FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Johannes Lampela
Tommy Meriläinen

MESSAGING MICROSERVICE FOR LOVELACE LEARNING ENVIRONMENT

Bachelor's Thesis
Degree Programme in Computer Science and Engineering
June 2023

Lampela J., Meriläinen T. (2023) Messaging Microservice For Lovelace Learning Environment. University of Oulu, Degree Programme in Computer Science and Engineering, 43 p.

ABSTRACT

Microservices have emerged as a key component in addressing the challenges of the modern era. Companies are constantly seeking better ways to improve the scalability, extensibility, and flexibility of their applications. Microservices provide an excellent solution to these problems, as they consist of multiple small services within a larger system, each serving a specific purpose.

Dividing the application into smaller microservices allows individual services to be updated and expanded without the need to rebuild the entire application. This is much more efficient than having a single monolithic system in place.

In this bachelor's thesis, our goal was to create a messaging microservice for the Lovelace learning environment, which is used at the University of Oulu for example in programming-related courses. The current issue with Lovelace is the lack of an internal messaging system, with communication taking place via email, which causes various problems.

In the thesis, we described how we intended to implement this particular microservice. We discussed about the tools we used and how we designed the microservices to function. At the end we drew conclusions on how well the microservices suited this project and assessed our performance.

Keywords: microservice, messaging, Python, PostgreSQL, Flask

Lampela J., Meriläinen T. (2023) Viestintä mikropalvelu Lovelace-oppimisympäristölle. Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 43 s.

TIIVISTELMÄ

Mikropalvelut ovat nousseet keskeiseen asemaan nykyajan haasteiden edellytyksenä. Yritykset etsivät jatkuvasti parempia tapoja parantaa sovellustensa skaalautuvuutta, laajentuvuutta ja joustavuutta. Mikropalvelut ovat erinomainen ratkaisu näihin ongelmiin, sillä ne koostuvat useasta pienestä palvelusta suuremmassa systeemissä palvelemassa tiettyä tarkoitusta.

Sovelluksen jakaminen useaan pienempään mikropalveluun mahdollistaa sen, että yksittäisiä palveluita voi päivittää ja laajentaa huolehtimatta siitä, että joutuisi rakentamaan koko sovellusta uudelleen. Tämä on paljon tehokkaampaa kuin se, että meillä olisi vain yksi yhtenäinen järjestelmä käytössä.

Tässä kandidaatintutkielmassa tavoitteemme oli luoda viestinvälitys mikropalvelu Lovelace-oppimisympäristöä varten, jota käytetään Oulun yliopistossa esimerkiksi ohjelmointiin liittyvillä kursseilla. Tämänhetkinen ongelma Lovelacessa on se, ettei siellä ole sisäistä viestintäjärjestelmää, vaan viestiminen tapahtuu sähköpostien kautta, joka aiheuttaa monia eri ongelmia.

Tutkielmassa selitimme miten aioimme toteuttaa kyseisen mikropalvelun. Kävimme läpi, millaisia työkaluja käytimme ja miten suunnittelimme mikropalvelun toimivan. Lopuksi teimme johdopäätöksiä siitä, miten hyvin mikropalvelu soveltui tähän projektiin ja kuinka hyvin suoriuduimme.

Avainsanat: mikropalvelu, viestintä, Python, PostreSQL, Flask

TABLE OF CONTENTS

FOREWORD

We would like to thank Mika Oja for providing such a interesting subject and for supervising our work.

Oulu, June 8th, 2023

Johannes Lampela
Tommy Meriläinen

LIST OF ABBREVIATIONS AND SYMBOLS

API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
ORM	Object–relational mapping
REST	Representational State Transfer
RPC	Remote Procedure Call
SQL	Structured Query Language
URL	Uniform Resource Locator
WSGI	Web Server Gateway Interface

1. INTRODUCTION

Microservices have become increasingly important in software development during the past two decades. They allow great flexibility and scalability in building complex applications compared to monolithic systems. The shift towards DevOps and Agile development practices have accelerated the importance of delivering softwares on the market as quickly as possible.

With this in mind, our task focused on the development of a microservice for the learning environment Lovelace designed and created by Miikka Salminen. More specifically, we built a backend for a messaging tool that will facilitate communication between the participants of the site. This tool provides a convenient platform for students to ask questions and for other people to answer them, and it will help teachers stay more organized and keep track of their students' interactions. The reason why we considered this project to be particularly important is because microservices are being increasingly used year by year, and this was a perfect opportunity to have a better understanding how microservices work in-depth.

At first, we wrote about related work. What are microservices in general and frameworks that could be used in the project. Also, we took many different databases into consideration, since there were many good options to choose from. Then we demonstrated the planned design and the final technology stack we decided to go with. After that, we defined the implementation process in-depth, how our microservice communicates with the database and how the requests work. Security risks were also a thing we wanted to investigate, since microservices must be secure in order to be used in the first place. From the evaluation point of view, we did a lot of testing and evaluations based on the results.

With the time constrain we have, the actual software deployment part is excluded from the thesis. The frontend part is also dismissed. We focused solely on the services functionality so that if wanted, someone could use our messaging microservice as a base on their own web application.

2. RELATED WORK

2.1. Microservices

Microservices are independently releasable services that are modeled around a business domain. The idea is to build complex systems from building blocks, that are much easier to handle as individuals. Microservices give you options to choose how to solve problems you face, as each microservice encapsulates its own database where required and hides as much information as possible from the outside world, exposing as little as possible via external interfaces.

One of the key benefits of microservices is independent deployability. This means that changes made to a microservice can be deployed and released to users without having to deploy any other microservices. However, to achieve independent deployability, microservices must be loosely coupled, meaning that changes to one service should not require changes to any other services. To achieve this, microservices should be modeled around a business domain, allowing services to be structured to better represent the real-world domain that the software operates in.

Another important aspect of microservices is owning their own state. Microservices should avoid the use of shared databases, as sharing databases goes against independent deployability, which is one of the main reasons microservices exist in the first place. Instead, if a microservice needs to access data held by another microservice, it should go and ask that second microservice for the data. This approach helps to ensure that each microservice can be worked on in isolation and released on demand.

Microservices offer flexibility in architecture, making it easier to adopt new technologies with ease. When a system is composed of multiple, collaborating microservices, it is beneficial to use different technologies inside each one. This allows for the selection of the right tool for each job rather than having to select a more standardized, one-size-fits-all approach that often ends up being the lowest common denominator.

One major advantage of microservices is the ease of isolating problems. If one component fails, it is easy to isolate the problem and solve it, while the rest of the system can carry on working. This contrasts with monolithic systems where if the service fails, everything stops working.

However, working with databases can be more difficult with microservices than with monolithic systems, as working with multiple different databases can be troublesome. It is important to consider this when deciding whether to adopt microservices as an architecture.

When the core concepts of microservices are properly understood and implemented, they can help create empowering, productive architectures that can help systems become more than the sum of their parts. [1]

2.2. Lovelace

Lovelace is a web-based virtual learning environment created by Miikka Salminen that is mainly utilized by students from the Faculty of Information Technology and Electrical Engineering. Lovelace offers courses that mainly focuses on programming

and associated sciences. One of the most significant advantages of Lovelace is the variety of exercises. They range from simple and easy-to-set-up multiple-choice questions to more complex projects that can be automatically. evaluated even with multiple files. Learning material can be delivered in different ways, including normal text, graphics, embedded videos, and JavaScript applications. On top of that teachers can make studying more interesting by implementing interactive learning materials into the course materials. There is a tool that can analyze and give immediate feedback on exercises, which not only saves time for teachers but also benefits students [2].

Currently Lovelace supports sending emails to individuals, those who have made reservations on the calendar, and the entire class all at once. On top of that students can send help requests to the courses email list, and the system can send automated messages if there are problems with the checkers. However, there are a couple of issues with email, which is why an internal messaging system is preferable.

Firstly, students may not read the email address marked in the university systems. Secondly, messages sent by the system can end up in the spam folder. Thirdly, emails on the teacher's end can get lost in their inbox if they don't respond immediately, although this is mitigated by the ticketing system.

The advantage of an internal messaging system is that unread messages can be displayed in the system's user interface, making it easier to notice urgent messages compared to having to check email. Additionally, messages from the internal messaging system can be easily reused in the FAQ function, meaning that teachers can publish their written responses to some common question.

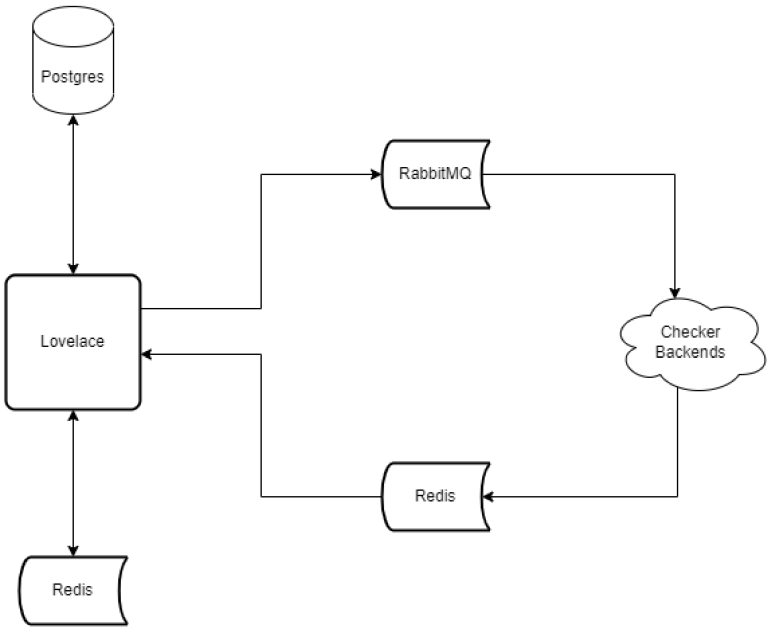

Figure 1. Current architecture for Lovelace.

As we can see from the Figure 1. the architecture model of Lovelace is very monolithic because only the checker processes have been isolated from the main domain for scaling and safety reasons. Monolithic model doesn't necessarily mean its something needed to be avoided as problematic since it's still very valid choice for architectural style [3]. But in this case, it makes it difficult to create new functionality

without creating any new bugs or larger problems within the system. To tackle these problems a change in the system is needed which we will go over more in the design part of this thesis.

2.3. Web Frameworks

Web frameworks drastically speed up the development of making web applications. They provide software tools that has structures and sets of libraries for building web applications.

Frameworks provide functionality in their code or through extensions to perform common operations required to run web applications, such as URL routing, input form handling and validation, different output formats with a templating engine, database connection configuration and persistent data manipulation through an object-relational mapper, Web security against common malicious attacks and session storage and retrieval.

Every web framework has their unique functionalities. For example, Django web application framework includes the Django ORM layer that allows a developer to write relational database read, write, query, and delete operations in Python code rather than SQL, but it cannot work without modification on non-relational databases such as MongoDB.

On the other hand, some other web frameworks such as Flask and Pyramid are easier to use with non-relational databases by incorporating external Python libraries. For the purposes of this project we were looking for a web framework that has good enough functionality for a microservice with easy extensibility [4].

Most of the self-hosted Python applications are often deployed using lightweight Nginx or with Web Server Gateway Interface (WSGI) servers because specially WSGI is a standardised way of working between Python web application frameworks and web servers. Both of them offer top-performance and are relatively easy to set up.

Nginx is a web server that includes many useful features such as load-balancing and basic authentication. Nginx is known for being simple yet high performant for application with high loads. Stand-alone WSGI servers often provide more efficient solutions while providing great performance when compared to traditional web servers. Few popular WSGI servers includes Gunicorn, Waitress and uWSGI, while Gunicorn being the recommended one for new Python web applications. All of them have their own pros and cons that needs to be evaluated when deploying the application. [5]

2.4. Databases

Databases play important role not only storing massive amounts of data, but they are essential part of many applications and services. There are multiple types of databases, but relational and Not Only SQL (NoSQL) databases are most common ones.

Relational databases store and provide access to data points that are related. Each row in the table has a unique ID, that can be easily used to make relationships between different data points. Relational databases has many benefits such as is the best at

maintaining data consistency even with large amounts of data and it can handle strict rules and policies. [6]

All NoSQL databases have one thing in common that they are not relational. They were created to be able to handle unstructured data and at the same time thrive where relational databases struggle. Advantages with NoSQL databases are that it can generally process data faster than relational databases. Another major advantage is that they are much more flexible than relational databases. [7]

In summary, databases are critical tools when needed to store large amount of data efficiently. Choosing a correct database for your application can be tricky since it depends on what kind of application you are building or what features you need from a database. Every database has their own pros and cons that need evaluated before deciding the one for your needs.

2.5. Social Media Messaging Services

There are many social media messaging services out there, each having their unique features and functions.

For example, Facebook Messenger is a multipurpose messaging service. Users can send messages, have a live video call, send pictures and other data to users. Whereas Snapchat is more focused on sending videos and pictures to other users that will disappear after set amount of time.

In general, the choice of which kind of messaging service the application needs to have completely depends on what kind of features you want to have in it. Each messaging service should mainly focus on the parts that are relevant to its functionality. There are so many different kinds of messaging services to choose from, and it should be taken into account when choosing which kind of application is used.

2.5.1. Discord

Discord has become one of the most popular messaging platform during the last few years. It is possible to send text messages, go in a voice call with someone or a group of people or you can even go in a video call with whoever you want with. There is also an option to choose whether you make a private or public chat rooms, where users can interact. There are many kinds of different servers, since it doesn't take much effort to make one. Everyone will find a server that fits their needs, whether it's related to programming, gaming or whatever you desire. Discord is popular due to its flexibility and customization, and it basically covers everything a person could ever need from a communication platform.

Due to massive increase in popularity, Discord uses ScyllaDB to handle enormous amounts of data. [8] It utilizes using nodes in a cluster, and more nodes are added as the data grows. System's performance also improves with these additions. Because of this, it doesn't rely on a single point of failure. The data is replicated across multiple nodes, minimizing the risk of data loss. [9]

Discord's API uses both, a HTTPS and REST API to operate. It also has persistent secure WebSocket based connection for sending and subscribing to real-time events.

This makes Discord a great application, it can handle enormous amounts of people communicating at the time. [10]

2.5.2. Telegram

Telegram is a cloud-based mobile and desktop messaging app that focuses on security and speed, greatly emphasized the security. The messages are heavily encrypted and they can self-destruct. [11] It uses end-to-end encryption, meaning that only you and the sender can see the data between the two of you. Only the intended recipient can decrypt it, meaning that it prevents third parties from accessing the data while transferring from one device to another. [12] Other than that, Telegram provides all the basic features that you would expect from a messaging service.

Telegram apps are open source, and they support reproducible builds. Telegram uses TDLib (Telegram Database Library), which is highly modifiable for third-party developers to create fast and secure Telegram apps. TDLib takes care of all the functionality features, such as network implementation details, local data storage and encryption. [13] The Telegram API is RPC-based, meaning that interacting with the API involves sending a payload representing a function calling and receiving a result. [14]

2.6. Application Programming Interface

Application Programming Interface (API) is a set of rules that allows different software components and programs to communicate and share information. APIs enable creating powerful connected applications and services for the users. APIs are nowadays used everywhere and they serve an important foundation for the digital world around us.

APIs can be categorized based on access permissions. Private APIs are used internally inside organizations while public APIs can be accessed by everyone. Partner APIs allows collaboration between companies by restricting the access. APIs can also be categorized by their architectural styles such as:

- Representational state transfer (REST)
- Simple Object Access Protocol (SOAP)
- GraphQL
- Webhooks
- gRPC

Each of the architectural styles serve specific purposes and offers different ways to approach creating an API. REST is easily the most popular one of the styles above, which uses standard HTTP methods to perform operations accessed via endpoints. [15]

2.7. Principles Of RESTful API

REST, like other architectural styles, obeys a set of guiding principles and constraints that define its nature. These principles must be satisfied for a service interface to be considered RESTful. According to the information provided by [16], the RESTful architecture encloses six guiding principles or constraints:

- Uniform Interface: The interface should uniquely identify each resource involved in the communication between the client and the server. Resources should have consistent representations in the server's responses. API consumers should utilize these representations to modify the state of resources on the server. Each resource representation should contain sufficient information to describe how to process the message and provide details on additional actions that the client can perform on the resource. The client should possess only the initial application URI, driving all other resource interactions dynamically using hyperlinks.

- Client-Server: The client and server should be separate and independent components, allowing them to evolve and scale independently.

- Stateless: Each request from the client to the server must contain all of the information necessary to understand and complete the request. The server cannot take advantage of any previously stored context information on the server.

- Cacheable: The cacheable constraint requires that a response should implicitly or explicitly label itself as cacheable or non-cacheable. If the response is cacheable, the client application gets the right to reuse the response data later for equilevant requests and a specified period.

- Layered System: Allows an architecture to be composed of hierarchical layers by constraining component behavior. For example, in a layered system, each component cannot see beyond the immediate layer they are interacting with.

- Code on Demand: REST allows client functionality to extend by downloading and executing code in the form of applets or scripts. The downloaded code simplifies clients by reducing the number of features required to be pre-implemented. Servers can provide part of features delivered to the client in the form of code, and the client only needs to execute the code. This constraint is optional.

3. DESIGN

3.1. Design Goal

The goal is to design a messaging microservice that provides a scalable solution for communication between users that solves the problems associated with the current system. The microservice needs to be its own standalone service and it must be easy to integrate into the existing system. Additionally, it must take authentication into account to ensure confidentiality. The microservice must be able to handle traffic performantly and be able tolerate failures and errors without crashing. Overall, the plan is to design a solution that is reliable, easy to integrate and is able perform adequately.

3.2. Minimum Requirements

We were given a few design requirements to offer precise instructions to successfully complete the project with desired result. The functional requirements given helped us to create clear guidelines on the expected functionality as it is important for the final product to meet the needs and expectations of the wanted results. The minimum functional requirements that were given for our project were:

- Messages include basic fields such as title, content, sender, and course space.
- Title- and content fields can be both in Finnish and English and additionally it can be easily to expanded to other languages too.
- Message receiver can be single student, group of students, whole course, course staff or the courses responsible teacher.
- The service must maintain information about which of the recipients has seen the message, and it must be able to retrieve information whether there are any unread messages and to retrieve them too.
- Messages can be linked to another message as a response.

On top of the given minimum requirements, we wanted to focus on making easy to read and clear code with documentation. That's because if someone else want to further develop the microservice it will be much easier.

3.3. System Design

We started design phase by getting familiar with all the different technology stack possibilities, wanted functionality, and with the architecture model. We decided to design the system in a way that it is very customisable and easy to swap or add different libraries or technologies.

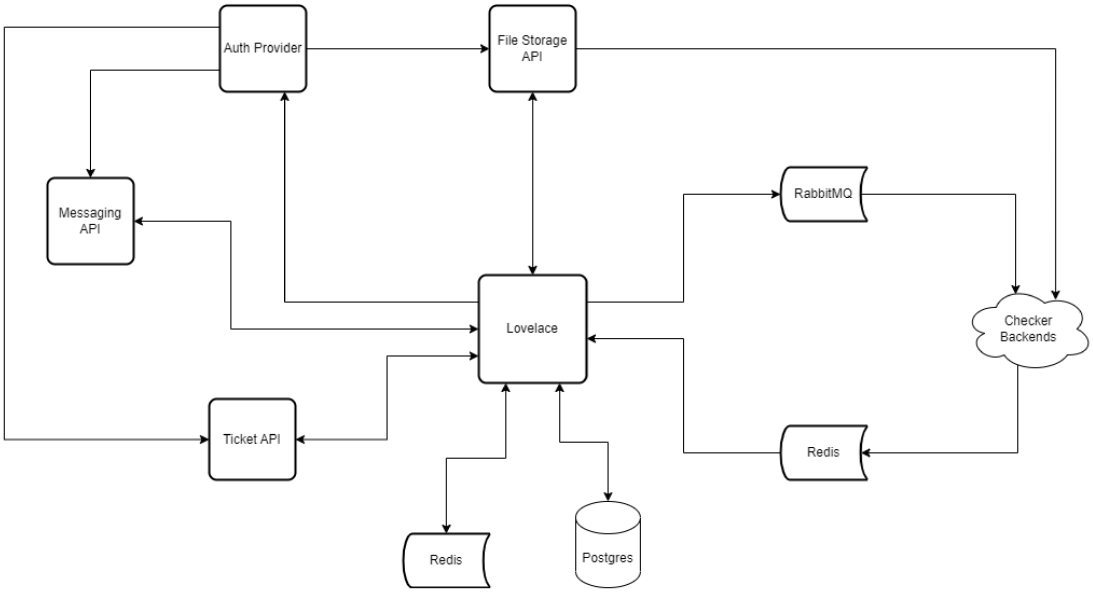

Figure 2. Lovelaces future architecture model.

Figure 2 shows the wanted future architecture model for Lovelace. When compared with the Figure 1 it is going to be much more modular and flexible because of the microservice model. Our thesis only focuses on the Messaging API even though Authentication Provider is going to be needed to provide a service that authenticates the users' messages. The Messaging microservice is the central component to allow users to send and receive messages. Using the microservice architecture allows us to create the above model without making major changes to the main Lovelace service. Additionally, using the microservice model allows us to create the microservice much more quickly without needing to go in depth with the main service.

Overall, all the new services for Lovelace are going to be isolated from the main domain as their own standalone services. It means that all the new functionality that is going to be created can be done one by one without needing to do a complete overhaul for Lovelaces main domain. One of the goal for these changes is to not only create new functionality but also make it more secure. This is done by making sure that any important personal information is not going to be moving outside the central server. The authorization is going to be done with authentication tokens that the authentication service is going to provide in the future.

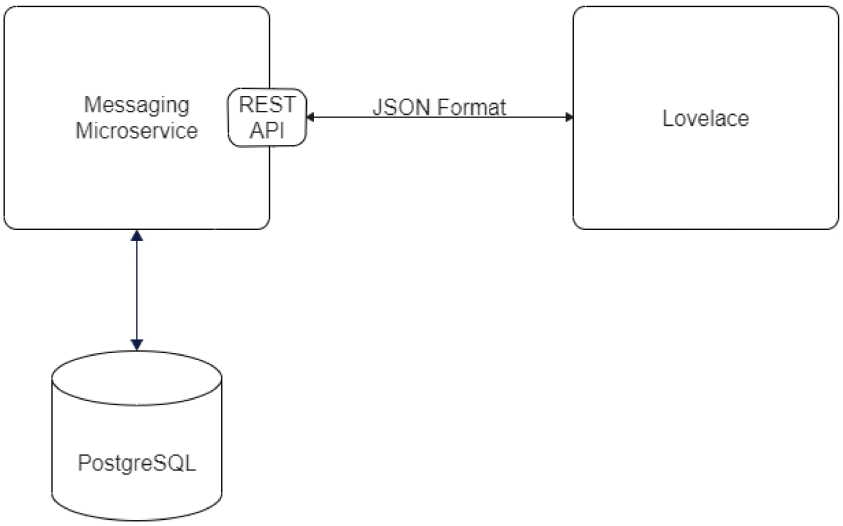

Figure 3. Messaging microservice model.

Communication between the Lovelace and Messaging microservice is done with using standard HTTP requests create, read, update, and delete (CRUD) operations. Therefore, we decided to go with REST architectural style over RPC. REST is designed to be stateless which makes it easier to scale it than RPC. REST is based on HTTP verbs GET, POST, PUT and DELETE to express different operations which makes it easy to understand versus RPCs more complex frameworks. [16] Even though RPC is more performant and efficient than REST it comes with a cost needing more work to implement it and being less scalable. [17] With the amount of traffic Lovelace generates REST should be more than efficient enough for this case. Additionally, our time was very limited, so REST seemed to be much better choice in our case.

These requests are going to use JavaScript Object Notation (JSON) that is a lightweight data-interchange format, which is easy to read and write. It's also easy to parse and generate since it's supported by all popular programming languages. JSON supports key-value pairs and an ordered list of values. One great thing with JSON is that it's universal and basically all programming languages supports it. [18]

When user sends a message, Lovelace sends it to Messaging API using POST request. The POST request includes the information such as title, content, sender and much more. Header in the request includes an authentication token that is being used to authenticate the user. Messaging API sends request to Auth Provider that returns if they match. If the authentication token is valid Messaging API can continue to save the message to the database and return a status code for success. Else if the authentication fails it won't save the message but returns a status code for authentication failure.

Similarly, if the user e.g., wants to retrieve all chats, Lovelace sends a GET request to Messaging API which again checks the authentication token and returns the chats with success status code if it matches. Else it would again return status code for authentication failure. The Messaging API supports the PUT method, which allows to e.g., update messages as read.

3.4. Technology Stack

We use Python as our programming language because not only was it required by the supervisor but it is packed with many great features. One of the main reasons is that we are familiar and have experience creating programs with it. Secondly, there is great support for web frameworks that allows to create microservices efficiently without having to handle low-level details as protocols, sockets, or process management. [19] Overall, the ease of use, large and active community, scalability and fast development makes it a great choice for our project.

3.4.1. Flask

As our web framework we decided to go with Flask micro web framework since we are working on a simple microservice with only a few API endpoints. We had multiple different options to go with such as FastAPI and Django. In the end we decided to go with Flask because it is rather easy to get started as a beginner since there is very little boilerplate needed to get app up and running. [20] Flask allows to developers to have complete control over their apps. Unlike other frameworks, Flask is made in a way that allows developers select or even create their own components. In addition, Flask supports numerous different database engines such as relational and NoSQL databases. [21]

Even though there are many more high-performance web framework, such as FastAPI, the reasons should not be only based on the speed alone. For example, features, and ease of development should be much more important factors since the bottleneck for most applications is going to be the design of the database and the architecture rather than the web framework. [22] In the end, Flask's adaptability, flexibility and lightweight approach and rather limited time to create our application made us choose it.

3.4.2. PostgreSQL

As our database of choice is PostgreSQL. PostgreSQL is an open-source object-relational database with decades of active development. PostgreSQL is known for being very reliable, feature robust, and high performant database. It has many helpful features such as strong support for JSON data types, powerful query language can handle structured data very well and highly scalable both in sheer quantity of data and number of users. [23] Since our data is going to be rather structured, with defined fields PostgreSQL appeared to be a strong choice for our project. On top of that as we need to track the messages read status and if messages are linked to another there PostgreSQL's strong transactional consistency comes in handy. There are dozens of database engines which we considered such as NoSQL database MongoDB. In the end we came into a conclusion that PostgreSQL would be a better choice for our project with the features mentioned above.

4. IMPLEMENTATION

4.1. Implementation Process

We started our project by creating a Github repository which allows us to easily stay organized and collaborate effectively. Github is a web-based platform that provides features such as cloud storage of the source code and version control. On top of the great features, Github is free which was a major key why we chose it. [24]

We utilized a Kanban board, which is a tool for visual representation of our project tasks and their status. Kanban board helped us to not only to maximize the efficiency but it helped us to stay on track of the current status of the project.

Based on our schedules, we tried to hold as many meetings as possible and discuss about potential issues that have arisen during the development of the project. The project itself got split into smaller tasks and we decided which tasks we would like to take. Eventually the tasks got sort of naturally split so that Meriläinen took the database tasks and Lampela focused on the messaging part. This strengthened the value of frequent meetings, since the database and the messages that will be sent have to go hand in hand in order to work.

Most of the time, the coding process went so that we were on a Discord call programming together. We utilized Visual Studio Code's built-in Live Share feature, which made it possible to work on the same code at the exactly same time. This further ensured that we were constantly on the same page about how we both imagine the code to function.

4.2. Microservice Implementation

We started the implementation process by getting familiar with the tools we would be using. In this case, Flask was the web framework we chose to use. We spent some time learning the basics of Flask, such as setting up basic web server and how to handle HTTP requests. After getting comfortable with the basics, we created a simple app that allowed both GET and POST requests.

Then we added a PostgreSQL database for the app using Flask extension called Flask-SQLAlchemy. It provides a SQL toolkit and Object-Relational Mapping (ORM) layer that allows not only to connect, but also to create everything from the tables to SQL queries easily with Python code. Flask-SQLAlchemy simplifies the process of working with databases because it allows to interact with databases using Object-Oriented programming (OOP) concepts, rather than writing raw SQL queries that can be cumbersome. On top of that, it's properly integrated with Flask so configuring everything is straightforward. [25]

With the database in place, we could easily start adding the wanted more complex functionality to our microservice. This way it's easier to start building functionality since we have a working foundation, which we can rely on.

4.2.1. Database

We started the actual implementation of the microservice by creating a database schema. We wanted to keep it simple but still being efficient, easy to maintain and ensures data consistency. As seen in Figure 4, we decided to create 3 different tables. One for participants, that allows to keep track of the users belonging to each chat. It also allows to create not only one-on-one chats but also group chats are possible with this model. Then there is the chats table, which saves the information belonging to specific chats. It includes information such as the topic and the course space it is created in. Lastly, there is the messages table that holds all the information belonging to the messages. It keeps track on if the message has been read by the receiver. The table keeps track if the message has been linked to another message. Messages and chats tables are referenced by an ID that is just a unique auto-incremented number that increases every time a record is inserted. Participants table has only two fields, one for referencing to chats table and another for messages table.

The reason why we decided to add two different IDs is that if we would used the chats table ID as the reference point for the API we would expose the size of the database for the users and possibly even create a security risk. Instead, we generate random Universal Unique Identifier (UUID) in order to disclose as little information about the inner workings as possible.

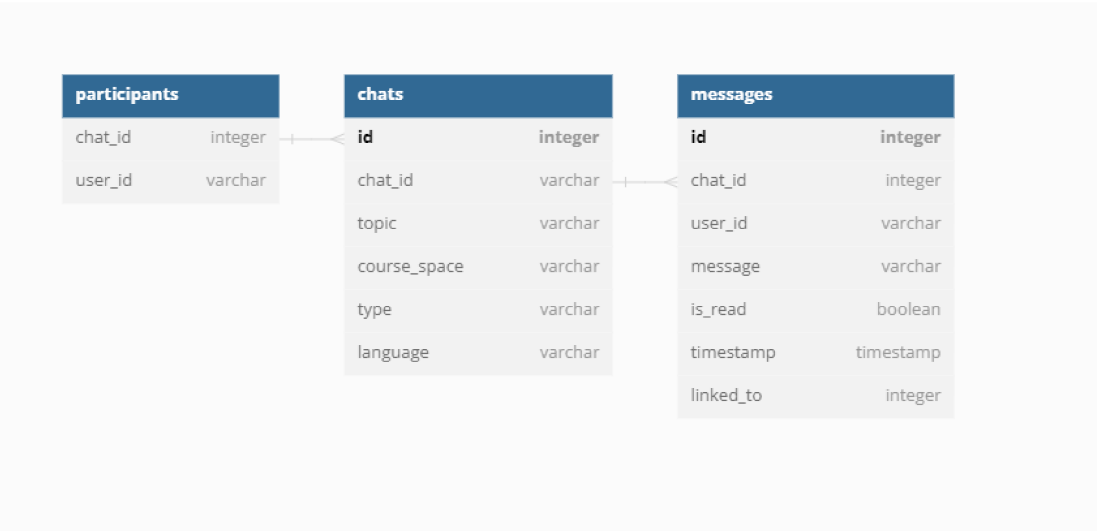

Figure 4. Database model.

Since we are using Flask-SQLAlchemy to implement the database it's rather straightforward process. Using ORM makes the database easier to write, maintain and makes it more secure because ORM is normally very good at mitigating against possible SQL injection vulnerabilities. [26]

```
class ChatModel(db.Model):

    __tablename__ = 'chats'

    id = db.Column(db.Integer, primary_key=True, unique=True)
    chat_id = db.Column(db.String(80), unique=True)
    topic = db.Column(db.String(80), nullable=False)
    course_space = db.Column(db.String(80), nullable=False)
    type = db.Column(db.String(80))
    language = db.Column(db.String(2))
    messages = db.relationship('MessageModel',
                               backref='messages',
                               lazy='dynamic',
                               order_by='messages.timestamp')
```

Figure 5. Example of chats database table using ORM.

As shown above, creating database tables is very swift since we only need to create a model, which is just a Python class. Each of the class variables represents a column in the database. First argument to db.Column tells SQLAlchemy what is the type of data it's going to be receiving. In our case, we are going to use integers, strings, booleans and datetimes. We can limit strings and integer lengths by giving the type an extra argument. Both chats and messages database tables use a simple auto incremented number every time a new record is inserted to table. That can be defined using the primary_key=True argument. Unique=True argument makes the column only accept unique values. Nullable argument defines if the column must have defined value or not. Db.relationship function creates a virtual column that connects with our messages model. First argument is the class we referencing to. Backref enables us to access the messages class. Lazy parameter controls how objects that are related will be loaded. Dynamic option loads the objects on access and before returning they can be filtered. Finally with order by argument we can order the messages by their timestamps.

4.2.2. REST API

We created the REST API using a extension called Flask-RESTful that adds support for quickly building REST APIs. This extension is lightweight abstraction, which is independent of ORM. One of the best things with this extension is that it's needs very minimal setuping and it encourages the best practices. Flask-RESTful allows to create API endpoints easily since it takes away much of the boilerplate code needed. We started by defining the main building block the extension provides called resources. Resources are built on top of Flask pluggable views, allowing you to quickly access multiple HTTP methods by defining methods on your resource. [27] This design pattern makes it easy to structure and organize the code in a clean way that it is much easier to maintain in the future.

```
class Chat(Resource):
    def get(self, userId, chatId):
        # Return chat with given chatId
    def post(self, userId, chatId):
        # Create a new message to given chatId
    def put(self, userId, chatId):
        # Update chat messages as read

class ChatLists(Resource):
    def get(self, userId):
        # Return list of users chats
    def post(self, userId):
        # Create a new chat with another user
```

Figure 6. Example how to create the resource classes.

We created two resource classes as shown above for our API. Both classes support the usual two HTTP methods: GET and POST. The chat resource supports the PUT method to allow update the chat messages read status. In order to these resources to work, we need to register routes to the resource classes. This is done by first initializing an API object, and creating an instance of it.

```
api.add_resource(ChatLists, 'api/chat/userId')
api.add_resource(Chat, '/api/chat/userId/chatId')
```

Figure 7. Example how to map the API endpoints

Then mapped the API endpoints to the API object using an add_resource method as seen above in the Figure 7. Then all the corresponding HTTP requests are linked to the correct resource. REST APIs are designed to be simple and intuitive so that each URL represents a specific value. In our case, the URL `api/chat/{userId}` is used to retrieve all current users chats or create a new one. Then with `api/chat/{userId}/{chatId}` URL you can retrieve the corresponding chat, send a new message into the specific chat or update read status of the messages.

HTTP Verb	CRUD	Entire Collection (e.g. api/chat/userId)	Specific item (e.g. api/chat/userId/chatId)
POST	Create	201 (Created), 'Location' header includes link to created resource. 400 (Fail), JSON validation error	201 (Created) 400 (Fail), JSON validation error 404 (Not found)
GET	Read	200 (OK) 404 (Not Found)	200 (OK) 404 (Not found)
PUT	Update/ Replace	Not implemented	200 (OK) 404 (Not found)
PATCH	Update	Not implemented	Not implemented
DELETE	Delete	Not implemented	Not implemented

Figure 8. HTTP methods used by the REST API

The table shown above summarizes the return values of the HTTP methods in combination with the resource Uniform Resource Identifier (URI) in our REST API. Sending a POST request successfully to an entire collection, the API returns HTTP status 201 with a link to the newly created chat in the 'Location' header. On the other hand, when sending a message to a specific resource, the API returns a status of either 200, 400 or 404, depending on whether the request was successful or not.

Similarly, sending a GET request successfully to the entire collection, the API returns HTTP status 200 along side with a list of chats that have been ordered by the newest message first. If no chats are found, the API returns a status of 404. Sending a GET request to a specific resource, the API returns status of 200 with all the messages related to the specific resource, or a status of 404 if the chat is not found. Lastly, we implemented a PUT request in order to enable the functionality to change the message read status in the database. Similarly, as other requests it return either 200 or 404 based on whether the request was successful or not. These return values provide a clear and consistent way to understand the REST API and its features.

4.2.3. Request Validation

JSON Schema is a declarative language that allows to annotate and validate JSON documents. It describes existing data format(s), provides clear documentation and validates data which is important for ensuring the quality of client submitted data. [28] JSON Schema itself is a JSON document and it provides a set of keywords and rules for defining the expected structure and data types of a JSON document. As we can see in the Figure 9, we have defined one of our used schemas so that the JSON sent by the client must include specific fields in a specific format in order to send it successfully. If one of these requirements are not met, it will return a validation error. This ensures that the data is correctly submitted, and errors will be detected early. We have schemas for every REST API endpoint.

```
chatlists_post_schema = {
    'type' : 'object',
    'properties' : {
        'receiver' : {'type' : 'string'},
        'course_space' : {'type' : 'string'},
        'topic' : {'type' : 'string'},
        'message' : {'type' : 'string'},
        'language' : {'type' : 'string'}
    },
    'required' : ['receiver', 'course_space',
                  'topic', 'message', 'language']
}
```

Figure 9. Example of schema used to validate a POST request.

We also used a "flask-expects-json" package made by Alfred Melch. The package uses JSON Schema to validate JSON data. It provides request validation for JSON payloads sent in the request body against a schema using JSON Schema. When a request is received with a JSON payload, it automatically parses the JSON data and validates it against the specified schema. When using this decorator - if the validation fails, it will return a validation error. This is really handy because we can add the JSON data validation without actually changing the code itself while profiting from an already established standard. [29]

4.2.4. Localization

We were given a requirement that fields can be in different languages, so we needed to implement a localization functionality for the application. Therefore, Flask-Babel extension to Flask comes in handy. It provides localization and internationalization support for Flask applications. Flask-Babel allows effortlessly add support for multiple languages. [30]

We configured Flask-Babel default locale and time zone variables to Finnish since most of the users are Finnish. Once configured, we could just start adding localized responses for our REST API endpoints using the functions provided by Flask-Babel. Additionally, Flask-Babel can not only provide localized responses but it can handle date and time formatting, number formatting and pluralization. Even though these are not used currently they can be useful in the future if the application is going to be further developed. In the end, adding localization using Flask-Babel was a swift process yet a valuable one since it makes the application more accessible and user-friendly.

4.2.5. Rate-Limiter

We decided to implement a rate limiter for the REST API that we can use to control the rate of requests client can send to the application. Main reason we wanted to add it was to prevent malicious users overwhelming the server with sending massive amounts of requests. Overwhelming the API with the requests can easily lead to poor performance or in the worst-case scenario server crashes. Additionally, rate limiter can help improve the overall performance by limiting the requests.

We decided to use Flask-Limiter extension as our rate limiter since it allows us to easily to configure various rate limits with minimal effort. It also allows to configure rate limits at different levels e.g., application wide or per resource. [31] This is not only a very useful feature but it's great for the future development of the application. Flask-Limiter support multiple different storage such as Memcached and Redis but we went with Memcached for its simplicity yet high-performance.

Adding the Flask-Limiter was very swift after configuring we only needed to add it to our resources. We gave all the HTTP requests rate limit of 30 requests per minute. We configured the rate limiter with a fixed window with elastic expiry strategy. If we would have used only a fixed window it would have allowed bursts within each window which allows to by-pass the rate limits. Using elastic expiry helps circumvent these bursts by locking out the attacker with a time penalty. [32]

4.3. Deployment

Flask comes with built-in development server, debugger, and reloader. The development server is great for local development, but it is not suitable for production use because it is not designed to be particularly secure, stable, efficient, or scalable. On top of that, the development server only provides a single synchronous process, which means that it can only handle 1 request at a time. [33] Therefore we decided to use an actual production server to get more accurate and reliable results. We used Gunicorn which is a production grade Web Server Gateway Interface (WSGI) server that can handle high volume of traffic with very acceptable performance. Gunicorn is simply implemented and can be deployed quickly. We also added nginx, an HTTP proxy server, that is strongly advised to use together with Gunicorn. Lastly, since we need to run multiple Gunicorn instances during the tests, we needed to containerize the microservice with Docker. This is because Gunicorn is a HTTP server for UNIX meaning that it doesn't support Windows. [34]

4.4. Risk Assessment

Risks	Likelihood	Impact
Security Risks	Unlikely	Major
Performance Risks	Common	Moderate
Availability Risks	Unlikely	Moderate

Table 1. Risk assessment

Security risks are major concern when creating a microservice. Adequate authentication and authorization mechanisms are key part of making a microservice secure. In our case, we do not create the authentication service as seen on Figure 2. Instead we need to worry about attacks such as SQL injection attacks. To prevent SQL injections, we have used a ORM package to help make our database more secure. Additionally, we have made a separate config file where we store different deployment environments with corresponding credentials. This way we can easily use the correct deployment environment.

Performance risks is the possibility that the microservice doesn't perform optimally under load, slow to respond, timeouts or even crashes. Likelihood of performance hiccups can be common, but we try to optimize the code and database in a way that we can avoid these hiccups as much as we can. There might be some slight hiccups but we try to optimize everything to avoid any larger performance issues. On top of that, we are going to test the microservice with tools that mimic the possible load to try find out any problems before deploying the code.

Availability risks is the possibility if the microservice is unavailable due to network or server issues. Luckily large availability risks are rather unlikely but not uncommon. To prevent these availability risks microservice performance needs to be regularly checked, e.g., databases performance issues before it causes any issues and use monitoring tools.

5. EVALUATION

5.1. Evaluation Plan

The way we are planning to evaluate the quality of the code includes testing functionality, performance, error handling and compliance with RESTful principles.

It is essential to ensure that the software's functions work as intended and it produces the correct results. We can accomplish this by performing functional testing and creating unit tests and integration tests. Unit tests are automated tests which ensure that a section of an application meets its design and behaves as intended. On the other hand, in integration testing, individual software modules are combined and tested as a group.

Evaluating the performance of the software is important because we want to know that the software is capable of handling the loads it is going to be facing. Here we use different load tests to find out whether the software can handle the data and deliver results in a reasonable time frame.

We also want our software to have proper error handling which helps users to identify the potential issues and provide them a solution to resolve the error. It is important to test the software with various error scenarios to make sure the system can handle unexpected errors effectively. Lastly, we want to make sure that our API is an actual REST API by evaluating compliance with RESTful principles.

5.2. Functional Testing

Functional testing is an essential part of this project since we want to ensure that the quality and functionality of the code meets our standards. During the tests, we wanted to make sure that our code work as intended and that they can handle errors as intended. This will not only allow us to fix any issues and ensure that our final product is high quality but it will help the future development.

We are using unit testing framework called unittest to write our test functions. The main goal is to make sure that our requests work properly and specially meets the minimum requirements. We can execute this by making different kinds of requests that tests posting certain data to the database and then using the get function to get it back to make sure it works or vice versa. We can also try getting messages from URIs that don't exist and make sure it handles the error properly. Making sure that we cover tests from the main code as broadly as possible is important because we can never know whether there is a slight bug in one line of the code. Also, we can't cheat on our test codes by forcing the tests to be successful, because it would give a false coverage rate.

A Python tool coverage comes helpful when we want to measure the tests code coverage. It allows us to see how extensively our code has been executed. The plan is to get the coverage rate as close to 100 as possible, but realistically we will be satisfied to land somewhere near 95 percent.

5.3. Performance and Load Testing

We are going to execute all the necessary performance and load testing to find out how our microservice handles the expected load. Performance testing ensures that our microservice can perform under different loads. Load testing ensures that the system can handle users at the same time without performance degradation. Doing these kinds of tests is also great to help identify any potential bottlenecks in our microservice, such as slow database queries. These tests can also ensure scalability whether it can scale to handle increased traffic. Identifying these bottlenecks helps to optimize our microservice and improve its performance.

We decided to use Apache JMeter as our application to perform our performance and load tests. JMeter is a open-source application that simulates real-user behaviours and testing environments, and it also provides a user-friendly GUI and easy installation for REST API testing. [35]

5.3.1. Baseline Tests

We wanted to do a baseline test in order to get a benchmark for our microservice performance during normal conditions. Baseline tests are important since it gives us a point of references for evaluating the performance under heavy loads or spikes. We created a test plan with JMeter that simulates the estimated average loads. We estimated that on a normal day it could be having around 100 people accessing the chat and we expect each user to make an average of 5 requests per minute. This is our estimate which is probably way above the actual average load.

Based on the estimates made, we started creating the baseline test in JMeter. We created the test plan in a way that it reflects the actual use as realistic as possible. Meaning that we do not only send one specific request e.g., sending only GET requests to get the chat list since the actual users are creating new chats, reading and sending new messages to specific chats. That is why we added different kinds of HTTP request to the test plan to reflect the real usage more. On top of that, we added a constant throughput timer to achieve the actual output by trying to produce a constant throughput. With the estimates made, we set the target throughput to little under 9 requests per second. Lastly, we decided to run the test for around 10 minutes to get more reliable results since if there happen any anomalies during a short test it can impact the results massively.

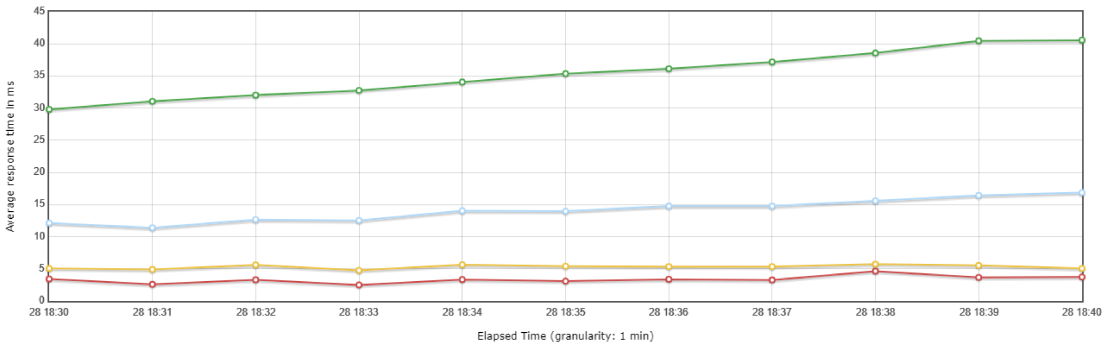

Figure 10. Response times during the baseline test.

As seen on the figure 10, the graph shows on the y-axis the average response times in milliseconds and the x-axis shows the elapsed time. In the graph, the green line represents the POST request that are sent when creating a new chat, blue one represents a GET request which is sent every time you want to get your chat list. Yellow line represents sending a message to specific chat and red line represents a GET request when you want to get all the messages from specific chat. This first baseline test was ran only with one server instance to see how well it can handle the load and to see more clearly if there are any large bottlenecks in the microservice.

Baseline test	Samples	Average	Min	Max	Throughput
Creating a new chat	1355	35	22	80	2.1/sec
Getting the chat list	1352	14	4	56	2.1/sec
Sending a message to a chat	1325	5	3	42	2.1/sec
Getting a specific chat	1311	3	1	62	2.1/sec
Total	5343	14	1	80	8.4/sec

Table 2. Results of the baseline test.

As the graph and table above shows, our microservice was easily able to handle the loads we estimated with good performance and without any failures. Table above shows that, creating a new chat is obviously most demanding request since it needs to query and write to the database more than other requests. Still, it was able to perform very well with the average of 35 milliseconds which is more than fast enough. Together the requests averages at 14 milliseconds, which is even better we anticipated before hand which shows that the microservice can easily handle the average loads we estimated before the tests.

5.3.2. Load Tests

Load testing simulates high volume of traffic to the application in order to measure how well it can perform under load. This also allows to identify any bottlenecks that impacts the performance under heavier loads. We estimated that if during normal conditions there is around 100 people accessing the chat, we decided to test how it performs if the traffic is 6 times more so around 600 users accessing and using the chat. Here we used

a similar setup as in the previous baseline test meaning that we have many different kind of HTTP request sent to the application. Due to these decision, we set the target throughput to around 60 requests per second.

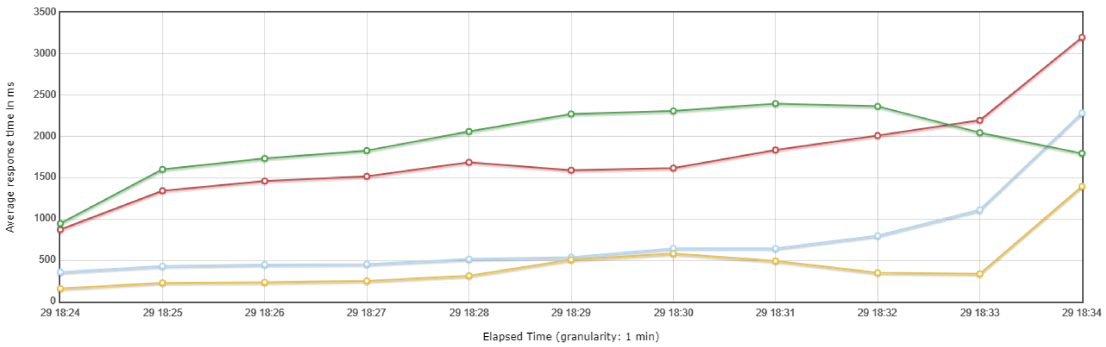

Figure 11. Response times during the load test.

Now looking at the above graph, response times increased dramatically. There is a few reasons why the response times rose so much. Firstly, flooding the single server instance with so many requests causes it to have task queue depth up to 30 tasks. This obviously increases the response times a lot because the requests need to wait for the others to complete before it can complete it. This is the main reason for the increase in response times.

Load test	Samples	Average	Min	Max	Throughput
Creating a new chat	9240	1917	66	4313	15.4/sec
Getting the chat list	9248	583	13	4814	15.4/sec
Sending a message to a chat	9266	338	2	4533	15.4/sec
Getting a specific chat	9272	1579	2	4595	15.4/sec
Total	37026	1104	2	4814	61.6/sec

Table 3. Results of the load test.

The table above strengthens the ideas why the response times have become much slower. One great finding from the load testing was that even though it couldn't perform as quickly we wanted it was still able to handle all the requests error free. After seeing the results, we added 3 more server instances and decided to add even more requests for the test. With 4 server instances, we set the request throughput to around 83 requests per seconds, which means that it simulates around 1000 users using the chat.

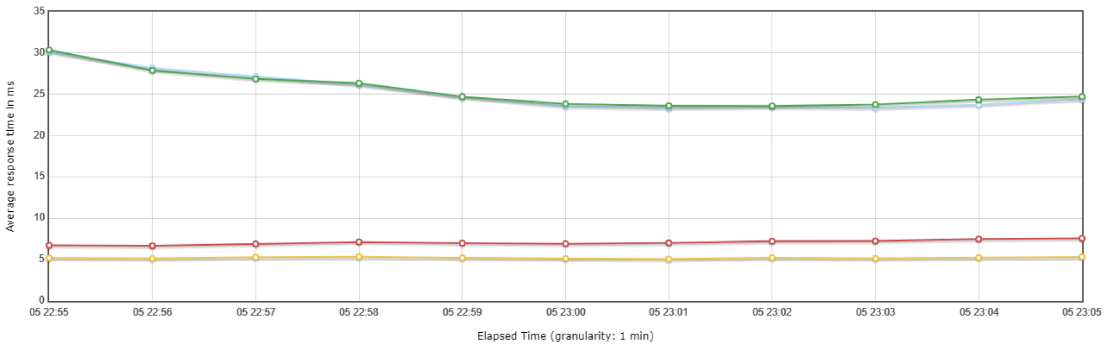

Figure 12. Response times during second the load test.

Comparing with Figure 11, the moving averages are much better. The lines show much better performance with longer the run time goes the better results we are seeing versus with previously the response times slowly started to increase. Now they are much more stable and don't have so much dispersion in them.

Load test	Samples	Average	Min	Max	Throughput
Creating a new chat	13326	25	13	88	20.8/sec
Getting the chat list	13332	25	13	93	20.8/sec
Sending a message to a chat	13346	5	3	42	20.8/sec
Getting a specific chat	13341	7	5	30	20.8/sec
Total	53345	15	3	93	83.3/sec

Table 4. Results of the load test.

After adding more server instances, together average response times being 15 milliseconds basically the same as during the baseline tests means that even with much higher amounts of users it can perform very well, as long as there is enough server instances to handle the loads it's been given. Throughput is over 30 percent more and the response times still can be measured in tens of milliseconds not in seconds means that even with much higher loads the microservice can perform as we want.

5.3.3. Results

During the baseline test, the microservice performed exceptionally well. Even with the most demanding request, involving querying and writing to the database, achieved very good response times. Overall, all the requests averaged at 14 milliseconds, surpassing our initial exceptions. These results indicates that the microservice can easily handle the estimated loads error free.

On the other hand, load tests with much more traffic caused the response times to increase significantly. Main reason for the increased response times was the high task queue depth in the server, causing requests to wait for completion, obviously rocketing the response times. Adding more server instances helped tremendously the response times since the average response time with single instance was 1.1 seconds and with multiple instances it was 15 milliseconds.

Despite the response times being higher with single instance the microservice was still able to complete every single requests successfully without any errors. The load test should be taken with a grain of salt because it doesn't actually represent the actual usage patterns of users, but it allowed us to identify one possible problem and find an answer for it. To avoid similar performance issues in the future, the performance should be investigated further. To improve the overall performance of the microservice, there could be done things such as:

- Add more instances
- Scale the infrastructure
- Optimize database calls
- Asynchronous processing

With the steps mentioned above, the microservice performance could be even more increased especially during the high loads. In summary, the tests demonstrated that the microservice could easily handle the different loads with great performance. All in all, these tests provided valuable information into the performance capabilities of the microservice under different kinds loads and give ideas how to enhance the performance.

5.4. Compliance with the RESTful Principles

In our opinion, the microservice mainly obeys the RESTful principles appropriately. We used the guiding principles of REST provided by [16] when we designed and programmed the API.

Uniform interface principles are mainly achieved because each resource is uniquely identified, resources have uniform representation in the response and messages are self-descriptive. Current version doesn't fully follow the hypermedia as the engine of application state (HATEOAS), but the code can be easily modified to comply the constraint fully. The client and the server components can evolve independently, except that databases and requests have to follow the defined schemas. The server doesn't take advantage of any previously stored context information on the server. Each request from the client to the server can be done with the information that is coming from the client side.

Our system is layered so that our requests and database are apart from each other and they can communicate with each other. The client does not have a third layer, so this is kind of trivial. If we were to make a HTML widget, it would have been the third layer and it could have only communicated with the request part. The widget would never know a database exists. The client application can obviously reuse the data sent to the database in the form of getting the chats. That is why the API is considered as cacheable.

Code on Demand is only optional principle of the RESTful architecture. It could have been implemented, but we didn't implement anything that would return applets or scripts. This can be done with the current implementation, but we only provide a REST API that web apps consumes. Therefore, it wasn't relevant regarding the thesis, hence why this part was unaccounted.

6. DISCUSSION

6.1. Reflection on the Project

Our project goal was to design and implement a microservice that will provide messaging functionalities for the Lovelace online learning environment. We were given minimum requirements to successfully complete the project with desired results and our microservice covers all the minimum requirements given at the start. We think that we fulfilled the design goals and additional requirements we decided together. The microservice was implemented with Python, Flask web framework and PostgreSQL as the database. Whole implementation process was surprisingly easy, for example it was easy to setup the database and use it with SQLAlchemy that allowed easy reading and writing to the database.

Current microservice version supports both one-on-one and group chats. The microservice allows to keep track of messages if it's linked to another message or if there are any unread messages. Additional features we added includes for example, request validation and a rate-limiter. We performed a lot of both manual and automated testing to find out that the microservice works correctly. Performance and load testing was done to verify the microservices performance. We found out that it could easily handle the loads we expected it is going to be having.

There are obviously many extra features and different functionality we wanted to add, for example a possibility to use a HTML widget. But the time constraints we had made it difficult to create everything we wanted. Therefore, we needed to focus mainly on the core functionality. We are satisfied how easy it is to read the code and understand it since we tried to keep any complex code at a minimum. We benefited a lot from all the books, documentation and different tutorials we came familiar with during the process. Especially, books contained surprisingly precise and useful information you would ever need in order to build a microservice. Now looking back at the technology choices made, we think that they were all appropriate for the use case. FastAPI could potentially been a stronger choice over Flask, but the benefits would have probably been so small that it wouldn't matter much.

6.2. Future Work

Next logical step for the project would be to integrate the microservice to Lovelace. But there are still a few things that need and should be implemented before doing so. Most important functionality that needs to be added is the authentication provider as seen on the Figure 2. Without e.g., token based authentication the service can not be used securely. Therefore, our current version of the microservice should not be yet to deployed into actual use.

6.2.1. HTML Widget

Since currently the microservice only provides a RESTful API without any visualised interface, implementing a HTML widget for the microservice could be beneficial for simpler integration and it would enforce data validation even further.

6.2.2. Logger

Another feature we would add before deploying the microservice would be adding a logger that allows to track messages the microservice generates. Since someone needs to maintain the microservice, a logger is pretty much a must have feature to trace out any possible errors. Even if the code is perfect there is always the possibility to have non-code related errors happen and without a logger it is very time consuming to find out the problem source. Python inbuilt logging system works with Flask but we would add more advanced service called Sentry. Sentry is open-source error tracking service that not only allows to monitor and fix errors in real time, but it allows performance monitoring to see any code related sluggishness. [36]

6.2.3. Deployment

When all the wanted features and functionalities are added to the microservice and it's ready to be integrated to Lovelace comes deploying to production into picture. Since the development server provided with Flask shouldn't be used in production there should be made a choice what server to use. There is a lot of great opinions for this but Gunicorn would be our choice since it is simply implemented, light on server resources, and fairly speedy. [34]

7. CONCLUSIONS

Currently Lovelace learning environment has issues with the sent emails not being read for multiple reasons. They can be easily go unread for both teachers and students. They can end up in the spam folder, students may not read the email address marked or simply can get lost if not responded immediately. Therefore, an internal messaging service is preferred.

To tackle these issues, we designed and implemented a messaging microservice. We started the project by getting familiar with the microservice service model and all the different options that we could use to make the best possible messaging microservice. Then we started to design the microservice based on the minimum requirements given to us. We added a few extra key points to focus on to create an application which is easier to further develop. Implementing the application was done by splitting the project into smaller tasks between us. In the end, we tested the application broadly to find out that it works correctly with various of different testing methods.

Overall, there are things that can be added and further developed with the changes proposed earlier but our thesis project fulfills all the minimum requirements we were given at the start. Microservice is a great addition to Lovelace learning environment and on top of that, the source code can be used anywhere else aswell if wanted. Learning about microservices changed our perspective on how to make an efficient and robust application.

8. REFERENCES

[1] S. N. (2021) Building Microservices: Designing Fine-Grained Systems Second Edition. O'Reilly, 3-34 p.

[2] Lovelace-ohje opettajille. URL: https://lovelace.oulu.fi/lovelace-ohje-opettajille/lovelace-ohje-opettajille/. Accessed 21.2.2023.

[3] S. N. (2021) Building Microservices: Designing Fine-Grained Systems Second Edition. O'Reilly, 18 p.

[4] Python web frameworks. URL: https://www.fullstackpython.com/web-frameworks.html. Accessed 21.2.2023.

[5] Web applications and frameworks. URL: https://docs.python-guide.org/scenarios/web/. Accessed 4.6.2023.

[6] What is a relational database? URL: https://www.oracle.com/database/what-is-a-relational-database/. Accessed 28.3.2023.

[7] Leavitt N. (2010), Will nosql databases live up to their promise? URL: http://www.leavcom.com/pdf/NoSQL.pdf. Accessed 28.3.2023.

[8] How discord stores trillions of messages. URL: https://discord.com/blog/how-discord-stores-trillions-of-messages. Accessed 30.5.2023.

[9] Scylladb documentation. URL: https://www.scylladb.com/product/technology/. Accessed 30.5.2023.

[10] Discord documentation. URL: https://discord.com/developers/docs/reference. Accessed 30.5.2023.

[11] Telegram documentation. URL: https://telegram.org/privacy?setln=fa. Accessed 30.5.2023.

[12] End-to-end encryption (e2ee) definition. URL: https://www.techtarget.com/searchsecurity/definition/end-to-end-encryption-E2EE. Accessed 31.5.2023.

[13] Telegram database library documentation. URL: https://core.telegram.org/tdlib. Accessed 4.6.2023.

[14] Telegram api introduction. URL: https://towardsdatascience.com/introduction-to-the-telegram-api-b0cd220dbed2. Accessed 4.6.2023.

[15] Postman: What is an api? URL: https://www.postman.com/what-is-an-api/. Accessed 4.6.2023.

[16] About restful apis. URL: https://restfulapi.net/. Accessed 21.2.2023.

[17] About grpc. URL: https://grpc.io/docs/what-is-grpc/faq/. Accessed 21.2.2023.

[18] Json introduction. URL: https://www.json.org/json-en.html. Accessed 21.2.2023.

[19] Web frameworks for python. URL: https://wiki.python.org/moin/WebFrameworks. Accessed 21.2.2023.

[20] Flask web framework. URL: https://www.fullstackpython.com/flask.html. Accessed 21.2.2023.

[21] M. G. (2018) Flask Web Development: Developing Web Applications with Python. O'Reilly, 4 p.

[22] A realistic look at python web frameworks. URL: https://suade.org/12-requests-per-second-with-python. Accessed 21.2.2023.

[23] About postgresql. URL: https://www.postgresql.org/about. Accessed 21.2.2023.

[24] Github documentation. URL: https://docs.github.com/en/get-started/learning-about-github/githubs-products. Accessed 30.5.2023.

[25] Sqlalchemy source code. URL: https://github.com/sqlalchemy/sqlalchemy. Accessed 28.3.2023.

[26] Dwyer G. Aggarwal S. S.J. (2017) Flask: Building Python Web Services. Packt Publishing, 237 p.

[27] Flask-restful documentation. URL: https://flask-restful.readthedocs.io/en/latest/. Accessed 28.3.2023.

[28] Jsonschema documentation. URL: https://json-schema.org/. Accessed 28.3.2023.

[29] Flask-expects-json package documentation. URL: https://pypi.org/project/flask-expects-json/. Accessed 28.3.2023.

[30] Flask-babel documentation. URL: https://python-babel.github.io/flask-babel/. Accessed 28.4.2023.

[31] Flask-limiter documentation. URL: https://flask-limiter.readthedocs.io/en/stable/. Accessed 28.4.2023.

[32] Flask-limiter strategies. URL: https://flask-limiter.readthedocs.io/en/stable/strategies.html. Accessed 28.4.2023.

[33] Flask deploying to production. URL: https://flask.palletsprojects.com/en/2.3.x/deploying/. Accessed 28.4.2023.

[34] Gunicorn documentation. URL: https://gunicorn.org/. Accessed 30.5.2023.

[35] Jmeter documentation. URL: https://jmeter.apache.org/. Accessed 30.5.2023.

[36] Sentry documentation. URL: https://docs.sentry.io/platforms/python/?original_referrer=https%3A%2F%2Fsentry.io%2F. Accessed 31.5.2023.

9. APPENDICES

Appendix 1 HTTP requests JSON validation formats

```
chatlists_post_schema = {
    'type' : 'object',
    'properties' : {
        'receiver' : {'type' : 'string'},
        'course_space' : {'type' : 'string'},
        'topic' : {'type' : 'string'},
        'message' : {'type' : 'string'},
        'language' : {'type' : 'string'}
    },
    'required' : ['receiver', 'course_space',
                  'topic', 'message', 'language']
}
```

```
chatlists_get_schema = {
    'type' : 'object',
    'properties' : {
        'language' : {'type' : 'string'}
    },
    'required' : ['language']
}
```

```
chat_post_schema = {
    'type' : 'object',
    'properties' : {
        'message' : {'type' : 'string'},
        'linked_to' : {'type' : 'string'},
        'language' : {'type' : 'string'}
    },
    'required' : ['message', linked_to, 'language']
}
```

```
chat_get_schema = {
    'type' : 'object',
    'properties' : {
        'language' : {'type' : 'string'}
    },
    'required' : ['language']
}
```

Appendix 2 Database tables with SQLAlchemy

```
class ChatModel(db.Model):

    __tablename__ = 'chats'

    id = db.Column(db.Integer, primary_key=True, unique=True)
    chat_id = db.Column(db.String(80), unique=True)
    topic = db.Column(db.String(80), nullable=False)
    course_space = db.Column(db.String(80))
    type = db.Column(db.String(50))
    language = db.Column(db.String(2))
    messages = db.relationship('MessageModel',
                               backref='chats',
                               lazy='dynamic',
                               order_by='MessageModel.
                                         timestamp')
    participants = db.relationship('Participants',
                                   lazy='dynamic')


class MessageModel(db.Model):

    __tablename__ = 'messages'

    id = db.Column(db.Integer, primary_key=True, unique=True)
    chat_id = db.Column(db.Integer, db.ForeignKey('chats.id'),
                                                  nullable=False)
    sender = db.Column(db.String(50), nullable=False)
    message = db.Column(db.String(500), nullable=False)
    is_read = db.Column(db.Boolean, default=False)
    timestamp = db.Column(db.DateTime, default=datetime.
                          datetime.utcnow)
    linked_to = db.Column(db.Integer,
                          db.ForeignKey('messages.id'))
```

```
class Participants(db.Model):

    __tablename__ = 'participants'

    id = db.Column(db.Integer, primary_key=True,
                   unique=True)
    chatId = db.Column(db.Integer, db.ForeignKey('chats.id'),
                       nullable=False)
    userId = db.Column(db.String(50), nullable=False)
```

Appendix 3 Work distribution

Group Member	Stage 1 Hours	Contributions
Johannes Lampela	36	Studying and researching technologies, planning, creating presentation and writing thesis
Tommy Meriläinen	32	Studying and researching technologies, planning, creating presentation and writing thesis
	Stage 2	
Johannes Lampela	47	Designing and implementing the project, creating presentation and write thesis
Tommy Meriläinen	42	Designing and implementing the project, studying databases and write thesis
	Stage 3	
Johannes Lampela	28	Coding the project, writing thesis and creating presentation
Tommy Meriläinen	25	Coding the project, writing thesis and creating presentation
	Stage 4	
Johannes Lampela	114	Coding and testing the code, writing thesis, creating presentation and refining the project
Tommy Meriläinen	119	Testing the code, writing thesis, creating presentation and evaluating
	Total	
Johannes Lampela	225	
Tommy Meriläinen	218	