Hoiiday Application

Name: Pîrvulescu Carmen-Gabriela

Group: 30431

Contents

1 Deliverable 1 . . . . . **2**
- 1.1 Project Specification 2
- 1.2 Functional Requirements 2
- 1.3 Use Case Model 3
  - 1.3.1 Use Case Identification 3
  - 1.3.2 UML Use Case Diagram 3
- 1.4 Supplementary Specification 4
  - 1.4.1 Non-functional Requirements 4
  - 1.4.2 Design Constraints 4
  - 1.4.3 Glossary 4

2 Deliverable 2 . . . . . **5**
- 2.1 Domain Model 5
- 2.2 Architectural Design 5
  - 2.2.1 Conceptual Architecture 5
  - 2.2.2 Package Design 6
  - 2.2.3 Component and Deployment Diagram 7

3 Deliverable 3 . . . . . **8**
- 3.1 Design Model 8
  - 3.1.1 Dynamic behavior 8
  - 3.1.2 Class Diagram 9
- 3.2 Data Model 13

4 System Testing **15**
- 4.1 Test Environment 15
- 4.2 Test cases 15
  - 4.2.1 Additional System Testing Activities 15

5 Future Improvements **15**

6 Conclusion **16**

7 Bibliography **16**

1 Deliverable 1

1.1 Project Specification

Hoiiday is a web-based booking application inspired by a passion for travel and exotic destinations. It is designed to provide a user experience similar to well-known platforms like Booking.com. Users can search for available properties, view detailed information — such as amenities, policies, and rules — and make bookings for specific dates.

A distinctive feature of the application is the "Tips & Tricks" section, where users can find travel advice, destination suggestions based on the season, and packing recommendations. In the future, the application may also offer optional features like car rentals and flight bookings.

1.2 Functional Requirements

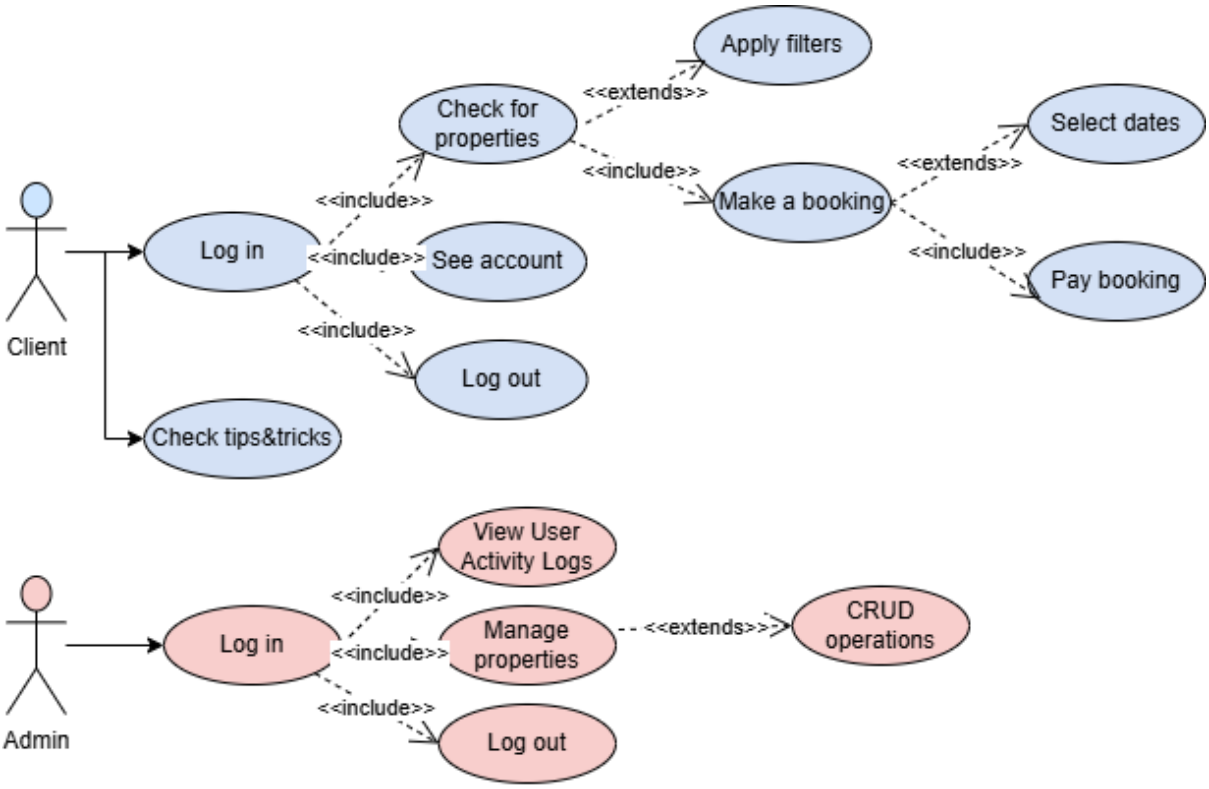
The system supports the following core functionalities:

- Users can create an account and log in, with roles such as client or admin.
- Users can search for properties using filters like location, number of guests, or number of rooms.
- Users can book a property if it is available during the selected period.
- Users can access travel-related tips and destination recommendations.
- Users can make payments by card and receive an invoice after booking.
- Admins can add, update, or delete property listings.
- Admins can view user accounts, monitor activity logs, and manage the platform.

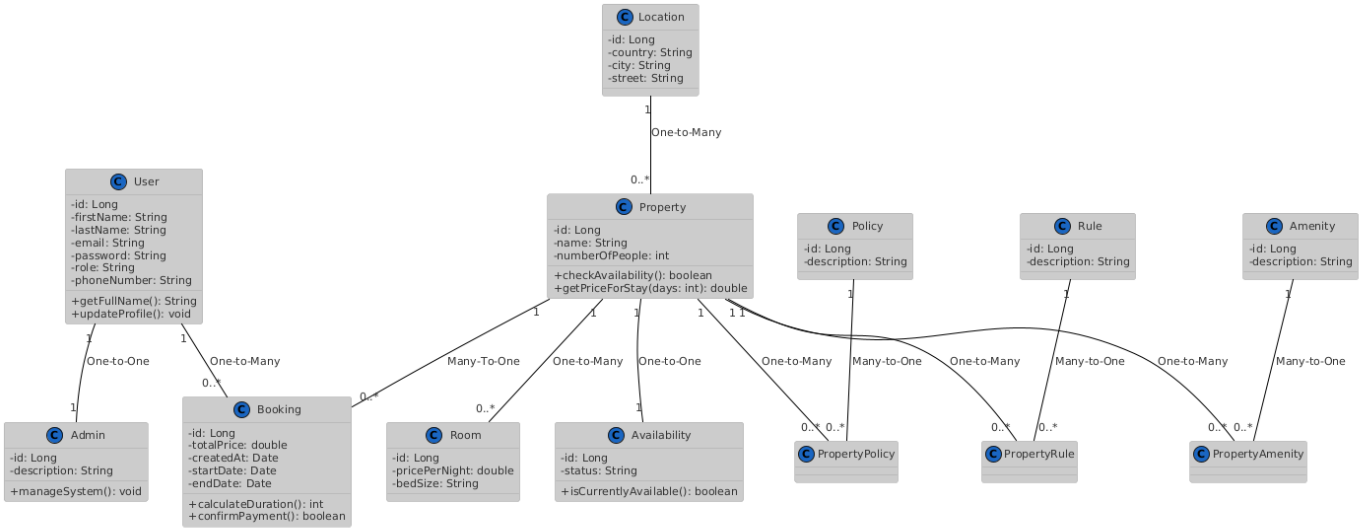
Planned features include booking flights and renting cars.

1.3 Use Case Model

1.3.1 Use Case Identification



1.3.2 UML Use Case Diagram



1.4 Supplementary Specification

1.4.1 Non-functional Requirements

- **Performance**: The application should quickly respond to user actions, ensuring minimal loading times and smooth interaction.
- **Security**: User passwords will be stored securely using encryption techniques. OAuth2 will be used for login to enhance authentication security.
- **Usability**: The interface should be intuitive, modern, and simple to navigate, avoiding unnecessary complexity.
- **Scalability**: The application must support a growing number of listings without a drop in performance.

1.4.2 Design Constraints

- **Programming Language**: The backend will be developed using Java with the Spring Boot framework.
- **Frontend Framework**: The frontend will be built using Angular 18, leveraging standalone components for modularity.
- **Database**: The system will use a MySQL relational database to store user and property data, including encrypted passwords.
- **Authentication**: The login process will use both standard credentials and Google OAuth2 for secure access.
- **Architecture**: The system will follow a RESTful client-server architecture, structured into multiple layers such as the data access layer, service layer, and presentation layer.

1.4.3 Glossary

- **Booking**: Reserving a property for a selected time interval.
- **Admin**: A user with full access rights who can manage properties, users, and system logs.
- **Tips & Tricks**: A section providing users with useful travel recommendations and seasonal advice.
- **Amenity**: A service or feature offered by a property, such as Wi-Fi, pool, or parking.
- **OAuth2**: A secure authentication protocol allowing login via third-party services like Google.
- **Invoice**: A digital receipt issued after payment is completed.
- **REST API**: A style of backend design using HTTP methods for communication between client and server components.

2 Deliverable 2

2.1 Domain Model

The domain model represents the core entities and their relationships within the Hoiiday application. The key entities include **User**, **Admin**, **Property**, **Booking**, **Room**, and **Location**, which form the backbone of the application. These entities are connected through various relationships.

The **Conceptual Class Diagram** is depicted below, showing the classes and their relationships.

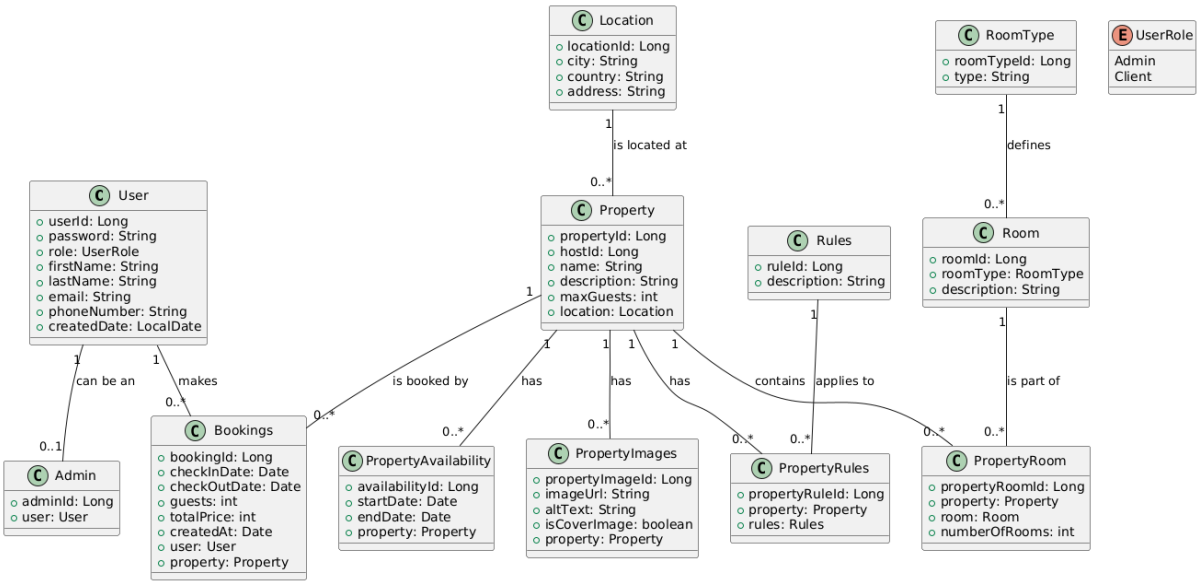


Figure 1: Conceptual Class Diagram for the Hoiiday Application

2.2 Architectural Design

2.2.1 Conceptual Architecture

The conceptual architecture for the Hoiiday application follows the **Layered Architecture Pattern**. It is divided into three main layers: the **Presentation Layer**, **Service Layer**, and **Data Access Layer**.

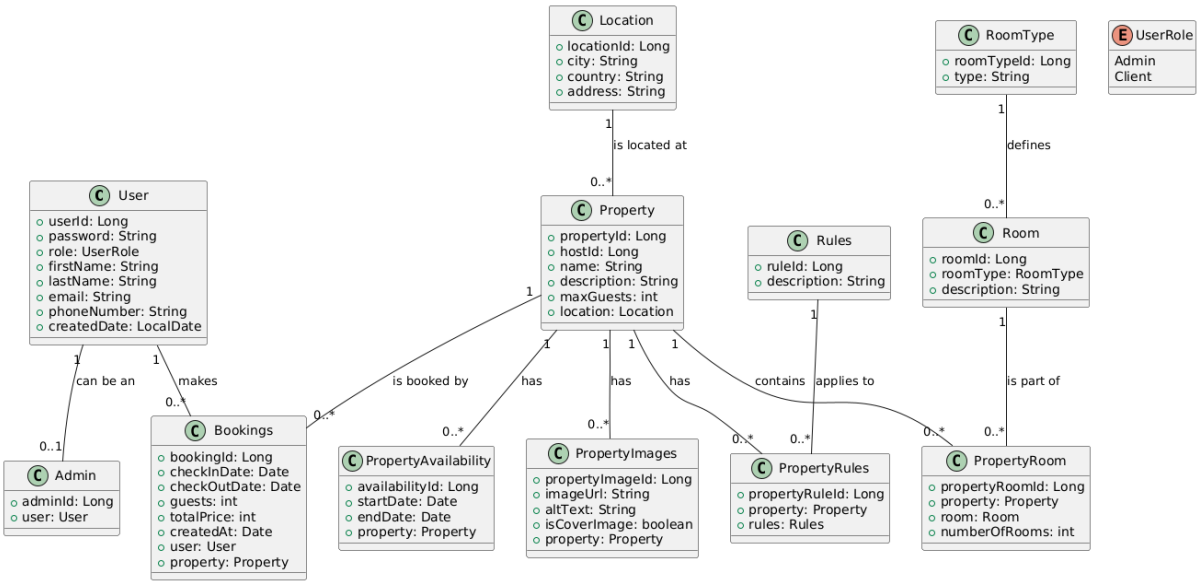


Figure 2: Conceptual Architecture of the Hoiiday Application

2.2.2 Package Design

The package design organizes the Hoiiday application into distinct and logically grouped modules, each responsible for specific functionality. This modular organization helps in maintaining clean code.

- **Controller Package**: - This package contains the **controllers** responsible for handling the incoming HTTP requests from the frontend.
- **Service Package**: - The **service** layer contains the **business logic** of the application. Services communicate with the database layer (via the **repository** package) to fetch or update data.
- **Repository Package**: - The **repository** package is responsible for interacting with the database. It uses **Spring Data JPA** to manage entities. Repositories are responsible for CRUD operations.
- **Model Package**: - The **model** package contains the **entity classes** that represent the core objects of the application. These classes are mapped to the corresponding database tables using **JPA annotations**.
- **DTO Package**: - The **DTO** (Data Transfer Object) package contains the **data transfer objects** that are used to transfer data between layers of the application.
- **Mapper Package**: - The **mapper** package is responsible for converting between **entities** and **DTOs**. It uses a mapping layer to transfer data between the service and the DTO layer.
- **Authentication Package** (optional): - This package handles **authentication and authorization**. It manages user login, logout, token generation (JWT, for example), and role-based access control (RBAC). It ensures that only authorized users can access certain features of the application.

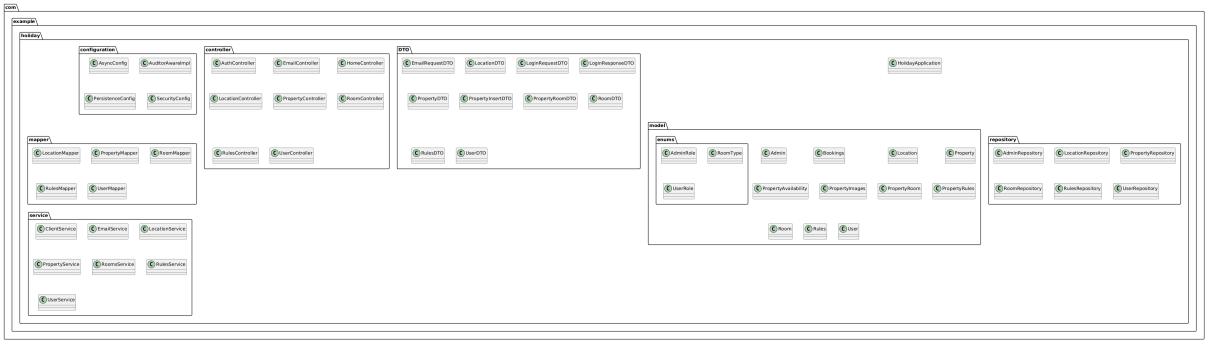


Figure 3: Package Design for the Hoiiday Application

2.2.3 Component and Deployment Diagram

The component and deployment diagrams illustrate the system structure and its deployment across different environments.

- **Component Diagram**: - The **component diagram** shows the system's software components (e.g., **Angular frontend**, **Spring Boot backend**, **database**) and their interactions via well-defined interfaces, such as REST APIs. - It highlights how services like **User Service** and **Booking Service** communicate with the database and each other.

- **Deployment Diagram**: - The **deployment diagram** shows how components are deployed across physical servers. It illustrates the deployment of the **frontend** on a **web server**, the **backend** on an **application server**, and the **database** on a **database server**. - It also shows how these servers communicate, ensuring smooth interaction between components.

Both diagrams provide insight into the system's logical and physical architecture, helping identify potential **bottlenecks** and ensuring **scalability**.

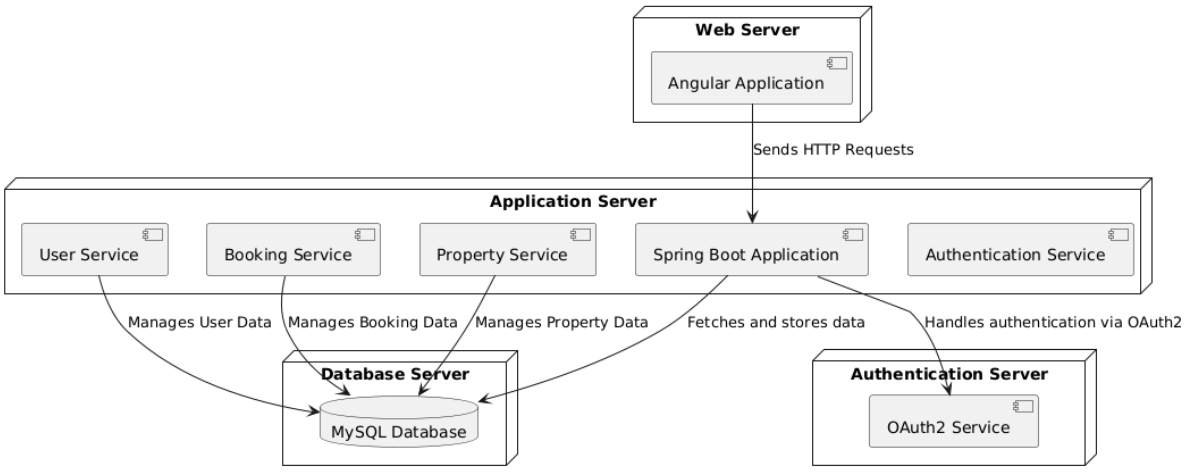


Figure 4: Deployment Diagram for the Hoiiday Application

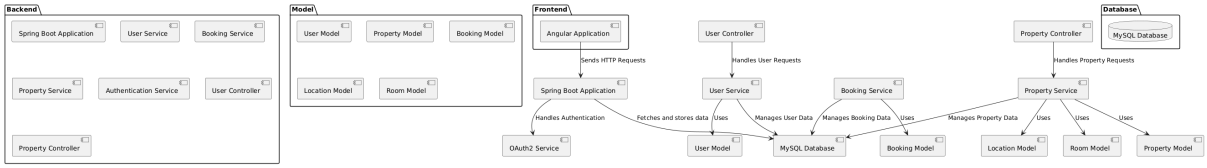


Figure 5: Component Diagram for the Hoiiday Application

3 Deliverable 3

3.1 Design Model

3.1.1 Dynamic behavior

In software design, **dynamic behavior** refers to how a system changes over time in response to events or interactions. It's about capturing the time-dependent aspects of a system, rather than just its static structure (like classes and their relationships).

Exemplification for two cases: **login demo** and **booking a property** demo.
Actors: Users, Frontend, Backend Actions:

- User enters credentials.
- The AuthController and Validator validates the credentials.
- The repository and database are checked.
- After a successful login, the homepage is opened.

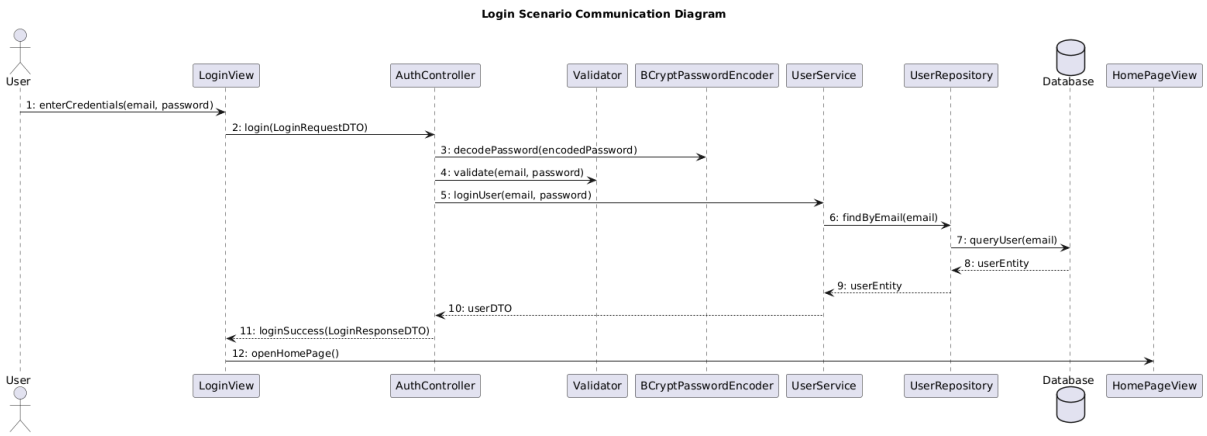


Figure 6: Sequence diagram for Login

Actors: Users, Frontend, Backend Actions:

- User views properties in the home page.
- The user chooses to view details for a specific property in order to book it.

- The availability is checked.
- If the property is available, the booking request is registered.
- A PDF with the details is generated.

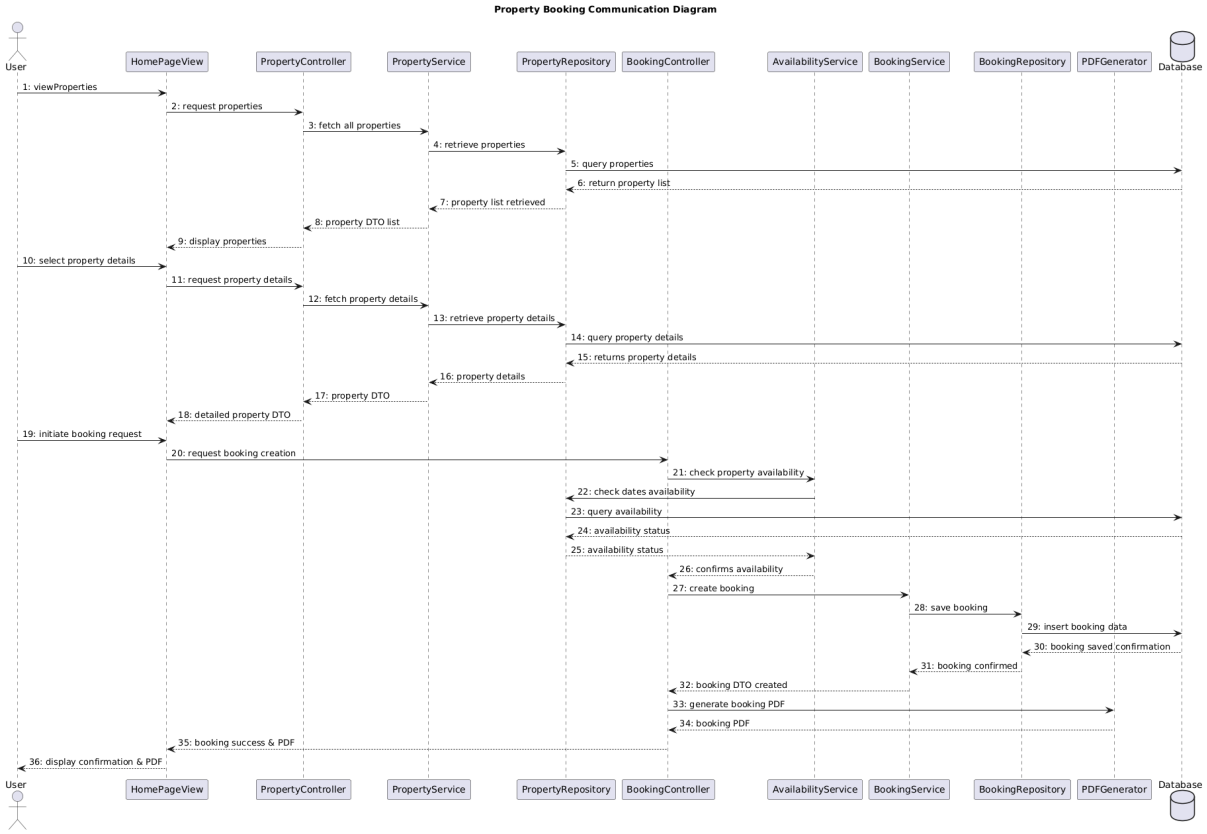


Figure 7: Sequence diagram for Booking a property

3.1.2 Class Diagram

The Class diagram visually represents the static structure of a system by illustrating its classes, their attributes, methods, and the relationships among these classes.

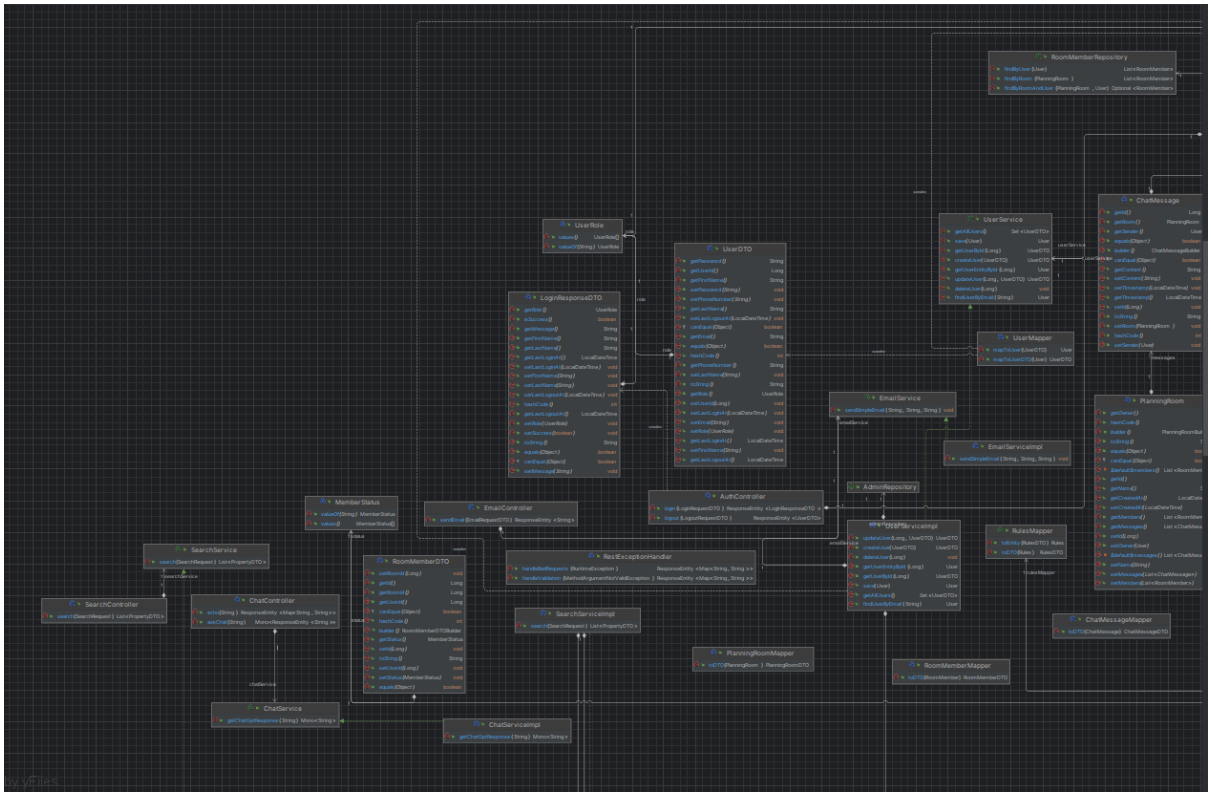


Figure 8: Class diagram frame 1

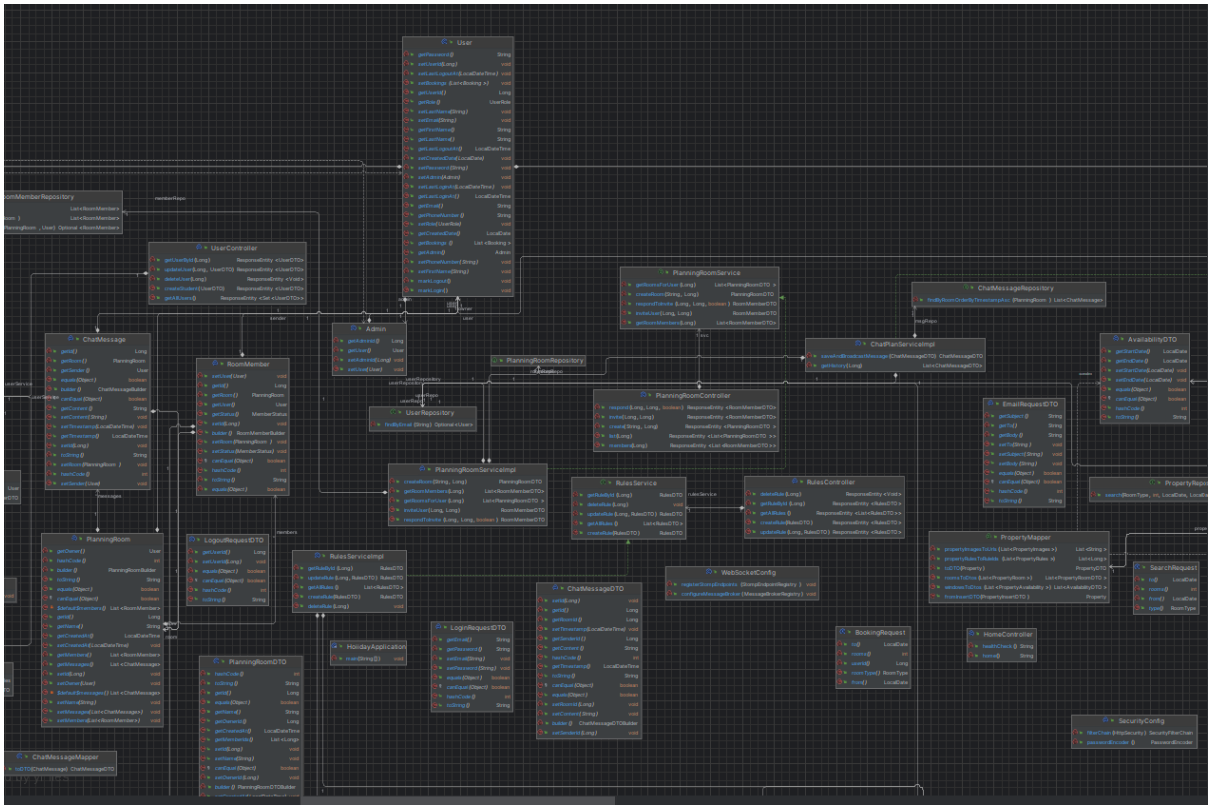


Figure 9: Class diagram frame 2

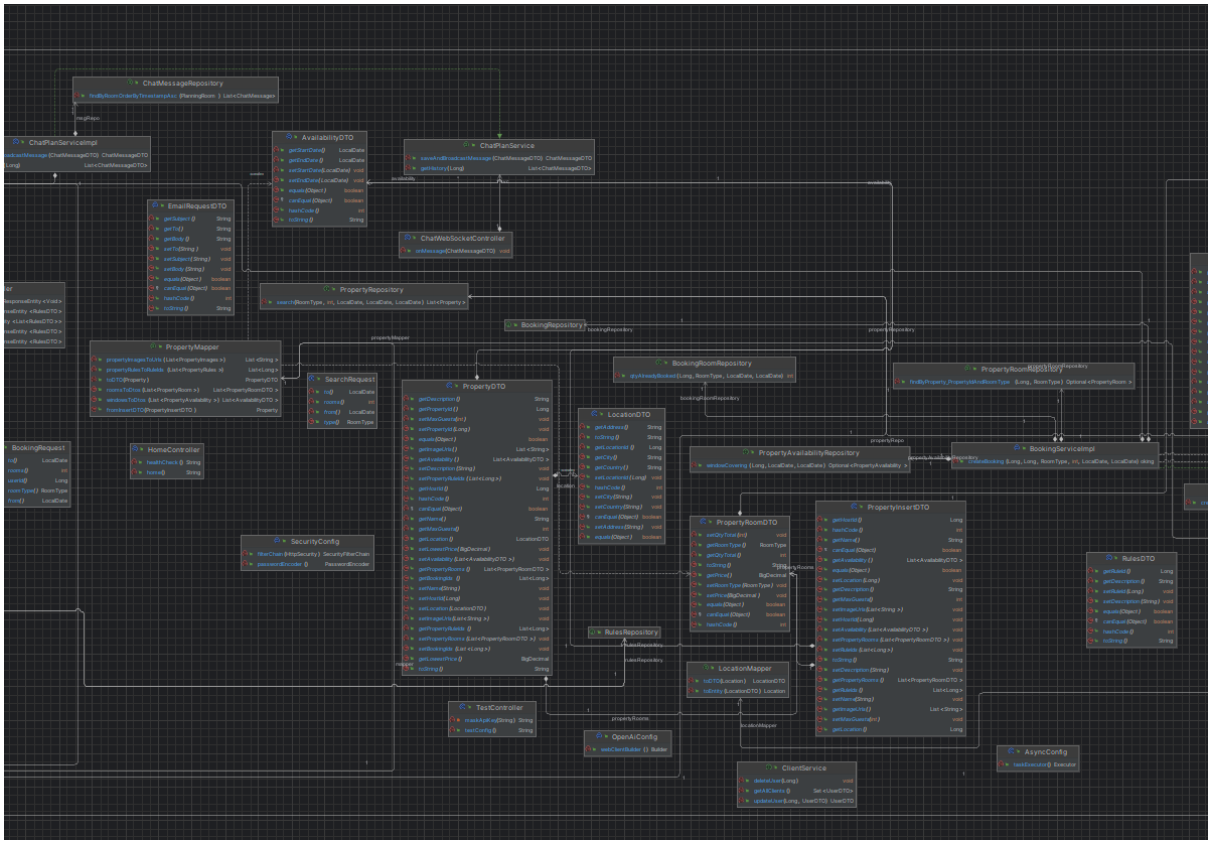


Figure 10: Class diagram frame 3

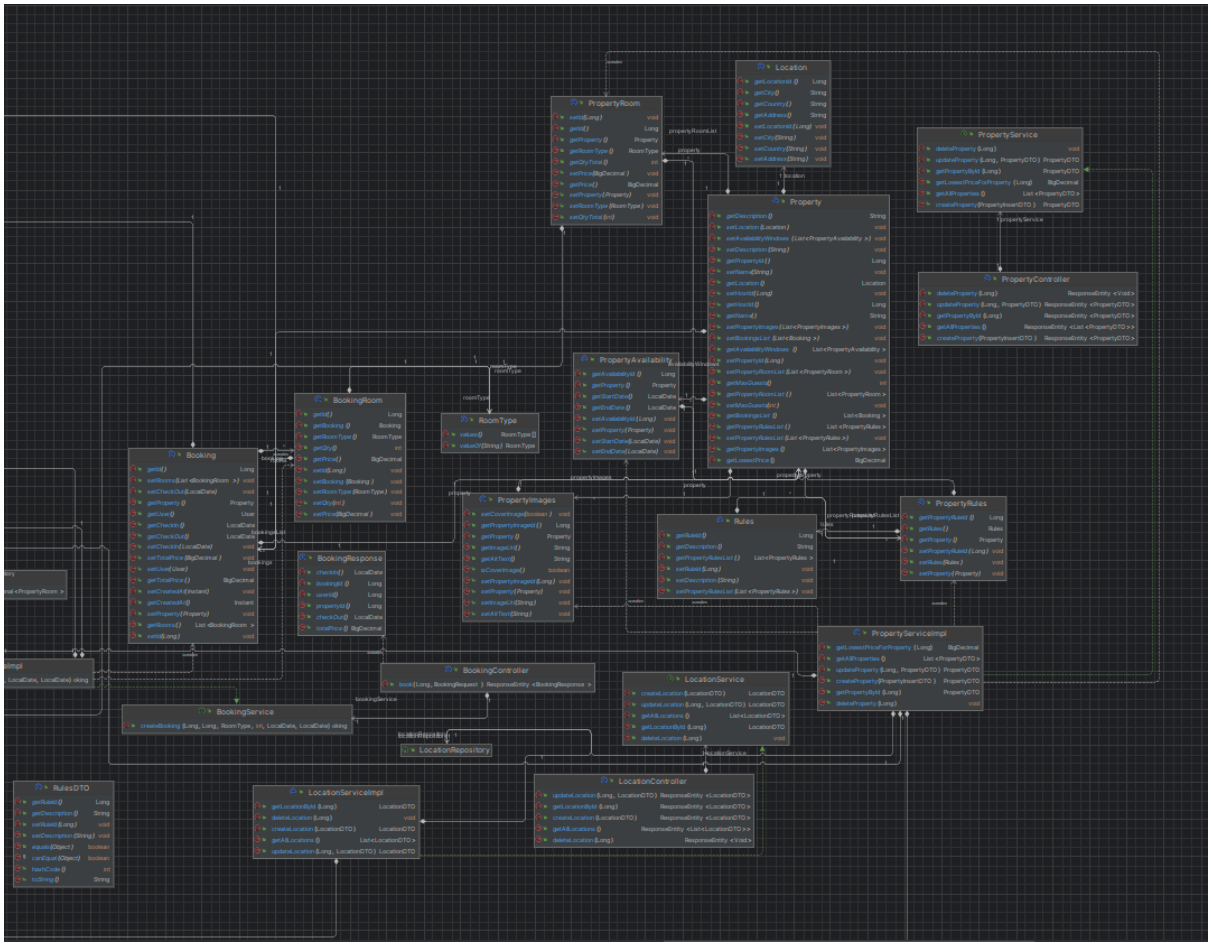
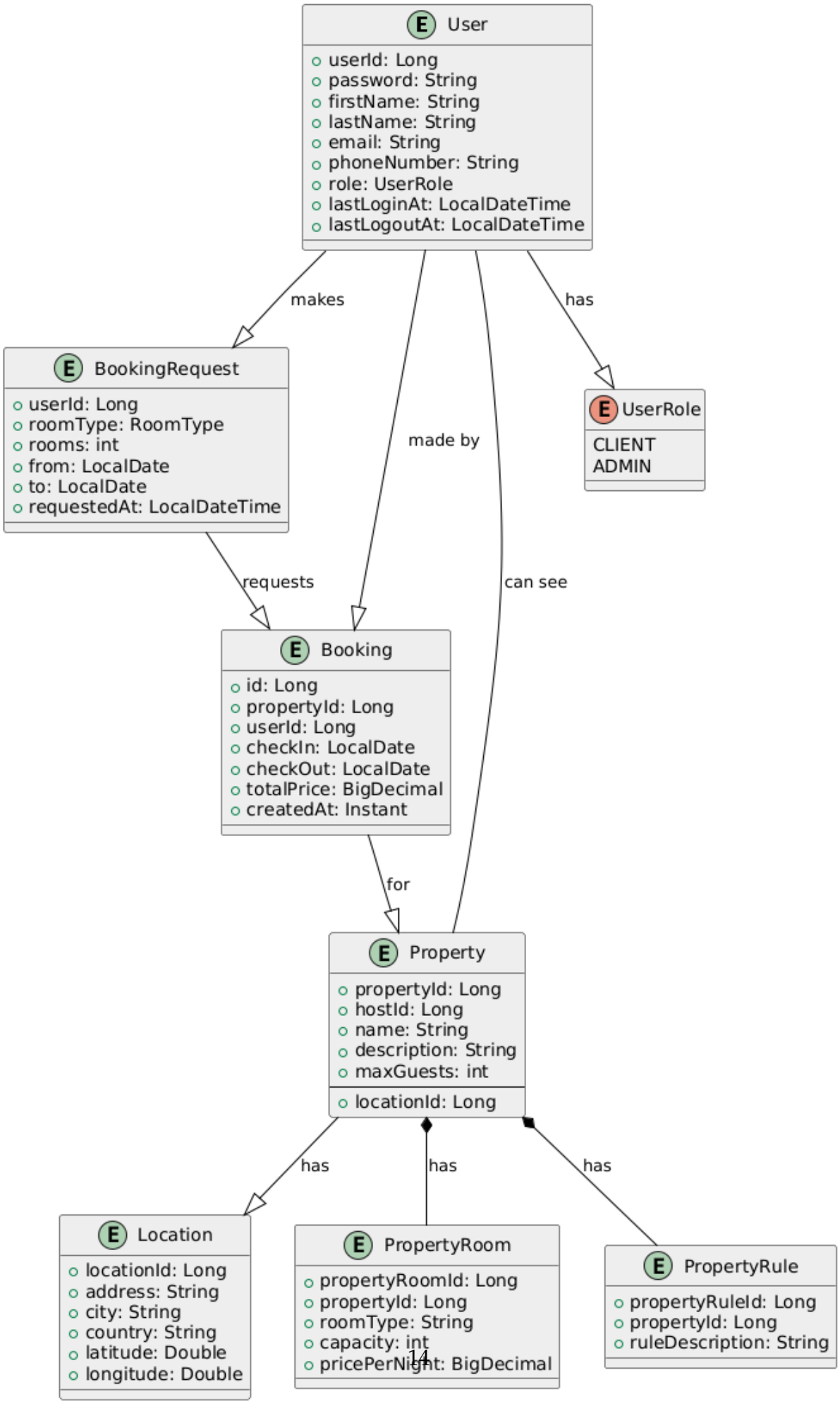


Figure 11: Class diagram frame 4

3.2 Data Model

The data model is an abstraction that represents the core data entities. It serves as the foundation for database design and system development.

- A single user can create multiple booking requests.
- A user can have multiple confirmed bookings.
- Multiple bookings can refer to the same property, but each booking is specific to one property.
- Each property has exactly one geographic location, one set of rules and rooms.
- Each user has one defined role (either CLIENT or ADMIN).


User
userId: Long
password: String
firstName: String
lastName: String
email: String
phoneNumber: String
role: UserRole
lastLoginAt: LocalDateTime
lastLogoutAt: LocalDateTime
makes
has
BookingRequest
userId: Long
roomType: RoomType
rooms: int
from: LocalDate
to: LocalDate
requestedAt: LocalDateTime
UserRole
CLIENT
ADMIN
made by
requests
can see
Booking
id: Long
propertyId: Long
userId: Long
checkIn: LocalDate
checkOut: LocalDate
totalPrice: BigDecimal
createdAt: Instant
for
Property
propertyId: Long
hostId: Long
name: String
description: String
maxGuests: int
locationId: Long
has
has
has
Location
locationId: Long
address: String
city: String
country: String
latitude: Double
longitude: Double
PropertyRoom
propertyRoomId: Long
propertyId: Long
roomType: String
capacity: int
pricePerNight: BigDecimal
PropertyRule
propertyRuleId: Long
propertyId: Long
ruleDescription: String

4 System Testing

The purpose of system testing is to verify that all application components (frontend, backend and database) work together correctly and meet the specified requirements.

4.1 Test Environment

- **API server**: 'http://localhost:8080' (Spring Boot, Java 21)
- **Frontend**: Angular 16 served at 'http://localhost:4200'
- **Database**: PostgreSQL with a dedicated test schema, pre-loaded with seed data
- **Tooling**: Postman v10.14.7

4.2 Test cases

4.2.1 Additional System Testing Activities

In addition to the manual Postman tests listed above, we performed:

- **Angular Form Validation:** Manual and automated checks of the client-side validators (required fields, email pattern, 10-digit phone number) to confirm the UI blocks invalid submissions before any network call.
- **End-to-End Sanity Checks:** Manual walkthroughs in the browser (Chrome) covering full login $\rightarrow$ search $\rightarrow$ booking flows, verifying that between UI validation and backend JWT checks the application behaves end-to-end as designed.

5 Future Improvements

While Hoiiday already offers a solid foundation for property search and booking, there are several areas I plan to enhance in future iterations:

- **Payment Gateway Integration:**
  - Support for major credit cards (Visa, MasterCard, AMEX) via Stripe or PayPal.
  - Handling of reservations vs. immediate charges, refunds and cancellations.
- **Credentialed Guides & "Tips & Tricks" Expansion:**
  - On-demand booking of vetted local guides, integrated into the booking flow.
  - Rich "Tips & Tricks" articles authored by professional travel writers, with tags for season, budget, and traveler type.
  - User-generated Q&A and ratings for each tip to surface the most helpful advice.
- **Multi-language and Localization:**

  - Full UI translations into Spanish, French, Mandarin, and German.
  - Currency conversion based on user locale.
  - Date/time and number formatting following local conventions.

- **Mobile App and Offline Support:**
  - Native Android/iOS apps for on-the-go booking and push notifications.
  - Offline access to saved itineraries and downloaded "Tips & Tricks."
  - In-app messaging between guests and hosts.

6 Conclusion

The Hoiiday application delivers a comprehensive, user-friendly platform for searching and booking accommodations, Through a RESTful Spring Boot backend, an Angular frontend with client-side validation, and a PostgreSQL database, all core functional and non-functional requirements have been met.

System testing—both via Postman for API endpoints and manual end-to-end browser walkthroughs—has verified proper integration of authentication, property search, booking flows, and XML export functionality. Frontend form validations prevent invalid emails and phone numbers before network calls, while JWT guards ensure secure access control on protected endpoints.

Looking ahead, my roadmap includes payment gateway integration, expanded travel content with credentialed guides, user review systems, and multilingual support, as well as native mobile applications and advanced analytics for hosts. These future improvements will further enhance Hoiiday's scalability, usability, and value proposition, positioning it as a competitive solution in the online travel booking market.

7 Bibliography

References

[1] Spring Boot – `https://spring.io/projects/spring-boot`

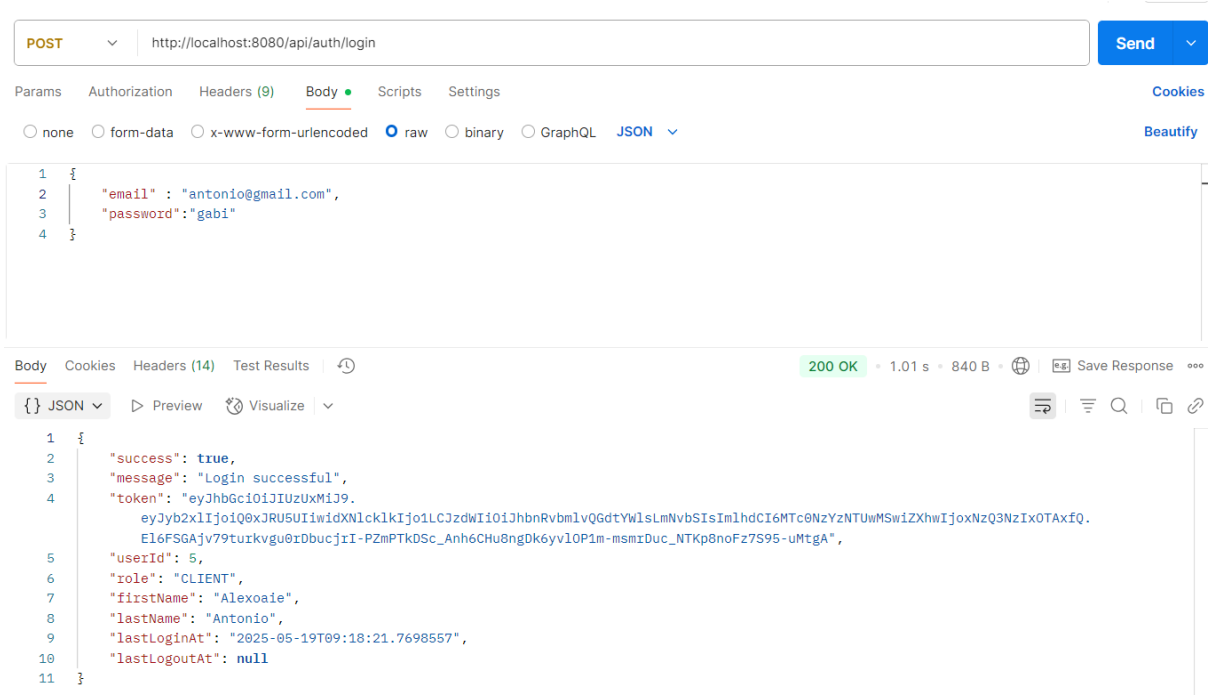
[2] Angular – `https://angular.io/`

[3] PostgreSQL – `https://www.postgresql.org/`

[4] JSON Web Tokens – `https://jwt.io/`

Table 1: API Test Cases

#	Endpoint	Method	Pre-condition	Test Data / Steps	Expected Result
1	`/api/auth/login`	POST	User exists in DB	`{"email+pass"}`	200 OK+JSON {…token}



2	`/api/properties`	GET	With valid JWT	Header: `Authorization: Bearer <token>`	200 OK + list

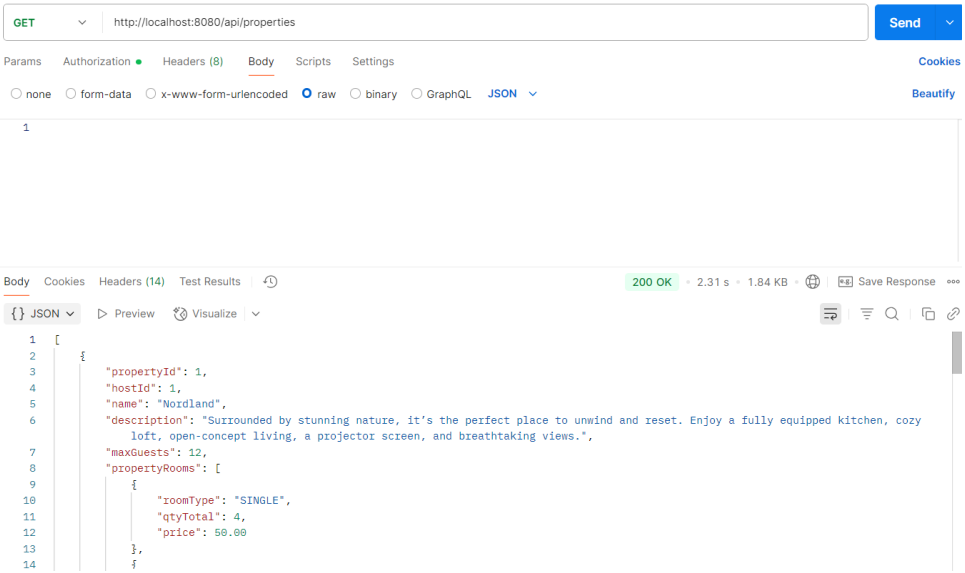
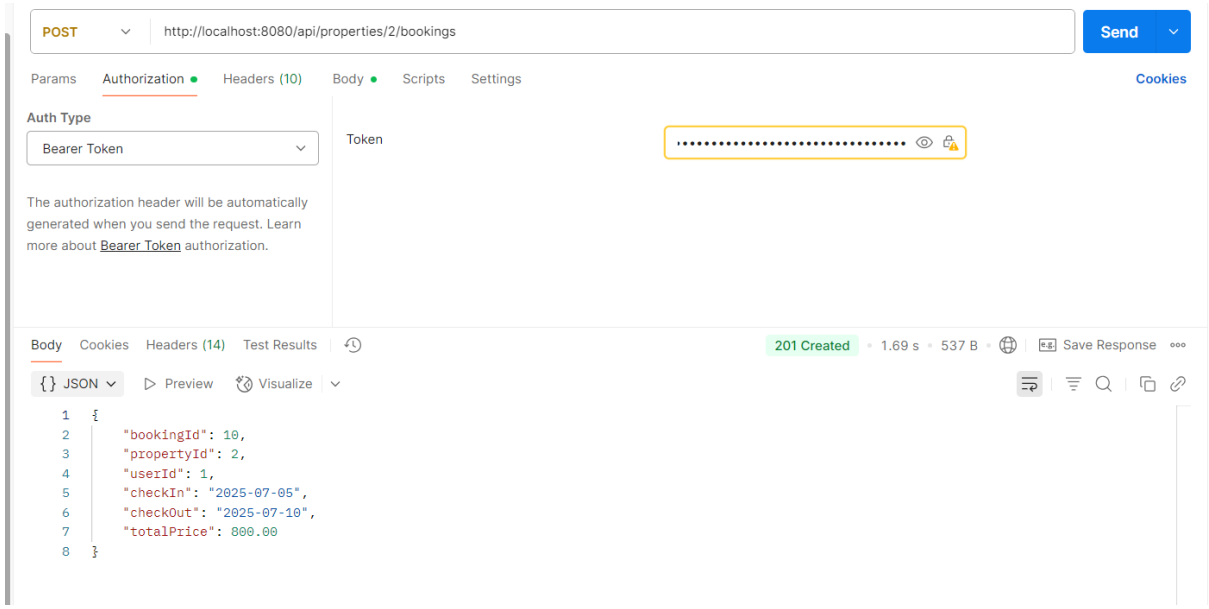


Table 2: API Test Cases

#	Endpoint	Method	Pre-condition	Test Data / Steps	Expected Result
3		POST	Authenticated	valid booking payload	201 Created + confirmation JSON



4	`ex. of troubleshooting`	POST	Authenticated or public	same booking payload	400 BAD REQUEST + error