

**Hotel Reservation System**

Name: ***Mărcuș Cristian-Viorel***

Group: ***30236***

Table of Contents

[Deliverable 1 3](#_Toc64843130)

[Project Specification 3](#_Toc64843131)

[Functional Requirements 3](#_Toc64843132)

[Use Case Model 3](#_Toc64843133)

[Use Cases Identification 3](#_Toc64843134)

[UML Use Case Diagrams 3](#_Toc64843135)

[Supplementary Specification 3](#_Toc64843136)

[Non-functional Requirements 3](#_Toc64843137)

[Design Constraints 3](#_Toc64843138)

[Glossary 3](#_Toc64843139)

[Deliverable 2 3](#_Toc64843140)

[Domain Model 3](#_Toc64843141)

[Architectural Design 4](#_Toc64843142)

[Conceptual Architecture 4](#_Toc64843143)

[Package Design 4](#_Toc64843144)

[Component and Deployment Diagram 4](#_Toc64843145)

[Deliverable 3 4](#_Toc64843146)

[Design Model 4](#_Toc64843147)

[Dynamic Behavior 4](#_Toc64843148)

[Class Diagram 4](#_Toc64843149)

[Data Model 4](#_Toc64843150)

[System Testing 4](#_Toc64843151)

[Future Improvements 4](#_Toc64843152)

[Conclusion 4](#_Toc64843153)

[Bibliography 4](#_Toc64843154)

# Deliverable 1 (week 5)

## Project Specification

The project is created to suit a Reservation System for a hotel chain, as present at any hotel desk, operated by a receptioner. In a general manner, it enables the user of the system to make bookings for guests, manage rooms and roomtypes, add guest to the database, manage their details/credentials, view users/admins. There are two types of users: regular and admin priviledges (\*see glossary). The regular user can login into the platform, make bookings, manage guests and their data. Admins will login with superuser priviledges and they are able to create new rooms and roomtypes, as well as manage current registered users.

Regarding the technology stack used, I decided to implement it as a full-stack app in Java using Spring Boot for web support. The system architecture uses a mechanism provided by Spring where the data is stored by Repositories and it is retrieved and managed by Service classes. All the functionality was coded so it is easy to maintain and upgrade, since I depended upon abstractions (for instance, using interfaces for services).

## Functional Requirements

* **User registration**: allows users to create user profiles using name and password. The information is stored into an encrypted form in the user database;
* **Main dashboard**: users can modify, delete and add bookings, query guests by their address, email, phone number, modifiy account credentials, view current vacant rooms with their corresponding pricing etc.;
* **Admin dashboard**: admins (users with priviledges), besides a regular user, can add new rooms of the hotel in the system, new roomtypes, modify prices of the current available options, run seasonal sales etc.;
* **Payment**: currently, the sistem will manage payments by setting a flag-like field in a booking entry (called isPaid).

## Use Case Model

### Use Cases Identification

|  |  |  |
| --- | --- | --- |
| 1 | Use-case | login operation |
| Level | user goal level |
| Primary actor | user/admin |
| Main success scenario | * user is prompted with two fields that require input (name and password); * after competion, “login” button may be pressed; * the system will let them know if the credentials match a current user; * the system displays a success splash screen. |
| Extensions | * if the user does not have an account, they are prompted to create one. |

|  |  |  |
| --- | --- | --- |
| 2 | Use-case | create account operation |
| Level | user goal level |
| Primary actor | user/admin |
| Main success scenario | * user will press “create account” button; * user is prompted with two fields that require input (name, password and other credentials); * after competion, “register” button may be pressed * the system will let them know if the current name/email is already assigned to an user; * the system displays a success splash screen. |
| Extensions |  |

|  |  |  |
| --- | --- | --- |
| 3 | Use-case | add (hotel) guests |
| Level | user goal level |
| Primary actor | user/admin |
| Main success scenario | * user will login to the platform; * if success, the user will press “add guests” button; * user will complete information about guests (name, email, phone, address); * an available room will be chosen from the UI; * submit button will be pressed by the user. |
| Extensions |  |

|  |  |  |
| --- | --- | --- |
| 4 | Use-case | add booking |
| Level | user goal level |
| Primary actor | user/admin |
| Main success scenario | * user will login to to the platform; * if success, the user will press “add booking” button; * user will complete information about bookings (check-in date, check-out date, total, paid status); * submit button will be pressed by the user. |
| Extensions | * if the paid status is set to false upon creation, the user will follow up later for modification. |

|  |  |  |
| --- | --- | --- |
| 5 | Use-case | add room type |
| Level | user goal level |
| Primary actor | admin only |
| Main success scenario | * admin will login to to the platform; * if success, the admin will press “add new room type” button; * admin will complete information about room types (name, cost and description); * submit button will be pressed by the admin. |
| Extensions |  |

|  |  |  |
| --- | --- | --- |
| 6 | Use-case | add room |
| Level | user goal level |
| Primary actor | Admin only |
| Main success scenario | * admin will login to to the platform; * if success, the admin will press “add new room” button; * admin will complete information about rooms (number and room type); * submit button will be pressed by the admin. |
| Extensions |  |

|  |  |  |
| --- | --- | --- |
| 7 | Use-case | browse for/modify bookings/guests |
| Level | user goal level |
| Primary actor | user/admin |
| Main success scenario | * user will login to to the platform; * if success, the user will press “browse bookings” or “browse guests” button; * user can view or modify current bookings or guests. |
| Extensions |  |

|  |  |  |
| --- | --- | --- |
| 8 | Use-case | browse for/modify rooms/room types |
| Level | user goal level |
| Primary actor | admin |
| Main success scenario | * admin will login to to the platform; * if success, the admin will press “browse rooms” or “browse room types” button; * admin can view or modify current rooms or room types. |
| Extensions |  |

|  |  |  |
| --- | --- | --- |
| 9 | Use-case | change account credentials |
| Level | user goal level |
| Primary actor | user/admin |
| Main success scenario | * user will login to to the platform; * if success, the user will press “change credentials” button; * user will modify the fields; * the system will let them know if the credentials match a current user; * if true, an error will pop up; * else, the system displays a success splash screen; * the user will be prompted to login with the new credentials. |
| Extensions |  |

### UML Use Case Diagrams

Diagram

Description automatically generated

## Supplementary Specification

### Non-functional Requirements

* **Availability and reliability**: the system should be available 24/7 to the receptionist and to the admins. It must suit any type of hotels, especially the ones who operate the front desk during graveyard shifts, where there will be no technical support at that time so it is strictly needed for my implementation;
* **Usability**: the system must be easy to use and browse for users with different levels of computer knowledge. The efficient feedback from the user interface is needed when an action is made;
* **Compliance**: the system must comply with the current GDPR regulations. The guest and user data stored in the system must be encrypted and protected;
* **Scalability**: the system should be easily scalable to accommodate future changes. It is important since a future change could imply adding a roomservice system integrated alltogether, or even integration with a hotel chain. It should also handle big numbers of users, hotel rooms and roomtypes and bookings.

### Design Constraints

Regarding the design constraints, I have chosen to use Java along with Spring Boot because it enables you to create standalone apps that run on their own, without relying on an external server. The mechanism behind it uses Tomcat during the initialization process.

As for the dependency manager, I used Maven due to its flexibility and for its user experience, as its longer tenure makes it to be supported by many IDEs (including IntelliJ, the IDE used for this project).

Mockito framework was used for testing purposes. It is an extensive tool great for safe refactoring, for its wide annotation support and also for the lack of handwriting stuff, since you don’t have to write your own mock objects (mock artifacts are generated during runtime only).

## Glossary

(\*) In my system implementation, ***user*** is the most elementary type of actor. With that, an ***admin*** is inherently an user by default, which also acquired elevated access to the system data, such as adding new rooms in the hotel and/or adding new roomtypes. An admin can also remove users or restrict access to the system. The flag present in the User class that marks the superuser is named explicitly “***isAdmin***”.

As a consequence, in the following documentation, ***admin*** and ***user*** terms will be sometimes interchanged accordingly.

# Deliverable 2 (week 8)

## Domain Model

My system consists of 6 main model which create the overall domain model: *Booking, Guest, GuestData, Room, RoomType, User*.

|  |  |  |
| --- | --- | --- |
| 1 | Name | Booking |
| Attributes | id, checkInDate, checkOutDate, total, isPaid, user |
| Associations | Each booking instance will have an user associated to it (the creator) in a **ManyToOne** relationship with User entity. |

|  |  |  |
| --- | --- | --- |
| 2 | Name | Guest |
| Attributes | id, booking, room, guestData |
| Associations | Each guest instance will have a booking associated to it (every guest in the hotel will have a reservation) in a **ManyToOne** relationship with Booking entity. Also, it will have a room associated to it (every guest will have a room assigned to) in a **ManyToOne** relationship with Room entity. GuestData will be stored for each guest through an **OneToOne** relationship. |

|  |  |  |
| --- | --- | --- |
| 3 | Name | GuestData |
| Attributes | id, guest, name, email, phone, address |
| Associations | Each guestData instance will have a guest associated to it (every guest will have an unique corresponding guest data) in a **OneToOne** relationship with Guest entity. |

|  |  |  |
| --- | --- | --- |
| 4 | Name | Room |
| Attributes | id, number, guests (List), roomType |
| Associations | Each room instance will have one or more guests associated to it in a **OneToMany** relationship with Guest entity. Also, it will have a roomType associated to it (every room must have a room type) in a **ManyToOne** relationship. |

|  |  |  |
| --- | --- | --- |
| 5 | Name | RoomType |
| Attributes | id, name, cost, description, rooms (List) |
| Associations | Each roomType instance will have one or more rooms associated to it in a **OneToMany** relationship with Room entity. |

|  |  |  |
| --- | --- | --- |
| 6 | Name | User |
| Attributes | id, name, pass, isAdmin, bookings (List) |
| Associations | Each User instance will have one or more bookings associated to it in a **OneToMany** relationship with Booking entity. |

Diagram

Description automatically generated

## Architectural Design

### Conceptual Architecture

Naturally, Spring Boot is built following a ***layered architecture*** in which each layer communicates to other layers. Organized into horizontal layers, where each layer has a specific responsibility and communicates with adjacent ones. This will help achieve separation of concerns and modularity, making the system easier to maintain and modify. Besides that, this approach will also make the system very scalable and flexible as new resources are added to the system later on – so that each part of the system can evolve independently. Testing is also made easier by isolating different parts of the system, as we can test them independently.

*Presentation layer* consists of views (the front-end of the application). It handles the HTTP requests and performs authentication. It will convert the JSON’s to Java Objects and vice-versa. Once it finishes processing, the business layer is called.

The *business logic layer* contains all the operations on data provided, so it acts as the “brain” or the processor of the system. It consists of services classes. It is responsible for validation and authorization.

The *persistence layer* contains all the database storage logic. It is responsible for converting business objects to the database row and vice-versa.

*Database layer* will consist of CRUD operations. Creating, retrieving, updating and deleting data will be assured by this layer.

The layered architecture is implemented using **dependency injection** (using **@Autowired**) and inversion of control.

### Package Design

Diagram

Description automatically generated

### Component and Deployment Diagram

Diagram

Description automatically generated

# Deliverable 3

## Design Model

### Dynamic Behavior

[Create the interaction diagrams (1 sequence, 1 communication diagrams) for 2 relevant scenarios]

### Class Diagram

A screenshot of a computer

Description automatically generated with medium confidence

Singleton Pattern – spring uses singleton to restrict one object per IoC container. In other words, it will create only one bean for each type per application context. It is present in Controllers to fields which are injected (@Autowired).

Factory Method Pattern – spring uses dependency injection framework that will treat a bean container as a factory that produces beans (BeanFactory).

Template Method Pattern – is used when managing queries on the database (connection, execution of the query, cleanup and closing the connection).

## Data Model

[Create the data model for the system.]

# System Testing

[Describe the testing methides and some test cases.]

# Future Improvements

[Present some features that apply to the application scope.]

# Conclusion

# Bibliography