Awesome Music Booking System

Luca Signore

March 19, 2025

1 Introduction

This document provides a detailed overview of the Booking System, outlining its architecture, database structure, and core functionalities.

The system allows users to book rehearsal rooms without requiring registration. Reservations can be made for three different time slots: morning, afternoon, and evening. A studio manager reviews incoming bookings and can approve or reject them. Users receive a unique booking code to track their reservation status.

2 System Overview

The system is developed using Java 17 and Spring Boot 3.4.3. It follows a layered architecture consisting of Controllers, Services, Repositories, and Entities. The backend is structured to handle room bookings efficiently by providing APIs for managing rooms, slots, and bookings. The system integrates with a MySQL database for data persistence.

The project includes a dedicated package ‘converter‘, which contains classes responsible for formatting date and datetime values. The formats are declared as ‘public static final‘ variables in the ‘Setting.java‘ class within the ‘config‘ package. This ensures a consistent approach to handling date formats across the system.

To manage booking statuses efficiently, an entity ‘BookingStatus‘ of type ‘enum‘ has been created. This allows predefined states such as ‘PENDING‘, ‘ACCEPTED‘, ‘REFUSED‘, and ‘CANCELED‘, making it easier to handle status transitions within the system.

The service layer is divided into interfaces and implementations. The interfaces define the contract for business logic operations, while the implementations provide the actual logic. This separation allows for better maintainability, easier unit testing, and dependency injection.

Additionally, the ‘mapper‘ package includes methods for converting between ‘RoomDto‘ and ‘Room‘ entities, facilitating seamless data transformation between the entity layer and the DTO layer.

2.1 Dependencies

The project utilizes the following dependencies:

- **spring-boot-starter-data-jpa**: Provides integration with JPA (Java Persistence API) to manage database operations.
- **spring-boot-starter-validation**: Supports validation of incoming request payloads using annotations.
- **spring-boot-starter-web**: Enables RESTful web service development with Spring MVC.
- **spring-boot-devtools**: Provides utilities for automatic application restart and live reload during development.
- **mysql-connector-j**: The official MySQL JDBC driver for database connectivity.
- **lombok**: Reduces boilerplate code by automatically generating getters, setters, constructors, and other utility methods.
- **spring-boot-starter-test**: Provides testing support, including JUnit, Mockito, and Spring Test utilities.
- **springdoc-openapi-starter-webmvc-ui**: Integrates OpenAPI (Swagger) for API documentation and UI representation.

3 Entity-Relationship Diagram

The Entity-Relationship Diagram (EER) illustrates the database structure of the system. The main entities are:

- **Booking**: Represents a reservation, storing details like customer name, booking date, booking code, and status.
- **Slot**: Defines available time slots for bookings, with attributes such as name, start hour, and end hour.
- **Room**: Represents different rooms available for booking, each with a name and description.

Relationships:

- A **Booking** is associated with a **Slot** and a **Room**.
- A **Slot** and a **Room** can have multiple bookings but are uniquely referenced in each booking instance.

This model ensures data integrity and efficient booking management.

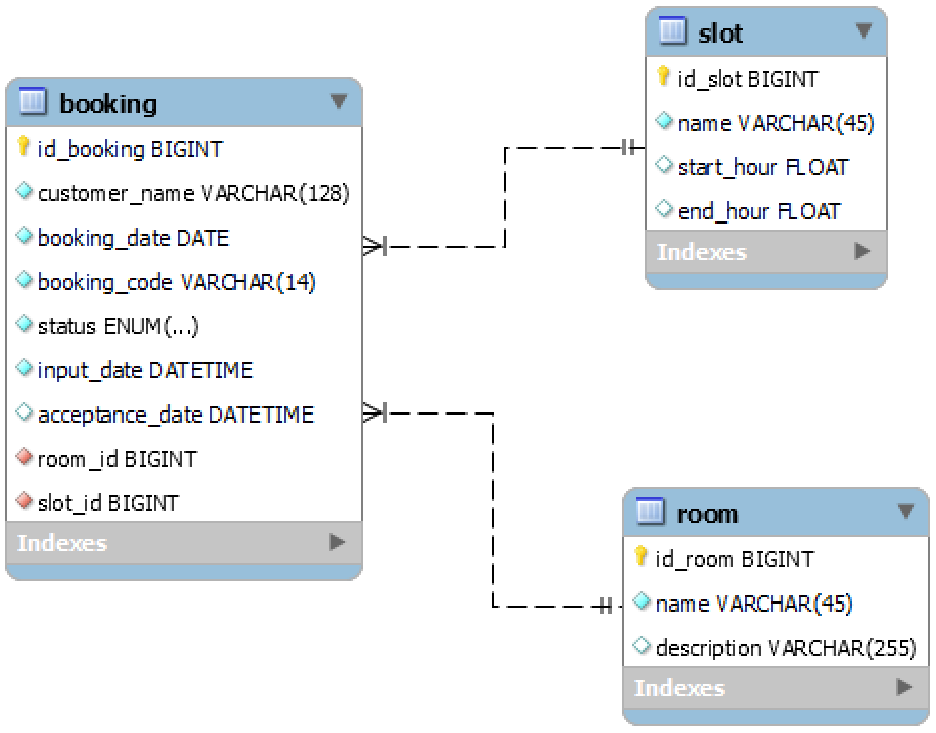

Figure 1: Entity-Relationship Diagram of the Booking System

4 System Architecture

The system follows a layered architecture:

- **Controller Layer**: Handles HTTP requests and responses, delegating the business logic processing to the service layer.
- **Service Layer**: Implements business logic, interacting with repositories to fetch and manipulate data.
- **Repository Layer**: Manages database operations through JPA repositories, abstracting direct database access.
- **Entity Layer**: Defines the data model using JPA entity classes, mapping database tables to Java objects.
- **DTO Layer**: Contains Data Transfer Objects (DTOs) used for exchanging data between client and server while keeping entities independent from external requests.

5 API Documentation

The API follows the REST principles and exposes the following endpoints:

5.1 RoomController

- **GET /api/rooms** - Retrieves a list of all rooms.

- **GET /api/rooms/available** - Retrieves a list of available rooms based on booking date and slot name.
  - **Request Parameters:**
    - `bookingDate` (LocalDate, required) - The date of the booking.
    - `slotName` (String, required) - The name of the slot.
- **POST /api/rooms** - Creates a new room.
  - **Request Body:** JSON object representing RoomDto.

5.2 SlotController

- **GET /api/slots** - Retrieves a list of all slots.
- **POST /api/slots** - Creates a new slot.
  - **Request Body:** JSON object representing Slot.
- **GET /api/slots/{name}** - Retrieves a slot by name.
  - **Path Parameter:** `name` (String, required) - The name of the slot to retrieve.

5.3 BookingController

- **POST /api/bookings** - Creates a new booking. Returns the booking code if successful; otherwise, returns an error message.
  - **Request Body:** JSON object representing BookingRequest.
- **GET /api/bookings** - Retrieves all bookings.
- **GET /api/bookings/{bookingCode}** - Retrieves a booking using the booking code.
  - **Path Parameter:** `bookingCode` (String, required) - The unique booking code.
- **PATCH /api/bookings/{id}/status** - Updates the status of a booking.
  - **Path Parameter:** `id` (Long, required) - The ID of the booking.
  - **Query Parameter:** `status` (String, required) - The new status.
- **PATCH /api/bookings/{id}/accept** - Accepts a booking.
  - **Path Parameter:** `id` (Long, required) - The ID of the booking.

5.4 Swagger Docs

Once the application is running, the API documentation can be accessed at:

```
http://localhost:8080/swagger-ui/index.html
```

Here, users can explore and test the available endpoints interactively.

6 Testing

Testing is an essential part of the development process to ensure the reliability and correctness of the system. The project includes unit tests and integration tests to validate the expected behavior of different components.

6.1 Controller Testing

Controller tests validate the API endpoints by simulating HTTP requests and verifying the responses. Below is an example of a test class for the `BookingController`:

```
@SpringBootTest
public class BookingControllerTest {

    @MockitoBean
    private BookingService bookingService;

    @Autowired
    private ObjectMapper objectMapper;

    private MockMvc mockMvc;

    @BeforeEach
    public void setup(WebApplicationContext wac) {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(wac).build();
    }

    @Test
    public void testCreateBooking() throws Exception {
        BookingRequest request = new BookingRequest();

        // Set needed fields
        Booking booking = new Booking();
        booking.setBookingCode("CU0004SL032025");

        when(bookingService.createBooking(any(BookingRequest.class)))
                .thenReturn(Optional.of(booking));

        mockMvc.perform(post("/api/bookings")
                .contentType("application/json")
                .content(objectMapper.writeValueAsString(request)))
                .andExpect(status().isOk())
                .andExpect(content().string("CU0004SL032025"));
    }

    @Test
    public void testUpdateBookingStatus() throws Exception {
        Booking booking = new Booking();
        booking.setStatus(BookingStatus.ACCEPTED);

        Mockito.when(bookingService.updateBookingStatus(
                    Mockito.anyLong(), Mockito.anyString()))
               .thenReturn(booking);

        mockMvc.perform(patch("/api/bookings/1/status")
                .param("status", "ACCEPTED"))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.status").value("ACCEPTED"));
    }
}
```

These tests ensure that the booking process and status update functionalities work as expected. More tests should be added to cover different edge cases and error handling scenarios.

7 Conclusion

This document provides a detailed overview of the Booking System.
Potential future developments for the system include:

- Adding a user authentication and authorization system to enable role-based access control.
- Implementing real-time notifications for booking status updates.
- Extending reporting capabilities with analytics and insights on room bookings.
- Integrating third-party calendar services (e.g., Google Calendar) for booking synchronization.
- Allowing users to book additional **Equipment** along with their room reservations.
- Restricting the availability of **Equipment** based on the selected **Room**, ensuring compatibility and better resource management.

Contents