HBnB UML

Part 1: Technical Documentation

HBnB Evolution Application

High-level Package Diagram | Class Diagram | Sequence Diagram

Group 1

Grâce Kayembe
Crystal Chiam
Emily Chew

Introduction

This document provides the technical design artifacts for the **HBnB project**, a simplified accommodation booking platform inspired by services like Airbnb. The goal of HBnB is to allow users to create accounts, list properties, manage amenities, make reservations, and leave reviews, while maintaining clear business rules and data integrity.

The purpose of these diagrams is to visualise the architecture and behaviour of the system, serving as a guide for development and a shared reference point for the project team. Each diagram highlights a different perspective of the application:

- **High-Level Package Diagram** – illustrates the overall architecture and separation of responsibilities across the main layers of the system (presentation, business logic, persistence).

- **Class Diagram (Business Logic Layer)** – provides a detailed view of the core entities (User, Place, Review, Amenity, Reservation), their attributes, methods, and relationships, ensuring consistency and maintainability through the use of an abstract BaseModel.

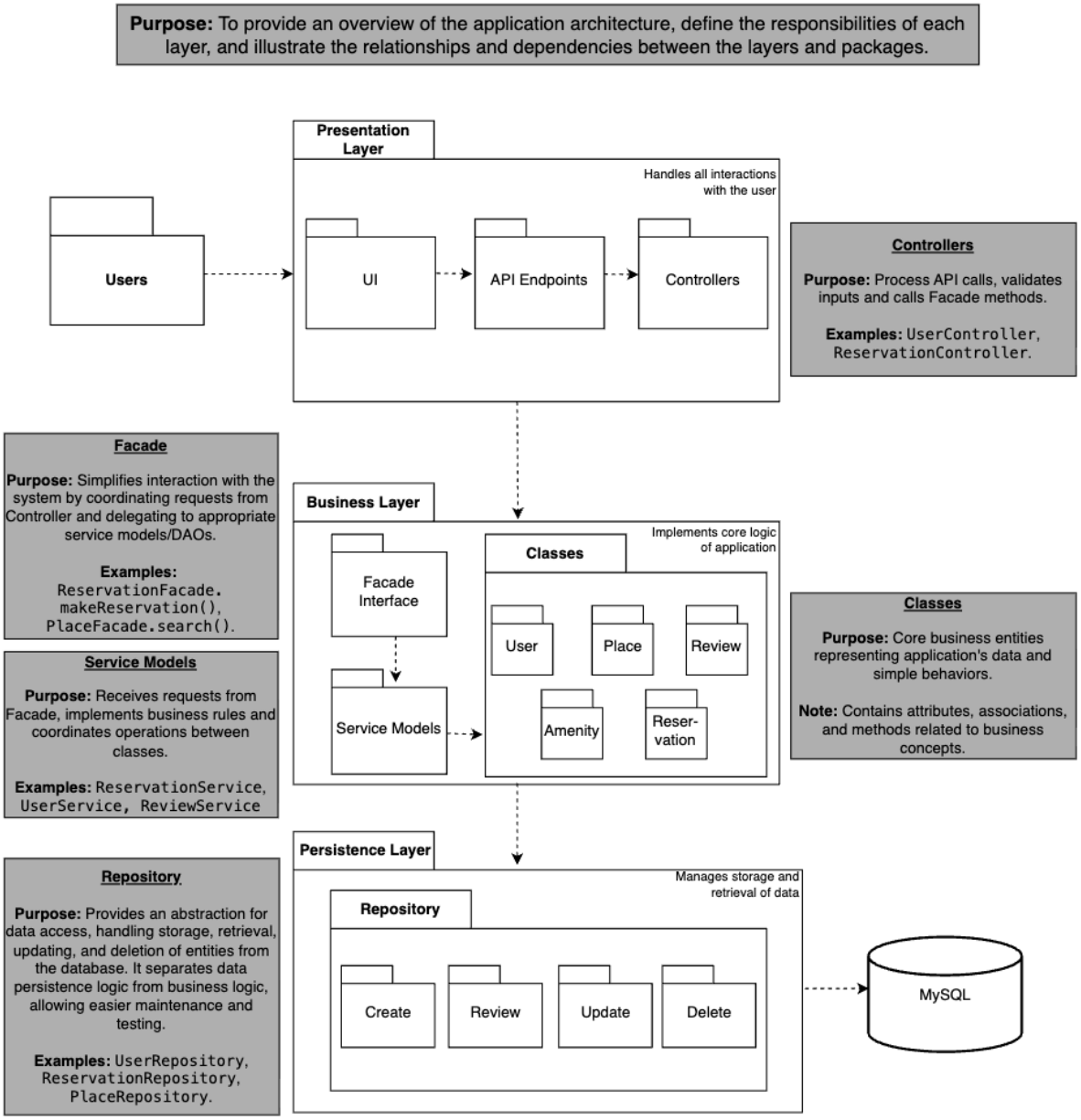
- **Sequence Diagrams (API Calls)** – describe the dynamic interactions between the user interface, API controllers, and the business logic during key operations, such as creating user accounts, booking places, or posting reviews.

Together, these diagrams provide a **comprehensive blueprint** of the HBnB application, guiding both the implementation process and future extensions. By defining structure and behaviour before coding, the team ensures clarity, consistency, and alignment with the project's requirements.

High-Level Architecture

The purpose of the diagram is to provide a high level overview of the application architecture, define the responsibilities of each layer, and illustrate the relationships and dependencies between packages. It sits above class-level or sequence diagrams in the design hierarchy and serves as a foundation for the more detailed UML diagrams that can drill into specific components.

Package Diagram



The package diagram comprises three main packages/layers: Presentation Layer, Business Layer and Persistence Layer. The layers and their key components are described in further detail below:

Key Components

1. **Presentation Layer:** Handles all interactions between the user and the system - translating user actions into operations and displaying results.
    - **User Interface (UI):** the web interface users interact with. It presents data (e.g., available places, reviews) and collects inputs (e.g., login details, search criteria).
    - **API Endpoints:** URLs or routes through which user requests access application features.
    - **Controllers:** process incoming requests, validate inputs, call the Facade Interface in the Business Layer, and return responses (e.g., booking confirmations, errors).
2. **Business Layer:** encapsulates the application's core logic and connects the Presentation and Persistence layers.
    - **Facade Interface:** the main entry point for the Presentation Layer. It acts as a coordinator to receive requests from the Controller, decides which business logic to invoke and delegates tasks to respective service models.
    - **Service Models:** Contain reusable business logic and rules, often depending on multiple classes or DAOs. For example:
        - *UserService:* manage user profiles (hosts and guests), update account information
        - *PlaceService*: add or remove listings, update availability calendars, manage photos, descriptions and amenities
        - *ReviewService:* Post a review after a stay, calculate average ratings
        - *BookingService:* checks if place is available, creates a reservation, triggers payment processing, notify host and guest
    - **Classes:** core entities representing the application's data and simple behaviors, such as:
        - **User**: Stores account information and booking history.
        - **Place**: Represents a property available for rent.
        - **Review**: Captures user feedback on places.
        - **Amenity**: Represents features or services (e.g., Wi-Fi, parking).
        - **Reservation**: Represents bookings made by users.

3. **Persistence Layer:** Manages data storage and retrieval via the Repository, which manages communication and access directly with the database.

    - **Repository:** responsible for managing access to data sources. Its main role is to encapsulate all the logic required to retrieve, store, update, and delete data, typically interacting with a database.
        - **Create:** Insert new records (e.g., adding a new Place).
        - **Read**: Retrieve existing records (e.g., fetching reviews for a Place).
        - **Update**: Modify existing records (e.g., updating reservation dates).
        - **Delete**: Remove records (e.g., deleting a Review).
    - **Database (MySQL):** Underlying storage mechanism for application data.

Design Decisions and Rationale

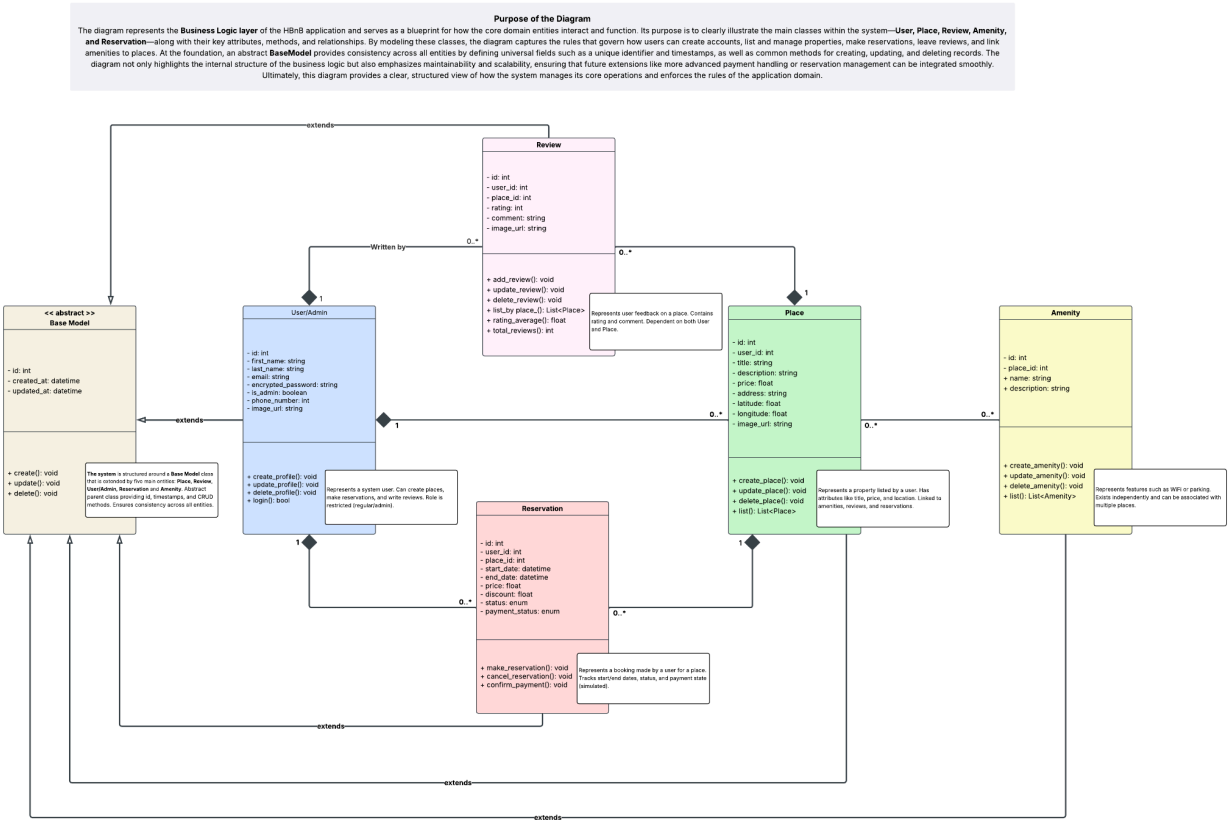
1. **Layered Separation**
   The application was divided into Presentation, Business, and Persistence layers to achieve a clean separation of concerns. Each layer has distinct responsibilities, improving modularity and is easier to maintain as changes in one layer are less likely to have a significant impact on other layers, reducing the risk of unwanted consequences.
2. **Implementing a Facade Pattern**
   A facade pattern provides a simplified and unified interface between the presentation layer and the underlying business logic. It allows controllers or UI components to perform complex operations—such as creating a reservation, processing payment, and sending notifications—through a single, high-level method call. By hiding the complexity of multiple interacting services, the Facade reduces coupling between layers.
3. **Adding a Service Layer/Model**
   Keeping the Facade simple ensures its role is limited to coordinating requests and responses, preventing it from becoming cluttered with complex logic. This design also supports scalability—new rules or integrations can be added to the Service Layer without modifying the Facade, making the system easier to maintain, test, and extend as the project evolves.
4. **Data Access Objects**
   Using DAOs within the Persistence Layer makes database operations interchangeable and more maintainable, as changes to the database system (e.g., switching from MySQL to another database) will minimally impact upper layers.

Business Logic Layer

The Business Logic Layer defines the core domain entities of the HBnB application and enforces the rules that govern their behaviour. This layer ensures that users can interact with the system in a consistent, reliable way by providing the essential operations for account management, property listings, reservations, amenities, and reviews. To maintain uniformity across all entities, the layer is built on top of an abstract Base Model, which provides a unique identifier and timestamp fields, along with standard methods for creation, update, and deletion.

Class Diagram

The diagram below represents the structure of the Business Logic Layer. It highlights the five main entities—**User, Place, Review, Amenity, and Reservation**—and shows their attributes, methods, and relationships. All entities extend from the Base Model class to ensure consistency.



Key Components

BaseModel (abstract)

- Defines universal attributes and methods to ensure consistency across all entities
- Fields: id, created_at, updated_at

User

- Represents system users where users can create places, make reservations, and post reviews
- Fields: id, first_name, last_name, email, encrypted_password, is_admin, phone_number, image_url

Place

- Represents properties listed by users. Places are associated with reviews, amenities, and reservations.
- Fields: id, user_id, title, description, price, address, latitude, longitude, image_url

Review

- Represents user feedback on a place. Reviews are strongly tied to both a User and a Place
- Fields: id, user_id, place_id, rating, comment, image_url

Amenity

- Represents features such as WiFi or parking. Amenities exist independently and can be linked to multiple places through a many-to-many relationship.
- Fields: id, place_id, name, description

Reservation

- Represents a booking made by a user for a place. Reservations are dependent on both a user and a place, modeled as composition relationships.
- Fields: id, user_id, place_id, start_date, end_date, status, payment_status

Design Decisions and Rationale

1. **Inheritance with BaseModel**
    - An abstract base model was introduced to reduce redundancy and ensure every entity has a consistent lifecycle field and methods
2. **Entities that are Composition relationships**
    - User & Place
        - A User *owns* their places. If the User is deleted, all their places should also be removed.
        - This models a composition relationship because the Place cannot exist without its owning User
    - User & Review

        - Reviews are tied directly to Users. If the User is deleted, their reviews should also disappear.
        - This ensures no orphaned reviews exist in the system.
    - **Place & Review**
        - Reviews are also tied to a specific Place. If the Place is deleted, its associated reviews should also be removed
        - Together, this means a Review is doubly dependent: on both a User and a Place.
    - **User & Reservation**
        - Reservations cannot exist without the User who made them. Deleting the User removes their reservations.
    - **Place & Reservation**
        - Reservations cannot exist without the Place being booked. If the Place is deleted, associated reservations must also be deleted.
3. **Separation of concerns between models**
    - Reservation was introduced as a standalone entity rather than being embedded in Place or User.
    - This isolates booking logic, making the system easier to extend with features like cancellation policies or payment history later.
4. **Support for future extensibility**
    - The design leaves room for future services (e.g., PaymentService, NotificationService) without breaking existing logic.
    - By isolating responsibilities, new classes can integrate smoothly with the existing entities

API Interaction Flow

The purpose of the sequence diagrams is to illustrate the interactive flow between the layers of the HBnB Evolution application for key API calls. These diagrams represent the step-by-step communication between the Presentation Layer (API/Services), the Business Logic Layer (core models and operations), and the Persistence Layer (repositories and database).

The diagrams clarify how user requests are processed, how business rules are applied, and how data is stored and retrieved. They also demonstrate how the Facade pattern simplifies interactions by providing a unified interface between layers.

User Registration

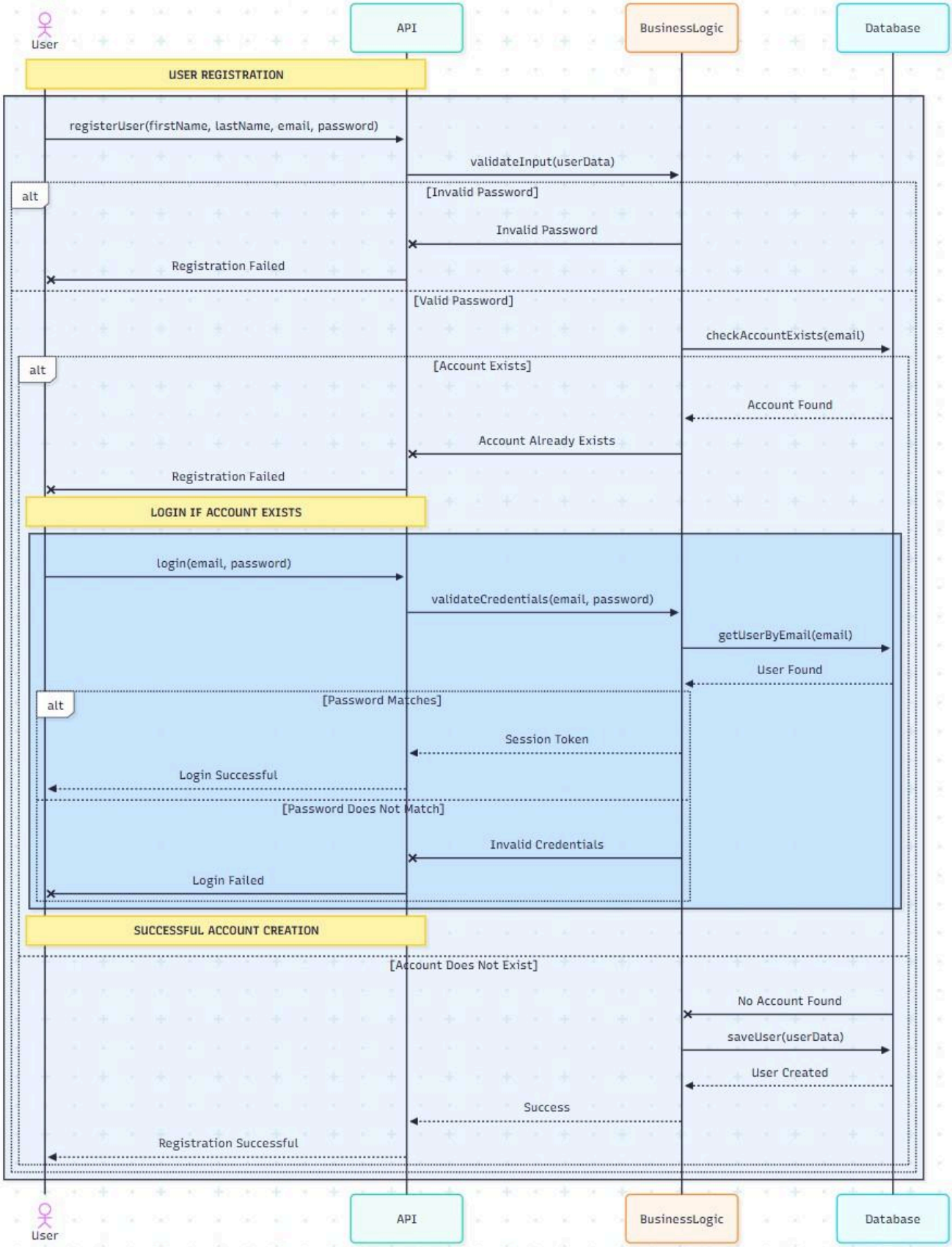
This sequence diagram illustrates the user flow of account creation and login. Registration validates input (e.g., email uniqueness and password rules) before persisting data. If the account already exists, the flow transitions to login where credentials are checked.

Key Components:

- actor User (client)
- participant API (Presentation Layer)
- participant BusinessLogic (Business Layer)
- participant Database as (Persistence Layer)

Flow in Architecture:

- User data flows from the API → Business Logic → Database for validation and persistence. Responses return with either registration success, login success, or error messages.


User
API
BusinessLogic
Database
USER REGISTRATION
registerUser(firstName, lastName, email, password)
validateInput(userData)
alt
[Invalid Password]
Invalid Password
Registration Failed
[Valid Password]
checkAccountExists(email)
alt
[Account Exists]
Account Found
Account Already Exists
Registration Failed
LOGIN IF ACCOUNT EXISTS
login(email, password)
validateCredentials(email, password)
getUserByEmail(email)
User Found
alt
[Password Matches]
Session Token
Login Successful
[Password Does Not Match]
Invalid Credentials
Login Failed
SUCCESSFUL ACCOUNT CREATION
[Account Does Not Exist]
No Account Found
saveUser(userData)
User Created
Success
Registration Successful
User
API
BusinessLogic
Database

Place Creation

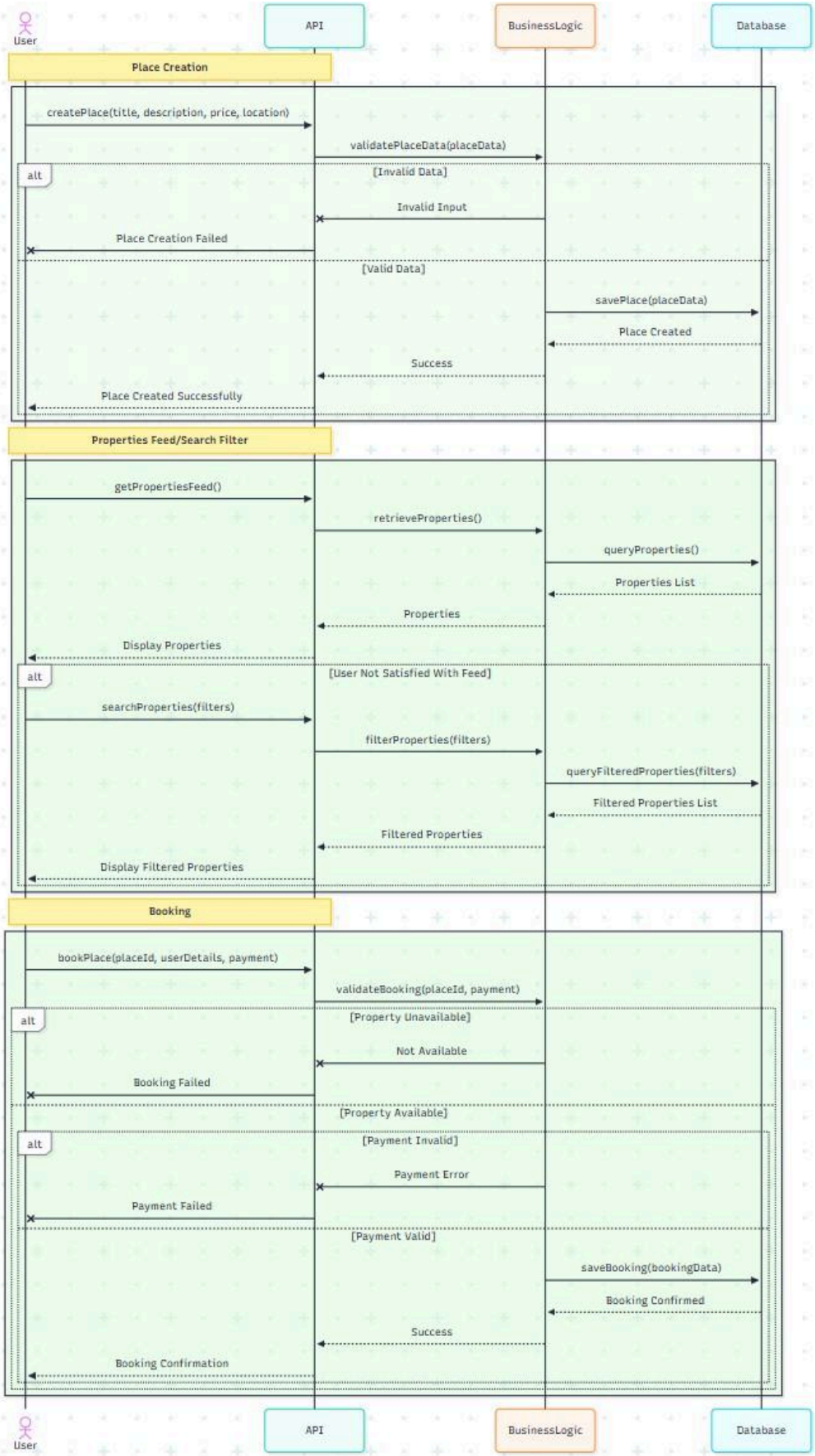
This diagram demonstrates how users interact with the system to create listings, browse/search available properties, and book a place. Only authenticated users can create places, and each place must be linked to an owner. The Business Logic enforces ownership and booking rules, including availability and payment validation.

Key Components:

- actor User (client)
- participant API (Presentation Layer)
- participant BusinessLogic (Business Layer)
- participant Database as (Persistence Layer)

Flow in Architecture:

- **Place creation:** API receives details → Business Logic validates ownership and data → Repository saves the place → Database confirms creation.
- **Browsing/search:** API forwards requests to Business Logic → Database queried for properties → Results returned to user.
- **Booking:** API sends booking request → Business Logic checks availability and validates payment → Booking saved to Database → Confirmation returned.


User
API
BusinessLogic
Database
Place Creation
createPlace(title, description, price, location)
validatePlaceData(placeData)
alt
[Invalid Data]
Invalid Input
Place Creation Failed
[Valid Data]
savePlace(placeData)
Place Created
Success
Place Created Successfully
Properties Feed/Search Filter
getPropertiesFeed()
retrieveProperties()
queryProperties()
Properties List
Properties
Display Properties
alt
[User Not Satisfied With Feed]
searchProperties(filters)
filterProperties(filters)
queryFilteredProperties(filters)
Filtered Properties List
Filtered Properties
Display Filtered Properties
Booking
bookPlace(placeId, userDetails, payment)
validateBooking(placeId, payment)
alt
[Property Unavailable]
Not Available
Booking Failed
[Property Available]
alt
[Payment Invalid]
Payment Error
Payment Failed
[Payment Valid]
saveBooking(bookingData)
Booking Confirmed
Success
Booking Confirmation
User
API
BusinessLogic
Database

Review Submission

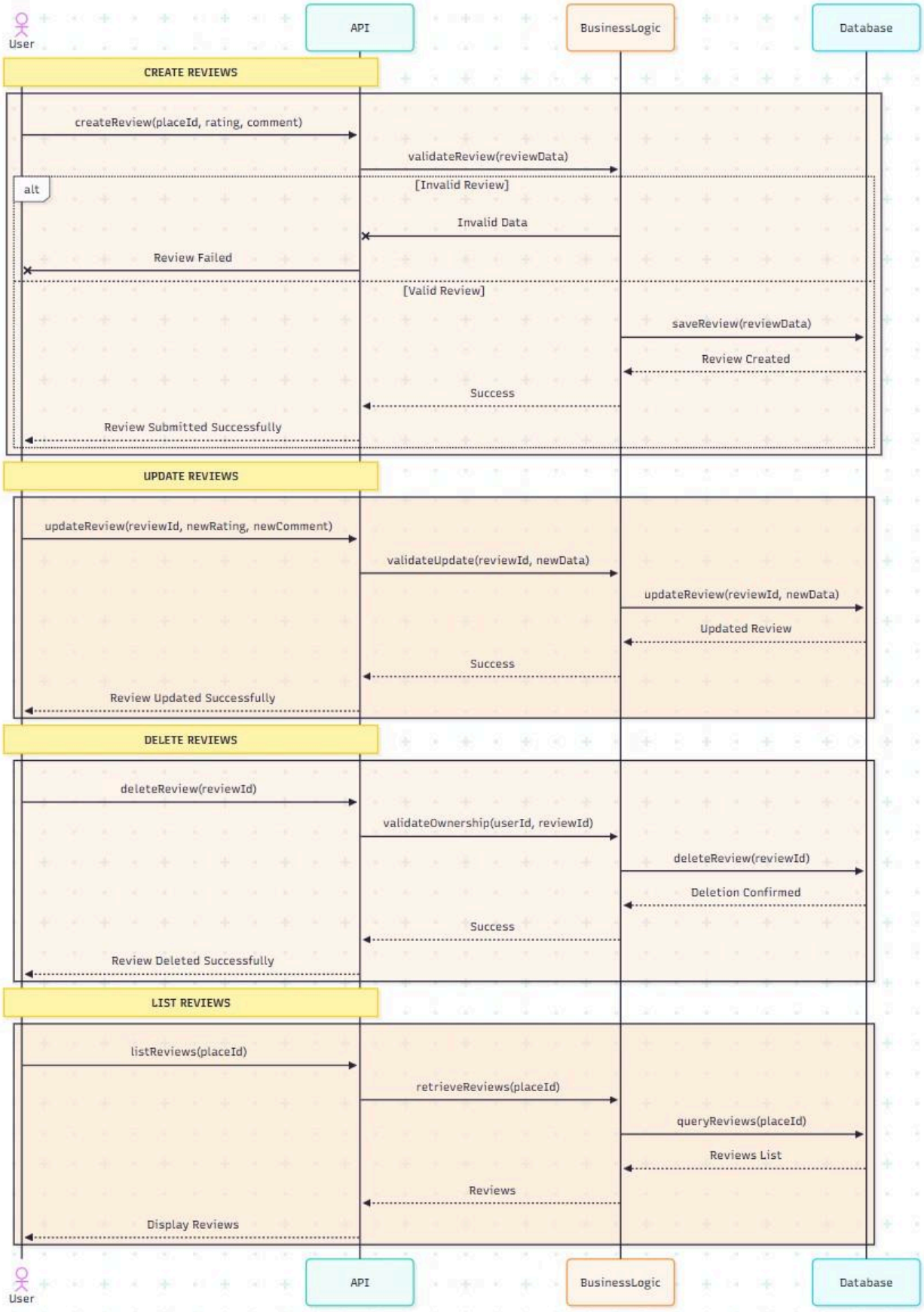
This diagram covers the complete lifecycle of reviews: creation, update, deletion, and retrieval. Reviews must be linked to both a place and a user. The Business Logic validates rating values, ensures the referenced place exists, and enforces ownership for updates and deletions.

Key Components:

- actor User (client)
- participant API (Presentation Layer)
- participant BusinessLogic (Business Layer)
- participant Database as (Persistence Layer)

Flow in Architecture:

- **Create Review:** API forwards submission → Business Logic validates → Repository saves → Database confirms.
- **Update Review:** API forwards update → Business Logic checks ownership and data → Repository updates → Database confirms.
- **Delete Review:** API forwards delete → Business Logic validates ownership → Repository deletes → Database confirms.
- **List Reviews:** API requests reviews → Business Logic queries repository (data access layer) → Database returns list → User sees reviews.


User
API
BusinessLogic
Database
CREATE REVIEWS
createReview(placeId, rating, comment)
validateReview(reviewData)
alt
[Invalid Review]
Invalid Data
Review Failed
[Valid Review]
saveReview(reviewData)
Review Created
Success
Review Submitted Successfully
UPDATE REVIEWS
updateReview(reviewId, newRating, newComment)
validateUpdate(reviewId, newData)
updateReview(reviewId, newData)
Updated Review
Success
Review Updated Successfully
DELETE REVIEWS
deleteReview(reviewId)
validateOwnership(userId, reviewId)
deleteReview(reviewId)
Deletion Confirmed
Success
Review Deleted Successfully
LIST REVIEWS
listReviews(placeId)
retrieveReviews(placeId)
queryReviews(placeId)
Reviews List
Reviews
Display Reviews
User
API
BusinessLogic
Database

Amenities And Fetching Places

This diagram shows how users retrieve a list of available amenities. Business Logic handles query abstraction so the API only requests high-level operations, while the persistence layer handles direct database interactions.

Key Components:

- actor User (client)
- participant API (Presentation Layer)
- participant BusinessLogic (Business Layer)
- participant Database as (Persistence Layer)

Flow in Architecture:

User sends a request to the API → Business Logic retrieves data from the repository → Database returns amenities → API sends results back to the user.