Technical Design Document: HBnB Project

Project: HBnB Project (Part 1) **Version:** 1.0 **Date:** DEC, 26, 2026 **Status:** Initial Design Phase

1. Introduction

1.1 Purpose of this Document

The purpose of this Technical Design Document is to provide a comprehensive architectural blueprint for the **HBnB Evolution** application. It consolidates the high-level architectural design, detailed business logic definitions, and API interaction flows into a single reference source. This document serves as the primary guide for the development team during the implementation phase, ensuring consistency in design, logic, and data handling.

1.2 Project Overview

HBnB Evolution is a simplified AirBnB-like web application that facilitates property rentals. The system allows users to register, list properties ("Places"), manage amenities, and leave reviews. The application is designed with scalability and maintainability in mind, utilizing a modular layered architecture to separate concerns between the user interface, business rules, and data storage.

2. High-Level Architecture

2.1 Architectural Pattern

The HBnB Evolution application follows a **Layered Architecture** pattern. This approach groups code into distinct functional layers, ensuring that changes in one layer (e.g., changing the database) have minimal impact on others (e.g., the User Interface).

The system is composed of the following three layers:

- **Presentation Layer (Services & API):** This is the entry point of the application. It exposes RESTful API endpoints that allow users (via frontend or external clients) to interact with the system. It handles HTTP requests and responses but delegates the actual processing to the layer below.
- **Business Logic Layer (Models):** This is the core of the application. It contains the entities (User, Place, Review, Amenity) and the rules that govern them. It ensures that data is valid and that business operations (like calculating a rating or linking an amenity) are executed correctly.

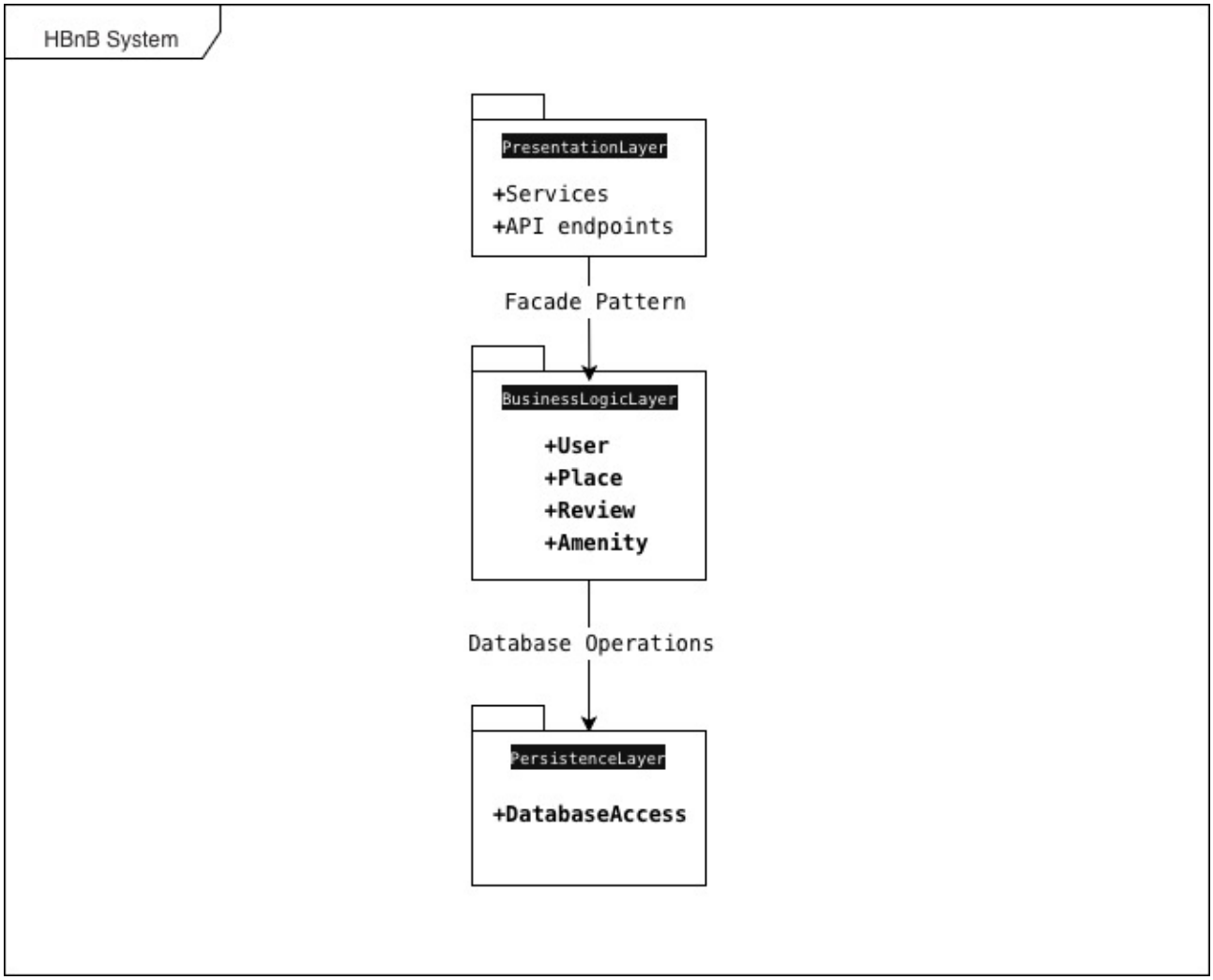
- **Persistence Layer:** This layer is responsible for the permanent storage of data. It abstracts the database interactions (SQL or File Storage) using the Repository pattern, allowing the Business Logic layer to save and retrieve objects without knowing the details of the underlying database technology.

2.2 The Facade Pattern

To manage communication between the Presentation Layer and the Business Logic Layer, we implement the **Facade Pattern**.

The HBnBFacade class acts as a simplified interface (a "front door") for the API. Instead of the API needing to understand the complex relationships between Users, Places, and Repositories, it simply calls methods on the Facade (e.g., create_user()). The Facade then coordinates the necessary actions within the Business Logic and Persistence layers.

2.3 High-Level Package Diagram



3. Business Logic Layer

3.1 Overview

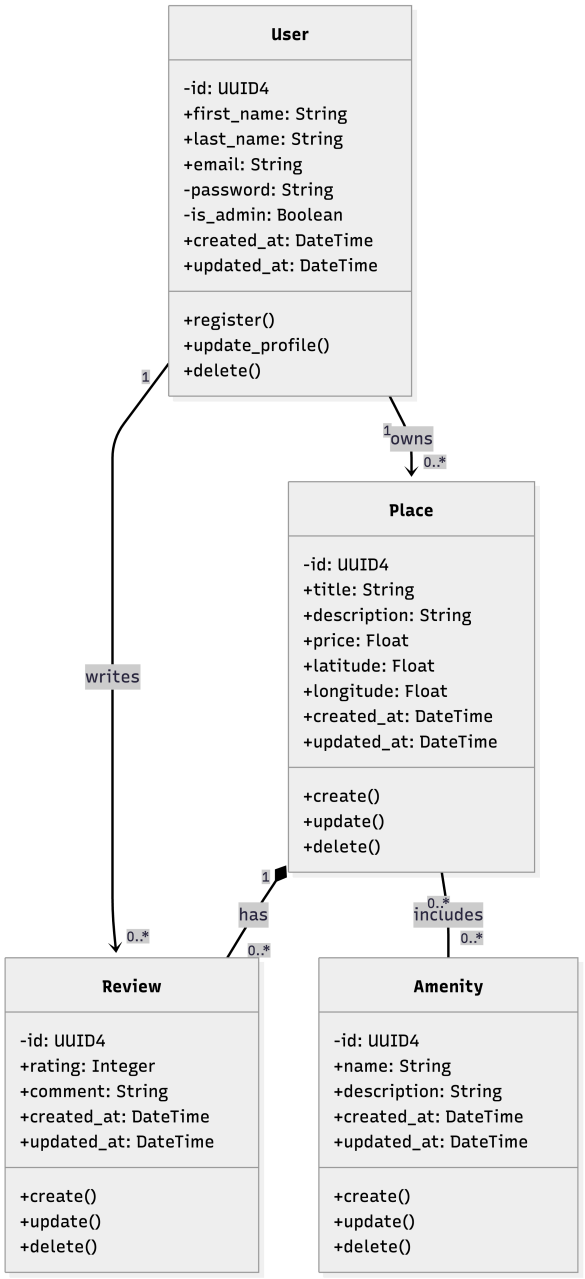
The Business Logic layer is defined by the entities that represent real-world objects in our system. Each entity is implemented as a Class with specific attributes, methods, and relationships.

3.2 Key Entities & Relationships

Based on the system requirements, the class structure is defined as follows:

- **BaseModel:** The parent class for all entities. It manages the unique ID (UUID4) and timestamps (created_at, updated_at) to ensure auditability across the system.
- **User:** Represents a registered member of the platform.
    - *Attributes:* first_name, last_name, email, password, is_admin (boolean).
    - *Relationships:* A User has a **One-to-Many** relationship with **Place** (one user can host many places) and **Review** (one user can write many reviews).
- **Place:** Represents a property listing.
    - *Attributes:* title, description, price, latitude, longitude.
    - *Relationships:* Belongs to a **User** (owner). Has a **Many-to-Many** relationship with **Amenity** and a **One-to-Many** relationship with **Review**.
- **Review:** Represents feedback left by a user.
    - *Attributes:* rating, comment.
    - *Relationships:* Links a specific **User** to a specific **Place**.
- **Amenity:** Represents a feature of a property (e.g., WiFi, AC).
    - *Attributes:* name, description.

3.3 Detailed Class Diagram



4. API Interaction Flow

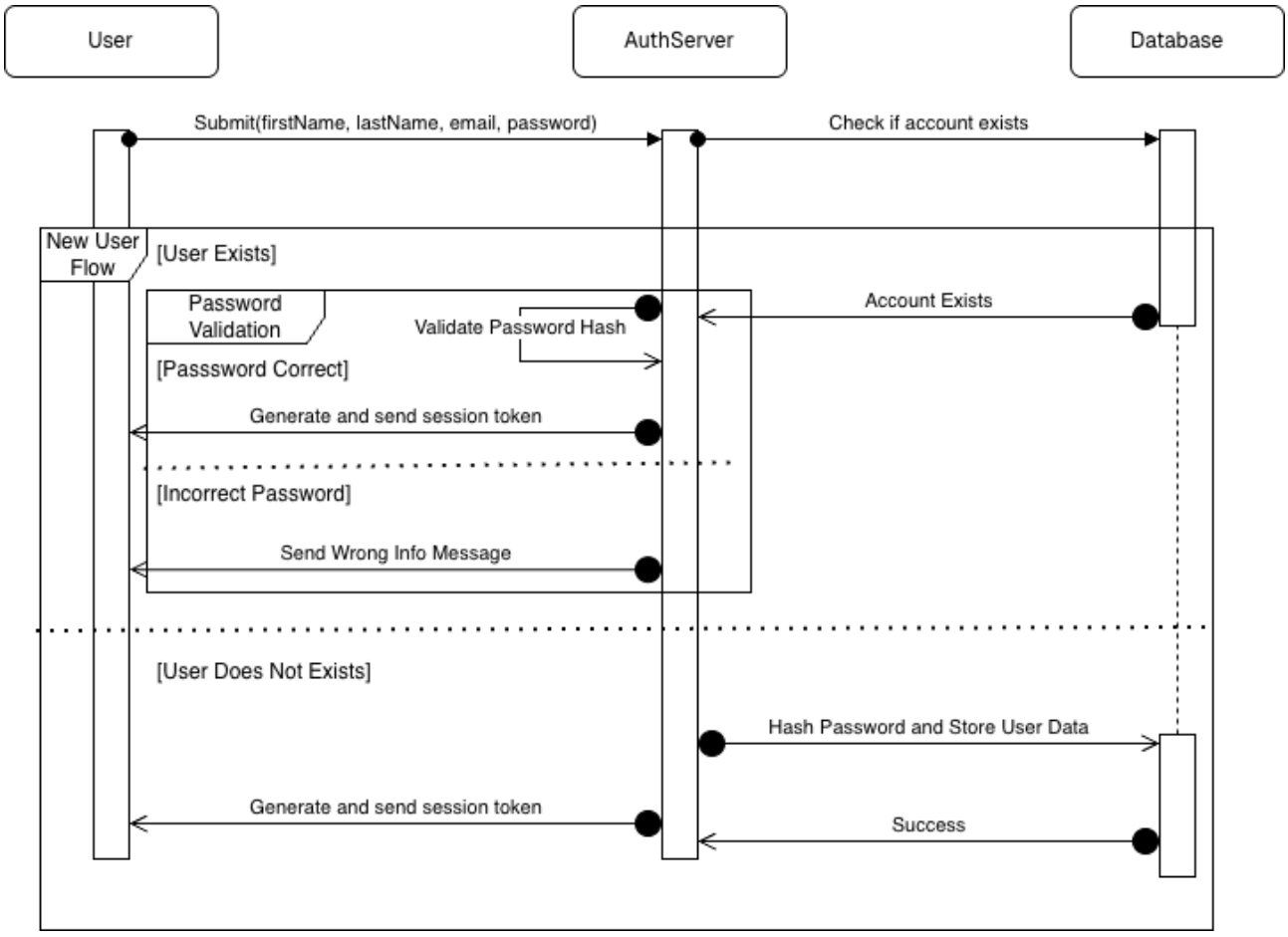
4.1 Overview

This section details how the system handles key user operations. We use Sequence Diagrams to map the step-by-step flow of messages from the API, through the Facade, to the Business Logic and Persistence layers.

4.2 User Registration

Description: This flow describes how a new user account is created.

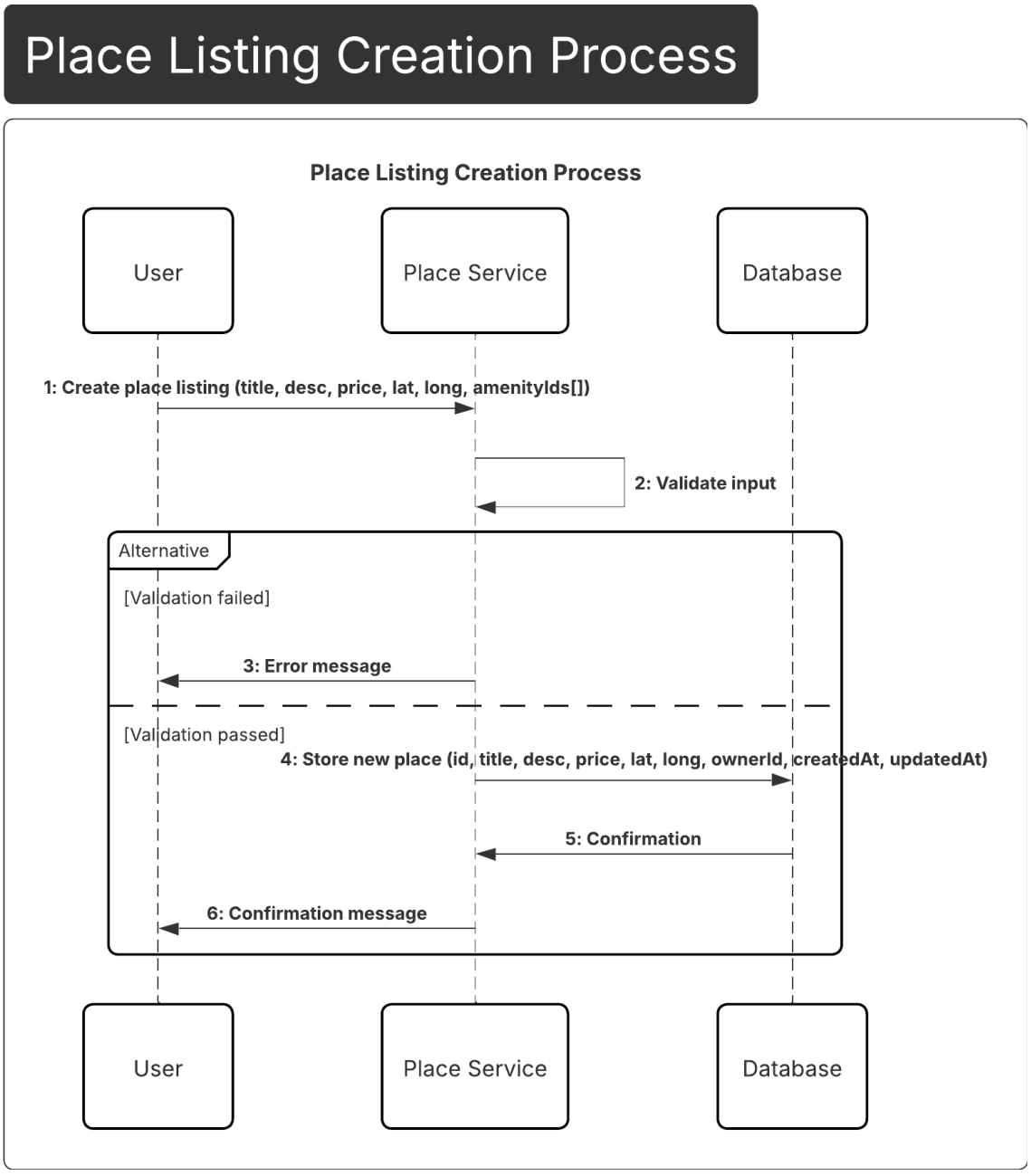
1. **API:** Receives POST /users request with user details.
2. **Facade:** Validates that the email is unique.
3. **Business Logic:** Creates a new User instance.
4. **Persistence:** Saves the new User to the repository.
5. **API:** Returns a success message and the User ID.



4.3 Place Creation

Description: This flow details how a registered user lists a new property.

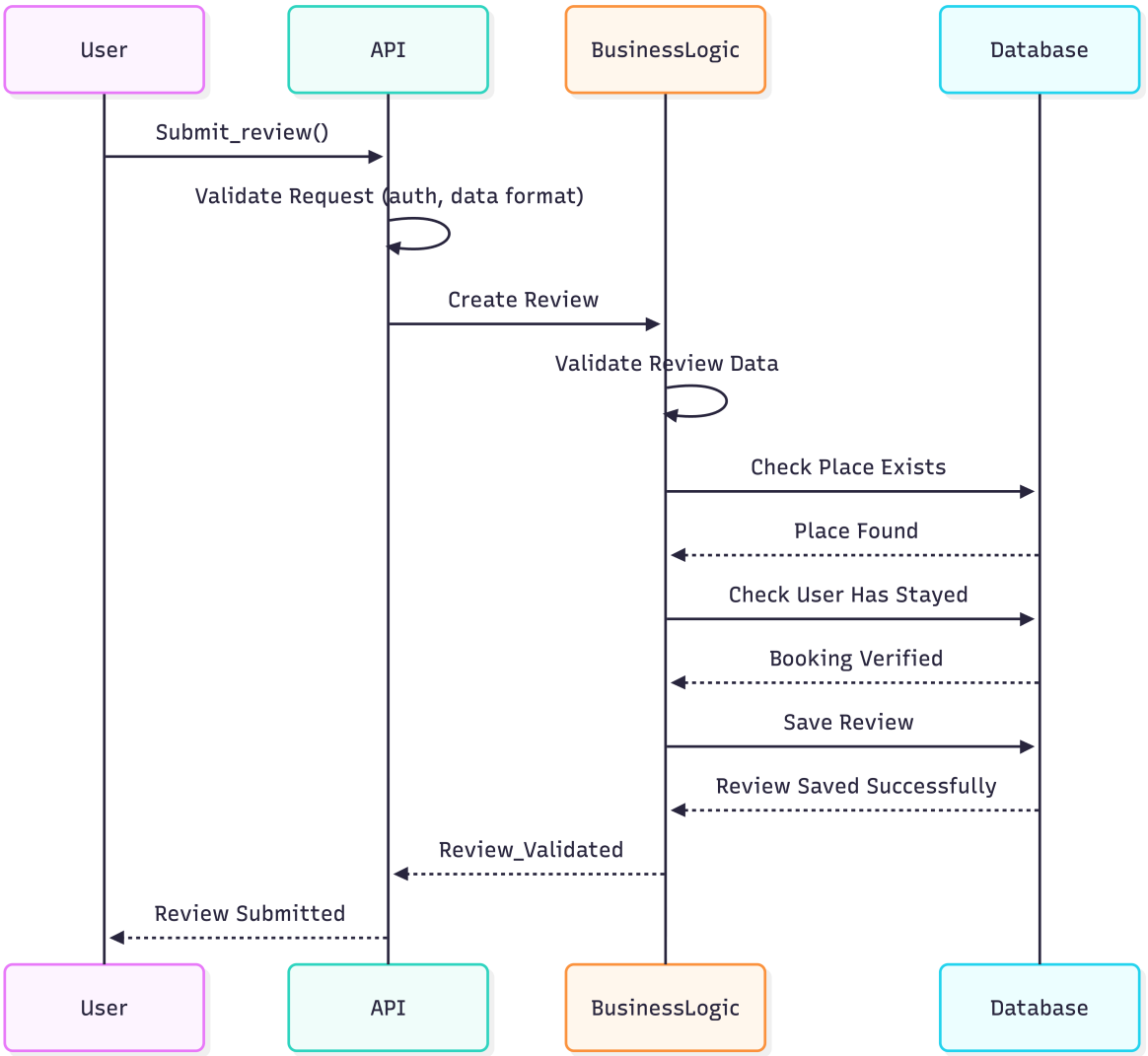
1. **API:** Receives POST /places request.
2. **Facade:** Verifies the user_id corresponds to an existing User.
3. **Business Logic:** Creates a Place object linked to that User.
4. **Persistence:** Stores the Place in the database.



4.4 Review Submission

Description: This flow shows how a review is added to a place.

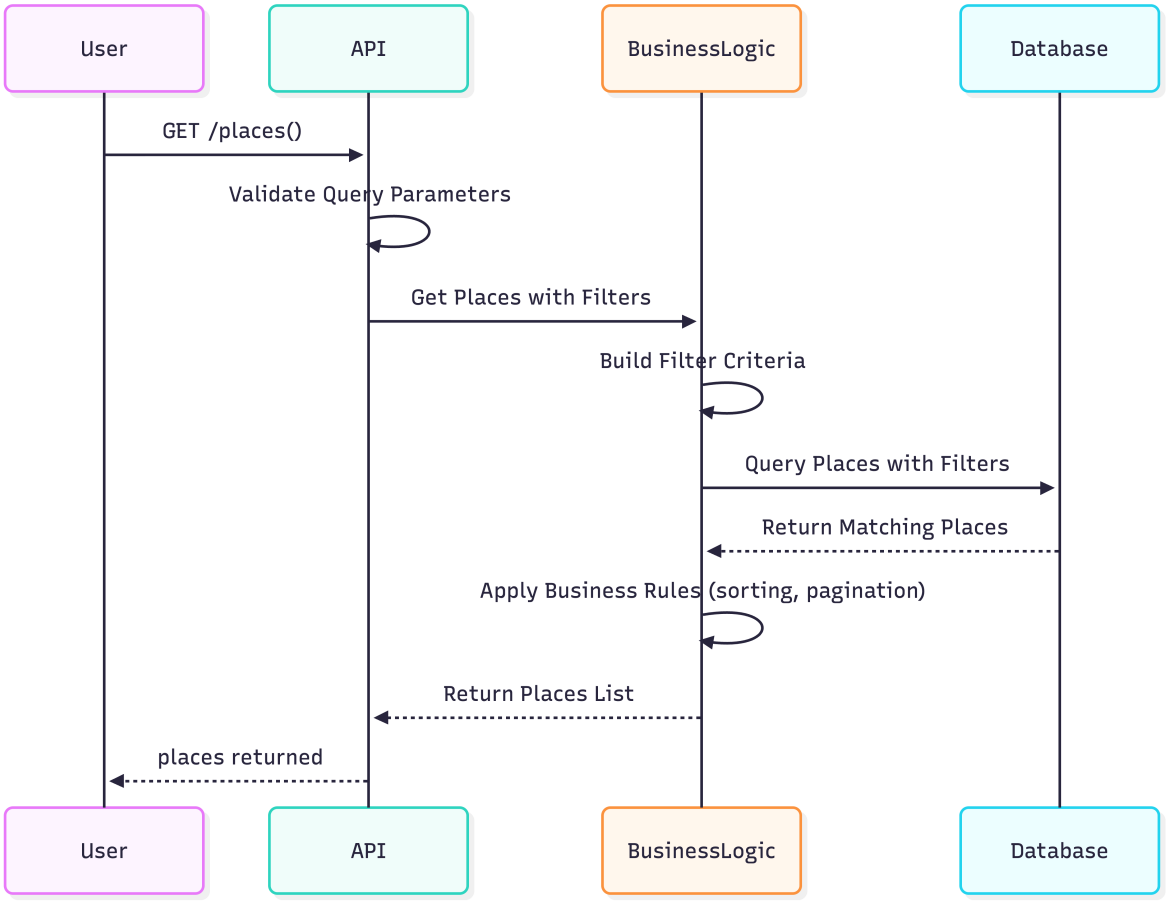
1. **API:** Receives POST /reviews with rating, comment, user_id, and place_id.
2. **Facade:** Validates that both the User and Place exist.
3. **Business Logic:** Instantiates a Review object.
4. **Persistence:** Saves the Review.



4.5 Fetching a List of Places

Description: This flow demonstrates retrieving data for display.

1. **API:** Receives GET /places request.
2. **Facade:** Calls the repository to fetch all place objects.
3. **Persistence:** Returns the list of data.
4. **API:** Serializes the list to JSON and sends it to the client.



5. Conclusion

This architectural documentation defines the roadmap for the HBnB Evolution project. By adhering to the **Layered Architecture** and utilizing the **Facade Pattern**, we ensure that the application remains modular. This separation allows developers to work on API endpoints, business logic, and database operations independently while maintaining a cohesive system. The detailed diagrams included herein provide the necessary clarity to proceed to the implementation phase with confidence.