Purpose

The system design is documented in the System Design Document (SDD). It describes additional design goals set by the software architect, the subsystem decomposition (with UML class diagrams), hardware/software mapping (with UML deployment diagrams), data management, access control, control flow mechanisms, and boundary conditions. The SDD serves as the binding reference document when architecture-level decisions need to be revisited.

Audience

The audience for the SDD includes the system architect and the object designers as well as the project manager.

Table of Contents

1. Introduction 2
   1.1 Overview 2
   1.2 Definitions, acronyms, and abbreviations 2
   1.3 References 2
2. Design Goals 2
3. Subsystem decomposition 3
4. Hardware/software mapping 3
5. Persistent data management 3
6. Access control and security 4
7. Global software control 4
8. Boundary conditions 5

Document History

Rev.	Author	Date	Changes
01	Jakob Mayerhofer	16.07.2022	
02	Jakob Mayerhofer	20.07.2022	

1. Introduction

The purpose of the Introduction is to provide a brief overview of the software architecture. It also provides references to other documents.

1.1 Overview

The software architecture of this project consists of 2 parts: the client and the server. Both use different architectural styles. The server is built using a layer architecture, while the client is built using a component based GUI with global state management.

1.2 Definitions, acronyms, and abbreviations

Server: in this document, the part of the application that handles persistent data and provides interfaces to get that data via REST API is referred to as server.
Client: in this document, the part of the application that uses the data provided by the server and displays it in a meaningful way is referred to as client.

1.3 References

To read more about the goals and requirements of this project, the RAD describes these aspects in detail.

2. Design Goals

This section describes the design goals and their prioritization (e.g. usability over extensibility). These are additional nonfunctional requirements that are of interest to the developers. Any trade-offs between design goals (e.g., usability vs. functionality, build vs. buy, memory space vs. response time), and the rationale behind the specific solution should be described in this section. Also the rationale of all other decisions must be consistent with described design goals.

The focus of this software development project is usability. The client has emphasized multiple times that usability and ease of use must be prioritized above other non-functional requirements such as robustness, safety, functionality etc. Additionally, the project is bound by a rather strict time frame and low financial resources. That is why the core functionalities described be the functional requirements and their correct realization drive the development process. The development team has also set code cleanliness and error minimization as internal goals. Furthermore, internal documentation as well as transparency standards have been set to comply with deliverables requirements more easily and to facilitate working on both the client– and server side.

3. Subsystem decomposition

This section describes the decomposition of the system into subsystems and the services provided by each subsystem. The services are the seed for the APIs detailed in the Object Design Document.

The system can be divided into two main parts: the server and the client. The server is constructed using the layer architectural style, meaning that it can be further divided into the controller, service, and repository layer for each entity respectively. Each of these layer architectures provides interfaces through which an application can interact with it. These interfaces are then bundled into one interface, namely that of the server.
The client then uses these interfaces to display data meaningfully. The client can be dived into the global state subsystem (context) and the pages. The global state contains information about the current search query and reservation, as well as methods to use the services provided by the server. Each pages uses the global state to get data and call API calls.

4. Hardware/software mapping

This section describes how the subsystems are mapped onto existing hardware and software components. A UML deployment diagram accompanies the description. The existing components are often off-the-shelf components. If the components are distributed on different nodes, the network infrastructure and the protocols are also described.

As this project is a greenfield project, there are no existing hardware/software components.
However, software project is realised with a continuous integration process by using Docker containers, which then run both the server and the client. This container runs on a server. The browser is then able to render the client via URL.

5. Persistent data management

This section describes how the entity objects are mapped to persistent storage.
It contains a rationale of the selected storage scheme, file system or database, a description of the selected database and database administration issues.

The persistent data storage of this project is realized via a database called h2 using the in-memory mode. Reasons for this choice include ease of setup, compatibility with spring boot and speed to name a few. The selected database complies with the requirements, is easy to test and debug. Entity Objects are mapped to persistent storage using Spring boot, which automatically converts java objects into a format that can be stored in h2. The schemas of the entities and their respective relations can be viewed in the *db > entity* subfolder. Queries are formed using the JPA query

language JPQL, which is a subset of SQL and highly compatible with Spring Boot. All available queries can be seen in the *db* > *repository* subfolder. When querying the database, the entries are mapped to JSON using response mapper (can be seen in the *controller* > *responseMapper* subfolder), which can then be used by client-side applications that display the data. The whole persistent data management process uses the controller-service-repository approach. Therefore, functionalities can be found in the corresponding folders accordingly.

6. Access control and security

This section describes the access control and security issues based on the nonfunctional requirements in the requirements analysis document. It also describes the implementation of the access matrix based on capabilities or access control lists, the selection of authentication mechanisms and the use of encryption algorithms.

Access control for this software development project is relatively simple, as the application only supports one type of user (customer). Furthermore, none of the classes used have different versions, meaning that each actor has the same access rights to all instances of a class. That is why an access matrix does not contribute to a better understanding of the access control structures of this project and was therefore omitted to easier meet the client's timeline requirement.
However, some information such as reservation data and the right to cancel/confirm a reservation is subject to access control, as this data should not be publicly available for all users. That's why UUID's are used in our API infrastructure to ensure that only the user issuing the reservation can also view it. Additional security measures have been taken regarding the confirmation of reservations. In addition to the reservation's UUID an access token is needed to confirm a reservation, which is only available to the customer after they have received the according email described in the requirements. This security measure is also implemented using API structures.

7. Global software control

This section describes the control flow of the system, in particular, whether a monolithic, event-driven control flow or concurrent processes have been selected, how requests are initiated and specific synchronization issues.

For the control flow of the system a monolithic, polling-based, and centralized approach has been chosen. The flow is polling based, to ensure high performance and improve user friendliness. The system is designed using a centralized design to comply with the client requirements and, most notably, to meet the timeline set by the client. A decentralized approach would take too much time to implement due to higher overhead in communication.

8. Boundary conditions

This section describes the use cases how to start up the separate components of the system, how to shut them down, and what to do if a component or the system fails.

The server and the client can be run independent from another, but there is also an option to start them at the same time. The system is built to handle some failures. In case the system fails nonetheless, it is easy to restart it by terminating the application and building it again. This is true for the server and the client.