

Master Thesis



Czech  
Technical  
University  
in Prague

**F3**

Faculty of Electrical Engineering  
Department of Computer Science

## Design and implementation of a multiple time-slot appointment calendar for effective planning of consultations

**Bc. Anton Striapan**

Supervisor: RNDr. Ladislav Serédi  
May 2023



## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Striapan** Jméno: **Anton** Osobní číslo: **478877**  
Fakulta/ústav: **Fakulta elektrotechnická**  
Zadávací katedra/ústav: **Katedra počítačů**  
Studijní program: **Otevřená informatika**  
Specializace: **Softwarové inženýrství**

## II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

**Návrh a implementace kalendáře s vícenásobnými časovými úseky pro schůzky pro efektivní plánování konzultací**

Název diplomové práce anglicky:

**Design and implementation of a multiple time-slot appointment calendar for effective planning of consultations**

Pokyny pro vypracování:

Provedte rešerši existujících sdílených kalendářů s ohledem na plánování vícenásobných událostí (např. řadu konzultací) v daném časovém úseku. Analyzujte způsoby zlepšení uživatelské přívětivosti rozhraní kalendáře a navrhnete architekturu pro její implementaci. Vytvořte prototyp sdíleného kalendáře s implementací oboustranného potvrzení schůzek. Zaměřte se na interoperabilitu s dalšími kalendáři (i.e. Google Calendar, Microsoft Outlook) prostřednictvím API pro vytvoření vícenásobných událostí („aggregated time slots“). Umožněte sdílení těchto událostí mezi populárními aplikacemi typu kalendář. Implementované řešení otestujte: do build pipeline doplňte jednotkové testy, přidejte integrační testy a specifické testy pro front-end. Provedte neformální uživatelské testování důležitých funkcionalit řešení, jako například vytvoření účtu, přihlášení, dále vytvoření, import, export a nastavení viditelnosti kalendáře, vytvoření a správu konzultací jak ze strany pedagoga tak i studenta. Diskutujte výsledky uživatelských testů, a rovněž slabé a silné stránky Vašeho řešení.

Seznam doporučené literatury:

Scheduling Meetings in Distance Learning, Jian Wang, Changyong Niu & Ruimin Shen  
[https://link.springer.com/chapter/10.1007/978-3-540-76837-1\\_63](https://link.springer.com/chapter/10.1007/978-3-540-76837-1_63)  
Time Management - Meaning and its Importance, MSG MANAGEMENT STUDY GUIDE,  
<https://www.managementstudyguide.com/time-management.htm>  
Meeting Scheduling: Face-to-Face, Automatic Scheduler, and Email Based Coordination, Bongsik Shin & Kunihiro Higa,  
[https://www.tandfonline.com/doi/abs/10.1207/s15327744joc1502\\_3](https://www.tandfonline.com/doi/abs/10.1207/s15327744joc1502_3)

Jméno a pracoviště vedoucí(ho) diplomové práce:

**RNDr. Ladislav Serédi kabinet výuky informatiky FEL**

Jméno a pracoviště druhého(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **15.02.2023**

Termín odevzdání diplomové práce: **26.05.2023**

Platnost zadání diplomové práce: **22.09.2024**

RNDr. Ladislav Serédi  
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Mgr. Petr Páta, Ph.D.  
podpis děkana(ky)

### III. PŘEVZETÍ ZADÁNÍ

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

\_\_\_\_\_  
Datum převzetí zadání

\_\_\_\_\_  
Podpis studenta

## Acknowledgements

I am very thankful to RNDr Ladislav Serédi, my mentor, for his significant contribution to my writing assignment. His valuable comments throughout the semester have been immensely helpful in refining my work. I would also like to express my gratitude to all those who have supported and aided me in this journey. I am grateful to my family and friends for their unwavering support and encouragement throughout my academic journey. Their love and belief in me have constantly motivated and inspired me. Lastly, I would like to thank all of those people who have contributed to the development of the resources and tools used or referenced in my project. Their efforts have enabled me to conduct my research effectively and efficiently.

## Declaration

I declare that this work is all my work and I have cited all sources I have used in the bibliography. Prague, May , 2023

## Abstract

The thesis begins with a comprehensive literature review of existing calendar solutions for multi-slot appointments, particularly for planning consultations. This includes analyzing their features, usability, and user experience. This analysis identifies pain points and areas for improvement in existing solutions. The thesis proposes an architecture for a new shared calendar solution that addresses these issues and provides a more intuitive and user-friendly experience. The proposed calendar solution includes multiple time-slot reservations, two-way appointment approval, and easy interoperability with other calendar tools such as Google Calendar and Outlook. This is achieved by implementing an API interface for communication with other calendar tools, allowing for the creation of persistent aggregated time slots and easy import and export of appointments in an appropriate format for other popular calendar tools. The thesis also includes the implementation of a prototype of the proposed calendar solution. This prototype demonstrates the functionality and usability of the proposed architecture, as well as how it improves the user experience of calendar solutions for multi-slot appointments. Finally, the thesis concludes by discussing the results and recommendations for future work. The proposed calendar solution is easy to use, efficient, and interoperable with other calendar tools, providing a better user experience for scheduling multi-slot appointments, particularly for planning consultations.

**Keywords:** Calendar solutions, Multi-slot appointments, Consultations, Usability, User experience, Architecture, User-friendly, Two-way appointment approval, Interoperability, API interface, Aggregated time slots, Import/export, Prototype, Functionality, Testing, Feedback, Efficiency, Recommendations

**Supervisor:** RNDr. Ladislav Serédi  
Praha, Resslova 9, E-429 (vchod Karlovo náměstí 13)

## Abstrakt

Práce začíná obsáhlým přehledem literatury o existujících řešeních kalendáře pro schůzky s více časovými úseky, zejména pro plánování konzultací. Zahrnuje to analýzu jejich funkcí, použitelnosti a uživatelských zkušeností. Tato analýza identifikuje nedostatky a oblasti pro zlepšení stávajících řešení. Práce navrhuje architekturu nového řešení sdíleného kalendáře, které tyto problémy řeší a poskytuje intuitivnější a uživatelsky přívětivější prostředí. Navržené řešení kalendáře zahrnuje vícenásobné rezervace časových slotů, obousměrné schvalování schůzek a snadnou interoperabilitu s dalšími nástroji kalendáře, jako jsou Kalendář Google a Outlook. Toho je dosaženo implementací rozhraní API pro komunikaci s jinými kalendářovými nástroji, které umožňuje vytváření trvalých agregovaných časových slotů a snadný import a export schůzek ve vhodném formátu pro jiné populární kalendářové nástroje. Součástí práce je také implementace prototypu navrhovaného kalendářového řešení. Tento prototyp demonstruje funkčnost a použitelnost navržené architektury a také způsoby, kterými zlepšuje uživatelské prostředí kalendářových řešení pro schůzky s více časovými sloty. V závěru práce jsou diskutovány výsledky a doporučení pro další práci. Navržené řešení kalendáře je snadno použitelné, efektivní a interoperabilní s jinými nástroji kalendáře a poskytuje lepší uživatelský zážitek při plánování schůzek s více sloty, zejména při plánování konzultací

**Klíčová slova:** Řešení kalendáře, Schůzky s více časovými sloty, Konzultace, Použitelnost, Uživatelská zkušenost, Architektura, Uživatelsky přívětivé, Oboustranné schvalování schůzek, Interoperabilita, Rozhraní API, Sdružené časové sloty, Import/export, Prototyp, Funkčnost, Testování, Zpětná vazba, Efektivita, Doporučení

**Překlad názvu:** Návrh a implementace kalendáře s více násobnými časovými úseky pro schůzky pro efektivní plánování konzultací

# Contents

<b>1 Introduction</b>	<b>1</b>	6.2.1 Architecture description . . . .	53
1.1 Non Functional requirements . . . .	2	6.2.2 Data Validation . . . . .	54
1.1.1 Common requirements . . . . .	2	6.2.3 Calendar Export . . . . .	58
1.1.2 Student requirements . . . . .	2	6.2.4 CalendarEvent creation . . . . .	59
1.1.3 Staff requirements . . . . .	2	<b>7 Authorization</b>	<b>63</b>
1.2 User scenarios . . . . .	3	7.1 Internal Authorization . . . . .	63
1.2.1 Login and connect CTU account	3	7.1.1 Frontend . . . . .	64
1.2.2 Connect new calendar . . . . .	4	7.1.2 Backend . . . . .	64
1.2.3 Create calendar event . . . . .	4	7.1.3 Database . . . . .	65
1.2.4 Assign on event . . . . .	5	7.2 External Authorization . . . . .	66
<b>2 Done research</b>	<b>7</b>	7.2.1 Google API . . . . .	66
2.1 Calendar Server . . . . .	8	7.2.2 CTU API . . . . .	68
2.1.1 Zuul OAAS . . . . .	9	<b>8 Testing</b>	<b>71</b>
2.1.2 Sirius API . . . . .	11	8.1 Backend . . . . .	71
2.2 SSO Gate . . . . .	15	8.2 Frontend . . . . .	72
2.3 KOS API . . . . .	15	8.3 Gitlab CI/CD . . . . .	73
2.4 Google Calendar API . . . . .	17	<b>9 Conclusion</b>	<b>75</b>
2.4.1 Concepts overview . . . . .	17	<b>A Application setup instructions</b>	<b>77</b>
2.4.2 Calendars events . . . . .	18	A.1 How to run and serve application	78
2.4.3 Sharing & attendees . . . . .	19	A.1.1 With backend . . . . .	78
<b>3 Proposed architecture</b>	<b>21</b>	A.1.2 With mocked backend . . . . .	79
3.1 Database . . . . .	22	<b>B Glossary</b>	<b>81</b>
3.2 Backend . . . . .	23	<b>C List of Abbreviations</b>	<b>83</b>
3.3 Frontend . . . . .	26	<b>D Bibliography</b>	<b>85</b>
<b>4 Implementation</b>	<b>29</b>		
4.1 Backend . . . . .	29		
4.2 Frontend . . . . .	30		
4.2.1 Architecture and Business Logic			
Description . . . . .	30		
4.2.2 UI/UX . . . . .	31		
<b>5 User service</b>	<b>37</b>		
5.1 Database . . . . .	37		
5.1.1 Architecture description . . . .	38		
5.1.2 ORM implementation . . . . .	39		
5.1.3 Repository . . . . .	40		
5.2 Backend . . . . .	41		
5.2.1 Architecture description . . . .	41		
5.2.2 Data Validation . . . . .	42		
5.2.3 Eureka Service visibility . . . .	44		
<b>6 Calendar service</b>	<b>45</b>		
6.1 Database . . . . .	45		
6.1.1 Architecture description . . . .	46		
6.1.2 ORM implementation . . . . .	48		
6.1.3 Repositories . . . . .	50		
6.2 Backend . . . . .	52		



## Figures

1.1 Flowchart diagram of login and subsequent connecting of CTU account process . . . . .	3	4.7 Frontend Calendar Page, Week View Mode . . . . .	35
1.2 Flowchart diagram of connecting the new calendar to user account process . . . . .	4	4.8 Frontend Calendar Page, Calendar Event Creation . . . . .	36
1.3 Flowchart diagram of creating a new calendar event for the user with role STAFF process . . . . .	4	5.1 UML diagram of the UserService database . . . . .	38
1.4 Flowchart diagram of how can student assign himself for timeslot .	5	5.2 UserService layered architecture	41
2.1 Home page of AppManager service	9	6.1 Architecture of calendar database	46
2.2 Project settings page of AppManger service . . . . .	10	7.1 Frontend login page . . . . .	64
2.3 Available additional services for mine project . . . . .	10	7.2 Google console homepage . . . . .	66
2.4 Sirius API documntation . . . . .	11	7.3 Google Cloud setting up environment . . . . .	67
2.5 Sirius API request example . . . . .	12	7.4 Google cloud credentials . . . . .	67
2.6 Sirius API response example . . . . .	13	7.5 CTU login page . . . . .	68
2.7 SSO gate authorization page . . . . .	15	8.1 Gitlab CI/CD . . . . .	73
2.8 KOS API documntation . . . . .	16		
2.9 KOS API user controller documentation . . . . .	16		
2.10 Google concept of shared calendars diagram . . . . .	17		
2.11 Relationships between the user, calendar, and event entities . . . . .	18		
3.1 Entity relationship diagram of the database . . . . .	22		
3.2 Ordinary microservices architecture . . . . .	24		
3.3 Mine version of microservices architecture for this project . . . . .	25		
3.4 Ngrx state management lifecycle diagram . . . . .	27		
4.1 Frontend sign up page . . . . .	31		
4.2 Frontend login page . . . . .	32		
4.3 Frontend User Config age . . . . .	33		
4.4 Frontend Calendar Config Page .	34		
4.5 Frontend Calendar Page, Calendar Is Not Private . . . . .	34		
4.6 Frontend Calendar Page, Month View Mode . . . . .	35		

## Tables

2.1 Example of the interface that will be mocked .....	14
5.1 User entity mapped on the Java class with ORMl .....	39
5.2 Location entity mapped on the Java class with ORMl .....	39
5.3 Implementation of UserRepository for database data accessl.....	40
5.4 Example of aggregated validation function for registration datal ....	42
5.5 Example of validation function that adds validation for name field	43
5.6 Eureka dependency .....	44
5.7 Eureka dependency .....	44
6.1 ORM implementation for calendar entity .....	48
6.2 ORM implementation for calendarDate entity .....	49
6.3 ORM implementation for calendarEvent entity .....	49
6.4 ORM implementation for calendar entity .....	50
6.5 CalendarRepository implementation .....	50
6.6 CalendarEventRepository implementation .....	51
6.7 Example of aggregated validation function for calendar data.....	54
6.8 Example of aggregated validation function for calendar event creation data .....	56
6.9 Example of aggregated validation function for calendar timeslot data	57
6.10 Example of export calendar service .....	58
6.11 Example createEvent function .	60
6.12 Example create TimeSlots function .....	61
7.1 Example of user repository .....	65
7.2 Example uri returned after code retrieval from CTU auth server ...	69
7.3 Example JSON response with token.....	69



# Chapter 1

## Introduction

The goal of this master thesis is to study user cases of interaction with shared Calendars and create a new tool with new functionality which is not used in another solution. The future tool will combine Calendars from different sources, sources can be as cloud Calendar or imported from \*.ICal format file. The tool will be connected to the CTU info system, which will allow retrieving user roles, such as **STUDENT** or **STAFF**, to load their calendars, for future work with them. The default user will be able to create his own sets of Calendars, set them a level of visibility, **PUBLIC** or **PRIVATE**, if **PRIVATE** type was selected the calendar won't be visible for other users of the application, it can be visible only if they were added to the whitelist. The type of Calendar can be changed in Calendar settings. It will allow creating shared Events, in our case main type of Events is teacher-student consultations, with better user experience, Events can be split into time slots, and users, mostly with role **STUDENT** can book them. An additional functionality, which can be not implemented is to create different time areas, depending on the user with role **STAFF** schedule, with different possibilities of additional consultation time zones: green, yellow, and red. For example, students want to offer a consultation for the teacher and the teacher has a lesson at that time, and the area will be background painted red. If the teacher will create a calendar event for consultation and define how many time slots it will be divided it will be background painted with green. And all other times will be background painted yellow. But for this type of consultation, the student should apply, and add a note, about why the teacher should approve it. So after the creation of this consultation student should wait for approval. It can be accepted or declined. Calendars can be also exported into \*.ICal format file, and transferred into another more preferred calendar.

## ■ 1.1 Non Functional requirements

In the next chapter, I will describe the basic nonfunctional requirements through the application. They will be divided into three different groups:

- Common requirements
- Unique student requirements
- Unique staff requirements

### ■ 1.1.1 Common requirements

Common requirements for all types of users will be

- Create account
- Log in
- Login to CTU workspace
- CRUD on Calendar
- CRUD on Event
- Assign the time slot of the Event to an authorized user
- Import external Calendars
- Export Calendar in \*.Ical format

### ■ 1.1.2 Student requirements

For users with role STUDENT will be the next requirements

- Suggest an extra time slot for a consultation, not during the defined consultation time period
- Assign or unassign the time slot of consultation to the Event to an authorized user

### ■ 1.1.3 Staff requirements

As staff users will be possible, to

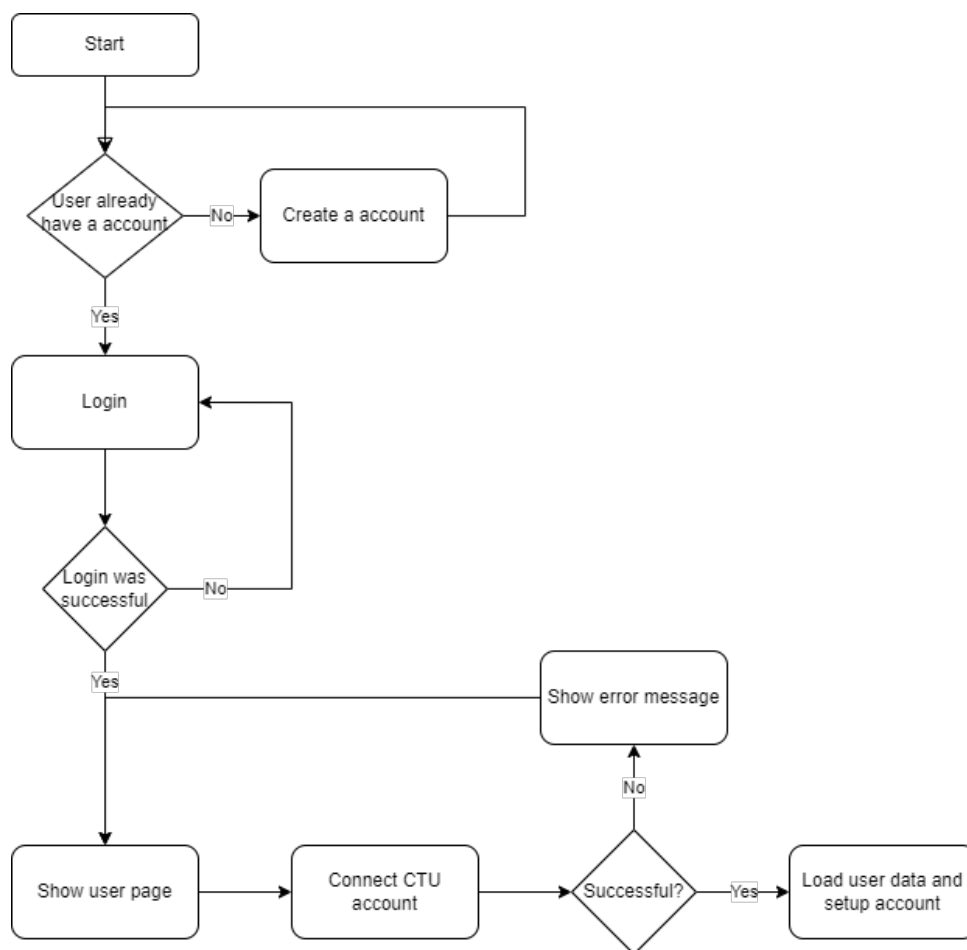
- To approve extra consultation
- Create consultation Event
- Setup consultation settings such as duration, amount of time slots on which this event will be divided, amount of students who can be assigned into one slot

## 1.2 User scenarios

In this section, I will show diagrams of some use case scenarios for a better understanding of application features. Diagrams will be placed in a logical order, so each next diagram can reuse described processes above.

### 1.2.1 Login and connect CTU account

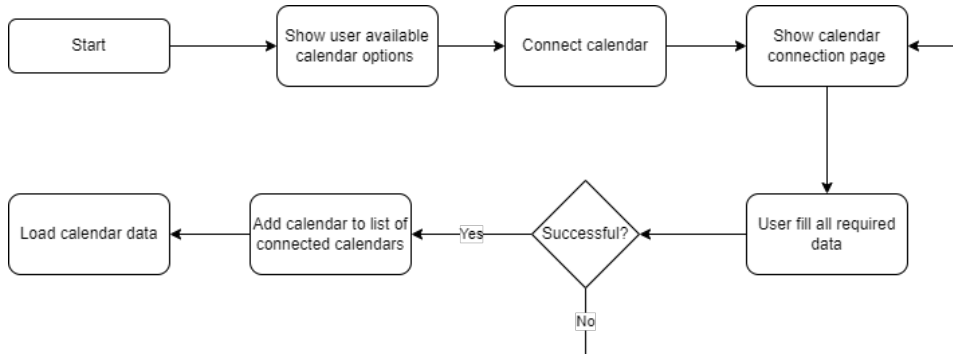
This diagram shows us the process of login into or creating an account in the application, then the user can connect the CTU account to it. After that depending on user data loaded from CTU servers, the system will detect the role of the user, STUDENT or STAFF and finishes the user account setup.



**Figure 1.1:** Flowchart diagram of login and subsequent connecting of CTU account process

### 1.2.2 Connect new calendar

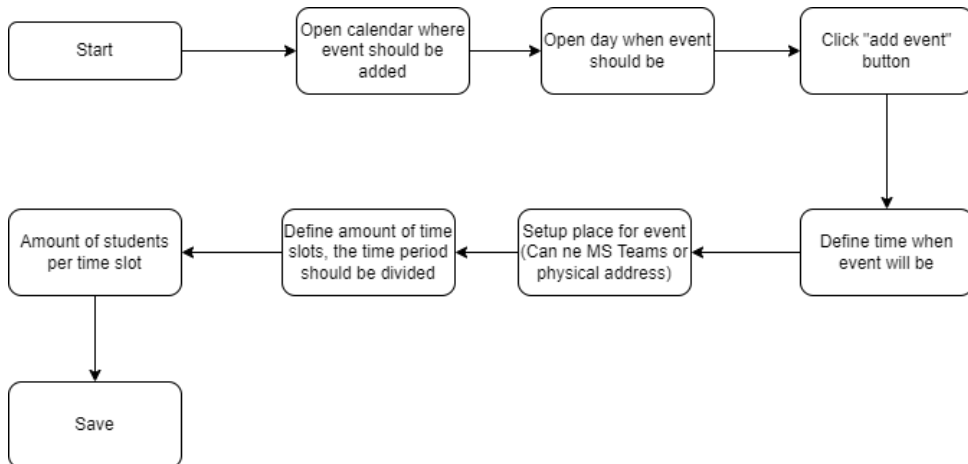
This diagram shows the process of adding a new calendar to the user, and if the calendar was added successfully the data from it will be displayed to the user.



**Figure 1.2:** Flowchart diagram of connecting the new calendar to user account process

### 1.2.3 Create calendar event

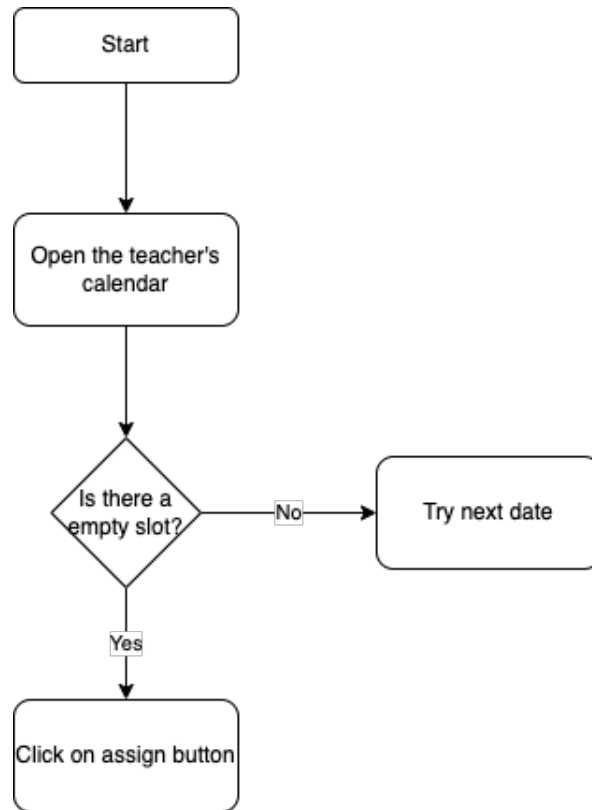
Here I will show how users with role STAFF can create a time slot for consultation



**Figure 1.3:** Flowchart diagram of creating a new calendar event for the user with role STAFF process

### 1.2.4 Assign on event

This diagram shows how students can assign themselves to available timeslots for consultation.



**Figure 1.4:** Flowchart diagram of how can student assign himself for timeslot







## Chapter 2

### Done research

In this chapter, I will describe the first steps which should be done before starting to design the system architecture or implementation phase. At this moment I expect to find out how to efficiently parse user information from CTU open API, calendar events, which technologies to use, and the approximate architecture of the application. In the next chapter I will figure out how it can be implemented calendar server and a general idea of implementations.

## 2.1 Calendar Server

After basic research implementation of a calendar server can be done in two ways, the first is to use one of the existing solutions ( CalDAV ) servers and the second one is to implement a new calendar server for my own purposes.

Calendaring Extensions to WebDAV, or CalDAV, is an Internet standard allowing a client to access and manage calendar data along with the ability to schedule meetings with users on the same or on remote servers. It lets multiple users share, search and synchronize calendar data in different locations. It extends the WebDAV (HTTP-based protocol for data manipulation) specification and uses the iCalendar format for the calendar data. The access protocol is defined by RFC 4791. Extensions to CalDAV for scheduling are standardized as RFC 6638. The protocol is used by many important open-source applications. Known CalDAV servers are: Google Calendar, Apple, Bedework, Chandler Server, DavMail, Oracle Siebel CRM

Implementation of my own calendar server will cost much longer time but can be more efficient in terms of functionality and exploring time. The basic requirements for a calendar server are:

- Allows storing the events.
- Allows merging events into a defined sequence.
- Allows editing and deleting events.
- Allows setting the visibility and accessibility of calendars and calendar events.
- Allows importing data from third-party APIs.
- Allows exporting data into different formats.

Unfortunately, my attempts to test and evaluate the various APIs under consideration were hindered by privacy concerns and the lack of available contacts. Despite these challenges, I was able to make some progress towards identifying a suitable API for my proposed solution. Specifically, I was fortunate to receive documentation from the individual responsible for the KOS API, which provided valuable insights into its functionality and potential applicability.

Additionally, I learned that FelSight is currently developing an API under the Sirius platform, which would provide access to calendar events. Although this information could not be incorporated into my project within the required timeframe, it could be leveraged to create a dummy connector interface that utilizes the appropriate date formats and API calls. By doing so, we could save significant implementation time in the future when integrating this service.

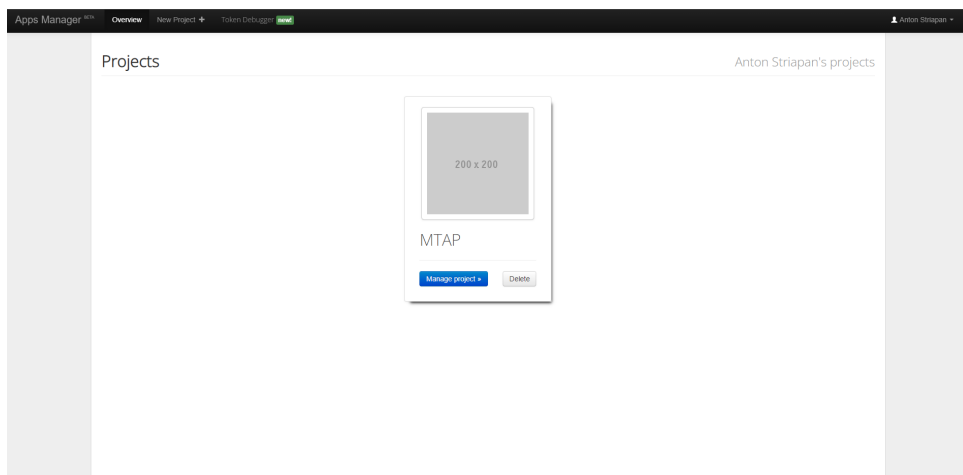
Despite the setbacks encountered during my research, I remain confident that the insights gained through this process will prove valuable in identifying the most suitable API for my proposed solution. Moving forward, I plan to continue exploring other potential APIs, leveraging my newfound knowledge to evaluate their functionality and applicability. Ultimately, the goal is to develop a robust and efficient solution that meets the needs of our users, while also adhering to the highest standards of data privacy and security.

### ■ 2.1.1 Zuul OAAS

CTU develops its own OAuth 2.0 authorization server (hereinafter referred to as OAAS), which is called Zuul OAAS. It's open-source and you can find it on GitHub. It is a separate OAAS, i.e. it is not part of the own service exposing the API (resource server), but multiple separate services can use one (remote) OAAS that issues and validates tokens. OAAS runs at <https://auth.fit.cvut.cz/> and can be used by anyone from the CTU academic community (not only FIT). The addresses of the individual "endpoints" are as follows:

- Authorization Endpoint
- Token Endpoint
- Check Token Endpoint

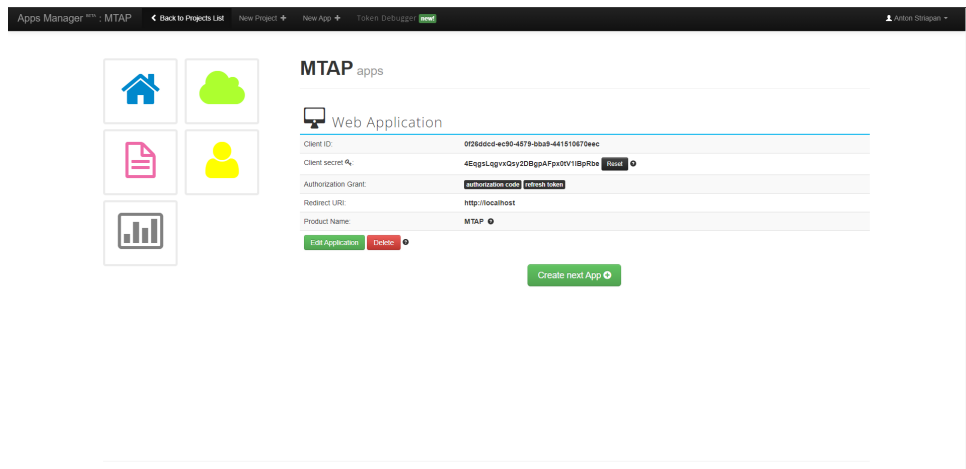
The next step will be to log into AppsManager and create a new project.



**Figure 2.1:** Home page of AppManager service

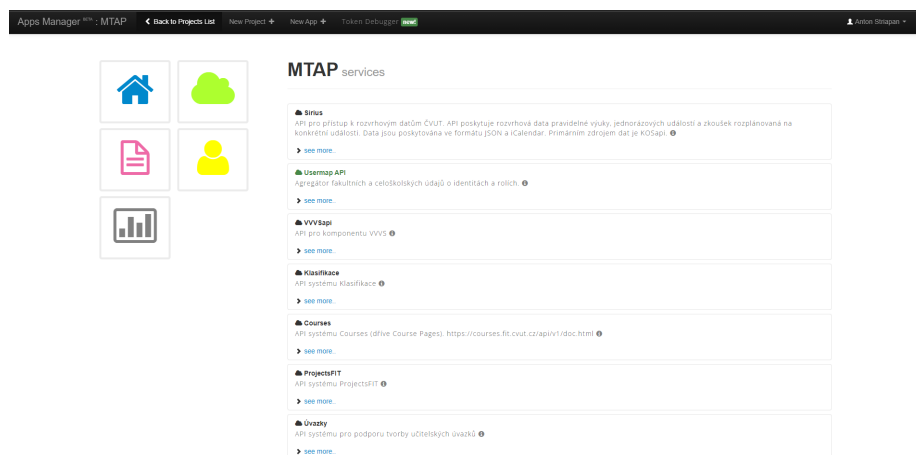
## 2. Done research

After the successful creation of the project, we will receive `client_id` and `client_secret` which will be used in future APIs calls.



**Figure 2.2:** Project settings page of AppManger service

In the services tab, we can enable or request access for specific CTU API



**Figure 2.3:** Available additional services for mine project

After that preparation is done we can start the authentication and data-retrieving processes, which will be described in the chapters below.

## 2.1.2 Sirius API

So the best API to communicate with to receive data about calendar events such as lectures, labs, and other university events is Sirius API. By utilizing Sirius API, we can obtain the latest information on all the upcoming university events, including their dates, times, and locations. The API also allows us to filter the data based on various criteria, such as the event type or the department offering it. This flexibility ensures that we will receive only the information that is relevant to us, rather than having to sift through a mountain of irrelevant data. In the figure below we can see the Swagger styled API description which provides us a clear vision of what endpoints it provides, how to trigger them and what data should be sent, and what data will be received in the response.

CTU Sirius API API documentation version v1  
<https://sirius.fit.cvut.cz/api/v1>

Endpoint	Method
/events	GET
/events/{eventId}	GET
/people/{username}	GET
/people/{username}/events	GET
/teachers/{username}/events	GET
/rooms/{kosid}/events	GET
/courses/{courseCode}/events	GET

Search results in sidebar:

- /events
- /people/{username}
- /teachers/{username}/events
- /rooms/{kosid}/events
- /courses/{courseCode}/events
- /faculties/{facultyId}
- /schedule\_exceptions
- /semesters
- /search

**Figure 2.4:** Sirius API documntation

In the next figure, we can see an example of what parameters can be added to the request to receive more accurate data.

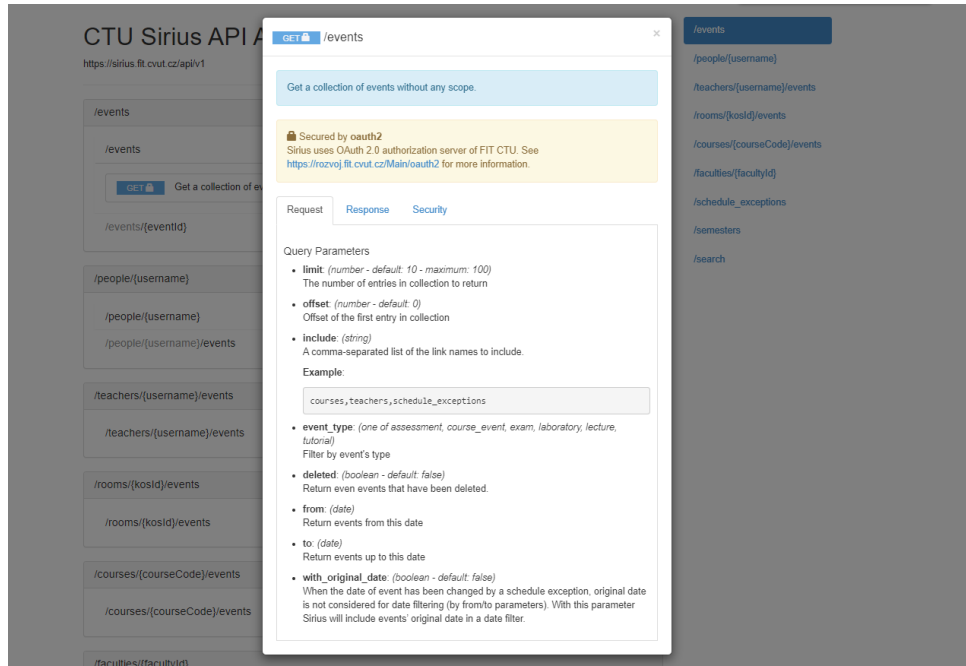


Figure 2.5: Sirius API request example

The figure below is an illustration of sample data that is intended to be transmitted as a response. This set of data is used as a framework for defining the interface that specifies the type and structure of the expected response payload. Subsequently, this interface is employed to generate mock objects that replicate the behavior and format of the actual response data.

The interface serves as a contract between the client and the server, ensuring that the data is transmitted in a structured and consistent manner. The interface specifies the data elements, their types, and the structure of the payload. In essence, it defines the blueprint for the expected response payload.

By defining this interface beforehand, we can effectively model the behavior and structure of the data that will be transmitted, and subsequently use that interface to generate mock objects that simulate the behavior and structure of the actual response data.

These mock objects are especially useful in our situation where there is insufficient time to develop and integrate the complete set of system functionality. In this case, mock objects can serve as a stand-in or substitute for the actual data, allowing us to test and verify the overall system behavior and integration, without the need for fully functional components.

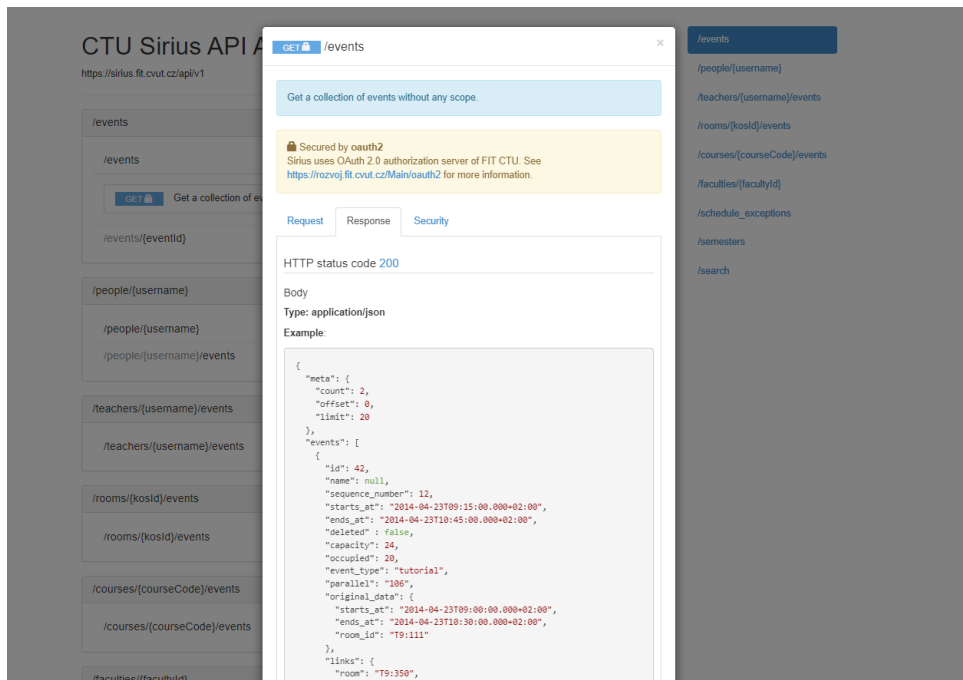


Figure 2.6: Sirius API response example

The provided code below is a Java class called `EventsResponse` that represents a response from an API call. It uses the Lombok library to generate `getter`, `setter`, and `setter` methods, reducing the amount of boilerplate code required.

The `EventsResponse` class has two instance variables: `meta` and `events`. `meta` represents metadata about the response, including the number of items returned, the offset used for pagination, and the maximum number of items that can be returned. `events` is a list of objects that represent individual events.

There are three nested classes defined within the `EventsResponse` class: `Meta`, `Event`, `OriginalData`, and `Links`.

The `Meta` class contains the metadata variables mentioned earlier. The `Event` class contains information about an individual event, such as its ID, name, start and end times, capacity, and occupancy. Additionally, the `Event` class contains two other nested classes, `OriginalData` and `Links`, which contain further information about the event.

The `OriginalData` class contains data specific to the event, such as the start and end times, and the ID of the room in which it will take place. Finally, the `Links` class contains links to other resources related to the event, such as the room, course, and lists of teachers and students involved.

---

```
import java.util.List;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
public class EventsResponse {

    private Meta meta;
    private List<Event> events;

    @Data
    @NoArgsConstructor
    public static class Meta {
        private int count;
        private int offset;
        private int limit;
    }

    @Data
    @NoArgsConstructor
    public static class Event {
        private int id;
        private String name;
        private int sequence_number;
        private String starts_at;
        private String ends_at;
        private boolean deleted;
        private int capacity;
        private int occupied;
        private String event_type;
        private String parallel;
        private OriginalData original_data;
        private Links links;
    }

    @Data
    @NoArgsConstructor
    public static class OriginalData {
        private String starts_at;
        private String ends_at;
        private String room_id;
    }

    @Data
    @NoArgsConstructor
    public static class Links {
        private String room;
        private String course;
        private List<String> teachers;
        private List<String> students;
        private List<Integer> applied_exceptions;
    }
}
```

**Table 2.1:** Example of the interface that will be mocked



## 2.2 SSO Gate

SSO means Single Sign-On, you can get access to all participant applications at one time by entering your username and password. This principle is a lot on the university websites like KOS, Moodle, and CourseWare. It allows access to other systems of the CTU information system (IS CTU) for 10 hours without re-entering the name and password. For user authentication, it is necessary to use CTU Password.

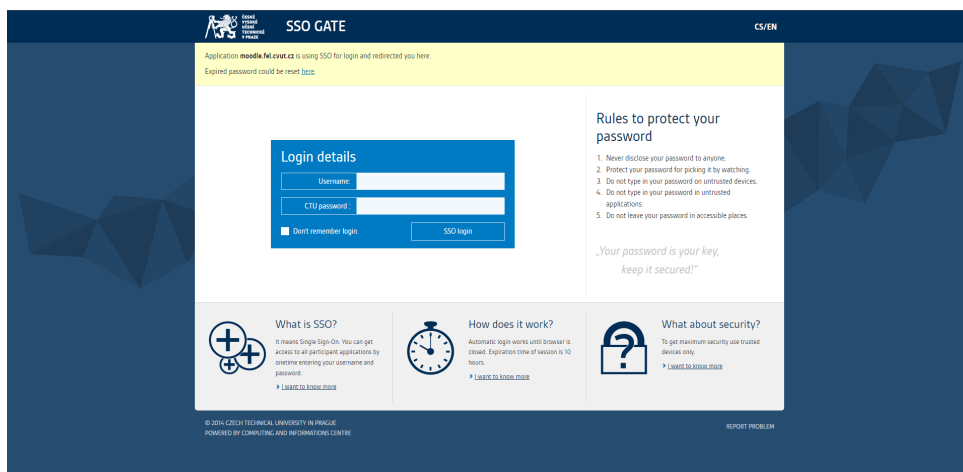


Figure 2.7: SSO gate authorization page

Throughout the entire duration of the project, no individual contacts or documentation were discovered. This can be attributed to the fact that authorization logic was implemented on other servers (OAuth), while my application merely serves as a redirect to that resource. The responsibility of managing the authorization process falls under a different entity, while my application's task is to securely store the access token and utilize it for request authorization.

## 2.3 KOS API

KOSapi provides an application interface as a RESTful web service that accesses a selected portion of KOS data.

It enables and supports the creation of school and student applications that require online application access to education-related data for their operation. It eliminates the need to process exports, the constant duplication of all data and the hassle of maintaining it. It builds on the proven concepts of the Web as a distributed environment of interrelated information.

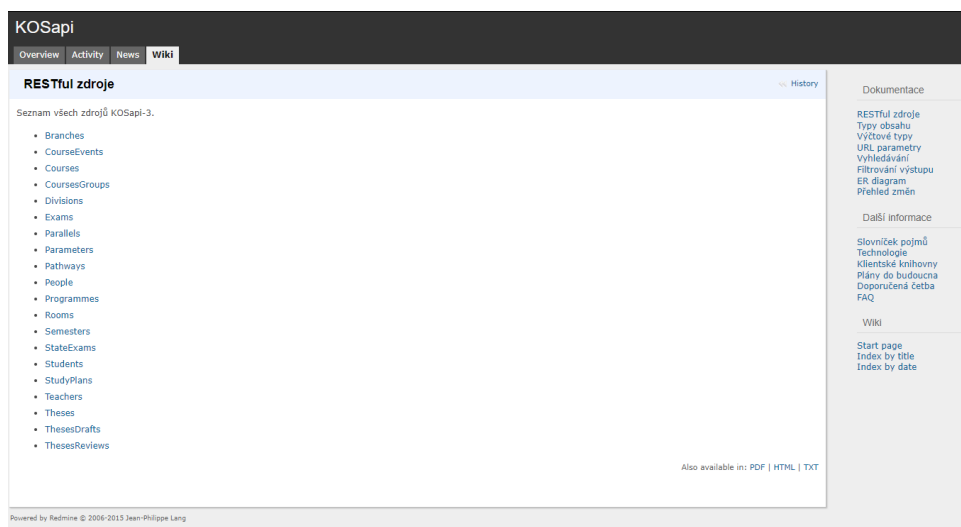


Figure 2.8: KOS API documntation

As opposed to the Sirius AP, KOS API doesn't provide good-looking and well-formed documentation from where would be visible what data structure should be sent and what will be received.

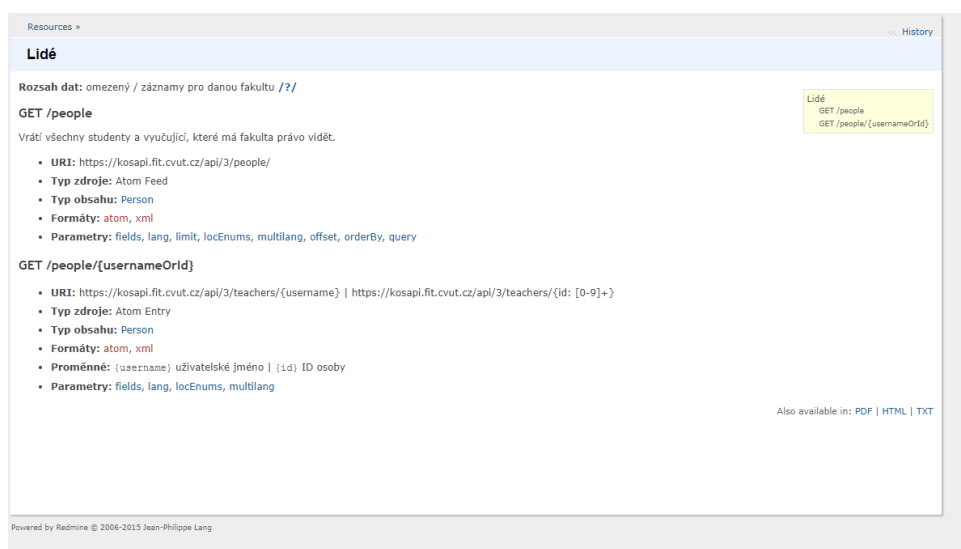


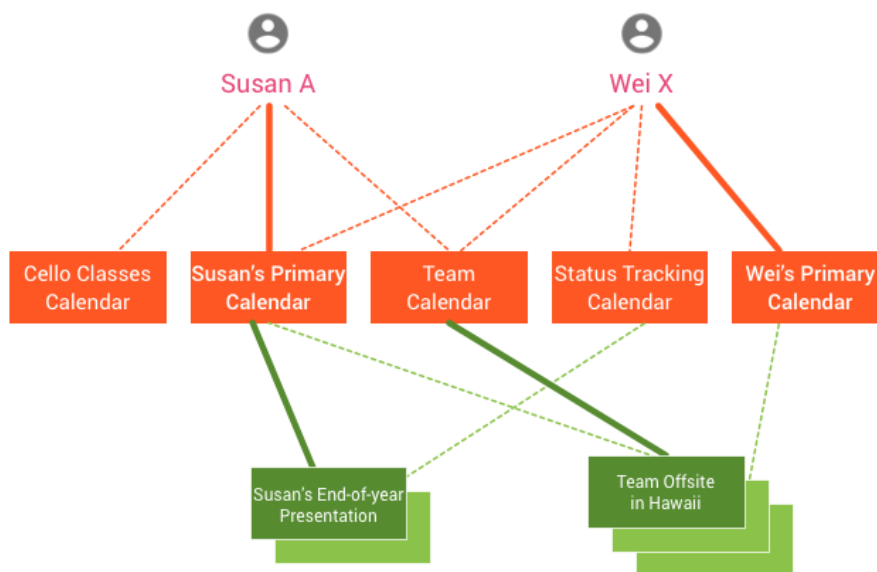
Figure 2.9: KOS API user controller documentation

## 2.4 Google Calendar API

The Google Calendar API is a RESTful API that can be accessed through explicit HTTP calls or via the Google Client Libraries. The API exposes most of the features available in the Google Calendar Web interface.

### 2.4.1 Concepts overview

Each Calendar user is associated with a primary Calendar and a number of other Calendars that they can also access. Users can create Event and invite other users, as shown in the following diagram:



**Figure 2.10:** Google concept of shared calendars diagram

This example shows two users, Susan A and Wei X. Each has a primary Calendar and several other associated Calendars. The example also shows two Events: an end-of-year presentation and a team offsite. Here are some facts shown in the diagram:

- Susan's Calendar list includes her primary Calendar as well as Calendars for her team and cello lessons.
- Wei's Calendar list includes his primary Calendar as well as the team Calendar, a status tracking Calendar, and Susan's primary Calendar.
- The end-of-year presentation event shows Susan as the organizer and Wei as an attendee.
- The team off-site in Hawaii Event has the team Calendar as an organizer (meaning it was created in that Calendar) and copied to Susan and Wei as attendees.

## 2.4.2 Calendars events

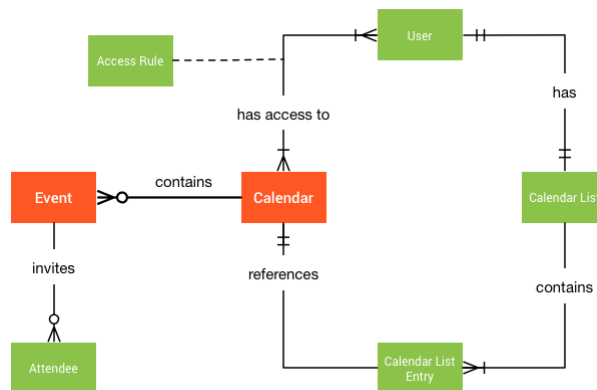
A calendar is a collection of related events, along with additional metadata such as **summary**, **default time zone**, **location**, etc. Each calendar is identified by an ID which is an **email address**. Calendars can have multiple owners. An event is an object associated with a specific date or time range. Events are identified by a unique ID. Besides a start and end date-time, events contain other data such as **summary**, **description**, **location**, **status**, **reminders**, **attachments**, etc. Google Calendar supports single and recurring Events:

- A single Event represents a unique occurrence.
- A recurring Event defines multiple occurrences.

Events may also be timed or all day:

- A timed Event occurs between two specific points in time. Timed Events use the `start.DateTime` and `end.DateTime` fields to specify when they occur.
- An all-day Event spans an entire day or consecutive series of days. All-day Events use the `start.date` and `end.date` fields to specify when they occur. Note that the `timezone` field is insignificant for all-day Events.

Events have a single organizer which is the Calendar containing the main copy of the Event. Events can also have multiple attendees. An attendee is usually the primary Calendar of an invited user. The following diagram shows the conceptual relationship between Calendar, Events, and other related elements:



**Figure 2.11:** Relationships between the user, calendar, and event entities

### ■ 2.4.3 Sharing & attendees

There are two ways to share Calendar and Event data with others. Firstly, you can share an entire Calendar, with a specified level of access. For example, you can create a team Calendar, and then do things like:

- Grant all members of your team the right to add and modify Events in the Calendar
- Grant your boss the right to see the Events on your Calendar
- Grant your customers the right to only see when you are free or busy, but not the details of the Events

You can also adjust the access to individual Events on the shared Calendar. Alternatively, you can invite others to individual Events on your Calendar. Inviting someone to an Event will put a copy of that event on their Calendar. The invitee can then accept or reject the invitation, and to some Event also modify their copy of the Event — for example, change the color it has in their Calendar, and add a reminder.





## Chapter 3

### Proposed architecture

When research is done we can move into the system architecture phase. From the information that we got we can make next conclusions:

- System will have a frontend and backend parts.
- System will need an internal database for storing sensitive data.
- Backend should be time efficient with big data sets and multiple users.
- Backend should be divided into micro-services architecture for reducing the point of failure.
- Backend should have authorization, for secure communication.

Parts of the system will be described more in the chapters below:

- Frontend: Angular, NGRX, Rx js, Angular Material
- Backend: Java Spring Boot, Micro-services
- Database: MongoDB

### 3.1 Database

We don't have any restrictions for the database, so the choice of the database is not that important. For this project, I would like to use a NoSQL database such as MongoDB. MongoDB is a source-available cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with optional schemes. The core of the MongoDB developer data platform is a multi-cloud database service built for resilience, scale, and the highest levels of data privacy and security. MongoDB Atlas provides a reliable and user-friendly platform for managing data, allowing for deployment in various regions on multiple cloud providers, and automatically implementing security measures for optimal performance. It is a flexible solution that can scale to meet global, multi-region, or multi-cloud needs. I will show one of the first possible future database views in the following diagram.

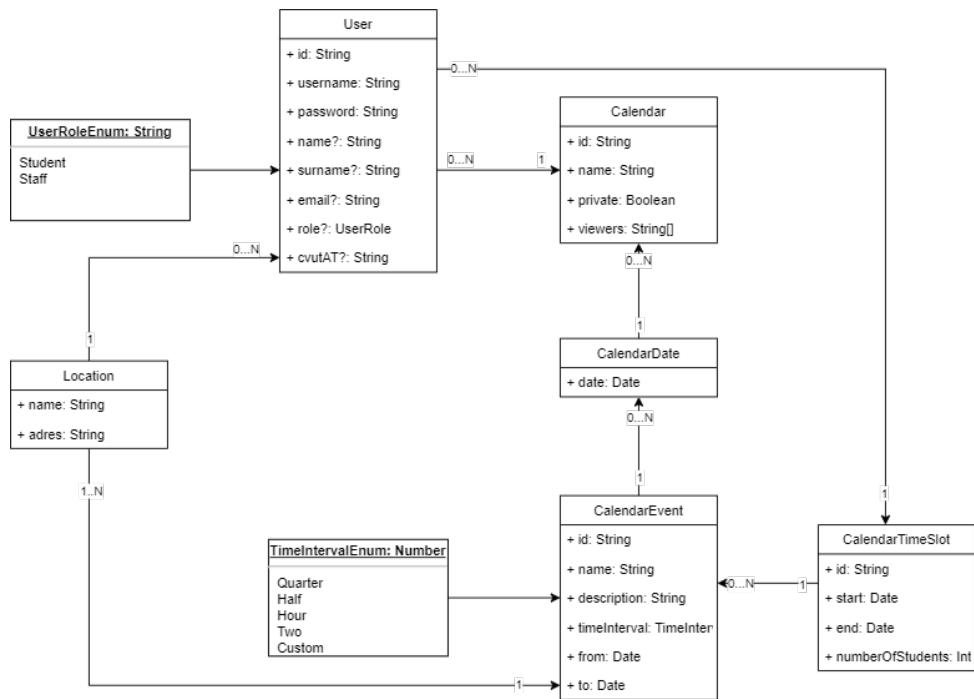


Figure 3.1: Entity relationship diagram of the database



As we can see there are 6 basic database entities, which will be used with ORM:

- Location
- User
- Calendar
- CalendarDate
- CalendarEvent
- CalendarTimeSlot

## ■ 3.2 Backend

As I described above, there are basic requirements for backend

- Backend should be time efficient with big data sets and multiple users.
- Backend should be divided into micro-services architecture for reducing the point of failure.
- Backend should have authorization, for secure communication.

For the back-end side application was chosen Java Spring Boot was. I stopped my view on Spring Boot because of the features and benefits it offers as given here:

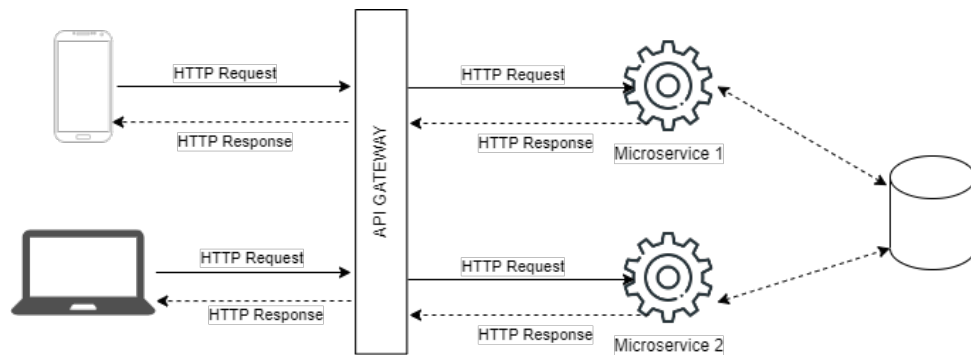
- It provides a flexible way to configure Java Beans, XML configurations, and Database Transactions.
- It provides powerful batch processing and manages REST endpoints.
- In Spring Boot, everything is auto-configured; no manual configurations are needed.
- It offers an annotation-based spring application
- Eases dependency management
- It includes Embedded Servlet Container

### 3. Proposed architecture

As backend architecture was chosen microservices architecture. Microservices is an architectural style that structures an application as a collection of services that are:

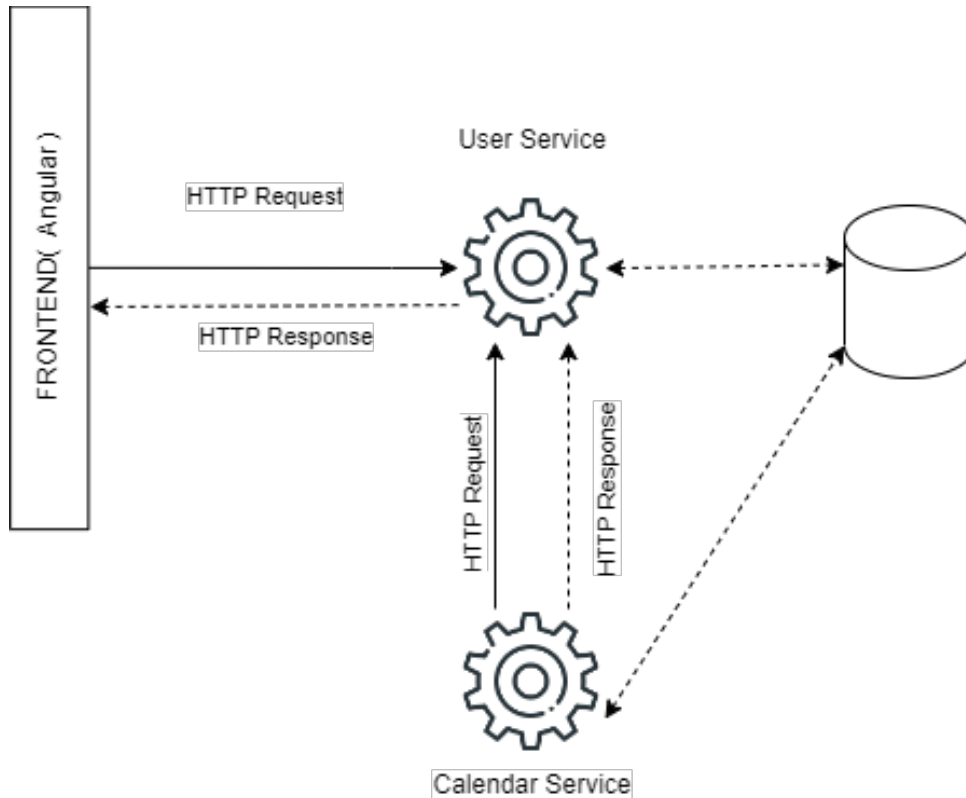
- Distributed coarse-grained services provide functionality for each other.
- Shared user interface.
- Shared database (logical partitioning).
- Flexible, domain-driven.
- Highly maintainable, and testable.
- Loosely coupled.
- Independently deployable.
- Organized around business capabilities.
- This architecture can be owned by a small team, in my case it will be only me.

The microservice architecture enables the rapid, frequent, and reliable delivery of large, complex applications. It also enables an organization to evolve its technology stack. In the diagram below I will show you the basic micro-services structure.



**Figure 3.2:** Ordinary microservices architecture

But due to the specifics of my implementation, which will be described later, I changed a bit of basic architecture but the idea is still the same. Changes can be found in the next diagram.



**Figure 3.3:** Mine version of microservices architecture for this project

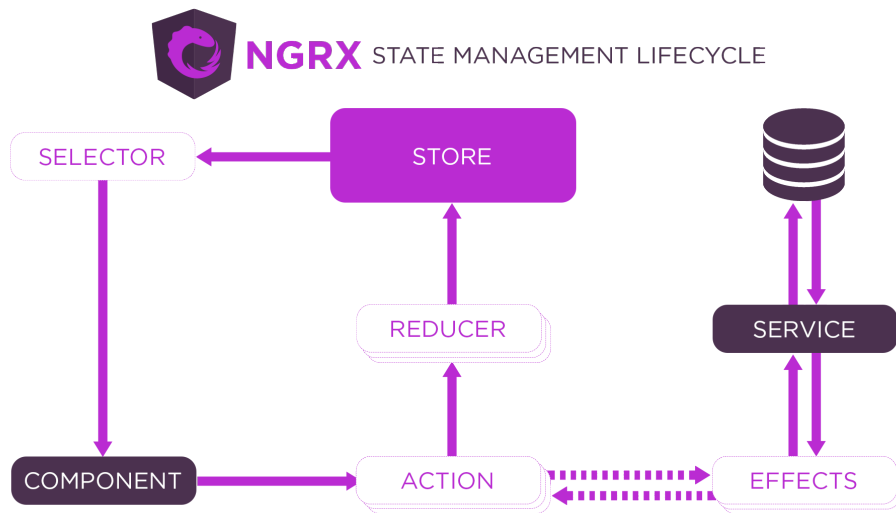
The idea was to connect Api Gateway gateway and user service, due to the amount of should be done work and the limited time. But they can be split in the future. The main functionality of Api Gateway is to accept and filter all HTTP requests, hide all other backend services, so they won't be visible and accessible from the global network and redirect requests to different non-visible services according to the data which should be returned. But for the correct functioning of this architecture, we need to create a local network, so our backend services will be hidden, and implement authorization methods on each service, to protect from unexpected sensitive data stilling. And this will cost a lot of time which I don't have for now. My solution is, to add a gateway functionality to the user service, so it will handle all requests, filter them, perform authorization of the user, and if it will be successful returns or request the data from another service. It will be more time and work effectively for my case but will keep the basic ideas, functionality, and pros from microservices architecture.

## 3.3 Frontend

For the frontend side was chosen Angular, because I have a lot of experience with it, so the development will be done much faster and with a lower amount of problems during the development. Angular is a development platform, built on TypeScript. As a platform, Angular includes:

- A component-based framework for building scalable web applications.
- A collection of well-integrated libraries that cover a wide variety of features, including routing, forms management, client-server communication, and more.
- A suite of developer tools to help you develop, build, test, and update your code.

With Angular, you're taking advantage of a platform that can scale from single-developer projects to enterprise-level applications. Angular is designed to make updating as straightforward as possible, so take advantage of the latest developments with minimal effort. For the design of UI components Angular Material was chosen because it provides a lot of implemented solutions that can be reused, reworked, or extended for my application. As state manager, I decided to choose NGRX library. NGRX is a library for Angular that implements the Redux pattern. It is used for managing state in Angular applications. The Redux pattern is a way of managing the application state in a predictable and consistent manner. It is based on the principles of immutability and unidirectional data flow. NGRX provides a set of tools for implementing this pattern in Angular applications, including a centralized store for the application state, actions and reducers for the modifying state, and selectors for the querying state. NGRX also provides integration with the RxJS library, which allows for powerful and efficient handling of asynchronous data streams. This allows NGRX to handle complex, real-world scenarios such as handling network requests and handling complex user interactions. NGRX is a popular library among Angular developers as it helps to make their application more predictable and easy to test by centralizing the state and logic of the application and making it more manageable. The data flow is described in the diagram below.



**Figure 3.4:** NgRx state management lifecycle diagram

Also, as second main technology will be used `Nx monorepo`, which brings us a different project structure. A monorepo is a code management strategy in which multiple projects are stored in a single repository. In an `Nx monorepo`, these projects are managed using the `Nx workspace` tool. There are many benefits to using an `Nx monorepo` for your project.

One key advantage is improved code sharing and reuse. Because all projects are stored in a single repository, it's easy to share code between them. This means that if we have multiple projects that require similar functionality, we can write the code once and reuse it across all the projects. This can significantly reduce development time and increase code quality.

Another advantage of an `Nx monorepo` is simplified configuration and tooling. Because all projects are managed using the `Nx workspace` tool, we only need to configure tooling once for the entire monorepo. This can reduce the amount of time spent on setup and configuration and can help ensure consistency across projects.

In addition, an `Nx monorepo` can make it easier to maintain a consistent coding style across projects. Because all projects are managed using the same tooling and configuration, it's easier to enforce coding standards and ensure consistency.

An `Nx monorepo` can also make it easier to manage dependencies. Because all projects are stored in a single repository, it's easier to manage dependencies between them. This can help reduce the likelihood of dependency conflicts and make it easier to upgrade dependencies when necessary.

Furthermore, an `Nx monorepo` can help improve testing and code quality. Because all projects are stored in a single repository, it's easier to test them together and ensure that changes made to one project don't break another. This can help improve overall code quality and reduce the likelihood of bugs and regressions.

Another benefit of an `Nx monorepo` is improved deployment and release

management. Because all projects are managed using the same tooling and configuration, it's easier to deploy and release them together. This can help reduce the likelihood of deployment issues and make it easier to roll back changes if necessary.

An **Nx monorepo** can also make it easier to scale my project. Because all projects I stored in a single repository, it's easier to add new projects and features as my project grows. This can help reduce the overhead associated with managing multiple repositories and make it easier to keep track of everything.

In addition, an **Nx monorepo** can make it easier to refactor my code. Because all projects are stored in a single repository, it's easier to make changes to shared code without having to update multiple repositories. This can help reduce the time and effort required for refactoring.

Finally, an **Nx monorepo** can help reduce development costs. Because all projects are stored in a single repository, it's easier to manage them and reduce duplication of effort. This can help reduce development time and costs and improve overall project efficiency.

In conclusion, there are many benefits to using an **Nx monorepo** for this project. From improved code sharing and reuse to simplified configuration and tooling, also **Nx monorepo** can help improve project efficiency.

## Chapter 4

### Implementation

This chapter thoroughly explains how a calendar application has been implemented. It encompasses the frontend and backend design, as well as database design. The primary emphasis is on the collaboration of the two backend services: the `CalendarService` and the `UserService`. In this chapter, we will explain the process of storing and retrieving data for each service. Moreover, we will guide you through the implementation of different features in a calendar application, such as user registration, login, data modification, calendar administration, and event scheduling from both front-end and back-end perspectives. In addition to functional requirements, we will cover non-functional requirements like performance, availability, and security. We will then clarify how they are incorporated into the back-end, front-end, and database. Ultimately, this chapter will provide you with a comprehensive comprehension of the design and architecture of the calendar application.

#### 4.1 Backend

The backend of this application consists of two parts: the `CalendarService` and the `UserService`. These services have their own databases and are responsible for different functions of the application. The `CalendarService` manages users' calendars, events, and time slots. Users can create, update, and delete their calendars and events. They can also assign other users to specific time slots and import external calendars to keep all their appointments and events in one place.

The `CalendarService` is capable of handling a large number of users and events without slowing down, making it scalable. In contrast, the `UserService` takes care of managing user data and authentication processes, such as user sign-up, login, and account information. It also ensures that user data is kept secure, encrypted, and adheres to the application's privacy requirements.

The purpose of the `UserService` is to make sure that users can access their accounts at all times, even if there are problems. The application has a prototype that demonstrates how these services work, including features like sign-up, log-in, managing user data, updating calendars and events, assigning users to specific time slots, importing external calendars, exporting calendars,

and more.

The application features are specifically designed to allow testing of all use cases and functionality. Aside from the functional requirements, there are also several non-functional requirements that the application must meet. These include high performance to ensure quick access to user data, high availability to allow users access to their data anytime, and security to protect and encrypt user data during transmission and while at rest.

## ■ 4.2 Frontend

### ■ 4.2.1 Architecture and Business Logic Description


The frontend part of the application is built using Angular with an nx monorepo structure. The application is organized into different libraries that handle various parts and features. The four basic library types used are data-access, component, feature, and utils. The frontend provides a range of functionalities for the users. Users can sign up or sign in to the application, and their access tokens are stored securely in the local storage of the browser. This allows for seamless authentication and authorization throughout the user's session. The application supports CRUD operations for both user-related entities and calendar-related entities. Users can create, read, update, and delete records associated with users and calendars. These operations are implemented using ngrx, which provides a convenient and efficient way to manage state and handle actions. To enhance authentication capabilities, the frontend integrates with Google and CTU. Users can authenticate using their Google accounts, which is achieved through the integration with the Google Calendar API. Additionally, the integration with the CTU API enables users to authenticate using CTU credentials. One of the core functionalities of the frontend is the ability to fetch calendars from the Google Calendar API and the Sirius API. This allows users to access and manage their calendars within the application. The fetched calendar data can be displayed, edited, and synchronized with the backend. In order to facilitate development and testing, the frontend also includes a `mockProviderService`. This service allows the application to run without a backend, providing simulated responses for API calls and enabling a smooth development experience.



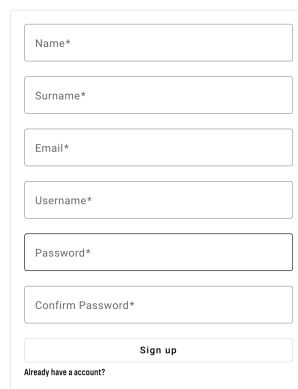
## 4.2.2 UI/UX

### Sign up

In this figure we can see the frontend sign-up page, where the user should fill all input forms because all of them are required, also frontend will validate each input by a different rule, for example for email field will be used email regex patterns to check the if the user provides a valid email address. After successful registration user will be notified and redirected to the login page to process the next steps in user authentication.

ConsultEase 

---



The image shows a sign-up form with the following fields and elements:

- Name\*
- Surname\*
- Email\*
- Username\*
- Password\*
- Confirm Password\*
- Sign up button
- Already have an account? link

**Figure 4.1:** Frontend sign up page

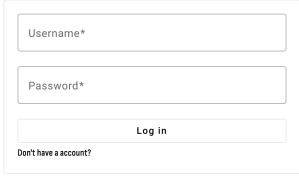
## Login

In this figure, there is a screenshot of the login page of the application. It also has all fields marked as required and after data validation `/login` endpoint will be triggered to retrieve `access token`. If the response code is 200 the user will be redirected to the `/home` frontend page where he will be able to check his calendar.

---

ConsultEase 

---



The screenshot shows a login form with the following elements:

- A text input field labeled "Username\*" with an asterisk indicating it is required.
- A text input field labeled "Password\*" with an asterisk indicating it is required.
- A button labeled "Log in".
- A link labeled "Don't have an account?" located below the "Log in" button.

**Figure 4.2:** Frontend login page

## User Config Page

If the user clicks on the cogwheel on the **header** he will be redirected to the `/settings` page. By default to user will be displayed user config page, where he can change basic personal data and connect external calendars such as Google or CTU. Same as for the login or sign-up process all input fields will be validated on the frontend part also. On the right side, the user has a list of his calendars, if he clicks on it, the calendar config page will be opened.

The screenshot displays the 'User Config Page' in a web application. At the top, the header includes the 'ConsultEase' logo and a settings gear icon. The page is divided into two main sections. On the left, a user profile for 'Anton Striapan' is shown, along with a list of connected calendars: 'CVUT calendar' and 'Google calendar'. On the right, there is a form for connecting a CTU calendar. The form includes a 'Connect CTU' button and several input fields: 'Name' (filled with 'Anton'), 'Surname' (filled with 'Striapan'), 'Email' (filled with 'striaant@fel.cvut.cz'), and 'Username' (filled with 'striaant'). A 'Save' button is located at the bottom of the form.

**Figure 4.3:** Frontend User Config age

## Calendar Config Page

On this page, the user can change the visibility of his calendar, and the name of the calendar, by default external imported calendars are created with a private option.

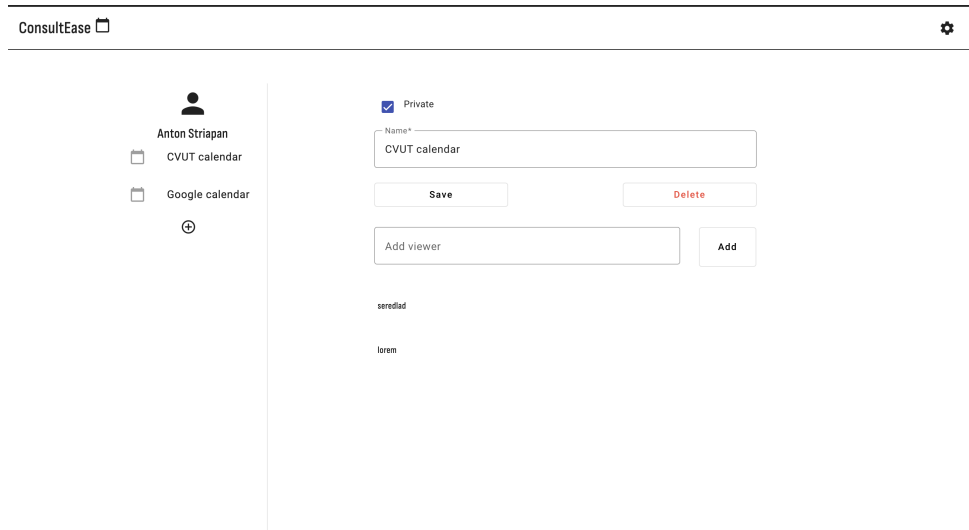


Figure 4.4: Frontend Calendar Config Page

Also, the user can add or remove viewers for the calendars, if the user has a calendar marked with private `false` this block won't be shown.

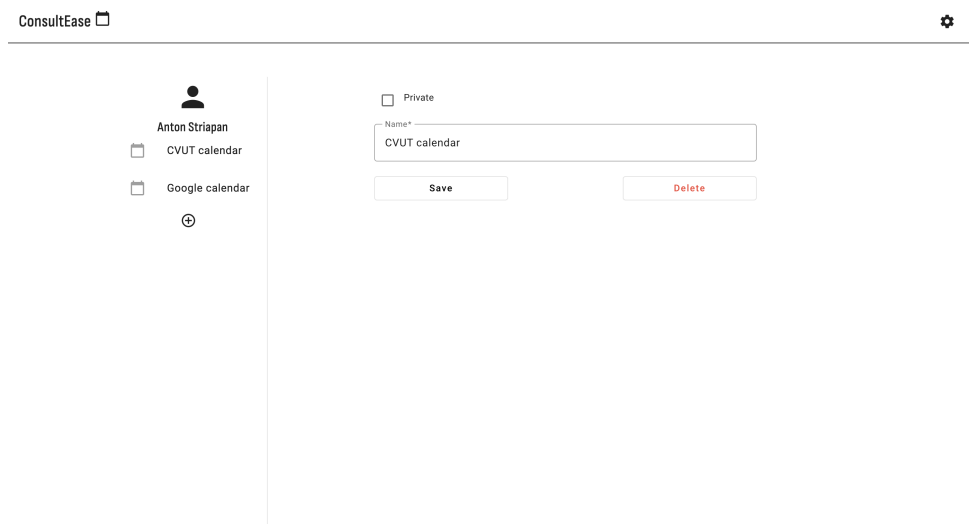
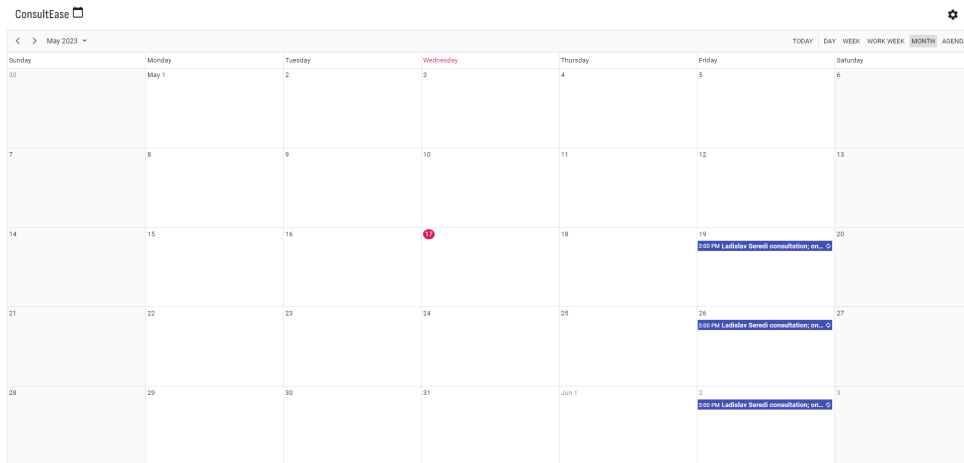


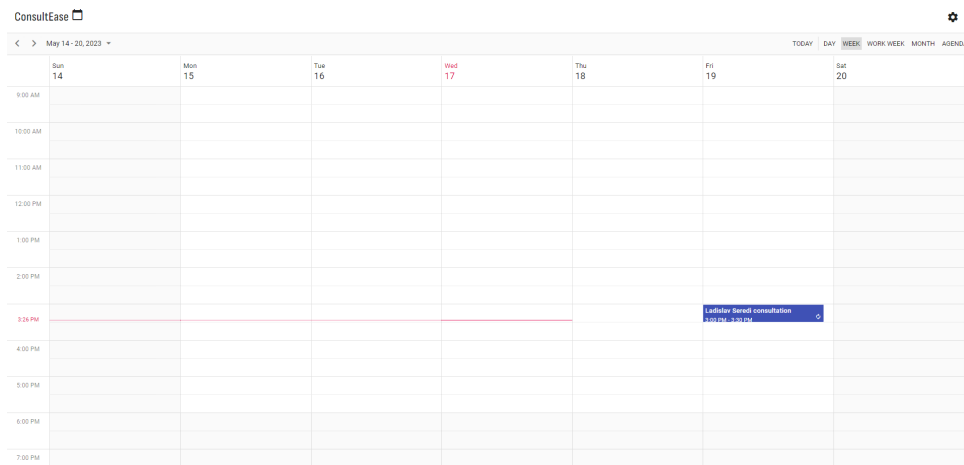
Figure 4.5: Frontend Calendar Page, Calendar Is Not Private

## Calendar Page

This is the main component of this application and allows the user to perform all necessary operations with the calendar itself. Display options can be changed in the right top corner between - Day, Week, and Month. Screenshots of Week and Month viewer mode will be bellow.



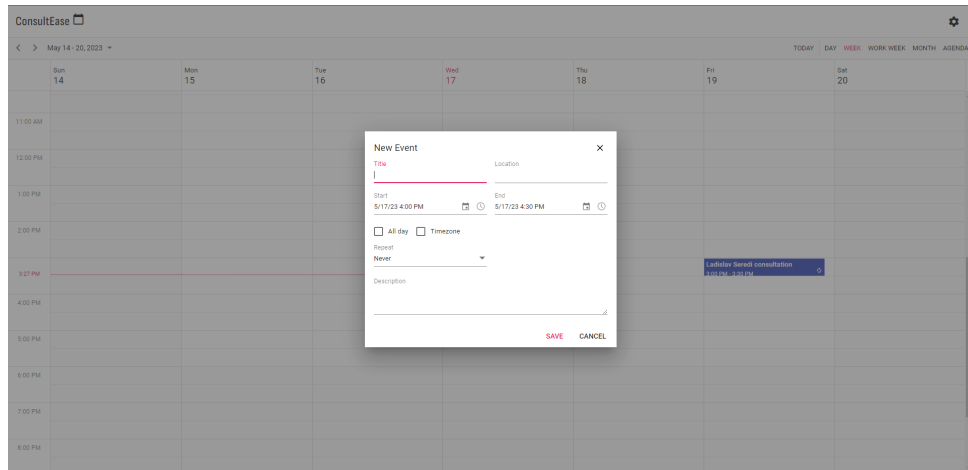
**Figure 4.6:** Frontend Calendar Page, Month View Mode



**Figure 4.7:** Frontend Calendar Page, Week View Mode

## Calendar Event Creation

This figure shows us a calendar event creation UI. Same as other forms of the application they have local validation. And after that, the process will be started on the frontend part and finished on the backend.



**Figure 4.8:** Frontend Calendar Page, Calendar Event Creation

## Chapter 5

### User service

In this section, we will dive into the implementation details of the `UserService`, which serves as the root of the application and handles user authentication and management. This service manages user data and redirects HTTP requests to the `CalendarService`, which manages calendars and events. Additionally, the `UserService` handles service visibility by `Eureka`, a service discovery tool that allows services to find and communicate with each other.

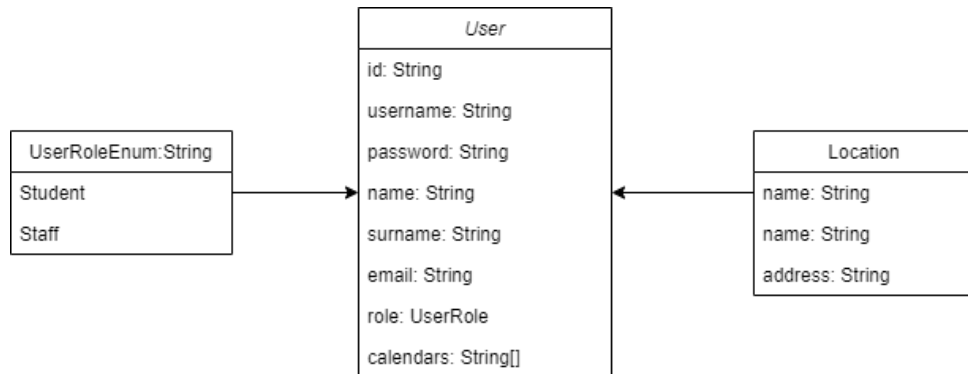
We will also cover the database design for this service, which stores user data and supports various application functionalities. Specifically, we will describe how user sign-up and login processes are implemented, and how user data is stored and retrieved from the database. Additionally, we will cover the implementation of functionality for updating user data, such as email addresses, Access tokens, and other settings. We will also discuss how the `UserService` interacts with the `Eureka` service registry to register and discover other services in the application.

Finally, we will discuss the non-functional requirements of the `UserService`, such as security and performance, and how they are implemented in the database design and service discovery mechanisms. By the end of this section, you will have a comprehensive understanding of the implementation details of the `UserService` and the associated database and service discovery tools.

### 5.1 Database

In this section, we will discuss the database architecture for the `UserService`, which is responsible for managing user data and authentication. A well-designed database is critical to the success of any application, and the `UserService` is no exception. The database must be able to efficiently store and retrieve user data, support various functionalities of the application, and maintain data integrity and security. We will begin by discussing the requirements for the database, including the types of data that need to be stored and how it will be accessed. We will then cover the design decisions for the database schema, such as table structure, relationships between tables, and data types. Additionally, we will describe the database management system used for the `UserService` and how it handles tasks such as data indexing, query optimization, and data backup and recovery.

### 5.1.1 Architecture description



**Figure 5.1:** UML diagram of the UserService database

The diagram shows the database schema for the **UserService**, which is responsible for managing user data and authentication. The central entity in the schema is the **User** entity, which has attributes:

- `id`
- `username`
- `password`
- `name`
- `surname`
- `email`

The `id` attribute is a string that uniquely identifies each user in the system. The **User** entity also has a relationship with the **UserRoleEnum** entity, which is an enumeration that defines two values: `STUDENT` and `STAFF`. This relationship indicates the role of the user in the system. Additionally, the **User** entity can store the user's location with three attributes:

- `id`
- `name`
- `address`

The **User** entity has an array of strings with `calendarIds`. This attribute stores an array of strings that represent the unique identifiers of the calendars associated with the user. In the final representation of the application, the `cvutAC` parameter has been removed, explanations of this decision will be given in the next chapters. This UML database diagram provides a clear representation of the schema for the **UserService**, which is critical to the success of the application.



## 5.1.2 ORM implementation

---

```

@Document
@Data
@NoArgsConstructor
public class User {

    @Id
    private String id;
    private String username;
    private String password;
    private String name;
    private String surname;
    private String email;
    private LocalDateTime createdAt;
    private List<Role> roles;
    private List<Location> locations;
    private List<Integer> calendars;
    private List<Integer> calendarsTimeSlots;
}

```

---

**Table 5.1:** User entity mapped on the Java class with ORMI

---

```

@Document
@Data
@NoArgsConstructor
public class Location {
    @Id
    private String id;
    private String name;
    private String address;
}

```

---

**Table 5.2:** Location entity mapped on the Java class with ORMI

These Java classes are designed to work with an Object-Relational Mapping (ORM) framework that connects to MongoDB databases. To indicate that a class should be treated as a MongoDB document, the `@Document` annotation from the Spring Data MongoDB library is used. This annotation, combined with `MongoTemplate`, associates the Java class with a MongoDB collection. Additionally, the `@Data` annotation from Lombok generates getter and setter methods, as well as `toString`, `equals`, and `hashCode` methods for the class automatically, reducing the amount of code that needs to be written.

The annotation `@NoArgsConstructor` can be used with Lombok to generate a constructor that has no arguments. This allows you to create an instance of the class without specifying any field values. The field `id` is marked as the primary key using the `@Id` annotation. The other fields rep-

resent the properties of the document. The `List<Role>`, `List<Location>`, `List<Integer>`, and `List<Integer>` fields are used to represent relationships to other documents in the database.

The Java classes are used to define the schema for MongoDB documents in the application, and the ORM framework handles the mapping of these documents to and from Java objects.

### ■ 5.1.3 Repository

In the figure bellow, I will show how is implemented access to the database via the repository.

---

```
public interface UserRepository extends MongoRepository<User,
    String> {

    User findByEmail(String email);

    User findByUsername(String username);

    boolean existsUserByUsername(String username);

    boolean existsUserByEmail(String email);
}
```

---

**Table 5.3:** Implementation of UserRepository for database data accessl

The interface called `UserRepository` works with MongoDB to perform CRUD operations on the `User` entity. It builds upon the basic CRUD functions already provided by the `MongoRepository` interface. Along with standard CRUD methods, the `UserRepository` interface contains four extra methods: `findByEmail()`, `findByUsername()`, `existsUserByUsername()`, and `existsUserByEmail()`. The methods `findByEmail()` and `findByUsername()` help users search for a `User` entity using their email address or username respectively. On the other hand, `existsUserByUsername()` and `existsUserByEmail()` methods are used to verify the existence of a `User` entity in the database with the provided username or email address. Using repositories such as `UserRepository` can offer several advantages. The main benefit is that they simplify the process of working with a database. This means I do not have to deal with low-level database queries or handle database connections on my own as the repository manages these complex details. Moreover, repositories provide a uniform approach to accessing data, ensuring that the consistency of the application is maintained. But repositories have their disadvantages. One drawback is that when not optimized properly, repositories can slow down performance. Moreover, repositories may pose challenges when executing intricate database queries that exceed basic CRUD operations.

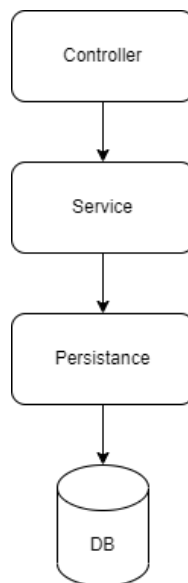
## 5.2 Backend

The `UserService` is a crucial component of the system that requires secure user management, particularly in a microservices architecture. It is responsible for user authentication, authorization, and validation, and is designed to be highly scalable, making it suitable for systems that require secure user management on a large scale. In a microservices architecture, services need to communicate with each other, and `Eureka` service visibility is essential for allowing this communication to happen securely.

The `UserService` implements the `Eureka` service visibility feature, which ensures that only authenticated users can access the system's resources, including other microservices. Furthermore, the `UserService` is designed to integrate seamlessly with other services within the system, such as the `CalendarService`. Its ability to handle a large number of users and integrate with other services makes it an essential part of any system that requires secure user management in a microservices architecture.

### 5.2.1 Architecture description

In the next diagram, I will show how layered architecture `UserService` handle requests that are coming from the frontend part.



**Figure 5.2:** UserService layered architecture

Every request will be handled with this approach:

- Controller will be triggered.
- Controller will call the service method.
- If it is `DELETE` or `GET` then no data validation is required, in all other cases the data will be validated.

- Service will perform the call to the MongoDB to retrieve the data.
- After that the needed data will be serialized and returned to the controller.
- Controller will return data to the frontend as JSON object.

I will provide a more detailed description of each of the processes in the chapters below using sequential diagrams for better visual acceptance.

## ■ 5.2.2 Data Validation

In the `UserService`, it is important to validate the data of each DTO that the application will use in the future. This will ensure that the data being transferred is accurate and meets the system requirements, which helps avoid errors and inconsistencies. Validation involves verifying for missing or incorrect values, checking that data types are suitable, and ensuring that data falls within acceptable limits or ranges. In addition to checking for errors, validation can also involve verifying if a user has enough permissions to perform a certain action. By validating DTOs before storing them in the database, we can ensure accurate and clean data. This helps to avoid problems such as data corruption, data loss, and security vulnerabilities. Validating DTOs is important as it helps to detect and resolve potential issues early in the development process, thereby improving the overall quality of the system. It also helps to save time and resources required for testing and debugging since problems are caught early. To ensure an application's accuracy, reliability, and security, it's crucial to validate DTOs before processing and storing them in the database.

In the next figures, I will show how is been handled validation for registration data.

---

```
public ValidCreateUserRequestDto
    validateRegister(CreateUserRequestDto request, PasswordEncoder
        passwordEncoder) {
    var builder = ValidCreateUserRequestDto.newBuilder();
    var validations = new AggregatedValidation()
        .add(FirstNameSecondName.validate(request.getName(),
            request.getSurname()), builder::name)
        .add(validateEmail(request.getEmail()), builder::email)
        .add(Password.validateAndHash(request.getPassword(),
            passwordEncoder), builder::password)
        .add(Username.validate(request.getUsername()),
            builder::username);

    return
        ValidationUtils.getCheckedResult(validations.validate(builder::build));
}
```

---

**Table 5.4:** Example of aggregated validation function for registration data

On this figure is a validation function that is used to validate a `CreateUserRequestDto` object, which contains information about a user that is being registered. The function takes in two parameters: the `CreateUserRequestDto` object and a `PasswordEncoder` object that is used to encode the user's password.

The function returns a `ValidCreateUserRequestDto` object that contains the validated information about the user. The `ValidCreateUserRequestDto` object is created using the `ValidCreateUserRequestDto.newBuilder()` method.

The validation process is performed by creating an `AggregatedValidation` object that is used to aggregate the results of multiple validation checks. The `AggregatedValidation` object is instantiated with the `new AggregatedValidation()` method.

The validation checks are performed by calling a series of methods on the `AggregatedValidation` object. The validation checks include validating the user's first name and second name, email, password, and username.

For each validation check, a lambda function is passed to the `add()` method of the `AggregatedValidation` object. The lambda function takes the validated result of the validation check and sets the corresponding field in the `ValidCreateUserRequestDto` object.

Once all the validation checks have been performed, the `ValidCreateUserRequestDto` object is built using the `builder::build` method.

Finally, the `ValidationUtils.getCheckedResult()` method is called to get the checked result of the validation process. If any errors occur during the validation process, a `ValidationException` is thrown with the error message. If no errors occur, the validated `ValidCreateUserRequestDto` object is returned.

---

```
private static Validation<FieldError, String> validateName(String
    name, String fieldName, boolean required) {
    return new StringValidator(name, fieldName, required)
        .deduplicateSpaces()
        .minLength(2)
        .maxLength(40)
        .patterns(NAME_PATTERN)
        .validate();
}
```

---

**Table 5.5:** Example of validation function that adds validation for name field

The above figure displays a validation function for a person's name. It requires three parameters: the name as a `String`, the `fieldName` as a `String`, and a boolean value indicating if the field is mandatory. The function returns a `Validation` object that deals with validation errors. The `Validation` object is a generic class that comprises of a `FieldError` object and a `String` representing the field that is being validated.

To validate the input string, we create a `StringValidator` object within the function and call a series of methods on it. These methods include `deduplicateSpaces()` to remove extra spaces, `minLength()` to set a mini-

imum length of 2 characters, `maxLength()` to set a maximum length of 40 characters, and `patterns()` to apply any custom regular expressions.

The validation process is carried out by calling the `validate()` method on the `StringValidator` object. In case any errors arise, a `FieldError` object is returned with details about the field name, error message, and error type.

### ■ 5.2.3 Eureka Service visibility

Eureka is a tool that enables microservices to register themselves with a server and find other dependent services, which improves communication reliability. This helps enhance the scalability and resiliency of the microservices within a system.

Eureka simplifies the management of microservices by offering a directory of services. This directory helps users locate and communicate with other services within the system. It can also balance the load, which means incoming requests can be divided among multiple instances of a service to improve performance and availability.

Eureka offers health checks that can keep track of the real-time availability and status of services. By doing so, administrators can detect and promptly address any issues that may arise, leading to better system reliability and uptime.

In order to utilize Eureka within a microservices architecture, it is necessary for each service to register itself with the Eureka server upon initiation. This can be accomplished using a client library such as the Eureka client for Spring Boot, which can be added as a dependency in your microservice's `pom.xml` file.

---

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

---

**Table 5.6:** Eureka dependency

Next we should configure the Eureka server in `application.properties` file:

---

```
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

---

**Table 5.7:** Eureka dependency

After registering a service, other services can find its location by searching the Eureka server. This makes it possible for services to interact with each other using RESTful APIs or other protocols.

## Chapter 6

### Calendar service

In this chapter, we will discuss the implementation details of the CalendarService, including its architecture, design patterns, and code structure. We will also explore its functionality, key features, and integration with other services and components in the system. Furthermore, we will cover the challenges encountered during its development and the corresponding solutions.

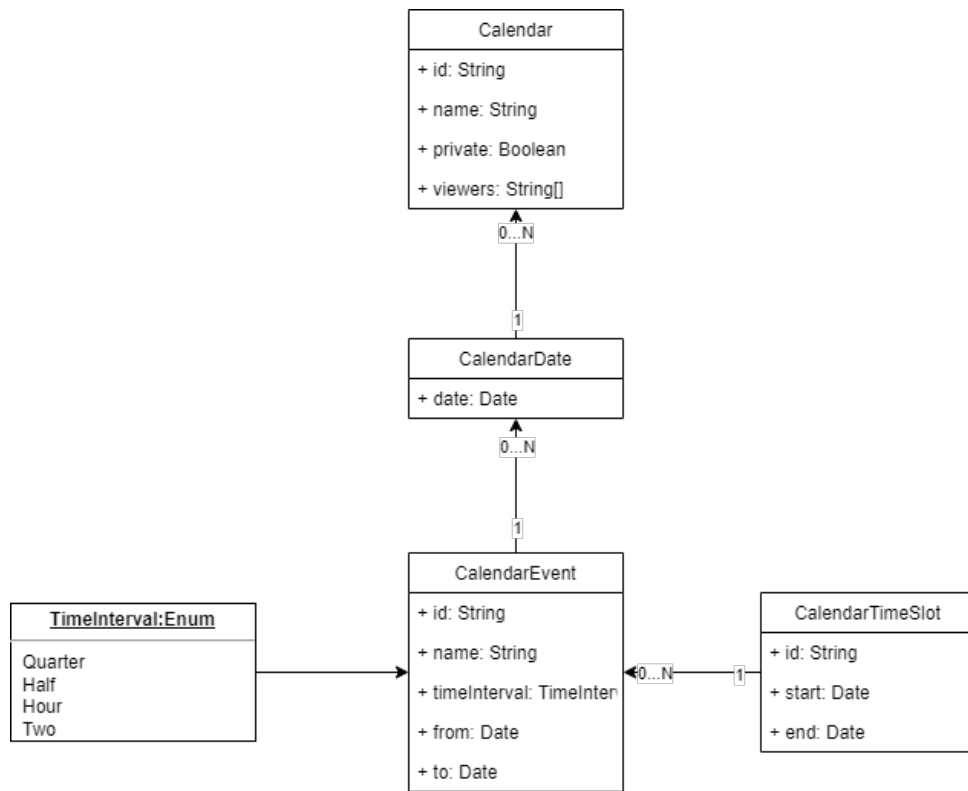
#### 6.1 Database

The implementation of the CalendarService database will be responsible for managing all calendar-related operations across various entities, including calendar, calendarDate, calendarEvent, and calendarTimeSlot. In addition, the database will support the functionality of fetching data from external calendars for importing events and exporting operations for extracting calendars in Ical format.

Designing a well-structured database is crucial for the efficient functioning of any application, and CalendarService is no exception. To achieve this, we must identify the types of data that need to be stored and accessed efficiently. We will then proceed to make decisions about the database schema, such as table structure, relationships between tables, and data types.

Furthermore, we must ensure that the database management system used for the CalendarService can perform crucial tasks such as data indexing, query optimization, and data backup and recovery. These features will guarantee the integrity and security of the data stored in the database.

### 6.1.1 Architecture description



**Figure 6.1:** Architecture of calendar database

The database architecture for the UML diagram containing the entities `CalendarEvent`, `CalendarDate`, and `CalendarTimeSlot` involves multiple tables and relationships. The `CalendarEvent` table contains the parameters:

- `id`
- `name`
- `description`
- `from`
- `to`
- `timeInterval` (represented by the `TimeIntervalEnum` enumeration with options `Quarter`, `Half`, `Hour`, `Two`)
- `numberOfStudents`
- `creator`

This table has a Many-to-one relationship with the `CalendarDate` table, allowing multiple events to be associated with a single date.

The `CalendarDate` table contains the parameters:



- id
- date

and has a Many-to-one relationship with the `Calendar` table. The `Calendar` table includes the fields:

- id
- name
- isPrivate
- viewers
- owner

and represents the parent entity for the `CalendarDate` table.

Finally, the `CalendarEvent` table has a One-to-many relationship with the `CalendarTimeSlot` table, which contains the parameters:

- id
- start
- end
- numberOfParticipants
- users

This relationship allows multiple time slots to be associated with a single event.

In summary, the UML diagram's database architecture involves three tables: `CalendarEvent`, `CalendarDate`, and `CalendarTimeSlot`, with various fields and relationships between them. The structure of this database is designed to support the requirements of a calendar-based application, enabling efficient management of events, dates, and time slots.

### 6.1.2 ORM implementation

The provided code is written in Java and demonstrates a `Calendar` model that uses ORM with the help of MongoDB and Lombok annotations. The code includes several annotations that serve different purposes. The `@Document` annotation indicates that this class is going to be mapped to a MongoDB document. `@Data` annotation is used to generate methods like getters, setters, equals, hashCode, and toString. Lastly, `@NoArgsConstructor` is used to create a no-argument constructor.

The `Calendar` model consists of fields such as `id`, `name`, `isPrivate`, `viewers`, `owner`, and `dates`. The `id` field is designated as the Primary key for the `Calendar` document and is annotated with `@Id`. The information about a calendar is stored in specific fields. The `name` field contains the calendar's name. The `isPrivate` field indicates if the calendar is private. The `viewers` field is a set of user IDs who can view the calendar. The `owner` field contains the ID of the user who made the calendar. And lastly, the `dates` field is a list of `CalendarDate` objects connected to the calendar.

The code shows how to use ORM along with the MongoDB database to manage data persistence in a Java application. The use of Lombok annotations helps to reduce repetitive code and improve the readability and maintenance of the code.

---

```

@Document
@Data
@NoArgsConstructor
public class Calendar {

    @Id
    private String id;
    private String name;
    private Boolean isPrivate;
    private Set<String> viewers;
    private String owner;
    private List<CalendarDate> dates;
}

```

---

**Table 6.1:** ORM implementation for calendar entity

The following Java ORM code provided represents the `CalendarDate` model in MongoDB using Lombok annotations. The model is designed to store and retrieve calendar dates along with the events associated with them. The use of Lombok annotations helps in reducing the boilerplate code and makes the code more concise and readable.

The `id` field is of type `String` and is used to store the Primary key. The `date` field is of type `LocalDate` and is used to store the calendar date. The `events` field is a List of `CalendarEvent` objects and is used to store the events associated with the calendar date.

---

```

@Document
@Data
@NoArgsConstructor
public class CalendarDate {

    @Id
    private String id;
    private LocalDate date;
    private List<CalendarEvent> events;
}

```

---

**Table 6.2:** ORM implementation for calendarDate entity

This model is intended for saving and accessing details about calendar events. The field labeled `id`, which is of type `String`, serves as the primary key. The field labeled `name`, also a `String`, is meant to hold the name of the event. As for the `description` field, it is also a `String` and is intended for storing the event's description.

The fields for storing the start and end times of an event are named `from` and `to`, respectively, and they are both of type `LocalDateTime`. The field for storing the time interval of the event is of type `TimeInterval` and is named `timeInterval`. Finally, the field for storing the time slots of the event is a List of `CalendarTimeSlot` objects named `calendarEventTimeSlots`.

The "location" field is a string that stores the event's location, while the "numberOfStudents" field is an integer that stores the count of students attending the event. The "creator" field is a string that holds the name of the event's creator.

---

```

@Document
@Data
@NoArgsConstructor
public class CalendarEvent {
    @Id
    private String id;
    private String name;
    private String description;
    private LocalDateTime from;
    private LocalDateTime to;
    private TimeInterval timeInterval;
    private List<CalendarTimeSlot> calendarEventTimeSlots;
    private String location;
    private Integer numberOfStudents;
    private String creator;
}

```

---

**Table 6.3:** ORM implementation for calendarEvent entity

Next figure represents the Calendar model in MongoDB using Lombok

annotations. The `id` field is of type `String` and is used to store the Primary key. The `name` field is a `String` and is used to store the name of the calendar.

The `isPrivate` field is a `Boolean` and is used to indicate whether the calendar is private or not. The `viewers` field is a `Set` of `String` objects and is used to store the names of the viewers who have access to the calendar. The `owner` field is a `String` and is used to store the name of the owner of the calendar.

The `dates` field is a `List` of `CalendarDate` objects and is used to store the calendar dates associated with the calendar. Each `CalendarDate` object in the list represents a date and the events associated with it.

---

```

@Document
@Data
@NoArgsConstructor
public class Calendar {

    @Id
    private String id;
    private String name;
    private Boolean isPrivate;
    private Set<String> viewers;
    private String owner;
    private List<CalendarDate> dates;
}

```

---

**Table 6.4:** ORM implementation for calendar entity

### 6.1.3 Repositories

---

```

public interface CalendarRepository extends
    MongoRepository<Calendar, String> {

    List<Calendar> findCalendarsByOwner(String owner);

    Calendar findCalendarByOwnerAndId(String owner, String id);

    Calendar findCalendarByOwnerAndName(String owner, String name);

    Boolean existsByName(String name);
}

```

---

**Table 6.5:** CalendarRepository implementation

---

```

public interface CalendarEventRepository extends
    MongoRepository<CalendarEvent, String> {
    Boolean existsByCreatorAndName(String creatorId, String name);

    List<CalendarEvent> findAllByCreator(String creator);

    CalendarEvent findByCreatorAndId(String creator, String id);
}

```

---

**Table 6.6:** CalendarEventRepository implementation

The following Java code includes two interfaces - `CalendarRepository` and `CalendarEventRepository`. These interfaces are employed to carry out CRUD (Create, Read, Update, and Delete) operations on two MongoDB database models - `Calendar` and `CalendarEvent`.

The `CalendarRepository` interface extends the functionality of the `MongoRepository` interface by leveraging its existing methods for performing CRUD operations. The `MongoRepository` interface requires two generic parameters. The first parameter corresponds to the model class that is being managed, which in this case is the `Calendar` model. The second parameter denotes the primary key type of the model class, which is a `String` in this particular case.

The `CalendarRepository` interface has custom methods that can be used to search for specific information in the database. One such method is `findCalendarsByOwner`, which takes an `owner` parameter and returns a list of all the `Calendar` objects that belong to that particular owner. Another method is `findCalendarByOwnerAndId`, which takes both `owner` and `id` parameters and returns a single `Calendar` object that matches these criteria. The function called `findCalendarByOwnerAndName` requires an `owner` and a `name` parameter to find and return a matching `Calendar` object. Whereas, the `existsByName` function only needs a `name` parameter to check if a `Calendar` object with that name exists in the database and will return a boolean accordingly.

The interface called `CalendarEventRepository` builds upon the `MongoRepository` interface. It includes additional methods that can be used to search for particular information about the `CalendarEvent` model in the database. One of these methods is called `existsByCreatorAndName`. This method requires a `creatorId` and a `name` as input and returns a boolean value indicating whether a `CalendarEvent` object with the specified `creator` and `name` can be found in the database. The methods `findAllByCreator` and `findByCreatorAndId` both require a `creator` parameter to be provided. The first method returns a list of all `CalendarEvent` objects created by the specified `creator`, while the second method returns a single `CalendarEvent` object that matches both the specified `creator` and `id` parameters.

## 6.2 Backend

The `CalendarService` backend part plays a pivotal role in the system, providing a range of functionalities related to calendar management. While the `UserService` handles user authentication/authorization and other operations related to the user, the `CalendarService` focuses on handling external calendar connections, calendar exports, and other CRUD operations involving calendar entities.

In the microservices architecture, the `CalendarService` operates as an independent service, employing its own set of logic to manage calendars effectively. It follows the principles of microservice architecture, enabling it to function autonomously while seamlessly integrating with other services within the system.

One of the primary responsibilities of the `CalendarService` is to establish connections with external calendars, allowing users to synchronize their events and schedules. It leverages various protocols and APIs to facilitate this synchronization process, ensuring that events from external calendars are accurately reflected within the system.

Additionally, the `CalendarService` provides functionalities for exporting calendar data in different formats, enabling users to share their schedules with external applications or individuals. It implements logic to generate standardized calendar files, such as iCal or Google Calendar files, that can be easily imported by other calendar applications.

Alongside the external calendar integrations, the `CalendarService` implements typical CRUD operations for managing calendar entities within the system. This includes creating new calendars, retrieving existing calendars, updating calendar details, and deleting calendars when necessary. These operations are performed securely, ensuring that only authorized users can manipulate calendar data within the system.

It's important to note that while the `CalendarService` handles various aspects of calendar management, it does not provide authorization functionality. The responsibility for user authorization lies with the `UserService`, which determines the access privileges of users within the system. By separating the concerns of user management and calendar management, the system achieves a modular and scalable design.

### ■ 6.2.1 Architecture description

Similar to the `UserService`, the `CalendarService` shares the same layered architecture approach described in the previous chapter. It handles requests related to calendar management and integrates seamlessly with the microservices architecture. Let's briefly explore how the `CalendarService` aligns with the established architecture:

- Controller Trigger
  - When a request targeting calendar operations is received from the frontend, the corresponding controller in the `CalendarService` is triggered.
- Service Method Invocation
  - The controller invokes the appropriate service method in the `CalendarService`.
  - The service method encapsulates the business logic and data processing related to calendar operations.
- Data Validation
  - Data validation is performed based on the specific requirements of the calendar operations.
  - Depending on the request type, such as creating a new calendar or updating existing calendar data, the incoming data is validated to ensure its accuracy and completeness.
- Retrieval of Data from MongoDB
  - The `CalendarService` interacts with the MongoDB database to fetch or manipulate calendar-related data.
  - The service leverages MongoDB's capabilities to efficiently query and manage calendar information.
- Serialization and Return of Data
  - Retrieved or processed data is serialized into a suitable format, in our case JSON
  - Serialization enables the structured representation of calendar data for seamless transmission and interpretation.
- Response to Frontend
  - The controller receives the serialized data from the service method.
  - The controller sends the JSON-formatted data back to the frontend as a response to the initial calendar-related request.

Adhering to the established layered architecture, the `CalendarService` seamlessly integrates with the microservices architecture while providing dedicated logic for handling external calendar connections, calendar exports, and other CRUD operations with calendar entities.

## 6.2.2 Data Validation

The following code represents a class that is responsible for validating calendar-related requests and names.

- `validateCreate(CreateCalendarDto request)` takes a `CreateCalendarDto` object as input, representing the request for creating a calendar. It creates a `ValidCreateCalendarRequestDto` builder instance to construct a validated create calendar request. The method performs validation using an `AggregatedValidation` object, which aggregates multiple validation rules. One specific validation is performed on the name field of the `CreateCalendarDto` object using the `Name.validate` method. The validated name value is set using the builder's `name` reference method. The `isPrivate` field of the `CreateCalendarDto` object is set directly on the builder. The final validated `ValidCreateCalendarRequestDto` object is returned after checking the result of all validations using `ValidationUtils.getCheckedResult`.
- `validateName(String name)` takes a name string as input, representing the name of a calendar. It creates a `ValidCalendarName` builder instance to construct a validated calendar name object. The method performs a single validation on the name string using the `Name.validate` method. The validated name value is set using the builder's `name` reference method. The final validated `ValidCalendarName` object is returned after checking the result of the validation using `ValidationUtils.getCheckedResult`.

---

```
public class CalendarValidator {

    public ValidCreateCalendarRequestDto
        validateCreate(CreateCalendarDto request) {
        var builder = ValidCreateCalendarRequestDto.newBuilder();
        var validations = new AggregatedValidation()
            .add(Name.validate(request.getName()), builder::name);

        builder.isPrivate(request.getIsPrivate());

        return
            ValidationUtils.getCheckedResult(validations.validate(builder::build));
    }

    public ValidCalendarName validateName(String name) {
        var builder = ValidCalendarName.newBuilder();
        var validation = new AggregatedValidation()
            .add(Name.validate(name), builder::name);
        return
            ValidationUtils.getCheckedResult(validation.validate(builder::build));
    }
}
```

---

**Table 6.7:** Example of aggregated validation function for calendar data



The `validateRequest` method in the next part of the code is responsible for validating a `CalendarEventDto` object, which represents a request for creating a calendar event.

- It takes a `CalendarEventDto` object as a parameter.
- Creates a `ValidCreateCalendarEventRequestDto` builder to construct a validated create calendar event request.
- Initializes an `AggregatedValidation` object to combine multiple validation rules.
- The `AggregatedValidation` object performs the following validations:
  - Validates the name and description fields of the `CalendarEventDto` object using the `CalendarEventInfo.validate` method. The validated values are set on the builder using the `builder::info` reference method.
  - Validates the `timeInterval` field of the `CalendarEventDto` object using the `validateInterval` method. The validated value is set on the builder using the `builder::timeInterval` reference method.
  - Validates the `from`, `to`, and `timeInterval` fields of the `CalendarEventDto` object using the `FromTo.validate` method. The validated values are set on the builder using the `builder::fromTo` reference method.
  - Validates the `numberOfParticipants` field of the `CalendarEventDto` object using the `validateNumberOfParticipants` method. The validated value is set on the builder using the `builder::numberOfParticipants` reference method.
- Sets the `location` and `calendarId` fields of the `CalendarEventDto` object directly on the builder.
- Returns the validated `ValidCreateCalendarEventRequestDto` object after checking the result of all validations using `ValidationUtils.getCheckedResult`.

---

```

public ValidCreateCalendarEventRequestDto
    validateRequest(CalendarEventDto calendarEventRequestDto) {
    var builder = ValidCreateCalendarEventRequestDto.newBuilder();
    var validations = new AggregatedValidation()
        .add(CalendarEventInfo.validate(calendarEventRequestDto.getName(),
            calendarEventRequestDto.getDescription()),
            builder::info)
        .add(validateInterval(calendarEventRequestDto.getTimeInterval()),
            builder::timeInterval)
        .add(FromTo.validate(calendarEventRequestDto.getFrom(),
            calendarEventRequestDto.getTo(),
            calendarEventRequestDto.getTimeInterval()),
            builder::fromTo)
        .add(validateNumberOfParticipants(calendarEventRequestDto.getNumberOfParticipants()),
            builder::numberOfParticipants);

    builder.location(calendarEventRequestDto.getLocation());
    builder.calendarId(calendarEventRequestDto.getCalendarId());
    return
        ValidationUtils.getCheckedResult(validations.validate(builder::build));
    }

```

---

**Table 6.8:** Example of aggregated validation function for calendar event creation data

The following code snippet demonstrates the implementation of the validation for the `timeSlotDto`.

- The method `validCreateTimeSlotDto` takes a `CreateTimeSlotDto` parameter named `createTimeSlotDto`, representing the data of a time slot to be validated.
- Inside the method, a new `ValidCreateTimeSlotDto` builder object is created using `ValidCreateTimeSlotDto.newBuilder()`. This builder will be used to construct the validated time slot DTO.
- A `Validations` object named `validations` is created using `AggregatedValidation()`. This object will hold all the validations to be performed on the `createTimeSlotDto`.
- The validations are added to the `validations` object using the `add` method. Two validations are performed:
  - The `FromTo.validate` method is called with the start and end times from the `createTimeSlotDto`. The result is assigned to the `fromTo` field of the builder using a method reference.
  - The `validateNumberOfParticipants` method is called with the number of participants from the `createTimeSlotDto`. The result is assigned to the `numberOfParticipants` field of the builder using a method reference.

- The `ownerId` and `eventId` fields of the `builder` are set with the corresponding values from the `createTimeSlotDto`.
- Finally, the `validations` are executed using `ValidationUtils.getCheckedResult` to obtain the validated `ValidCreateTimeSlotDto` object. This object is returned from the method.

---

```
public class CalendarTimeSlotValidator {  
  
    public ValidCreateTimeSlotDto  
        validCreateTimeSlotDto(CreateTimeSlotDto createTimeSlotDto){  
        var builder = ValidCreateTimeSlotDto.newBuilder();  
        var validations = new AggregatedValidation()  
            .add(FromTo.validate(createTimeSlotDto.getStart(),  
                createTimeSlotDto.getEnd()), builder::fromTo)  
            .add(validateNumberOfParticipants(createTimeSlotDto.getNumberOfParticipants()), builder::validateNumberOfParticipants);  
  
        builder.ownerId(createTimeSlotDto.getOwnerId());  
        builder.eventId(createTimeSlotDto.getEventId());  
  
        return  
            ValidationUtils.getCheckedResult(validations.validate(builder::build));  
    }  
}
```

---

**Table 6.9:** Example of aggregated validation function for calendar timeslot data

### 6.2.3 Calendar Export

This chapter will explore a class called `CalendarExportService`. It has a method called `exportToICal` which is responsible for converting a list of `CalendarEvent` objects into the iCalendar (iCal) format. The `exportToICal` method needs a List of `CalendarEvent` objects as input. Once executed, it will return a `String` that represents the generated `CalendarEvent` data. The code creates a `StringBuilder` named `iCalData` to store iCalendar data. The method then adds the necessary headers to the `iCalData` `StringBuilder`, including the `BEGIN:VCALENDAR`, `VERSION:2.0`, and `PRODID` lines. After that, it loops through each `CalendarEvent` in the `calendarEvents` list. The method adds the line `"BEGIN:VEVENT"` to the iCalendar data for each `CalendarEvent`, marking the beginning of a new event. Then, the UID, start time, end time, title (summary) and event description are added to the `iCalData` `StringBuilder` in the correct iCalendar format.

---

```
import java.util.List;

public class CalendarExportService {

    public String exportToICal(List<CalendarEvent> calendarEvents) {
        StringBuilder iCalData = new StringBuilder();

        iCalData.append("BEGIN:VCALENDAR\r\n");
        iCalData.append("VERSION:2.0\r\n");
        iCalData.append("PRODID:-//CVUT//MTAP//EN\r\n");

        for (CalendarEvent event : calendarEvents) {
            iCalData.append("BEGIN:VEVENT\r\n");
            iCalData.append("UID:").append(event.getUid()).append("\r\n");
            iCalData.append("DTSTART:").append(event.getStartTIme()).append("\r\n");
            iCalData.append("DTEND:").append(event.getEndTime()).append("\r\n");
            iCalData.append("SUMMARY:").append(event.getTitle()).append("\r\n");
            iCalData.append("DESCRIPTION:").append(event.getDescription()).append("\r\n");
            iCalData.append("END:VEVENT\r\n");
        }

        iCalData.append("END:VCALENDAR");

        return iCalData.toString();
    }
}
```

---

**Table 6.10:** Example of export calendar service

## 6.2.4 CalendarEvent creation

In this section, we will explore how the entity `CalendarEvent` is created. The process includes validating the data, and once the data is validated successfully, a private method, `createCalendarEvent`, is executed.

The method creates a `CalendarEvent` object using the given `ValidCreateCalendarEventRequestDto` and owner ID. It begins by initializing the object and then parsing the `from` and `to` fields from the `ValidCreateCalendarEventRequestDto` into `LocalDateTime` objects using the `LocalDateTime.parse` method. The `ValidCreateCalendarEventRequestDto.timeInterval` field is converted into a `TimeInterval` object via the `parseTimeInterval` method. The `name`, `description`, `location`, `from`, `to`, and `timeInterval` fields of the resulting `CalendarEvent` object are then populated with the corresponding fields from the `ValidCreateCalendarEventRequestDto`.

The value of the `timeInterval` field in the `CalendarEvent` object depends on the value of the `timeInterval` field in the `ValidCreateCalendarEventRequestDto`. If the value is one of the predetermined options (15 min, 30 min, 60 min, 120 min), the `timeInterval` field is set to that value. Otherwise, a custom `TimeInterval` object is created with the specified number of minutes and used as the `timeInterval` for the `CalendarEvent`.

The provided owner ID is used to set the `creator` field of the `CalendarEvent`. To set the `calendarEventTimeSlots` field of the `CalendarEvent`, we call the `calendarTimeSlotService.createTimeSlots` method and pass the `from`, `to`, `timeInterval`, and `numberOfParticipants` from the `ValidCreateCalendarEventRequestDto`. More information about the `createTimeSlots` method will be provided below.

The `numberOfStudents` field of the `CalendarEvent` is set with the value of `requestDto.getNumberOfParticipants()`. The created `CalendarEvent` object is then returned by the method. This method effectively creates a `CalendarEvent` object based on the provided request DTO and owner ID, initializing its fields with the corresponding values. It also handles the conditional assignment of the `timeInterval` field based on the value of `requestDto.getTimeInterval()`.

---

```

private CalendarEvent
    createCalendarEvent(ValidCreateCalendarEventRequestDto
        requestDto, String ownerId) {
    var calendarEvent = new CalendarEvent();
    var from = LocalDateTime.parse(requestDto.getFromTo().from());
    var to = LocalDateTime.parse(requestDto.getFromTo().to());
    var timeInterval =
        parseTimeInterval(requestDto.getTimeInterval());

    calendarEvent.setName(requestDto.getInfo().name());
    calendarEvent.setDescription(requestDto.getInfo().description());
    calendarEvent.setLocation(requestDto.getLocation());
    calendarEvent.setFrom(from);
    calendarEvent.setTo(to);
    if (requestDto.getTimeInterval() == 15 ||
        requestDto.getTimeInterval() == 30 ||
        requestDto.getTimeInterval() == 60 ||
        requestDto.getTimeInterval() == 120) {
        calendarEvent.setTimeInterval(timeInterval);
    } else {
        TimeInterval customInterval = TimeInterval.CUSTOM;
        customInterval.setMinutes(requestDto.getTimeInterval());
        calendarEvent.setTimeInterval(customInterval);
    }
    calendarEvent.setCreator(ownerId);
    calendarEvent.setCalendarEventTimeSlots(calendarTimeSlotService.createTimeSlots(
        from, to, timeInterval,
        requestDto.getNumberOfParticipants()
    ));
    calendarEvent.setNumberOfStudents(requestDto.getNumberOfParticipants());

    return calendarEvent;
}

```

---

**Table 6.11:** Example createEvent function

Next, we will have a look at the createTimeSlots method which was mentioned above, that is responsible for creating a list of calendar time slots within a specified time range. The method takes four parameters:

- from - a LocalDateTime object representing the start of the time range.
- to - a LocalDateTime object representing the end of the time range.
- timeInterval - a TimeInterval object representing the duration of each time slot.
- numberOfStudents - an Integer indicating the number of students that can be accommodated in each time slot.

The method begins by initializing an empty list called calendarTimeSlots to store the created time slots. Next, it calls a method TimeIntervalCreator.create()

to generate a list of intervals based on the provided `from`, `to`, and `timeInterval` values. The `TimeIntervalCreator.create()` method calculates the intervals between the start and end time based on the duration specified in minutes. Then it iterates over each interval in the intervals list. For each interval, it calls a method `createTimeSlot()` to create a `CalendarTimeSlot` object using the interval's start and end times, along with the provided `numberOfStudents`. The created `CalendarTimeSlot` object is added to the `calendarTimeslots` list. After adding the time slot to the list, it appears that the code also persists the `CalendarTimeSlot` object by calling `calendarTimeSlotRepository.save()`. This suggests that there is a repository or database interaction involved in storing the created time slots. Finally, the method returns the `calendarTimeslots` list containing all the created time slots.

---

```

@Override
public List<CalendarTimeSlot> createTimeSlots(LocalDateTime from,
    LocalDateTime to, TimeInterval timeInterval, Integer
    numberOfStudents) {
    List<CalendarTimeSlot> calendarTimeslots = new ArrayList<>();
    List<TimeIntervalCreator.Interval> intervals =
        TimeIntervalCreator.create(from, to,
            timeInterval.getMinutes() * 60);
    for (TimeIntervalCreator.Interval interval : intervals) {
        var calendarTimeSlot = createTimeSlot(interval.start(),
            interval.end(), numberOfStudents);
        calendarTimeslots.add(calendarTimeSlot);
        calendarTimeSlotRepository.save(calendarTimeSlot);
    }
    return calendarTimeslots;
}

```

---

**Table 6.12:** Example create TimeSlots function







## Chapter 7

### Authorization

One of the most important parts of an application is security so in this chapter, I will describe it in terms of different parts of the MTAP app. The authorization process is divided into two parts - internal and external. Internal authorization is authorization in the MTAP app using a username and password which is user-defined during the registration process. External authorization is authorization and granting access to a different online calendar.

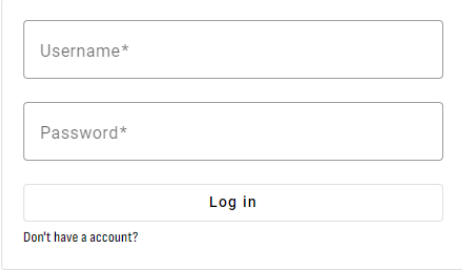


#### 7.1 Internal Authorization

In this part, we will have a closer look at internal authorization and how it works on different levels of application. It is divided into 3 parts - Database, Back end, and Front-end. DB is responsible for storing and searching the data. It can be granted access, get contact info about the user, and get all tokens that are connected to the user. The back end is responsible for security and DB manipulations and the Front end for the user interface.

### 7.1.1 Frontend

In the next figure, we can see a screen shoot from the application showing us a login page.



The image shows a simple login form with three main components: a text input field for 'Username\*', a text input field for 'Password\*', and a 'Log in' button. Below the button, there is a link that says 'Don't have an account?'. The form is enclosed in a light gray border.

**Figure 7.1:** Frontend login page

For sign-in user should fill in all mandatory fields (all mandatory fields are marked with \*). The frontend will validate all fields before sending data to the back end and preventing XSS, CSRF attacks. After control data will be converted to JSON format and sent to `user/authentication` endpoint. As a response, a JWT token will be received. This token will be injected in all next backend calls to allow us API's calls such as getting current signed users, getting calendars, updating entities, adding new events, and so on. Also, this JWT token will be stored in local storage.

### 7.1.2 Backend

The internal authorization back end is responsible for authorizing the user's access for performing operations that will change the data structure, storing data in the database, and authenticating the user. For this version of the application, I used JWT-based authorization. What does it mean? If a user with this combination of username and password exists in the database user will get the JWT token, frontend will store it for future usage in local storage. After the user gets his JWT token the front end will automatically sign all his HTTP requests with it. The back end will get the HTTP GET/POST request with an authorization header and will validate the JWT token from there. If the token is valid - the HTTP request will be processed, if not - the

user will get a 403 error - “Forbidden error”

### ■ 7.1.3 Database

The role of the database, specifically MongoDB in this case, in the authorization process is quite simple - it serves as a storage system to securely store the usernames and their corresponding hashed passwords for future usage. MongoDB is a NoSQL database that provides a flexible and scalable solution for managing data.

To facilitate the storage and retrieval of user information, a repository is created.

---

```
User findByUsername(String username);  
  
boolean existsUserByUsername(String username);
```

---

**Table 7.1:** Example of user repository

This repository serves as a bridge between the application and MongoDB database, facilitating efficient search and modification of user records. Its functions include storage, updating, and searching of user data in the database.

Using MongoDB and the repository improves the security of the authorization process by allowing user credentials to be securely stored and retrieved. When a user sets or changes their password, the repository encrypts the password and stores it in the database. To authenticate a user, the repository searches for the username and retrieves the corresponding hashed password. It then returns the user object to the backend for further authorization.

## 7.2 External Authorization

### 7.2.1 Google API

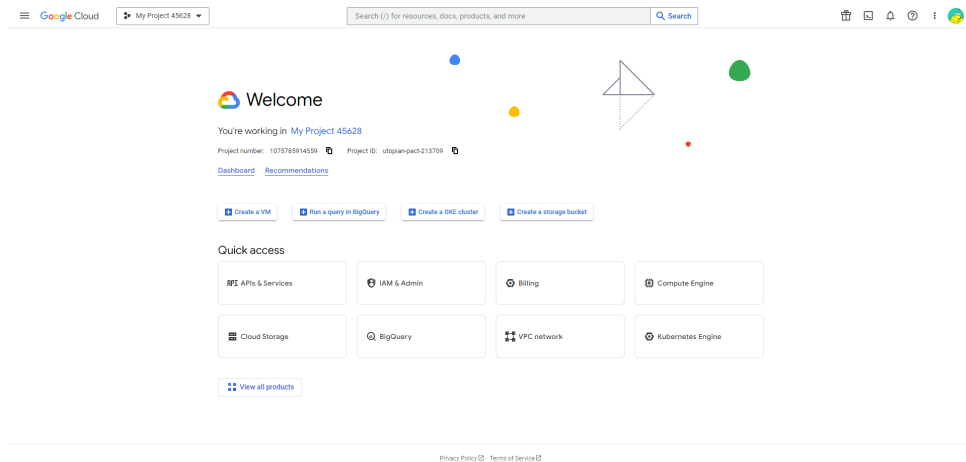


Figure 7.2: Google console homepage

To begin the process, we must navigate to the Google Cloud Platform in order to create a project. The specific name given to the project is not crucial, although it may be beneficial to align the project name with the API that will be utilized. The project serves as a container wherein the `OAuth 2.0` client ID will be housed. Once the project has been successfully created, the next step involves accessing the credentials screen and generating an `OAuth Client ID` using the provided `Create Credentials` dropdown option.

In some cases, it may be necessary to create an `OAuth` consent screen prior to being able to generate the `OAuth Client ID`. This particular step can seem somewhat daunting due to the various questions that need to be answered, as the consent screen can serve multiple purposes beyond the API authentication being discussed. However, accepting the default options and proceeding should be sufficient in most cases. The user type required for this process is typically `External`.

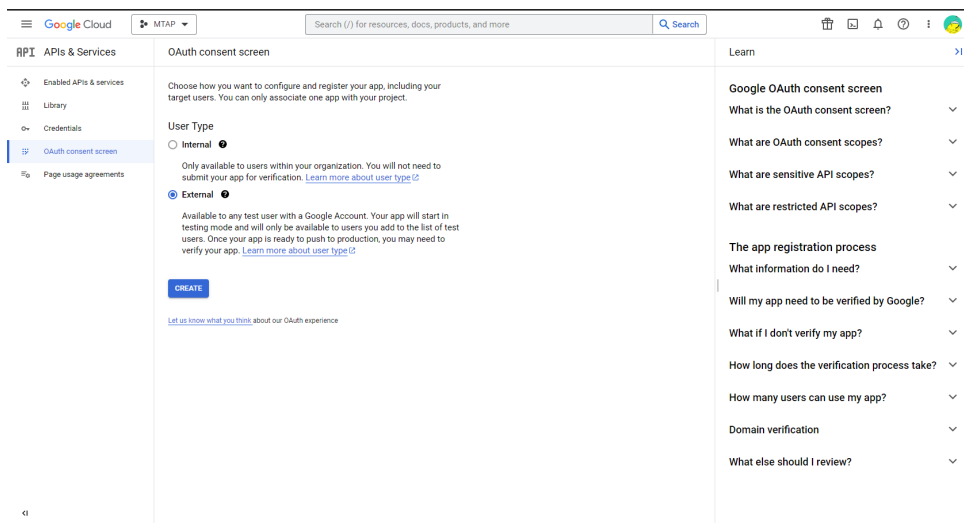


Figure 7.3: Google Cloud setting up environment

At this point, an app registration prompt will appear, where only a name (which can be any name) and an email address are required. Scopes do not need to be a concern at this stage. It is important to note that either planning to publish the app or setting up as a test user will be necessary to authenticate with the app. Progressing further through the process will lead to the creation of the OAuth consent screen, which is needed to generate the OAuth client ID.

Creating the OAuth client ID can be slightly confusing, as it requires selecting the "TVs and Limited Input devices" application type. This specific application type is chosen in order to obtain a refresh token that can be utilized in the future to acquire tokens for accessing Google APIs.

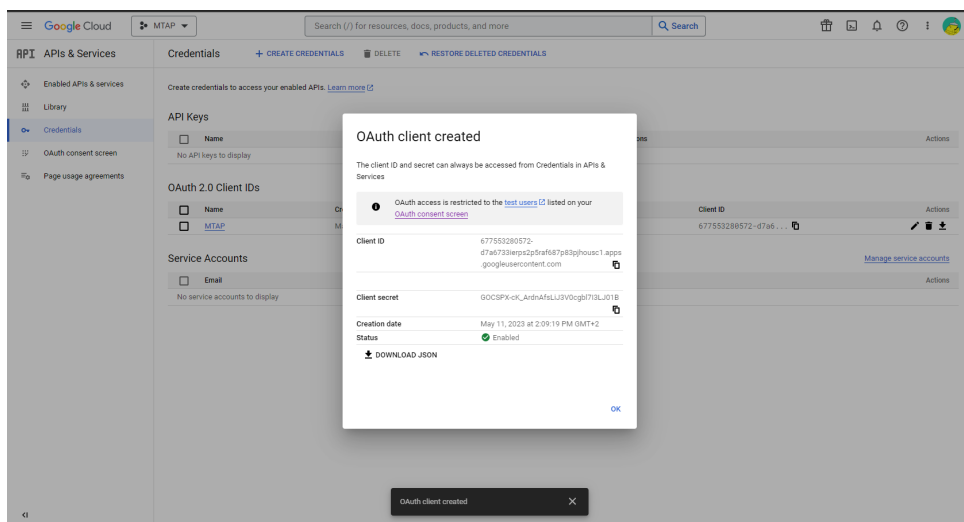


Figure 7.4: Google cloud credentials

Once the OAuth client ID has been successfully created, it can be down-

loaded from the Google Cloud Platform. It is crucial to securely store the `client_id`, `client_secret`, and `redirect_uris` and avoid committing the `client_id` and `client_secret` to source control. The Google authentication provider must be provided with the `client_id` and `client_secret`, and in return, it will provide an authentication URL.

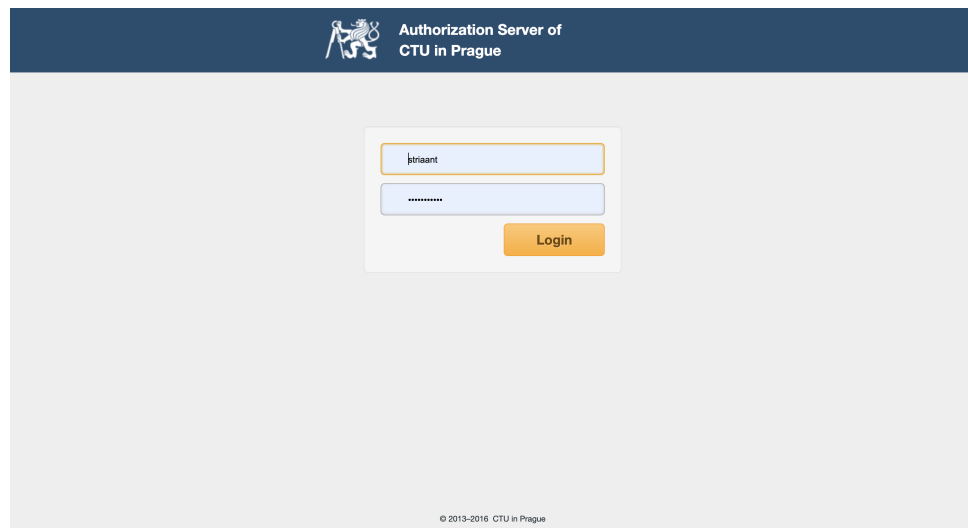
Opening the authentication URL in a browser and granting consent will result in the provider issuing a code. This code needs to be provided to the Google authentication provider, along with the `client_id` and `client_secret`. As a result, the provider will obtain and supply the user with a refresh token that can be used for subsequent token acquisition to access Google API's.

### 7.2.2 CTU API

To start CTU authorization process frontend should trigger the `oauth/oauth/authorize` endpoint and send the next data as URL parameters:

- `response_type=code`
- `client_id=<our_generated_client_id_from_apps_manager>`
- `state=<defined_state>`
- `redirect_uri=<redirect_uri_defined_in_apps_manager>`

After that user will be showed log in page



**Figure 7.5:** CTU login page

After successful login and granting access to third-party service user will be redirected to the user-settings page and the URL will contain a code that is needed for further steps in the authorization process.

---

```
http://localhost:4200/?code=v1v5JI&state=xyz
```

---

**Table 7.2:** Example uri returned after code retrieval from CTU auth server

After that application will perform on the background API call for retrieving the CTU access token that will be saved in the local storage for future injection in the API calls to the CTU API.

---

```
{
  "access_token": "ea173f10-babc-404f-b88d-f0ee8d95ff7c",
  "token_type": "bearer",
  "refresh_token": "aba6d32d-4f17-49e6-afcc-1f042f3e6d3c",
  "expires_in": 1209599,
  "scope": "urn:zuul:oauth:kosapi:public.readonly"
}
```

---

**Table 7.3:** Example JSON response with token

Here is an example of the response of CTU API with an access token that will be used in request authorization.





## Chapter 8

### Testing

Testing will be done in two different ways because we can't use a single approach, time, and resources to test the back and frontend parts effectively. The full application can be user tested and depending on the result application will be changed. For both parts, a CI/CD pipeline can be created for testing and deployment.

#### 8.1 Backend

Mock Unit tests will be used to cover the backend using the JUnit framework. Each part of the backend will be tested with a mocked level that follows it in the three-layer architecture. JUnit testing is essential in ensuring the quality and reliability of Java backend development. To incorporate JUnit testing into the development pipeline, specific steps need to be taken to ensure the process's success. During the development process, the Java backend code undergoes several stages of construction and refinement. Maven, a powerful build automation tool, plays a crucial role during this process. It helps in managing dependencies and executing project-related tasks, including JUnit testing. The Maven framework has two important commands - "maven package" and "maven compile." They compile the source code, package it and resolve required dependencies, ensuring proper building and preparation of the project for testing and deployment. Following these commands can help developers ensure readiness for further development stages. In addition, Maven has a testing framework that comes with a user-friendly wizard. This framework can execute JUnit tests smoothly for developers. It conducts all unit tests for the Java backend application. Through this approach, developers can ensure that the codebase is correct and consistent. To ensure the code's expected behavior and requirements are met, it's important to evaluate the test results. Passing tests ensure a smooth process while failing tests don't necessarily mean a complete pipeline failure. Instead, failures are flagged, giving developers feedback on specific test cases that need debugging and refinement until all tests pass successfully.

## 8.2 Frontend

I don't want to use manual user testing for the single frontend features, because it is more time-consuming than regular `Unit` tests. So the strategy for a frontend will be: create `Unit` tests with `Jest` framework, and set up a pipeline on git, so every new feature will be tested in case of regression older features, perform manual user testing only on major versions, and such a way, lower the time spent on testing. By default, Angular applications are tested by `Karma` framework out of the box, but `nx monorepo` provide us a `Jest` framework for it. Let's have a look at why we should migrate our project to the `Jest`.

- **Faster Execution:** `Jest` has a reputation for being significantly faster than `Karma`. `Jest` optimizes test execution by running tests in parallel, which can result in faster feedback during development.
- **Zero Configuration:** `Jest` requires minimal configuration out of the box, making it easier to set up and get started with testing Angular applications. On the other hand, `Karma` requires more configuration to set up test runners, browsers, and frameworks.
- **Built-in Features:** `Jest` comes bundled with many useful features for testing, such as mocking, code coverage, snapshot testing, and built-in assertions. This reduces the need for additional libraries or configurations.

While `Karma` is a widely-used testing framework and can be suitable for testing Angular applications, `Jest` offers a more modern and streamlined approach to testing, resulting in faster test execution, better developer experience, and built-in features that make testing Angular applications more efficient and straightforward. I provide the `Jest` tests mostly for all features that I develop in `MTAP` application and by doing that I earn I good code base so every time when I'm adding a new feature I have a lot of the tests that should be run in order to not broke some old functionality.

## 8.3 Gitlab CI/CD

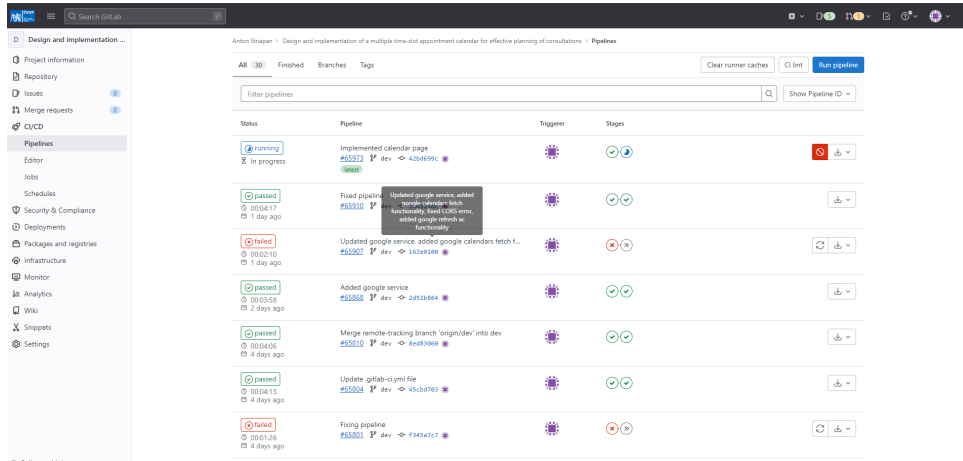


Figure 8.1: Gitlab CI/CD

In this figure, you can see the list of pipelines that were executed in the development. So every time when `git commit` is detected the TEST and BUILD pipeline is called. Firstly it runs the `test` target for both the backend and frontend parts, for example for the frontend that stage contains `lint`, `test`, `code format` checks. And after the successful TEST stage, it calls `build` target for frontend and backend parts. After successful `build` pipeline can be easily extended by the push to the docker registry and deploy pipelines.



## Chapter 9

### Conclusion

In the following section, I would like to sum up all the experiences and problems I faced in this project. In the beginning, the project was meant to be a common micro-services project, but due to the low amount of developers on the project, because it is only me, and my lack of experience in projecting and developing the micro-services architecture for the project, I decided to change into the more simple micro-services model. It will still have divided logic as usual micro-services but the logic of Api Gateway and Find-Service will be moved into User service. Also, I choose the database for my project - MongoDB. With this part, I didn't have any problems, because it runs under Docker container and I had experience working with it. Another component is the CTU API, it has several approaches to how to start, but none of them is entirely clear.

The first one is to use the official CTU gateway for authentication of students and teachers - SSO Gateway, but the main problem with this is a lack of documentation and communication. The second way is to use Zuul OAAS which is - in spite of the problems with documentation - still possible. I would prefer to use the SSO Gate because it looks like a more finished project than Zuul OAAS, but we will see what we will have access to. But unfortunately, I didn't manage to implement this type of authentication and it ended up with the Zuul OAAS authorization server. They provide the same functionality, but on my side, CTU gateway provides a better UI for user experience.

Also, I would like to sum up the done work about shared calendars and events. I studied some Google Calendar documentation about it and I've shared some basic practices in this document. The main idea of the application is to allow users to combine their calendars in one interface, to achieve this first application should allow the import of external calendars, in my project I would like to implement it in several ways, so user can import their calendar via a link or via \*.Ical format. But unfortunately, I didn't manage to implement import calendars from file and for now Google and CTU options are available. External calendars could be injected into the MTAP application with the same name as it was in the external system, but with `private` option, which can be changed immediately. All changes are stored locally and available to be exported via \*.ical format. The next version of the application can be

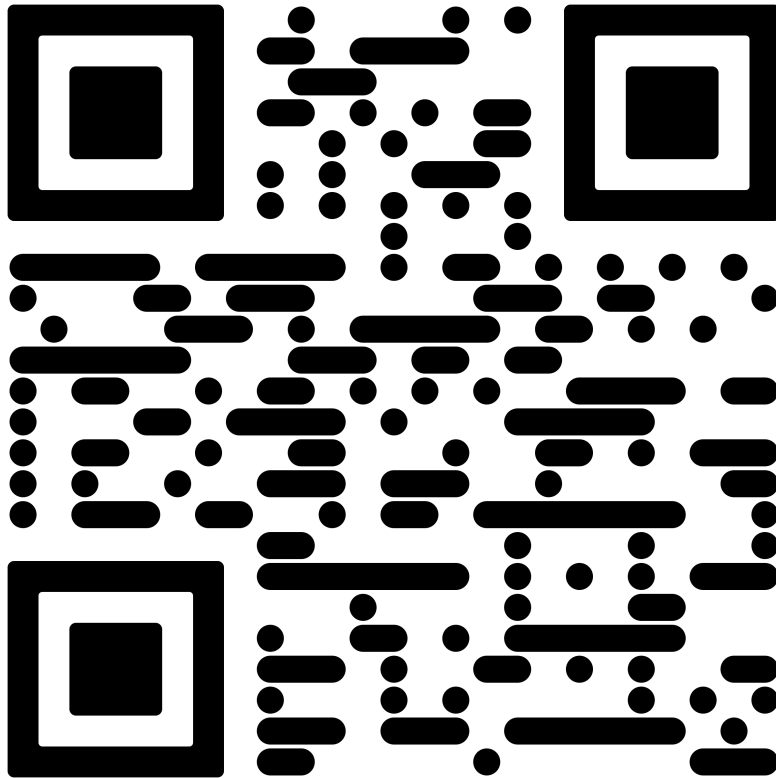
extended by `dynamic` option that will allow to dynamically update external calendars on the fly.

Also, I implemented an option of exporting a calendar as `*.Ical` file, because for better user experience I should allow users to extract calendars and use the tools that they are used to more. After exploring calendar and time planning literature I got a lot of ideas about which features should be must-have features, like automatic time slot calculation, better view for events and event members, calendar export, calendar sharing, calendar visibility, calendar import as a link, and which can be delayed - automatic export to Google Calendar or Outlook, advanced view modes.

This semester gave me a lot of experience and understanding of what steps should be done and when. I increased my skills in frontend development, even though this is my major specialization. I earned new experience in architecture, development, and delivery of software projects. I used new technologies in all steps of development, using docker container for the database storing, creating visible microservices with Eureka, using ngrx framework for better store management on the frontend side, and creating CI/CD pipelines that will allow to detect of all changes and test them for not breaking any functionality.

## Appendix A

### Application setup instructions



To start using the application firstly you should clone the git repository from QR code above (also you can click on it). Instructions how to run and serve the application you can find below.

## ■ A.1 How to run and serve application

### ■ A.1.1 With backend

- Open `application.properties` file for each of the projects (User and Calendar services)
- Set up you database there
  - You should have MongoDB installed or started MongoDB docker container
  - Add your credentials to line 20 in `application.properties` file
  - Add the name of the database to line 21, or leave it as it is
  - Run the docker command which is commented on the line 20 to initiate the database access
- Run the `CalendarApplication.java` in the `back/src/main/java/cz/cvut/fel/calendar` folder
- Run the `UserApplication.java` in the `back/src/main/java/cz/cvut/fel/user` folder
- You should have Node.js and npm package manager
- You can download Node.js from (<https://nodejs.org/en/>), it will install Node.js and npm
- Then run the `npm install -q @angular/cli` command in terminal window
- Open the `front` folder with your favorite ide/text editor (my choice is VS code)
- Use `npm I -force` command for installing all dependencies from `package.json`
- To start the application open the `package.json` and run the `start_local_network` command.









## Appendix B

### Glossary

**Access token** Access tokens are used in token-based authentication to allow an application to access an API. The application receives an access token after a user successfully authenticates and authorizes access, then passes the access token as a credential when it calls the target API.. 37

**Angular** Angular is an application-design framework and development platform for creating efficient and sophisticated single-page apps.. 26

**API** An application programming interface (API) is a way for two or more computer programs to communicate with each other. It is a type of software interface, that offers a service to other pieces of software.. viii, 8–11, 13, 17, 19, 52, 64, 67–69, 75

**Api Gateway** An API gateway is an API management tool that sits between a client and a collection of backend services.. 25, 75

**Calendar** A collection of events. Each calendar has associated metadata, such as calendar description or default calendar time zone. The metadata for a single calendar is represented by a Calendar resource.. 1, 2, 17–19

**CTU Password** CTU password is the main password for controlling access and protecting your information within the university and other involved reliable applications. 15

**Docker** is a set of the platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers.. 75

**DTO** is an object that carries data between processes. The motivation for its use is that communication between processes is usually done by resorting to remote interfaces (e.g., web services), where each call is an expensive operation.. 42

**Event** An event on a calendar containing information such as the title, start and end times, and attendees. Events can be either single events or recurring events. An event is represented by an Event resource.. 1, 2, 17–19

**HTTP** The Hypertext Transfer Protocol (HTTP) is an application layer protocol in the Internet protocol suite model for distributed, collaborative, hypermedia information systems.. 8, 17, 25, 37

**Many-to-one** A many-to-one relationship refers to one entity (typically a column or set of columns) that contains values and refers to another entity (a column or set of columns) that has unique values. 46, 47

**NGRX** is a library for Angular that implements the Redux pattern. It is used for managing state in Angular applications.. 21, 26

**Non Functional requirements** describe the general properties of a system. They are also known as quality attributes.. viii, 2

**OAuth** is the industry-standard protocol for authorization. OAuth 2.0 focuses on client developer simplicity while providing specific authorization flows for web applications, desktop applications, mobile phones, and living room devices. . 15

**One-to-many** is the most common kind of relationship. In this kind of relationship, a row in table A can have many matching rows in table B. But a row in table B can have only one matching row in table A.. 47

**ORM** in computer science is a programming technique for converting data between type systems using object-oriented programming languages. This creates, in effect, a "virtual object database" that can be used from within the programming language. There are both free and commercial packages available that perform object-relational mapping, although some programmers opt to construct their own ORM tools.. 23, 39, 40, 48

**Primary key** is the column or columns that contain values that uniquely identify each row in a table.. 48, 50

**Swagger** is a set of rules and tools for building and documenting RESTful APIs. It provides a standardized way of describing the structure and functionality of an API, allowing developers to understand and interact with it easily.. 11

**UML** is a general-purpose modeling language that is intended to provide a standard way to visualize the design of a system. 38



## Appendix C

### List of Abbreviations

- CRUD - Create Read Update Delete
- API - Application Programming Interface
- REST - Representational State Transfer
- HTML - HyperText Markup Language
- SCSS - Sassy CSS
- CSS - Cascade Style Sheets
- SQL - Structured Query Language
- JSON - JavaScript Object Notation
- JWT - JSON Web Token
- XSS - Cross-Site Scripting
- CSRF - Cross-Site Request Forgery
- DB - Database
- PM - Project Manager
- CORS - Cross-Origin Resource Sharing
- UI - User Interface
- ORM - Object Relation Mapping



## Appendix D

### Bibliography

- [1] CTU SSO About [online] <https://ist.cvut.cz/pojmy/vyhody-a-nevyhody-prihlaseni-ss/>
- [2] FIT Zuul OAAS Documentation [online] <https://rozvoj.fit.cvut.cz/>
- [3] Google Calendar API Documentation [online] <https://developers.google.com/calendar/api>
- [4] MongoDB Documentation [online] <https://www.mongodb.com/atlas/database>
- [5] Angular Documentation [online] <https://angular.io/>
- [6] Implementing a Shared Calendar. In: Practical Liferay. Apress. [chapter] [https://doi.org/10.1007/978-1-4302-1848-7\\_8](https://doi.org/10.1007/978-1-4302-1848-7_8)
- [7] Tullio, J., Mynatt, E.D. (2007). Use and Implications of a Shared, Forecasting Calendar. In: Baranauskas, C., Palanque, P., Abascal, J., Barbosa, S.D.J. (eds) Human-Computer Interaction – INTERACT 2007. INTERACT 2007. Lecture Notes in Computer Science, vol 4662. Springer, Berlin, Heidelberg. [book] [https://doi.org/10.1007/978-3-540-74796-3\\_26](https://doi.org/10.1007/978-3-540-74796-3_26)
- [8] KOS API documentation [online] <https://kosapi.fit.cvut.cz/>
- [9] Web API Design <https://offers.apigee.com/web-api-design-ebook/>
- [10] RESTful Java with JAX-RS (O'Reilly, 2009) <http://ww7.freshwap.net/ebooks/53935-restful-java-with-jax-rs.html>
- [11] RESTful Web Services Cookbook (O'Reilly, 2010) <http://ww7.freshwap.net/ebooks/239061-restful-web-services-cookbook-solutions-for.html>
- [12] Integrated user interface for effective utilization of multiple project management tools, Anton Striapan (2021) <https://dspace.cvut.cz/handle/10467/94713>