Bachelor's thesis

BUBEN CLUB RESERVATION SYSTEM

Artem Kuznetsov

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Marek Suchánek, Ph.D. et Ph.D.
May 15, 2025

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Buben Club Reservation System
Student:	Artem Kuznetsov
Supervisor:	Ing. Marek Suchánek, Ph.D. et Ph.D.
Study program:	Informatics
Branch / specialization:	Software Engineering 2021
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2025/2026

Instructions

Efficient room reservations for student activities present various challenges, where addressing these can significantly streamline the process for both users and administrators. One such challenge is ensuring smooth communication between systems that manage reservations and room access. The goal of this project is to develop a reservation system for the Buben Student Club that automates the entire room booking process. The application will integrate with the student information system (IS) of the Buben Club to retrieve necessary user data, add the created event to Google Calendar, and efficiently grant access to the reserved room.

- Analyze the domain of reservation services with a focus on managing various types of room reservation and calendars in it and build on the different rights in the club hierarchy. Use conceptual modelling to describe relevant processes and structure.
- Conduct a brief review of key solutions for reservation and access management systems.
- Compile a requirements catalog for the custom solution and prepare use cases for the application.
- Design the application based on the requirements and use cases. Consider the integration of external systems like student information systems (IS), Google Calendar and room access management via chip cards.
- Implement a prototype of the backend using the FastAPI framework for building APIs in Python, leveraging Python's type hints for clarity and robustness. Justify the selection of any additional technologies.

Electronically approved by Ing. Michal Valenta, Ph.D. on 26 November 2024 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

- Test and document the resulting solution.
- Evaluate the benefits for both club managers and users, and propose potential areas for future development.

Electronically approved by Ing. Michal Valenta, Ph.D. on 26 November 2024 in Prague.

I would like to thank my supervisor, Ing. Marek Suchánek, Ph.D. et Ph.D., for his expert guidance, help, patience, and valuable advice in the completion of this work. My deep gratitude also goes to my family and friends, who supported me both during my studies and throughout this project.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

I declare that I have used AI tools during the preparation and writing of my thesis. I have verified the generated content. I confirm that I am aware that I am fully responsible for the content of the thesis.

In Praze on May 15, 2025

Abstract

This Bachelor's thesis focuses on the development of a reservation system for the Buben Club, located in the Bubeneč dormitory. The thesis covers an analysis of existing solutions, system requirements, technology selection, application design, implementation, testing, and documentation. Potential future developments are outlined, and an evaluation of the achieved results is provided. The system automates the reservation process for club members, allowing them to authenticate via the club's information system and book available rooms. The design minimizes manual intervention, ensuring that reservation approvals—such as those required for night reservations or reservations exceeding allowed capacity—are handled directly within the application. The backend is developed using FastAPI and integrates with the Google Calendar API for managing reservations. The system is also connected to an access control system, enabling members to enter reserved rooms using their access cards instead of physical keys. Reservation rules are enforced based on user roles within the club. As a result of this thesis, a functional web application was developed, significantly streamlining the room reservation process for club members, reducing administrative overhead, and improving user experience.

Keywords web application, reservation system, backend, card access system, BUK, Python, FastAPI, IS.BUK, Google Calendar

Abstrakt

Tato bakalářská práce se zaměřuje na vývoj rezervačního systému pro klub Buben, který se nachází na koleji Bubeneč. Práce zahrnuje analýzu existujících řešení, systémové požadavky, výběr technologií, návrh aplikace, implementaci, testování a dokumentaci. Jsou nastíněny možné budoucí rozšíření a provedeno vyhodnocení dosažených výsledků. Systém automatizuje proces rezervace pro členy klubu, umožňuje jejich autentizaci prostřednictvím informačního systému klubu a rezervaci dostupných místností. Návrh minimalizuje nutné ruční zásahy a zajišťuje, že schvalování rezervací—například v případě nočních rezervací nebo rezervací překračujících povolenou kapacitu—probíhá přímo v aplikace. Backend je vyvinut pomocí FastAPI a je integrován s Google Calendar API pro správu rezervací. Systém je rovněž propojen s přístupovým systémem, který členům umožňuje vstup do rezervovaných místností pomocí jejich přístupových karet namísto fyzických klíčů. Pravidla pro rezervace jsou aplikována na základě rolí uživatelů v rámci klubu. Výsledkem této práce je funkční webová aplikace, která výrazně zjednodušuje proces rezervace místností pro členy klubu, snižuje administrativní zátěž a zlepšuje uživatelskou zkušenost.

Klíčová slova web aplikace, rezervační systém, backend, přístupový systém na kartu, BUK, Python, FastAPI, IS.BUK, Kalendář Google

Contents

List of Figures

List of Tables

List of code listings

List of abbreviations

ACS	Access Control Systems
API	Application Programming Interface
ASGI	Asynchronous Server Gateway Interface
BH	Better Hotel
CORS	Cross-Origin Resource Sharing
CRUD	Create, Read, Update, Delete
GC	Google Calendar
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
I/O	Input/Output
ISIC	International Student Identity Card
ISKAM	Information System of the Halls of Residence and Refectory
JSON	JavaScript Object Notation
LDAP	Lightweight Directory Access Protocol
ORM	Object-Relational Mapper
REST	REpresentational State Transfer
RFID	Radio Frequency Identification
RS	Reservation System
SU CTU	Student Union of the Czech Technical University
UI	User Interface
UML	Unified Modeling Language

Introduction

Many student clubs within the Student Union of the Czech Technical University (SU CTU) have dedicated rooms available for reservations by their members. However, the reservation process is largely manual, requiring members to send emails with repetitive information each time they wish to book a room. This not only creates an inefficient and time-consuming experience for users but also places a significant administrative burden on room managers, who must manually process and respond to reservation requests.

To address this issue, this thesis presents the development of an automated web-based Reservation System (RS) specifically for the Buben Student Club (hereinafter referred to as BUK) [1] at the Bubenec dormitory [2]. The system aims to streamline the booking process, making it more efficient and user-friendly for both club members and managers. The solution will integrate with the BUK's Information System (hereinafter referred to as IS.BUK) to automatically retrieve member details, reducing the need for repetitive data entry. Additionally, it will synchronize reservations with Google Calendar (GC), allowing users to manage their bookings seamlessly while providing managers with real-time scheduling updates.

A key feature of the system is its integration with the dormitory's and BUK's Access Control Systems (ACS). Approved reservations will automatically grant room access via Radio Frequency Identification (RFID) card authentication, eliminating the need for physical key handovers. This ensures a smooth and secure experience for users while reducing the workload for managers.

The development of this application is a collaborative effort. My colleague from the BUK is responsible for frontend development, while I focus on backend implementation. Together, we aim to create an intuitive and reliable system that enhances the accessibility and utilization of club facilities.

This thesis will document the design, development, and evaluation of the proposed RS, assessing its impact on operational efficiency, user convenience, and overall BUK management.

Chapter 1

Goals

The primary goal of this thesis is to develop a **web-based RS for the BUK** that automates the booking process, integrates with the club's IS.BUK, and enables card-based room access. This system will streamline reservations, ensuring compliance with club policies while reducing room manager workload.

To achieve this, the thesis will first analyze the current manual reservation process and its inefficiencies, identifying key challenges such as delays, lack of centralization, and the need for manual confirmation. Based on this analysis, the project will define the functional and technical requirements for an automated RS that integrate with **IS.BUK**, **GC** and **ACS**.

The design phase will focus on creating a user-friendly and accessible web application that allows members to book rooms. Key features will include role-based access control, automated restriction enforcement, real-time calendar event creation, and direct integration with RFID-enabled cards. This integration will ensure that approved reservations automatically grant room access via ACS, eliminating the need for manual key handling.

The implementation phase will involve developing a working prototype of the RS, integrating it with GC, IS.BUK and ACS. The development process will be thoroughly documented, and the system will undergo rigorous testing to validate its functionality and usability.

Finally, the thesis will evaluate the effectiveness of the developed solution, assessing its impact on the BUK's operations and user experience. The evaluation will determine how well the system improves reservation efficiency, automates ACS, and enhances overall management of club facilities.

Chapter 2
Analysis

In this section, the discussion will focus on the challenges associated with room reservations for student activities, particularly within the BUK [3]. The analysis will explore how the proposed RS can streamline the booking process, improve accessibility, and enhance overall efficiency. Additionally, existing solutions for room reservation and ACS will be reviewed, highlighting their strengths and limitations. This will provide a foundation for demonstrating how the proposed system offers improvements and fills gaps in current approaches, particularly through its integration with external systems like IS.BUK, GC, and chip-based ACS.

2.1 Introduction to BUK RS

I think room reservations play a significant role in managing student organizations, especially those like the BUK, which regularly hosts events, meetings, and various activities. The club's rooms are not just spaces; they are crucial resources that help attract new members and foster student engagement. Ensuring these rooms are efficiently managed is vital, as it directly influences the overall club experience.

The current manual reservation process introduces various challenges, including administrative overhead, miscommunication, and delays. In this thesis, the limitations of the current system are examined, followed by the proposal of a new solution designed to improve efficiency and user experience. This includes outlining the core objectives of the proposed RS, describing key technical components, and discussing both implementation strategies and anticipated challenges.

2.2 Overview of the BUK

The **BUK** [3] is a student-run organization dedicated to enhancing life at the **Bubenec Dormitory** [2], with the goal of making the environment more engaging and enjoyable for its 400 members. The club organizes a variety of activities aimed at enriching the student experience, including social events, recreational activities, and community-building initiatives.

One of the key responsibilities of the BUK is managing and maintaining various services and spaces within the dormitory, such as the computer network and the gym. Additionally, the club manages several rooms that serve as venues for various social events, gatherings, and recreational activities.

The club's activities are primarily driven by volunteers from within the membership, who contribute their time and enthusiasm to ensure the smooth operation of events and resources.

The **BUK** is part of **SU CTU** 2.1, a diverse student organization that supports a wide range of clubs and activities. SU CTU provides a platform for students to pursue their interests, whether it's organizing events, traveling, engaging in creative projects, studying, or participating in sports. SU CTU encourages students to get involved and fully experience all aspects of student life [4].

Figure 2.1 About Us – Studen Union CTU [4]

2.3 Current Challenges in Room Reservations

Despite the importance of room reservations to the smooth functioning of the club, the current process presents notable inefficiencies. At present, room bookings are handled through email correspondence. Users are required to submit reservation requests via Google Groups [5] associated with individual rooms. Each room has a dedicated page on the club's internal wiki, which outlines specific reservation rules. Managers manually review these requests, verify the necessary information, and subsequently create an event in a shared GC 2.2.

These calendars are publicly viewable, allowing club members to verify room availability via links provided on the BUK's wiki [1]. Once a reservation is confirmed, the manager and the requesting member coordinate a time for the physical handover of the room keys. Although the system is functional, it is dependent on human coordination and repetitive communication, which often leads to delays and misunderstandings.

Furthermore, the current RS must adhere to several important regulatory constraints. For instance, reservations are generally not permitted during nighttime hours. However, as a benefit for their contributions, active members of the club may be granted the privilege to make nighttime reservations. Similarly, individuals who have demonstrated reliability and have not caused any issues in the past may also be allowed to reserve rooms during these restricted hours. Unfortunately, this exception is not extended to all members, and such privileges are typically granted on a case-by-case basis, at the discretion of the club managers.

Additionally, if a planned event exceeds a certain number of participants—depending on the specific room being reserved—formal approval must be obtained from the dormitory administration. This process involves completing a registration form and receiving authorization from both the dormitory head and dormitory manager before the event can proceed. The same approval process applies to events organized by BUK itself, particularly when conducting larger-scale initiatives or public activities. These requirements add an extra administrative burden and extend the lead time necessary for planning certain events.

2.3.1 Manual and Inefficient Booking Processes

The email-based approach to room reservations imposes a considerable administrative burden on both users and managers. Users must repeatedly supply similar information, and managers often need to reiterate rules that are already published on the wiki. This lack of automation and centralization increases the likelihood of errors, miscommunication, and inefficiencies in handling high volumes of requests—especially during busy periods such as the start of semesters.

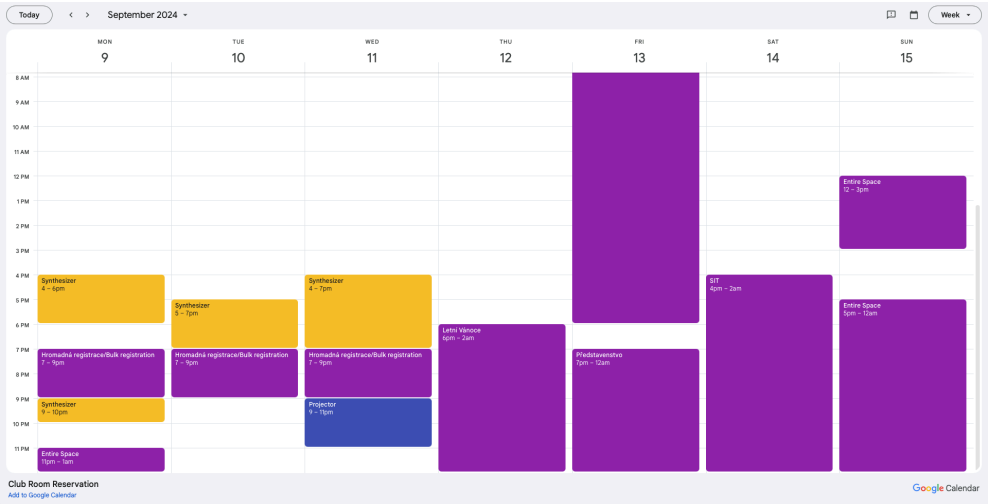

Figure 2.2 Club Room Reservations – Google Calendar [6]

2.3.1.1 Challenges for Club Members

1. **Finding Reservation Instructions:** New users may struggle to find up-to-date information on where to send their reservation requests and how to format their emails correctly for approval.
2. **Repetitive Data Entry:** Every time a reservation is made, users must manually enter the same details, which could be streamlined through automation.
3. **Uncertainty and Delays:** Users must wait for a manager's response, which may take a long time, causing uncertainty about the status of their reservation.

2.3.1.2 Challenges for Room Managers

1. **Manual Review and Response to Emails:** Managers must regularly check their emails and individually respond to each reservation request, which is time-consuming and prone to delays.
2. **Verification of Reservation Details:** Each request must be manually reviewed to ensure:
   a. The requested time slot is allowed under club policies.
   b. The room is available and does not conflict with other reservations.
   c. The user is an active club member and does not have any bans for the requested room.
   d. The requester's role within the club aligns with the reservation rules for that room.

3. **Reiterating Reservation Rules:** Depending on the room type and the time of reservation (e.g., stricter rules after 10 PM), managers often need to remind users of specific usage guidelines, leading to repetitive communication.

2.3.1.3 Proposed Solution

To address these issues, the implementation of an automated RS would significantly improve efficiency for both managers and users. Key features of such a system include:

1. **User Authentication and Data Storage:** The system should store necessary user information and authenticate users internally or through an external system. This would eliminate redundant data entry and ensure accurate user identification.
2. **Automated Data Validation:** The application would automatically check reservation details against predefined rules, preventing incorrect or conflicting reservations.
3. **Rule-Based Room Reservation:** Each room may have different rules and conditions for reservations, which the system can enforce automatically to ensure compliance.
4. **Role-Based Access Control:** The system would verify a user's role within the club and apply the appropriate reservation rules accordingly.
5. **Membership and Ban Status Verification:** The system would check a user's club membership status and ensure they do not have active bans before allowing them to make a reservation.
6. **Real-Time Reservation Tracking:** Once a reservation is made, it would be immediately recorded in the system, allowing users to track its status without waiting for a manual response from a manager.

By implementing these features, the proposed RS would reduce the administrative workload, minimize errors, and provide a more efficient and user-friendly booking experience.

2.3.2 Problem granting access

The activities of the BUK were temporarily suspended due to a long-term dormitory renovation that lasted several years [7]. Following the renovation, a significant technological upgrade was introduced—most dormitory doors, including those for club rooms, were fitted with electronic locks. These locks can be accessed via RFID [8] chips like International Student Identity Card

(ISIC) [9], allowing students to use the same card for university and dormitory access.

This upgrade presents an opportunity to automate room access management by integrating it with the RS. Ideally, once a club member makes a reservation, their ISIC card would be granted access to the reserved room automatically. The following section explores how this system benefits club members, the challenges involved in its implementation, and potential solutions.

2.3.2.1 Advantages for Club Members

If automatic access permissions are successfully implemented, the reservation experience for club members would improve significantly:

1. Members would no longer need to coordinate key exchanges when booking a room.
2. Access would be granted instantly at the time of reservation, eliminating wait times.
3. Members could be assured that only they, as the registered reservers, have access to the room during their reservation period.

2.3.2.2 Advantages for Room Managers

Automation would also greatly reduce the workload of club managers by:

1. Eliminating the need for managers to be physically present at the dormitory to handle key transfers before and after reservations.
2. Ensuring that only the authorized person receives access, as reservations would be tied to the user's personal card.
3. Reducing administrative overhead by automating verification processes, such as checking membership status and ban lists.

2.3.2.3 Challenges in Implementing ACS

The biggest challenge in implementing this system is gaining control over the dormitory's access management infrastructure. Currently, electronic locks are managed by the dormitory's IT department, and access is granted manually by emailing the responsible staff member. This process is inefficient, inconvenient for BUK members, and limits direct control for club administrators.

The dormitory currently utilizes ACS provided by the company SALTO 2.3, which supports RFID-enabled cards (such as ISIC) for room entry [10]. While the system is robust and widely adopted in institutional settings, its

centralized administration presents obstacles to integration with external applications. This limitation makes real-time access provisioning difficult and prevents seamless automation through the club's RS.

Figure 2.3 SALTO ACS [10]

Potential Solutions to Overcome These Challenges:

1. **Dormitory ACS:** Integration with the existing system
   a. The most optimal solution would be to integrate the BUK's RS with the dormitory's existing ACS via Application Programming Interface (API) access [11].
   b. While this requires cooperation with the dormitory's IT department, it ensures a reliable and unified approach to access management.
   c. The main drawback is dependency on the dormitory's IT team to provide and maintain API access, which could introduce delays.
2. **Implementing a separate BUK ACS:** Another approach would be to install independent electronic locks controlled directly by the club. However, this comes with significant challenges
   a. The cost of purchasing and installing electronic locks for all club rooms would heavily impact the club's budget.

b. The dormitory administration may not approve replacing their locks with a club-controlled system.

c. In emergency situations, dormitory staff may require access, which could be complicated if the locks are entirely managed by the club.

3. **Hybrid Approach:** Combining Both Systems

a. A practical solution could be to integrate both systems—leveraging the dormitory's infrastructure for primary access while implementing a club-managed system where additional control is required.

b. The dormitory's system would handle general room access, while the club's system could be used for securing specific assets, such as storage cabinets for board games or gaming consoles.

c. This approach balances security, efficiency, and feasibility while ensuring flexibility in areas not covered by the dormitory's system.

2.3.2.4 Conclusion

Automating access control would significantly enhance the reservation experience for both club members and managers. While there are challenges in integrating with the dormitory's existing system, a hybrid approach could provide an effective solution. The club's independent access management system will be explored further chapters.

2.4 Alternative Approaches to Room Reservation Management

Before developing a dedicated RS for the BUK, several existing solutions were explored as potential alternatives. While some of these solutions offer partial functionality for room booking and access control, they also present significant limitations that make them unsuitable for the club's specific requirements. This section examines the most relevant alternatives and evaluates their applicability.

2.4.1 Google Calendar as a Reservation System

One of the simplest and most widely used tools for managing reservations is GC 2.4. Initially, this approach was considered as a possible solution, where club members would submit booking requests, and managers would manually approve them by adding events to a shared calendar [6].

While GC provides a structured way to display reservations, its limitations quickly became apparent:

Manual approval process: Every request must be reviewed and manually entered by a manager, increasing administrative workload.

Lack of automated verification: There is no way to ensure that only eligible club members can book rooms. Additional verification steps would be necessary.

No direct access control integration: GC does not support linking reservations to digital keycard systems, meaning access to rooms would still require manual intervention.

Due to these constraints, GC is not a scalable solution for managing club room reservations efficiently.

However, despite these limitations, GC offers a powerful and well-documented API [12] that enables seamless programmatic interaction with calendars. While GC itself may not be suitable as the primary interface for room reservations, its API provides valuable integration opportunities. By leveraging the API, the system can automatically synchronize reservations, update availability in real-time, and take advantage of native calendar features such as reminders, event visibility, and user notifications. This hybrid approach combines the reliability of the GC infrastructure with the flexibility of a custom-built RS.

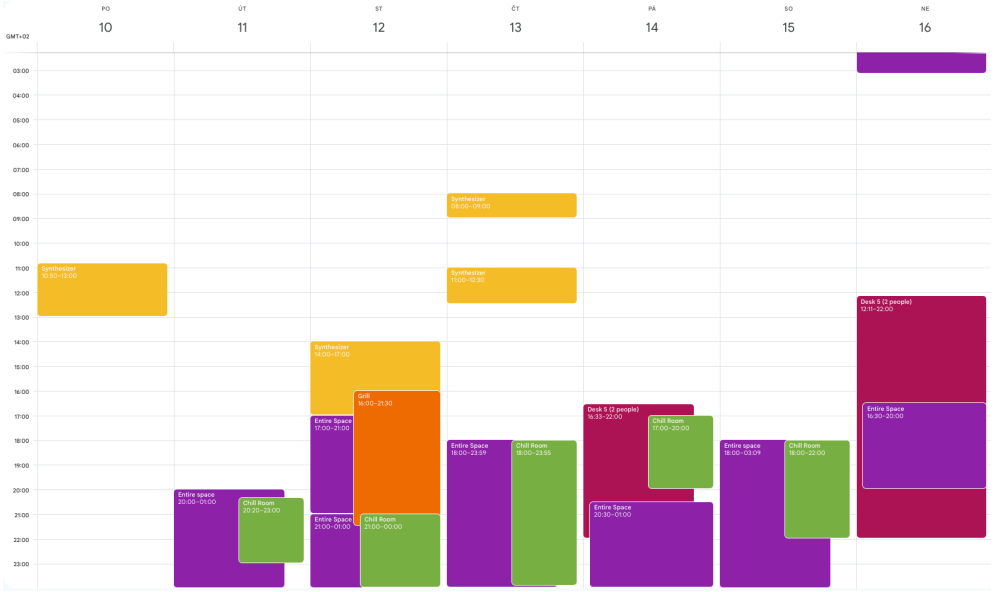

Figure 2.4 Google Calendar Interface [6]

2.4.2 RS in Other Dormitory Clubs

Several student clubs in similar dormitory environments have developed their own RS. Two notable examples are **Pod-O-Lee [13] (Olymp)** and **Silicon Hill [14] (has several RS)** , both of which have implemented custom booking solutions.

1. **Pod-O-Lee (Olymp) RS:**
   a. The system is based on an internal form where members submit booking requests.
   b. Managers manually review and approve each request, adding reservations to a calendar.
   c. Room keys are still handed over manually, meaning access control is not automated.
   d. The process remains dependent on human intervention, creating inefficiencies similar to those in GC-based reservations.
2. **Silicon Hill RS:**
   a. Silicon Hill operates a more advanced RS (for example, SHerna 2.5) that integrates with ACS.
   b. However, most reservations still require manual approval by managers before access is granted.
   c. Multiple separate RS exist for different club rooms, leading to fragmentation and complexity.
   d. The lack of a unified interface means members must navigate different platforms depending on the room they wish to book.

Although these systems offer partial solutions, they do not provide a fully automated and centralized approach, which is one of the key objectives of the proposed system.

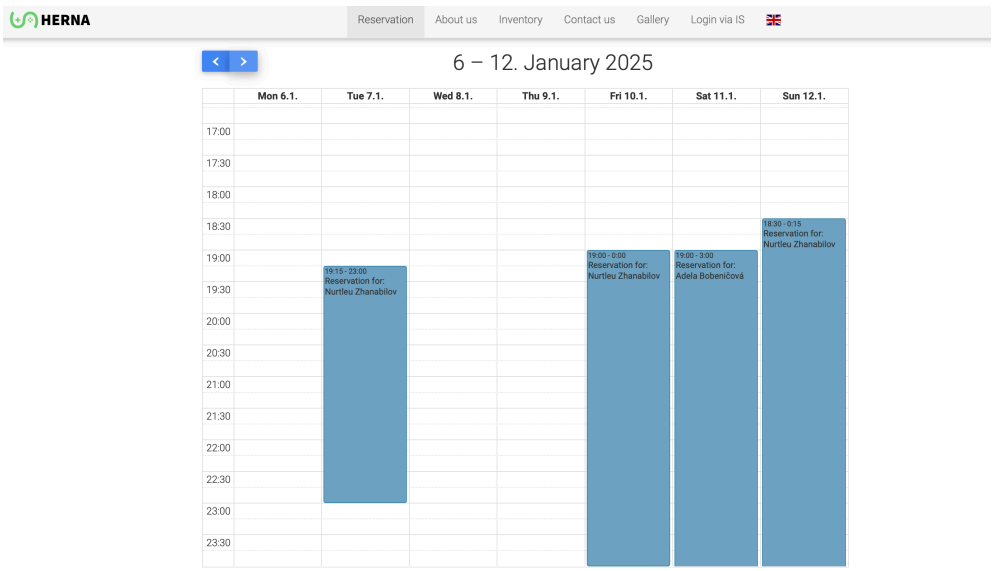

Figure 2.5 SHerna – Reservation [15]

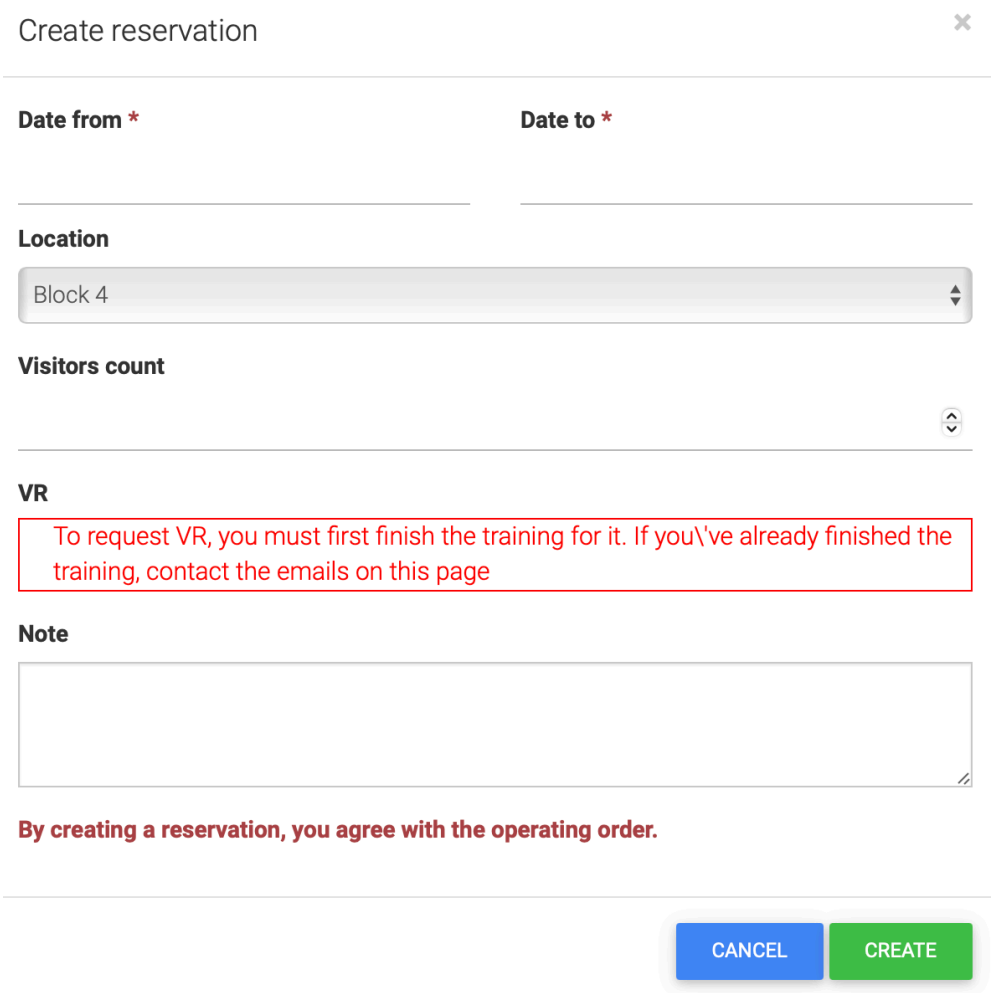

Figure 2.6 SHerna – Reservation Form [15]

2.4.3 Commercial and Third-Party Booking Systems

Another potential alternative is using a commercial booking platform, similar to those used by coworking spaces, hotels, or conference centers.

One such example is Better Hotel (BH) 2.7 by the company Mervis, a widely used system across the Czech Republic in hotels, hostels, apartments and etc. Based on personal experience working in a hostel that utilized this platform, it offers a comprehensive suite of tools—ranging from basic room reservations to advanced integrations with different ACS through external APIs. While the platform is feature-rich and highly modular, it is primarily tailored for hospitality businesses. It includes functionalities such as guest profile management, check-in/check-out processing, inventory control, invoicing, and automated reporting. However, these features far exceed the requirements of a simple room reservation system designed for a student organization. Additionally, BH is a commercial, subscription-based solution, which entails ongoing financial costs and maintenance overhead. Its design philosophy centers on serving revenue-generating businesses, not internal organizational scheduling or member-based access, making it poorly aligned with the needs of the BUK [16].

These systems typically offer:

- **Comprehensive booking management** with features such as availability tracking, automated reminders, and reporting.
- **Paid access control modules** that integrate with digital keycards or PIN-based entry systems.
- **Ongoing support and updates** managed by the software provider.

However, there are several drawbacks to adopting an external solution:

Lack of integration with the IS.BUK: There is no direct way to verify club membership status, requiring additional administrative oversight.

Limited customization and adaptability: If a new feature is required, the club would have to request changes from the software provider or find a different platform that meets its needs.

High costs: Most third-party solutions are subscription-based, making them financially impractical for a student organization.

Unnecessary features: Many of these systems include functionalities designed for businesses (e.g., invoicing, customer segmentation, marketing tools), which are irrelevant to a club room RS.

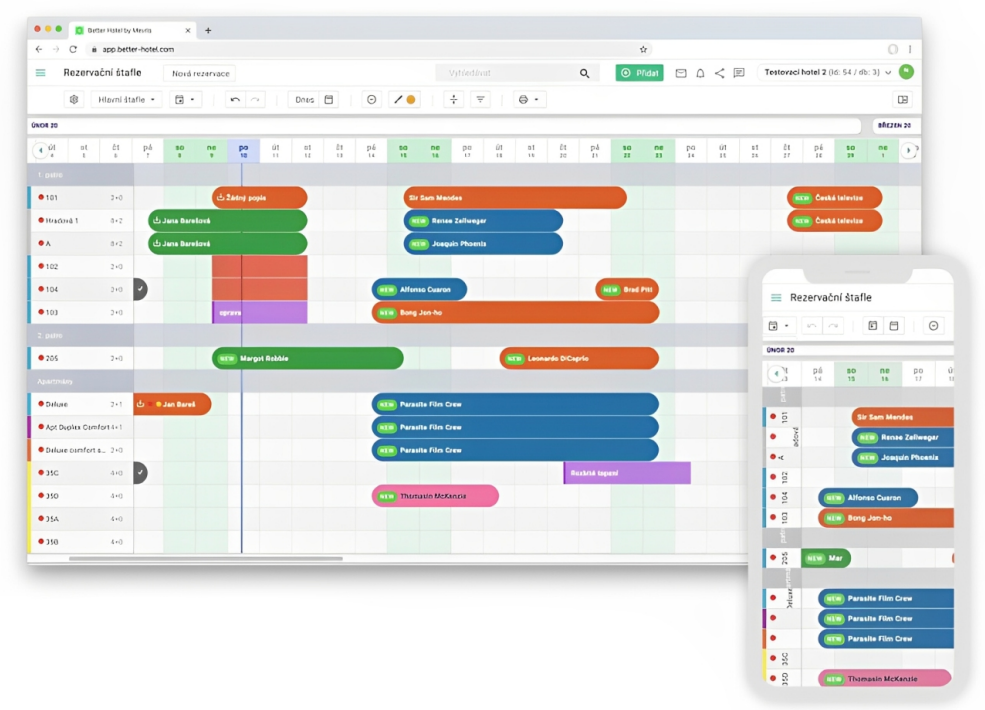

Figure 2.7 Better Hotel – Hotel System [16]

2.4.4 Limitations of Existing Alternatives

Most of the alternatives explored—whether manual, club-developed, or commercial—fail to meet the specific needs of the BUK for several reasons:

Lack of integration with the club's membership system: Without seamless authentication, additional administrative steps are required to verify bookings.

Absence of automated access control: Nearly all existing solutions rely on manual key handovers or separate approval processes, increasing inefficiencies.

Fragmentation and inconsistency: Some clubs have multiple separate RS for different rooms, while commercial platforms offer overly complex feature sets that do not align with club requirements.

Limited flexibility and control: Many solutions do not allow for easy modification or expansion of features, requiring external support for every change.

2.4.5 Why a Custom System is the Optimal Solution

Given these challenges, the development of a **dedicated club-controlled RS** emerges as the most effective approach. Such a system would provide:

- **A centralized platform** where all club room reservations are managed in a single interface.
- **Automated access control** by linking reservations to digital keycards, reducing manual intervention.
- **Seamless integration with the club's membership database**, ensuring only authorized users can book rooms.
- **Full customization and expandability**, allowing new features to be added as needed without reliance on third-party providers.

By addressing the limitations of existing alternatives and incorporating features tailored to the club's specific needs, a custom-built system ensures greater efficiency, automation, and user experience improvements compared to any of the available external solutions.

2.5 Requirements

Clearly defining system requirements is a critical step in software development, as it ensures that all essential functionalities and constraints are properly addressed. For this project, I will use the FURPS method [17] to systematically categorize requirements and the MoSCoW prioritization [18] to establish their priority levels. These approaches will help structure the development process efficiently, ensuring that the most crucial aspects are implemented while allowing flexibility for additional features if resources permit.

2.5.1 FURPS Method

The FURPS method 2.8 provides a structured way to classify both functional and non-functional requirements, ensuring comprehensive coverage of all critical system aspects. It consists of the following five categories:

Functional: Specifies the core capabilities and actions the system must perform to meet user needs.

Usability: Addresses the user experience, including interface intuitiveness and accessibility.

Reliability: Defines performance expectations in terms of stability, fault tolerance, and error recovery.

Performance: Establishes efficiency benchmarks, such as response times and processing speed.

Supportability: Covers aspects like maintainability, extensibility, and ease of debugging or updating.

By employing the FURPS method, I will ensure a clear distinction between different types of requirements, making the development process more structured and manageable.

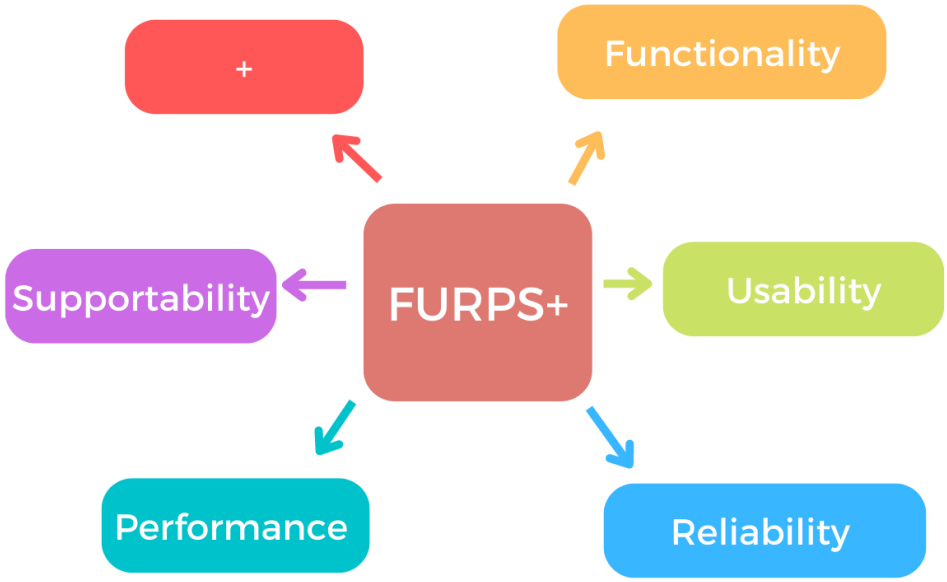

Figure 2.8 FURPS+ Method Diagram [17]

2.5.2 MoSCoW Prioritization

To determine the importance of each requirement, I will use the MoSCoW prioritization 2.9, which classifies features into four priority levels:

Must have: Essential requirements without which the system cannot function as intended.

Should have: Important features that significantly enhance the system but are not strictly necessary for basic operation.

Could have: Additional functionalities that would be beneficial but can be excluded if time or resources are limited.

Won't have: Features that are intentionally omitted from the current development phase but may be considered for future iterations.

This prioritization technique will guide decision-making throughout the project, ensuring that critical functionalities are developed first while maintaining adaptability for further enhancements if time permits.

Figure 2.9 MoSCoW Prioritization [18]

2.5.3 Functional requirements

Functional requirements define the essential operations and behaviors a software system must exhibit to fulfill its intended purpose. These requirements outline the core functionalities that enable the system to achieve its objectives, focusing on the interactions and processes users will engage with. To ensure clarity and precision, functional requirements should be explicitly defined, quantifiable, and testable, allowing developers to verify that the system performs as expected.

In the following functional requirements and throughout the subsequent text, three distinct user roles are referenced. These roles determine the level of access and the specific functionality available to a given user within the application:

- **Regular Member:** A standard user of the system, typically a club member with basic access to general functionalities such as authentication and room reservation.

- **Manager:** A designated individual responsible for a specific club room. In addition to the capabilities of a regular member, managers possess elevated permissions related to the configuration and oversight of the room under their management.
- **Section Head:** The head of the club section currently responsible for overseeing the BUK's RS. This role has the highest level of access and administrative privileges across the entire application. For generality and future-proofing, the specific section name is not stated, as the system may eventually fall under the responsibility of a different section or be renamed.

F01 – User Authentication (Must have): The system must integrate with IS.BUK via OAuth to authenticate users, ensuring secure identification and session management.

F02 – Log Out (Must have): The system must allow users to log out securely, terminating their session.

F03 – Role-Based Access Control (Must have): The application should enforce access control based on user roles (e.g., regular member, active member, manager, section head).

F04 – Create Service (Must have): Section head must be able to define and configure new services with attributes.

F05 – View Service (Must have): The system must allow all authorized users to access basic information about individual services.

F06 – List Services (Must have): The system must provide a browsable list of all services available to the user's role.

F07 – Edit Service (Must have): Section head must be able to update service attributes.

F08 – Delete Service (Must have): Section head must be able to soft-delete or hard-delete services.

F09 – Restore Deleted Service (Must have): Services removed via soft-delete must be recoverable by section head.

F10 – Create Calendar (Must have): Managers must be able to create calendars within services, configuring time slots, rules, and constraints.

F11 – View Calendar (Must have): All users must be able to view the available reservation types (calendars).

F12 – Edit Calendar (Must have): Managers must be able to modify calendar rules, time restrictions, and constraints.

F13 – Delete Calendar (Must have): The system must support both soft-deletion and hard-deletion of calendars by managers.

F14 – Restore Deleted Calendar (Must have): Managers must be able to restore calendars removed through soft-deletion.

F15 – Create Mini-Service (Must have): Managers must be able to create mini-services linked to a service (e.g., board games, grills and so on).

F16 – List Mini-Services (Must have): The system must provide a list of all mini-services per reservation type.

F17 – Edit Mini-Service (Must have): Managers must be able to update the details of mini-services.

F18 – Delete Mini-Service (Must have): Mini-services can be soft-deleted or hard-deleted by managers.

F19 – Restore Deleted Mini-Service (Must have): Previously soft-deleted mini-services must be restorable by privileged users.

F20 – Create Reservation (Must have): Users must be able to submit reservation requests for available services and time slots.

F21 – List Reservations (Must have): Users must be able to view a list of their current and past reservations.

F22 – Cancel Reservation (Must have): Users must be able to cancel their own reservations.

F23 – Request Reservation Modification (Should have): Users should be able to request changes to their reservations, subject to manager approval.

F24 – Approve Reservation (Should have): Managers should be able to approve or reject reservations that are pending due to user modifications or special constraints.

F25 – Real-Time Notifications (Should have): The system should notify users in real-time regarding reservation updates, approvals, and cancellations.

F26 – Reservation Confirmation (Could have): Users may receive confirmation emails upon successful reservation creation or approval.

F27 – Rental of Club Equipment (Won't have): The system will not initially include an equipment rental feature, but it may be considered for future implementation.

F28 – Multi-Language Support (Won't have): The system will not include multi-language support in its initial release but may incorporate this feature in future updates based on user demand.

2.5.4 Usability requirements

Usability requirements define the criteria that determine how intuitive, accessible, and user-friendly a software system is for its intended audience. These requirements focus on optimizing the user experience by ensuring that interactions with the system are efficient, seamless, and satisfying. A well-designed usability framework enhances user adoption, reduces errors, and improves overall system effectiveness. To achieve these objectives, usability requirements should be clearly specified, measurable, and aligned with established usability principles, ensuring that the system is both functional and easy to navigate.

U01 – Support for the most popular browsers (Must have): The app will be available in Google Chrome, Firefox and Safari.

U02 – Quick managers training (Should have): A new manager can learn how to use the app in half an hour.

2.5.5 Reliability requirements

Reliability requirements define the system's ability to function correctly and consistently under expected conditions, ensuring stable performance over time. These requirements focus on minimizing failures, handling unexpected errors, and maintaining system availability, all of which contribute to user trust and operational efficiency. A reliable system should be resilient to faults, capable of recovering from failures, and designed to prevent data loss or corruption. To ensure robustness, reliability requirements must be well-defined, measurable, and rigorously tested under various conditions.

R01 – System Uptime (Must have): The system must maintain a minimum uptime of 90% to ensure consistent availability and minimize service interruptions.

R02 – Fault Tolerance (Should have): The application must handle minor faults gracefully, preventing system-wide crashes and ensuring continuity of service.

R03 – Data Integrity (Should have): The system must safeguard against data corruption and ensure consistency, even during unexpected failures or shutdowns.

R04 – Error Recovery (Could have): The system should support automatic recovery mechanisms to restore functionality after transient errors or disruptions.

2.5.6 Performance requirements

Performance requirements define the expected speed, responsiveness, and efficiency of a software system under varying conditions. These requirements ensure that the system can handle user interactions and data processing within acceptable time limits while maintaining stability and scalability. Clearly specifying performance criteria allows developers to optimize system architecture, minimize latency, and enhance user experience. Performance requirements should be measurable, ensuring that the system meets predefined benchmarks for execution speed, load handling, and resource utilization.

P01 – System Response Time (Must have): The application must process user interactions and return responses within a maximum of 3 seconds under normal operating conditions.

P02 – Concurrent User Support (Must have): The system must support at least 10 simultaneous users without significant degradation in performance.

P03 – Data Processing Speed (Should have): Queries related to reservation management must execute within 1 second for a database containing up to 10,000 records.

P04 – Resource Efficiency (Could have): The application should optimize memory and CPU usage, ensuring that it operates efficiently on devices with limited computational power.

P05 – Scalability (Won't have): The system should be able to scale horizontally to accommodate increased traffic by dynamically distributing load across multiple servers.

2.5.7 Supportability requirements

Supportability requirements define the ease with which a software system can be maintained, extended, and diagnosed over its lifecycle. These requirements ensure that the system remains adaptable to future changes, facilitates troubleshooting, and supports efficient updates or enhancements. Well-defined supportability criteria improve system longevity, reduce maintenance costs, and enhance overall operational efficiency. By implementing structured logging, modular design, and comprehensive documentation, developers can ensure that the system remains scalable and manageable.

S01 – API Documentation (Must have): All external and internal APIs must be documented with clear specifications to support third-party integrations and future system modifications.

S02 – Automated Testing Framework (Must have): The application should include an automated testing framework to streamline regression testing and ensure system stability after updates.

S03 – Configuration Management (Should have): The system should allow administrators to adjust key settings (e.g., access permissions, notification preferences) without modifying the underlying code.

S04 – System Logging and Monitoring (Could have): The application must generate detailed logs for all critical operations, including user authentication and reservation management, to facilitate debugging and auditing.

2.6 Use cases

Given the scope of this project, Use Cases [19] have been defined in a structured list format, specifying key interactions between users and the system. The primary actors identified in the system are the user and the administrator. While both roles share common functionalities, the administrator possesses additional privileges for managing system settings and configurations.

By establishing well-defined use cases, this project ensures a clear understanding of the intended functionality, allowing for a systematic approach to software development that aligns with the needs of its users.

2.6.1 Regular club member

- UC01 – User Authentication and Authorization

  Description: The user logs into the system using OAuth authentication via the IS.BUK information system. Upon successful authentication, a session is created to identify the user within the application.

  Preconditions: The user must have valid credentials in the IS.BUK system.

  Postconditions: The user gains access to the application and its functionalities.

- UC02 – View Available Rooms and Services

  Description: The user accesses a list of reservable rooms and can view detailed information about each room, including available reservation types and mini-services.

- UC03 – Create Reservation

  Description: The user initiates a reservation by selecting a desired room, providing required details such as time and number of attendees, and selecting a reservation type. Optional mini-services can be added to enhance the reservation experience.

  Main Scenario:

  1. The user selects a room from the list of available rooms.
  2. The user provides required reservation details including date, start and end time, number of participants, reason for reservation, and contact email.
  3. The user chooses the desired reservation type from the options available.
  4. The user optionally selects additional mini-services (e.g., access to board games or consoles).
  5. The system performs validation checks, such as verifying the user's club membership status, ensuring that the selected time slot is available, and that the number of participants does not exceed the allowed limit.
  6. If all validations pass and no additional permissions are required, the reservation is successfully created and reflected in the GC.
  7. The user see a confirmation message indicating successful reservation.

  Alternative Scenarios:

  - **Validation Failure:** If the user does not pass system validation (e.g., inactive club membership, reservation during a restricted period, invalid data and so on), the reservation is blocked. The user is shown an error message detailing the issue.
  - **Night Reservation Without Privilege:** If the reservation occurs during restricted nighttime hours and the user does not have the privilege (e.g., not an active club member or manager), the reservation is submitted but marked as pending. It requires approval from the room manager before it can be confirmed.
  - **Reservation Exceeding Capacity Limit:** If the number of participants exceeds the maximum allowed for the room, an additional permission process is triggered. The user must fill out a registration event form which is automatically generated and sent via email (as a PDF attachment) to the room manager. The manager reviews the request and, if approved, forwards it to the dormitory head for final approval. Upon approval, the reservation is confirmed in the system. If denied at any stage, the user is notified and the reservation is rejected.

Post conditions: The reservation is either stored in the system as a confirmed booking or held in a pending state awaiting manual approval. Users are informed of the outcome through email notifications. If the reservation is confirmed, access rights are also granted in the ACS for the reserved service and mini-services.

- UC04 – Cancel Reservation

  Description: The user may cancel their existing reservation through the interface.

- UC05 – Request Modification to Reservation

  Description: The user submits a request to modify an existing reservation. Possible changes include updating the reservation time. This process allows flexibility while maintaining control through administrative oversight.

  Main Scenario:

  1. The user navigates to their list of active reservations.
  2. The user selects a reservation and initiates a modification request.
  3. The user updates reservation time.
  4. The modification request is submitted for manager review.
  5. The responsible manager receives the request and evaluates its feasibility based on availability, rules, and potential conflicts.

  Alternative Scenarios:

  - **Approval:** If the requested changes comply with reservation rules and are feasible, the manager approves the modification. The reservation is updated accordingly, and the user is notified of the successful change.
  - **Rejection:** If the modification request introduces scheduling conflicts, the manager may reject it. The user is informed of the rejection along with a reason or suggested alternatives.

  Postconditions: The reservation is either updated with the approved changes or remains unchanged if the request is denied. If the reservation is updated, corresponding access permissions in the ACS are also updated to reflect the new reservation details.

- UC06 – View Own Reservations

  Description: The user reviews a list of their upcoming and historical reservations.

2.6.2 Manager

- UC07 – Calendars and Mini-Services

  Description: Managers create, edit, and delete (soft delete) calendars and mini-services.

- UC08 – Approve Reservations Requiring Special Permission

  Description: Managers review flagged reservations that exceed normal constraints (e.g., night usage, high attendance) and determine whether to approve or reject them.

- UC09 – Cancel Reservations

  Description: Managers have the authority to cancel any reservation under their scope.

- UC10 – Respond to User Modification Requests

  Description: Managers assess user-submitted modification requests and update the reservation accordingly.

2.6.3 Section Head

- UC11 – Create and Modify Services

  Description: The section head may create, update and delete (soft-delete) new services existing ones throughout the reservation system.

2.7 Table of coverage of functional requirements of use cases

Functional requirements define the system's capabilities from an application perspective, whereas use cases describe interactions from the user's standpoint. Due to their close interrelation, it is essential to ensure that every use case aligns with at least one functional requirement and vice versa. To achieve this, a coverage matrix was utilized to systematically verify that all functional requirements are addressed through corresponding use cases 2.10.

-	UC01	UC02	UC03	UC04	UC05	UC06	UC07	UC08	UC09	UC10	UC11
F01	+										
F02											
F03	+										
F04											+
F05		+									
F06		+									
F07											+
F08											+
F09											+
F10							+				
F11		+									
F12							+				
F13							+				
F14							+				
F15							+				
F16		+									
F17							+				
F18							+				
F19							+				
F20			+								
F21						+					
F22				+	+				+		
F23					+				+		
F24								+		+	
F25				+	+			+	+	+	
F26			+								
F27											
F28											

Figure 2.10 Coverage of Functional Requirements of Use Cases

Chapter 3

Design

With a comprehensive analysis completed and the application's requirements clearly defined, the next step is to transform these insights into a robust system design. The design phase plays a crucial role in ensuring that the final product meets the expectations of its users while maintaining scalability, maintainability, and performance.

This chapter presents the overall design of the RS for BUK. It covers the selection of suitable technologies, programming language, and tools, as well as the definition of system architecture, data models, and database structure. The goal is to establish a solid foundation that supports reliable functionality, efficient development, and seamless integration with external systems. Emphasis is placed on building a well-structured, maintainable monolithic architecture that meets current requirements while remaining flexible enough to support future improvements with minimal refactoring.

3.1 Technology

This section provides an overview of the technologies used in the development of the RS. It covers the choice of programming language, backend frameworks, API integrations, and the development environment. The selected tools and technologies were chosen to ensure the system's maintainability, performance, and integration capabilities while aligning with my prior experience and the project's functional requirements.

3.1.1 Programming Language

Python [20] was chosen as the primary programming language for this project based on a combination of technical, practical, and experiential factors. The following subsections outline the key reasons behind this decision and provide an overview of both its advantages and limitations in the context of the RS.

3.1.1.1 Advantages of Python

Python was chosen as the most suitable language for the specific needs of this application, due to the following advantages:

Popularity and Community Support: Python is currently one of the most widely used programming languages in the worldz 3.1. This popularity ensures a large and active developer community, which contributes to a wealth of publicly available resources, documentation, and reusable code. For this project, such support plays an essential role in simplifying maintenance and ensuring that future developers can easily continue work on the application if necessary.

Rich Ecosystem and Library Support: The Python ecosystem includes a vast number of mature libraries and frameworks for tasks such as web development, data handling, API integration, and testing. This is particularly important for the current project [21].

Code Readability and Maintainability: One of Python's core strengths is its emphasis on simplicity and readability [22]. Its clean syntax allows developers to write concise and understandable code, which is especially beneficial in team environments or academic projects, where clarity is crucial. This facilitates easier debugging, code review, and future updates, ultimately reducing the technical debt of the application.

Compatibility with Selected Technologies: The decision to use FastAPI [23] as the backend framework and integrate services such as GC API [12] naturally pointed to Python as the most suitable language. Choosing a language that is natively supported by these tools avoids unnecessary complexity and maximizes development efficiency.

Personal Experience and Development Efficiency: I personal have prior experience in building web applications and collaborating in development teams using Python. This familiarity allows for the use of well-established best practices, the creation of effective code structures, and smooth navigation through the entire development process, from prototyping to deployment.

Platform Independence and Rapid Prototyping: Python is platform-independent and allows for rapid prototyping [24], making it a great choice for academic software. These characteristics enable iterative development and fast testing of ideas, which are valuable during the exploratory stages of building a custom system like the reservation platform.

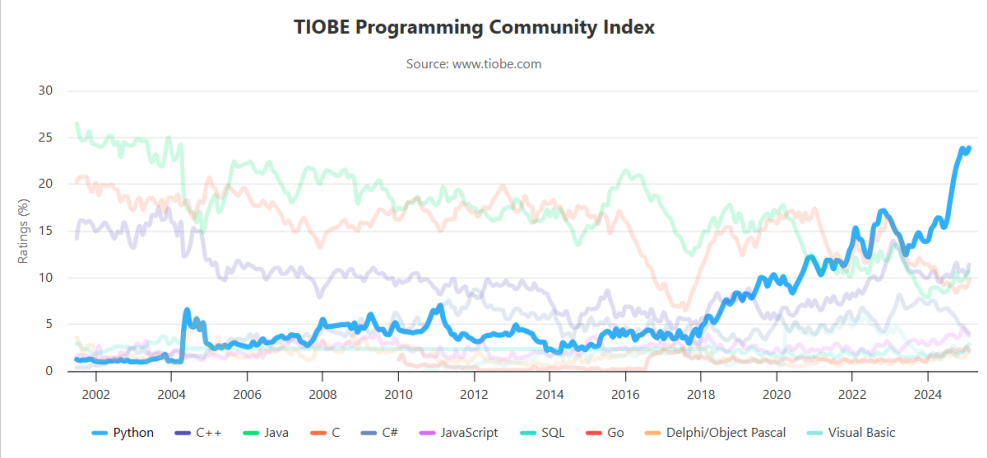

Figure 3.1 Most Popular Programming Languages [25]

3.1.1.2 Limitations of Python

While Python offers many benefits, it is important to acknowledge its limitations:

Execution speed: Python is an interpreted language and generally slower than compiled languages such as C++ or Java. However, the performance requirements of this project are minimal, as it is not designed for compute-intensive operations.

Dynamic typing: Python's dynamically typed nature may lead to type-related runtime errors. Nonetheless, the use of type hints, static analysis tools (e.g., mypy), and linters can significantly mitigate this risk.

3.1.1.3 Conclusion

Considering the language's popularity, ecosystem, readability, compatibility with selected tools, and the author's personal experience, Python was selected as the most appropriate choice for the development of the RS. Despite a few drawbacks, its strengths align well with the requirements of this project.

3.1.2 Frameworks

To ensure a scalable, performant, and maintainable architecture, several frameworks and libraries were selected for the development of the RS. These components work together to provide robust support for API design, database management, asynchronous communication, and system configuration. The following subsections describe the key frameworks used in the project and the rationale for their selection.

3.1.2.1 FastAPI: Core Web Framework

The core of the application is built on **FastAPI** [23], a modern, asynchronous web framework designed for building high-performance REpresentational State Transfer (REST) APIs [26]. FastAPI is based on the Asynchronous Server Gateway Interface (ASGI) [27] standard and is optimized for speed, type safety, and developer experience. It features automatic data validation and documentation generation using **Pydantic** [28] and **OpenAPI** [29] standards.

Key advantages of FastAPI:

Asynchronous programming support: FastAPI enables the development of highly concurrent applications, which is beneficial for systems that depend on external APIs or databases.

Type safety and validation: Integration with Pydantic allows for type-checked request and response models, reducing bugs and improving code clarity.

Auto-generated documentation: Built-in Swagger UI [30] interfaces are created automatically, easing testing and collaboration with other developers.

Minimal boilerplate: FastAPI requires less code compared to traditional frameworks while achieving high functionality.

These features make FastAPI particularly suitable for rapid development of scalable backend services and align well with the goals of the RS.

3.1.2.2 Supporting Frameworks and Libraries

The RS is built on FastAPI and supported by additional frameworks that enhance functionality, simplify development, and ensure robust performance.

Several auxiliary frameworks complement FastAPI in this project:

Starlette: A lightweight ASGI framework that powers FastAPI's underlying infrastructure. It provides routing, background task management, and WebSocket support [31].

SQLAlchemy: A powerful Object-Relational Mapper (ORM) used for database modeling and interaction. It simplifies communication with the relational database, ensuring clean and maintainable data access layers [32].

Alembic: A database migration tool used alongside SQLAlchemy. It allows version control of database schema changes and ensures consistent migrations across environments [33].

FastAPI-Mail: A mailing library designed to integrate seamlessly with FastAPI. It provides asynchronous email functionality with templating via Jinja2, used here for confirmation messages and administrative notifications [34].

Uvicorn: A lightning-fast ASGI server that serves as the runtime environment for the FastAPI application. Uvicorn is built on uvloop and httptools, providing excellent performance for asynchronous Python applications. It is essential for deploying FastAPI in both development and production environments, ensuring non-blocking Input/Output and support for concurrency [35].

3.1.2.3 FastAPI Alternatives

While evaluating frameworks, **Django** and **Flask** were also considered due to their popularity and broad community support.

Django [36] is a high-level web framework that follows the "batteries-included" philosophy, offering a wide range of built-in features such as an ORM, admin interface, authentication system, and form handling. It is particularly suited for developing large-scale, monolithic applications with a strong emphasis on convention over configuration.

Flask [37], in contrast, is a lightweight and flexible micro-framework that provides only the essential tools to build web applications. It allows developers to add extensions as needed, making it highly customizable and easier to understand for smaller projects.

However, FastAPI was chosen for the following comparative reasons:

Asynchronous support: Unlike Django (which only recently introduced limited async support) and Flask (which is traditionally synchronous), FastAPI is built natively for asynchronous operations, making it more efficient for concurrent request handling.

Modern architecture: FastAPI uses Python type hints for request and response models, which results in cleaner, safer code. Django and Flask require more boilerplate and additional packages for equivalent functionality.

Documentation and developer experience: FastAPI automatically generates interactive API documentation using OpenAPI, unlike Flask and Django which require third-party tools (e.g., Swagger or Django REST framework (DRF) for Django).

Performance: Benchmarks show that FastAPI can outperform both Flask and Django in request handling throughput, particularly under high concurrency [38].

Lightweight but structured: Like Flask, FastAPI remains lightweight and modular, but unlike Flask, it encourages a more structured approach to application design.

In contrast:

- **Django's** monolithic nature and built-in features can introduce unnecessary complexity for smaller, API-focused applications.
- **Flask**, while simpler, lacks built-in tools for async support, data validation, or automated documentation, which would require extensive integration of third-party libraries to match FastAPI's capabilities.

However, some limitations of FastAPI include:

Smaller ecosystem: Unlike Django, FastAPI does not offer a complete full-stack solution out of the box (e.g., admin panel, ORM, authentication).

Steeper learning curve for async programming: Developers unfamiliar with asynchronous paradigms may face initial challenges.

3.1.2.4 Conclusion

FastAPI and its supporting tools were selected for their modern design, performance, and developer-oriented features. Their synergy enables rapid development of reliable and maintainable backend services, tailored to the specific requirements of the BUK RS. The flexibility, asynchronous capabilities, and compatibility with other components make FastAPI the optimal choice for this project.

3.1.3 API Integrations

To ensure seamless functionality and automation, this project integrates several external systems through their APIs. These integrations are essential for providing robust user authentication, reservation management, and physical access control. The three primary integrations utilized in this project include:

IS.BUK: The internal information system of the Buben club, used for user authentication, role identification and service verification.

GC API: A reliable and scalable solution for managing reservation data.

ACS: Integration with dormitory and club-level access control systems to provide automated physical access to reserved rooms.

Each of these integrations plays a critical role in maintaining system efficiency, security, and reliability.

3.1.3.1 IS.BUK

The **IS.BUK** serves as the central platform for user management within the club. This system stores essential data such as member profiles, current membership status, assigned roles, and service affiliations [39].

Integration with IS.BUK allows the RS to:

- Authenticate users securely using existing login credentials.
- Access user role and service information to determine appropriate permissions.
- Verify active membership status before processing reservation requests.
- Automatically link reservations to the appropriate user for tracking and accountability.

By leveraging IS.BUK, the application avoids duplicating identity and permission management logic, maintaining consistency across club systems and reducing administrative overhead.

3.1.3.2 Google Calendar API

The **GC API** is used to manage the scheduling and storage of reservaation events. Rather than building a custom calendar system from scratch, the application uses Google's mature, reliable, and continuously maintained calendar infrastructure [12].

Key advantages of integrating with GC include:

Event and calendar management: The application facilitates comprehensive management of events and calendars through the API, allowing the creation, editing, deletion, and organization of events, as well as managing calendar configurations directly through the API.

Resilience and data backup: In the event of a local system failure (e.g., network outage at the dormitory), reservation data remains accessible via the Google account.

Cross-platform accessibility: Authorized user (main manager) can view, modify, or cancel reservations directly through the GC web or mobile interface if needed.

Advanced calendar features: The API supports recurrence, time zone handling, and rich metadata, enhancing the flexibility of the RS.

In the future, the integration with GC will be further expanded to eliminate the need for direct interaction with the GC User Interface (UI). All actions related to event and calendar management will be fully managed within the RS, ensuring a more streamlined and efficient user experience without requiring users to access GC's interface.

3.1.3.3 Access Card System

The final core integration involves the ACS, which manages physical access to reserved rooms and additional club resources, such as lockers and secondary areas.

Two independent systems are planned for integration:

Dormitory access system: Managed by the dormitory administration, this system controls entrance to common facilities.

Club-managed access system: A more flexible system operated internally by the BUK, used to control access to club-specific areas.

Dormitory Access System: The dormitory's official access control API is still under development. A test version is available and was used to design and prototype the integration within the reservation system. Although full integration is not yet possible, the backend will supports the required communication logic [40].

The test API communicates over HTTPS using JSON-formatted POST requests, authenticated via an API key. It supports essential operations such as adding or removing access rights and retrieving assigned permissions. Once the production API is finalized, only minor adjustments will be needed to complete the integration.

Club-Managed Access System: Integration with the club's internal ACS is implemented via IS.BUK, which supports external API-based authorization. While IS.BUK handles basic access rights, it cannot restrict entry strictly to reservation times. To address this, an external API is configured to perform real-time checks against the RS.

When access is granted locally, IS.BUK sends a POST request to the external API with the user's UID, room ID, and device ID. The API must return a boolean response, where only a literal `true` permits access; any other response is treated as a denial. Whether this API will be handled directly by the RS or through an intermediate service will be defined based on implementation constraints.

3.1.4 Development Environment and Tools

A well-structured development environment is crucial for ensuring productivity, consistency, and maintainability throughout the software development lifecycle. For the implementation of the BUK's RS, several industry-standard tools were used to support version control, coding, testing, and dependency management.

Integrated Development Environment (IDE): The application was developed using **PyCharm**, a widely adopted Python IDE developed by JetBrains [41]. PyCharm provides advanced code editing, real-time syntax checking, refactoring tools, and seamless integration with version control systems and virtual environments. These features significantly improve the efficiency of writing, debugging, and maintaining Python code.

Version Control: **Git** was employed as the version control system for tracking changes to the codebase [42]. The project repository is hosted on Silicon Hill club **GitLab** [43], which supports collaborative development, issue tracking, and continuous integration workflows. Git enables the creation of development branches, ensuring that experimental features and bug fixes are isolated before being merged into the main branch.

Virtual Environment: To maintain a clean and reproducible development setup, the project utilized **Conda** [44], a powerful package and environment manager. Conda allows for the creation of isolated virtual environments, ensuring that the project dependencies remain consistent across different machines and deployment stages. This approach minimizes compatibility issues and simplifies the installation of required packages.

Containerization and Deployment: **Docker** was used to containerize the application and its dependencies, facilitating a consistent deployment environment across development, testing, and production stages [45]. Docker enables the application to run in isolated containers, reducing the risk of environment-related issues and ensuring reproducibility. It also simplifies collaboration among developers by allowing them to work with a unified runtime configuration.

Summary: The use of PyCharm, Git, GitLab, and Conda contributed to a stable and manageable development workflow. These tools ensured that code quality, collaboration, and dependency control were maintained throughout the project.

3.2 Domain Conceptual Model

Before proceeding with the implementation, it is essential to define a domain conceptual model that structures the key entities and their relationships within the system. This model is based on the system requirements and serves as a foundation for the database structure and business logic. By clearly outlining the entities involved, their attributes, and interconnections, the model ensures that the system aligns with the needs of its users.

The primary entities in the RS are User, Reservation Service, Mini Service, Calendar, and Event. Each of these plays a crucial role in managing the reservation process while ensuring a seamless user experience.

To represent the conceptual model, a UML (Unified Modeling Language) class diagram is used [46]. UML provides a standardized way to visualize the structure and relationships between entities in a system. Its clear graphical notation improves communication between developers and stakeholders, and helps ensure the design aligns with the domain logic. Using UML also makes the model easier to understand, extend, and maintain throughout the development process 3.2.

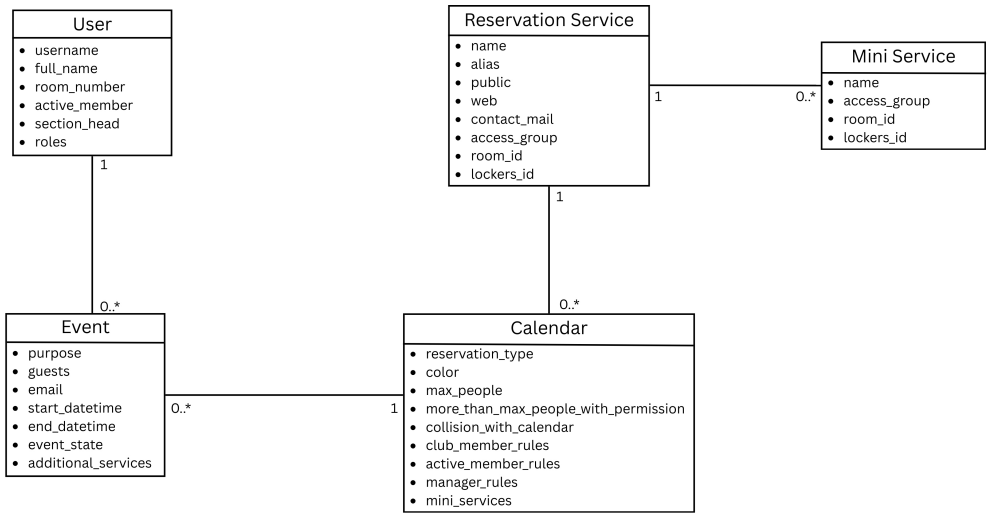

Figure 3.2 Domain Conceptual Model [46]

3.2.1 User

The User entity represents individuals who interact with the RS. Unlike traditional user models, this system does not store sensitive personal data but instead retrieves user information from the club's **IS**. Each user is uniquely identified by their ID, which matches their IS identifier rather than being generated by the database 3.1.

Additionally, users may possess **different roles** depending on their roles in the IS.BUK. These roles define the scope of access and control each user has within the system. In particular, they determine which parts of the RS the user can manage. This is especially relevant when a club member is assigned as a manager of a specific reservation service, thereby granting them administrative privileges in this.

Attribute name	Description
id	Unique identifier (corresponding to IS.BUK ID)
username	The name used within the RS
full_name	The user's full name, retrieved from IS.BUK
room_number	The user's assigned room number, retrieved from IS.BUK
active_member	Boolean flag indicating whether the user is an active club member (i.e., contributes to club development)
section_head	Boolean flag indicating whether the user holds a managerial position in a specific club section
roles	An optional list of roles assigned to the user, which may define additional permissions

Table 3.1 Description of User Attributes

Relationship	Description
User → Events	A user can create multiple events, each representing a reservation.

Table 3.2 Relationships of a User

3.2.2 Reservation Service

The **Reservation Service** entity represents the room or space that users can reserve within the club. Each service has a set of attributes that determine its availability, visibility, and additional information 3.3 3.4.

Attribute name	Description
id	Unique identifier for the reservation service
name	Name of the service (e.g., Study Room, Gaming Room, Club Lounge)
alias	Alternative or short name used for internal purposes
public	Boolean flag indicating whether the service is accessible to all club members or restricted to certain users (e.g., active members only)
web	A URL linking to additional information about the service (e.g., club wiki or other documentation)
contact_mail	Email address for inquiries related to the reservation service
access_group	Name of the access control group for syncing with the dormitory card system
room_id	Identifier of the room where card readers is located (used in the club's access system)
lockers_id	List of reader IDs located within the room specified by `room_id`

Table 3.3 Attributes of a Reservation Service

Entity	Relationship Description
Reservation Service → Calendars	A reservation service can have multiple calendars, defining different types of reservations available for that space
Reservation Service → Mini Services	A reservation service can also have multiple mini services, which provide additional functionality or access to specific features within the reserved space

Table 3.4 Relationships of a Reservation Service

3.2.3 Mini Service

A **Mini Service** represents a **secondary service or access privilege** associated with a reservation. These services are optional and can be selected during the booking process to grant additional permissions to the user 3.5 3.6.

Examples of Mini Services:

- **Bar Access** – Grants permission to use the club's bar.

- **Board Game Cabinet** – Allows access to a collection of board games stored in the room.
- **Console Area** – Enables the use of gaming consoles available in the club.

Attribute name	Description
id	Unique identifier for the mini service.
name	Name of the mini service.
access_group	Name of the access control group for syncing with the dormitory card system
room_id	Identifier of the room where card readers is located (used in the club's access system)
lockers_id	List of reader IDs located within the room specified by `room_id`
reservation_service_id	Foreign key linking the mini service to a specific Reservation Service.

Table 3.5 Attributes of a Mini Service

Relationship	Description
Belongs to Reservation Service	Each Mini Service belongs to a single Reservation Service, meaning it is only available when reserving that particular space.

Table 3.6 Relationships of a Mini Service

3.2.4 Calendar

The **Calendar** entity defines **specific reservation types** within a given **Reservation Service**. It allows for multiple reservation options within the same space, ensuring **flexibility** and **efficient resource utilization** 3.7 3.8.

For example, in a club room, users may reserve:

- **The entire room** for exclusive use.
- **A specific section** (e.g., a **billiards table**, **projector area**, or **gaming station**).
- **A shared space** (e.g., an individual study desk in a study room).

Each **Calendar** is linked to a **Google Calendar** for synchronization and scheduling purposes.

Attribute name	Description
id	Unique identifier matching the corresponding GC ID.
reservation_type	Defines the type of reservation (e.g., “Entire Room,” “Billiards Table,” “Study Desk”).
color	The color assigned to this calendar in the RS interface.
max_people	The maximum number of people allowed per reservation.
more_than_max_people_with_permission	Boolean flag determining whether exceeding the max limit is allowed with special permissions.
collision_with_itself	Boolean flag indicating whether overlapping reservations of the same type are allowed.
collision_with_calendar	A list of other calendar IDs with which this calendar cannot have overlapping reservations.
club_member_rules	Rules governing reservations for regular club members.
active_member_rules	Rules defining special permissions for active club members.
manager_rules	Rules specifying privileges for section heads and managers.
reservation_service_id	Foreign key linking the calendar to a specific Reservation Service.
mini_services	A list of associated Mini Services that can be selected during booking.

Table 3.7 Attributes of a Calendar

Relationship	Description
Belongs to Reservation Service	Each Calendar belongs to a single Reservation Service.
Linked Mini Services	A Calendar can have multiple Mini Services linked to it.
Calendar → Events	A calendar can have multiple events, each representing a specific reservation made for a time slot within that calendar.

Table 3.8 Relationships of a Calendar

3.2.5 Event

The **Event** entity represents an **individual reservation** made by a user for a specific time slot within a Calendar. Events store detailed information about the purpose, timing, and scope of the reservation. This entity is central to managing all reservations in the system.

Attribute name	Description
id	Unique identifier for the event.
purpose	The stated reason or activity planned for the reservation.
guests	Number of guests attending the reservation.
email	Usr contact email (for notifications or coordination).
start_datetime	The start time and date of the reservation.
end_datetime	The end time and date of the reservation.
event_state	The approval status of the event (e.g., Not Approved, Update Requested, Confirmed, Canceled).
user_id	Foreign key referencing the User who created the event.
calendar_id	Foreign key referencing the Calendar associated with the reservation.
additional_services	Optional list of additional services.

Table 3.9 Attributes of an Event

Relationship	Description
Belongs to User	Each event is created and owned by a single User.
Belongs to Calendar	Each event is associated with one Calendar that defines the reservation context.

Table 3.10 Relationships of an Event

3.2.6 Conclusion

The domain conceptual model structures the key entities within the RS and their relationships. It ensures that:

- **Users** are authenticated via the club's Information System while maintaining privacy.
- **Reservation Services** represent different rooms or spaces available for booking.
- **Mini Services** provide additional functionality during reservations.
- **Calendars** allow for multiple reservation types within a space and ensure seamless integration with GC.
- **Events** (reservations) will be linked to users and calendars, enabling a structured and automated booking system.

This model establishes the foundation for implementing a **flexible, efficient, and automated RS** tailored to the needs of the student club.

3.3 Backend

The backend of the RS is implemented using a modular monolithic architecture [47]. While the application is packaged and deployed as a single unit, its internal organization follows a clear modular structure. Functional components such as user authentication, reservation logic, calendar integration, and ACS are encapsulated in separate packages or folders. This organization improves code maintainability, supports separation of concerns, and simplifies future refactoring or scaling efforts.

3.3.1 Structure

The backend of the RS will follow a modular monolithic architecture. Although the application will be deployed as a single unit, its internal structure will be organized into logically distinct modules that encapsulate specific functionalities, such as authentication, reservation handling, GC synchronization, and ACS integration.

The system will implement an asynchronous web service model using the FastAPI framework. Incoming Hypertext Transfer Protocol (HTTP) [48] requests will be processed by endpoints defined in the `api` module. This module will serve as the primary entry point for client interactions, responsible for routing, request validation, and formatting.

These endpoints will then delegate control to the corresponding functions within the `services` module, which will contain the core business logic. This separation will allow the application logic to remain independent of the routing logic, thereby enhancing testability and maintainability. The service layer will manage internal operations such as data manipulation, orchestration of workflows, and coordination with the data layer.

Direct communication with the database will be abstracted through the `crud` module. This module will encapsulate all low-level operations related to persistent storage, ensuring that database access is consistently implemented and decoupled from the business logic. This design will promote code reuse, reduce duplication, and simplify unit testing of the logic layer.

In addition, a separate `db` module will manage the database session lifecycle and configuration, while the `models` module will define the SQLAlchemy ORM models representing the database schema. The `schemas` module will contain Pydantic models that define the structure of requests and responses, serving as a validation layer between the client and the internal logic.

The application will also include a `migration` structure powered by Alembic to support version-controlled schema changes.

Communication with external systems—such as the GC API or the ACS—will be explicitly handled within the routing layer (`api`) rather than within the business logic. This design will ensure that integration logic remains localized, simplifying debugging and maintaining a clear separation of concerns.

Responses will be serialized and returned to the client in a structured JSON [49] format. The entire backend will be built upon the ASGI [27] standard, which will enable non-blocking Input/Output (I/O) operations and facilitate efficient handling of concurrent requests, ensuring high throughput and system responsiveness under load.

3.3.2 Static Analysis and Code Quality

To support long-term maintainability and early detection of potential issues, the backend design will incorporate static analysis tools as an integral part of the development workflow. Specifically, the system will utilize **pylint** and **mypy**, two widely adopted tools for analyzing Python code without executing it.

Linting: **Pylint** is a static code analysis tool that checks for coding standard violations, detects code smells, and offers suggestions for improving readability and maintainability [50]. It enforces a consistent coding style throughout the codebase, helping to reduce human error and improve collaboration among developers. By integrating `pylint` into the development pipeline, the project will benefit from automated quality control and early feedback on code quality.

```
backend/

    app/
        api/                Entry point for routing and HTTP endpoints
        core/               Core configuration and startup
        crud/               Database access layer (CRUD operations)
        db/                 Database initialization and session
        migration/          Alembic-generated migration scripts
        models/             ORM models (SQLAlchemy)
        schemas/            Pydantic models for validation
        services/           Business logic layer
        templates/          Templated responses or emails (if any)
        main.py             Entry point of the FastAPI app

    tests/                  Unit and integration tests
    docker-compose.yml      Development container orchestration
    dockerfile              Container build specification
    environment.yml         Conda environment specification
    README.md
```

Figure 3.3 Project Structure of the Backend Application

Type checking: **Mypy** is a static type checker for Python that verifies the consistency of type annotations [51]. Given the dynamic nature of Python, type-related bugs may go unnoticed until runtime. The adoption of `mypy` will mitigate such risks by ensuring that function inputs, return values, and variable assignments conform to the declared types. This will enhance the reliability of the code and improve developer confidence, especially when working with asynchronous operations and complex data structures.

Integration: Both tools will be configured via project-level configuration files (e.g., `.pylintrc`, `mypy.ini`) to enforce project-specific conventions. These tools will be executed during local development and as part of the continuous integration pipeline to ensure consistent enforcement of standards across all environments.

By integrating static analysis tools into the backend, the project will promote code clarity, reduce technical debt, and support the development of a robust and maintainable codebase.

3.3.3 API Documentation

Comprehensive and accessible documentation is an essential part of a modern web application, particularly for systems that expose a public API. The backend of the RS will utilize FastAPI's built-in support for automatic API documentation based on the OpenAPI [29] specification. This feature will allow the system to generate interactive documentation interfaces (Swagger UI and ReDoc) without requiring additional development effort.

FastAPI leverages Python's type hints and Pydantic models to infer input and output schemas, validation rules, and response structures. These elements will be annotated directly in the route handlers and schema definitions, ensuring that the resulting documentation remains consistent with the actual API behavior.

The generated documentation will be accessible via the default FastAPI endpoints `/docs` (Swagger UI) [30] 3.4 and `/redoc` (ReDoc) [52], providing interactive interfaces for testing and understanding API behavior. These tools will serve as both a development aid and a form of technical documentation for users and future maintainers.

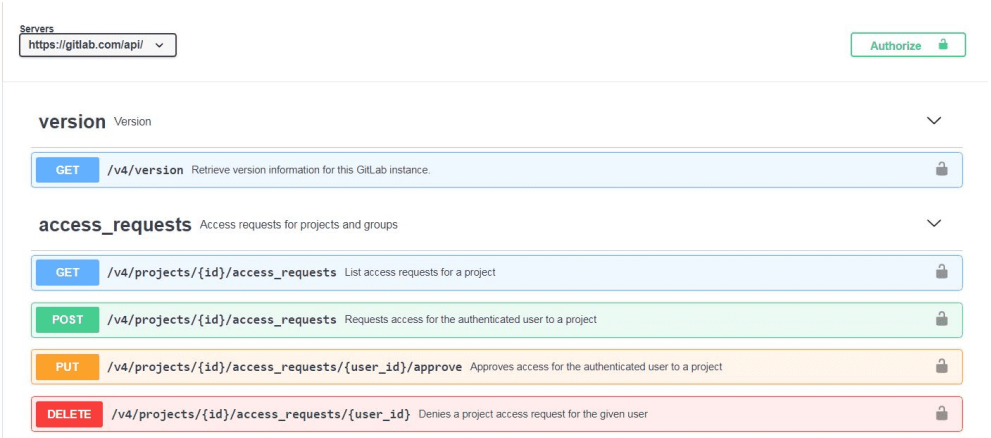

Figure 3.4 Swagger UI API Documentation [30]

3.4 Frontend

The frontend of the RS is developed as a separate web application, maintained by a fellow member of the BUK. While this thesis focuses on the backend implementation, the frontend forms an integral part of the complete system and is therefore included in the design overview.

3.4.1 Technology Stack

The frontend of the RS will be built using the **React** framework [53], a widely adopted JavaScript library for creating user interfaces. Development will be facilitated by **Vite** [54], a modern build tool known for its fast development server and efficient hot module replacement. Styling will be managed using **Tailwind CSS** [55], which enables rapid UI development through utility-first classes.

A notable part of the planned frontend architecture will be its integration with the **FullCalendar** library [56] 3.5, which will be used to render and manage reservation events. This library will directly connect to the **GC API** using a client-side API key. By offloading rendering and event management to a mature, battle-tested library, the system will avoid redundant backend requests and improve the responsiveness and reliability of the user experience.

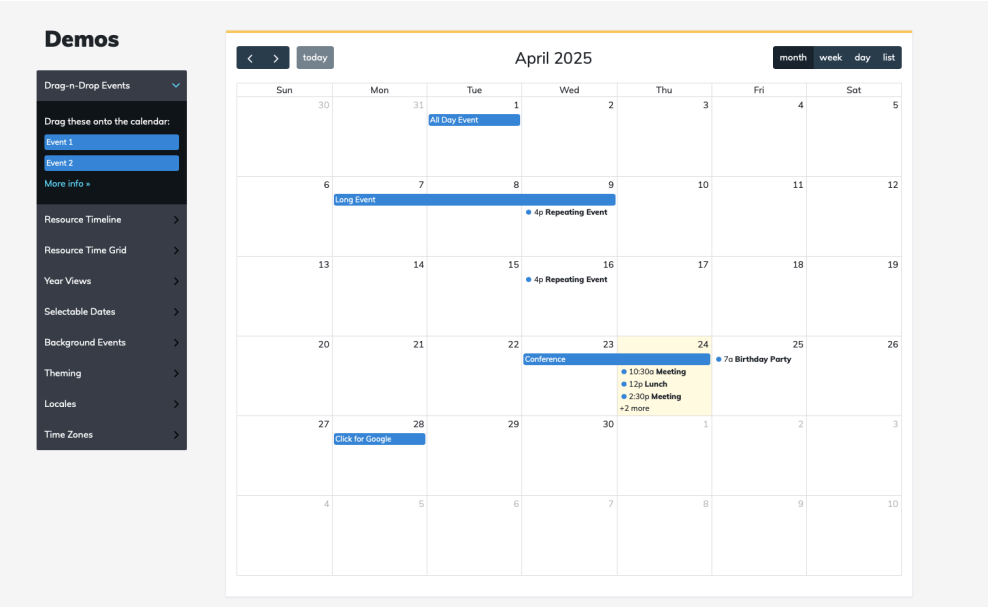

Figure 3.5 FullCalendar Demos [56]

3.4.2 Collaboration with Frontend Developer

Although the frontend and backend will be developed independently, close collaboration between both parts will be essential to ensure a smooth and reliable integration. As the initiator of the project and author of the backend implementation, I will also coordinate the feature planning and API design with my colleague, who will be responsible for the frontend development.

To maintain consistency and avoid integration issues, we will agree to meet regularly after the implementation of significant backend functionality. These meetings serve to review progress, align on user requirements, and discuss feasible frontend implementations. During these sessions, I propose feature specifications and request specific functionalities, while my colleague provides feedback on how they can best be realized from the frontend perspective.

In addition to synchronous planning, we will utilize asynchronous collaboration methods. I will document implementation requests and suggestions as **GitLab issues** in the frontend repository [57]. Conversely, my colleague will refer to the OpenAPI-based documentation [29] generated automatically by the backend to understand the structure of endpoints, request formats, and response schemas.

This collaborative approach—combining proactive coordination, structured issue tracking, and self-updating documentation—will ensure that both frontend and backend evolve in tandem, reducing the risk of mismatches and streamlining the overall development process.

3.4.3 User Experience Goals

The frontend design will prioritize ease of use, minimal cognitive load, and quick access to key functionalities such as booking, viewing current reservations, and checking room availability. The interface will be designed to be mobile-friendly and will adhere to modern accessibility guidelines, ensuring a smooth experience for all user types, including both regular members and room managers.

Chapter 4

Implementation

This chapter outlines the concrete realization of the system design introduced in the previous chapter. While the design phase focused on architectural concepts, technology selection, and modular organization, this chapter is concerned with how those decisions were implemented in practice to form a functioning backend application.

The implementation process followed the modular structure laid out in the design, where each logical component—such as user authentication, calendar logic, or reservation management—was developed within its respective domain. The FastAPI framework provided a consistent and efficient base for structuring API endpoints, integrating external services, and organizing internal business logic.

Code development emphasized maintainability and clarity, adhering to asynchronous patterns enabled by the ASGI [27] standard. The routing layer was kept responsible for request validation and external communication (e.g., with IS.BUK or the Google Calendar API), while the service layer encapsulated core logic and orchestrated application workflows. Database interactions were delegated to the `crud` module, isolating persistence logic from business rules.

This chapter presents selected areas of the backend in greater detail, highlighting implementation patterns, design decisions, and illustrative code examples that demonstrate how the system meets its functional requirements in practice.

4.1 Core Application Setup

This section outlines the foundational components of the backend application and describes how the system is initialized and prepared for runtime. It focuses on the implementation of the core infrastructure that enables the reservation system to function reliably and efficiently. The goal is to provide a detailed

view of how essential elements—such as application instantiation, configuration, and database initialization—are integrated into the project.

Each of these components plays a critical role in ensuring the system's stability, extensibility, and maintainability from the moment it is started.

4.1.1 Application Lifecycle and Entry Point

The entry point of the backend application is defined in the `main.py` module, which is responsible for launching the FastAPI server and initializing key system components. This module orchestrates the setup of application routing, middleware configuration, exception handling, and startup events.

At the heart of the file is the instantiation of a **FastAPI** object, configured with metadata such as the application name, version, and OpenAPI [29] documentation tags. During startup, the application executes a `startup_event` function, defined as an asynchronous context manager. This function ensures that the application is properly initialized—most importantly by calling `init_db()`, which prepares the database schema before any HTTP requests are processed. However, this initialization approach is intended only for early development stages. In the later phases of implementation, the direct schema creation via `init_db()` is replaced with a more robust migration system based on Alembic, as will be discussed later in the text.

The application structure adheres to a modular approach by including API routers from dedicated modules such as `users`, `events`, `calendars`, and `reservation_services`. These routers encapsulate specific endpoints and route them to the appropriate logic layers, thereby supporting separation of concerns and improving code maintainability.

To support client interactions across origins, Cross-Origin Resource Sharing (CORS) is configured using `CORSMiddleware` [58]. Additionally, the use of `SessionMiddleware` enables secure session management with a secret key defined in the application configuration.

Finally, the application is launched using `uvicorn` [35] when the module is executed directly. The server settings—such as host and port—are retrieved from the centralized configuration module. This structure ensures flexibility and separation between environment-specific settings and core logic.

An excerpt from the actual application entry point is shown for reference in Code listing 4.1

This structure enables a clean and maintainable approach to launching the backend service, ensuring that all critical components are ready before the application begins serving requests.

```
app = FastAPI(
    title=fastapi_docs.NAME,
    description=fastapi_docs.DESCRIPTION,
    version=fastapi_docs.VERSION,
    openapi_tags=fastapi_docs.get_tags_metadata(),
    lifespan=startup_event
)

app.include_router(users.router)
app.include_router(events.router)
app.include_router(reservation_services.router)
app.include_router(calendars.router)
app.include_router(mini_services.router)
app.include_router(emails.router)
app.include_router(access_card_system.router)

app.add_exception_handler(BaseAppException, app_exception_handler)

app.add_middleware(SessionMiddleware, secret_key=settings.SECRET_KEY)
app.add_middleware(CORSMiddleware,
    allow_origins=["https://develop.reservation.buk.cvut.cz",
    "https://reservation.buk.cvut.cz", "https://is.buk.cvut.cz"],,
    allow_credentials=True,
    allow_methods=["GET", "POST", "PUT", "DELETE"],
    allow_headers=["*"],
)
```

Code listing 4.1 Main.py with FastAPI App Entry Point

4.1.2 Configuration and Settings Management

The configuration of the application is centralized in the **Settings** class located in the `core/config.py` module. This class inherits from `BaseSettings` provided by the `Pydantic Settings Management` framework, which allows for structured, type-safe, and environment-based configuration.

The settings include parameters for application metadata (e.g., name, host, port), database connection, OAuth credentials for third-party integrations (such as IS.BUK and GC), and email server configuration. Most of these values are expected to be loaded from external environment files, which are prioritized and located through the `get_env_file_path` function. This function dynamically generates paths to multiple environment files (`.env.dev`, `.env.secret`, and `.env`) to support development, sensitive credentials, and fallback defaults.

The settings include parameters for application metadata (e.g., name, host, port), database connection, OAuth credentials for third-party integrations (such as IS.BUK and GC), and email server configuration. Most of these values are expected to be loaded from external environment files, which are prioritized and located through the `get_env_file_path` function. This function dynamically generates paths to multiple environment files—`.env.dev`,

```
@validator("POSTGRES_DATABASE_URI", pre=True)
def assemble_db_connection(cls, value, values):
    if isinstance(value, str):
        return value
    return str(PostgresDsn.build(
        scheme=values.get("SQLALCHEMY_SCHEME", "postgresql"),
        username=values.get("POSTGRES_USER"),
        password=values.get("POSTGRES_PASSWORD"),
        host=values.get("POSTGRES_SERVER"),
        port=values.get("POSTGRES_PORT"),
        path=f'{values.get("POSTGRES_DB")}'
    ))
```

Code listing 4.2 Validator Method for Assembling the PostgreSQL Connection

`.env.secret`, and `.env`—to support different layers of configuration. Specifically, `.env.secret` contains sensitive credentials (e.g., API keys, passwords) that should not be committed to version control. To ensure confidentiality, the `.env.secret` file is listed in the project's `.gitignore`. The fallback `.env` file may be used as a catch-all or for local overrides.

The database URI is assembled automatically using a `@validator`, which constructs the full `PostgresDsn` string from individual settings such as host, port, user, password, and database name. This mechanism avoids hardcoding the full URI while maintaining readability and flexibility 4.2.

To make the configuration easily accessible throughout the application, a global instance of the **Settings** class is created at the end of the `core/config.py` module.

This instance is then imported wherever needed, enabling unified access to environment-specific parameters without duplication or tight coupling to specific modules. This design ensures consistency and simplifies configuration changes across the codebase.

4.1.3 Database Initialization and Models

The application utilizes SQLAlchemy [32] as the ORM to define and manage interactions with the relational database. A central aspect of this setup is the use of a shared **Base** class, defined via the `@as_declarative` decorator. This base class provides common functionality for all models and automatically derives table names from class names using the `__tablename__` attribute, configured via the `@declared_attr` decorator.

Database Initialization: The database connection is configured through SQLAlchemy's `create_engine` method, which uses a connection string assembled from environment variables. The function `get_db()` serves as a dependency-injected generator that manages session lifecycle during request

handling. For initial table creation during development, the `init_db()` function calls `Base.metadata.create_all()`, which inspects the metadata and generates corresponding SQL statements. Although this approach facilitates quick prototyping, it is intended to be phased out in favor of a migration-based workflow using Alembic, as discussed in a subsequent section.

Soft Deletion: To support reversible deletion, all core models inherit from a `SoftDeleteMixin`, which provides soft-deletion functionality through the `sqlalchemy_easy_softdelete` package [59]. Instead of permanently removing records from the database, this pattern marks records as deleted by assigning a `deleted_at` timestamp, thereby preserving data integrity and auditability.

ORM Models: The system defines several key domain entities using SQLAlchemy models. For example, the **ReservationService** model captures general information about a service, including its public visibility and contact details. It is related to two subordinate entities: **Calendar** and **MiniService**, which reference the service through foreign keys and bidirectional relationships.

The **Calendar** model is of particular significance, as it encapsulates reservation related metadata such as maximum allowed participants, collision configuration, and per-role reservation rules. These rules are implemented using a custom SQLAlchemy **TypeDecorator** (**RulesType**), which allows seamless serialization and deserialization of Pydantic models (specifically the **Rules** schema) into a JSON-based text column.

Each model is defined in a separate file within the `models` directory, promoting a modular and maintainable codebase architecture.

Example Model Definitions: A simplified example of the **ReservationService** and **Calendar** models is shown in Code listing 4.3

Through this design, the system achieves a maintainable and extensible data layer that aligns with the overall modular architecture of the backend.

Note on Future Changes: It is important to note that the database initialization logic and the structure of the ORM models described in this section reflect an earlier, synchronous stage of the application's development. As the system evolves, several architectural improvements are planned. Specifically, the backend will transition to fully asynchronous operation, leveraging an async-compatible database engine and restructured session management. Additionally, the ORM models will be refactored to align with the SQLAlchemy 2.0 declarative syntax. Schema initialization using `init_db()` will be deprecated in favor of a robust migration workflow managed by Alembic. These changes and their rationale will be addressed in a dedicated section later in this chapter.

```
class ReservationService(Base, SoftDeleteMixin):
    __tablename__ = "reservation_service"
    name: Mapped[str] = mapped_column(unique=True, nullable=False)
    alias: Mapped[str] = mapped_column(unique=True, nullable=False)
    public: Mapped[bool] = mapped_column(nullable=False, default=True)
    web: Mapped[str] = mapped_column(nullable=True)
    contact_mail: Mapped[str] = mapped_column(nullable=False)
    access_group: Mapped[str] = mapped_column(nullable=True)
    room_id: Mapped[int] = mapped_column(nullable=True)

    calendars: Mapped[list["Calendar"]] = relationship(
        back_populates="reservation_service", lazy="selectin")
    mini_services: Mapped[list["MiniService"]] = relationship(
        back_populates="reservation_service", lazy="selectin")
    lockers_id: Mapped[list[int]] = mapped_column(ARRAY(Integer),
    ↪  nullable=False, default=list)

class Calendar(Base, SoftDeleteMixin):
    __tablename__ = "calendar"
    id: Mapped[str] = mapped_column(primary_key=True)
    reservation_type: Mapped[str] = mapped_column(unique=True, nullable=False)
    color: Mapped[str] = mapped_column(default="#05baf5", nullable=False)
    max_people: Mapped[int] = mapped_column(default=0, nullable=False)
    more_than_max_people_with_permission: Mapped[bool] =
    ↪  mapped_column(nullable=False, default=True)
    collision_with_itself: Mapped[bool] = mapped_column(default=False,
    ↪  nullable=False)
    collision_with_calendar: Mapped[list[str]] = mapped_column(ARRAY(String),
    ↪  nullable=True)

    club_member_rules: Mapped[Rules] = mapped_column(RulesType(),
    ↪  nullable=True)
    active_member_rules: Mapped[Rules] = mapped_column(RulesType(),
    ↪  nullable=False)
    manager_rules: Mapped[Rules] = mapped_column(RulesType(), nullable=False)

    reservation_service_id: Mapped[UUID] = mapped_column(UUID(as_uuid=True),
                            ForeignKey("reservation_service.id"))

    reservation_service: Mapped["ReservationService"] = relationship(
        back_populates="calendars")
    events: Mapped[list["Event"]] = relationship(
        back_populates="calendar", lazy="selectin")
    mini_services: Mapped[list[str]] = mapped_column(ARRAY(String),
    ↪  nullable=True)
```

Code listing 4.3 Reservation and Calendar Models

4.2 User Authentication via IS.BUK

User authentication in the Reservation System is implemented via the OAuth2 protocol by integrating with the IS.BUK system, which serves as the central user identity and role management platform for the BUK. The authentication process is initiated from the `/users/login` endpoint, where the user is redirected to the IS.BUK authorization server using an OAuth2 session. Upon successful authentication, IS.BUK returns an authorization code that is exchanged for an access token.

This token is then stored in the session and used to query IS.BUK for the authenticated user's profile, associated roles, and services. These data points are used to construct or update a corresponding user record in the local database. The **UserService** layer is responsible for determining role-specific attributes such as active membership or managerial privileges by parsing the returned role and service assignments.

The callback logic, session management, and communication with external services are all handled at the API routing level (`api/users.py`), ensuring that the OAuth2 protocol remains isolated from internal service logic. This separation allows for better maintainability, as changes in the authentication provider's API or session handling can be localized without affecting core business operations.

The authentication flow includes the following steps:

1. The client initiates authentication by accessing `/users/login`, which redirects to the IS.BUK authorization URL 4.1.
2. Upon user consent, IS.BUK redirects back to `/users/callback` with an authorization code.
3. The backend exchanges the code for an access token and retrieves user information using asynchronous HTTP requests.
4. The received data is validated and mapped to internal models using Pydantic schemas.
5. The user record is either created or updated via the **UserService**, and the session is updated accordingly.

Authenticated users can then retrieve their identity through the `/users/me` endpoint, which validates the stored session and resolves the user object from the database.

Example Code Snippet. A simplified version of the authentication controller is shown in Code listing 4.4 to illustrate the login and callback logic.

```python
@router.get("/login_dev")
async def login_local_dev(request: Request):
    authorization_url = (
        f"{settings.IS_OAUTH}/authorize?client_id={settings.CLIENT_ID}"
        "&response_type=code&scope=location"
    )
    oauth = get_oauth_session()
    authorization_url, state = oauth.authorization_url(authorization_url)
    request.session['oauth_state'] = state
    return RedirectResponse(authorization_url)

@router.get("/callback")
async def callback(
        user_service: Annotated[UserService, Depends(UserService)],
        request: Request
) -> Any:
    oauth = get_oauth_session()
    authorization_response_url = modify_url_scheme(str(request.url), 'https')
    try:
        token = oauth.fetch_token(
            settings.IS_OAUTH_TOKEN,
            client_secret=settings.CLIENT_SECRET,
            authorization_response=authorization_response_url
        )
    except Exception as exc:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail=f"There's some problem with getting token. "
                   f"Control this url: {authorization_response_url}",
            headers={"WWW-Authenticate": "Bearer"},
        ) from exc

    request.session['oauth_token'] = token
    user = await authenticate_user(user_service, token['access_token'])
    request.session['user_username'] = user.username
    return {"username": user.username, "token_type": "bearer"}
```

Code listing 4.4 IS.BUK OAuth2 Login and Callback Handlers

This approach ensures that only verified club members with valid IS.BUK credentials are granted access to the system, and that their permissions are automatically synchronized with the club's internal role assignments.

Limitations of Role Management in IS.BUK: During integration with IS.BUK, a notable limitation was encountered regarding role management. IS.BUK provides a predefined and relatively inflexible set of user roles. Fortunately, assigning room managers posed no issue—each room was associated with a distinct service in IS.BUK, and the existing "service admin" role was sufficient to designate managers.

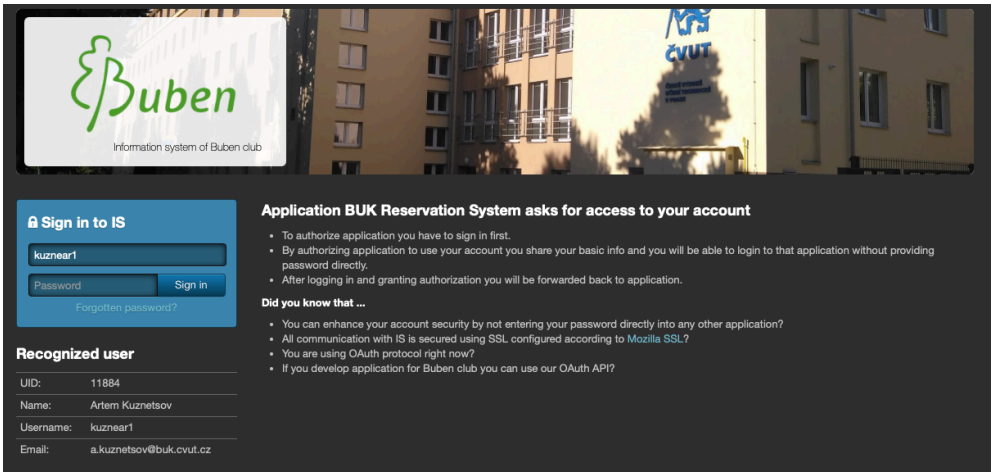

Figure 4.1 IS.BUK Authorization Flow [39]

However, more nuanced roles such as "active club member" and "section head" proved more challenging. Since custom roles cannot be defined, a workaround was implemented: a dedicated "active member" service was created, and during user synchronization, the presence of this service determines whether a user is considered active.

For the section head role, creating a dedicated service was not justifiable due to the one-to-one nature of the role. As a temporary solution, a note is added to the user's profile in IS.BUK, which only a limited number of trusted administrators can modify. While not ideal, this approach ensures a basic level of role validation until a better solution is developed.

Alternative mechanisms such as tags and Lightweight Directory Access Protocol (LDAP) groups [60] were considered, but ultimately dismissed due to critical limitations. Tags are an implemented feature within IS.BUK, but they are user-defined free-form strings that can be modified by any user with tag management privileges—without scope restrictions. This lack of control renders them unsuitable for secure role validation. On the other hand, LDAP groups would have offered a structured and secure solution for role representation. However, since the LDAP directory is accessible only in read-only mode, it is not possible to define or modify groups externally, which makes this approach unfeasible for integration purposes.

In the long term, role synchronization will require a more robust and secure solution. Extending IS.BUK is beyond the scope of this project, but introducing a dedicated role management API could offer centralized and consistent role governance across integrated systems.

4.3 REST API Endpoints

This section describes the design and structure of the RESTful API endpoints that form the external interface of the backend system. These endpoints enable clients—such as frontend applications or third-party systems—to interact with core entities and perform operations like creation, retrieval, modification, and deletion.

4.3.1 Entity Management Endpoints

The backend API exposes a set of endpoints for managing key domain entities such as **ReservationService**, **Calendar**, and **MiniService**. These endpoints follow RESTful conventions and are organized within the `api` module. Each endpoint is connected to the business logic layer through dependency-injected [61] service classes, ensuring clear separation of concerns.

For example, the `reservation_services` routes handle operations related to creating, retrieving, updating, and deleting reservation services. Only authenticated users with sufficient privileges—namely, those with the "Section Head" role—are authorized to perform write operations, such as creation or modification.

Endpoints in this category follow a uniform design pattern:

- **POST** routes allow creation of single or multiple entities using Pydantic schemas for input validation.
- **GET** routes retrieve entities either by ID, alias, or name, with optional support for including soft-deleted entries.
- **PUT** routes update existing entities, enforcing both validation and role-based access control.
- **DELETE** routes support both soft and hard deletion, depending on the provided query parameters.

The example is shown in Code listing 4.5 illustrates a typical endpoint structure using the **ReservationService** entity

These API definitions are annotated with OpenAPI-compatible metadata, allowing automatic generation of interactive documentation, which ensures that frontend developers and third-party consumers can discover and integrate the endpoints efficiently.

Access control logic and exception handling (e.g., **UnauthorizedException**, `PermissionDeniedException`) are consistently integrated across endpoints to enforce security and provide meaningful feedback for unauthorized operations.

```python
@router.post("/create_reservation_service",
             response_model=ReservationService,
             status_code=status.HTTP_201_CREATED)
async def create_reservation_service(
    service: Annotated[ReservationServiceService,
    ↪  Depends(ReservationServiceService)],
    user_service: Annotated[UserService, Depends(UserService)],
    user: Annotated[User, Depends(get_current_user)],
    token: Annotated[Any, Depends(get_current_token)],
    reservation_service_create: ReservationServiceCreate
) -> Any:
    reservation_service = await service.create_reservation_service(
        reservation_service_create, user
    )
    if not reservation_service:
        raise BaseAppException()
    await authenticate_user(user_service, token)
    return reservation_service
```

Code listing 4.5 Example: Reservation Service Creation Endpoint

4.4 Google Calendar Integration

An essential part of the Reservation System's functionality is its integration with GC, which serves as the platform for storing and managing reservation events. The integration allows the system to programmatically create, retrieve, and manage calendars and events while ensuring seamless synchronization between user reservations and the GC infrastructure.

4.4.1 Authentication with Google APIs

The system uses OAuth2 authentication to securely access the GC API. A custom `auth_google` function is responsible for managing the authentication flow. It first attempts to retrieve previously stored credentials from a local `token.json` file. If the credentials are invalid or missing, it triggers a local server-based OAuth2 authorization process, where new credentials are obtained from the Google authorization server using the project's client ID and secret. Once retrieved, the credentials are cached to the `token.json` file for future use, minimizing repeated authentication overhead.

This authentication mechanism enables the application to interact with Google services on behalf of a system-level account, rather than requiring individual users to authorize access. Example Authentication Code is shown in Code listing 4.6

```python
def auth_google(creds):
    if os.path.exists("token.json"):
        creds = Credentials.from_authorized_user_file("token.json")

    if not creds or not creds.valid:
        if creds and creds.expired and creds.refresh_token:
            creds.refresh(Request())
        else:
            flow = InstalledAppFlow.from_client_config({
                "installed": {
                    "client_id": settings.GOOGLE_CLIENT_ID,
                    "project_id": settings.GOOGLE_PROJECT_ID,
                    "auth_uri": "https://accounts.google.com/o/oauth2/auth",
                    "token_uri": "https://oauth2.googleapis.com/token",
                    "auth_provider_x509_cert_url":
                    ↪  "https://www.googleapis.com/oauth2/v1/certs",
                    "client_secret": settings.GOOGLE_CLIENT_SECRET,
                    "redirect_uris": ["http://localhost"],
                }
            }, settings.GOOGLE_SCOPES)
            creds = flow.run_local_server(port=0)

        with open("token.json", "w", encoding="utf-8") as token:
            token.write(creds.to_json())

    return creds
```

Code listing 4.6 Authentication with Google APIs

4.4.2 Calendar Creation and Management

When a new `reservation type` is introduced into the system, a corresponding GC can either be linked (if it already exists) or automatically created. During the calendar creation process, the system:

1. Initializes a GC API client using authenticated credentials.
2. If a calendar ID is provided, it verifies that the calendar exists in GC.
3. If no calendar ID is provided, it creates a new calendar in GC, setting attributes such as the calendar's summary (title) and timezone.
4. After creation, it assigns public read-only access to the calendar to ensure that all users can view event availability without needing individual Google authentication.

The newly created or validated calendar is then saved into the system's database for further event management. Example Calendar Creation Code is shown in Code listing 4.7

```
@router.post("/create_calendar")
async def create_calendar(service, user, calendar_create):
    google_calendar_service = build("calendar", "v3",
    ↪  credentials=auth_google(None))
    if calendar_create.id:
        try:
            google_calendar_service.calendars(). \
                get(calendarId=calendar_create.id).execute()
        except HttpError as exc:
            raise BaseAppException("This calendar not exist in Google
            ↪  calendar.",
                                   status_code=404) from exc
    else:
        try:
            calendar_body = {
                'summary': calendar_create.reservation_type,
                'timeZone': 'Europe/Prague',
            }
            created_calendar = (google_calendar_service.calendars().
                                insert(body=calendar_body).execute())
            calendar_create.id = created_calendar.get('id')

            rule = {
                'role': 'reader',
                'scope': {
                    'type': 'default'
                }
            }
            (google_calendar_service.acl().
             insert(calendarId=calendar_create.id, body=rule).execute())
        except HttpError as exc:
            raise BaseAppException("Can't create calendar in Google
            ↪  Calendar.") from exc

    calendar = await service.create_calendar(calendar_create, user)
    if not calendar:
        raise BaseAppException()
    return calendar
```

Code listing 4.7 Creating a GC

4.4.3 Reservation Posting to GC

When a user submits a reservation request through the frontend interface, the system validates the request against internal business rules (such as collision detection, maximum capacity, and night-time restrictions). If the request passes these checks, the backend posts the event directly to the relevant GC.

The flow for posting an event is as follows:

1. The system retrieves user services information by querying IS.BUK using the user's OAuth2 session.
2. It identifies the appropriate calendar based on the reservation type and fetches its associated reservation service.
3. A GC API client is initialized using the system credentials.
4. Collision detection is performed to ensure the requested time slot does not overlap with existing events.
5. An event object is constructed, embedding necessary metadata (such as reservation details and user information).
6. If the event violates specific rules (e.g., exceeding the allowed number of participants or booking during night hours), the event is still created but is marked as **Not approved** in the calendar and a notification email is sent to the user.
7. Otherwise, the event is posted to the GC, and an approval notification is sent.

This design delegates calendar event management to Google's infrastructure, ensuring reliability, real-time synchronization, and scalability without introducing significant overhead to the backend system. Example Event Posting Code is shown in Code listing 4.8

4.4.4 Summary

By tightly integrating with the GC API, the RS achieves real-time visibility, robust scheduling, and operational transparency for club members and managers. The solution leverages proven external services while maintaining control over access policies and internal club rules.

4.5 Email API

The Email API manages the creation and delivery of system emails related to reservations. It supports sending registration forms with PDF attachments and automated notifications for reservation events. All emails are sent asynchronously using FastMail to avoid blocking the request-response cycle.

Email content is rendered using Jinja2 templates with dynamically injected context based on event, user, calendar, and reservation service data. This ensures that recipients — including both users and club managers — receive relevant and well-formatted information.

```python
@router.post("/create_event")
async def create_event(service, calendar_service, user, token, event_create):
    services = ServiceList(services=await get_request(token,
    ↪  "/services/mine")).services
    calendar = await
    ↪  calendar_service.get_by_reservation_type(event_create.reservation_type)
    if not calendar:
        raise EntityNotFoundException(Entity.CALENDAR,
        ↪  event_create.reservation_type)

    reservation_service = await
    ↪  calendar_service.get_reservation_service_of_this_calendar(
        calendar.reservation_service_id
    )
    google_calendar_service = build("calendar", "v3",
    ↪  credentials=auth_google(None))

    if not control_collision(google_calendar_service, event_create, calendar):
        return JSONResponse(status_code=200, content={"message": "Collision"})

    event_body = await service.post_event(event_create, services, user,
    ↪  calendar)
    if not event_body or (len(event_body) == 1 and 'message' in event_body):
        return JSONResponse(status_code=200, content=event_body or {"message":
        ↪  "Error"})

    # Too many guests
    if event_create.guests > calendar.max_people:
        ...
        return {"message": "Too many people"}

    # Night-time reservation
    if not check_night_reservation(user):
        if not control_available_reservation_time(event_create.start_datetime,
        ↪  event_create.end_datetime):
            ...
            await preparing_email(service, event, create_email_meta(...))
            return {"message": "Night time"}

    # Normal reservation
    event_google_calendar = google_calendar_service.events().insert(
        calendarId=calendar.id, body=event_body
    ).execute()

    event = await service.create_event(event_create, user,
    ↪  EventState.CONFIRMED, event_google_calendar['id'])
    await preparing_email(service, event, create_email_meta(...))

    return event_google_calendar
```

Code listing 4.8 Posting an Event to Google Calendar

The API includes logic to clean up temporary files (e.g., generated PDFs) after sending, and supports sending different versions of the same message depending on the recipient role (e.g., member vs. manager). This makes it a robust and flexible component of the system's communication layer.

4.6 Access Card System Integrations

This section outlines the system's integrations with external ACS, which enable automatic room access management based on user reservations. Two separate integrations are described: one with the dormitory's centralized ACS, and another with the club's internal access system, implemented within IS.BUK. Both serve distinct operational roles but share the common goal of enforcing reservation-based access restrictions.

4.6.1 Dormitory Access System Integration

The backend implementation includes a prototype for interacting with the dormitory's ACS, currently available only in a test variation. To facilitate communication with this external system, a dedicated FastAPI router was developed, although most of its endpoints remain commented out and inactive by default.

The implemented communication is based on JSON-formatted `POST` requests, authenticated using an API key and sent to a predefined URL. The backend includes a utility function `send_request` responsible for constructing and sending these requests, as well as handling errors. This function is reused across different internal helper endpoints designed to simulate API calls such as:

- adding or removing a variable symbol to/from an access group;
- retrieving all groups accessible with the provided API key;
- querying access rights associated with a specific user or group.

To simplify testing, all ACS payloads are currently generated using hard-coded data, including a fixed variable symbol. In the production version, this variable symbol will be manually entered and stored in IS.BUK by the club's administrators, once an additional attribute for this purpose is added to the IS.BUK user model. This variable symbol will serve as a link to ISKAM [62] (the official dormitory information system), where it uniquely identifies each user.

The most critical integration logic is embedded in two asynchronous internal functions: `add_or_update_access_to_reservation_areas` and `delete_access_to_reservation_areas`. These functions are intended to be triggered upon reservation creation, modification, or deletion. They analyze which

access groups (linked to reservation services and mini-services) need to be updated for a specific user and construct the appropriate payloads. The corresponding ACS requests are then issued through the aforementioned helper.

While time-based access control is currently implemented at the date level (i.e., from a start date to an end date), the underlying API appears to support time precision. This opens the possibility for future discussions about introducing more granular restrictions, such as hourly access, though whether such features will be adopted ultimately depends on the direction taken by the dormitory's system.

At present, this entire integration is considered experimental. It is included in the system to verify future compatibility and to ensure the reservation system is prepared for seamless integration as soon as the production version of the API becomes available.

4.6.2 Authorization Endpoint for Club Access System

For access control managed by the club, the RS system provides a dedicated endpoint that receives authorization requests from IS.BUK whenever a user attempts to enter a room. IS.BUK sends a `POST` request containing the user's UID, room ID, and reader device ID. Upon receiving this data, the RS system performs a real-time check to determine whether the user has an active reservation for the specified room and time.

If a valid reservation is found, the system responds with `true`, granting access. Otherwise, access is denied. This logic is implemented in the business layer and allows the club to enforce reservation-based access restrictions without duplicating scheduling logic within IS.BUK.

> *Access control using the authorization system:*
>
> If the URL for the authorization system API and its fingerprint are provided, it will be used whenever the user is successfully authorized according to the local access settings of the information system. In order for the API to be called, the user must first be authenticated by the information system and granted access based on the local rules. Then, access is authorized by the remote system. The API is called via the `POST` method with parameters: the user's UID `[uid]`, the room ID `[room_id]`, and the device ID of the reader `[device_id]`. The remote system returns a boolean response, i.e., either `true` or `false`. If the remote system returns anything other than `true`, it is treated as `false`, and the first 256 characters of the response are logged.

(Excerpt from the IS.BUK Room Access [39])

4.7 Business Logic Layer (Services)

The `services` module in the backend architecture encapsulates the core business logic of the RS. It serves as an intermediary between the external `api` layer and the lower-level `crud` layer responsible for direct database interaction. This separation of concerns ensures that domain-specific rules, permission checks, and application-level validations remain decoupled from both data access and HTTP request handling logic.

Each service class is responsible for orchestrating operations related to a particular domain entity—such as users, calendars, reservation services and etc. These services consume data from the `crud` layer, perform additional logic (e.g., access control, collision detection, or soft deletion handling), and return results in a form suitable for the `api` layer.

Almost all service (except EmailService) classes inherit from a generic base class **CrudServiceBase**, which implements standard CRUD operations like `create`, `get`, `get_all`, `update`, `remove`, and `soft_remove`. Domain-specific services then override or extend these methods to enforce additional policies, permissions, or side effects.

Structure and Abstractions. Each service is typically structured as a pair:

- An **AbstractService** interface, which defines the expected operations and promotes consistency and testability.
- A concrete **Service** implementation that provides the actual logic and orchestrates calls to the `crud` layer.

The implementation is dependency-injected [61] with a scoped asynchronous SQLAlchemy session and internally creates an instance of the corresponding CRUD class for data manipulation. This pattern allows services to remain stateless and reusable across request lifecycles.

Domain-Specific Services: The following domain services are implemented within the system:

- **UserService** – manages user creation and synchronization from IS.BUK, including role-based logic and profile updates.
- **ReservationServiceService** – governs the creation, visibility, and lifecycle of reservation services, with permission checks based on user roles.
- **CalendarService** – handles calendar creation, integrates with GC API, and manages reservation-related metadata.
- **MiniServiceService** – manages subordinate services linked to calendars and reservation services.

```python
class AbstractCRUDService(ABC, Generic[Model, Crud, CreateSchema,
↪  UpdateSchema]):
  """
  Abstract base class for a CRUD service.
  Defines methods for retrieving, creating, updating, and deleting objects.
  """
  @abstractmethod
  async def get(self, uuid: UUID) -> Model | None:
      """Retrieve an object by its UUID."""

  @abstractmethod
  async def create(self, obj_in: CreateSchema) -> Model | None:
      """Create a new object."""



class CrudServiceBase(AbstractCRUDService[Model, Crud, CreateSchema,
↪  UpdateSchema]):
  def __init__(self, crud: Crud):
      self.crud: Crud = crud

  async def get(self, uuid: UUID) -> Model | None:
      return await self.crud.get(uuid)

  async def create(self, obj_in: CreateSchema) -> Model | None:
      return await self.crud.create(obj_in)
```

Code listing 4.9 Abstract and Base Classes for Asynchronous CRUD Services

- **EventService** – handles creation, validation, and lifecycle management of reservation events, including permission checks, rule enforcement, and state transitions.
- **EmailService** – generates PDF-based event registration forms and prepares email messages for dormitory management. This service operates independently of the database layer and does not require access to the application's session (i.e., it performs no data persistence operations).
- **AccessCardSystemService** – manages communication with external access control systems, including authorization logic and access group operations via ACS API.

Code Example: The code 4.9 illustrates the abstract base class for CRUD operations and demonstrates how service classes can implement business logic while delegating data access to the `crud` layer. The key part of this structure is the separation of concerns between the service (which contains domain logic) and the CRUD operations (which handle database interaction).

4.7.1 UserService

The **UserService** encapsulates the business logic related to the management of user accounts within the system. It extends the generic **CrudServiceBase** and implements additional methods tailored to user synchronization and role handling based on data received from an external identity system IS.BUK. Its responsibilities include creating or updating user records, evaluating role-based permissions, and detecting the user's status within the organization.

The service is structured as an interface **AbstractUserService** that defines the contract for core operations, such as `create_user` and `get_by_username`, and a concrete implementation **UserService** that carries out these tasks. This design allows for better testability and abstraction while ensuring adherence to the domain contract.

Business Logic. The `create_user` method is the core of the service, designed to either update an existing user or create a new one based on synchronization data from IS.BUK. The following business rules are applied:

- User roles are extracted from the `Role` list, and only valid aliases corresponding to existing reservation services are preserved.
- The user is marked as an `active_member` if they are linked to the "active" service or explicitly designated as a section head.
- If a user with the given username already exists, their profile is updated. Otherwise, a new user is created using the `UserCreate` schema.

These conditions ensure that the user's membership and permissions are consistent with their responsibilities in the BUK.

Design Characteristics. The **UserService** depends on both the `CRUDUser` and `CRUDReservationService` classes. This cross-domain dependency is necessary to verify that user-assigned roles reference valid reservation services. Furthermore, the service is instantiated via dependency-injection with a scoped asynchronous session, ensuring compatibility with FastAPI's request lifecycle and non-blocking database access.

4.7.2 ReservationServiceService, CalendarService, and MiniServiceService

The `ReservationServiceService`, `CalendarService`, and `MiniServiceService` all extend a shared CRUD service base, inheriting the same core access control and lifecycle management logic. These services enforce key business rules for each entity while utilizing common methods for database interactions.

```
async def create_user(
        self, user_data: UserIS,
        roles: list[Role],
        services: list[ServiceValidity],
        room: Room
) -> UserModel:
    user = await self.get_by_username(user_data.username)

    user_roles = []

    for role in roles:
        if role.role == "service_admin":
            for manager in role.limit_objects:
                if manager.alias in await
                ↪  self.reservation_service_crud.get_all_aliases():
                    user_roles.append(manager.alias)

    active_member = False
    for service in services:
        if service.service.alias == "active":
            active_member = True

    section_head = False
    if user_data.note.strip() == "head":
        active_member = True
        section_head = True

    if user:
        user_update = UserUpdate(
            room_number=room.door_number,
            active_member=active_member,
            section_head=section_head,
            roles=user_roles,
        )
        return await self.update(user.id, user_update)

    user_create = UserCreate(
        id=user_data.id,
        username=user_data.username,
        full_name=f"{user_data.first_name} {user_data.surname}",
        room_number=room.door_number,
        active_member=active_member,
        section_head=section_head,
        roles=user_roles,
    )
    return await self.crud.create(user_create)
```

Code listing 4.10 Role and Status Evaluation Logic in UserService

```
if not user.section_head:
    raise PermissionDeniedException("You must be the head of PS to delete
    ↪  services.")

if await self.crud.get_by_alias(service.alias, include_removed=True):
    raise BaseAppException("Service with this alias already exists.")
```

Code listing 4.11 Common logic for access control and uniqueness check.

Common Business Logic. Each service applies role-based access control, ensuring that only users with the correct permissions can create, update, or delete entities. Typically, only `section_head` users (or administrators of the relevant reservation service) are allowed to perform these operations. Additionally, all services support both soft deletion (where entities are marked as deleted but can be restored) and hard deletion (where entities are permanently removed). Services also validate uniqueness for critical fields, such as names and aliases, ensuring that no duplicate entities are created, even among soft-deleted records. These common logic pieces are shared by all three services, ensuring consistent data integrity and user access control.

Differences and Specific Purposes:

- **ReservationServiceService:** This service is responsible for managing reservation service entities. It enforces the creation and deletion of services, ensuring uniqueness by checking aliases and service names, even across soft-deleted entries. The service also allows access to public services, filtering out private ones.
- **CalendarService:** This service focuses on managing calendars linked to specific reservation services. In addition to role-based access control, it ensures that each calendar is associated with a valid reservation service. It also handles scoped access, allowing users to view only those calendars associated with services where they have a role.
- **MiniServiceService:** Similar to `CalendarService`, the `MiniService-Service` operates on auxiliary services related to reservation services. It also checks permissions and supports both deletion modes. However, a key difference is that `MiniServiceService` proactively prevents duplication of mini-services by returning the existing record if a mini-service with the same name already exists under the same reservation service.

4.7.3 EventService

The **EventService** is responsible for constructing and preparing event data that is to be posted to GC, ensuring that all reservation conditions and business rules are satisfied before doing so.

Internally, the service coordinates validation and event preparation using a layered approach:

- It performs a sequence of permission and rule validations:
  1. Verifies that the user is eligible to make the reservation via the `first_standard_check` utility.
  2. Ensures the guest count does not exceed calendar-defined limits, unless the calendar allows more people with specific permissions (via the `more_than_max_people_with_permission` flag in Calendar attribute).
  3. Retrieves user-specific rules based on their role.
  4. Confirms the reservation does not exceed the maximum allowed duration.
  5. Validates whether the reservation is made in an allowed time window — both sufficiently in advance and not too early (defined by advance and prior rules).
- If all checks are passed, the service constructs a valid event body using the `ready_event` utility.

This decoupled logic allows the actual posting to GC to remain simple and focused on external integration, while the complex internal validation is handled in a modular and reusable way.

Concrete Implementation: Implementation coordinates with multiple utility functions to validate input and construct the final event object. The core method is `post_event` 4.12, which performs validation via the private method `__control_conditions_and_permissions` 4.13, and then constructs the event body if access is granted.

Rule Enforcement Logic: The validation checks leverage helper functions such as `first_standard_check`, `dif_days_res`, and `reservation_in_advance`. These encapsulate checks like service membership, date/time consistency, maximum reservation length, and time window policies. By abstracting these into dedicated helpers, the system ensures reusability and simplifies the core service logic.

Additional Functionality: In addition to validation and event construction, the service provides a full set of operations over reservations:

- `create_event`: stores a new event in the database after receiving confirmation from GC.
- `get_by_user_id`: retrieves all reservations for a given user, enriched with extra context.

```
async def post_event(
        self, event_input: EventCreate,
        services: list[ServiceValidity], user: User,
        calendar: Calendar
) -> Any:
    if not calendar:
        return {"message": "Calendar with that type not exist!"}

    message = await self.__control_conditions_and_permissions(
        user, services, event_input, calendar)

    if message != "Access":
        return message

    return ready_event(calendar, event_input, user)
```

Code listing 4.12 Core Logic of EventService

- `get_by_event_state_by_reservation_service_alias`: retrieves events of a specific state (e.g. NOT_APPROVED, CONFIRMED) within a reservation service.
- `get_reservation_service_of_this_event`, `get_calendar_of_this_event`, `get_user_of_this_event`: helper methods to fetch related entities by navigating associations.
- `get_current_event_for_user`: determines if the user currently holds an active reservation (based on current time).
- `approve_update_reservation_time` and `request_update_reservation_time`: provide a flow for changing reservation times, including user request and admin approval.
- `cancel_event`: marks an event as cancelled in the system.
- `confirm_event`: confirms a previously pending reservation.

In summary: The **EventService** plays a critical role in safeguarding the integrity of reservations by applying all business rules before event construction. This architectural choice ensures clean separation of concerns between validation, business logic, and integration layers. Additionally, the rich set of provided methods ensures complete life-cycle management of event-based reservations in the system.

```
    async def __control_conditions_and_permissions(
            self, user: User,
            is_info: InformationFromIS,
            event_input: EventCreate,
            calendar: CalendarModel
    ) -> str | dict:
        reservation_service = await self.reservation_service_crud.get(
            calendar.reservation_service_id)

        standard_message = first_standard_check(is_info, reservation_service,
                                                event_input.start_datetime,
                                                event_input.end_datetime)
        if not standard_message == "Access":
            return standard_message

        if not calendar.more_than_max_people_with_permission and \
                event_input.guests > calendar.max_people:
            return {"message": f"You can't reserve this type of "
                               f"reservation for more than
                               ↪  {calendar.max_people} people!"}

        user_rules = await self.__choose_user_rules(user, calendar)

        if not dif_days_res(event_input.start_datetime,
        ↪  event_input.end_datetime, user_rules):
            return {"message": "You can reserve on different day."}

        message = reservation_in_advance(event_input.start_datetime,
        ↪  user_rules)
        if not message == "Access":
            return message

        return "Access"
```

Code listing 4.13 Checking Conditions and Permissions for Event Creation

4.7.4 EmailService

The **EmailService** class defines a contract for services responsible for handling all email-related functionality within the system. This service encapsulates the logic necessary for preparing, formatting, and packaging emails for dispatch, acting as a central layer for email processing across various use cases.

For example method `prepare_registration_form` 4.14 is responsible for generating a structured PDF document based on a provided event registration form. This document is intended for distribution to stakeholders such as dormitory heads. The method proceeds as follows:

- It loads a predefined PDF template from the filesystem.
- The template is copied and modified in-memory using the `pypdf` library.

- Relevant fields such as event purpose, number of guests, organizer details, space to be reserved, and formatted start/end dates are injected into the document.
- The filled form is saved as a new PDF file.
- An `EmailCreate` object is constructed containing recipient email addresses (including both the requester and the responsible manager), a subject line, a message body, and the path to the generated PDF as an attachment.

This encapsulation allows for seamless integration of form generation and email preparation logic, ensuring a consistent and auditable process for formal communication.

4.7.5 AccessCardSystemService

The responsibility for enforcing reservation-based access rules is encapsulated within the `AccessCardSystemService`, which implements the abstract interface `AbstractAccessCardSystemService`. This service acts as the business logic layer for both verifying access requests and preparing structured data for communication with external access systems.

At the core of this service is the `reservation_access_authorize` method, which governs access validation for room entry attempts. When the system receives an access request (including UID, room ID, and reader device ID), it executes several domain-specific checks in sequence 4.15:

1. **User Verification:** The service attempts to load the user by UID from the database. If the user does not exist in the local system, access is immediately denied.
2. **Room Association:** The service checks whether the specified room is associated with a reservation service or a mini-service. If no such mapping exists, the room is considered outside the system's scope, and access is denied.
3. **Active Event Resolution:** The user's currently active reservation (i.e., event) is retrieved using the `EventService`. If no active reservation is found, access is denied.
4. **Permission Matching:** Access is granted only if one of the following conditions is satisfied:
   - The event includes a mini-service linked to a specific room (i.e., `room_id` is not `None`), and both the room and the `device_id` match.
   - The event includes a reservation-service linked to a specific room (i.e., `room_id` is not `None`), and both the room and the `device_id` match. Locker checks also consider mini-services without their own room that extend the reservation-service with additional `lockers_id`.

```
def prepare_registration_form(
          self, registration_form: RegistrationFormCreate,
          full_name: str
  ) -> EmailCreate:
      base_dir = os.path.dirname(os.path.abspath(__file__))
      original_pdf_path = os.path.join(base_dir, '..', 'templates',
      ↪  'event_registration.pdf')
      output_path = "/tmp/event_registration.pdf"

      shutil.copy(original_pdf_path, output_path)
      reader = PdfReader(output_path)
      writer = PdfWriter()
      writer.append(reader)

      formatted_start_date = registration_form.event_start.strftime("%H:%M,
      ↪  %d/%m/%Y")
      formatted_end_date = registration_form.event_end.strftime("%H:%M,
      ↪  %d/%m/%Y")

      writer.update_page_form_field_values(
          writer.pages[0],
          {
              "purpose": registration_form.event_name,
              "guests": str(registration_form.guests),
              "start_date": formatted_start_date,
              "end_date": formatted_end_date,
              "full_name": full_name,
              "email": str(registration_form.email),
              "organizers": registration_form.organizers,
              "space": registration_form.space,
              "other_spaces": ", ".join(registration_form.other_space or
              ↪  []),
              "today_date": datetime.today().strftime("%d/%m/%Y"),
          }
      )

      with open(output_path, "wb") as output_pdf:
          writer.write(output_pdf)

      email_create = EmailCreate(
          email=[registration_form.email,
          ↪  registration_form.manager_contact_mail],
          subject="Event Registration",
          body=(
              f"Request to reserve an event for a member {full_name}"
          ),
          attachment=output_path
      )

      return email_create
```

Code listing 4.14 Implementation of PDF-Based Event Registration Preparation in EmailService

This step-by-step resolution ensures that both direct room reservations and indirect access rights (via mini-services) are properly enforced according to the system's domain rules.

In addition to access authorization, the `AccessCardSystemService` also provides methods that construct structured payloads for the dormitory's ACS integration. These include:

- Adding or updating a user (identified by variable symbol) in an access group;
- Removing a user from a group;
- Retrieving available access groups for the provided API key;
- Querying access groups for a specific user or for a specific group name.

Each method returns a dictionary formatted according to the expected structure of the external ACS API, but does not directly perform any HTTP communication. This separation of concerns ensures testability and avoids tight coupling between business logic and external I/O operations.

Overall, the service layer defines and enforces access control policies based on reservation data, user identity, and configured services. It provides a consistent and extensible foundation for controlling access behavior at physical entry points while maintaining a clean interface for integration with external systems.

4.8 CRUD Module

The `crud` module in the original version of the application was responsible for encapsulating the logic related to data persistence and retrieval using SQLAlchemy ORM. It consisted of an abstract, reusable base class for standard operations and concrete implementations tailored to specific domain models. This module aimed to separate concerns by isolating direct database interactions from higher-level business logic, promoting reusability and testability.

At the core of the module was the `AbstractCRUDBase`, a generic and extensible interface defining common operations such as `get`, `get_all`, `create`, `update`, `remove`, and `soft_remove`. These methods provided structured access to records while supporting features such as soft deletion and filtering by domain-specific attributes like reservation service identifiers.

The `CRUDBase` class implemented this abstract interface and performed actual database queries using SQLAlchemy's query interface. Notable capabilities included:

- **Soft Deletion**: Rather than permanently removing a record, the `soft_remove` method updated a `deleted_at` timestamp. This allowed for data recovery or archival visibility via the `include_removed` flag.

```
async def reservation_access_authorize(
            self,
            service_event: Annotated[EventService, Depends(EventService)],
            access_request: ClubAccessSystemRequest
    ) -> bool:
        user = await self.user_crud.get(access_request.uid)

        if user is None:
            raise PermissionDeniedException("This user isn't exist in
            ↪  system.")

        reservation_service = await
        ↪  self.reservation_service_crud.get_by_room_id(
            access_request.room_id)
        mini_service = await
        ↪  self.mini_service_crud.get_by_room_id(access_request.room_id)

        if (reservation_service is None) and (mini_service is None):
            raise PermissionDeniedException("This room associated with some
            ↪  service isn't exist "
                                            "in system or use another access
                                            ↪  system")

        event = await service_event.get_current_event_for_user(user.id)

        if event is None:
            raise PermissionDeniedException("No available reservation exists
            ↪  at this time.")

        if mini_service and (mini_service.name in event.additional_services):
            if access_request.device_id in mini_service.lockers_id:
                return True

        if reservation_service == await
        ↪  service_event.get_reservation_service_of_this_event(event):
            if access_request.device_id in reservation_service.lockers_id:
                return True

            for mini_service_name in event.additional_services:
                mini_service = await
                ↪  self.mini_service_crud.get_by_name(mini_service_name)
                if (mini_service.room_id is None) and
                (access_request.device_id in
                 mini_service.lockers_id):
                    return True

        raise PermissionDeniedException("No matching reservation exists at
        ↪  this time "
                                        "for this rules.")
```

Code listing 4.15 Reservation Access Authorize

- **Schema-Agnostic Operations**: Through the use of Python generics and `TypeVar`, the base class was reusable across different SQLAlchemy models and Pydantic schemas.
- **Model Recovery**: The `retrieve_removed_object` method enabled restoring previously soft-deleted entries, offering additional robustness in data lifecycle management.

Model-Specific Extensions. Beyond the general-purpose CRUD functionality, the module also included specialized implementations for specific domain models. For example, the `CRUDReservationService` class extended `AbstractCRUDReservationService`, which itself inherited from `CRUDBase`, and introduced additional query capabilities relevant to this model.

This specialization pattern allowed each CRUD class to reflect domain-specific requirements while maintaining the benefits of abstraction and consistency. It also ensured that database logic remained close to the data structure definitions, simplifying future schema migrations or enhancements.

Design Implications and Future Transition. This earlier design coupled the SQLAlchemy session directly with the CRUD classes, which while practical, introduced constraints for asynchronous execution contexts and limited flexibility in managing transactional boundaries across services. As the application evolved, the CRUD layer was refactored to better align with asynchronous patterns and domain-driven design principles.

4.9 Transition to Asynchronous Architecture

This section discusses the transition from a synchronous to an asynchronous architecture in the backend application. The motivation for this change stems from the need to improve scalability, performance, and maintainability, as well as to leverage the asynchronous capabilities of FastAPI and SQLAlchemy. This shift was prompted by challenges in handling database schema migrations and the desire to modernize the codebase to align with best practices in web development. The section details the changes in database session management, model definitions, CRUD operations, and the service layer, as well as the integration of Alembic for managing database migrations asynchronously.

4.9.1 Motivation for the Transition

The transition to an asynchronous architecture was initially motivated by the desire to introduce proper database schema migrations using Alembic. In the earlier development stages, the application relied on SQLAlchemy's `create_all()` mechanism to generate tables, which proved insufficient as the project grew. Modifying models—such as adding new fields or introducing

related entities—required dropping and recreating the entire schema, making iteration cumbersome and error-prone.

Upon researching Alembic integration, many examples and best practices emphasized the use of asynchronous patterns, especially in FastAPI applications. While synchronous migrations were also possible, the opportunity to align the backend with FastAPI's native asynchronous capabilities became a compelling direction. Furthermore, earlier sections of this work had already emphasized FastAPI's advantages in speed and concurrency. Thus, optimizing the application by fully embracing asynchronous execution promised improved performance, resource utilization, and alignment with the framework's strengths.

The architectural shift was also driven by a desire to modernize and future-proof the codebase—ensuring better scalability, smoother developer experience, and consistency with FastAPI ecosystems.

4.9.2 Implementation Changes

The shift to asynchronous execution affected multiple layers of the application. This subsection outlines the most significant changes introduced during this transition.

Session Management: The original synchronous session management was based on the `SessionLocal` factory and yielded a `Session` instance via a synchronous dependency function 4.16.

This was replaced by a fully asynchronous session manager encapsulated in a `DatabaseSession` class, using `create_async_engine`, `async_sessionmaker`, and `async_scoped_session` 4.16.

This change ensures compatibility with FastAPI's asynchronous request handlers and allows for concurrent, scoped usage of database sessions.

Model Definitions: SQLAlchemy ORM models were migrated from the legacy style to SQLAlchemy 2.0-style type annotations using `Mapped[]` and `mapped_column()`. Relationships were also annotated accordingly, and loading strategies such as `lazy="selectin"` were introduced to improve query efficiency 4.17.

Declarative Base Refactor: The legacy base class using `@as_declarative` was replaced with SQLAlchemy 2.0's new `DeclarativeBase` system. The new base explicitly defines a UUID primary key and retains the auto-generated table name logic.

CRUD Layer Refactor: All methods within the `crud` layer were rewritten to support `AsyncSession` and fully utilize `await` syntax. This includes

```
# Before
engine = create_engine(DATABASE_URI, pool_pre_ping=True)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

def get_db() -> Generator[Session, None, None]:
    with SessionLocal() as db:
        yield db

# ------------------------------------------------------------

# After
self.engine = create_async_engine(url=DATABASE_URI, pool_pre_ping=True)
self.session_factory = async_sessionmaker(
    bind=self.engine, autoflush=False, autocommit=False,
    ↪  expire_on_commit=False
)

async def session_dependency(self) -> AsyncGenerator[AsyncSession, None]:
    async with self.session_factory() as session:
        yield session
```

Code listing 4.16 Synchronous and Asynchronous Session Setup

```
# Before
id = Column(UUID(as_uuid=True), primary_key=True, default=uuid4)
name = Column(String, unique=True, nullable=False)
alias = Column(String, unique=True, nullable=False)
public = Column(Boolean, nullable=False, default=True)
web = Column(String, nullable=True)
contact_mail = Column(String, nullable=True)

calendars = relationship("Calendar", back_populates="reservation_service")
mini_services = relationship("MiniService",
↪  back_populates="reservation_service")


# ------------------------------------------------------------

# After
name: Mapped[str] = mapped_column(unique=True, nullable=False)
alias: Mapped[str] = mapped_column(unique=True, nullable=False)
public: Mapped[bool] = mapped_column(nullable=False, default=True)
web: Mapped[str] = mapped_column(nullable=True)
contact_mail: Mapped[str] = mapped_column(nullable=True)

calendars: Mapped[list["Calendar"]] = relationship(
    back_populates="reservation_service", lazy="selectin")
mini_services: Mapped[list["MiniService"]] = relationship(
    back_populates="reservation_service", lazy="selectin")
```

Code listing 4.17 Legacy Model and Modern SQLAlchemy 2.0 Style

```python
# Before
    def __init__(self, model: Type[Model], db: Session):
        self.model: Type[Model] = model
        self.db: Session = db

    def get(self, uuid: UUID | str | int,
            include_removed: bool = False) -> Model | None:
        if uuid is None:
            return None
        return self.db.query(self.model) \
            .execution_options(include_deleted=include_removed) \
            .filter(self.model.id == uuid).first()

# ------------------------------------------------------------

# After
    def __init__(self, model: Type[Model], db: AsyncSession):
        self.model: Type[Model] = model
        self.db: AsyncSession = db

    async def get(self, uuid: UUID | str | int,
                  include_removed: bool = False) -> Model | None:
        if uuid is None:
            return None
        stmt = select(self.model).filter(self.model.id == uuid)
        if include_removed:
            stmt = stmt.execution_options(include_deleted=True)
        result = await self.db.execute(stmt)
        return result.scalar_one_or_none()
```

Code listing 4.18 Comparison of Synchronous and Asynchronous Implementations of the CRUD Layer

database queries, insertions, and transaction management. Sessions are now injected externally rather than being managed within the `crud` classes, enabling more flexible composition with service logic and improving testability 4.18.

Service Layer Refactor: All service methods were made asynchronous to support seamless interaction with the refactored `crud` layer. Function calls between services and `crud` modules were updated to use `await`, and blocking code was removed or adapted. This change enabled consistent behavior across services and improved compatibility with FastAPI's async request handling.

Alembic Migrations Integration: The migration from `create_all()` to Alembic-based migrations marked a fundamental improvement in managing schema evolution. With Alembic, changes to models are reflected in migration scripts, allowing for safer and more granular schema updates in production

environments.

Asynchronous I/O: All previously synchronous I/O operations were updated to use asynchronous equivalents. This includes not only database operations but also any I/O-bound interactions in services, making the entire application architecture consistent with async principles.

Summary: These changes together represent a comprehensive transition toward a modern, scalable, and asynchronous FastAPI backend architecture. This refactoring improves performance, testability, and maintainability, aligning the application with industry best practices.

4.10 Development Process

This section outlines the practical aspects of how the application was developed, tested, and deployed. It describes the tools and environments used throughout the implementation process, including IDE, testing workflows, and containerized infrastructure. By highlighting the development workflow and deployment practices, this section emphasizes the importance of reproducibility, maintainability, and developer efficiency in modern backend systems.

Development Environment: The application was developed using **PyCharm**, a widely adopted Python IDE by JetBrains [41]. The development setup included integration with `conda` environments and support for debugging asynchronous applications. PyCharm was configured to directly connect to the PostgreSQL database container, allowing convenient inspection and management of data through a graphical interface. This setup streamlined database introspection, schema updates, and manual cleanups during development.

Testing and Debugging: During implementation, the application was primarily run and debugged directly from the IDE, which provided a convenient interface for executing test cases and tracing asynchronous flows. The PostgreSQL database was not installed natively but instead operated in an isolated environment via Docker, improving portability and ensuring environmental consistency. This approach facilitated seamless transitions between development, testing, and deployment setups.

4.10.1 Containerized Infrastructure

A container-based setup was adopted using **Docker** [45] and **Docker Compose** [63]. The database and backend service were both encapsulated in containers, defined declaratively via configuration files. This ensured reproducibility and reduced overhead when switching machines or onboarding new environments.

4.10.1.1 Dockerfile for the Backend

The `Dockerfile` for the backend uses a two-stage build process. In the build stage, a `conda` environment is created based on the `environment-dev.yml` specification, and then packed into a virtual environment to reduce image size. The runtime stage is based on a minimal `debian` image, which only contains the application code and the prepacked environment:

- **Build Stage**: Installs dependencies with `conda`, packages the environment, and removes unnecessary files.
- **Runtime Stage**: Extracts the environment archive, activates it, and runs the FastAPI application.

4.10.1.2 Docker Compose for Multi-Container Setup

The `docker-compose.yml` defines two services: the backend and the PostgreSQL database.

- The `db` service uses the official `postgres:13.4` image, exposes port `5432`, and is configured with default credentials.
- The `backend` service depends on the database, uses the prebuilt backend image, and mounts the project directory to enable live development.

This configuration enables simultaneous orchestration of the backend and database services. The isolation between containers enforces separation of concerns while facilitating local development and integration testing in an environment closely resembling production.

4.10.2 Static Analysis and Code Style Enforcement

Mypy Type Checking: To ensure type safety and catch potential runtime errors early, the application codebase was regularly validated using **Mypy**, a static type checker for Python [51]. A custom Bash script was used to simplify execution and support optional report generation. The script accepts parameters to generate HTML and XML reports, making it suitable for both local inspection and integration with reporting tools. Example usage includes:

- `.scripts/run-mypy.sh` — run Mypy with default configuration.
- `.scripts/run-mypy.sh -h` — generate an HTML type-checking report.

This automation streamlined type validation and encouraged consistent use of type annotations across the codebase.

Pylint for Code Quality: **Pylint** was employed to enforce coding standards and identify stylistic issues [50]. A wrapper script was implemented to recursively check both source and test directories, with support for optional scoring thresholds via the `--fail-under` flag. This made it possible to enforce a minimum acceptable score during checks, which helps maintain long-term code quality. Example usage includes:

- `./run-pylint.sh` — run Pylint on all source and test files.
- `./run-pylint.sh -f 8.0` — fail if score falls below 8.0.

By integrating type checks and linter scripts into the daily workflow, the project maintained a high standard of readability and robustness during development.

4.11 Deployment Architecture and Server Configuration

This section describes the deployment setup and server-side infrastructure of the application. It outlines the hosting environment, the usage of containerization via **Docker**, orchestration through **Docker Compose**, and the configuration of a central **Nginx** reverse proxy. The goal is to provide a detailed view of how the backend and frontend services are deployed and integrated in both development and production environments, ensuring scalability, maintainability, and secure access.

4.11.1 Overview of Hosting Environment

The final application was deployed on club-owned infrastructure, specifically on a physical server maintained by the BUK. Two **Virtual Machines** [64] were provisioned: one dedicated to the production environment and the other for development purposes. This separation facilitated safe testing and rollout of new features while maintaining a stable production system.

The deployed system comprises three Git repositories:

- **Backend:** Implements the FastAPI-based server logic.
- **Frontend:** Contains a React-based client interface and its own Nginx configuration.

- **Documentation:** Serves as an orchestration repository combining all components via Docker Compose, containing the main Nginx reverse proxy configuration and certificates.

While only the backend was developed in this thesis, full-stack deployment was carried out to ensure a fully functional application for BUK use.

4.11.2 Frontend Deployment and Nginx Configuration

The frontend application is deployed using a two-stage Docker build. In the first stage, a Node.js [65] image compiles the React application into static assets. In the second stage, these static files are served by an Nginx container.

Nginx is a high-performance web server and reverse proxy server that is widely used to serve static content, handle HTTP routing, and manage load balancing. It is particularly suitable for deploying single-page applications such as those built with React, as it efficiently handles file serving and supports client-side routing by falling back to the main entry point (`index.html`) when a requested path does not correspond to a physical file [66].

4.11.3 System Orchestration Using Docker Compose

The orchestration of the backend, frontend, database, and reverse proxy was managed via Docker Compose, defined in the `docker-compose.yml` file 4.19 of the documentation repository

This configuration ensures modularity, with each service isolated in its own container and connected via a shared internal Docker network. The system can be launched and rebuilt consistently across different environments.

4.11.4 Central Reverse Proxy and HTTPS Termination

The Nginx container in the `documentation` repository acts as a central reverse proxy for routing requests to the appropriate services and handling HTTPS termination via Let's Encrypt [67]. A sample configuration is shown in Code listing 4.20:

This setup handles both HTTP redirection to HTTPS and domain-based routing. The frontend is served via `reservation.buk.cvut.cz`, and the backend API is accessible at `api.reservation.buk.cvut.cz`. Certificates are issued and auto-renewed via Certbot, with support for ACME challenges via the `.well-known/acme-challenge` location block.

```yaml
version: '3.0'
services:
  nginx:
    image: nginx:stable-alpine
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - './nginx.conf:/etc/nginx/nginx.conf'
      - '/etc/letsencrypt:/etc/letsencrypt'
    depends_on:
      - backend
      - frontend
    networks:
      - internal

  db:
    image: postgres:13.4
    environment:
      - POSTGRES_USER=dev
      - POSTGRES_PASSWORD=pass
      - POSTGRES_DB=devdb
    ports:
      - "5432:5432"
    networks:
      - internal

  backend:
    image: darkrader/buk-reservation-backend-amd64:latest
    environment:
      - POSTGRES_SERVER=db
    volumes:
      - ../backend/:/app
    ports:
      - "8000:8000"
    depends_on:
      - db
    networks:
      - internal

  frontend:
    build:
      context: ../frontend
    ports:
      - "3000:3000"
    networks:
      - internal

networks:
  internal:
```

Code listing 4.19 Excerpt from docker-compose.yml in Documentation Repository

```
user  root;
worker_processes  1;

events { }

http {
    server {
        listen 80;
        server_name reservation.buk.cvut.cz api.reservation.buk.cvut.cz;
        return 301 https://$host$request_uri;
    }

    server {
        listen 443 ssl;
        server_name reservation.buk.cvut.cz;

        location / {
            proxy_pass http://frontend:3000/;
        }

        ssl_certificate
        ↪  /etc/letsencrypt/live/reservation.buk.cvut.cz/fullchain.pem;
        ssl_certificate_key
        ↪  /etc/letsencrypt/live/reservation.buk.cvut.cz/privkey.pem;

        location /.well-known/acme-challenge/ {
            root /var/www/certbot;
        }
    }

    server {
        listen 443 ssl;
        server_name api.reservation.buk.cvut.cz;

        location / {
            proxy_pass http://backend:8000/;
        }

        ssl_certificate
        ↪  /etc/letsencrypt/live/reservation.buk.cvut.cz/fullchain.pem;
        ssl_certificate_key
        ↪  /etc/letsencrypt/live/reservation.buk.cvut.cz/privkey.pem;

        location /.well-known/acme-challenge/ {
            root /var/www/certbot;
        }
    }
}
```

Code listing 4.20 Central Nginx Configuration

```
#!/usr/bin/env bash

set -e

src_dir="../app"

# Parse command line options
while getopts "crf:" opt; do
  case ${opt} in
  r) GENERATE_COVERAGE_REPORT=1 ;;
  c) RUN_COVERAGE=1 ;;
  f) USE_FAIL_UNDER=1; COVERAGE_LIMIT_PERCENTAGE="$OPTARG" ;;
  *) exit 0 ;;
  esac
done

cd tests
# Select the correct pytest command
if [ $GENERATE_COVERAGE_REPORT -eq 1 ]; then
  pytest --cov=$src_dir --cov-report=term --cov-branch \
         --cov-report=html:coverage --cov-report=xml:coverage.xml \
         ${USE_FAIL_UNDER:+--cov-fail-under="$COVERAGE_LIMIT_PERCENTAGE"}
elif [ $RUN_COVERAGE -eq 1 ]; then
  pytest --cov=$src_dir --cov-report=term --cov-branch \
         ${USE_FAIL_UNDER:+--cov-fail-under="$COVERAGE_LIMIT_PERCENTAGE"}
else
  pytest
fi
```

Code listing 4.21 Pytest Runner Script

4.11.5 Conclusion

This deployment approach ensures a robust, reproducible, and scalable environment suitable for both development and production usage. The separation of concerns via Docker containers and centralized configuration management via Nginx allows for maintainability and future extensibility.

4.12 Testing Approach

To ensure correctness and robustness of the backend system, an extensive test suite was developed using `pytest` [68] — a mature testing framework widely adopted in the Python ecosystem. The entire test suite was executed using a Bash script, which provided support for optional coverage analysis and threshold enforcement. This script accepted flags for basic test execution, coverage tracking, and generating HTML/XML coverage reports using the `pytest-cov` plugin 4.21

```
@pytest_asyncio.fixture
async def test_user(user_crud):
    return await user_crud.create(UserCreate(
        id=2142,
        username="fixture_user",
        active_member=True,
        section_head=False,
        roles=["club", "grill"]
    ))
```

Code listing 4.22 CRUD Fixture Example for Test User

4.12.1 Database Configuration for Tests

The backend application relied on an asynchronous PostgreSQL database accessed via SQLAlchemy. To provide isolated and reproducible environments for database tests, the `testcontainers` library was used to spin up a lightweight PostgreSQL container at the start of each test session.

To manage the asynchronous engine and session, a dedicated utility class `TestDatabaseSession` was implemented. This class established a connection using `asyncpg` and provided mechanisms to create, drop, and reset the database schema before and after each test.

The session-scoped fixture `pg_container` ensured the lifecycle of the test container, while the `async_session` and `shared_session` fixtures provided isolated sessions and schema management, depending on whether schema recreation was needed per test or once per session.

4.12.2 Test Structure and Fixtures

The tests were organized in modular fashion, with dedicated `conftest.py` files per layer (e.g., `models`, `schemas`, `crud`, `services`, and `api`). These `conftest.py` files defined fixtures that created valid entities or schema objects used across different tests.

Model-level fixtures defined reusable instances of ORM entities such as `UserModel`, `ReservationServiceModel`, and `CalendarModel`, initialized and committed within an asynchronous test session. These were useful for both model validation and integration-level tests.

In the `crud` testing layer, fixtures leveraged the respective CRUD service implementations (e.g., `CRUDUser`, `CRUDCalendar`) to create and return prepopulated entities from actual database operations 4.22.

4.12.3 Test Execution Strategy

Tests were grouped by domain entity and application layer, each with its own set of fixtures and assertions. By maintaining separation of concerns, each

test focused on verifying the functionality and correctness of a specific layer in isolation.

This structure, combined with dynamic PostgreSQL containerization and transactional session management, ensured repeatability, test isolation, and reliability across the test suite.

4.12.4 Test Coverage and Limitations

As of the final development stage, the automated test suite comprises over 200 passing tests with no failures, providing broad validation for core backend logic, database interactions, and business rules:

200 passed, 28 warnings in 23.23s

The test suite achieved a total code coverage of **67%**, as measured using the `pytest-cov` plugin [69]. Table 4.1 presents a summarized view of code coverage across different components.

Component	Stat. Cov.	Branches Cov.	Cov. (%)
Schemas	100%	100%	100%
CRUD Layer	88–100%	81–100%	95–100%
Services Layer	64–100%	76–100%	64–100%
Models	73–100%	87–100%	73–100%
Core Utilities	96–100%	88–100%	88–100%
API Layer	16–100%	21–100%	**30% (average)**

Table 4.1 Code Coverage Summary by Component

The lower coverage in the API layer is primarily due to challenges encountered in configuring a dedicated testing middleware for session injection and request handling in the test environment. Despite several attempts, proper test-time session propagation via middleware could not be achieved consistently, which led to incomplete endpoint-level testing.

Additionally, some of the final features, such as event management and the ACS, were implemented late in the development cycle. Due to time constraints, comprehensive test coverage for these modules could not be completed.

4.12.5 Manual Testing and Verification

In addition to the automated test suite, the application was continuously tested manually throughout the development process. Swagger-generated OpenAPI documentation was actively used to interact with RESTful endpoints and verify correctness during incremental implementation phases. This facilitated early identification of integration bugs and allowed for fast feedback cycles.

Moreover, during frontend integration, coordinated manual testing sessions were conducted with the frontend developer to ensure full-stack consistency and correct client-server interaction. These sessions proved particularly valuable in verifying complex workflows such as user authentication, calendar interactions, and event reservation handling. Despite the lack of full API test automation, this hands-on testing approach ensured the practical reliability of all major features within the live environment.

Chapter 5
Evaluation

This chapter evaluates the developed RS in terms of its functional completeness, technical implementation, and future maintainability. The evaluation is based on the requirements defined in the analysis phase, the results of testing, and the anticipated needs of its target users. It also outlines areas for potential improvement and presents directions for future development aimed at broadening the system's applicability and robustness.

5.1 Results

This chapter presents the results achieved during the development of the RS for BUK. The system's primary goal was to automate the room booking process for club members, replacing the existing manual and email-based workflow with a streamlined web-based solution. The implemented backend, built using FastAPI and asynchronous SQLAlchemy sessions, allows members to create reservations, while managers can dynamically configure rooms, rules, and services through a flexible permission-based interface.

As a result of this work, all core functional requirements defined in the analysis phase were successfully implemented. The system supports real-time reservation management, integration with Google Calendar, and role-based access control. Managers are now able to define services and tailor reservation rules per space without needing to involve developers or perform manual data entry.

Over the course of approximately six months, the first version of the system was successfully deployed and tested in a real-world setting. Since early October, club members have been actively using the application to reserve rooms through the web interface. Although this initial version included only basic functionality—such as the ability to book a specific room, select a reservation type, and assign optional mini services—it already demonstrated significant practical value. The application substantially accelerated the reservation pro-

cess for both members and managers, while dramatically reducing the volume of manual communication previously conducted via email.

Although the majority of the critical features were implemented, one major aspect—automated access control via RFID cards—was only partially realized. Integration with dormitory readers is still in development on their side, and while a prototype using a test API was implemented to demonstrate feasibility, full production integration remains pending. Similarly, access synchronization for club-managed hardware was illustrated via example, but not deployed due to infrastructure limitations. Additionally, role management within IS.BUK proved restrictive, limiting the system's flexibility in defining custom permissions.

The system's backend is structured with maintainability and scalability in mind. It supports container-based deployment and includes authentication via Google OAuth2. The codebase is covered by a suite of automated tests, achieving 67% test coverage across models, schemas, crud, services and api modules.

Overall, the resulting system automates the reservation process, improves efficiency, and significantly reduces the workload for room managers. It lays a solid foundation for future extensions, such as full ACS integration and richer role synchronization with IS.BUK.

5.2 Future development

Although the RS has achieved its primary goals for the BUK, its development is expected to continue with several planned improvements and extensions. From the outset, the intention has been not only to serve the needs of a single club, but also to provide a reusable and adaptable solution for other student dormitory clubs within the SU CTU. This broader vision motivates the following directions for future development:

1. **Full integration with ACS.** The system aims to support synchronization with both the dormitory's ACS and custom club-managed ACS setups. The goal is to make access automation more adaptable and allow clubs without IS.BUK information system to still benefit from seamless ACS integration.

2. **Improved role management.** The current system is limited by the constraints of IS.BUK, which restricts dynamic role creation and customization. A more flexible and independent role management module is planned, enabling the creation of additional permission levels specific to RS workflows and potential future features.

3. **Support for LDAP-based authentication.** As many clubs operate their own information systems but have access to centralized LDAP services, introducing LDAP-based authentication will enhance the system's compatibility and usability across multiple clubs. It remains to be evaluated whether LDAP will act as an alternative or a complementary method to IS.BUK authentication.

4. **Expansion of reservation use cases.** Future iterations will introduce new types of reservable services beyond room booking. For example, club-organized events such as sports training sessions could be managed via the RS, allowing members to join based on limited capacity constraints.

5. **Integration of club events and campaigns.** Currently, events are added manually to a GC. The planned functionality includes in-system event creation by section leads, with automated formatting and synchronization to the SU CTU's GC. This would streamline the event management process and ensure consistent visibility across platforms.

6. **Implementation of equipment reservations.** Inspired by solutions used in other clubs, the RS will be extended to support reservations of club-owned equipment. Integration with ACS-controlled lockers is also envisioned, allowing members to retrieve reserved items via RFID cards, with prior preparation by managers.

7. **Backend improvements.** Although the current implementation adheres to solid engineering practices, several areas require further refinement. Planned enhancements include improving code modularity, reducing repetition, expanding test coverage, refining error handling, and optimizing the codebase to minimize effort required for introducing new features. These improvements will make the system more maintainable and scalable in the long term.

It is important to note that the backend system was developed entirely by a single individual as part of this bachelor thesis. Consequently, development speed was limited by the scope and time constraints of the project. However, once the thesis is complete and the application is made open to contributions from other interested developers, the implementation of these planned features is expected to accelerate. The overarching goal remains to make the reservation process as seamless and efficient as possible for all BUK's members.

Conclusion

The main objective of this thesis was to design and implement a web-based room RS for BUK that automates the booking process, reduces the workload of room managers, and ensures integration with existing club infrastructure such as IS.BUK and ACS.

The work began with an analysis of the current manual reservation process and its limitations. Based on the identified needs and functional requirements, a modular backend architecture was developed using FastAPI, asynchronous SQLAlchemy, and PostgreSQL. The system supports role-based access control, real-time reservation management, and GC integration. It is containerized for deployment via Docker and covered by automated tests to ensure maintainability and reliability.

During development, a flexible domain model was introduced, allowing managers to dynamically define services and configure rules without developer intervention.

While the core functionality was successfully completed, some parts—particularly the full integration with RFID-based ACS—remain under development due to external constraints. Nonetheless, a working prototype was implemented to demonstrate feasibility, and future integration points have been clearly defined.

The results confirm that the system meets its primary objectives. It replaces the previously manual workflow with a flexible, automated process, empowering both users and managers. Furthermore, the RS lays a solid foundation for future enhancements such as support for equipment reservations, LDAP authentication, event synchronization, and broader adoption by other student dormitory clubs.

This thesis demonstrates the feasibility of building a modern, scalable, and adaptable reservation platform in a student club context. Implementation represents a complete and functional base for continued development and adoption. To conclude, all tasks defined in the official thesis assignment were successfully accomplished.

Bibliography

1. BUBEN CLUB. *Welcome to Buben Wiki* [online]. 2024. Available also from: https://wiki.buk.cvut.cz/en/home. [Accessed 13.04.2025].
2. SUZ CUT. *Bubeneč Dormitory* [online]. © 2019. Available also from: https://www.suz.cvut.cz/en/dormitory-accommodation/bubenec-dormitory. [Accessed 04.04.2025].
3. STUDENT UNION CTU. *Buben Student Club* [online]. [N.d.]. Available also from: https://su.cvut.cz/cs/kluby/kolejni-kluby/buben. [Accessed 04.04.2025].
4. STUDENT UNION CTU. *Student Union CTU - About Us* [online]. [N.d.]. Available also from: https://su.cvut.cz/cs/o-studentske-unii-cvut/o-nas. [Accessed 04.04.2025].
5. GOOGLE LLC. *Get started with Groups* [online]. © 2025. Available also from: https://support.google.com/a/users/answer/9304806. [Accessed 13.04.2025].
6. GOOGLE LLC. *Google Calendar* [online]. [N.d.]. Available also from: https://calendar.google.com. [Accessed 05.04.2025].
7. CUT. *CTU Inaugurated the Reconstructed Bubeneč Dormitory* [online]. 2023. Available also from: https://aktualne.cvut.cz/en/reports/20230901-ctu-inaugurated-the-reconstructed-bubenec-dormitory. [Accessed 04.04.2025].
8. HAYES, Adam. *Radio Frequency Identification: What It Is, How It Works* [online]. 2024. Available also from: https://www.investopedia.com/terms/r/radio-frequency-identification-rfid.asp. [Accessed 13.04.2025].
9. ISIC ASSOCIATION. *Welcome to ISIC* [online]. © 2022. Available also from: https://www.isic.org. [Accessed 13.04.2025].
10. SALTO SYSTEMS, S.L. *Smart Access Reimagined* [online]. © 2025. Available also from: https://saltosystems.com/en/. [Accessed 13.04.2025].

11. POSTMAN, INC. *What is an API* [online]. © 2025. Available also from: https://www.postman.com/what-is-an-api/. [Accessed 13.04.2025].
12. GOOGLE LLC. *Google Calendar API overview* [online]. [N.d.]. Available also from: https://developers.google.com/workspace/calendar/api/guides/overview. [Accessed 05.04.2025].
13. POD-O-LEE CLUB. *Dormitory club Pod-O-Lee* [online]. © 2025. Available also from: https://podolee.cz/en. [Accessed 05.04.2025].
14. SILICON HILL CLUB. *Silicon Hill Club - Who We Are* [online]. © 1996-2025. Available also from: https://www.siliconhill.cz. [Accessed 05.04.2025].
15. SILICON HILL CLUB. *SHerna - Reservation* [online]. © 2017-2025. Available also from: https://sherna.siliconhill.cz/pages/reservation. [Accessed 05.04.2025].
16. MEVRIS GROUP S. R. O. *Better Hotel - hotel system* [online]. [N.d.]. Available also from: https://better-hotel.com/en/. [Accessed 18.04.2025].
17. ZIEMEK, Marcin. *Documenting non-functional requirements using FURPS+* [online]. 2022. Available also from: https://www.marcinziemek.com/blog/content/articles/8/article_en.html. [Accessed 13.04.2025].
18. *Understanding the MoSCoW prioritization | How to implement it into your project* [online]. 2023. Available also from: https://community.atlassian.com/forums/App-Central-articles/Understanding-the-MoSCoW-prioritization-How-to-implement-it-into/ba-p/2463999. [Accessed 13.04.2025].
19. NDEMBERA, Innocencia. *Understanding Use Cases* [online]. 2023. Available also from: https://medium.com/@queen_shecoder/understanding-use-cases-e72211b1b236. [Accessed 13.04.2025].
20. PYTHON SOFTWARE FOUNDATION. *Welcome! This is the official documentation for Python* [online]. © 2001-2025. Available also from: https://docs.python.org. [Accessed 13.04.2025].
21. HOSSAIN, Maruf. *How Python's Rich Ecosystem of Libraries is Transforming the Way Developers Approach Complex Projects* [online]. 2024. Available also from: https://dev.to/marufhossain/how-pythons-rich-ecosystem-of-libraries-is-transforming-the-way-developers-approach-complex-534g. [Accessed 13.04.2025].
22. GRUPPETTA, Stephen. *Python Readability, the PEP 8 Style Guide, and Learning Latin* [online]. 2021. Available also from: https://thepythoncodingbook.com/2021/10/11/python-readability-the-pep-8-style-guide-and-learning-latin/. [Accessed 13.04.2025].
23. TIANGOLO. *FastAPI framework, high performance, easy to learn, fast to code, ready for production* [online]. [N.d.]. Available also from: https://fastapi.tiangolo.com. [Accessed 05.04.2025].

24. SHAHEERSHAIK. *Why Python is Platform Independent and Portable* [online]. 2024. Available also from: https://medium.com/@shaheershaik81/why-python-is-platform-independent-and-portable-47bc16ed1263. [Accessed 13.04.2025].

25. MOJAHAR, Ali. *20 Most Popular Programming Languages in 2025* [online]. 2025. Available also from: https://www.index.dev/blog/most-popular-programming-languages-. [Accessed 13.04.2025].

26. RED HAT, INC. *What is a REST API* [online]. © 2025. Available also from: https://www.redhat.com/en/topics/api/what-is-a-rest-api. [Accessed 05.04.2025].

27. ASGI TEAM. *ASGI Documentation* [online]. © 2018. Available also from: https://asgi.readthedocs.io/en/latest/. [Accessed 05.04.2025].

28. *Welcome to Pydantic* [online]. [N.d.]. Available also from: https://docs.pydantic.dev/latest/. [Accessed 05.04.2025].

29. SMARTBEAR SOFTWARE. *What Is OpenAPI* [online]. © 2025. Available also from: https://swagger.io/docs/specification/v3_0/about/. [Accessed 05.04.2025].

30. SMARTBEAR SOFTWARE. *Swagger UI* [online]. © 2025. Available also from: https://swagger.io/tools/swagger-ui/. [Accessed 05.04.2025].

31. *The little ASGI framework that shines.* [Online]. [N.d.]. Available also from: https://www.starlette.io. [Accessed 05.04.2025].

32. BAYER, Mike. *The Python SQL Toolkit and Object Relational Mapper* [online]. © 2010-2025. Available also from: https://www.sqlalchemy.org. [Accessed 05.04.2025].

33. BAYER, Mike. *Welcome to Alembic's documentation* [online]. © 2010-2025. Available also from: https://alembic.sqlalchemy.org. [Accessed 05.04.2025].

34. *The fastapi-mail is a simple lightweight mail system, for sending emails and attachments* [online]. [N.d.]. Available also from: https://sabuhish.github.io/fastapi-mail/. [Accessed 05.04.2025].

35. *An ASGI web server, for Python* [online]. [N.d.]. Available also from: https://www.uvicorn.org. [Accessed 05.04.2025].

36. DJANGO SOFTWARE FOUNDATION. *Why Django? - Overview* [online]. © 2005-2025. Available also from: https://www.djangoproject.com/start/overview/. [Accessed 05.04.2025].

37. PALLETS. *Welcome to Flask's documentation.* [Online]. © 2010. Available also from: https://flask.palletsprojects.com/en/stable/. [Accessed 05.04.2025].

38. TIANGOLO. *FastAPI Benchmarks.* [Online]. [N.d.]. Available also from: https://fastapi.tiangolo.com/benchmarks/. [Accessed 05.04.2025].

39. SILICON HILL CLUB. *OAuth API informačního systému* [online]. [N.d.]. Available also from: https://is.buk.cvut.cz/oauth_api?change_language=en. [Accessed 05.04.2025].

40. SILICON HILL CLUB. *OAuth API informačního systému* [online]. [N.d.]. Available also from: https://is.buk.cvut.cz/oauth_api?change_language=en. [Accessed 05.04.2025].

41. *Access API – Documentation* [online]. 2025. Available also from: https://agata-new.suz.cvut.cz/pristupAPI/dokumentaceAPI.html. [Accessed 05.04.2025].

42. ATLASSIAN. *What is Git* [online]. © 2025. Available also from: https://www.atlassian.com/git/tutorials/what-is-git. [Accessed 13.04.2025].

43. GITLAB, INC. *GitLab is the most comprehensive AI-powered DevSecOps Platform* [online]. © 2025. Available also from: https://about.gitlab.com. [Accessed 13.04.2025].

44. ANACONDA, INC. *Conda Documentation* [online]. © 2017. Available also from: https://docs.conda.io. [Accessed 13.04.2025].

45. DOCKER, INC. *Develop faster. Run anywhere.* [Online]. © 2025. Available also from: https://www.docker.com. [Accessed 18.04.2025].

46. JGRAPH, LTD. *Create UML class diagrams* [online]. © 2005-2023. Available also from: https://www.drawio.com/blog/uml-class-diagrams. [Accessed 18.04.2025].

47. POWELL, Phill; SMALLEY, Ian. *What is monolithic architecture* [online]. 2024. Available also from: https://www.ibm.com/think/topics/monolithic-architecture. [Accessed 18.04.2025].

48. CLOUDFLARE, Inc. *What is HTTP* [online]. © 2025. Available also from: https://www.cloudflare.com/en-gb/learning/ddos/glossary/hypertext-transfer-protocol-http/. [Accessed 18.04.2025].

49. *Introducing JSON* [online]. [N.d.]. Available also from: https://www.json.org/json-en.html. [Accessed 18.04.2025].

50. PYTHON SOFTWARE FOUNDATION. *python code static checker* [online]. 2025. Available also from: https://pypi.org/project/pylint/#description. [Accessed 18.04.2025].

51. LEHTOSALO, Jukka; MYPY CONTRIBUTORS. *Welcome to mypy documentation* [online]. © 2012-2025. Available also from: https://mypy.readthedocs.io/en. [Accessed 18.04.2025].

52. GONCHAROV, Ivan. *REDOC – AN OPENAPI-POWERED DOCUMENTATION UI* [online]. 2016. Available also from: https://swagger.io/blog/api-development/redoc-openapi-powered-documentation/. [Accessed 19.04.2025].

53. META PLATFORMS, INC. *React | The library for web and native user interfaces* [online]. [N.d.]. Available also from: https://react.dev. [Accessed 24.04.2025].

54. *Getting Started* [online]. [N.d.]. Available also from: https://vite.dev/guide/. [Accessed 24.04.2025].

55. TAILWIND LABS, INC. *Get started with Tailwind CSS* [online]. © 2025. Available also from: https://tailwindcss.com/docs/installation/using-vite. [Accessed 24.04.2025].

56. FULLCALENDAR LLC. *FullCalendar Documentation* [online]. © 2025. Available also from: https://fullcalendar.io/demos. [Accessed 24.04.2025].

57. GITLAB, INC. *Issues* [online]. [N.d.]. Available also from: https://docs.gitlab.com/user/project/issues/. [Accessed 24.04.2025].

58. MOZILLA FOUNDATION. *Cross-Origin Resource Sharing (CORS)* [online]. © 1998-2025. Available also from: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS. [Accessed 21.04.2025].

59. FLIPBIT03. *SQLAlchemy Easy Soft-Delete* [online]. 2023. Available also from: https://pypi.org/project/sqlalchemy-easy-softdelete/. [Accessed 21.04.2025].

60. MICROSOFT CORPORATION, INC. *Understand lightweight directory access protocol (LDAP) basics in Azure NetApp Files* [online]. © 2025. Available also from: https://learn.microsoft.com/en-us/azure/azure-netapp-files/lightweight-directory-access-protocol. [Accessed 25.04.2025].

61. TIANGOLO. *Dependencies | What is "Dependency Injection"* [online]. [N.d.]. Available also from: https://fastapi.tiangolo.com/tutorial/dependencies/. [Accessed 25.04.2025].

62. *ISKAM* [online]. [N.d.]. Available also from: https://web.suz.cvut.cz. [Accessed 12.05.2025].

63. DOCKER, INC. *How Compose works* [online]. © 2013-2025. Available also from: https://docs.docker.com/compose/intro/compose-application-model/. [Accessed 03.05.2025].

64. MICROSOFT CORPORATION, INC. *What is a Virtual Machine* [online]. © 2025. Available also from: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-a-virtual-machine. [Accessed 03.05.2025].

65. OPENJS FOUNDATION. *About Node.js* [online]. [N.d.]. Available also from: https://nodejs.org/en/about. [Accessed 03.05.2025].

66. NGINX, INC. *NGINX about* [online]. [N.d.]. Available also from: https://nginx.org/en/. [Accessed 03.05.2025].

67. INTERNET SECURITY RESEARCH GROUP. *About Let's Encrypt* [online]. 2021. Available also from: `https://letsencrypt.org/about/`. [Accessed 03.05.2025].

68. KREKEL, Holger. *Pytest: Helps You Write Better Programs* [online]. © 2025. Available also from: `https://docs.pytest.org`. [Accessed 03.05.2025].

69. BATCHELDER, Ned. *Coverage.py* [online]. © 2009–2025. Available also from: `https://coverage.readthedocs.io/en/7.8.0/`. [Accessed 13.05.2025].

Contents of the attachment

```
/
├── readme.txt ..... brief description of the media contents
├── src
│   ├── impl ..... source code of the implementation
│   └── thesis ..... source files of the thesis in LaTeX format
├── appendix ..... additional materials
│   └── coverage.xlsx ..... table of coverage of fun. req. of uc
└── text ..... text of the thesis
    └── thesis.pdf ..... thesis in PDF format
```