ASSIGNMENT OF BACHELOR’S THESIS

Title:	Reservation System as a Service
Student:	Peter Matta
Supervisor:	Ing. Adam Hořčica
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2020/21

Instructions

Design and implement a reservation system for a medium-sized organization. The provided solution should be customizable for the individual needs of asset owners. Asset owners should be isolated from each other in multitenant meaner. The result should be a web application. API for further customization of front end and 3rd party integration is a must. Demonstrate solution as a pilot in the Silicon Hill student club at Strahov dormitory.

Thesis realization process requirements:
1) analysis of the current situation in the organization and existing solutions,
2) design of the system,
3) implementation of the system,
4) testing and the documentation,
5) pilot deployment.

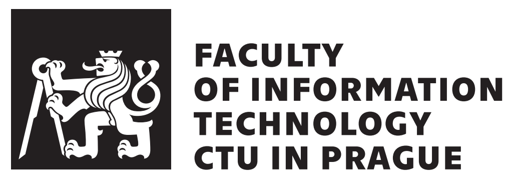
References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 12, 2020



Bachelor's thesis

Reservation System as a Service

Peter Matta

Department of Software Engineering

Supervisor: Ing. Adam Hořčica

June 5, 2020

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. I further declare that I have concluded an agreement with the Czech Technical University in Prague, on the basis of which the Czech Technical University in Prague has waived its right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act. This fact shall not affect the provisions of Article 47b of the Act No. 111/1998 Coll., the Higher Education Act, as amended.

In Prague on June 5, 2020 ....

Czech Technical University in Prague

Faculty of Information Technology

© 2020 Peter Matta. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Matta, Peter. *Reservation System as a Service.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstract

This thesis covers the development process of a SaaS platform for the creation and management of reservation systems. It is a multi-tenant, workspace based system, inspired by the successful format of well known SaaS products such as Slack or Microsoft Teams. The platform was developed in collaboration with and for the Silicon Hill student club and represents the first attempt to merge its existing reservation systems into one. This thesis aims to provide such a platform while covering the requirement gathering process, iterative development of the platform, and the reasoning behind the architectural decisions. The conditions and environment in which the platform was developed led to several unconventional software design decisions, such as following Domain Driven-Design principle to achieve proper modularization and using Event Sourcing technique for storing the data to achieve the flexibility of the domain.

Keywords Reservation System, Booking, Open Source, Multi Tenant, Event Sourcing

Abstrakt

Táto práca zachytáva proces tvorby SaaS platformy pre tvorbu a správu rezervačných systémov. Je to multitenantný systém založený na workspace-och (tj. z pohľadu užívateľa mnohých oddelených inštanciách rovnakého systému). Táto štruktúra systému bola inšpirovaná známymi a úspešnými SaaS produktami s rovnakým formátom, ako Slack a Microsoft Teams. Platforma predstavuje prvý pokus o zlúčenie rezervačných systémov využívaných študentským klubom Silicon Hill, pre ktorý, a v spolupráci s ktorým bola vyvinutá. V rámci tejto práce popisujeme zber požiadaviek na túto platformu, jej postupný vývoj a rozhodnutia vedúcie k jej finálnej architektúre. Vzhľadom na podmienky a prostredie, v akých bola platforma vyvinutá, sme si zvolili niektoré, z hľadiska softwarovej architektúry nezvyčajné, prístupy, ako napr. filozofiu Domain Driven Design, alebo techniku Event Sourcing (využitá pri ukladaní dát).

Klíčová slova Rezervačný Systém, Rezervácia, Open Source, Multi Tenant, Event Sourcing

Contents

Introduction **1**
Aim of the Thesis 2
Structure of the Thesis 2

1 Analysis **3**
1.1 User Roles 3
1.2 Requirements 3
1.2.1 Functional Requirements 4
1.2.2 Non-Functional Requirements 5
1.2.3 Constraints 7
1.3 Domain 7
1.3.1 Bounded Contexts 7
1.3.2 Conceptual Model 8
1.3.3 Domain Events 8
1.3.4 Aggregates 9
1.3.5 Reservation State 10
1.4 Existing solutions 11
1.4.1 Easy!Appointments 11
1.4.2 Booked 12
1.4.3 WebCalendar 12
1.4.4 Reservio 12
1.4.5 Summary 12

2 Architecture **15**
2.1 System Context 15
2.2 System Architecture 16
2.3 Client Architecture 16
2.3.1 Client Applications 16
2.4 Server Architecture 17

2.4.1 API Gateway . . . 17
2.4.2 Monolithic vs. Microservices Architecture . . . 17
2.4.3 Modularization . . . 18
2.4.4 Core Module Integration . . . 19
2.5 Data Model . . . 21
2.5.1 CRUD . . . 21
2.5.2 Event Sourcing . . . 22

3 Technologies Used **25**
3.1 Programming Language . . . 25
3.2 Client Technologies . . . 26
3.3 API Gateway . . . 26
3.4 Dependency Injection . . . 26
3.5 Data Storage . . . 27
3.6 Database Integration . . . 27
3.7 Email Integration . . . 27
3.8 Continuous Integration . . . 27
3.9 Containerization . . . 27

4 Implementation **29**
4.1 Workspace Isolation . . . 29
4.1.1 Entity Isolation . . . 29
4.1.2 Credentials Isolation . . . 29
4.1.3 Domain Isolation . . . 30
4.2 Authentication and Authorization . . . 30
4.2.1 Authentication . . . 30
4.2.2 Authorization . . . 30
4.2.3 Authorization Tokens . . . 31
4.2.4 Email Verification . . . 32
4.2.5 Integration with Silicon Hill OAuth2.0 . . . 32
4.3 Personal Data and Anonymization . . . 34
4.3.1 Personal Data Storage . . . 34
4.3.2 Data Anonymization . . . 34
4.4 REST API . . . 35
4.4.1 Authorization . . . 35
4.4.2 API Versioning . . . 35
4.4.3 Console API . . . 35
4.4.4 Setup API . . . 38
4.4.5 Workspace API . . . 39
4.5 Event Sourcing . . . 46
4.5.1 Overview . . . 46
4.5.2 Aggregate Service . . . 46
4.5.3 Aggregate Repository . . . 48
4.5.4 Event Store . . . 50

4.5.5 Snapshotting . . . 51
4.6 Message Broker . . . 53
4.6.1 Overview . . . 53
4.6.2 Outbox . . . 54
4.6.3 Inbox . . . 54
4.6.4 Event Consumers . . . 56
4.7 Webhooks . . . 58
4.7.1 Supported Events . . . 58
4.7.2 Webhook Realization . . . 58
4.7.3 Payload Validation . . . 58
4.7.4 Testing of Webhooks . . . 58
4.8 Testing . . . 59
4.8.1 Client Testing . . . 59
4.8.2 Server Testing . . . 59
4.8.3 User Testing . . . 63

Conclusion **65**

Bibliography **67**

A Domain Events **73**
A.1 Diagram notation . . . 73
A.2 List of Domain Events . . . 73
A.2.1 Workspace Request Life cycle . . . 73
A.2.2 Workspace Life cycle . . . 76
A.2.3 Resource Life cycle . . . 77
A.2.4 Reservation Life Cycle Events . . . 78
A.2.5 Reservation Comments . . . 80
A.2.6 User Life cycle . . . 82

B Acronyms **85**

List of Figures

1.1 Conceptual Diagram . . . 9
1.2 Reservation State Diagram . . . 11

2.1 System Context Diagram . . . 16
2.2 Server API Gateway Pattern . . . 17
2.3 Module Layers . . . 19
2.4 Module interactions when creating a reservation . . . 20
2.5 Event emission handling caused by an external stimulus . . . 21
2.6 Data model separation on module level . . . 22
2.7 Derivation of Reservation from Reservation Events . . . 23

4.1 User Entity Relation with Workspace . . . 29
4.2 Silicon Hill OAuth2.0 Authentication . . . 33
4.3 Identity Table . . . 34
4.4 Event Sourcing Overview . . . 47
4.5 Service-Repository Command execution . . . 47
4.6 Aggregate Repository . . . 48
4.7 Command Handler . . . 48
4.8 Event Handler . . . 49
4.9 Event Store . . . 50
4.10 Single Row of Event Store . . . 51
4.11 Events Table Structure . . . 52
4.12 Snapshot Store . . . 52
4.13 Snapshot Table . . . 53
4.14 Outbox Pattern – Successful Processing of Event . . . 55
4.15 Outbox Pattern – Failure Encountered During Event Processing . . . 55
4.16 Inbox Pattern – First Delivery of Event . . . 56
4.17 Inbox Pattern – Subsequent Delivery of Event . . . 57
4.18 Webhook Testing . . . 59
4.19 Test Pyramid . . . 60

A.1 Workspace Request Life cycle 75
A.2 Workspace Life cycle 77
A.3 Resource Life cycle 78
A.4 Reservation Life cycle 81
A.5 Reservation Comment Life cycle 82
A.6 User Life cycle 84

List of Tables

1.1 Existing Systems Comparison 13

4.1 Outbox Table 54
4.2 Inbox Table 54

Introduction

Managing reservations for various resources, such as common rooms or shared goods, is well known and understood problem domain which many organizations face daily. Such an organization is, for example, Silicon Hill.

Silicon Hill is a student club operating at the Strahov dormitories providing its members with services not related to the accommodation itself, such as internet connection, rental of common rooms, equipment, and many more.

Silicon Hill is a highly decentralized organization in its nature, with multiple departments responsible for the services they provide. The large variety of the provided services results in vast differences in the requirements that each department place on the reservation system, which has led to multiple reservation systems being created. Having numerous reservation systems, each developed, maintained, and tailored for a specific department, provides some advantages such as easy customization and specialization of the system for the type of services the department offers. However, as with any software design decision, tradeoffs had to be made.

Firstly, since every department is responsible for the development of its own reservation system, many departments manage reservations over email communication and/or shared spreadsheets. Doing so has proven to lack transparency and is hard to maintain long term. Secondly, the departments developing their own reservation system or taking advantage of the off the shelf solution often fail to leverage the existing software infrastructure already available. This includes accessing members' information from the club-wide Information System or using its Single Sign-On (SSO) feature. Lastly, the biggest issue that the departments have to face is the constant rotation of their members. The vast majority of the departments' members are students that study at Czech Technical University in Prague and/or reside at the Strahov dormi-

tory. Since the majority of the students finish their studies in 5 years or less, the people responsible for the maintenance and development of the reservation systems tend to leave without having an adequate fellow department member to take over.

Aim of the Thesis

The primary outcome of this thesis is to provide a centralized platform for creating and managing reservation systems for Silicon Hill departments. The platform should work in a workspace based fashion following the success of multiple Software as a Service (SaaS) type applications. It should allow each department to create and configure a reservation system based on their own needs, provide easy integration with Silicon Hill's existing software infrastructure, and enable centralized management of the systems. The software solution implementing the platform should be modular and flexible, allowing the system to be developed in an agile way. The final software solution should be deployed on the existing Silicon Hill infrastructure.

Structure of the Thesis

Analysis In this chapter, we cover the analysis that was made before the project began as well as throughout the development of the project. This chapter lists identified requirements, explores the problem domain, and investigates existing solutions.

Architecture The chapter focuses not only on the architecture itself but also on the architectural decisions that lead to the final system architecture.

Technologies Used In this chapter, we list the significant technology decisions that were made and discuss the reasons behind them.

Implementation This chapter explores the essential implementation details of the system. Its primary focus is on workspace isolation, event sourcing implementation, and application testing.

Chapter 1

Analysis

This chapter lists identified requirements placed on the system, explores the problem domain using the Domain-Driven Design (DDD) approach [1], and investigates existing solutions.

1.1 User Roles

A user role defines how a user of a software system is allowed to interact with it. The analysis of the existing software systems currently being used by Silicon Hill identified five such roles:

- System administrator
- Workspace owner
- Workspace administrator
- Workspace maintainer
- Workspace user

1.2 Requirements

The following section lists requirements that were identified during the initial analysis as well as the development process. The requirements are categorized into functional requirements, non-functional requirements, and constraints.

The initial set of requirements was gathered by analyzing existing reservation systems at Silicon Hill, and by meeting with department representatives maintaining the systems or the physical resources that could have been reserved

using the systems. Additional requirements were identified in several sessions throughout the development of the system, in which the various department representatives could interact with the system and give feedback on it.

1.2.1 Functional Requirements

Functional requirements state what the system must do and how it must behave or react to runtime stimuli. [2]

FR01 Any resident of Strahov dormitories shall be able to access any reservation system workspace.

FR02 Any Silicon Hill member shall be able to open a request for a reservation system workspace.

FR03 System administrators shall be able to confirm opened workspace requests.

FR04 System administrators shall be able to decline opened workspace requests.

FR05 System administrators shall be able to transfer ownership of the workspace to another workspace user.

FR06 A workspace user shall be able to create a reservation for one or more resources.

FR07 A workspace user shall be able to create a reservation for any time range that starts in the future.

FR08 A workspace user shall be able to cancel a reservation assigned to them.

FR09 A workspace user shall be able to delete their account.

FR10 A workspace maintainer shall be able to perform all actions any workspace user can.

FR11 A workspace maintainer shall be able to create a new resource.

FR12 A workspace maintainer shall be able to delete an existing resource.

FR13 A workspace maintainer shall be able to deactivate an active resource.

FR14 A workspace maintainer shall be able to activate an inactive resource.

FR15 A workspace maintainer shall be able to confirm a requested reservation.

FR16 A workspace maintainer shall be able to reject a requested reservation.

FR17 A workspace maintainer shall be able to cancel a confirmed reservation.

FR18 A workspace maintainer shall be able to place a ban on a workspace user.

FR20 A workspace maintainer shall be able to lift a ban from a workspace user.

FR21 A workspace maintainer shall be able to promote a workspace user to workspace maintainer.

FR22 A workspace maintainer shall be able to demote another workspace maintainer to a workspace user.

FR23 A workspace administrator shall be able to perform all actions a workspace maintainer can.

FR24 A workspace administrator shall be able to ban a workspace maintainer.

FR25 A workspace administrator shall be able to promote a workspace maintainer to a workspace administrator.

FR26 A workspace administrator shall be able to demote a workspace administrator to a workspace maintainer.

FR27 A workspace owner shall be able to perform all actions workspace an administrator can.

FR28 A workspace owner shall be able to transfer ownership of the workspace to a workspace user.

FR29 A workspace owner shall be able to delete the entire workspace.

1.2.2 Non-Functional Requirements

Non-Functional Requirements define the quality attributes of the system. These requirements are categorized by the quality attributes they define.

1.2.2.1 Modifiability

Since the system is supposed to serve many Silicon Hill departments, which all may place different requirements on the system, the system's modifiability is crucial.

NFR01 The system shall be modular so that the changes to the system are localized.

NFR02 The system shall allow easy changes to be made to its implementation.

NFR03 The system shall use continuous integration (CI) pipeline to minimize and automate any verification and build tasks.

NFR04 The system shall be easy to deploy.

1.2.2.2 Security

NFR05 Only authenticated users shall be able to access the system.

NFR06 The system shall not store any personal information that is not necessary.

NFR07 The system shall destroy all personal information after a user is deleted.

NFR08 The system shall destroy the personal information of all workspace users when the workspace is deleted.

NFR09 A system administrator shall not be able to access any workspace.

NFR10 A system administrator shall not see any information about workspace users (except the workspace owner).

NFR11 Workspace user shall not be able to access other users' personal information.

NFR12 Workspace user/maintainer/administrator/owner shall be able to access only the workspace they belong to.

NFR13 The correctness of any communication with an external system must be verifiable.

NFR14 Network communication within the system must use HTTPS.

1.2.2.3 Interoperability

NFR15 The system shall allow Silicon Hill members to use Silicon Hill OAuth2.0 API [3] to authenticate.

NFR16 The system shall allow workspace administrators to configure webhooks to be dispatched when a reservation is created, rejected, canceled, or completed.

NFR17 The system shall expose a REST API.

1.2.2.4 Debuggability

NFR18 The system shall provide logging of the system behavior.

NFR19 The system shall include high test coverage.

1.2.3 Constraints

CS01 The system shall be deployed on the Silicon Hill infrastructure.

CS02 The system shall use MySQL for its database.

CS03 The system shall be a web application.

CS04 The system shall be developed by June 1, 2020.

CS05 The system shall be accessible under a root domain `rs.sh.cvut.cz`.

1.3 Domain

Definition: *A sphere of knowledge, influence, or activity. The subject area to which the user applies a program is the domain of the software.* [1]

The following section explores core domain aspects such as entities, relationships between them, and domain events. The processes used to explore, describe, and model the domain, take inspiration from the Domain-Driven Design (DDD) [1]. Any non-essential elements of the system (e.g., integrations with external systems) are omitted for clarity.

1.3.1 Bounded Contexts

Definition: *A description of a boundary (typically a subsystem, or the work of a particular team) within which a particular model is defined and applicable.* [1]

The analysis of the domain revealed that the domain could be split into four bounded contexts:

Resource-Reservation The primary domain entities forming this bounded context are *Resource*, *Reservation*, *Assignee*, and *Reservation Manager*. All of the entities in this bounded context are scoped to a single workspace. *Reservation Managers* can create and edit *Reservations* for one or more *Resources*. Each *Reservation* must be assigned to a single *Assignee*. *Reservations* can be assigned *Reservation Comments* by *Comment Authors*. Workspace *Resources* can be managed by *Resource Managers*.

User-Access This bounded context encapsulates *Users*, their *Identities*, and *User Roles*. All of the entities in this bounded context are scoped to a single workspace. Each *User* must have a single, unique *Identity*. Each *User* must also have a *User Role* assigned.

Workspace-Management To create a *Workspace* a *Workspace Requester* must first open a *Workspace Request*. When a *Workspace Request* is opened, a *System Administrator* can confirm or decline it. If *Workspace Request* is confirmed, then a *Workspace* is created, making its *Workspace Requester* the *Workspace Owner* of the *Workspace*. If, however, a *Workspace Request* is declined, no *Workspace* is created, and all information related to the *Workspace Requester* is destroyed.

Console This bounded context encapsulates system management. It holds *Console Account* entities, which are responsible for the creation and deletion of other *Console Accounts*. There must be at least one *Console Account* in the system.

1.3.2 Conceptual Model

Diagram in figure 1.1 captures a conceptual model of entities described in subsection 1.3.1 and relationships between them. The entities captured in the diagram are grouped by the bounded context they belong to.

1.3.3 Domain Events

Definition: *Domain event captures the memory of something interesting which affects the domain.* [4]

The previous subsections, 1.3.1 and 1.3.2, capture the structure of the domain. This subsection, however, focuses on the behavior of the domain, which is usually done by identifying the domain events in the system.

There are many techniques for identifying domain events, one of which is called *Event Storming* [5][6]. The idea of event storming is first to identify what happens in the system, the *events*, then determine what actions cause these events to be produced, the *commands*, and lastly, what triggers those commands, the *actors*. A full list of identified domain events with diagrams can be found in appendix A.

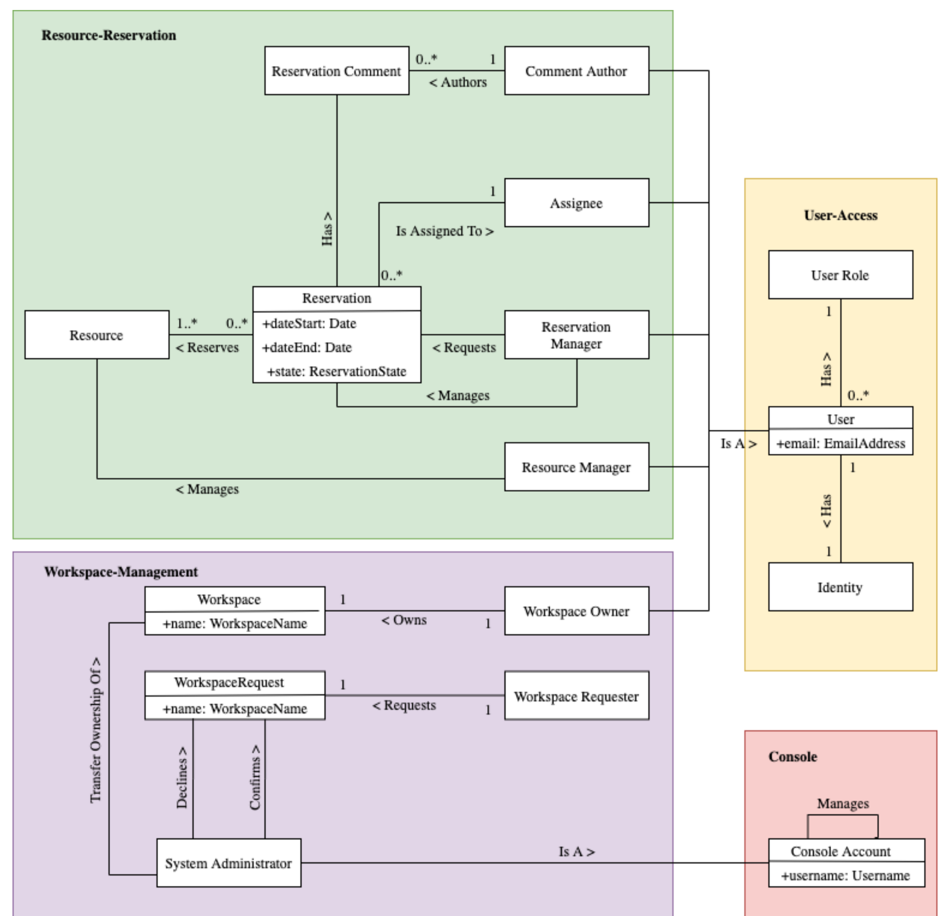


Figure 1.1: Conceptual Diagram

1.3.4 Aggregates

Definition *Aggregate is a cluster of domain objects that can be treated as a single unit.* [7]

The domain model identified in subsections 1.3.1, and 1.3.3, is quite complex, which would make it hard to maintain consistency across the entire system. The DDD, however, suggests clustering entities with their value objects to *aggregates* [1] and defining consistency boundaries around them.

Using the entities from the context diagram (figure 2.1) and the domain events associated with them, lead to identification of the following aggregates:

- Reservation
- Reservation Comment

- Resource
- User
- Workspace
- Workspace Request
- Console Account

1.3.5 Reservation State

The most important entity of the domain, which the entire system is built around, is *Reservation*. *Reservations* represent a time constrained allocation of one or more *Resources* by an *Assignee*. Once a *Reservation* is created, it cannot be deleted, only its state can be changed (figure 1.2).

When a *Reservation* is created, its state is set to the `requested` state–this represents an *Assignee* wanting to make a claim for selected *Resource* in given time range. At this point, the *Reservation* can be either canceled by the *Assignee*, or confirmed or rejected by the *Reservation Managers*. When the *Assignee* cancels the *Reservation*, the *Reservation* reaches a terminal, `canceled`, state. When a *Reservation Manager* rejects the *Reservation*, it reaches a terminal, `rejected`, state.

Confirmation of the *Reservation* moves the *Reservation* to the `confirmed` state. In each *Workspace* there must be at most one `confirmed` *Reservation* allocating any given set of *Resources* at any point in time. Once a *Reservation* is confirmed, it can be canceled by either its *Assignee* or any of the *Reservation Managers*, at which point the *Reservation* reaches a terminal, `canceled`, state. If the *Reservation's* end date reaches the present, it is automatically competed, reaching a terminal, `completed`, state.

Any state transitions not mentioned in this section, such as from `canceled` back to `requested`, are considered invalid state transitions and are not permitted. Each *Reservation* in either `requested` or `confirmed` state can be changed by their *Assignee* or any *Reservation Manager*.

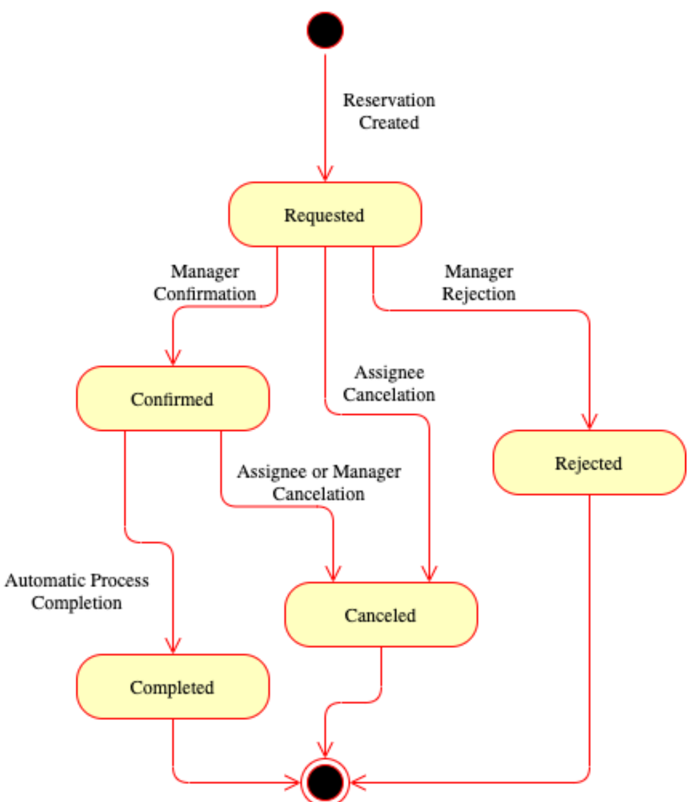


Figure 1.2: Reservation State Diagram

1.4 Existing solutions

There are many reservation systems ranging from open-source to paid SaaS solutions. The primary issue with the majority of them is that they are often single-purpose systems, e.g., restaurant/hotel booking systems or appointment systems. Only a small number of them are general enough to be used by an organization as large as Silicon Hill.

1.4.1 Easy!Appointments

Easy!Appointments [8] is an open-source web application focused on booking appointments with other people. It is well maintained with more than a thousand stars on Github [9]. It is highly customizable with optional integration with external services such as email and Google Calendar. It also provides Docker images for easy deployment.

However, it does not provide user authentication. It also does not offer a way to scope/group resources (people in this case), which would require an

instance of the application to be deployed per each Silicon Hill department, which is the primary issue with the current approach.

1.4.2 Booked

Booked [10] is a general-purpose scheduling system with a high degree of customization. It allows the configuration of different resources and their usage limits. It also provides user authentication out of the box and the ability to group and manage users. It has an active community of contributors formed around the SaaS version of the application.

On the other hand, the application does not provide a grouping of resources, which means the app could not be used as a Silicon Hill-wide reservation system. It also does not offer a way to allow user access scoped to specific resources. Using Booked would, therefore, result in an instance of the application needed to be deployed per each department without a way to manage the instances easily. The app is also expected to be run in a native environment, which adds additional complexity.

1.4.3 WebCalendar

WebCalendar [11] is another open-source scheduling system focused on maintaining personal or group schedules. It does have 62 reviews on Source Forge, and more than 80 stars on Github [12]. However, it seems like it is not well maintained anymore[1]. It also does not provide any integration with external services nor user authentication.

1.4.4 Reservio

Reservio [13] is a closed-source, SaaS-based, general-purpose reservation system. It does provide a multitude of customization and integration options. They offer a three-tier program with an option to add additional features. There are two issues, however. There is no option to host the software on-premise as Silicon Hill requires, and it does not provide a free tier.

1.4.5 Summary

Overall, none of the considered solutions fits all criteria for the system, as can be seen in table 1.1. Considered was also extending one of the existing Silicon Hill's reservation systems, such as SHerna [14]. However, all of them were either too inflexible or lacked in development support.

[1] At the time of writing the thesis, it seems that the WebCalendar has picked up some new development activity. However, at the time of the research, the latest commit of WebCalendar was more than six months old, without any signs of active development.

Name	Free	Authentication	Resource Scoping
Easy!Appointments	Yes	No	No
Bookend	Yes	Yes	No
WebCalendar	Yes	Yes	No
Reservio	No	Yes	Yes

Table 1.1: Existing Systems Comparison

Chapter 2

Architecture

Definition: *The software architecture of a system is the set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both.* [2]

Every software system has an architecture, either one that was planned for or one that emerged over the lifetime of the system. The architecture goal is to shape the system to achieve non-functional requirements (1.2.2) while not breaking any constraints (1.2.3). This chapter defines the architecture of our system, describes decisions that were made, and explores the tradeoffs of those decisions.

2.1 System Context

Almost no software system works on its own, but rather it is set in an environment in which it has to interact with various external entities or other systems. The context of the RSaaS is captured in the diagram in figure 2.1, which clearly defines the system's boundary.

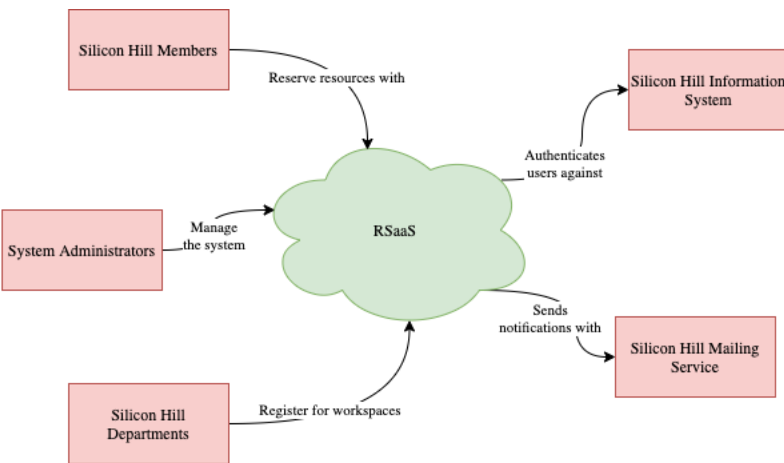


Figure 2.1: System Context Diagram

2.2 System Architecture

The constraint **CS03** defined the web to be a target platform. The two most commonly used top-level web architectures are *Client-Server* and *Peer-to-Peer* [2]. The *Client-Server* is, however, a better fit because of its centralized nature, which is essential to implement requirements **NFR15**, **NFR16**, and **NFR17**.

2.3 Client Architecture

In *Client-Server* architecture, there are two approaches to define the client's architecture: *Thin Client* [15] and *Thick Client* [16]. To make the system decoupled even on this abstraction level and to fit requirements **NFR01** and **NFR02**, *Thick Client* was chosen.

2.3.1 Client Applications

The client is split into three separate applications according to their responsibilities.

Landing Landing application is the entry point to the system. It lists existing workspaces and allows users to open requests for new workspaces.

Console Console application is used by system administrators to manage workspace requests, inspect existing workspaces, and change their owners.

Workspace Workspace application is used by workspace users to create reservations for selected resources as well as workspace maintainers, administra-

tors, and owners, to configure and attend the workspace. It is the primary application users use to interact with the system.

2.4 Server Architecture

2.4.1 API Gateway

The primary architectural driver for choosing the top-level architecture of the server was constraint **CS05**, which effectively requires the system to have a single entry point. The **CS05** led to the secondary architectural driver, which was to extract client assets serving from handling the REST API requests since Silicon Hill infrastructure does not provide any solution for serving static files. This naturally led to a pattern called Gateway Pattern [17][18] captured in the diagram in figure 2.2.

2.4.2 Monolithic vs. Microservices Architecture

There are many benefits and drawbacks to both approaches [19][20]. Microservices provide a great way to decouple the system into separate units, independently develop, and scale them, and provide a high degree of fault isolation. Together with the requirements **NFR01**, and **NFR02**, they seem like a better choice.

However, not all of the benefits are applicable in the context of our system. Independent development and deployment of the services make sense primarily in organizations with a large number of development teams [21]. Since the system is developed by a single person and is expected to be maintained

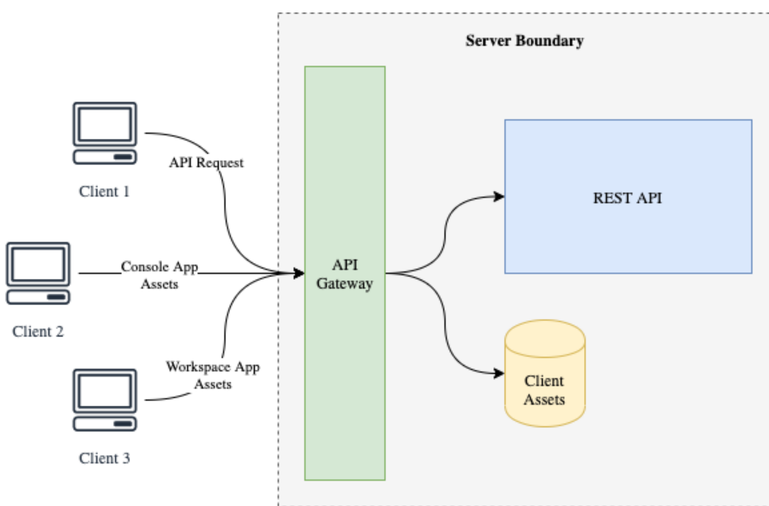


Figure 2.2: Server API Gateway Pattern

by no more than ten people, the operational complexity of the microservices approach outweighs this benefit. Independent scaling of the microservices does not apply either. The total number of residents at Strahov dormitories, the system's potential users, is less than five thousand, making horizontal scaling possible with a monolith, more than sufficient.

Decomposing the system into independently deployable units and integrating them over an explicitly defined interface significantly decouples the system's parts. A similar degree of separation can also be achieved in the monolithic system by proper modularization (2.4.3). On the other hand, the same degree of fault isolation is not.

The monolithic approach has many advantages, as well. The monolithic approach yields a single application and is developed in a single codebase. Therefore it is much easier to test, debug, and deploy the application. But, more importantly, it has a much lower operational overhead compared to the microservices approach.

After evaluating both methods and following guidelines for choosing appropriate architecture [22], the monolithic approach was chosen, making its lack of fault isolation, the only significant tradeoff that was made.

2.4.3 Modularization

Modular Monolith (sometimes referred to as Majestic Monolith [21]) suggests modularization of the system. There are many modularization patterns, such as Layered pattern [2], Multi-tier pattern [2], Pipes and Filters [23], Vertical Slices [24][25], and many others.

Early in the development process, the Vertical Slices pattern was chosen because it results in a system with high cohesion. However, the coupling between the modules started to be problematic after the system reached a certain size, requiring changes to more than one module to be made. Therefore it was decided to refactor out of this approach and create modules grouped into three layers: *edge*, *core*, and *infrastructure* reflecting the Onion Architecture pattern [26] on the module level captured in the diagram in figure 2.3. Within each layer, however, the modules still follow the Vertical Slices pattern.

Core Layer Modules in the core layer map directly to the bounded contexts identified in section 1.3.1. They are atomic, self-contained, and do not depend on any of the edge, infrastructure, or other core modules. They hold all of the core domain logic. Although core modules do not communicate directly, they can listen for domain events (1.3.3) emitted by other modules (2.4.4).

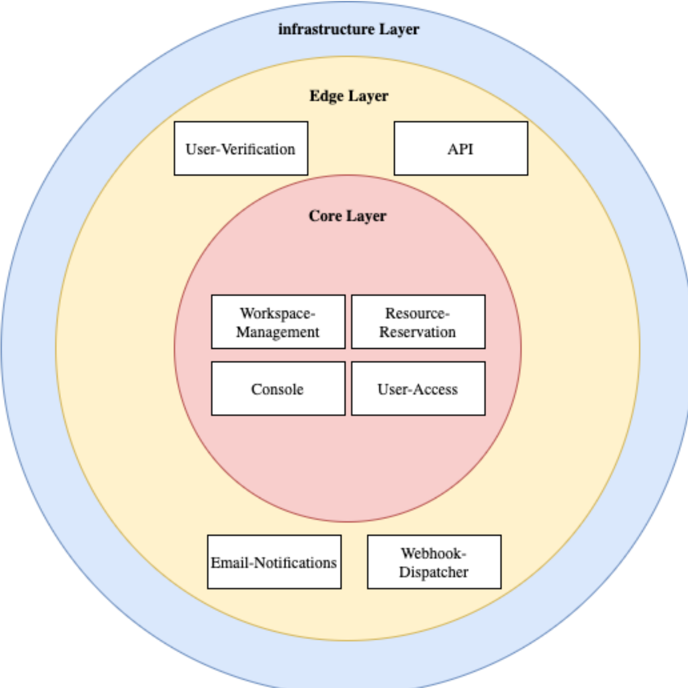


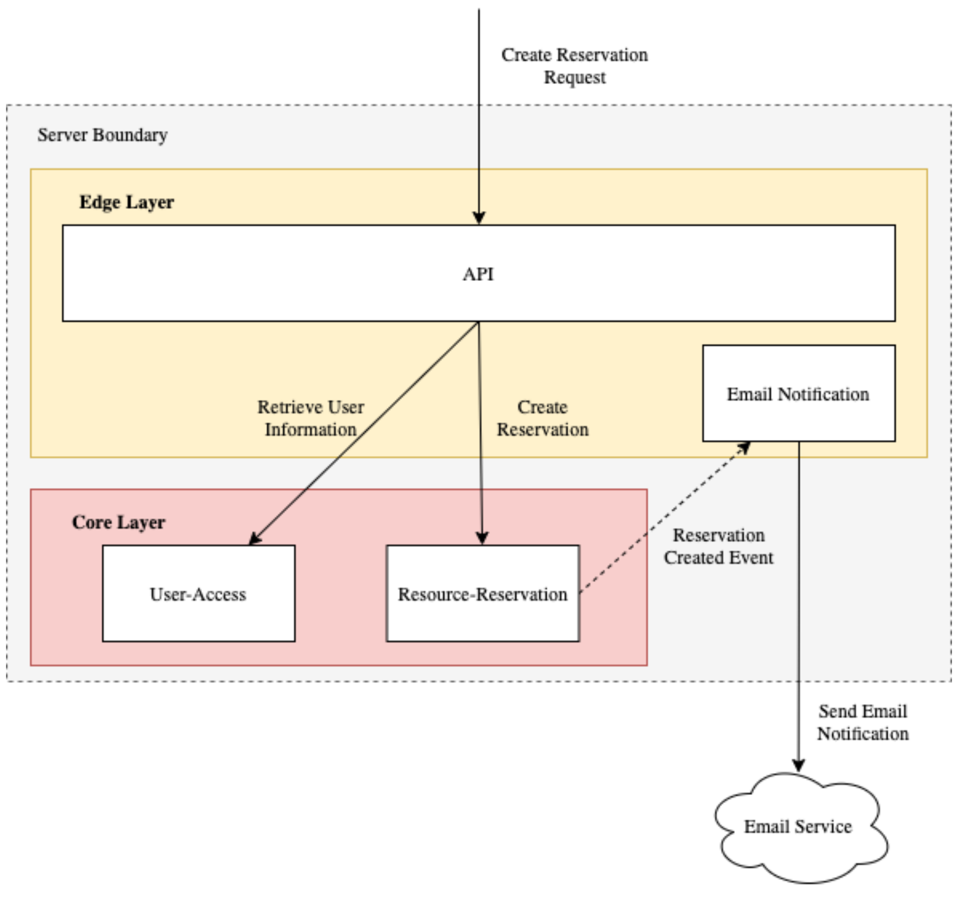
Figure 2.3: Module Layers

Edge Layer Modules in the edge layer are responsible for interacting with external systems (e.g., such interaction is captured in figure 2.4). They expose public API allowing clients to communicate with the system as well as allowing the system to interact with external systems. Since they do not hold any business logic related to the core domain, they communicate with core or other edge modules. These modules contain all validation logic that ensures the validity of the entities used by the core layer modules.

Infrastructure Layer Modules in the infrastructure layer are responsible for bootstrapping the entire system and implementation of underlying protocols, such as message broker integration. They do not hold any business logic.

2.4.4 Core Module Integration

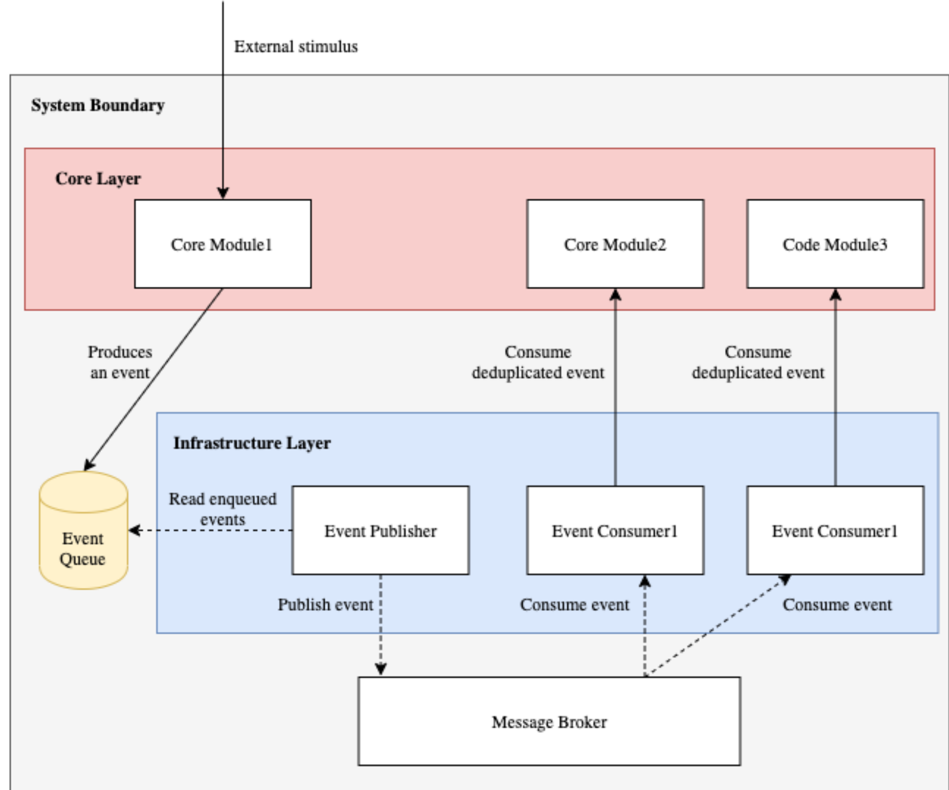
Besides the restrictions put in place in the subsection 2.4.3, edge or other core modules need to react to domain events produced by other core modules. Handling these cases synchronously would require each command entry-point module to know about every module interested in events possibly produced by the command. Processing events in such a way would cause too much coupling between the modules, which is why the asynchronous approach was chosen.



direct calls to given modules
asynchronous event consumption

Figure 2.4: Module interactions when creating a reservation

The integration is realized by a message broker [27] (figure 2.5), which allows modules to publish and subscribe to the domain events. Although asynchronous integration is a better fit for the system, using a message broker comes with drawbacks, too, primarily, eventual consistency [28] between the modules. To minimize this drawback, modules hold strong consistency within their boundaries.



calls that occur exactly once per event emission
calls that occur at least once per event emission

Figure 2.5: Event emission handling caused by an external stimulus

2.5 Data Model

Typically data model is shared throughout the entire system. However, using the vertical slices pattern [25] on the module level, each module defines and maintains its own data model (figure 2.6). In theory, each of the modules could even use a different storage mechanism, database, or schema.

2.5.1 CRUD

Usually, the data model is derived from the conceptual model (figure 1.1) of the system mapping its entities to respective database tables with associated CRUD operations. This approach allows us to model rich relationships between the tables and to perform various complex queries on them. The most significant drawbacks of this approach are the inflexibility of the model and the loss of data. This approach was chosen for entities that are either not part

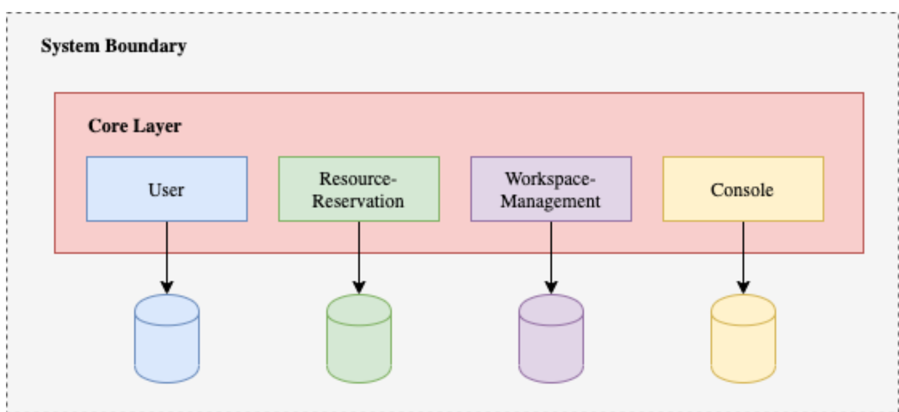


Figure 2.6: Data model separation on module level

of the core domain or were identified as unlikely to change, such as *Console Account*.

2.5.2 Event Sourcing

Another way to store the data is Event Sourcing [29], which, instead of capturing the current state of the system, saves the domain events (1.3.3), leading up to the current state in an append-only event log.

Event Sourcing offers many advantages [30], such as flexibility of the model. Since the state of the system is derived from the domain events (figure 2.7), which tend to change a lot less often than the way they are perceived, the system only needs to change how the system state is derived from the events.

Another great advantage of event sourcing is debuggability. Since all of the events leading up to a given state are persisted, then it is easy to figure out how the system ended up in such a state if a faulty state is encountered.

Event Sourcing comes with drawbacks as well. Since the events are immutable, the system cannot capture user personal or other sensitive information using the events. Also, a careless implementation may result in performance issues.

After careful consideration, event sourcing was chosen as the data model for the core domain entities most likely to change in the future, primarily because of the flexibility it provides, mitigating some of the drawbacks, as described in section 4.5.

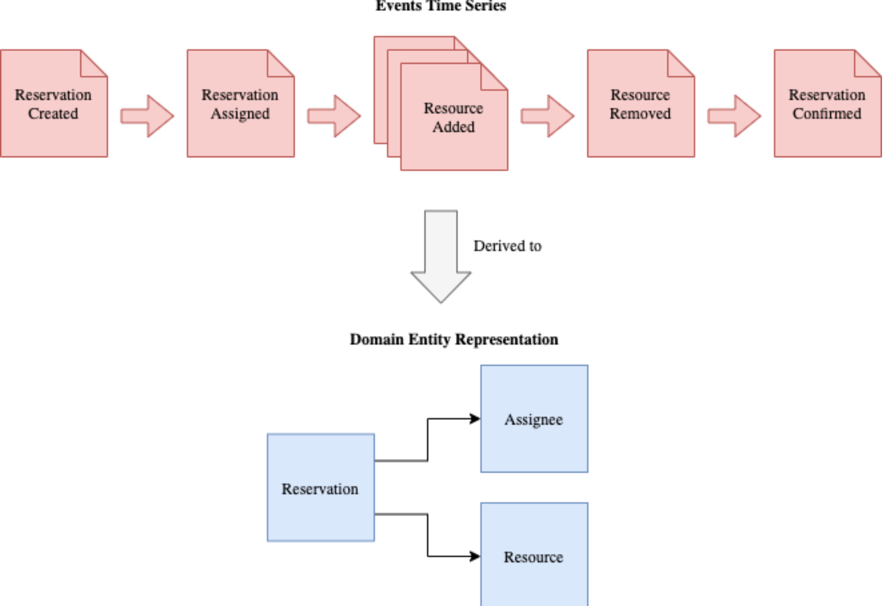


Figure 2.7: Derivation of Reservation from Reservation Events

Chapter 3

Technologies Used

This chapter lists the used technologies and describes why they were chosen based on the requirements (1.2) and the architecture of the system (2).

3.1 Programming Language

There are many languages specialized for programing both server and client-side applications.

Based on the decisions taken in section 2.3, and based on the most popular programming languages at the time of the beginning of the project (summer of 2018) [31], Javascript[2] seemed like the best choice. However, because of its rise in popularity, support of static typing, and interoperability with Javascript, Typescript[3] was chosen for the client implementation.

Because of its lightweight runtime and rise in popularity [31], Node.js[4] was chosen as the target backend platform. Furthermore, using Node.js on backend allows using the same language, Typescript, throughout the entire application stack, lowering the barrier for anyone to be able to contribute or support the project in the future.

Choosing Javascript/Typescript turned out to be the right decision. Javascript remained the most popular programming language, while Typescript gained significant popularity [32] over the past two years.

[2] https://developer.mozilla.org/en-US/docs/Web/javascript
[3] https://www.typescriptlang.org/
[4] https://nodejs.org/en/

3.2 Client Technologies

There is a large variety of front-end libraries/frameworks to choose from, each with their benefits.

At the time of the beginning of the project, the two most popular were Angular/Angular.js and React.js [31]. The most significant advantage of Angular[5] is its native Typescript implementation and the fact that it is a framework. Angular includes end-to-end support for writing client-side applications, from view generation, and state management, to network interactions. However, Angular being a framework, is also its biggest drawback.

React.js[6], on the other hand, is just a library, which means it is not as opinionated as Angular when it comes to structuring the application and choosing other technologies. It provides only a lightweight wrapper around the DOM interactions in a reactive way, without forcing the developer to use any specific technology for the state management or network interactions.

After evaluating both, the React.js was chosen, which proved to be the right decision, because of its flexibility and simplicity. Throughout building the project, it also became the most popular front-end library/framework [32].

3.3 API Gateway

In section 2.4.1, it was decided to use the API Gateway pattern. The first question that needed to be resolved was whether to use off-the-shelf software or whether to build a custom solution for it [33].

When evaluated the benefits and drawbacks of both approaches, it was decided to use Nginx[7] because it would mean less code to maintain and because Nginx turned out to be the most straightforward off-the-shelf piece of software to configure and set up compared to alternatives, such as httpd[8].

3.4 Dependency Injection

The primary goal for putting a dependency injection mechanism in place was achieving requirements **NFR01**, and **NFR02**, while not sacrificing the clarity of the implementation. Libraries/frameworks considered were: TSyringe[9],

[5] `https://angular.io/`
[6] `https://reactjs.org/`
[7] `https://nginx.org/`
[8] `https://httpd.apache.org/`
[9] `https://github.com/Microsoft/tsyringe`

injection-js[10], and Nest.js[11].

All of the libraries use decorators to support dependency injection. After developing a spike solution [34] using each method, Nest.js was chosen because of its simplicity of use and the quality of documentation. The added benefit of using the Nest.js turned out to be an integrated web framework it provides built on top of the dependency injection system.

3.5 Data Storage

The system uses MySQL[12], because of the constraint **CS02**.

3.6 Database Integration

Typeorm[13] library was chosen for the database integration, because of Nest.js' native support[14]. Also, because of its decorator-style integration, it fits in much better with the rest of the system compared to other libraries, such as Sequelize[15].

3.7 Email Integration

The only mature library for email sending that was found and eventually used was Nodemailer[16].

3.8 Continuous Integration

Continuous integration of the system is required by **NFR03**. For its implementation is used Gitlab CI/CD[17], because it is the only continuous integration tool available as part of Silicon Hill infrastructure.

3.9 Containerization

It was decided to containerize pieces of the application using Docker[18] to achieve easy deployment required by **NFR04**,

[10] `https://github.com/mgechev/injection-js`
[11] `https://nestjs.com/`
[12] `https://www.mysql.com/`
[13] `https://typeorm.io/`
[14] `https://docs.nestjs.com/techniques/database`
[15] `https://sequelize.org/`
[16] `https://nodemailer.com/about/`
[17] `https://docs.gitlab.com/ee/ci/`
[18] `https://www.docker.com/`

This proved to be very beneficial from the very beginning. Using Docker, it is possible to run the app in the production environment even locally.

It also allowed moving from hosting the system on a single virtual machine to a VIC cluster[19]. This change was required because of the problem with Silicon Hill infrastructure, that corrupted the virtual machine the application was running on.

[19] https://vmware.github.io/vic-product/

Chapter 4

Implementation

This chapter describes the most important aspects and features of the system.

4.1 Workspace Isolation

A workspace represents an instance of the reservation system. Since each user of the system may have accounts, each with a different role (1.1) assigned, in multiple systems at once, the system needs to isolate the workspaces.

The simplest way to achieve such isolation would be for the system to be single-tenant [35]. The single-tenancy is however not feasible, since it requires high infrastructure overhead. Therefore, the system is multi-tenant.

4.1.1 Entity Isolation

Each entity associated with a workspace is identified not only by its identifier but also by the workspace id it belongs to. For example, *User* entity has two primary identifiers, *UserId* and *WorkspaceId* (figure 4.1).

4.1.2 Credentials Isolation

Further isolation level is achieved by scoping authorization credentials (4.2.4) to a single workspace. Therefore if a user wishes to access multiple workspaces,

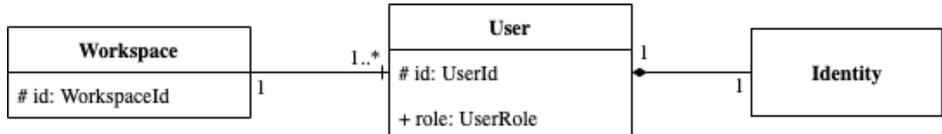


Figure 4.1: User Entity Relation with Workspace

they need to use multiple credentials, each scoped to a single workspace.

4.1.3 Domain Isolation

The third level of isolation is achieved by assigning each of the workspace client application (2.3.1), a unique domain. This is accomplished by pre-pending the root domain the workspace is hosted on, with the name of the workspace[20]. For example, a workspace with a name *test* would be accessible on domain *test.rs.sh.cvut.cz*.

4.2 Authentication and Authorization

This section describes authentication and authorization mechanisms the system uses to ensure proper user access to its resources.

4.2.1 Authentication

Before granting system access authorization, the system requires each user to authenticate. There are multiple methods of authentication based on the authorization requested.

Opening a Workspace Request When a user wishes to open a request for a new workspace, they are required to verify their email address. This is done either by sending a verification email (4.2.4) or using Silicon Hill Information System (4.2.5). After user email is verified, they are granted authorization to open a workspace request.

Logging in to a Workspace Before authorizing user access to a workspace, they need to authenticate with either, valid email and password combination associated with their workspace account, or by authenticating themselves against Silicon Hill Information System.

Logging in to Console Application Each System Administrator has an account with a unique username and password combination, which they need to authenticate with in order to gain authorization to access the Console Application.

4.2.2 Authorization

After a user authenticates themselves, they are, depending on their intent, granted a specific authorization token. There are three types of authorization token users can request.

[20]This does not break the constraint **CS05**, since each of the workspaces is just a subdomain of the root domain *rs.sh.cvut.cz*.

Workspace Access Token This token authorizes user to access a specific workspace and is grated to a user after logging in to a workspace with their workspace account.

Console Access Token This token authorizes user to perform system administration and is granted to a user after logging in to a system using their console account.

Email Verification Token This token authorizes user to open a workspace request and is grated to a user by sending it in an email to their email address.

4.2.3 Authorization Tokens

There are two typical ways to implement authorization in web systems: session cookies, and JSON web tokens (JWTs) [36]. Both have their own benefits. However, the most significant deciding factor that ruled in favor of using JWTs, was that since JWTs are self-contained, they allow the server to remain stateless.

When the authentication process completes, the system sends the client a pair of access and refresh token (listing 4.1).

Access Token A short-lived token (five minutes) that is used to authorize client requests.

Refresh Token A long-lived token (one day) is used to request a new access token without needing to re-authenticate the user.

Client applications store these tokens in the browser's local storage [37] and attach access token to each request it makes.

```
{
  "access_token": <access token JWT>,
  "expires_in": 3600,
  "token_type": "Bearer",
  "refresh_token": <refresh token JWT>,
  "scope": "",
  "workspace": "test"
}
```

Listing 4.1: Authentication Response for workspace named `test`

Each JWT holds information about what kind of token it is, its expiration, and the user it was issued for (listing 4.2).

```
{
  "jit": <opaque token value>,
  "sub": <user id>,
  "exp": 1588599277,
  "iat": 1588595677,
  "scope": [],
  "kind": "access_token",
  "user": {
    "id": <user id>,
    "workspace": "test",
    "role": "admin",
    "isOwner": true,
  },
}
```

Listing 4.2: Access Token JWT for Workspace `test`

4.2.4 Email Verification

User email is verified either using Silicon Hill's OAuth2.0 API (4.2.5), or by sending a verification email to user's email address. The email contains a URL link that contains user's email address, expiration time, unique workspace id, and HMAC signature [38] of the URL that is appended as query parameters. Once user navigates to the link, they can open the workspace request authenticating themselves with the email address, sent workspace id, expiration date and signature,

4.2.5 Integration with Silicon Hill OAuth2.0

The system leverages Silicon Hill's OAuth2.0 [39] API for user authentication to allow Silicon Hill members easier access. The entire authentication flow is captured in the diagram in figure 4.2.

The implemented flow slightly differs from a typical one. There is an extra step that resolves the redirect URI based on the workspace the user is trying to log in, which is achieved by storing workspace information in the `state` parameter [40].

If it is the user's first time logging in to a given workspace, then a workspace user account with the information fetched from the Silicon Hill is created before the authorization token pair is sent to the user.

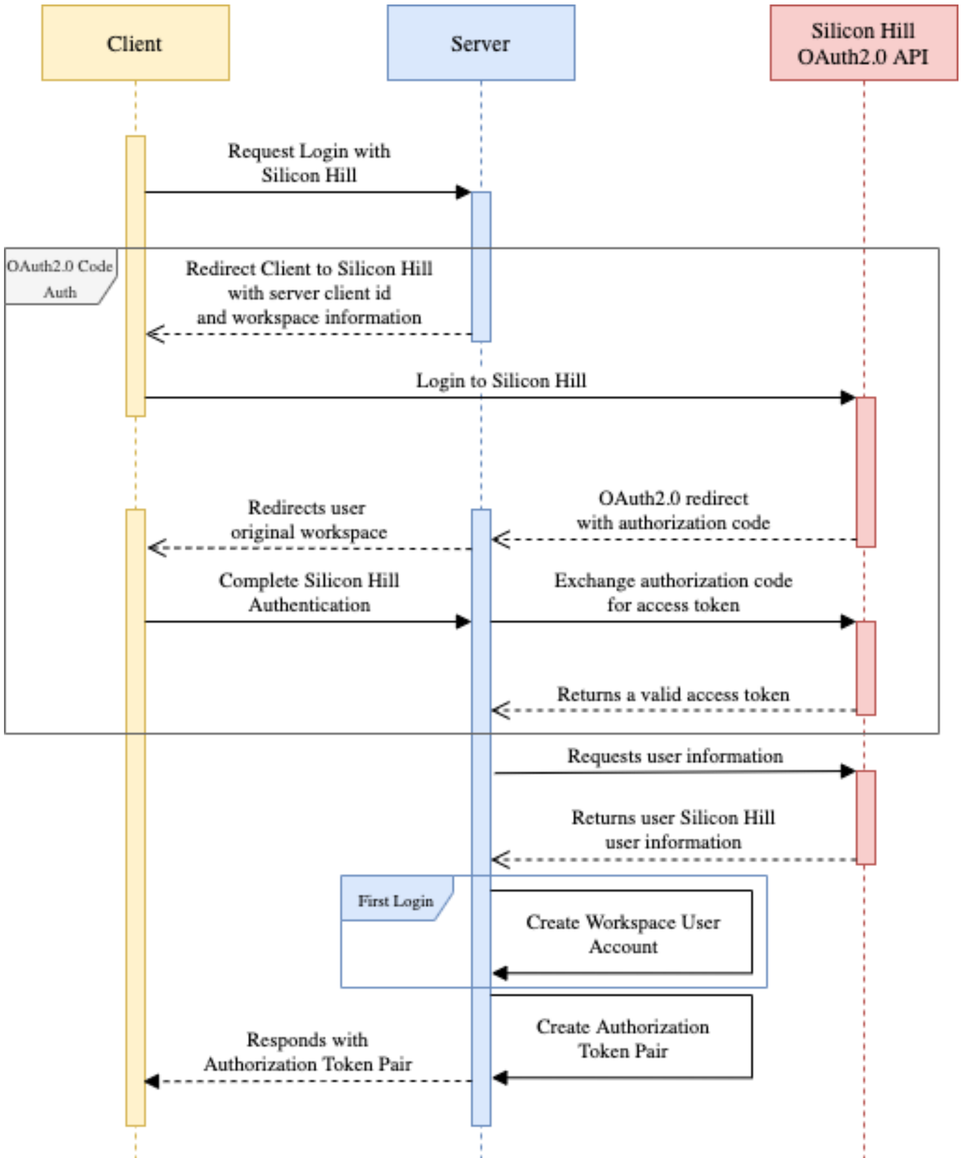


Figure 4.2: Silicon Hill OAuth2.0 Authentication

4.3 Personal Data and Anonymization

Use of personal data is essential for the system. As mentioned in section 1.3.1, each *Workspace User* entity is assigned an *Identity*, comprised of multiple personal information, such as email, first and last name, and phone number[21].

4.3.1 Personal Data Storage

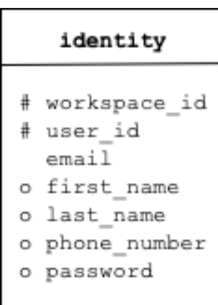
None of the personal information is stored in events that are immutable to ensure this data can be destroyed. Instead, each identity is mapped to a row in the `identity` table in the database (figure 4.3).

4.3.2 Data Anonymization

Since domain-specific user information, such as user role, is stored in events, the system cannot actually delete the user. A user is deleted once a user deleted event (A.2.6) is recorded in the event store (4.5.4). Therefore, personal user data needs to be destroyed in other way. To maintain user entity link to its identity, and, at the same time, compliance with GDPR [41], the personal data is anonymized.

Anonymization is done by encrypting personal information with a randomly generated, 256-bit encryption key, using an AES cipher [42], and throwing away the encryption key. This way, since there is no way to recover the used encryption key, all of the data is effectively destroyed. Yet, the system maintains information about what kind of information was associated with each user identity.

[21] Email is the only piece of personal information, that is required. First and last name, password, and phone number, can be optionally filled by either the user, or are automatically filled in if provided by the Silicon Hill OAuth2.0 API.



`#` primary column
`o` nullable column

Figure 4.3: Identity Table

4.4 REST API

To suffice requirement **NFR17**, as well as, to provide a unified interface for simple client interoperability, the server exposes a REST API using JSON. The API is built around resources and follows basic REST principles [43].

Similarly to the separation of client applications, the REST API is split into three groups: console, setup, and workspace.

4.4.1 Authorization

Depending on the required authorization method and the requested API, each request must include an authorization token (4.2.2). The setup API requires an email verification token to be included as part of the request in order to open a workspace request (4.2.4). Instead, the console and workspace APIs expect an access token to be present in the request. The system has relaxed rules, as for how the access token is included in the request. Users can attach the access token in the `Authorization` header of request (`Bearer <token>`), or include it in the query parameters (`access_token=<token>`), or even in the body of the request (`"access_token": <token>`).

The API routes described in the following subsections annotated by an asterisk symbol (*) require such authorization.

4.4.2 API Versioning

For future backward compatibility, all API routes are prefixed with a `/api/v1` to ensure that any new major API update would require users to just switch the route prefix, e.g., `/api/v2`.

4.4.3 Console API

These routes are prefixed with `/console`, and all relate to system administration. Therefore, the client application uses those API routes primarily. Accessibility of these routes is predicated on console account authentication.

4.4.3.1 Authentication

To retrieving an access token for the console API, a user needs to authenticate themselves with console account credentials. The authentication response of the server is a pair of authorization tokens (4.2.3).

4.4.3.2 Authentication API Routes

These routes expose functionality for obtaining console account authentication and registration, and authorization token management.

- `POST /console/login` This route is used for users to log in with their console account username and password credentials passed in the body of the request. If this request succeeds, the API responds with an authorization token pair and HTTP status code 200, If the username or password do not match any of the registered accounts, the API responds with status code 400.

- `POST /console/register`* This route is used to create a new console account with the given pair of username and password. If the creation of the new console account succeeds, the API responds with a response code 201. If an error occurs, such as username is already taken, the API responds with status code 400.

- `POST /console/token/refresh` This route is used to request a new access token using a refresh token. If the refresh token is valid, the API responds with status code 200, attaching an authorization token pair in the response body. If a refresh token is invalid (corrupted, revoked, or expired), the API responds with status code 400.

- `POST /console/token/revoke` This route is used to revoke an existing refresh token provided in the request. If the refresh token is valid, the API responds with status code 204 and an empty response body. If a refresh token is invalid (corrupted, revoked, or expired), the API responds with status code 400.

4.4.3.3 Console Account API Routes

These routes expose functionality for the management of console accounts.

- `GET /console/account`* This route is used for listing all existing console accounts in the system. The API responds with status code 200, including all console accounts in its body.

- `DELETE /console/account/{username}`* This method is used for deleting existing console accounts by their username included in the API route as a parameter. If a console account with the username exists, the API responds with status code 204 and an empty body. If a user tries to delete their own account, which they are logged in as, the API responds with status code 403. If a provided username is not associated with any console account, the API responds with status code 404.

4.4.3.4 Workspace API Routes

These routes expose functionality for the listing of existing workspaces and their ownership transfer.

- `GET /console/workspace`* This route is used for listing all existing workspaces. The API responds with status code 200, including all workspaces in its body.

- `POST /console/{workspace_id}/transfer`* This method is used for transferring ownership of the workspace to a different workspace user, whose id is expected to be present in the body of the request. If the user exists, the API responds with status code 200, attaching the updated workspace in its body. If the id provided in the request is not associated with any user in the given workspace, or if the workspace id is not associated with any workspace, the API responds with status code 404.

- `GET /console/{workspace_id}/user`* This method is used for listing workspace users of the workspace, which id is passed as route parameter. If the workspace exists, the API responds with status code 200, attaching its users in the body of the response. If the workspace does not exist, the API response with status code 404.

4.4.3.5 Workspace Request API Routes

These routes expose functionality for the listing of existing workspace requests and their confirmation and decline.

- `GET /console/workspace/request`* This route is used for listing all existing workspace requests. The API responds with status code 200, including all workspace requests in its body.

- `POST /console/request/{request_id}/confirm`* This route is used for confirmation of workspace requests. The API responds with status code 200, attaching the updated workspace request in its body. If the workspace request has already been closed[22], the API responds with status code 400. If the id does not correspond to any existing workspace request, the API responds with status code 404.

- `POST /console/request/{request_id}/decline`* This route is used for declining of workspace requests. The API responds with status code 200, attaching the updated workspace request in its body. If the workspace request has already been closed, the API responds with status code 400. If the id does not correspond to any existing workspace request, the API responds with status code 404.

[22] The workspace request is closed, if it is confirmed or declined

4.4.4 Setup API

These routes are prefixed with `/setup`, and all relate to the opening of workspace requests. They form the primary consumer of this API is the landing application.

4.4.4.1 Email Verification API Routes

These routes expose functionality related to the verification of the user's email address (4.2.4).

- `POST /setup/verify/email` This route is used for verifying user's email address passed in the body of the request using a verification email. The API responds with status code 202 and an empty response body.
- `GET /setup/verify/authority/{authority}` This route is used for verifying user's email address passed in the body of the request using an external OAuth2.0 compliant authority[23]. The API expects the authority identifier to be present as the router parameter, and the initial state is passed as the request's query parameter. The API responds with status code 301, redirecting the client to complete the OAuth2.0 authentication with the external authority. If the referenced authority does not exist, the API responds with status code 404.

4.4.4.2 Workspace Request Completion API Routes

These routes expose functionality for completing a setup process of opening a workspace request.

- `POST /setup/complete/email`* This route is used for opening a request for a workspace with an email address verified using a verification email. The API responds with status code 201, attaching the opened workspace request in its body. If verification information provided is invalid, the API responds with status code 401. If the name of the workspace is either invalid or collides with an already existing one, the API responds with status code 400.
- `POST /setup/complete/authority`* This route is used for opening a workspace request using the authority verification process. The API responds with status code 201, attaching the created workspace in its body. If verification completes with an error, the API responds with status code 401. If the name of the workspace is either invalid or collides with an already existing one, the API responds with status code 400.

[23] Currently the only authority implemented is Silicon Hill with authority identifier `silicon_hill`.

4.4.5 Workspace API

These are the rest of the API routes and present the primary way of interacting with a specific workspace. Therefore, the workspace application is the primary consumer of these routes. The access to majority of these routes is predicated on workspace account authentication.

4.4.5.1 Authentication API Routes

These routes are used to authenticate users with the system using external authorities, as well as to set up and/or change users' passwords.

- `GET /login/{authority}` This route is used for requesting the user authentication using an external OAuth2.0 compliant authority[24]. The API expects the authority identifier to be present as the router parameter and the initial state passed as the query parameter of the request. The API responds with status code 301, redirecting the client to complete the OAuth2.0 authentication with the authority. If, however, there's no authority associated with the given authority identifier, the API returns status code 404.
- `GET /auth/{authority}` External OAuth2.0 compliant authorities use this route as the redirect URI. Each supported authority is registered a version of this route with its specific authority identifier. This redirects clients to the URI encoded in the `state` query parameter of the request.
- `POST /login/authority` This route is used to complete the login using an external authority expecting OAuth2.0 authorization code, as well as authority identifier and the `state` passed in the body of the request. The API responds with status code 200 and a pair of authorization tokens (4.2.3), creating a workspace user account if it is the first time for the user logging in the given workspace. If the user is banned, the API responds with status code 400. If the workspace the user is trying to log in, or authority associated with the given identifier does not exist, the API responds with status code 404.
- `POST /password`* This route is used for setting a new account password, passed in the body of the request. The API responds with status code 204 and an empty response body.

4.4.5.2 OAuth2.0 API Routes

The system implements a subset of OAuth2.0 [39], namely the `password` and `refresh_token` grant types. The API also allows for refresh token revocation.

[24]Currently the only authority implemented is Silicon Hill with authority identifier `silicon_hill`.

- `POST /oauth2/token` This is a standard OAuth2.0 request except for the `password` grant type, which in addition to `username` and `password` values also expects a workspace name passed as `workspace`. The API responds with status code 200 and an authorization token pair in the response body. If the request is invalid, the API responds with status code 400. If an unsupported grant type is requested, the API responds with status code 501. In case of the `password` grant type, if a workspace name is invalid, the workspace with the given name does not exist, the user is banned, or either username or password do not match, the API responds with status code 400, as well. In case of `refresh_token` grant type, if given refresh token is invalid, the API responds with status code 400, too.
- `POST /oauth2/token/revoke`* This route is used for refresh token revocation. The API responds with status code 204 and an empty body. In case of an invalid refresh token being provided, the API responds with status code 400.

4.4.5.3 Workspace API Routes

These routes expose functionality for the listing of existing workspaces, their ownership transfer, and deletion.

- `GET /workspace` This route is used for the listing of workspaces. The API responds with status code 200 and a list of existing workspaces in its body. The route also supports the addition of a `name` query parameter, in which case the API returns just the workspace with the given name. If such workspace does not exist, the API responds with status code 404.
- `POST /workspace/transfer`* This route is used by workspace owners to transfer ownership of the workspace to a different user. The API responds with status code 204 and an empty response body. If a user trying to transfer the ownership of the workspace is not its owner, the API responds with status code 401. If the user the workspace is being transferred to does not exist, the API responds with status code 404.
- `DELETE /workspace`* This route is used for deleting the workspaces. The API responds with status code 200 and an updated workspace in its body. If a user tries to delete the workspace is not its owner, the API responds with status code 401.

4.4.5.4 User API Routes

These routes expose functionality for listing workspace users, changing their permissions, and user deletion.

- `GET /user`* This route is accessible only to workspace maintainers, admins, and the workspace owner. The API responds with status code 200, attaching a list of users in the body of the response.
- `GET /user/me`* This route responds with status code 200, attaching the currently logged in user in its body.
- `DELETE /user/me`* This route is used for deleting the user's account. The API responds with status code 204 and an empty response body. If a workspace owner tries to delete themselves, the API responds with status code 403.
- `PUT /user/me/profile`* This route updates user's profile to a new one, passed in the body of the request. The API responds with status code 200, attaching the updated user in its body.
- `GET /user/{id}`* This route returns users referenced by their id, that is passed as the route parameter. This route is accessible only to workspace maintainers, admins, and the workspace owner. The API responds with status code 200, attaching referenced user in its body. If the referenced user does not exist, the API responds with status code 404.
- `PATCH /user/{id}/role`* This route sets referenced user's role. This route is accessible only to workspace maintainers, admins, and the workspace owner. The API returns status code 200, attaching updated user in its body. If a requester tries to demote a user with higher privileges, or tries to grant higher privileges than their own, the API responds with status code 401. If a requester tries to change their own role, the API responds with status code 403. If the referenced user does not exist, the API responds with status code 404.
- `PATCH /user/{id}/ban`* This route toggles referenced the user's ban. This route is accessible only to workspace maintainers, admins, and the workspace owner. The API returns status code 200, attaching updated user in its body. If a requester tries to toggle the ban of a user with higher privileges, the API responds with status code 401. If a requester tries to toggle their own ban status, the API responds with status code 403. If the referenced user does not exist, the API responds with status code 404.

4.4.5.5 Resource API Routes

These routes expose functionality for listing and management of resources.

- `GET /resource`* This route lists all resources in the workspace responding with status code 200.

- `GET /resource/{id}`* This route is used to retrieve a resource referenced by its id passed as the route parameter. The API responds with status code 200, attaching the resource in its body. If the referenced resource does not exist, the API responds with status code 404.
- `POST /resource`* This route is accessible only to workspace maintainers, admins, and the workspace owner, and it is used to create new resources with given name and description passed in the body of the request. The API responds with status code 201, attaching the created resource in its body.
- `PUT /resource/{id}`* This route is accessible only to workspace maintainers, admins, and the workspace owner, and it is used to update the resource with a given name and description passed in the body of the request. The API responds with status code 200, attaching the updated resource in its body. If the referenced resource does not exist, the API responds with status code 404.
- `DELETE /resource/{id}`* This route is accessible only to workspace maintainers, admins, and the workspace owner, and it is used to delete the referenced resource. The API responds with status code 204 and an empty body. If the referenced resource does not exist, the API responds with status code 404.
- `PATCH /resource/{id}/toggle`* This route is accessible only to workspace maintainers, admins, and the workspace owner, and it is used to toggle the active status of the referenced resource. The API responds with status code 200, attaching the updated resource in its body. If the referenced resource does not exist, the API responds with status code 404.

4.4.5.6 Reservation API Routes

These routes expose functionality to create and manage reservations.

- `GET /reservation`* This route lists all reservations in the workspace responding with status code 200. The route also provides options for filtering based on the reservation time period. A requester can provide `date_start`, `date_end`, or both dates as query parameters forming a interval. If filtering parameters are provided, the API responds only with reservations, which intervals intersect with the one formed by the query parameters.
- `POST /reservation`* This route is used for creating reservations from the properties passed in the body of the request. The API responds

with status code 201, attaching the created reservation in its body. If a workspace user tries to create a reservation not assigned to them, the API responds with status code 401. If an invalid date range is selected, such as reservation starting in the past or ending before it starts, the API responds with status code 400.

- `GET /reservation/{id}`* This route is used for querying reservations by their id, which is passed as the route parameter in the request. The API responds with status code 200, attaching referenced reservation in its body. If the referenced reservation does not exist, the API responds with status code 404.

- `PUT /reservation/{id}`* This route is used for updating reservations from the properties passed in the body of the request. The API responds with status code 200, attaching updated reservation in its body. If a workspace user tries to reassign a reservation to someone else or try to update the reservation not assigned to them, the API responds with status code 401. If an invalid date range is selected, such as reservation starting in the past or ending before it starts, the API responds with status code 400. If the reservation has already been canceled, rejected, or completed, the API responds with status code 403 (1.3.5).

- `GET /reservation/{id}/events`* This route is used for querying events associated with the referenced reservation by its id, which is passed as the route parameter in the request. The API responds with status code 200 attaching a list of referenced reservation events, ordered in ascending order by their event id, in its body. If the referenced reservation does not exist, the API responds with status code 404.

- `PATCH /reservation/{id}/confirm`* This route is accessible only to workspace maintainers, admins, and the workspace owner, and it is used to confirm a reservation referenced by the route parameter. The API responds with status code 200, attaching updated reservation in its body. If one or more resources are unavailable in the requested time range, it begins in the past, or the confirmation of the reservation would cause an invalid state transition (1.3.5), then the API responds with status code 403. If the referenced reservation does not exist, the API responds with status code 404.

- `PATCH /reservation/{id}/cancel`* This route is used to cancel a reservation referenced by the route parameter. The API responds with status code 200, attaching the updated reservation in its body. If a workspace user tries to cancel a reservation that is assigned to someone else, the API responds with status code 401. If the cancelation of the reservation would cause an invalid state transition (1.3.5) or the reservation starts

in the past, the API responds with status code 403. If the referenced reservation does not exist, the API responds with status code 404.

- `PATCH /reservation/{id}/reject`* This route is accessible only to workspace maintainers, admins, and the workspace owner, and it is used to reject a reservation referenced by the route parameter. The API responds with status code 200, attaching updated reservation in its body. If the rejection of the reservation would cause an invalid state transition (1.3.5) or the reservation starts in the past, the API responds with status code 403. If the referenced reservation does not exist, the API responds with status code 404.

4.4.5.7 Reservation Comment API Routes

These routes expose functionality for adding comments to reservations, their editing, and deletion.

- `GET /reservation/{reservation_id}/comment`* This route lists comments attached to the reservation referenced by the route parameter, responding with status code 200. The route is accessible only to workspace maintainers, admins, and the workspace owners. If the reservation does not exist, the API responds with status code 404.

- `POST /reservation/{reservation_id}/comment`* This route is accessible only to workspace maintainers, admins, and the workspace owners and it creates a new comment from the properties attached in the request body for the referenced reservation. The API responds with status code 201, attaching the created reservation comment in its body. If the reservation does not exist, the API responds with status code 404.

- `PUT /reservation/{reservation_id}/comment/{id}`* Only to workspace maintainers, admins, and the workspace owners can access this route, which is used to update referenced comment by the route parameter from the properties attached in the request body. The API responds with status code 200, attaching the updated reservation comment in its body. If the user trying to edit the referenced comment is not its author, the API responds with status code 401. If the referenced comment or the reservation does not exist, the API responds with status code 404.

- `DELETE /reservation/{reservation_id}/comment/{id}`* This route is accessible only to workspace maintainers, admins, and the workspace owners and it is used to delete referenced comment by the route parameter. The API responds with status code 204 and an empty response

body. If the user trying to delete the referenced comment is not its author, the API responds with status code 401. If the referenced comment or the reservation does not exist, the API responds with status code 404.

4.4.5.8 Webhook API Routes

These routes expose functionality for configuring workspace webhooks.

- `GET /webhook`* This route is accessible only to workspace admins and the workspace owner, and it is used for the listing of webhooks. The API responds with status code 200, attaching all webhooks in its body.

- `POST /webhook`* This route is accessible only to workspace admins and the workspace owner, and it is used for creating webhooks from the properties attached in the body of the request. The API responds with status code 201, attaching the created webhook in its body.

- `GET /webhook/{id}`* This route is accessible only to workspace admins and the workspace owner, and it is used for querying webhooks by their id, that is present as the request route parameter. The API responds with status code 200, attaching referenced webhook in its body. If the referenced webhook does not exist, the API responds with status code 404.

- `PUT /webhook/{id}`* This route is accessible only to workspace admins and the workspace owner, and it is used for updating webhooks by their id, that is present as the request route parameter, using the properties from the body of the request. The API responds with status code 200, attaching the updated webhook in its body. If the referenced webhook does not exist, the API responds with status code 404.

- `DELETE /webhook/{id}`* This route is accessible only to workspace admins and the workspace owner, and it is used for deleting webhooks by their id, that is present as the request route parameter. The API responds with status code 204 with an empty response body. If the referenced webhook does not exist, the API responds with status code 404.

- `POST /webhook/{id}/test`* This route is accessible only to workspace admins and the workspace owner, and it is used for testing of the configured webhook referenced the by its id, that is present as the request route parameter. The API responds with status code 200, attaching response from the remote server in its body. If the referenced webhook does not exist, the API responds with status code 404.

- `PATCH /webhook/{id}/toggle`* This route is accessible only to workspace admins and the workspace owner, and it is used for toggling activity of the webhook referenced the by its id, that is present as the request route parameter. The API responds with status code 200, attaching updated webhook in its body. If the referenced webhook does not exist, the API responds with status code 404.

4.5 Event Sourcing

As it was mentioned in subsection 2.4.4, the system uses event sourcing to store the vast majority of the core domain data.

There are many ways the event sourcing could be implemented [29][44]. The one implemented, that fit the overall design of our system, was highly inspired by the one presented by Phillipa Avery, and Robert Reta, in their presentation [45] on implementing Downloads feature in Netflix.

4.5.1 Overview

The event sourcing adopted by RSaaS is built around aggregates. Aggregate is an entity that can be decomposed into a sequence of events (1.3.3). For each of them, there is an aggregate service that holds business logic associated with the aggregate. Each aggregate service has an aggregate repository, that is responsible for communication with event store, where the events are stored, applying the events onto aggregates, and executing commands. This flow is captured in the figure 4.4.

4.5.2 Aggregate Service

Aggregate services are implemented as classes following Command Query Separation (CQS) [46] principle. They expose methods for querying the aggregates (listing 4.3), as well as a method for executing commands on them (listing 4.4).

When executing commands, a service can query other services to ensure business invariants. However, it does not perform the commands directly. Instead, services use their associated repositories to execute the commands (figure 4.5).

```
public async getById(
  workspaceId: string,
  reservationId: string,
): Promise<Reservation> { ... }
```

Listing 4.3: Reservation Service Query Method

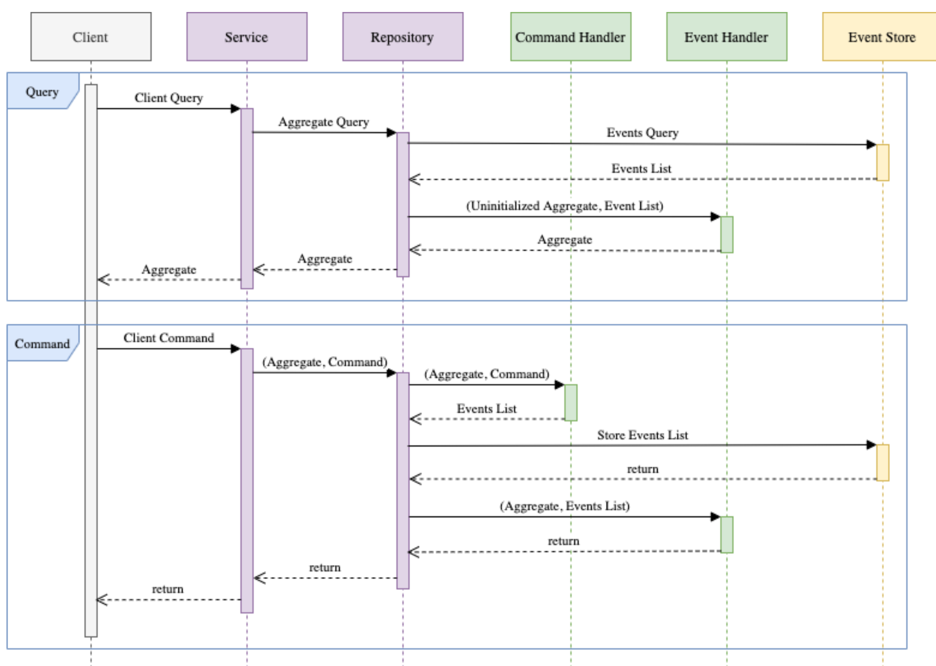


Figure 4.4: Event Sourcing Overview

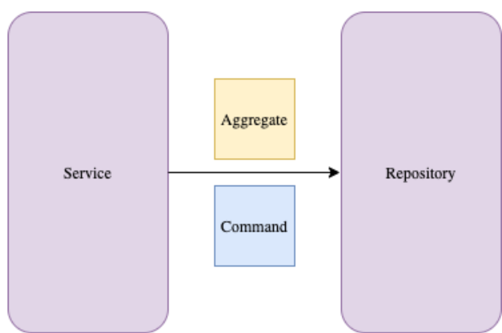


Figure 4.5: Service-Repository Command execution

```
public async confirm(
  reservation: Reservation,
  changedBy: ReservationManager,
): Promise<void> { ... }
```

Listing 4.4: Reservation Service Command Method

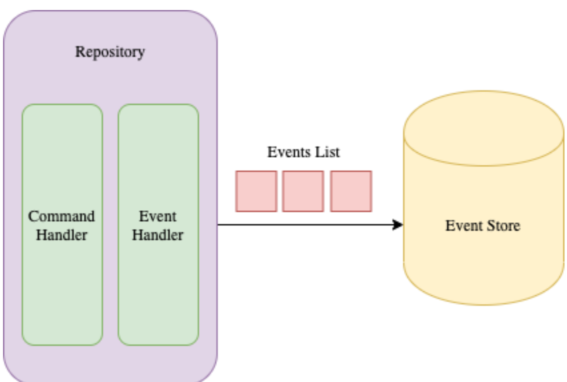


Figure 4.6: Aggregate Repository

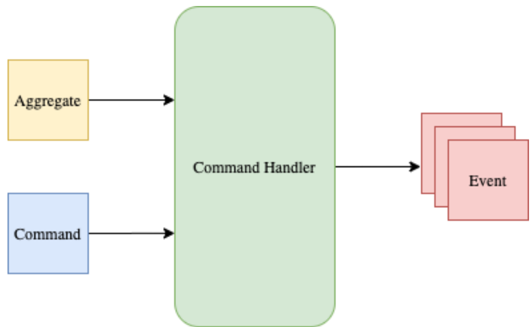


Figure 4.7: Command Handler

4.5.3 Aggregate Repository

Aggregate Repositories (figure 4.6) are implemented as classes, composed of a pair of command (4.5.3.1) and event handler (4.5.3.2). Repositories use these handlers for command execution, as well as applying events onto aggregates.

Each the aggregate repository derives from the base class, that implements behavior for command execution, event application, and event store communication. The derived classes provide a custom implementation of those handlers, specific to the associated aggregate.

4.5.3.1 Command Handler

Command handlers are responsible for executing commands on given aggregates. The output of command handlers is a list of events leading the aggregate to a new, desired state (figure 4.7). If, however, such a state is not possible to achieve (e.g., executing a confirm command on rejected reservation), the handler can reject such command by throwing an exception.

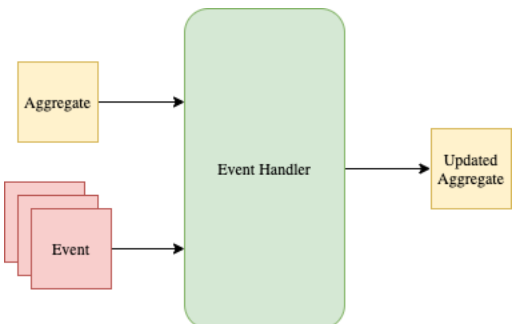


Figure 4.8: Event Handler

Command handlers are implemented as classes, which methods can be registered to handle specific commands (listing 4.5). Each handler class derives from a base class that instruments command execution using class' registered methods.

```
@CommandHandler(ReservationCommandType.create)
public executeCreate(
  aggregate: Reservation,
  command: ReservationCreateCommand,
): ReservationEvent[] { ... }
```

Listing 4.5: Reservation Command Handler Method Registration

4.5.3.2 Event Handler

Event handlers are responsible for applying events to given aggregates, taking a list of events, and aggregate as the input and outputs updated aggregate (figure 4.8). Application of events is an operation that *always* succeeds, resulting in a new aggregate state.

Event handlers, similarly to command handlers, are implemented as classes, which methods can be registered to handle specific events (listing 4.6). Each event handler class derives from a base class, that instruments event application using class' registered methods.

```
@EventHandler(ReservationEventType.created)
public applyCreated(
  aggregate: Reservation,
  event: ReservationCreatedEvent,
): Reservation { ... }
```

Listing 4.6: Reservation Event Handler Method Registration

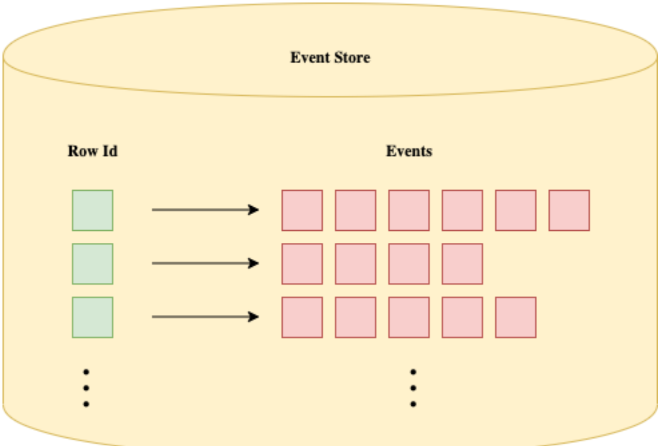


Figure 4.9: Event Store

4.5.4 Event Store

The event store provides an abstraction layer around the storing of the events. The event store forms a key-value store [47] that stores lists of events in rows identified by their *row id* (figure 4.9).

A *row id* is a structured string, formed by an aggregate name, and any number of related identifiers (such as workspace id) separated by colons, that logically groups related events together. Row id can be thought of as sort of an index to a specific group of events. For example, each reservation event is assigned a row id based on the workspace they belong to.

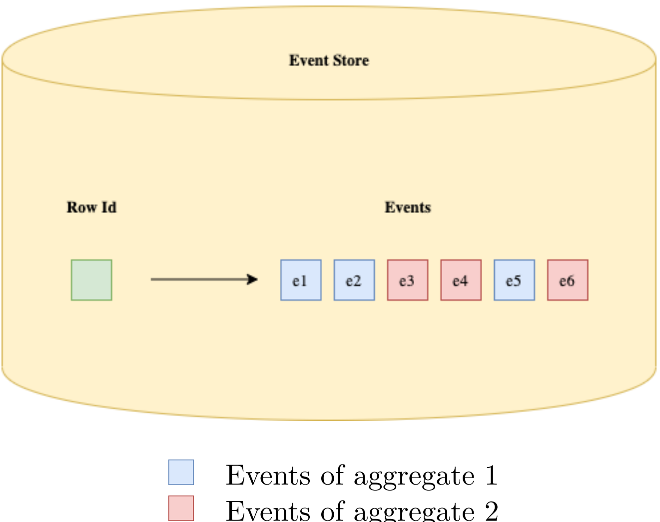
Each row within the event store can contain events related to multiple aggregates. However, each of them must be of the same type[25] (figure 4.10). The event store is append-only which means that events can only be appended at the end of each row, but not updated, nor deleted.

4.5.4.1 Tabular Representation

Since the system uses MySQL as the database, the event store is represented by a single table, the `events_table` (figure 4.11), with the event's properties mapped to table's columns.

Event Id This is the primary key of the table. It is sequentially incremented and assigned to each event upon its persistence in the database. Using this key, we can represent each event in a specific row, as just another row in

[25] This means that one row can store events of multiple reservation aggregates, however it cannot store events of reservations *and* users



Events of aggregate 1
Events of aggregate 2

Figure 4.10: Single Row of Event Store

the `events_table`. The benefit of having a unique identifier assigned to each event is an unambiguous ordering of events.

Row Id This column directly translates to the row id of the event store. This column is indexed and can be used to query the events.

Aggregate Id This column contains the identifier, a string, of the aggregate it belongs to. This column is indexed and can be used to query the events.

Timestamp This column contains the timestamp of event creation. It is indexed and is used to filter the events.

Type This column holds the information about the event type. It is not indexed and therefore not supposed to be queried against by the system. However, it can be used for debugging purposes.

Payload This column holds the JSON-encoded payload of the event. This column is not indexed.

4.5.5 Snapshotting

In section 2.5.2, we raised potential performance issues that poor implementation of event sourcing may cause. Specifically, processing a large number of events to complete a request.

One precaution taken is a grouping of events into rows index by their row id, as we mentioned in the previous section, which significantly decreases the number

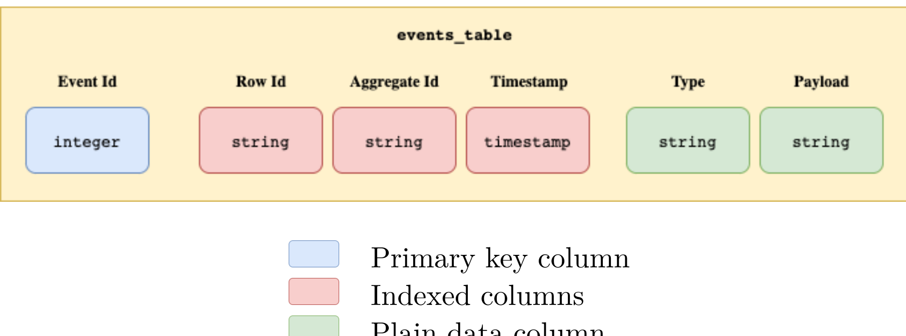


Figure 4.11: Events Table Structure

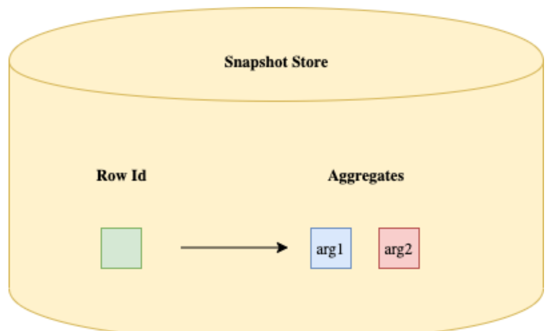


Figure 4.12: Snapshot Store

of events needed to be processed per request. However, if the number of events in a row increases significantly, it may no longer make sense to process every single event that happened since the beginning of the system. The system implements a technique called snapshotting [48] to avoid this situation.

4.5.5.1 Snapshots

A snapshot is a materialized view that encapsulates the entire history until a particular point in time. In the case of our system, a snapshot captures a list of aggregates with the same row id (figure 4.12).

Snapshots are created by an aggregate repositories when certain criteria that can be customized by each repository are met. Once the specific criteria are met, the repository captures the current state of the aggregates in a snapshot, that is serialized and stored in the snapshot store. From this moment onwards, the repository, first checks for the existence of the snapshot, and if it exists, processes only the events that were added after the snapshot was created.

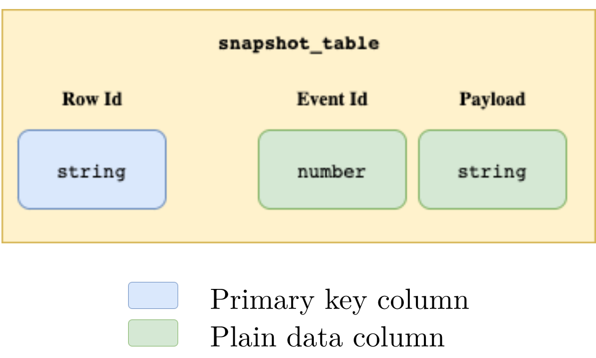


Figure 4.13: Snapshot Table

4.5.5.2 Tabular Representation

The snapshot store, as the event store, is represented by a single table, the `snapshot_table` (figure 4.13), mapping snapshot's properties to the table's columns.

Row Id This is the primary key of the table, representing the key by which the materialized aggregates are stored.

Event Id This column represents the latest event processed that led to the state of the materialized view of the aggregates. This column is not indexed.

Payload This column represents materialized aggregates, which are serialized into an array of JSON objects.

4.6 Message Broker

As it was mentioned in section 2.4.4, the system uses a message broker to realize asynchronous communication between the modules. Multiple off-the-shelf solutions, like Kafka [49] and RabbitMQ [50] were considered. The system even used RabbitMQ for a brief period of time throughout the development of the project. However, since Silicon Hill infrastructure does not include such messaging solution, the burden caused by maintenance and configuration of an external system dependency was the primary deciding factor for implementing a custom, in-memory solution.

4.6.1 Overview

The primary goal of the message broker was to allow event consumers to subscribe for domain events emitted in the system. The key qualities of the

Column name	Description
`event_id`	id of the event that should be published
`event`	serialized event

Table 4.1: Outbox Table

Column name	Description
`consumer_name`	name of the event consumer
`latest_consumed`	id of the latest consumed event

Table 4.2: Inbox Table

message were reliable publishing of those events (4.6.2), as well as, exactly-once delivery of them to the event consumers (4.6.3).

4.6.2 Outbox

The reliable publishing of the events is realized using the outbox pattern [51]. It is implemented using a single database table with columns captured in table 4.1. The publisher of the event is required to insert the entires in the same transaction as they are stored in the event store (4.5.4).

The `outbox` table forms a queue, which an outbox processor polls every 200ms for new entries. If there are some entries in the `outbox` table, the processor reads them ordered by their event id, and dispatches them to the message broker. The message broker then passes each of them to every subscriber. Once all of the subscribers successfully process the entry, the outbox processor removes it from the `outbox` table (figure 4.14).

If, however, the a failure is occurs (figure 4.15), the message is not deleted, and the outbox processor current invocation aborts. This ensures the entries are deleted only if they are successfully processed by the message broker.

4.6.3 Inbox

The system implements a simplified version of the inbox pattern [52], in which each event consumer maintains track of the latest consumed event in the `inbox` table 4.2.

Each event consumer is assigned to a subscriber, that is registered in the message broker. Once the subscriber receives an entry to consume, it checks whether the id of the event is higher than the latest consumed one for the given consumer.

If the value is indeed greater, the event is deserialized and passed to the event

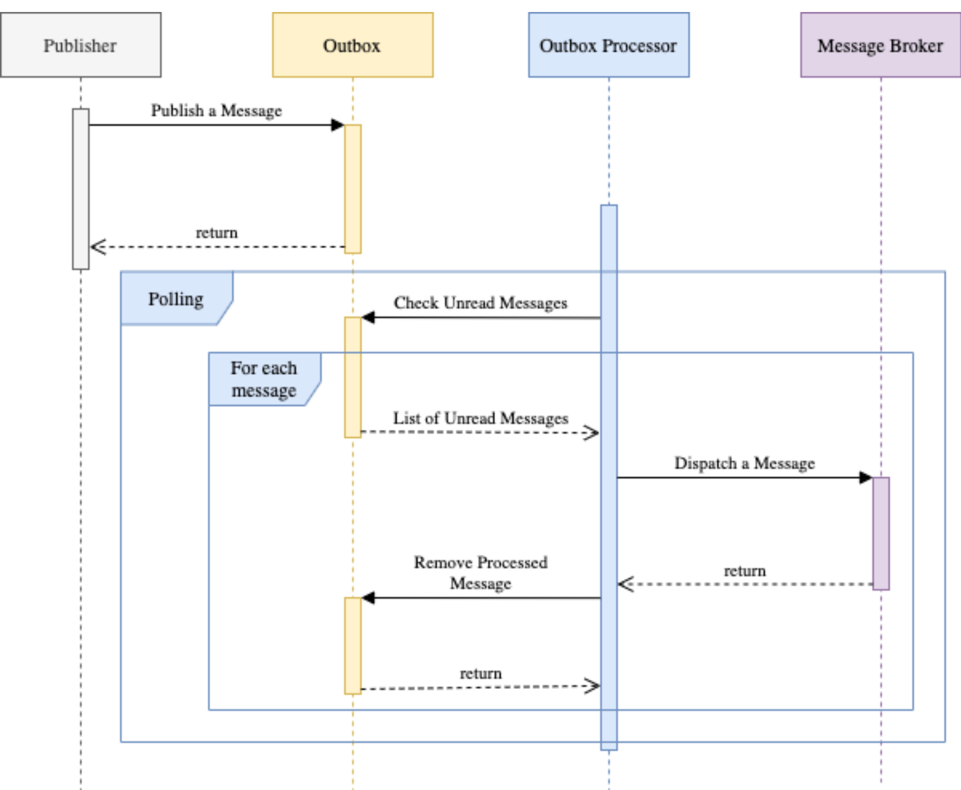


Figure 4.14: Outbox Pattern – Successful Processing of Event

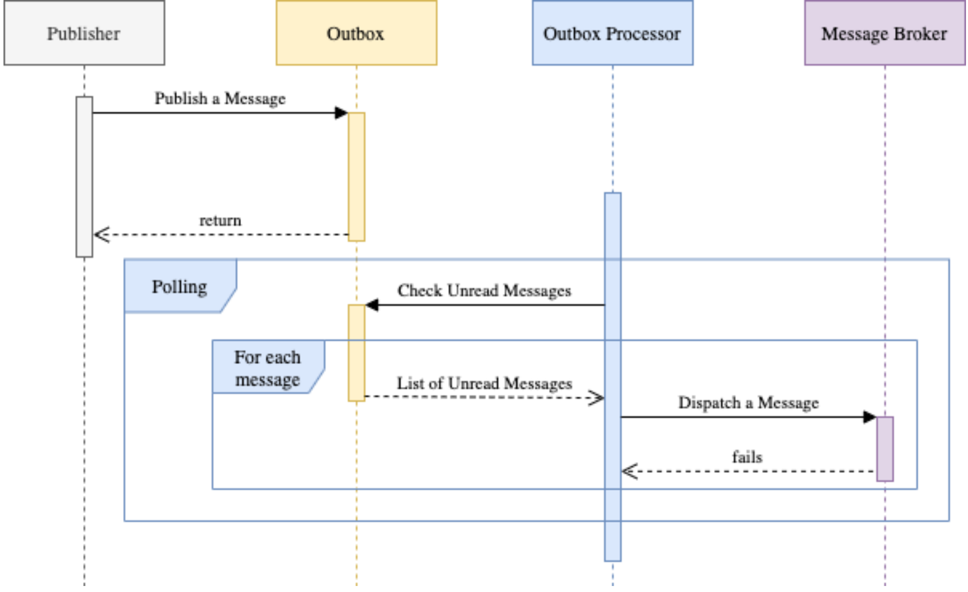


Figure 4.15: Outbox Pattern – Failure Encountered During Event Processing

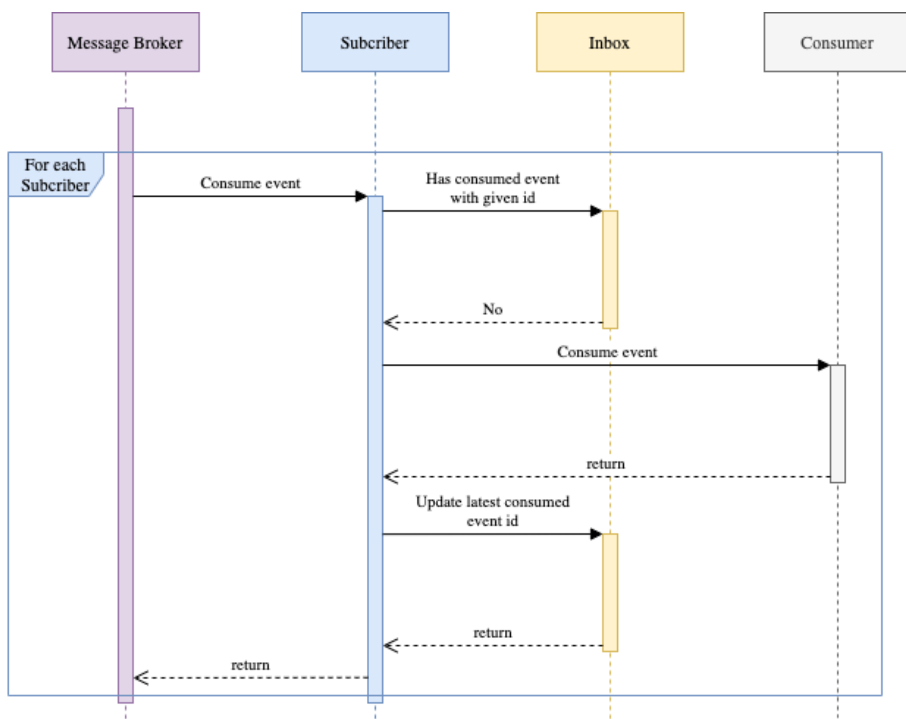


Figure 4.16: Inbox Pattern – First Delivery of Event

consumer (figure 4.16). If the consumer successfully processes the event, the latest consumed event id in the consumer inbox is updated.

If, however, subscriber is handed already processed event, the event is ignored (figure 4.17). This way subscriber ensures each event is consumed only once by the consumer.

4.6.4 Event Consumers

Any class in the system that wishes to consume events need to be annotated with an `EventConsumer` decorator (listing 4.7), which registers the class for event consumption. After the class is annotated, their methods can be annotated with any number of `EventPattern` decorators (listing 4.8), which specify what type of events the methods should process.

```
@EventConsumer()
class WorkspaceRequestConfirmationIngestorService {
  ...
}
```

Listing 4.7: Event Consumer Class Registration

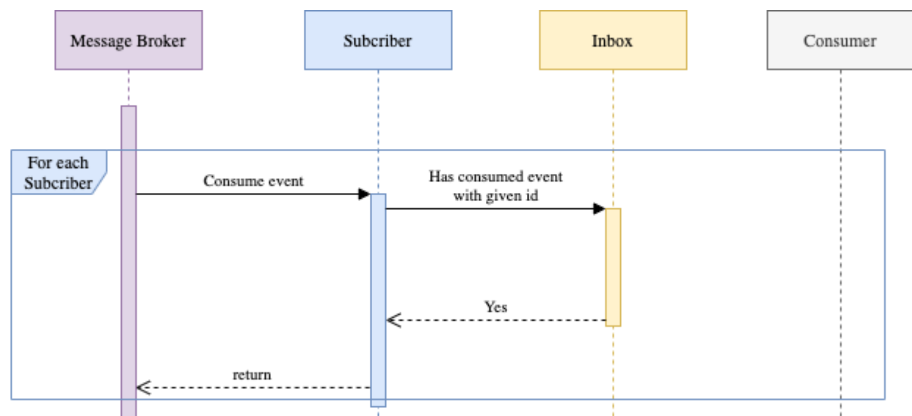


Figure 4.17: Inbox Pattern – Subsequent Delivery of Event

```
@EventPattern(WorkspaceRequestEventType.confirmed)
public async consumeConfirmed(
  event: WorkspaceRequestConfirmedEvent,
): Promise<void> { ... }
```

Listing 4.8: Event Pattern Method Registration

4.7 Webhooks

Webhooks are essentially configurable callbacks to remote services, which can be dispatched when specific events (4.7.1) occur in the system.

4.7.1 Supported Events

Currently, the system supports the following subset of reservation life cycle events (A.2.4):

- Reservation created
- Reservation confirmed
- Reservation rejected
- Reservation canceled

Each webhook can be configured to be dispatched when any one, or more of these events occur in the system. In the future it is planned to extend these events with all of the reservation life cycle events and also include resource and reservation comment events.

4.7.2 Webhook Realization

Webhooks are implemented as remote HTTP POST requests [53]. Each webhook can configure the URL the request should be sent to, and payload encoding type. The webhook dispatcher is implemented as an event consumer. Therefore, it is guaranteed that each event results in all registered webhooks being dispatched. Each webhook request should respond with a success HTTP status code (2xx) [54], otherwise, the request is retried. Each webhook request is retried up to three times with timeouts of 100, 200, and 400ms.

4.7.3 Payload Validation

Each webhook HTTP request also includes an `X-Reserve-Signature` header with an HMAC [38] signature of the request's payload using a secret key, which is part of the each webhook's configuration. This signature can be used to validate the correctness of the payload.

4.7.4 Testing of Webhooks

The system exposes the API for testing the webhooks with arbitrary payloads. A user configuring the webhook can invoke it with a custom payload. The

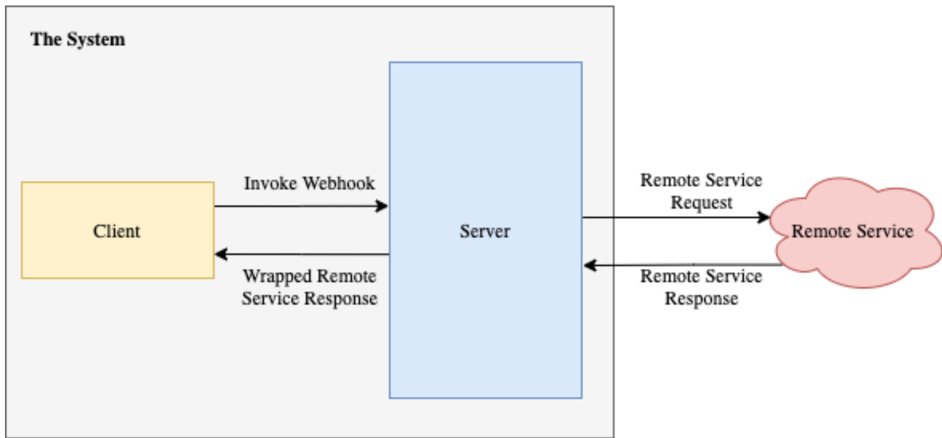


Figure 4.18: Webhook Testing

server then dispatches the webhook with the given payload passing response[26] of the remote server back to the user in the body of the server response (figure 4.18).

4.8 Testing

Testing is an integral part of any development process. There are many methods to test the system. This section describes how parts of the system are tested.

4.8.1 Client Testing

Client applications rely entirely on manual UI testing. This decision was predicated on the implementation and tooling complexity of automated UI testing, which was found ineffective throughout the development. Features provided by client application tools, such as hot reloading [55], provide an efficient way to change and debug the code.

4.8.2 Server Testing

Initially, the first version of the server part of the system was implemented without any automated tests. All testing was done through the UI or the REST API. This decision was made to minimize any unwanted coupling between the implementation and code in the beginning of the project, and to reduce the time to delivery of its initial version.

[26]Included are the response status code, headers and its body.

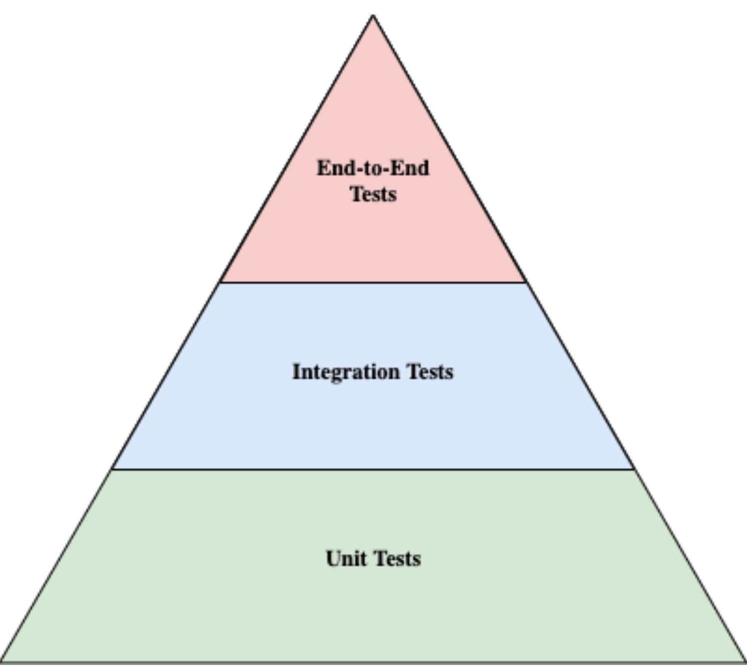


Figure 4.19: Test Pyramid

However, after the the system grew to a certain size, a large number of regressions started to slow down the development significantly. To minimize the regressions and to improve overall code quality, the techniques such as opportunistic refactoring [56], adding tests to refactored pieces of code, and Test-Driven Development (TDD) [57], when adding new features, got employed.

Using these techniques, the system achieved high test coverage, as required by **NFR19**, increased development velocity, as well as, reduced debugging time. Currently, the server tests form a test pyramid (figure 4.19), where there is a large number of unit tests (4.8.2.1), followed by integration tests (4.8.2.2) and a small number of end-to-end tests (4.8.2.3).

4.8.2.1 Unit Tests

Definition: *A unit test is an automated piece of code that invokes the unit of work being tested, and then checks some assumptions about a single end result of that unit. A unit test is almost always written using a unit testing framework. It can be written easily and runs quickly. It's trustworthy, readable, and maintainable. It's consistent in its results as long as production code hasn't changed.* [58]

The system deliberately follows such vague definition, especially around the

'unit of work' under test, because it depends on each test's context. In some cases 'unit of work' refers to a single method or a function, in others to a group of methods or an entire class.

Unit tests focus primarily on testing the business logic of the system, achieving a 100% test coverage of the business logic code. They form the majority of the system tests because they are the cheapest (fastest) to run.

To maintain the clarity of the unit tests, all of them follow Arrange-Act-Assert (AAA) structure [59] (example in code listing 4.9).

Arrange The arrange section is responsible for the test setup. This ranges from method argument initialization to instantiation of objects being tested. The key principle followed is that all methods or constructors used in this section should be part of the *public* API exposed by the classes/modules used in the test. Also, for test isolation, using fakes is preferred to using mocks or stubs so that the tests mimic conditions as close to the real, production environment, as possible.

Act The act section is responsible for performing the 'work' under test.

Assert The assert section is responsible for validating the results of the test against the expectations. The principle followed in this section is to have just one assert whenever possible. Usually, having more than one assert in a test suggests the test should be broken down into multiple ones.

4.8.2.2 Integration Tests

Definition: *Integration tests determine if independently developed units of software work correctly when they are connected to each other.* [60]

In the context of our system, the 'independently developed' units are its modules (2.4.3).

Since unit tests cover the functionality within each of the modules, integration tests do not need to repeat coverage of all of the edge conditions covered by the unit tests. Instead, they focus on module integration. These tests are typically found in modules in the edge module layer (2.4.3) that implement REST API. For example, the code listing 4.10 shows the resource REST API controller being tested, that uses both, user-access and resource-reservation modules.

Integration tests, similarly to unit tests, need to be isolated, specifically, from external services (e.g., database). For achieving this kind of isolation, integration tests, like unit tests, also prefer using fakes to mocks or stubs.

```
it('should confirm the reservation', async () => {
  // Arrange
  const reservation = make_defaultReservationBuilder().build();
  const assignee = new Assignee('99');
  const changedBy = new ReservationManager(
    '99',
    UserRole.maintainer,
  );
  await effect_createReservation(
    reservation,
    make_createParams('99', assignee, changedBy),
  );

  // Act
  await reservationService.confirm(reservation, changedBy);

  // Assert
  const result = await reservationService.getById('test', '99');
  expect(result.getState()).toBe(ReservationState.confirmed);
});
```

Listing 4.9: Unit Test Example

```
it('should create a new resource', async () => {
  await effect_createOwnerWithId('99');

  const result = await controller.create(
    make_ownerAuthorizedRequestWithId('99'),
    new CreateResourceDTO('test resource', 'test description'),
  );

  expect(result).toEqual(
    new Resource(
      expect.any(String),
      'test',
      'test resource',
      'test description',
    ),
  );
});
```

Listing 4.10: Integration Test Example

4.8.2.3 End-to-End Tests

End-to-end tests typically exercise the entire application stack. Since the system uses purely manual testing for all client applications, end-to-end tests omit the UI and focus on testing the entire server part of the system through its REST API.

The key principle, followed by end-to-end tests, is to mirror the system environment as close to the production one as possible. This includes usage of the same database (MySQL) and communicating with the system only through publicly accessible APIs.

Since end-to-end tests are the most expensive to run, they form only a small part of the system's tests.

4.8.2.4 Golden Master Test

Golden master testing is typically used when interacting with a legacy system or codebase. [61]

Usually, the test executes a valid scenario against the system, persisting its outputs. Then, if a change to the system or codebase is made, the same test scenario is re-run comparing the results. The primary goal of this test is to detect changes in behavior and ensure that all changes are accounted for by having an actual person validate the difference in results. If a change in the results is expected, the person updates the reference results.

The RSaaS uses a version of the golden master test as well. The fact that the system's core modules are event-sourced opens up a possibility of not recording just the communication with the system, but instead its event log (4.5.4). Therefore, any deviation in the results not only means that something in the system changed but rather that the actual state of the system changed.

4.8.3 User Testing

User testing was part of the development process since the first publicly accessible version. From the very beginning of the project, it was planned to deploy the system for users to try as soon as possible, even when the system was under active development, and to roll out bug fixes and features continuously.

The primary users of the systems were Silicon Hill department representatives, present on the meetings and discussions that were held throughout the development process, providing their feedback, and reporting found bugs. Even with a robust test suite in place, a number of bugs were reported on the majority of the sessions (e.g., invalid database string encoding).

During these sessions, there were also identified new requirements and domain aspects of the system, such as the need for workspace creation being predicated on confirmation of a workspace request by a system administrator or a need for reservation comments.

Conclusion

The main goal of this thesis was to create a platform for the creation and management of general-purpose reservation systems. This goal was definitely met. During the platform development we worked closely with various Silicon Hill departments, which are the primary clients of the system's pilot version. Thanks to the architectural decisions made early in the development process we were able to rapidly incorporate their continuous feedback, leading to the system's current feature set.

The first publicly accessible test version of the platform was deployed on the Silicon Hill infrastructure early in the Fall of 2019. We planned to release the platform's beta version, migrating selected reservation systems in the Spring of 2020. However, we were forced to postpone the release until the Summer of 2020 because of the complications caused by the coronavirus outbreak early that year.

We plan to evaluate the system's usage in Silicon Hill in the Fall of 2020. In the case of the positive results, Silicon Hill plans to migrate all existing reservation systems to our platform, making it the primary choice for any Silicon Hill department in need of a new reservation system. If, however, significant weaknesses are found, organization-wide systems migration will be put off until the raised issues will be addressed, and the next round of the evaluation will be conducted.

Bibliography

1. EVANS, Eric. Domain-driven design reference. *Definitions and Pattern Summaries. March*. 2015.
2. BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice: Software Architect Practice_c3*. Pearson Education, 2012. SEI Series in Software Engineering. ISBN 9780132942782. Available also from: https://books.google.cz/books?id=-II73rBDXCYC.
3. *Silicon Hill OAuth2.0 API*. Available also from: https://is.sh.cvut.cz/oauth_api.
4. FOWLER, Martin. Domain Event. Available also from: https://www.martinfowler.com/eaaDev/DomainEvent.html.
5. *A step by step guide to Event Storming – our experience*. Available also from: https://www.boldare.com/blog/event-storming-guide/.
6. *A step by step guide to Event Storming – our experience*. Available also from: https://www.eventstorming.com/.
7. FOWLER, Martin. DDD Aggregate. Available also from: https://www.martinfowler.com/bliki/DDD_Aggregate.html.
8. *Easy!Appointments*. Available also from: https://easyappointments.org/.
9. *Easy!Appointments*. Available also from: https://github.com/alextselegidis/easyappointments.
10. *Bookend*. Available also from: https://www.bookedscheduler.com/.
11. *WebCalendar*. Available also from: https://sourceforge.net/projects/webcalendar/.
12. *WebCalendar*. Available also from: https://github.com/craigk5n/webcalendar.

13. *Reservio*. Available also from: https://www.reservio.com/.
14. *SHerna*. Available also from: https://sherna.siliconhill.cz/rezervace.
15. *Thin Client*. Available also from: https://techterms.com/definition/thinclient.
16. *Thick Client*. Available also from: https://techterms.com/definition/thickclient.
17. RICHARDSON, Chris. Pattern: API Gateway / Backends for Frontends. Available also from: https://microservices.io/patterns/apigateway.html.
18. *The API Gateway Pattern*. Available also from: https://freecontent.manning.com/the-api-gateway-pattern/.
19. KHARENKO, Anton. Monolithic vs. Microservices Architecture. Available also from: https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59.
20. BETTS, Thomas. To Microservices and Back Again - Why Segment Went Back to a Monolith. Available also from: https://www.infoq.com/news/2020/04/microservices-back-again/.
21. HANSSON, David Heinemeier. Majestic Monolith. Available also from: https://m.signalvnoise.com/the-majestic-monolith/.
22. LUMETTA, Jake. When to Start With A Monolith. Available also from: https://nordicapis.com/should-you-start-with-a-monolith-or-microservices/#whentostartwithamonolith.
23. *Pipes and Filters pattern*. Available also from: https://docs.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters.
24. KREMIC, Ned. Horizontal and Vertical User Stories - Slicing the Cake. Available also from: http://www.deltamatrix.com/horizontal-and-vertical-user-stories-slicing-the-cake/.
25. BOGARD, Jimmy. Vertical Slice Architecture. Available also from: https://jimmybogard.com/vertical-slice-architecture/.
26. STENBERG, Jan. Domain-Driven Design with Onion Architecture. Available also from: https://www.infoq.com/news/2014/10/ddd-onion-architecture/.

27. *Asynchronous message-based communication*. Available also from: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/asynchronous-message-based-communication#asynchronous-event-driven-communication.
28. BAILIS, Peter; GHODSI, Ali. Eventual consistency today: Limitations, extensions, and beyond. *Queue*. 2013, vol. 11, no. 3.
29. FOWLER, Martin. Event Sourcing. Available also from: https://martinfowler.com/eaaDev/EventSourcing.html.
30. YOUNG, Greg. Why use Event Sourcing. Available also from: http://codebetter.com/gregyoung/2010/02/20/why-use-event-sourcing/.
31. *Developer Survey Results 2017*. Available also from: https://insights.stackoverflow.com/survey/2017#most-popular-technologies.
32. *Developer Survey Results 2019*. Available also from: https://insights.stackoverflow.com/survey/2019#most-popular-technologies.
33. SWORDS, Simon. Off-the-shelf Software: Advantages and Disadvantages. Available also from: https://www.atlascode.com/blog/the-advantages-and-disadvantages-of-off-the-shelf-software/.
34. WELLS, Don. Spike. Available also from: http://www.extremeprogramming.org/rules/spike.html.
35. FELDER, Ian. SaaS: Single Tenant vs Multi-Tenant - What's the Difference? Available also from: https://digitalguardian.com/blog/saas-single-tenant-vs-multi-tenant-whats-difference.
36. AHMED, Anamika. JSON Web Tokens vs. Session Cookies for Authentication. Available also from: https://medium.com/better-programming/json-web-tokens-vs-session-cookies-for-authentication-55a5ddafb435.
37. *Local Storage*. Available also from: https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage.
38. KRAWCZYK, Dr. Hugo; BELLARE, Mihir; CANETTI, Ran. *HMAC: Keyed-Hashing for Message Authentication* [RFC 2104]. RFC Editor, 1997. Available from DOI: 10.17487/RFC2104. Technical report.

39. HARDT, Dick. *The OAuth 2.0 Authorization Framework* [RFC 6749]. RFC Editor, 2012. Available from DOI: 10.17487/RFC6749. Technical report.
40. HARDT, Dick. *The OAuth 2.0 Authorization Framework* [RFC 6749]. RFC Editor, 2012. Available from DOI: 10.17487/RFC6749. Technical report.
41. *2018 reform of EU data protection rules* [online]. 2018 [visited on 2019-06-17]. Available from: https://ec.europa.eu/commission/sites/beta-political/files/data-protection-factsheet-changes_en.pdf.
42. SCHAAD, Jim. *Use of the Advanced Encryption Standard (AES) Encryption Algorithm in Cryptographic Message Syntax (CMS)* [RFC 3565]. RFC Editor, 2003. Available from DOI: 10.17487/RFC3565. Technical report.
43. *What is REST*. Available also from: https://restfulapi.net/.
44. RICHARDSON, Chris. Pattern: Event sourcing. Available also from: https://microservices.io/patterns/data/event-sourcing.html.
45. PHILLIPA AVERY, Robert Reta. *Scaling Event Sourcing for Netflix Downloads, Episode 2*. Available also from: https://www.youtube.com/watch?v=rsSld8NycCU.
46. FOWLER, Martin. CommandQuerySeparation. Available also from: https://www.martinfowler.com/bliki/CommandQuerySeparation.html.
47. *Key Value Store*. Available also from: https://www.techopedia.com/definition/26284/key-value-store.
48. *Event Sourcing and Snapshots*. Available also from: https://blog.jonathanoliver.com/event-sourcing-and-snapshots/.
49. *Kafka*. Available also from: https://kafka.apache.org/index.html.
50. *RabbitMQ*. Available also from: https://www.rabbitmq.com/.
51. GRZYBEK, Kamil. The Outbox Pattern. Available also from: http://www.kamilgrzybek.com/design/the-outbox-pattern/.
52. FILCIK, Joe. The “Inbox Pattern”. Available also from: https://productcoalition.com/the-inbox-pattern-2a2641e84eab.
53. FIELDING, Roy T.; RESCHKE, Julian. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content* [RFC 7231]. RFC Editor, 2014. Available from DOI: 10.17487/RFC7231. Technical report.
54. *HTTP response status codes*. Available also from: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status.

55. ABRAMOV, Dan. Hot Reloading in React. 2016. Available also from: https://medium.com/@dan_abramov/hot-reloading-in-react-1140438583bf.
56. FOWLER, Marting. OpportunisticRefactoring. 2011. Available also from: https://martinfowler.com/bliki/OpportunisticRefactoring.html.
57. FOWLER, Marting. TestDrivenDevelopment. 2005. Available also from: https://www.martinfowler.com/bliki/TestDrivenDevelopment.html.
58. ROY OSHEROVE Michael Feathers, Robert C. Martin. *The Art of Unit Testing, Second Edition with examples in C#*. Manning Publications, 2013. ISBN 9781617290893. Available also from: https://livebook.manning.com/book/the-art-of-unit-testing-second-edition?origin=product-look-inside.
59. WAKE, Bill. 3A – Arrange, Act, Assert. 2011. Available also from: https://xp123.com/articles/3a-arrange-act-assert/.
60. FOWLER, Martin. IntegrationTests. 2018. Available also from: https://martinfowler.com/bliki/IntegrationTest.html.
61. REZVINA, Sasha. Gold Master Testing. 2020. Available also from: https://codeclimate.com/blog/gold-master-testing/.

Appendix A

Domain Events

A.1 Diagram notation

All diagrams capturing identified domain events used in this chapter use the following notation:

- Red note shapes represent events.
- Blue note shapes represent commands.
- Stick figures represent actors triggering commands.
- Gear icons represent automatic processes triggering commands.
- Trigger associations between actors, automated processes, or events, are captured as a dashed lines with an arrow pointing towards a command they trigger.
- All events following a command in the same row are produced by that command.
- Orange diamonds are a conditional based on which specific events are produced.

A.2 List of Domain Events

A.2.1 Workspace Request Life cycle

The requirements **FR02**–**FR04** describe the life cycle of the workspace request. Investigation of these requirements led to the following scenarios, all of which can be found in the diagram in figure A.1.

Opening a Workspace Request

Actor: A *Workspace Requester* entity.

Command: Open Workspace Request.

Events Produced: Workspace Request Opened.

Creation of a Workspace Owner

Actor: A *Workspace Request Opened* event.

Command: Create Workspace Owner.

Events Produced: User Created, Ownership Claimed.

Workspace Request being Declined

Actor: A *System Administrator* entity.

Command: Decline Workspace Request.

Events Produced: Workspace Request Declined.

Anonymization of a Workspace Owner

Actor: A *Workspace Request Declined* event.

Command: Anonymize User Identity.

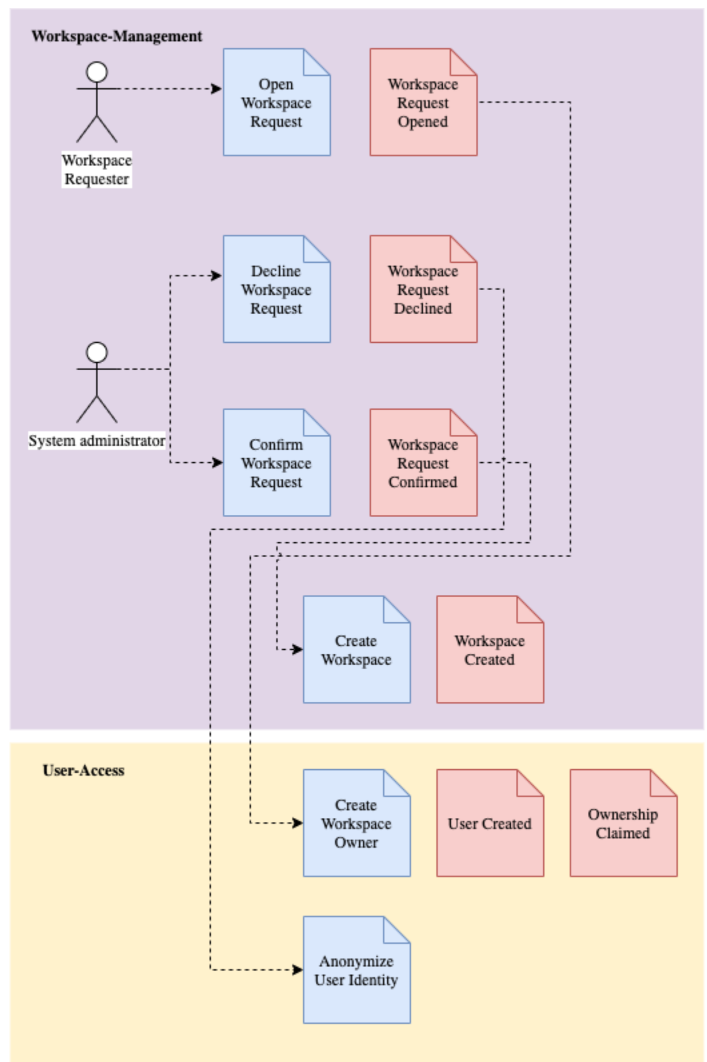
Events Produced: *None*

Confirmation of a Workspace Request

Actor: A *System Administrator* entity.

Command: Confirm Workspace Request.

Events Produced: Workspace Request Confirmed.



Following notation from section A.1

Figure A.1: Workspace Request Life cycle

Creation of a Workspace

Actor: A *Workspace Request Confirmed* event.

Command: Create Workspace.

Events Produced: Workspace Created.

A.2.2 Workspace Life cycle

The requirements **FR05**, **FR27**, and **FR28** describe a workspace life cycle. Diagram in figure A.2 displays the scenarios that revealed these requirements.

Ownership Transfer

Actor: A *Workspace Owner* or a *System Administrator* entity.

Command: Transfer Workspace Ownership.

Events Produced: Workspace Ownership Transferred.

Workspace Ownership Claim

Actor: A *Workspace Ownership Transferred* entity.

Command: Claim Workspace Ownership.

Events Produced: Workspace Ownership Claimed.

Giving Up Workspace Ownership

Actor: A *Workspace Ownership Transferred* entity.

Command: Give Up Workspace Ownership.

Events Produced: Workspace Ownership Gave Up.

Delete Workspace

Actor: A *Workspace Owner* entity.

Command: Delete Workspace.

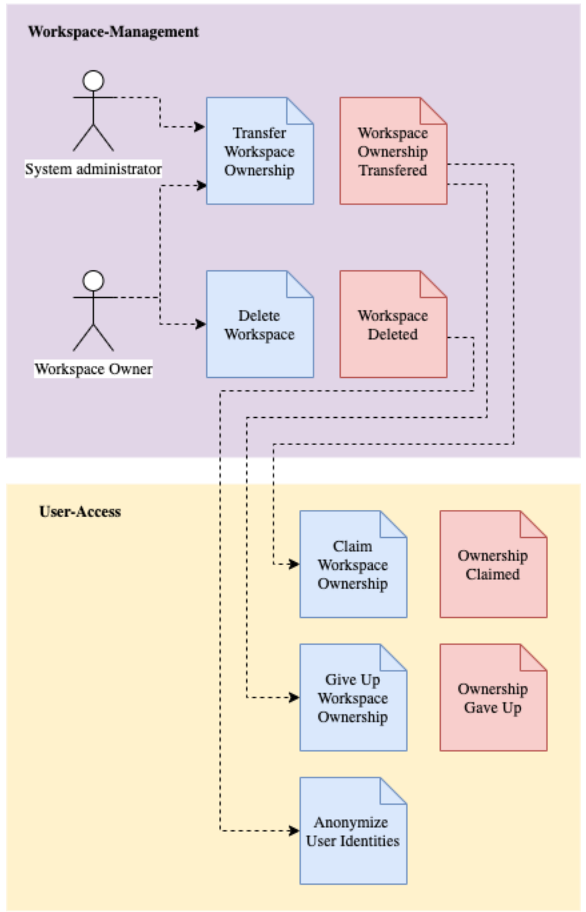
Events Produced: Workspace Deleted.

Workspace User Anonymization

Actor: A *Workspace Deleted* event.

Command: Anonymize User Identities.

Events Produced: *None.*



Following notation from section A.1

Figure A.2: Workspace Life cycle

A.2.3 Resource Life cycle

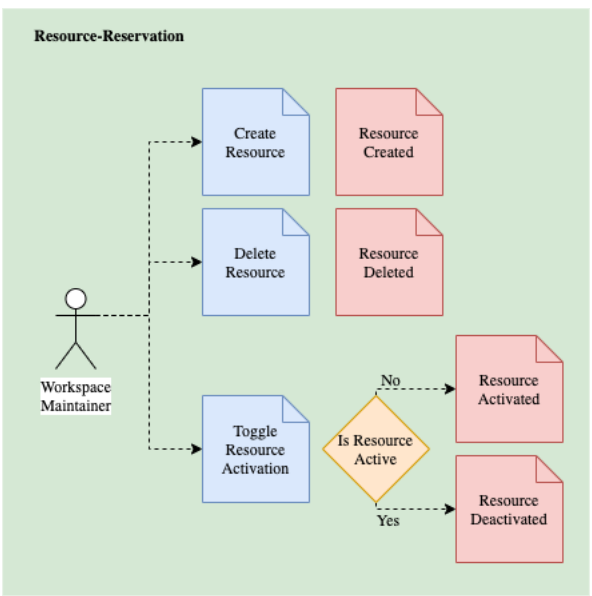
The requirements **FR10**–**FR13** describe the life cycle of the resources captured in the diagram in figure A.3.

Resource Creation

Actor: A *Workspace Maintainer*.

Command: Create Resource.

Events Produced: Resource Created.



Following notation from section A.1

Figure A.3: Resource Life cycle

Resource Deletion

Actor: A *Workspace Maintainer*.

Command: Delete Resource.

Events Produced: Resource Deleted.

Resource Activation Toggling

Actor: A *Workspace Maintainer*.

Command: Toggle Resource Activation.

Events Produced: Resource Activated if the *Resource* is currently inactive, Resource Deactivated otherwise.

A.2.4 Reservation Life Cycle Events

The requirements **FR06**, **FR07**, and **FR14**–**FR16** describe the life cycle of the reservations captured in the diagram in figure A.4.

Creation of Reservation

Actor: A *Reservation Requester* entity.

Command: Create Reservation.

Events Produced: Reservation Created, Reservation Assigned, one or more Resource Added.

Update of Reservation

Actor: A *Reservation Requester* or *Workspace Maintainer* entity.

Command: Update Reservation.

Events Produced: Reservation Date Changed, Reservation Assigned, zero or more Resource Added and zero or more Resource Removed (The events produced by the command depend on how did the reservation change. For example, if reservation did not change at all, no events will be produced. If, however there was one resource added to the reservation, and one removed, two events will be produced: *Resource Removed* and *Resource Added*).

Reservation Cancelation

Actor: A *Reservation Requester* or *Workspace Maintainer* entity.

Command: Cancel Reservation.

Events Produced: Reservation Canceled.

Reservation Confirmation

Actor: A *Workspace Maintainer* entity.

Command: Confirm Reservation.

Events Produced: Reservation Confirmed.

Reservation Rejection

Actor: A *Workspace Maintainer* entity.

Command: Reject Reservation.

Events Produced: Reservation Rejected.

Reservation Completion

Actor: An automated job triggered when the end of the reservation is no longer in the future.

Command: Complete Reservation.

Events Produced: Reservation Completed.

A.2.5 Reservation Comments

Even though there were no requirements mentioning reservation comments, the modeling of the bounded contexts 1.3.1 uncovered a need for such a concept. Diagram in figure A.5 captures events associates with them.

Reservation Comment Creation

Actor: A *Comment Author* entity.

Command: Create Reservation Comment.

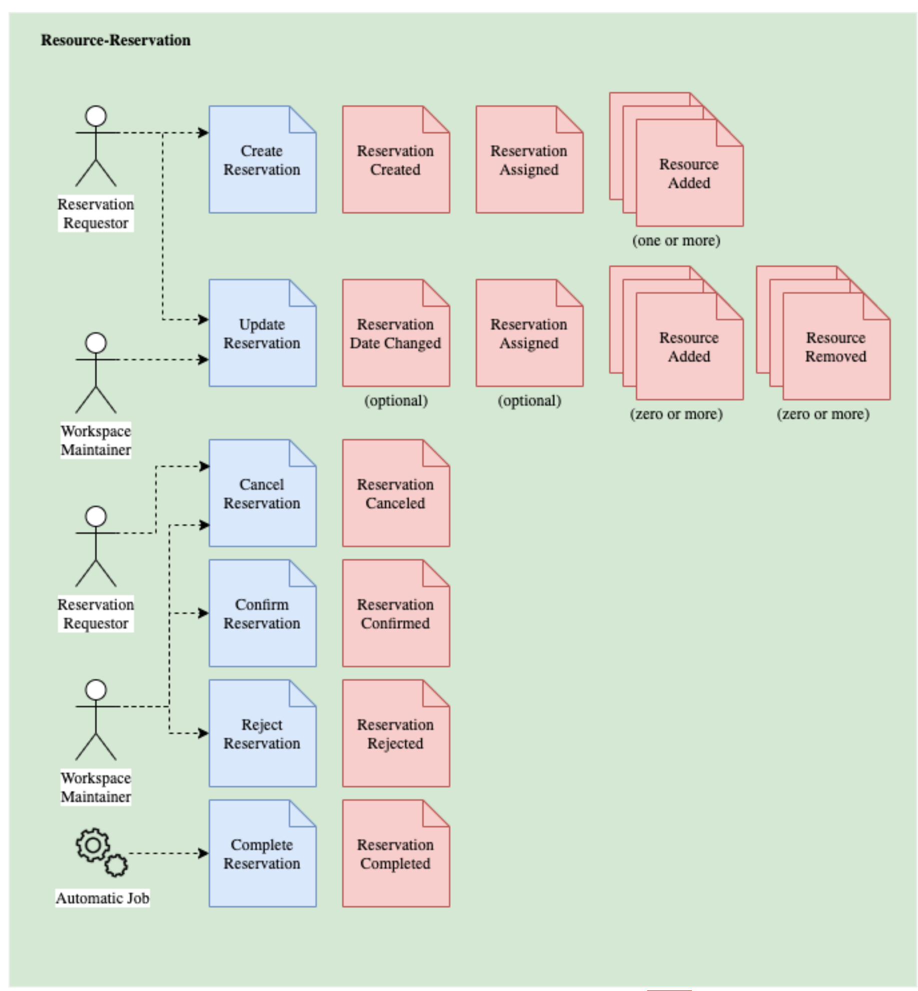
Events Produced: Reservation Comment Created.

Reservation Comment Editing

Actor: A *Comment Author* entity.

Command: Edit Reservation Comment.

Events Produced: Reservation Comment Edited.



Following notation from section A.1

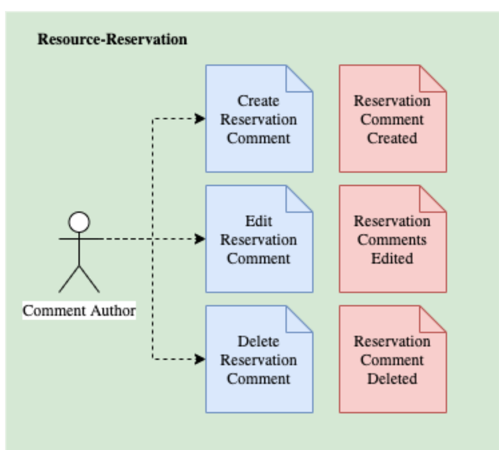
Figure A.4: Reservation Life cycle

Reservation Comment Deletion

Actor: A *Comment Author* entity.

Command: Delete Reservation Comment.

Events Produced: Reservation Comment Deleted.



Following notation from section A.1

Figure A.5: Reservation Comment Life cycle

A.2.6 User Life cycle

The requirements **FR09**, **FR20**–**FR22**, and **FR24**–**FR26** describe actions associated with users captured in the diagram in figure A.6.

Toggling of User Ban

Actor: A *Workspace Maintainer/Administrator/Owner* entity.

Command: Toggle User Ban.

Events Produced: User Ban Lifted if the *User* is currently banned, User Ban Placed otherwise.

Changing of User Role

Actor: A *Workspace Maintainer/Administrator/Owner* entity.

Command: Change User Role.

Events Produced: User Role Changed.

User Creation

Actor: A *User* entity.

Command: Create User.

Events Produced: User Created.

User Deletion

Actor: A *User* entity.

Command: Delete User.

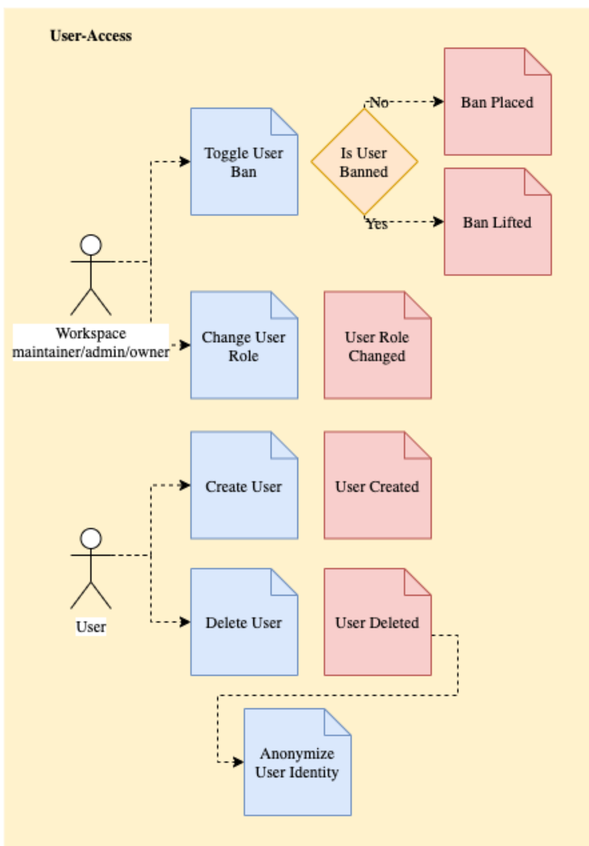
Events Produced: User Deleted.

User Identity Anonymization

Actor: A *User Deleted* event.

Command: Anonymize User Identity.

Events Produced: *None.*



Following notation from section A.1

Figure A.6: User Life cycle

Appendix B

Acronyms

ACID Atomicity, Consistency, Isolation, Durability

API Application Programming Interface

CD Continuous Delivery

CI Continuous Integration

CQS Command Query Separation

CRUD Create, Read, Update, Delete

DDD Domain-Driven Design

DOM Document Object Model

GUI Graphical user interface

HTTP Hypertext Transfer Protocol

REST Representational State Transfer

RSaaS Reservation System as a Service

SaaS System as a Service

URL Uniform Resource Locator

URL Uniform Resource Identifier

XML Extensible markup language