**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 3

1.1 Overview 3

1.2 Definitions, acronyms, and abbreviations 3

1.3 References 3

2. Design Goals 3

2.1 Additional nonfunctional  requirements 3

2.2 Design trade-offs 3

3. Subsystem decomposition 3

Layers vs. partitions 4

Architectural style 4

Cohesion & coupling 4

4. Hardware/software mapping 4

Identification of nodes 4

Special purpose systems 4

Buy vs. build 4

Network connectivity 4

5. Persistent data management 4

Storing persistent objects 4

Filesystem vs. database 4

6. Access control and security 5

Access control 5

ACL vs. capabilities 5

Security 5

7. Global software control 5

Monolithic 5

Event-driven 5

Conc. processes 5

8. Boundary conditions 5

Initialization 6

Termination 6

Failure 6

**Document History**

|  |  |  |  |
| --- | --- | --- | --- |
| Rev. | Author | Date | Changes |
|  | Ludwig Gröber | 08.07.2022 | Everything |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Introduction

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

## Overview

While designing the system we always had three principles in mind: Usability – Flexibility and Functionality. Throughout this document we are trying to elaborate on what design principles we used and what factors drove our decision-making to what we chose in the end.

In our System-Design Process we used a divide and conquer process where we identified design goals and addressed major design goals first. Compared to designing an algorithm for example, we found that in system design often the measure of success is less clear.

## Definitions, acronyms, and abbreviations

None.

## References

Please do see in reference to this document the various diagrams we provided. Especially the Class-Diagram, Use Case Diagram und the two Scenarios as well as Communication Diagrams. As well as API specification and UML deployment diagram.

# 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.*

## Additional nonfunctional  requirements

Because the project is graded based on functionality and usability mainly, these were our priorities in all design goal trade-offs that were of interest to us as developers.

From the nonfunctional requirements given in the problem statement, such as Usability, Conformance to guidelines as well as a Server System with Services we developed our Design-Goals. While developing these, there are always implicitly other NFRs present, such as an acceptable runtime as well as good maintainability and easy-to-read code.

## 2.2 Design trade-offs

In any design-goal trade off involving usability, we always prioritized usability, for others such as robustness vs rapid development we compromised to a point, where the system was subjectively robust, but also to be developed in a reasonable amount of time. For the sake of rapid development, we sacrificed some of the additional functionality we would have considered to be a nice extra. In tradeoffs such as backwards-compatibility vs readability we weren’t heavily involved, because our system only has to run on current hardware and software. So, our main focus here was on code readability.

Because speed was only an implicit requirement connected to usability, this also wasn’t a main concern, but somehow considered in the process.

# 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.*

## Layers vs. partitions

The first thing we did in the project was to define an API for the communication between server and client. From this specification on, the Server and Client sub-team developed the services or used the services according to the specification. The Server provides different services, namely: Reservations, Restaurant Layout, Restaurant, Reviews. Please see API specification attached to the project.

The server is partitioned into the different services, which each are divided into Model-Controller-Repository Sub-packages, to achieve low coupling and high cohesion.

The Client is partitioned into Components, Models, Styles, Views and Util to achieve low coupling and high cohesion.

## Architectural style

Inside the server, the services and functionality are structured in a MVC architectural style, while the whole system is designed in a Closed Multi-Layer Architecture. The goal of this is to enable a low coupling and high cohesion system by providing different layers within the Server and Client. The Client architecture is also based on a MVC Architecture, where namely components are processed in the model and displayed in the view.

The Server MVC is being split in model and controller packages, containing different controllers and models for e.g. price category. The services the server provides are Restaurant Service, Reservation Service, Review Service and Restaurant Layout Service.

The REST Architectural style is used for HTTP requests. Therefore, a distributed base of clients can have access to resources and services. Also REST acts as a kind of Façade between Server and Client to achieve low coupling.

## Cohesion & coupling

Because of the separation of Server and Client through the API specification and the MVC Architecture in Server and Client, an overall system can be achieved that has a minimum number of relationships between packages and manages the services provided as good as possible with high cohesion.

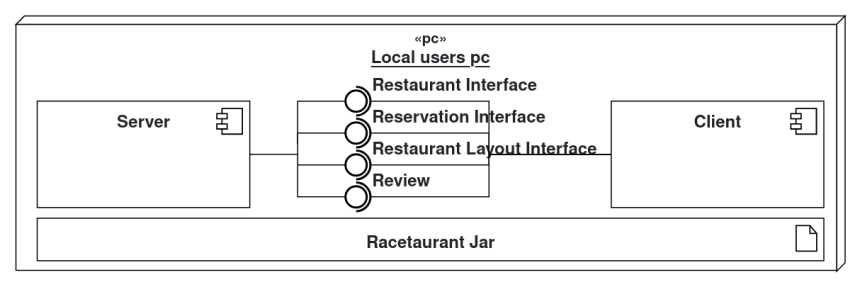
# Hardware/software mapping

*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.*

## Identification of nodes

The system consists of three hardware nodes, the client, the server and the database. It is a 3 layered architectural style, where the application consists of three hierarchically ordered layers.

To clarify, please find the UML deployment diagram below:



This diagram is for just running the binary on a single machine.

## Special purpose systems

None.

## Buy vs. build

There was no buy-vs-build decision to be made in our system. We had to build.

## Network connectivity

The system could be applied to any network out there, but is applied to a Browser-Based Application right now.

# 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.*

## Storing persistent objects

Restaurants, Reviews, Reservations, Restaurant-Layout are stored as persistent objects on the servers file system.

## Filesystem vs. database

The design decision whether to choose a file system or a database was really simple this time, because for structured data a relational database is the best choice.

A database achieves high performance and the data is persisted permanently.

Therefore, it the best choice for our REST-Server-Client Architecture.

# 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

The system is open to use and therefore no information needs to be hidden from certain users. As we don’t have accounts, we also don’t have access control.

## ACL vs. capabilities

No Access Control means no ACL and no Capabilities.

## Security

There is no big security aspect involved, because all the information we have, we also provide publicly to users. For security we would use the https web protocol once deployed to a real server, which offers state of the art encryption and security.

# 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.*

## Monolithic

We used a Monolithic design, where the architecture provides all the functionalities needed. There are some small concurrent processes involved in Spring, but overall, it can be considered a Monolithic design.

## Conc. processes

Some requests to the API can and will run concurrently, but because of the overall system architecture with low coupling and high cohesion where services are independent of each other, there is no problem with race conditions or the similar.

# 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.*

## Initialization

To initialize the system, the application jar has to be started and the Local IP Address has to be called in a Web-Browser.

## Termination

The system will terminate when the user closes it. Server will run persistently if it was deployed to a real server.

## Failure

When the system fails, it will try to handle the error and else terminate.