found error. If thesis is found, then it is checked whether it is not occupied by another student or if the student is already registered for another thesis. If successful, student is registered for that thesis. On validation failure, user receives a 400 bad request error.

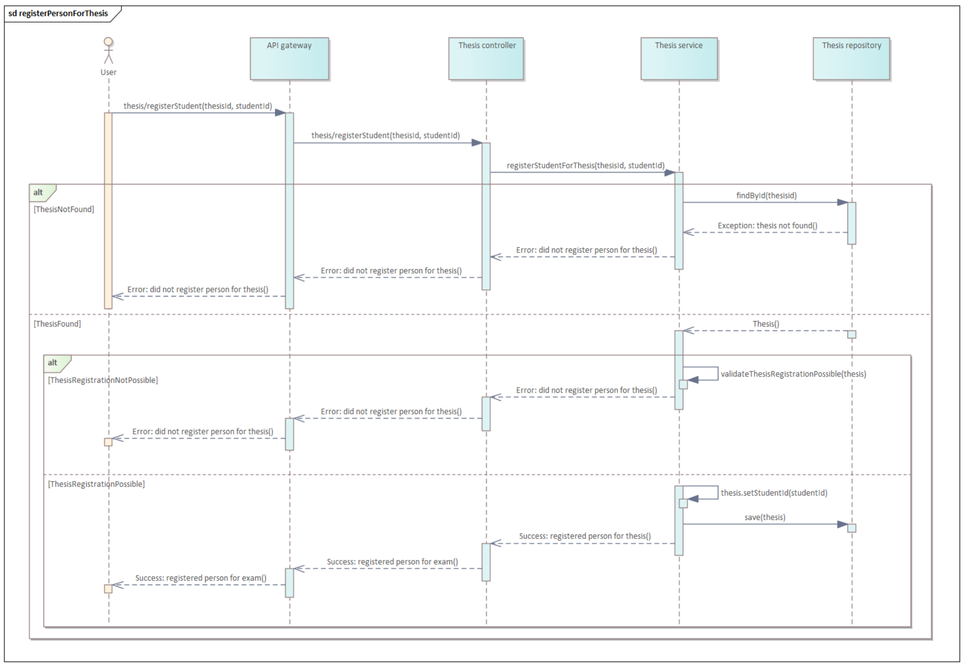

sd registerPersonForThesis
User
API gateway
Thesis controller
Thesis service
Thesis repository
thesis/registerStudent(thesisId, studentId)
thesis/registerStudent(thesisId, studentId)
registerStudentForThesis(thesisId, studentId)
alt
[ThesisNotFound]
findById(thesisId)
Exception: thesis not found()
Error: did not register person for thesis()
Error: did not register person for thesis()
Error: did not register person for thesis()
[ThesisFound]
Thesis()
alt
[ThesisRegistrationNotPossible]
validateThesisRegistrationPossible(thesis)
Error: did not register person for thesis()
Error: did not register person for thesis()
Error: did not register person for thesis()
[ThesisRegistrationPossible]
thesis.setStudentId(studentId)
save(thesis)
Success: registered person for thesis()
Success: registered person for exam()
Success: registered person for exam()

Figure 4.7: Sequence diagram registerPersonForThesis

4.4.3 Registering student for an exam

The exam registration request is sent from API gateway to exam controller. If the required exam can not be found in the repository, then user receives a 404 not found error. After the required exam is found, then it is checked if it has free capacity left and whether the student is not registered for another exam for this course. If successful, student is registered for that exam. On validation failure, user receives a 400 bad request error.

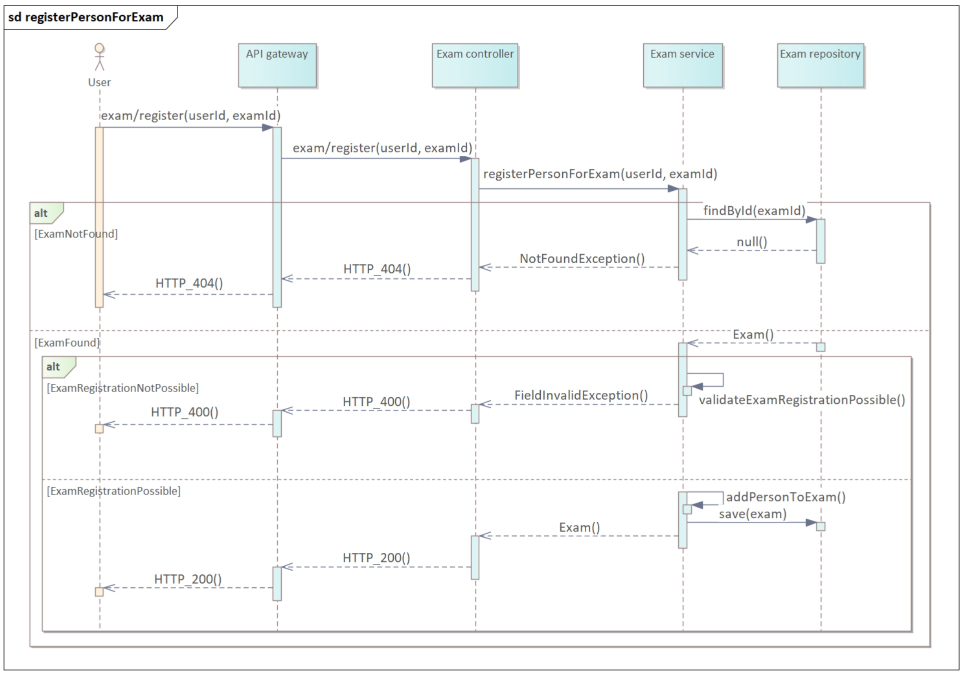


Figure 4.8: Sequence diagram registerPersonForExam

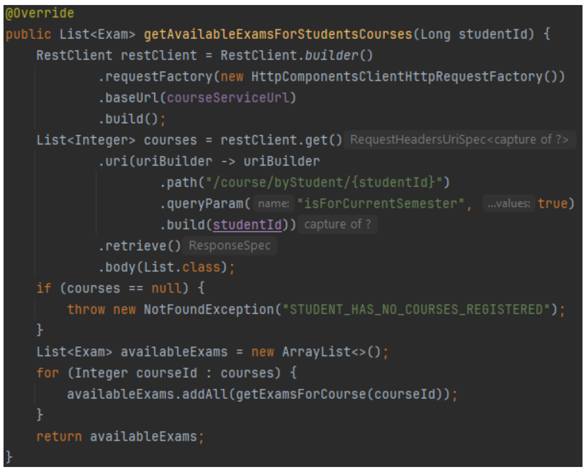
Chapter 5

Implementation

In this chapter we will have a detailed look at the implementation of the system we introduced earlier. The goal is to demonstrate how different parts of the application were implemented.

5.1 Intra-service communication

Sometimes, one service needs to get data from the other. In our case, for the exam service to display all the available exams to the student, it first must find out what courses is the student registered for. Since this data is owned by the course service, the exam service must send a corresponding request to it.



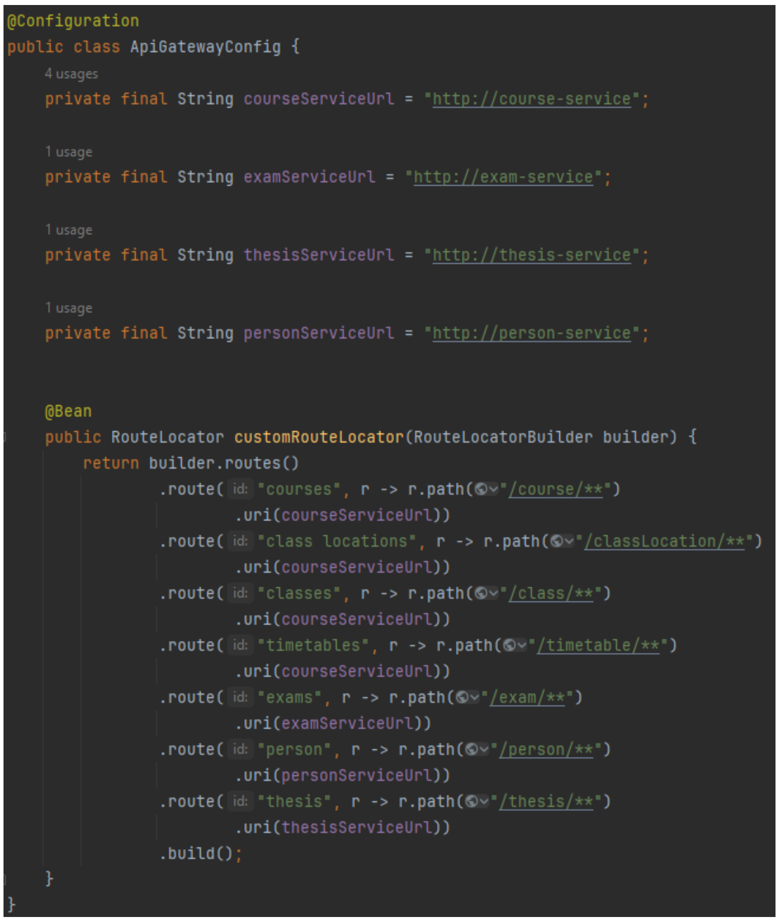
```java
@Override
public List<Exam> getAvailableExamsForStudentsCourses(Long studentId) {
    RestClient restClient = RestClient.builder()
            .requestFactory(new HttpComponentsClientHttpRequestFactory())
            .baseUrl(courseServiceUrl)
            .build();
    List<Integer> courses = restClient.get() RequestHeadersUriSpec<capture of ?>
            .uri(uriBuilder -> uriBuilder
                    .path("/course/byStudent/{studentId}")
                    .queryParam( name: "isForCurrentSemester",  ...values: true)
                    .build(studentId)) capture of ?
            .retrieve() ResponseSpec
            .body(List.class);
    if (courses == null) {
        throw new NotFoundException("STUDENT_HAS_NO_COURSES_REGISTERED");
    }
    List<Exam> availableExams = new ArrayList<>();
    for (Integer courseId : courses) {
        availableExams.addAll(getExamsForCourse(courseId));
    }
    return availableExams;
}
```

Figure 5.1: Example of an intraservice communication

In our case, this communication between different services is implemented with the RestClient library. First, we need to define the address of the target service, then we build a request by specifying the path and supplying the request parameters. In our case, we send the request to the course service with the two parameters: student id and whether we are interested in the current semester's data.

5.2 API gateway

In our application, API gateway was implemented as a separate component with the Spring Cloud Gateway library. As you can see in the code sample below, the gateway routes the incoming requests to the corresponding services based on their paths. Also, since our services are running in Kubernetes, we have to make sure that their addresses match the names of the service objects deployed in Kubernetes.

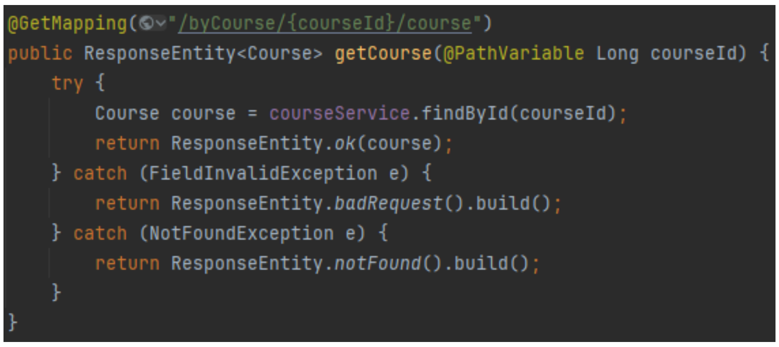


```
@Configuration
public class ApiGatewayConfig {
    4 usages
    private final String courseServiceUrl = "http://course-service";

    1 usage
    private final String examServiceUrl = "http://exam-service";

    1 usage
    private final String thesisServiceUrl = "http://thesis-service";

    1 usage
    private final String personServiceUrl = "http://person-service";


    @Bean
    public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
        return builder.routes()
                .route( id: "courses", r -> r.path("/course/**")
                        .uri(courseServiceUrl))
                .route( id: "class locations", r -> r.path("/classLocation/**")
                        .uri(courseServiceUrl))
                .route( id: "classes", r -> r.path("/class/**")
                        .uri(courseServiceUrl))
                .route( id: "timetables", r -> r.path("/timetable/**")
                        .uri(courseServiceUrl))
                .route( id: "exams", r -> r.path("/exam/**")
                        .uri(examServiceUrl))
                .route( id: "person", r -> r.path("/person/**")
                        .uri(personServiceUrl))
                .route( id: "thesis", r -> r.path("/thesis/**")
                        .uri(thesisServiceUrl))
                .build();
    }
}
```

Figure 5.2: API gateway config

5.3 Exception handling and HTTP response configuration

In order for the application to differentiate between different types of errors, custom exceptions were used. After the exception was thrown, controller needs to be able to send a different response to the user based on the caught exception. For this purpose, a ResponseEntity class was used. In figure 5.1 you can see that the controller sends the user the "not found" response if the service could not find the requested course and had thrown the "NotFoundException".

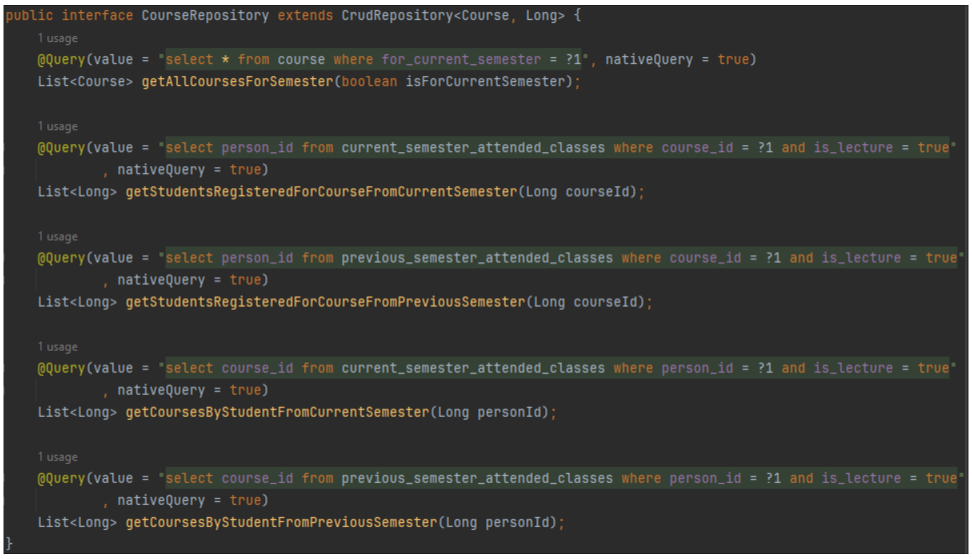


```java
@GetMapping("/byCourse/{courseId}/course")
public ResponseEntity<Course> getCourse(@PathVariable Long courseId) {
    try {
        Course course = courseService.findById(courseId);
        return ResponseEntity.ok(course);
    } catch (FieldInvalidException e) {
        return ResponseEntity.badRequest().build();
    } catch (NotFoundException e) {
        return ResponseEntity.notFound().build();
    }
}
```

Figure 5.3: HTTP response configuration example

5.4 Integration of optimizations done on the persistence layer

For our application to use the optimized queries, @Query annotation was used. For example, in the code fragment below you can see that we instruct custom repository method "getStudentsRegisteredForCourseFromCurrentSemester" to take data from the materialized view "current_semester_attended_classes", which we described in the chapter 4.3.1.



```java
public interface CourseRepository extends CrudRepository<Course, Long> {
    1 usage
    @Query(value = "select * from course where for_current_semester = ?1", nativeQuery = true)
    List<Course> getAllCoursesForSemester(boolean isForCurrentSemester);

    1 usage
    @Query(value = "select person_id from current_semester_attended_classes where course_id = ?1 and is_lecture = true"
            , nativeQuery = true)
    List<Long> getStudentsRegisteredForCourseFromCurrentSemester(Long courseId);

    1 usage
    @Query(value = "select person_id from previous_semester_attended_classes where course_id = ?1 and is_lecture = true"
            , nativeQuery = true)
    List<Long> getStudentsRegisteredForCourseFromPreviousSemester(Long courseId);

    1 usage
    @Query(value = "select course_id from current_semester_attended_classes where person_id = ?1 and is_lecture = true"
            , nativeQuery = true)
    List<Long> getCoursesByStudentFromCurrentSemester(Long personId);

    1 usage
    @Query(value = "select course_id from previous_semester_attended_classes where person_id = ?1 and is_lecture = true"
            , nativeQuery = true)
    List<Long> getCoursesByStudentFromPreviousSemester(Long personId);
}
```

Figure 5.4: @Query annotation usage example

Chapter 6

Deployment

This chapter describes how our demo application was containerized with Docker and then deployed in Kubernetes cluster. It also explains how the database replication was set up in Kubernetes.

6.1 Docker

In order to make our application more portable and further separate our services from each other, they had to be containerized. It was done by creating a Dockerfile for each individual service. These Dockerfiles were then used to build Docker images for each service. After that, they were pushed to a private container registry on GitLab.

6.1.1 Dockerfile configuration

In the listing below you can see an example Dockerfile for course service. In our application, Dockerfiles for other services have the same structure.

```
FROM openjdk:21
WORKDIR /
EXPOSE 8080
COPY target/course_service-0.0.1-SNAPSHOT.jar
    course_service-0.0.1-SNAPSHOT.jar
ENTRYPOINT exec java $JAVA_OPTS -jar course_service
    -0.0.1-SNAPSHOT.jar
```

Listing 6.1: Dockerfile for course service

Any Dockerfile starts with a FROM instruction, which specifies the base image for subsequent instructions. In our case, we use openjdk version 21 image. After that, we specify the working directory for our COPY instruction. The later instruction indicates that the container should listen on port 8080. Next, we copy our built application to the container's working directory that we specified earlier. The last instruction is a command which is ran at the startup of the container. Here, it just executes our application.

6.2 Kubernetes

To easily manage our containerized services, they were deployed to the Kubernetes cluster. For exemplary purposes, our application was deployed on my personal computer with the help of Minikube. Minikube is able to quickly set up a Kubernetes cluster on a local machine. For each our service, a Kubernetes deployment and Kubernetes service configurations were created.

6.2.1 Deployment and service configuration for the application

In the listing below you can see an example Kubernetes deployment and service configuration for course service. Both service and deployment are kept in one YAML configuration file. Same as with Dockerfiles, Kubernetes configurations for our services are pretty similar.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: course-service
spec:
  replicas: 3
  selector:
    matchLabels:
      app: course-service
  template:
    metadata:
      labels:
        app: course-service
    spec:
      containers:
      - name: course-service
        image: registry.gitlab.com/nalutmik/
               university-information-system/course-
    service:latest
        ports:
        - containerPort: 80
        env:
        - name: DB_HOST
          value: pgpool
      imagePullSecrets:
      - name: regcred
---
apiVersion: v1
kind: Service
metadata:
  name: course-service
```

```
spec:
  selector:
    app: course-service
  ports:
      - port: 80
        targetPort: 8080
        protocol: TCP
  type: ClusterIP
```

Listing 6.2: Kubernetes deployment and service configuration for course service

Deployment specifies which containers should run in a single pod. In our case, we specify the name of the container, it's port and the amount of replica pods to be created. Since this deployment will try to pull the container image from a private container registry, we created a secret object 'regcred', which contains authorization credentials for the registry. We have also set up an environment variable 'DB_HOST', which represents a hostname for our database.

Service enables network exposure for pods specified in the selector. In our case, it is a deployment object that we described earlier. We also specified on which port the service will listen and to which port it will forward incoming requests. Since we have API gateway acting as a LoadBalancer, all our main services are of the 'ClusterIp' type. ClusterIp makes the service reachable only within the cluster.

Even though we could have created an Ingress object for exposing our services to the external traffic, in the end we went with the LoadBalancer object for simplicity's sake.

6.2.2 Kubegres configuration for the database

In order to implement data replication across multiple database instances, Kubegres was used. Kubegres is a Kubernetes operator which is able to deploy one or more clusters of Postgresql instances.[26] In such cluster, a primary read-write pod is deployed and optionally any number of read-only replica pods. The data replication between the primary and the replica pods is automatic.

In the listing below you can see the configuration of our Kubegres cluster. Here, we set the amount of replicas as 3, which means that Kubegres will create one primary pod and two read-only replicas. We also specify the amount of resources that each pod can use. Finally, we need to define passwords for Postgresql's super user and a replication user. These passwords are kept in a separate secret resource.

```
apiVersion: kubegres.reactive-tech.io/v1
kind: Kubegres
metadata:
  name: mypostgres
  namespace: default
```

```

spec:
  replicas: 3
  image: postgres:latest

  database:
    size: 200Mi
    volumeMount: /var/lib/postgresql/data

  resources:
    limits:
      memory: "500Mi"
      cpu: "1.5"

  env:
    - name: POSTGRES_PASSWORD
      valueFrom:
        secretKeyRef:
          name: postgresql-secret
          key: superUserPassword

    - name: POSTGRES_REPLICATION_PASSWORD
      valueFrom:
        secretKeyRef:
          name: postgresql-secret
          key: replicationUserPassword
```

Listing 6.3: Kubegres configuration

After we apply this config, Kubegres automatically creates persistent volumes and persistent volume claims for each database instance. Moreover, it also creates two services that allow access to the primary instance and replica instances. In our case, it will create a service named 'mypostgres' which is used to access the primary node and another service called 'mypostgress-replica', which provides access to all replicas.

6.2.3 Pgpool-II configuration

To fully utilize the benefits of multiple database instances, read queries need to be distributed evenly across all replica instances. This is why Pgpool-II was used as a load balancer. Pgpool-II is a proxy software that sits between PostgreSQL servers and a PostgreSQL database client[27].

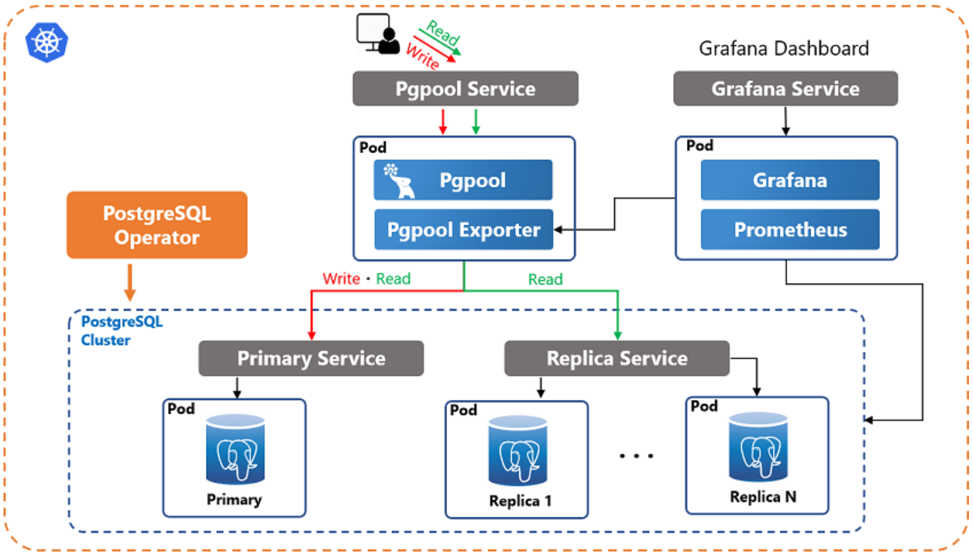


Figure 6.1: Pgpool-II architecture(source: [7])

For its function it requires a PostgreSQL operator, in our case it is Kubegres. Optionally, it can also collect and export database metrics for the monitoring system such as Prometheus, but that is a topic outside our work.

The easiest way to configure Pgpool-II is to include parameters in the pod as environmental variables. First of all, we need to specify the service names and ports for two backend nodes, that is primary service and replica service. In our case, we use the names of the services that were created by Kubegres. Another important setting is the backend_weight, which specifies the load balance ratio. By setting a higher weight for the replica, we move most of the read load from the primary node to replica nodes.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pgpool
spec:
  replicas: 1
  selector:
    matchLabels:
      app: pgpool
  template:
    metadata:
      labels:
        app: pgpool
    spec:
      containers:
      - name: pgpool
        image: pgpool/pgpool
        env:
            - name: PGPOOL_PARAMS_BACKEND_HOSTNAME0
              value: "mypostgres"
            - name: PGPOOL_PARAMS_BACKEND_PORT0
              value: "5432"
            - name: PGPOOL_PARAMS_BACKEND_WEIGHT0
              value: "1"
            - name: PGPOOL_PARAMS_BACKEND_FLAG0
              value: "ALWAYS_PRIMARY|DISALLOW_TO_FAILOVER"
            - name: PGPOOL_PARAMS_BACKEND_HOSTNAME1
              value: "mypostgres-replica"
            - name: PGPOOL_PARAMS_BACKEND_PORT1
              value: "5432"
            - name: PGPOOL_PARAMS_BACKEND_WEIGHT1
              value: "2"
            - name: PGPOOL_PARAMS_BACKEND_FLAG1
              value: "DISALLOW_TO_FAILOVER"
            - name: POSTGRES_USERNAME
              value: postgres
            - name: POSTGRES_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: postgresql-secret
                  key: superUserPassword
```

Listing 6.4: Pgpool-II configuration

Chapter 7
Testing

This chapter describes the performance test between scaled application and the same application but without scaling. It specifies how the test was setup and presents test results.

7.1 Locust

Locust is an open source performance/load testing tool[28]. It allows to generate a great amount of users which can simulate realistic use of the application. The testing scenarios are written in Python. The test run and its results can be observed in a web interface, which also collects performance statistics for each request.

7.2 Test approach

Since we want to simulate the usage of our application to be as realistic as possible, our Locust test was designed to prefer sending certain requests over others. For example, displaying user's timetable and browsing available classes for some course is a much more common operation than registering for an exam or thesis topic.

Each test was run for 5 minutes and had 240 generated users. Any amount of users above that threshold made the application throw connection errors for some requests after running for a while.

7.3 System's configuration

Testing was done on an application that is running in Kubernetes. The total amount of available hardware resources was 12 CPU cores and 7 gigabytes of RAM. These resources were used to create a total of 14 running pods. Out of these pods, one was reserved for Pgpool-II, while the database had three pods available. The application itself had 10 remaining pods, which were distributed according to the table 7.1.

Service name	Amount of pods
API gateway	3
Course service	3
Exam service	2
Thesis service	1
Person service	1

Table 7.1: Pod distribution for each microservice

Since the course service is by far the most used service in our application, it has three pods. API gateway also received three pods, because it has to be highly available to keep the application stable. Thesis service and person service each run only on one pod, which is enough for the load they are getting.

7.4 Test results

First, the test was run on the application without our optimizations. Application was altered to not use the database optimizations that we implemented. Also, to prevent the usage of the database replication, the amount of running PostgreSQL instances was scaled down to just one pod. Figure 7.1 showcases the complete test run.

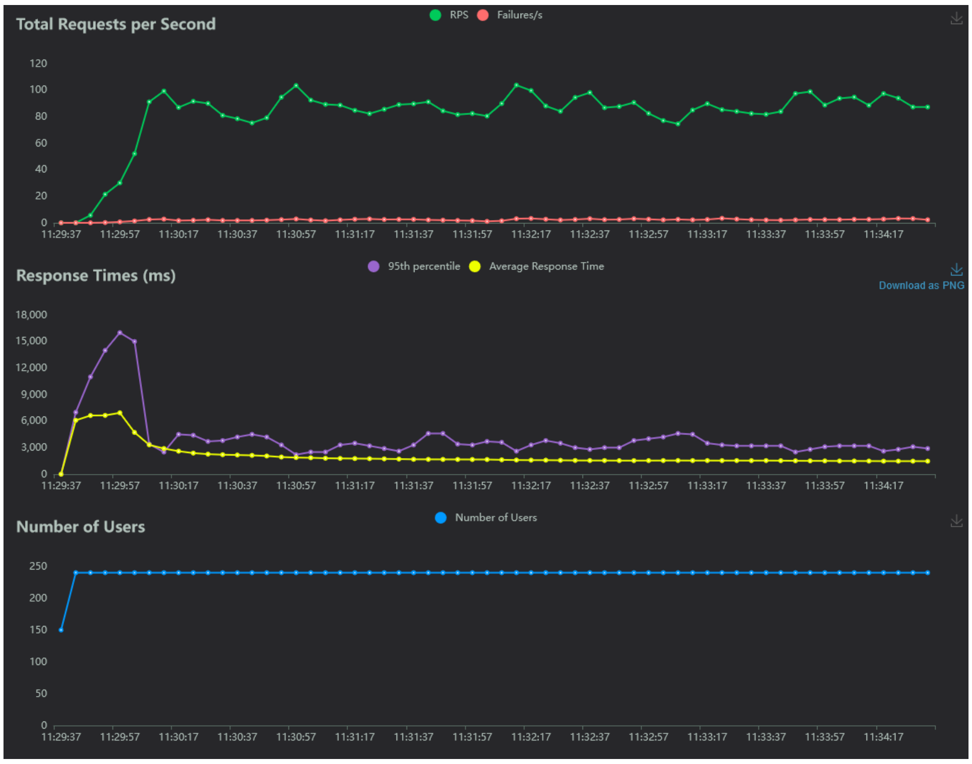


Figure 7.1: Test run without scaling

After that, the (changes to the application were reverted ?) application's ability to access our database optimization was brought back and the amount of PostgreSQL instances was increased to 3. You can see the test run below in the figure 7.2.

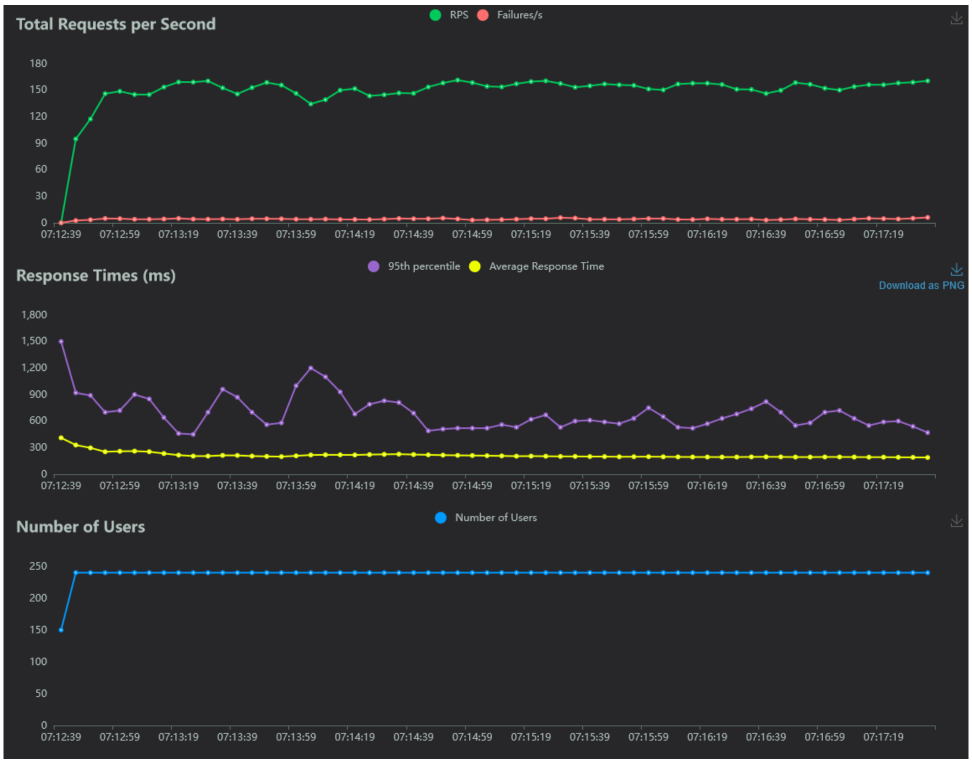
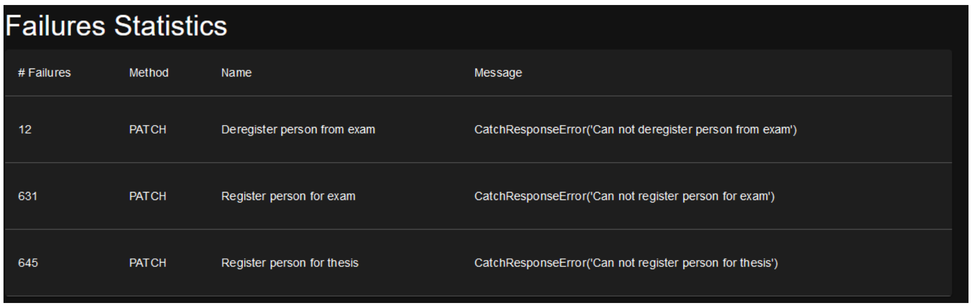


Figure 7.2: Test run with scaling

It is important to mention that both test runs had failures, but these are just exceptions thrown by the application itself. This happens because of the way we simulate registration requests in the test scenario code. You can see the example of failures in the figure 7.3.


Failures Statistics

# Failures	Method	Name	Message
12	PATCH	Deregister person from exam	CatchResponseError('Can not deregister person from exam')
631	PATCH	Register person for exam	CatchResponseError('Can not register person for exam')
645	PATCH	Register person for thesis	CatchResponseError('Can not register person for thesis')

Figure 7.3: Test run failures

In the table 7.2 you can see the overall test results. Thanks to our scaling, response time improved greatly, which almost doubled the median amount of requests per second. Naturally, a total number of processed requests has increased significantly.

	Response time in milliseconds	Requests per second	Number of requests
Application without optimizations	1466	87	25107
Application with optimizations	188	160	45479

Table 7.2: Test results with 240 users

7.5 Summary

Since our application is a read-heavy system, it is no surprise that the database scaling greatly increased the application's throughput. In a real production environment, where the amount of users is often measured in thousands, the scaling techniques that we used will still be relevant. For example, if the load surpasses the database's server capabilities, we could always increase the amount of database replicas.

Chapter 8

Conclusions

The goal of this thesis was to research different methods for database scaling and measure the effectiveness of some of these methods on a demo application.

In the first part of this thesis we researched different types of scaling such as horizontal and vertical. Then, methods for horizontal scaling and database optimization techniques were explained. After that we characterized microservice architecture, its advantages and disadvantages and ways how to implement a communication between services. Lastly, we discussed containerization and orchestration technologies.

In the analysis chapter, we described a demo application on which we would later demonstrate the effects of scaling. Then, we wrote down the functional requirements for the application and created a use case diagram.

In the design part, we first identified system's key entities and the relationships between them. After that, we described system's architecture by explaining each service's role and relationship between different components. Next, we specified which database scaling techniques were used and why we implemented them. Finally, we showcased some important functionalities of the system by creating three sequence diagrams.

In the implementation chapter, we explained how certain problems were resolved and what we used to accomplish it. More specifically, we talked about how specific parts of the application like intra-service communication, API gateway, HTTP response configuration and integration of optimizations were implemented in the code.

In the deployment part, we described how we containerized our application with Docker and then deployed on Kubernetes. We also presented the replication and load balancing tools that we used.

In the last chapter we compared our demo application which used scaling techniques with the same demo application but without scaling. It was done with the Locust performance testing tool. We described how the test was setup and then presented the test results.

Appendix A

Bibliography

[1] Microsoft. Replication pattern. https://learn.microsoft.com/en-us/previous-versions/msp-n-p/dn589787(v=pandp.10).

[2] Microsoft. Data partitioning. https://learn.microsoft.com/en-us/azure/architecture/best-practices/data-partitioning.

[3] Microsoft. Materialized view pattern. https://learn.microsoft.com/en-us/previous-versions/msp-n-p/dn589782(v=pandp.10).

[4] Microsoft. Microservice architecture. https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices/.

[5] Docker. Docker overview. https://docs.docker.com/get-started/overview/.

[6] Kubernetes. Cluster architecture. https://kubernetes.io/docs/concepts/architecture/.

[7] Pgpool-II. Pgpool-ii on kubernetes. https://www.pgpool.net/docs/pgpool-II-4.2.7/en/html/example-kubernetes.html.

[8] Statista. Average cost per hour of server downtime worldwide in 2017. https://www.statista.com/statistics/780699/worldwide-server-hourly-downtime-cost-vertical-industry/.

[9] Statista. Do you utilize microservices within your organization? https://www.statista.com/statistics/1236823/microservices-usage-per-organization-size/#statisticContainer.

[10] MongoDB. Database scaling. https://www.mongodb.com/basics/scaling.

[11] MongoDB. A guide to horizontal vs vertical scaling. https://www.mongodb.com/resources/basics/horizontal-vs-vertical-scaling.

[12] Microsoft. Interservice communication. https://learn.microsoft.com/en-us/azure/architecture/microservices/design/interservice-communication.

[13] Statista. Leading containerization technologies market share worldwide in 2023. https://www.statista.com/statistics/1256245/containerization-technologies-software-market-share/.

[14] Docker. Overview of docker hub. https://docs.docker.com/docker-hub/.

[15] IBM. Container orchestration. https://www.ibm.com/topics/container-orchestration.

[16] Kubernetes. Kubernetes. https://kubernetes.io/.

[17] Kubernetes. Kubernetes components. https://kubernetes.io/docs/concepts/overview/components/.

[18] Kubernetes. Pods documentation. https://kubernetes.io/docs/concepts/workloads/pods/.

[19] Kubernetes. Deployment documentation. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.

[20] Kubernetes. Statefulset documentation. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/.

[21] Kubernetes. Replicaset documentation. https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/.

[22] Kubernetes. Service documentation. https://kubernetes.io/docs/concepts/services-networking/service/.

[23] Kubernetes. Ingress documentation. https://kubernetes.io/docs/concepts/services-networking/ingress/.

[24] Kubernetes. Secret documentation. https://kubernetes.io/docs/concepts/configuration/secret/.

[25] Ministerstvo školství České republiky. Data o studentech, poprvé zapsaných a absolventech vysokých škol. https://www.msmt.cz/vzdelavani/skolstvi-v-cr/statistika-skolstvi/data-o-studentech-poprve-zapsanych-a-absolventech-vysokych?lang=1.

[26] Kubegres. Kubegres. https://www.kubegres.io//.

[27] Pgpool-II. Pgpool-ii documentation. https://www.pgpool.net/docs/42/en/html/intro-whatis.html.

[28] Locust. Locust documentation. https://docs.locust.io/en/stable/what-is-locust.html.