Hotel Management System Report
Group22
CS202
2022-2023 Fall

Introduction

This project is about a hotel management system that allows users to make bookings, check in, check out, pay for their stay, and access various amenities and services offered by the hotel. The system also allows administrators to manage the rooms, users, and amenities, as well as assign cleaning tasks to housekeeping staff.

Problem Definition

The problem that this project aims to solve is the manual management of hotel operations, which can be time-consuming and error-prone. By automating these processes, the hotel can improve efficiency and reduce the risk of mistakes.

Design Explanation

The design of the hotel management system in this project is centered around the concept of room bookings and the various entities that are involved in the process. The system includes tables for storing information about users, rooms, amenities, and bookings.

The Users table stores information about the different types of users that exist in the system, including guests, administrators, receptionists, and housekeepers. Each user has a unique UserID, a username and password for logging into the system, and a salary if they are an employee. The UserType column in this table is a foreign key that references the TypeID column in the UserTypes table, which stores the names of the different user types.

The Rooms table stores information about the different rooms in the hotel, including their names, whether they are currently booked, or cleaned, and the type of room they are. The

RoomType column in this table is a foreign key that references the TypeID column in the RoomTypes table, which stores the names of the different room types.

The Amenities table stores information about the different amenities that are available in the hotel, such as a minibar, TV, WiFi, pool, gym, spa, bar, restaurant, concierge, and laundry service. Each amenity has a unique AmenityID and a name.

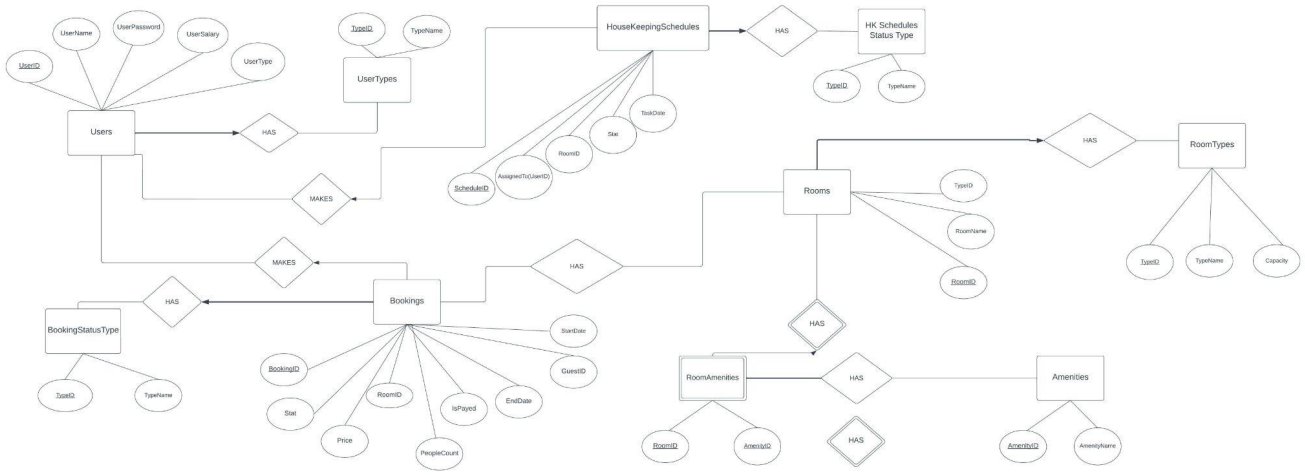
The RoomAmenities table is a many-to-many relationship table that stores the amenities that are available in each room. This table has two foreign keys, RoomID and AmenityID, which reference the primary keys of the Rooms and Amenities tables, respectively.

The Bookings table stores information about the bookings that have been made for the rooms in the hotel. Each booking has a unique BookingID, a start and end date, a price, and a status. The RoomID column in this table is a foreign key that references the RoomID column in the Rooms table, and the UserID column is a foreign key that references the UserID column in the Users table. The Status column is a foreign key that references the TypeID column in the BookingStatusTypes table, which stores the names of the different booking status types.

The HousekeepingSchedules table stores information about the tasks that have been assigned to the housekeepers in the hotel. Each task has a unique ScheduleID, a start and end date, and a status. The RoomID column in this table is a foreign key that references the RoomID column in the Rooms table, and the Stat column is a foreign key that references the TypeID column in the HKScheduleStatusTypes table.

In terms of user views, the different types of users in the system will have different views on the data in these tables. For example, guests will only be able to see their own bookings and any amenities that are available in their rooms. Administrators will have a more comprehensive view of the data in the system, including the ability to view and edit all of the data in the tables. Receptionists will have a similar level of access, but may not have the ability to edit certain sensitive data, such as user salaries. Housekeepers will only be able to view and edit the tasks that have been assigned to them and the status of those tasks.

ER Diagram



Primary Keys

The primary keys in this project are:

- Users: UserID
- UserTypes: TypeID
- Rooms: RoomID
- RoomTypes: TypeID
- Amenities: AmenityID
- RoomAmenities: AmenityID, RoomID
- Bookings: BookingID
- BookingStatusTypes: TypeID
- HousekeepingSchedules: ScheduleID
- HKSchedulesStatusType: TypeID

Foreign Keys

The foreign keys in this project are:

- Users: UserID
- Users: UserType
- Rooms: RoomType
- RoomAmenities: RoomID, AmenityID
- Bookings: RoomID, GuestID, Stat
- HousekeepingSchedules: RoomID, Stat
- RoomAminities: RoomID, AminityID

The primary keys are used to uniquely identify each record in a table, while the foreign keys are used to establish a relationship between different tables. In this project, the primary keys are used as the primary means of identifying and accessing the different records, while the foreign keys are used to link the records in different tables together.

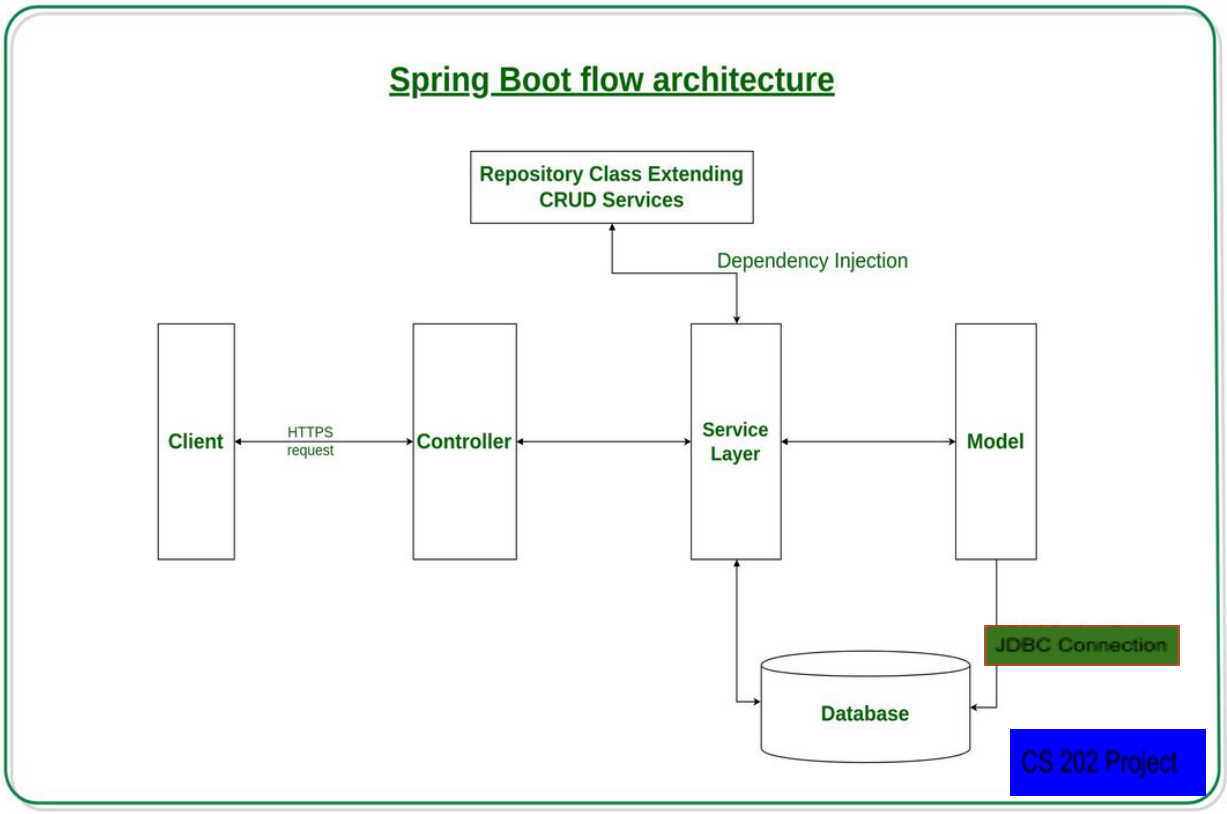
Functional Dependencies

- Users= UserID → Username, UserPassword, UserSalary, UserType
- UserTypes= TypeID → TypeName
- Rooms= RoomID → RoomName, RoomType
- RoomTypes= TypeID → TypeName, RoomCapacity
- HKScheduleStatusTypes= TypeID →TypeName
- HousekeepingSchedules= ScheduleID → RoomID, AssignedTo, TaskDate, Stat
- Amenities= AmenityID → AmenityName
- Bookings= BookingID → RoomID, GuestID, StartDate, EndDate, Price, IsPayed, PeopleCount, Stat
- BookingStatusTypes= TypeID → TypeName

BackEnd Design Decisions

In the spring boot backend part we have separated each possible major scenarios into controllers which are the entry point of the applications and each controller has its own related task independent from other controllers, also each controller has its own corresponding Serivce and Repository in order to write the business logic and get access to Mysql Database of our hotel management system for each of the endpoints.

Project DataFlow/Request Response Structure



Above is the data flow structure of the whole application, it also has Docker in addition to the local server runs.

Each request first will enter the controller and then corresponding Server and Repos and return according to the business logic implemented inside each of the endpoints.

We used JDBC for data access and also Docker script to build the Docker image of the project is provided as:

```
FROM adoptopenjdk/openjdk14
EXPOSE 8080
ARG JAR_FILE=target/project-1.0-SNAPSHOT.jar
ADD ${JAR_FILE} hotelManagementSystemProject.jar
ENTRYPOINT ["java","-jar","/hotelManagementSystemProject.jar"]
```

This script is a Dockerfile, used to create a Docker image for a Java application.

The first line "FROM adoptopenjdk/openjdk14" specifies the base image to use for the new image. In this case, it's using the OpenJDK 14 image provided by adoptopenjdk.

The next line "EXPOSE 8080" tells Docker that the container will listen on port 8080.

The next line "ARG JAR_FILE=target/project-1.0-SNAPSHOT.jar" defines an argument for the Dockerfile. This argument is used to specify the location of the jar file that the application is built from, in this case the file is located in the target folder, with the name of "project-1.0-SNAPSHOT.jar".

The next line "ADD ${JAR_FILE} hotelManagementSystemProject.jar" adds the jar file specified in the previous argument to the image, and renames it to "hotelManagementSystemProject.jar".

The last line "ENTRYPOINT ["java","-jar","/hotelManagementSystemProject.jar"]" specifies the command that will be run when the container is started. This tells Docker to run the command "java -jar /hotelManagementSystemProject.jar" which will start the Java application.

When you build this image with this Dockerfile, a new image will be created that includes the Java application and all of its dependencies, and is configured to run the application when the container starts.

Amenity Related Business Logic

The class has three fields, ADMIN which is an integer with a value of 2, userService and amenityRepository which are both objects. userService is an object of the UserService class and amenityRepository is an object of the AmenityRepository class.

The class has a constructor which takes two arguments, userService and amenityRepository, and initializes the fields of the class with the same name. This is called constructor injection, where the dependencies are being passed through the constructor.

The class has four methods, createAmenity, modifyAmenity, deleteAmenity and getAllAmenities.

The createAmenity method takes two arguments, amenityName and callerUserId. It first calls the getUserType method of the userService object to get the user type of the caller. If the user type is equal to ADMIN, it then finds all the amenities from the repository, iterates through the list of amenities and checks if the given amenityName already exists. If it does not exist, it creates a new amenity with the given name and saves it to the repository and returns true. If the user type is not equal to ADMIN or the amenity name already exists, it returns false.

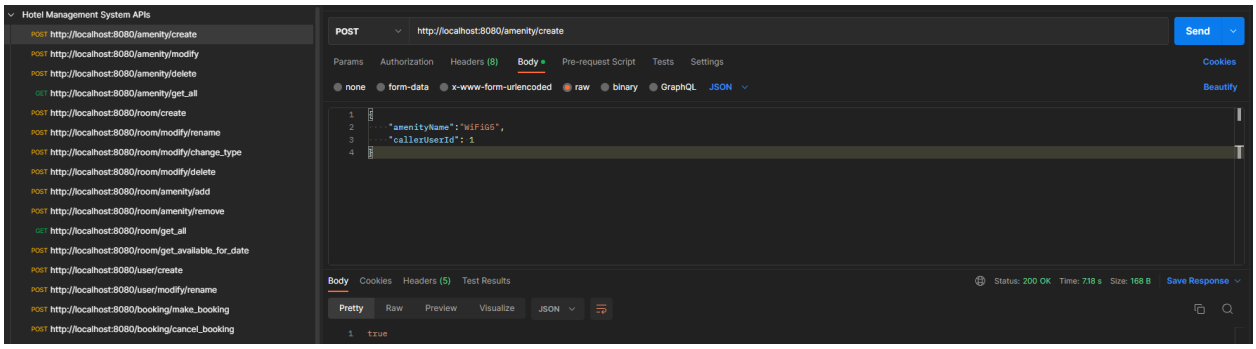
The modifyAmenity method takes three arguments, amenityName, amenityId and callerUserId. It first calls the getUserType method of the userService object to get the user type of the caller. If the user type is equal to ADMIN, it updates the amenity with the given id and name in the repository and returns true. If the user type is not equal to ADMIN, it returns false.

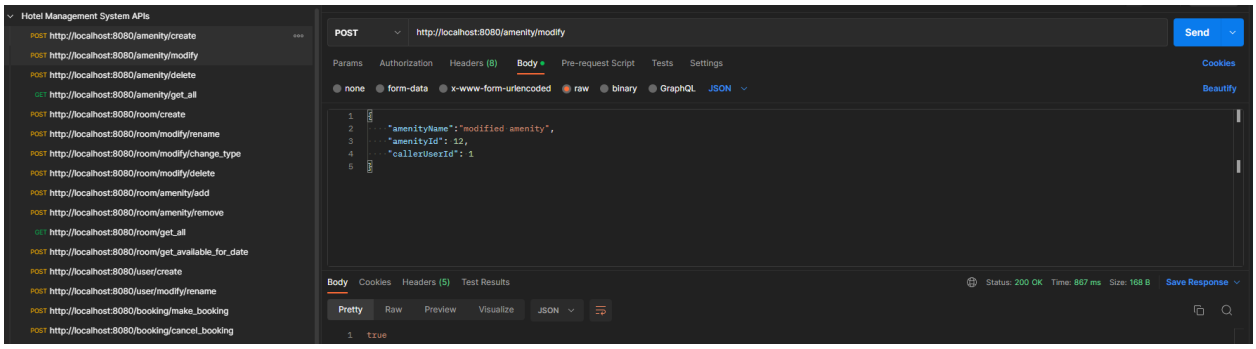
The deleteAmenity method takes two arguments, amenityId and callerUserId. It first calls the getUserType method of the userService object to get the user type of the caller. If the user type is equal to ADMIN, it deletes the amenity with the given id from the repository and returns true. If the user type is not equal to ADMIN, it returns false.

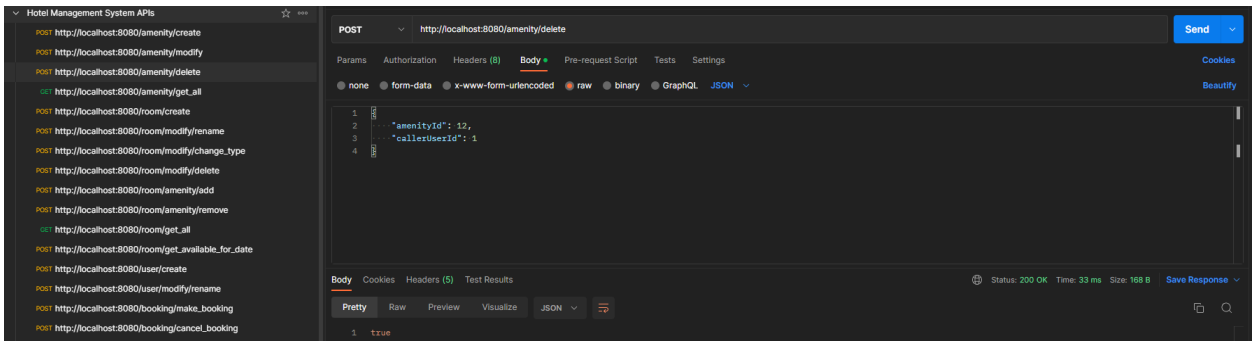
The getAllAmenities method returns all the amenities from the repository.

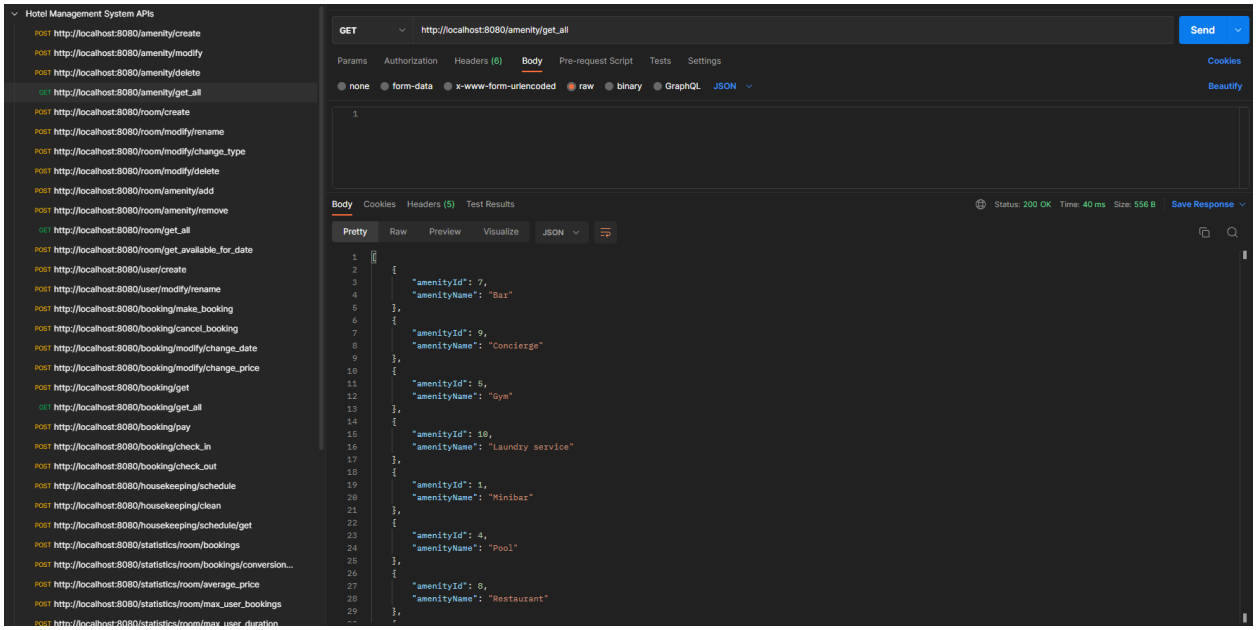
In general, this class provides methods to handle Amenities operations like creating, modifying and deleting amenities and also retrieve amenities. It also checks the user type of the caller to make sure that only admin users can perform these operations. The class uses the UserService class to check the user type and AmenityRepository class to perform CRUD operations on Amenities.

findAll(): This method returns all the Amenities from the database by executing a SELECT statement on the Amenities table, and using the AmenityRowMapper class to map the result set to Amenity objects.
findById(int amenityId): This method returns a single Amenity from the database by executing a SELECT statement on the Amenities table, using the amenityId as a parameter and using the AmenityRowMapper class to map the result set to Amenity objects.
save(Amenity amenity): This method saves a new Amenity to the database by executing an INSERT statement on the Amenities table, using the AmenityName as a parameter.
update(Amenity amenity): This method updates an Amenity in the database by executing an UPDATE statement on the Amenities table, using the AmenityName and AmenityId as parameters.
deleteById(int amenityId): This method deletes an Amenity from the database by executing a DELETE statement on the Amenities table, using the amenityId as a parameter.
AmenityRowMapper is an inner class which implements RowMapper interface. It maps the ResultSet obtained from the query to Amenity object.
In general, this class provides methods for performing CRUD operations on Amenities using JDBC template, which is an abstraction layer for JDBC that simplifies the use of JDBC and helps to avoid common errors. The class maps the ResultSet obtained from the query to Amenity object using the AmenityRowMapper class, which is an inner class inside the AmenityRepository class.


Hotel Management System APIs
POST http://localhost:8080/amenity/create
POST http://localhost:8080/amenity/modify
POST http://localhost:8080/amenity/delete
GET http://localhost:8080/amenity/get_all
POST http://localhost:8080/room/create
POST http://localhost:8080/room/modify/rename
POST http://localhost:8080/room/modify/change_type
POST http://localhost:8080/room/modify/delete
POST http://localhost:8080/room/amenity/add
POST http://localhost:8080/room/amenity/remove
GET http://localhost:8080/room/get_all
POST http://localhost:8080/room/get_available_for_date
POST http://localhost:8080/user/create
POST http://localhost:8080/user/modify/rename
POST http://localhost:8080/booking/make_booking
POST http://localhost:8080/booking/cancel_booking
POST http://localhost:8080/amenity/create
Send
Params Authorization Headers (8) Body Pre-request Script Tests Settings
Cookies
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Beautify
{
"amenityName":"WiFiG5",
"callerUserId": 1
}
Body Cookies Headers (5) Test Results
Status: 200 OK Time: 718 s Size: 168 B
Save Response
Pretty Raw Preview Visualize JSON
true


Hotel Management System APIs
POST http://localhost:8080/amenity/create
POST http://localhost:8080/amenity/modify
POST http://localhost:8080/amenity/delete
GET http://localhost:8080/amenity/get_all
POST http://localhost:8080/room/create
POST http://localhost:8080/room/modify/rename
POST http://localhost:8080/room/modify/change_type
POST http://localhost:8080/room/modify/delete
POST http://localhost:8080/room/amenity/add
POST http://localhost:8080/room/amenity/remove
GET http://localhost:8080/room/get_all
POST http://localhost:8080/room/get_available_for_date
POST http://localhost:8080/user/create
POST http://localhost:8080/user/modify/rename
POST http://localhost:8080/booking/make_booking
POST http://localhost:8080/booking/cancel_booking
POST http://localhost:8080/amenity/modify
Send
Params Authorization Headers (8) Body Pre-request Script Tests Settings
Cookies
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Beautify
{
"amenityName":"modified amenity",
"amenityId": 12,
"callerUserId": 1
}
Body Cookies Headers (5) Test Results
Status: 200 OK Time: 867 ms Size: 168 B
Save Response
Pretty Raw Preview Visualize JSON
true


Hotel Management System APIs
POST http://localhost:8080/amenity/create
POST http://localhost:8080/amenity/modify
POST http://localhost:8080/amenity/delete
GET http://localhost:8080/amenity/get_all
POST http://localhost:8080/room/create
POST http://localhost:8080/room/modify/rename
POST http://localhost:8080/room/modify/change_type
POST http://localhost:8080/room/modify/delete
POST http://localhost:8080/room/amenity/add
POST http://localhost:8080/room/amenity/remove
GET http://localhost:8080/room/get_all
POST http://localhost:8080/room/get_available_for_date
POST http://localhost:8080/user/create
POST http://localhost:8080/user/modify/rename
POST http://localhost:8080/booking/make_booking
POST http://localhost:8080/booking/cancel_booking
POST http://localhost:8080/amenity/delete
Send
Params Authorization Headers (8) Body Pre-request Script Tests Settings
Cookies
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Beautify
{
"amenityId": 12,
"callerUserId": 1
}
Body Cookies Headers (5) Test Results
Status: 200 OK Time: 33 ms Size: 168 B
Save Response
Pretty Raw Preview Visualize JSON
true



Room Related Business Logic

The class RoomService is a service class in Java that is annotated with @Service. The @Service annotation is a stereotype annotation that indicates that the class is a service class and is managed by the Spring framework.

The class has four fields, ADMIN which is an integer with a value of 2, userService, roomRepository, roomAmenityRepository, and bookingRepository which are all objects. userService is an object of the UserService class, roomRepository is an object of the RoomRepository class, roomAmenityRepository is an object of the RoomAmenityRepository class, and bookingRepository is an object of the BookingRepository class.

The class has a constructor which takes four arguments, userService, roomRepository, roomAmenityRepository, and bookingRepository, and initializes the fields of the class with the same name. This is called constructor injection, where the dependencies are being passed through the constructor.

The class has several methods:

createRoom: This method creates a new room in the database, it takes three arguments roomName, roomTypeId, and callerUserId. It first calls the getUserType method of the userService object to get the user type of the caller. If the user type is equal to
ADMIN, it then finds all the rooms from the repository, iterates through the list of rooms and checks if the given roomName or roomTypeId already exists. If it does not exist, it creates a new room with the given name, type and saves it to the repository and returns true. If the user type is not equal to ADMIN or the room name or type already exists, it returns false.

renameRoom: This method renames an existing room in the database, it takes three arguments roomId, newName, and callerUserId. It first calls the getUserType method of the userService object to get the user type of the caller. If the user type is equal to ADMIN, it then finds all the rooms from the repository, iterates through the list of rooms and checks if the given newName already exists. If it does not exist, it updates the room with the given roomId with the newName in the repository and returns true. If the user type is not equal to ADMIN or the room name already exists, it returns false.

changeRoomType: This method changes the type of a room in the database, it takes three arguments roomId, roomTypeId, and callerUserId. It first calls the getUserType method of the userService object to get the user type of the caller. If the user type is equal to ADMIN, it updates the room type with the given id and roomId in the repository and returns true. If the user type is not equal to ADMIN, it returns false.

deleteRoom: This method deletes a room from the database, it takes two arguments roomId and callerUserId. It first calls the getUserType method of the userService object to get the user type of the caller. If the user type is equal to ADMIN, it first deletes the amenities associated with the roomId from the roomAmenity repository, deletes the room with the given id from the room repository, and returns true. If the user type is not equal to ADMIN, it returns false.

addAmenityToRoom: This method adds an amenity to a room, it takes three arguments roomId, amenityId, and callerUserId. It first calls the getUserType method of the userService object to get the user type of the caller. If the user type is equal to ADMIN, it finds all the room amenities from the repository, iterates through the list of room amenities and checks if the given amenityId and roomId already exists. If it does not exist, it creates a new room amenity with the given roomId and amenityId and saves it to the repository and returns true. If the user type is not equal to ADMIN or the amenityId and roomId already exists, it returns false.

removeAmenityFromRoom: This method removes an amenity from a room, it takes three arguments roomId, amenityId, and callerUserId. It first calls the getUserType method of the userService object to get the user type of the caller. If the user type is equal to ADMIN, it deletes the room amenity from the repository based on the given roomId and amenityId and returns true. If the user type is not equal to ADMIN, it returns false.

getAllRooms: This method returns a list of all the rooms from the repository

getAvailableRoomsForDate: This method returns a list of available rooms for the given date range.This Java method takes in two strings, dayStart and dayEnd, that represent a date range. It converts these strings to Date objects, startDate and endDate. It then retrieves all bookings from a bookingRepository and initializes two empty lists, resRooms and roomMaps.

It then iterates through each booking, creating a RoomMap object for each one, setting the roomId to the id of the room associated with the booking and setting the stat to a boolean value based on whether the end date of the booking is before the start date of the desired range or the start date of the booking is after the end date of the desired range.

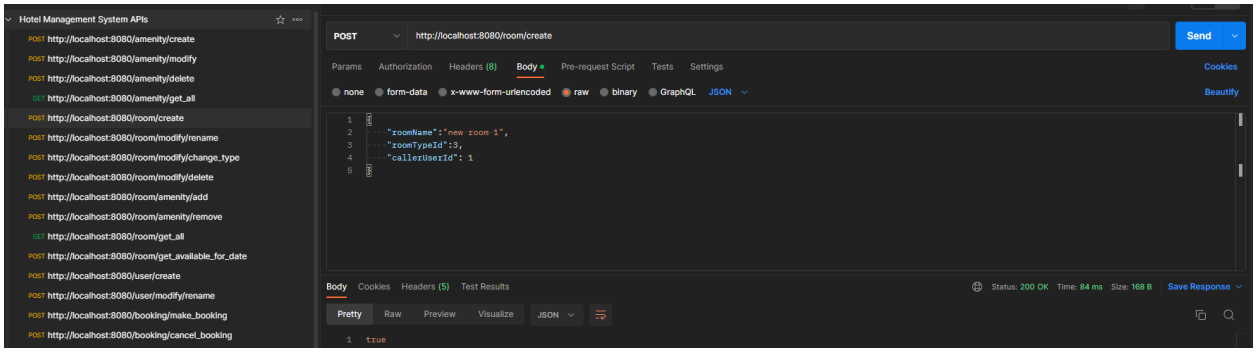
It then groups the RoomMap objects by roomId and iterates through each group, checking if the stat of all RoomMap objects in the group is true. If it is, the roomId is added to a list of roomIds. Finally, it iterates through the list of roomIds, adding the room associated with each id to the resRooms list, and returns this list of available rooms.

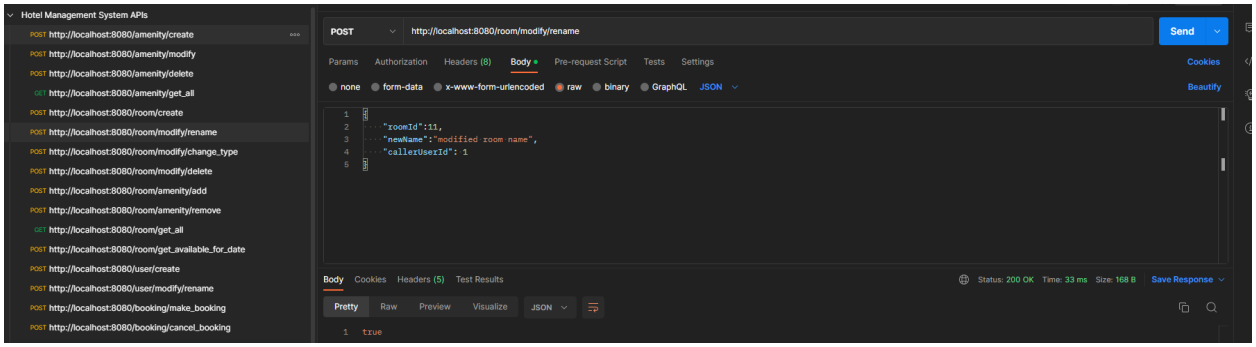
```java
public List<Room> getAvailableRoomsForDate(String dayStart, String dayEnd) {
    Date startDate = new Date(Long.valueOf(dayStart));
    Date endDate = new Date(Long.valueOf(dayEnd));

    List<Booking> bookings  = this.bookingRepository.findAll();
    List<Room> resRooms = new ArrayList<>();
    List<RoomMap> roomMaps = new ArrayList<>();

    for (Booking booking: bookings) {
        RoomMap roomMap = new RoomMap();
        roomMap.setRoomId(this.roomRepository.findById(booking.getRoom()).getRoomId());
        roomMap.setStat((booking.getEndDate().compareTo(startDate) < 0 || booking.getStartDate().compareTo(endDate) > 0));
        roomMaps.add(roomMap);
    }
    List<Integer> roomIds = new ArrayList<>();

    roomMaps.stream() Stream<RoomMap>
            .collect(Collectors.groupingBy(RoomMap::getRoomId)) Map<Integer, List<RoomMap>>
            .forEach((id, roomMapList) -> {
                boolean stat = roomMapList.stream() Stream<RoomMap>
                        .map(RoomMap::getStat) Stream<Boolean>
                        .reduce( identity: true, (a, b) -> a && b);
                if (stat) {
                    roomIds.add(id);
                }
            });
    // if endDate of the booked room is before the required startDate OR
    // if startDate of the booked room is after the required endDate.
      resRooms.add(this.roomRepository.findById(booking.getRoom()));
    for (Integer roomId: roomIds) {
        resRooms.add(this.roomRepository.findById(roomId));
    }
    return resRooms;
}
```

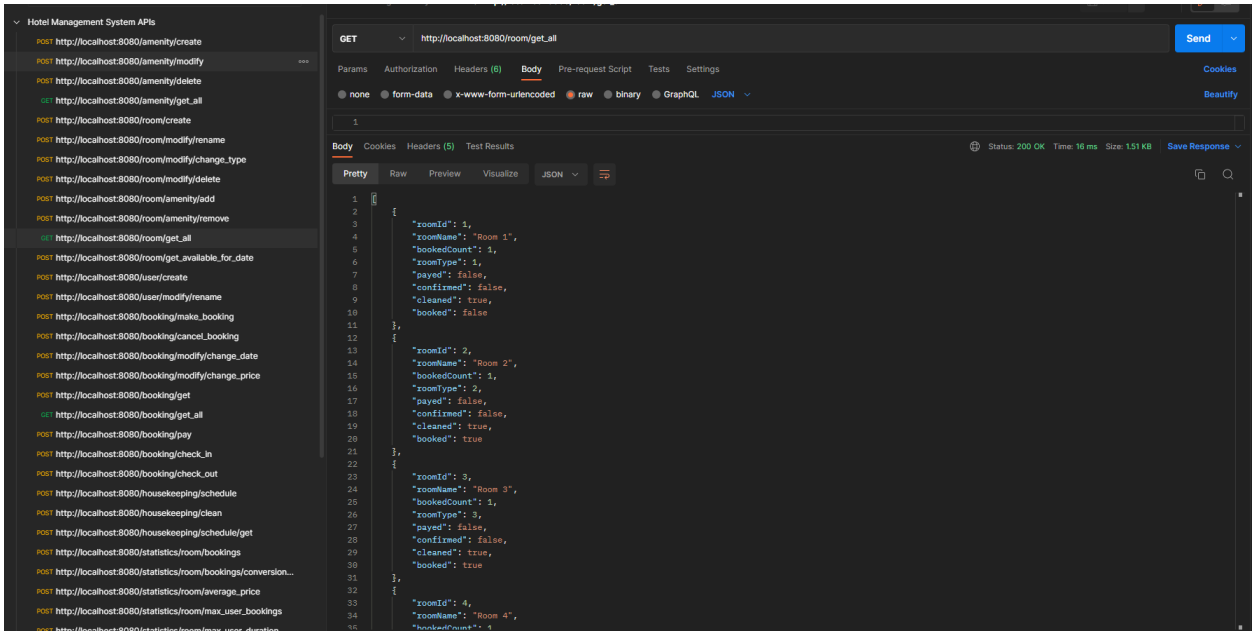
The RoomService class is responsible for performing various operations on rooms and the room amenities, such as creating, renaming, deleting, and updating rooms and amenities. It also provides methods for retrieving all rooms and available rooms for a given date range.
It's important to note that all methods in the RoomService class check if the user making the request is an admin before performing any operation. This is done by calling the getUserType method on the userService object and checking if the user type is equal to ADMIN. This is a simple form of access control, ensuring that only authorized users can perform certain operations on the data.

Additionally, the RoomService class makes use of several other classes and repositories to perform its operations such as UserService, RoomRepository, RoomAmenityRepository and BookingRepository . These classes and repositories are responsible for interacting with the underlying data storage, whether it be a database, file system, or other forms of storage.

It's worth noting that the RoomService class does not contain any JDBC specific code and does not directly interact with the database. Instead it relies on the RoomRepository, RoomAmenityRepository, BookingRepository and UserService classes to perform the database operations. This allows for better separation of concerns and makes the service class more testable, maintainable and flexible.


Hotel Management System APIs
POST http://localhost:8080/amenity/create
POST http://localhost:8080/amenity/modify
POST http://localhost:8080/amenity/delete
GET http://localhost:8080/amenity/get_all
POST http://localhost:8080/room/create
POST http://localhost:8080/room/modify/rename
POST http://localhost:8080/room/modify/change_type
POST http://localhost:8080/room/modify/delete
POST http://localhost:8080/room/amenity/add
POST http://localhost:8080/room/amenity/remove
GET http://localhost:8080/room/get_all
POST http://localhost:8080/room/get_available_for_date
POST http://localhost:8080/user/create
POST http://localhost:8080/user/modify/rename
POST http://localhost:8080/booking/make_booking
POST http://localhost:8080/booking/cancel_booking
POST http://localhost:8080/room/create
Send
Params Authorization Headers (8) Body Pre-request Script Tests Settings
Cookies
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Beautify
{
"roomName":"new room 1",
"roomTypeId":3,
"callerUserId": 1
}
Body Cookies Headers (5) Test Results
Status: 200 OK Time: 84 ms Size: 168 B
Save Response
Pretty Raw Preview Visualize JSON
true


Hotel Management System APIs
POST http://localhost:8080/amenity/create
POST http://localhost:8080/amenity/modify
POST http://localhost:8080/amenity/delete
GET http://localhost:8080/amenity/get_all
POST http://localhost:8080/room/create
POST http://localhost:8080/room/modify/rename
POST http://localhost:8080/room/modify/change_type
POST http://localhost:8080/room/modify/delete
POST http://localhost:8080/room/amenity/add
POST http://localhost:8080/room/amenity/remove
GET http://localhost:8080/room/get_all
POST http://localhost:8080/room/get_available_for_date
POST http://localhost:8080/user/create
POST http://localhost:8080/user/modify/rename
POST http://localhost:8080/booking/make_booking
POST http://localhost:8080/booking/cancel_booking
POST http://localhost:8080/room/modify/rename
Send
Params Authorization Headers (8) Body Pre-request Script Tests Settings
Cookies
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Beautify
{
"roomId":11,
"newName":"modified room name",
"callerUserId": 1
}
Body Cookies Headers (5) Test Results
Status: 200 OK Time: 33 ms Size: 168 B
Save Response
Pretty Raw Preview Visualize JSON
true


Hotel Management System APIs
POST http://localhost:8080/amenity/create
POST http://localhost:8080/amenity/modify
POST http://localhost:8080/amenity/delete
GET http://localhost:8080/amenity/get_all
POST http://localhost:8080/room/create
POST http://localhost:8080/room/modify/rename
POST http://localhost:8080/room/modify/change_type
POST http://localhost:8080/room/modify/delete
POST http://localhost:8080/room/amenity/add
POST http://localhost:8080/room/amenity/remove
GET http://localhost:8080/room/get_all
POST http://localhost:8080/room/get_available_for_date
POST http://localhost:8080/user/create
POST http://localhost:8080/user/modify/rename
POST http://localhost:8080/booking/make_booking
POST http://localhost:8080/booking/cancel_booking
POST http://localhost:8080/booking/modify/change_date
POST http://localhost:8080/booking/modify/change_price
POST http://localhost:8080/booking/get
GET http://localhost:8080/booking/get_all
POST http://localhost:8080/booking/pay
POST http://localhost:8080/booking/check_in
POST http://localhost:8080/booking/check_out
POST http://localhost:8080/housekeeping/schedule
POST http://localhost:8080/housekeeping/clean
POST http://localhost:8080/housekeeping/schedule/get
POST http://localhost:8080/statistics/room/bookings
POST http://localhost:8080/statistics/room/bookings/conversion...
POST http://localhost:8080/statistics/room/average_price
POST http://localhost:8080/statistics/room/max_user_bookings
GET http://localhost:8080/room/get_all
Send
Params Authorization Headers (6) Body Pre-request Script Tests Settings
Cookies
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Beautify
Body Cookies Headers (5) Test Results
Status: 200 OK Time: 16 ms Size: 1.51 KB
Save Response
Pretty Raw Preview Visualize JSON
[
{
"roomId": 1,
"roomName": "Room 1",
"bookedCount": 1,
"roomType": 1,
"payed": false,
"confirmed": false,
"cleaned": true,
"booked": false
},
{
"roomId": 2,
"roomName": "Room 2",
"bookedCount": 1,
"roomType": 2,
"payed": false,
"confirmed": false,
"cleaned": true,
"booked": true
},
{
"roomId": 3,
"roomName": "Room 3",
"bookedCount": 1,
"roomType": 3,
"payed": false,
"confirmed": false,
"cleaned": true,
"booked": true
},
{
"roomId": 4,
"roomName": "Room 4",

User Related Business Logic

We create the admin user while bootstrapping the project, we cannot create the admin user from the program.

The UserService class is a service class that provides methods for performing operations on users. The class has the following methods:

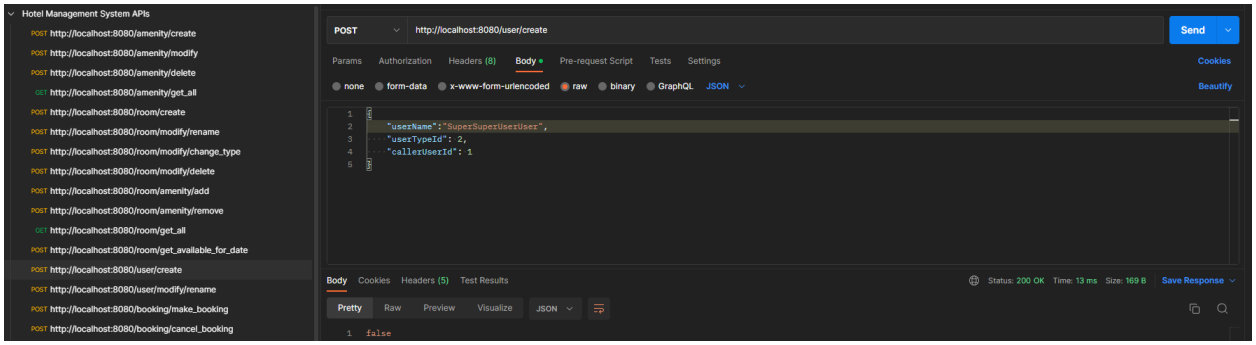
createUser: This method creates a new user in the database, it takes three arguments userName, userTypeId, and callerUserId. It first calls the getUserType method on UserService object to get the user type of the caller. If the user type is equal to ADMIN and the userTypeId is not equal to ADMIN, it then finds all the users from the repository, iterates through the list of users and checks if the given userName and userTypeId already exists. If it does not exist, it creates a new user with the given name, type, and default password and saves it to the repository and returns true. If the user type is not equal to ADMIN or the user name or type already exists or the new user is admin, it returns false.

renameUser: This method renames an existing user in the database, it takes three arguments userId, newName, and callerUserId. It first calls the getUserType method of the UserService object to get the user type of the caller. If the user type is equal to ADMIN, it updates the name of the user in the repository with the new name and returns true. Else it returns false.

getAllUsers: This method returns a list of all users from the repository.

getUserType: This method returns the user type of the user corresponding to the given user id.

It's important to note that the createUser method only allows the creation of new users if the caller is an admin and the new user is not an admin. This is a simple form of access control, ensuring that only authorized users can perform certain operations on the data. Additionally, the UserService class makes use of UserRepository to perform its operations. This class is responsible for interacting with the underlying data storage, whether it be a database, file system, or other forms of storage.



Booking Related Business Logic

The BookingService class is a service class that is responsible for handling all the operations related to creating, canceling, updating, and retrieving bookings. This class has several methods, including:

makeBooking(int userId, String dayStart, String dayEnd, int price,int guestCount, int callerUserId): This method is used to create a new booking for a user. The method takes several parameters such as userId, the start and end date of the booking, the price, the number of guests and the caller user Id. It checks if the caller is either an Admin or a Guest and then it checks if any rooms are available. If a room is available it sets the room as booked and then creates a new booking with the provided details.

```
public boolean makeBooking(int userId, String dayStart, String dayEnd, int price,int guestCount, int callerUserId,List<Integer> roomIdList) {
    Date startDate = new Date(Long.valueOf(dayStart));
    Date endDate = new Date(Long.valueOf(dayEnd));
    int userType = this.userService.getUserType(callerUserId);

    if (userType == ADMIN || userType == GUEST) {
        // find the available rooms in a given date period
        List<Room> availableRooms  = this.roomService.getAvailableRoomsForDate(dayStart,dayEnd);
        int roomCapacityCount = 0;
        if (availableRooms.containsAll(roomIdList)){
            for (Integer oneRoomId: roomIdList) {
                roomCapacityCount += this.roomRepository.getRoomCapacity(this.roomRepository.findById(oneRoomId).getRoomType());
            }
            if (guestCount <= roomCapacityCount){
                for (Integer oneRoom: roomIdList) {
                    this.bookingRepository.save(new Booking(oneRoom, userId, startDate, endDate, price,  isPayed: false, guestCount,  stat: 3));
                    return true;
                }
            }
        }

    }
    return false;
}
```

cancelBooking(int bookingId, int callerUserId): This method is used to cancel a booking. It takes the bookingId and the caller user Id as input. It checks if the caller is an Admin, then it sets the corresponding bookings stat as canceled.

```
public boolean cancelBooking(int bookingId, int callerUserId) {
    int userType = this.userService.getUserType(callerUserId);
    if ( userType == ADMIN){
        if (!this.bookingRepository.findById(bookingId).isPayed()){
            this.bookingRepository.cancelBooking(bookingId);
            return true;
        }
    }
    return false;
}
```

changeBookingDate(int bookingId, String dayStart, String dayEnd, int callerUserId): This method is used to update the start and end date of a booking. It takes the bookingId, the new start and end date and the caller user Id as input. It checks if the caller is either an Admin or a Receptionist, then it updates the date of the booking.

changeBookingPrice(int bookingId, int newPrice, int callerUserId): This method is used to update the price of a booking. It takes the bookingId, the new price and the caller user Id as input. It checks if the caller is either an Admin or a Receptionist, then it updates the price of the booking.

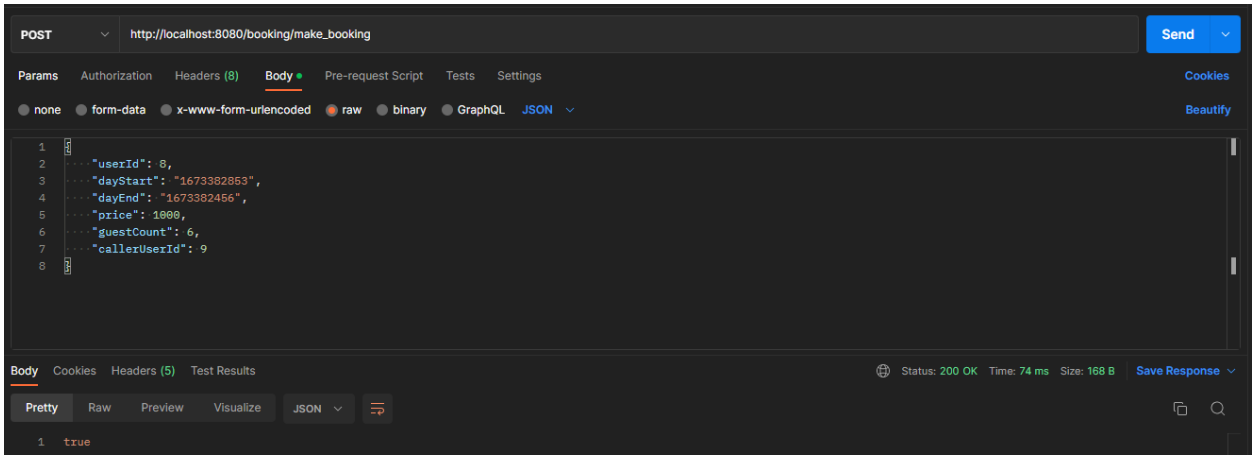
getBooking(int bookingId, int callerUserId): This method is used to retrieve a booking by its Id. It takes the bookingId and the caller user Id as input. It checks if the caller is either an Admin or a Receptionist, then it returns the booking.

getAllBookings(): This method is used to retrieve all the bookings. It doesn't require any input and returns a list of all the bookings in the system.

pay(int bookingId, int callerUserId): This method is used to mark a booking as paid. It takes the bookingId and the caller user Id as input. It checks if the caller is either an Admin or a Receptionist, then it sets the corresponding booking as paid.

The checkin method is used to mark a booking as checked in. It takes the bookingId and the callerUserId as inputs, and returns a boolean value indicating whether the check in was successful or not. The method first checks that the caller is a receptionist or an admin, and if so, it sets the isCheckedIn field of the booking with the corresponding bookingId to true in the database. If the check in was successful, it returns true, otherwise it returns false.

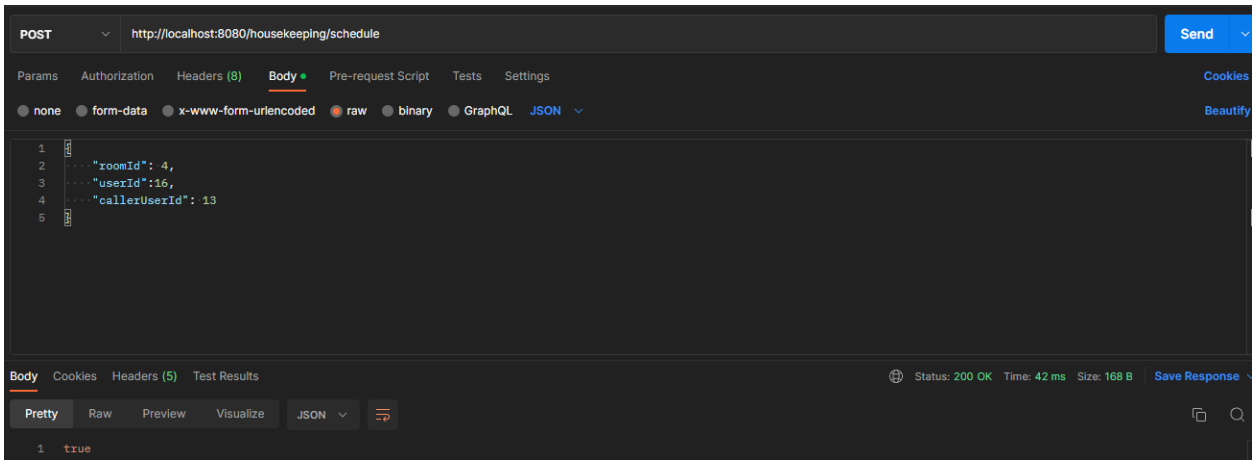
The checkout method is used to mark a booking as checked out. It takes the bookingId and the callerUserId as inputs, and returns a boolean value indicating whether the checkout was successful or not. The method first checks that the caller is a receptionist or an admin, and if so, it sets the isCheckedOut field of the booking with the corresponding bookingId to true in the database. If the checkout was successful, it returns true, otherwise it returns false. It also set the room status as available and update the room status in the database.



HouseKeeping Related Business Logic

The HousekeepingService class is a service class that provides functionality for scheduling cleaning of rooms and marking them as cleaned. It has several methods:

scheduleCleaning(int roomId, int userId, int callerUserId): This method schedules cleaning for a specific room, with the specified user responsible for cleaning it. It takes in the ID of the room to be cleaned, the ID of the user responsible for cleaning it, and the ID of the user making the request (the "caller"). The method checks the user type of the caller and allows the request only if the caller is an ADMIN or a RECEPTIONIST. It then saves the schedule in the HousekeepingSchedulesRepository and returns true if the schedule is saved successfully.
clean(int roomId, int callerUserId): This method marks a specific room as cleaned. It takes in the ID of the room and the ID of the user making the request. The method checks the user type of the caller and allows the request only if the caller is an ADMIN or a RECEPTIONIST. It then updates the 'isCleaned' field of the room in the RoomRepository and returns true if the update is successful.
getCleaningSchedule(int callerUserId): This method returns a list of rooms that are scheduled for cleaning. It takes in the ID of the user making the request. The method checks the user type of the caller and allows the request only if the caller is an ADMIN or a RECEPTIONIST. It then finds all the schedules in the HousekeepingSchedulesRepository that are set to '1' and returns the corresponding rooms from the RoomRepository.
All these methods are used to perform housekeeping related functionalities such as scheduling cleaning, marking room as cleaned and getting the cleaning schedule. The method does not allow to perform any operation unless the caller is an admin or receptionist.

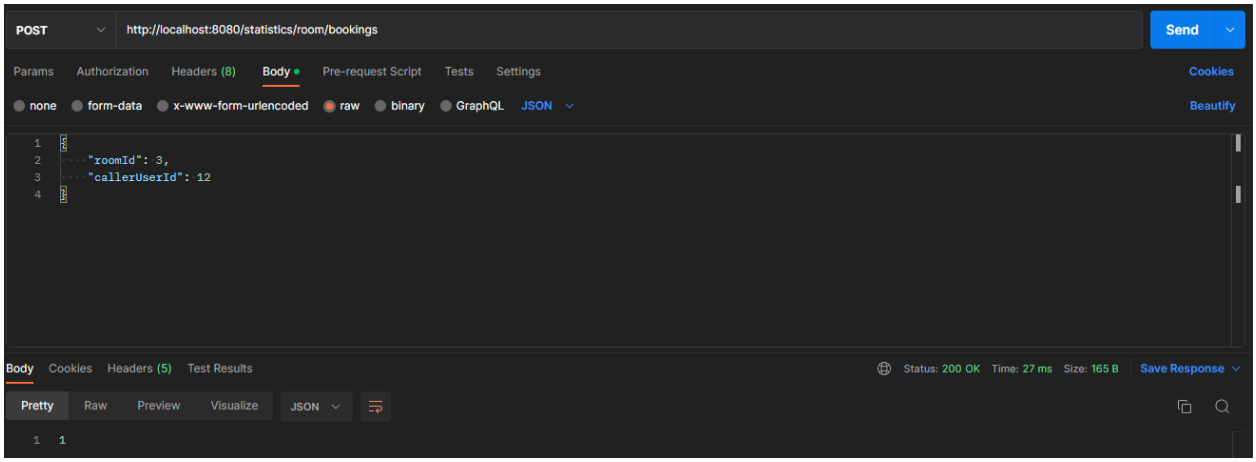


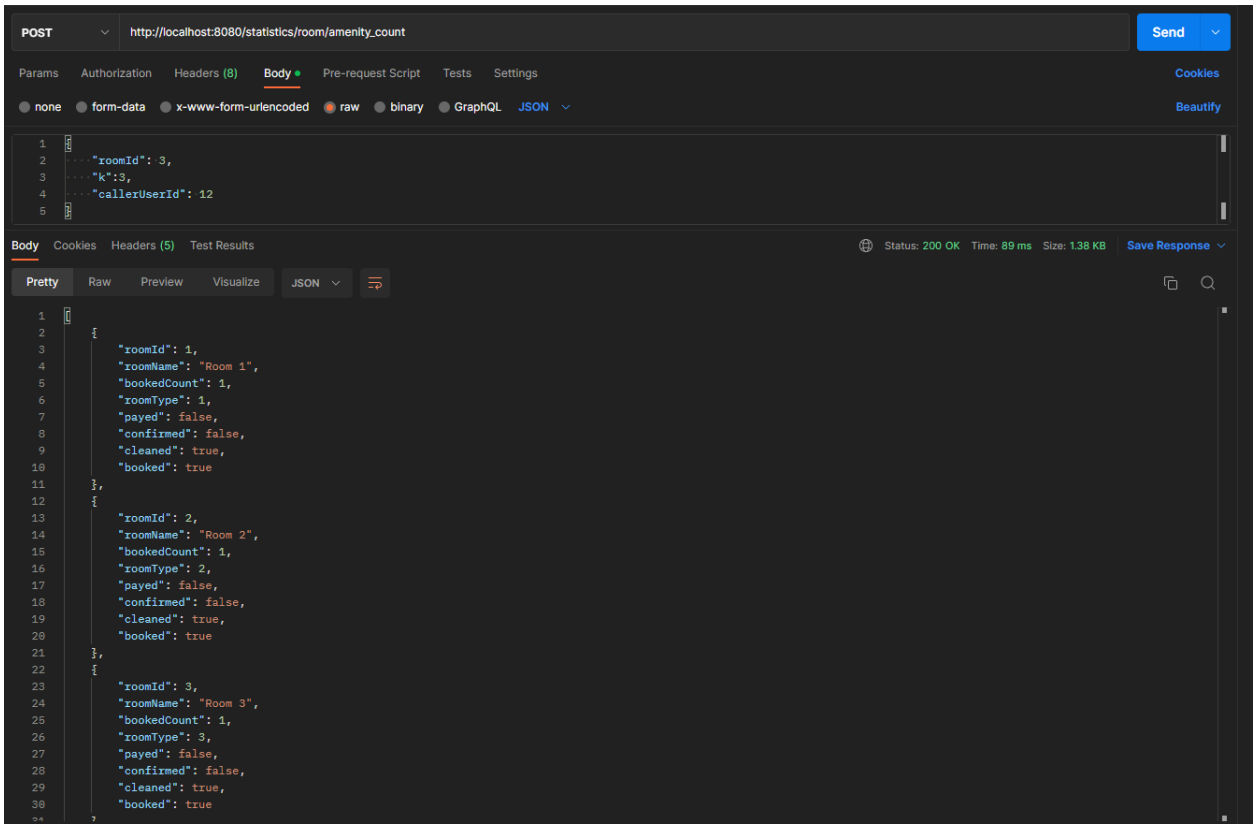
Statistics Related Business Logic

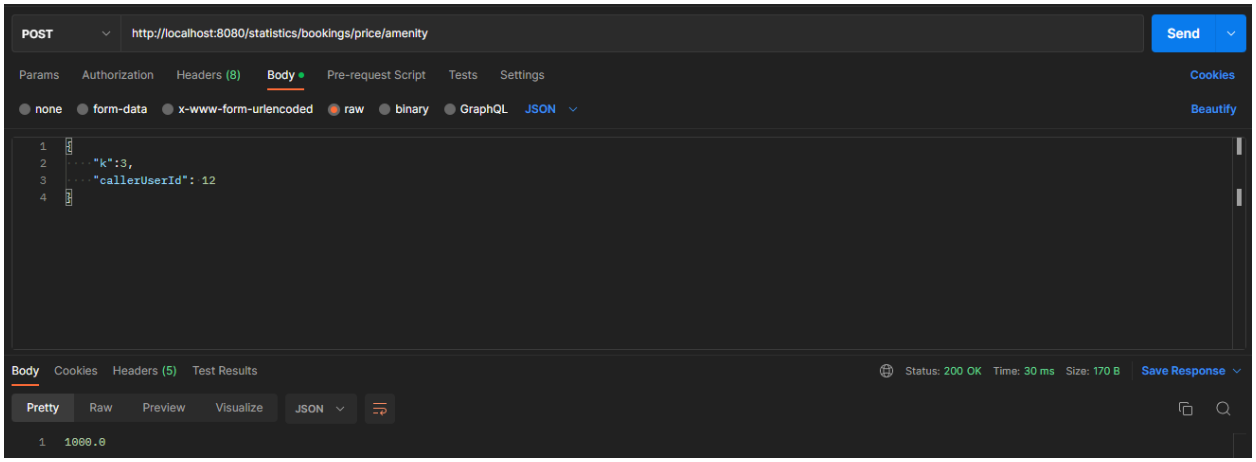
This is a Java class that provides various statistics for a hotel management system. The class has several methods, each of which returns a specific type of statistic. The methods include:

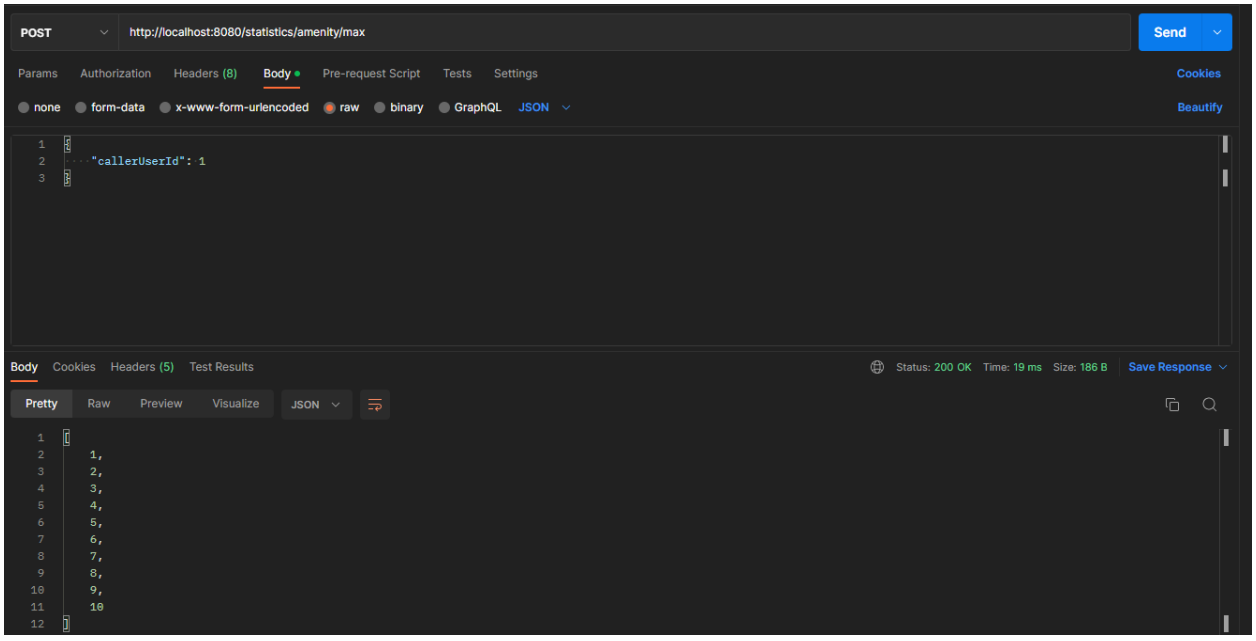
getBookingCount(int roomId, int callerUserId): returns the number of bookings for a specific room, based on the caller's user ID.
getConversionRate(int roomId, int callerUserId): returns the conversion rate of bookings, which is the ratio of bookings that were not canceled to the total number of bookings.
getAveragePrice(int roomId, int callerUserId): returns the average price of a booking for a specific room, based on the caller's user ID.
getMaxDurationUserId(int roomId, int callerUserId): returns the user ID of the person who booked the room for the longest duration.
getRoomsWithAmenityCount(int roomId, int k, int callerUserId): returns a list of rooms that have at least k amenities.
getUserIdWithMostBookingsForRoom(int roomId, int callerUserId): returns the user ID of the person who made the most bookings for a specific room.
getUserIdWithLongestDurationInRoom(int roomId, int callerUserId): returns the user ID of the person who stayed in a specific room for the longest duration.
getRoomsWithAtLeastKAmenities(int roomId, int k, int callerUserId): returns a list of rooms that have at least k amenities.
getTotalNumberOfFinishedHousekeepingTasks(int callerUserId): returns the total number of finished housekeeping tasks for a hotel.
It uses several Repository classes for interacting with the database. It also uses a UserService class for checking the user permission before returning the statistics data.
This is a Java class that defines a repository for interacting with the housekeeping schedules in a hotel management system. It uses the Spring Framework's JdbcTemplate to interact with a database and perform various CRUD operations on the housekeeping schedules. The methods include:

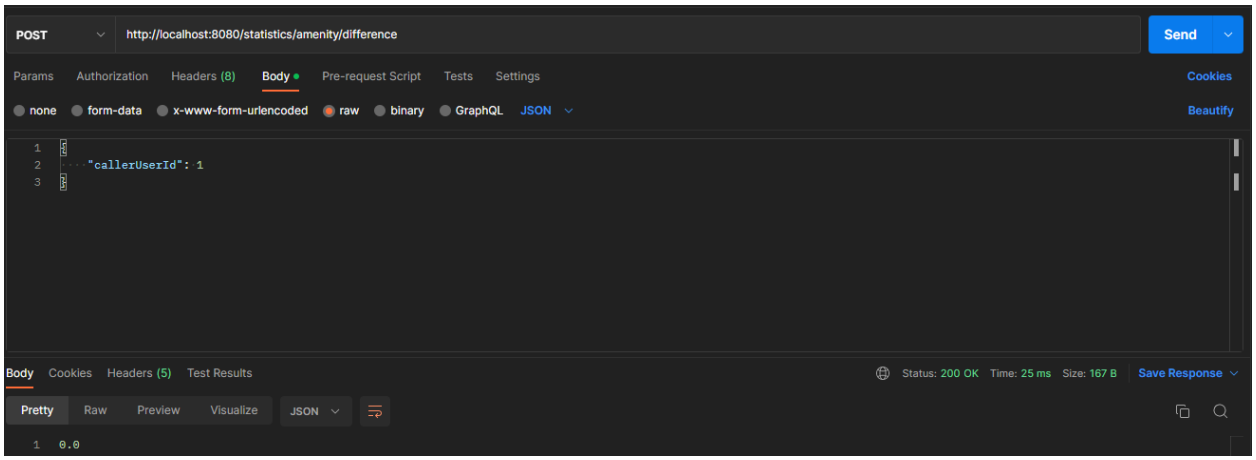
findAll(): returns a list of all housekeeping schedules in the database
findAllStat1(): returns a list of all housekeeping schedules with a specific status.
findById(int scheduleId): returns the housekeeping schedule with the specified ID.
save(HousekeepingSchedule schedule): saves a new housekeeping schedule to the database.
update(HousekeepingSchedule schedule): updates an existing housekeeping schedule in the database.
deleteById(int scheduleId): deletes the housekeeping schedule with the specified ID from the database.
housekeeperWithMostPendingTasks(): returns the ID of the housekeeper who has the most pending tasks.
roomsWithHighestNumberOfDifferentHousekeepers(): returns a list of rooms that have been cleaned by the highest number of different housekeepers.
It uses HousekeepingScheduleRowMapper class for mapping the result set into HousekeepingSchedule object.
It also uses another Repository class HKScheduleStatusTypesRepository for interacting with the status types of housekeeping schedules.


POST
http://localhost:8080/statistics/room/bookings
Send
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
{
"roomId": 3,
"callerUserId": 12
}
Body
Cookies
Headers (5)
Test Results
Status: 200 OK
Time: 27 ms
Size: 165 B
Save Response
Pretty
Raw
Preview
Visualize
JSON
1


POST
http://localhost:8080/statistics/room/amenity_count
Send
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
{
"roomId": 3,
"k":3,
"callerUserId": 12
}
Body
Cookies
Headers (5)
Test Results
Status: 200 OK
Time: 89 ms
Size: 1.38 KB
Save Response
Pretty
Raw
Preview
Visualize
JSON
[
{
"roomId": 1,
"roomName": "Room 1",
"bookedCount": 1,
"roomType": 1,
"payed": false,
"confirmed": false,
"cleaned": true,
"booked": true
},
{
"roomId": 2,
"roomName": "Room 2",
"bookedCount": 1,
"roomType": 2,
"payed": false,
"confirmed": false,
"cleaned": true,
"booked": true
},
{
"roomId": 3,
"roomName": "Room 3",
"bookedCount": 1,
"roomType": 3,
"payed": false,
"confirmed": false,
"cleaned": true,
"booked": true


POST
http://localhost:8080/statistics/bookings/price/amenity
Send
Params Authorization Headers (8) Body Pre-request Script Tests Settings
Cookies
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Beautify
{
"k":3,
"callerUserId": 12
}
Body Cookies Headers (5) Test Results
Status: 200 OK Time: 30 ms Size: 170 B
Save Response
Pretty Raw Preview Visualize JSON
1000.0


POST
http://localhost:8080/statistics/amenity/max
Send
Params Authorization Headers (8) Body Pre-request Script Tests Settings
Cookies
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Beautify
{
"callerUserId": 1
}
Body Cookies Headers (5) Test Results
Status: 200 OK Time: 19 ms Size: 186 B
Save Response
Pretty Raw Preview Visualize JSON
[
1,
2,
3,
4,
5,
6,
7,
8,
9,
10
]


POST
http://localhost:8080/statistics/amenity/difference
Send
Params Authorization Headers (8) Body Pre-request Script Tests Settings
Cookies
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Beautify
{
"callerUserId": 1
}
Body Cookies Headers (5) Test Results
Status: 200 OK Time: 25 ms Size: 167 B
Save Response
Pretty Raw Preview Visualize JSON
0.0

SQL Injection Prevention

To prevent SQL injection in our application using JDBC, we used prepared statements. A prepared statement is a pre-compiled SQL statement that can be reused with different input parameters.
JdbcTemplate automatically uses prepared statements concept in order to prevent the SQL injection for example in our userCreate() method:

```
public void save(User user) {
    String sql = "INSERT INTO Users(Username, Password, UserSalary, UserType) VALUES(?, ?, ?, ?)";
    //Encrypt the password
    String encryptedPassword = BCrypt.hashpw(user.getPassword(), BCrypt.gensalt());
    jdbcTemplate.update(sql, user.getUsername(), encryptedPassword, user.getUserSalary(), user.getUserType());
}
```

In the above method: the '?' characters in the SQL query are placeholders for input parameters. When the query is executed, the input parameters are automatically escaped and enclosed in single quotes, so they are treated as string literals rather than part of the SQL syntax. This makes it impossible for an attacker to inject malicious SQL code into the query. And also we encrypt the password while saving it into our DB.

We can decrypt it when we need to using below simple logic as:

```
if (BCrypt.checkpw(password, encryptedPassword)) {
      //password is correct
} else {
      //password is incorrect
 }
```

To run project, we:

→ Install JDK 14.0.2.
→ Install Intellij or any other IDE, run spring-boot application in Intellij.
→ Configure the JDK on your PC.

NOTES:

- Disable the compiler argument called “**enable-preview**” on pom.xml, there were some environmental issue between runtime java version and compile time java version and according to our research, we had to disable this compiler arg.

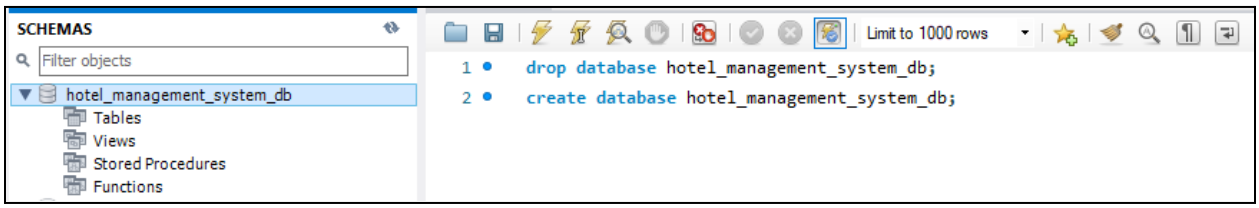
```
<configuration>
    <release>${java.version}</release>
        <compilerArgs>-->
            <arg>&#45;&#45;enable-preview</arg>-->
        </compilerArgs>-->
    <forceJavacCompilerUse>true</forceJavacCompilerUse>
    <parameters>true</parameters>
    <source>${java.version}</source>
    <target>${java.version}</target>
    <compilerArgs>
        <arg>-Amapstruct.suppressGeneratorTimestamp=true</arg>
        <arg>-Amapstruct.defaultComponentModel=spring</arg>
    </compilerArgs>
</configuration>
```

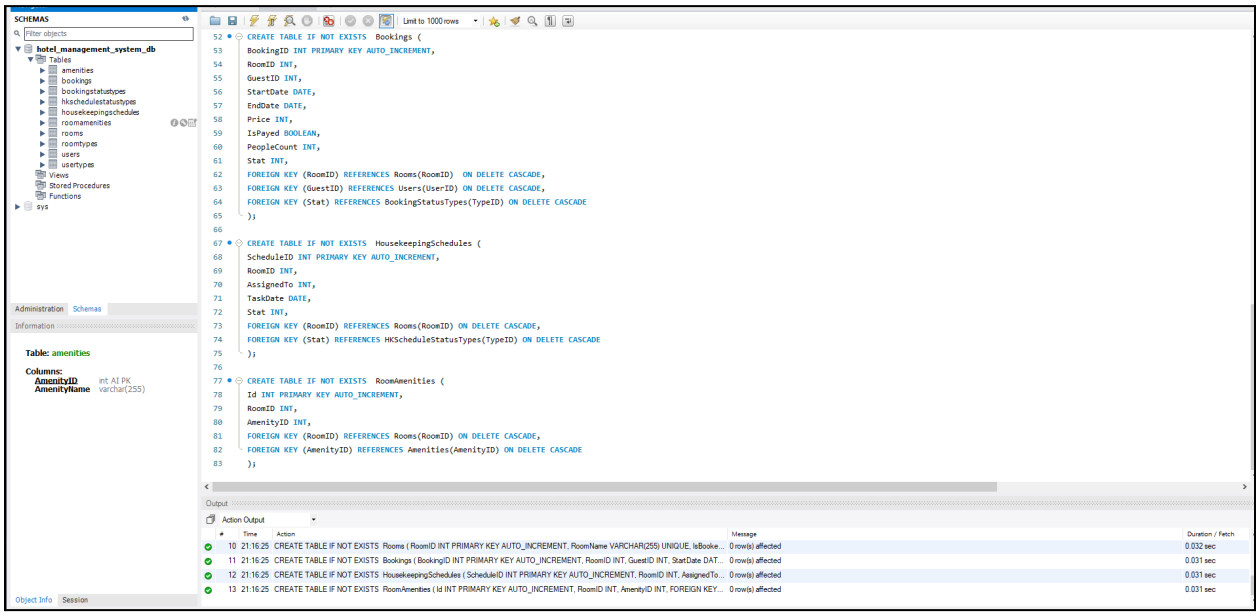
- UserPassword is used to implement SQL injection in the project.
- We added userSalary parameter into the Users table, according to our research about hotel management systems, users should have userSalary info in them, it does not have any other functionalities.
- We changed most of the parts of the Room table, in the previous version we stored the data about cleaning, booking status, etc.. in the Room table. Current version stores roomID, roomName, roomType and cleaning, paying, booking data stored in Booking Table. They all updated in the ER Diagram.

Docker Image

We run, “docker build -t springallicationdemo .”, in the terminal. Then we run “docker image”' to find the desired image ID, and finally, run “docker run -port8080:8080 [first 3 character of the corresponding image ID”' to start the spring boot application on docker hub

Schema, DDLs, DMLs





Links And Resources

- https://www.javatpoint.com/java-jdbc
- https://spring.io/projects/spring-boot
- https://www.youtube.com/watch?v=MV4o_b6vt0g&t=344s&ab_channel=BurakKutbay

- https://www.docker.com/
- https://hevodata.com/learn/spring-boot-mysql/
- https://docs.spring.io/spring-data/data-commons/docs/1.6.1.RELEASE/reference/html/repositories.html
- https://www.educba.com/spring-boot-repository/
- https://app.diagrams.net/