Welcome to the Hbnb project

from Holberton School

of Jarod Lange and Cyril Iglesias.

Introduction:

HBnb is a rental property platform inspired by AirBnB. the application allows user to :

- Register and manage their user profile.
- Publish property listing for rent.
- Search properties on various criterias.
- Leave reviews for properties.
- Manage amenities.
-

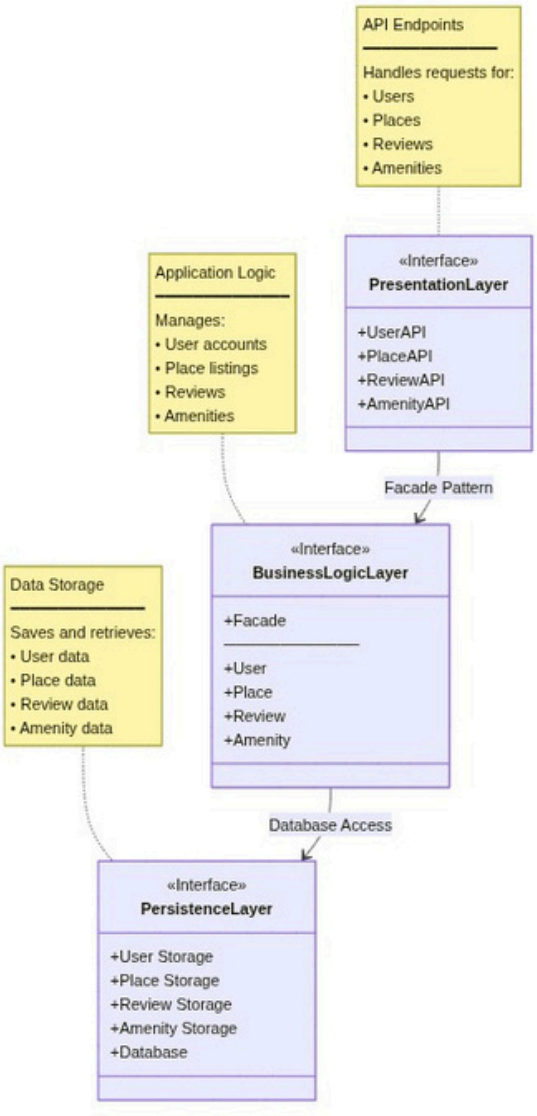
This document present the complete technical architecture of the HBnB system throught:

✅ A package diagram illustrating the 3-layer architecture.
✅ A detailed class diagram of the data model.
✅ Sequence diagrams showing interaction flows.
✅ Technical explanations for each component.

The system of the HBnB is based on several fundamentals architecturals principes:

- Separation of concerns: Each layer has a distinct and well-defined role
- Facade Pattern: Unified interface for inter-layer communication
- Object-oriented inheritance: BaseModel as common parent class
- Data persistence: Audit trail with automatic timestamps

Diagram Package



📌 Objective

The goal of this diagram is to illustrate the three-layer architecture of the HBnB Evolution application. The diagram provides a conceptual overview of how the main components of the system are organized and how they communicate with each other using the Facade Pattern.

🔍 Diagram Explanation

The HBnB Evolution system follows a classic 3-layer architecture that separates concerns into three distinct levels:

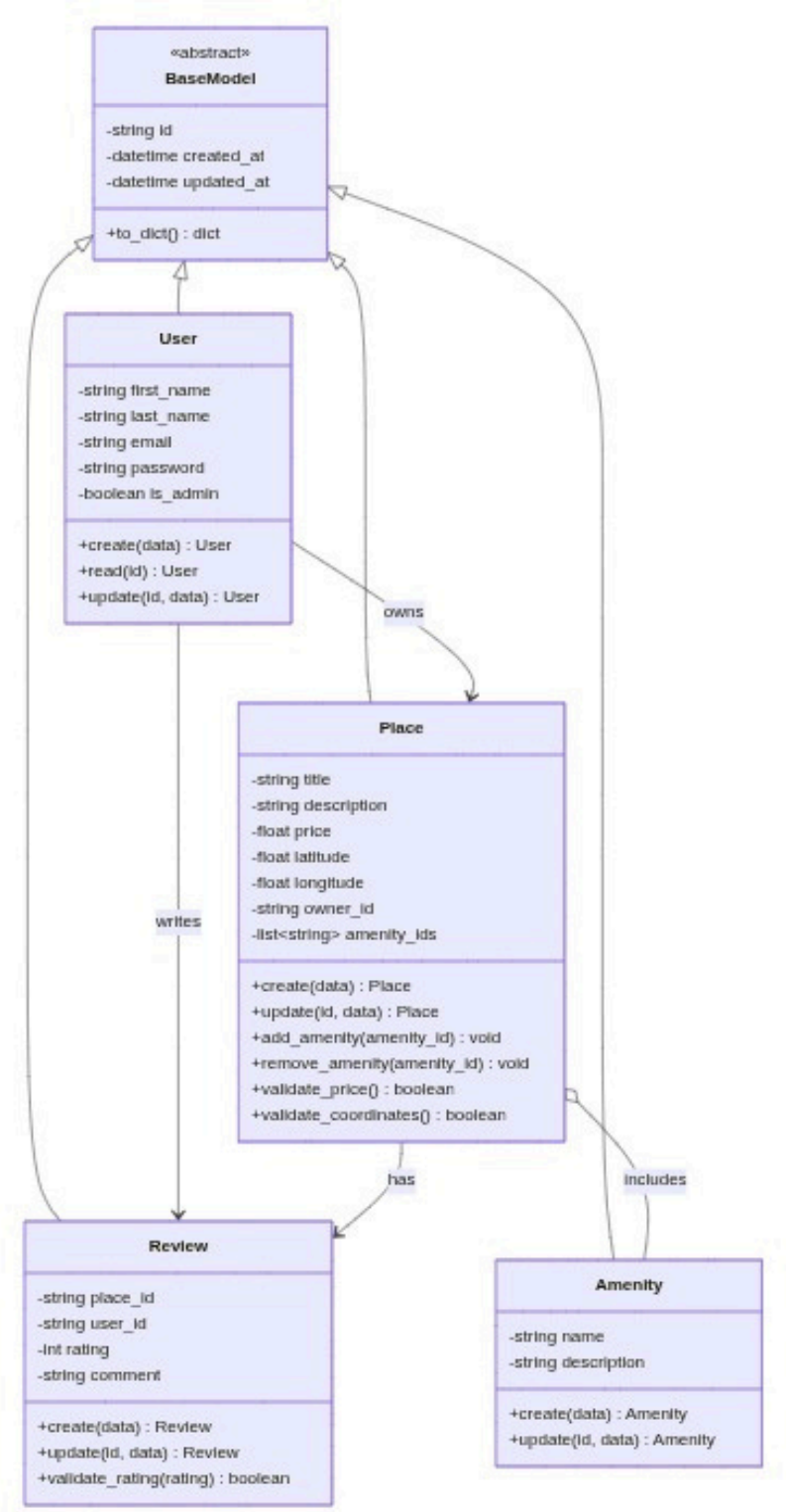
Presentation Layer:

This layer handles the interaction between the user and the application. It includes all the services and APIs that are exposed to users. The Presentation Layer contains four main API components:

- UserAPI: Manages all user-related requests such as registration, login, and profile updates
- PlaceAPI: Handles operations related to property listings including creation, updates, and deletion
- ReviewAPI: Manages review submissions and retrieval for places
- AmenityAPI: Handles the management of amenities that can be associated with places

The Presentation Layer is responsible for receiving HTTP requests, validating the format of incoming data, and returning appropriate JSON responses to the client. It does not contain any business logic.

Business diagram



📌 Objective

The purpose of this diagram is to provide a detailed view of the **Business Logic Layer**, showing the core entities (User, Place, Review, and Amenity), their attributes, methods, and relationships. This diagram serves as the blueprint for the data model of the HBnB Evolution application.

🔍 Diagram Explanation

BaseModel (Abstract Parent Class)

BaseModel is an abstract parent class that provides common attributes inherited by all entities in the system. It includes three essential attributes: a unique identifier (id), a creation timestamp (created_at), and an update timestamp (updated_at). These attributes enable auditing and tracking of all entities throughout the application. The to_dict() method converts objects to dictionary format for JSON serialization. Since BaseModel is abstract, it cannot be instantiated directly—only its subclasses can be created.

User

The **User** class represents people using the platform, whether they are property owners or travelers. It stores personal information including first name, last name, email (which serves as the unique login identifier), password (always stored in hashed form for security), and an admin flag to distinguish administrators from regular users. The class provides CRUD methods (create(), read(), update()) for managing user accounts. A User can own multiple properties and write multiple reviews about places they've visited.

Place

The **Place** class represents rental properties listed on the platform. It contains descriptive information (title and description), pricing information (price per night), geographic location (GPS coordinates with latitude and longitude), ownership information (owner_id referencing the User), and associated amenities (amenity_ids list). The class includes methods for creating and updating places, managing amenities (add_amenity() and remove_amenity()), and validating business rules such as ensuring positive prices and valid GPS coordinates. Each Place belongs to one User and can have multiple Reviews and Amenities.

Review

The **Review** class represents user feedback about properties. It links a specific user to a specific place through foreign keys (user_id and place_id), includes a numeric rating from 1 to 5 stars, and provides a text comment for detailed feedback. The class offers methods for creating and updating reviews, along with rating validation to ensure values stay within the 1-5 range. An important business rule enforced at the Facade level prevents users from reviewing their own properties, maintaining review integrity.

Amenity

The **Amenity** class represents features and services available at properties, such as WiFi, Pool, Parking, or Air Conditioning. Each amenity has a unique name and an optional description providing more details. Amenities are designed to be reusable—one "WiFi" amenity can be associated with multiple properties, ensuring consistency across the platform. The class

provides basic CRUD operations for creating and updating amenities. The many-to-many relationship with Place allows properties to offer multiple amenities and amenities to appear in multiple properties.

🧭 Class Relationships

Inheritance

All four entities (User, Place, Review, Amenity) inherit from **BaseModel**, gaining the common attributes (id, created_at, updated_at) and the to_dict() method automatically. This follows the DRY principle and ensures consistent behavior across all entities.

Associations

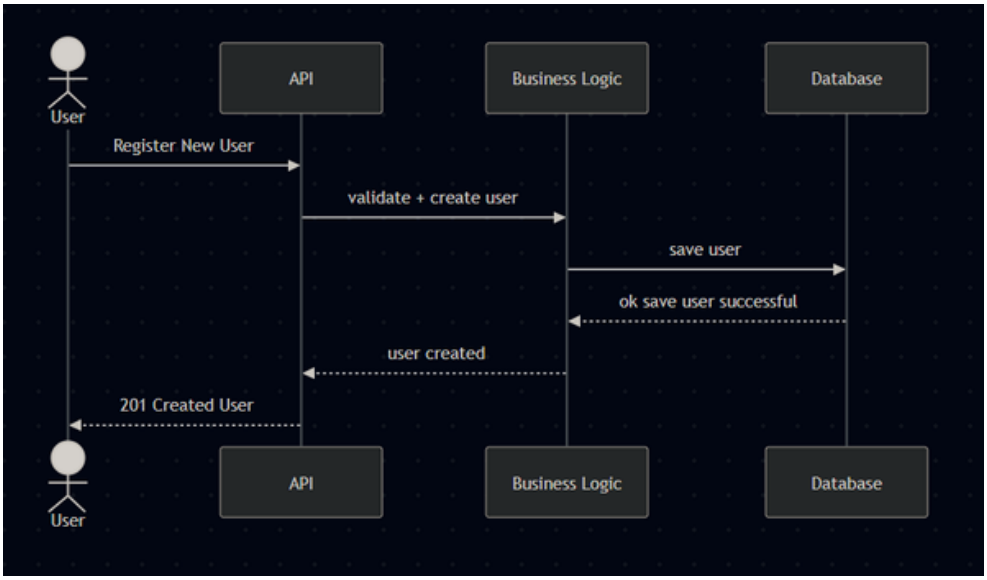
User → **Place (owns)**: One-to-many relationship where a User can own multiple Places, implemented via owner_id in Place.

User → **Review (writes)**: One-to-many relationship where a User can write multiple Reviews, implemented via user_id in Review.

Place → **Review (has)**: One-to-many relationship where a Place can receive multiple Reviews, implemented via place_id in Review.

Place ↔ **Amenity (includes)**: Many-to-many relationship where Places can have multiple Amenities and Amenities can belong to multiple Places, implemented through a join table in the database.

User diagram



📌 Objective

This sequence describes how a new user account is created in the system.

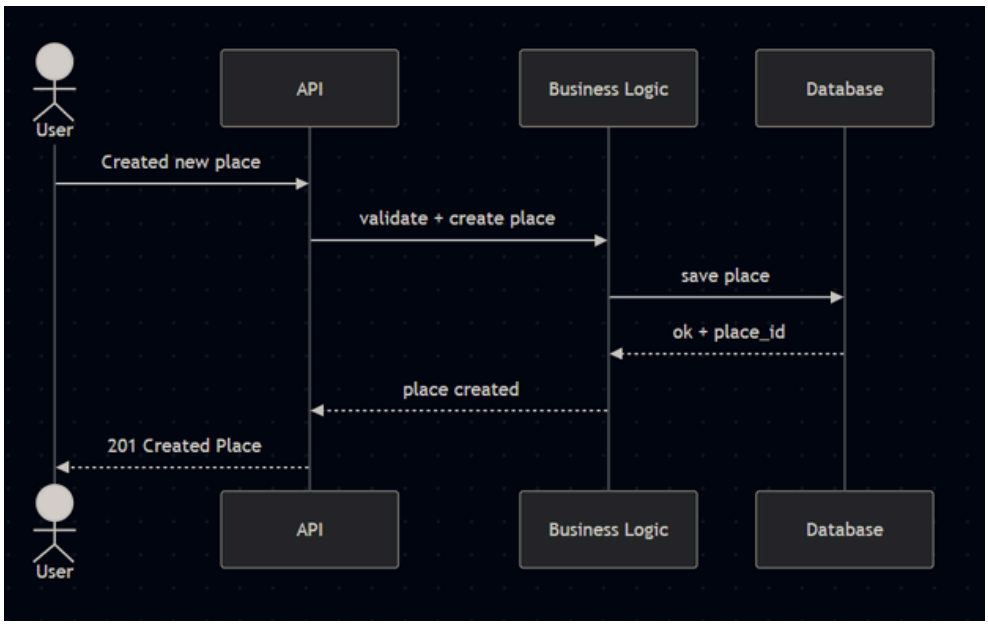
🔍Diagram Explanation

1. The User sends a registration request to the API.
2. The Presentation Layer (UserAPI) receives and validates request format.
3. The request is forwarded to the Business Logic layer.
4. Business Logic:

- Validates email uniqueness
- Hashes the password
- Creates the User entity

5. The entity is saved in the Database layer.
6. The database confirms persistence.
7. A **201 Created** response is returned to the client.

Place Creation diagram



📌 Objective

This sequence describes how a property listing is created.

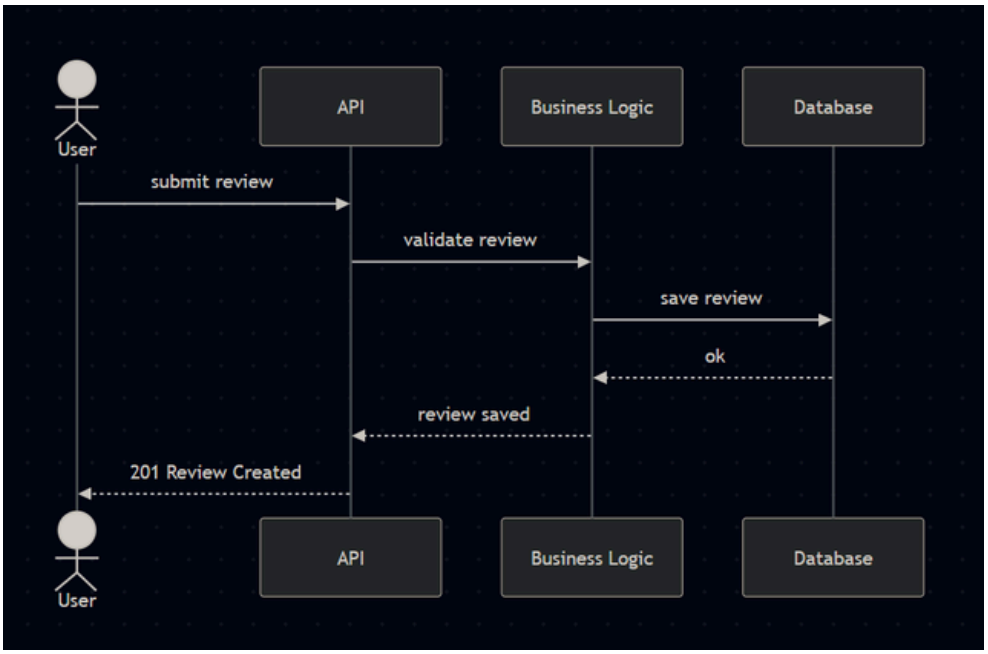
🔍Diagram Explanation

1. The User submits a new place.
2. The API layer validates request structure.
3. The request is passed to the Business Logic layer.
4. Business Logic:

- Validates price > 0
- Validates latitude/longitude
- Verifies owner exists
- Creates Place entity

5. The Place is saved in the database.
6. The system returns **201 Created**

Review Submission diagram



📌 Objective

This sequence describes how a user submits a review for a place.

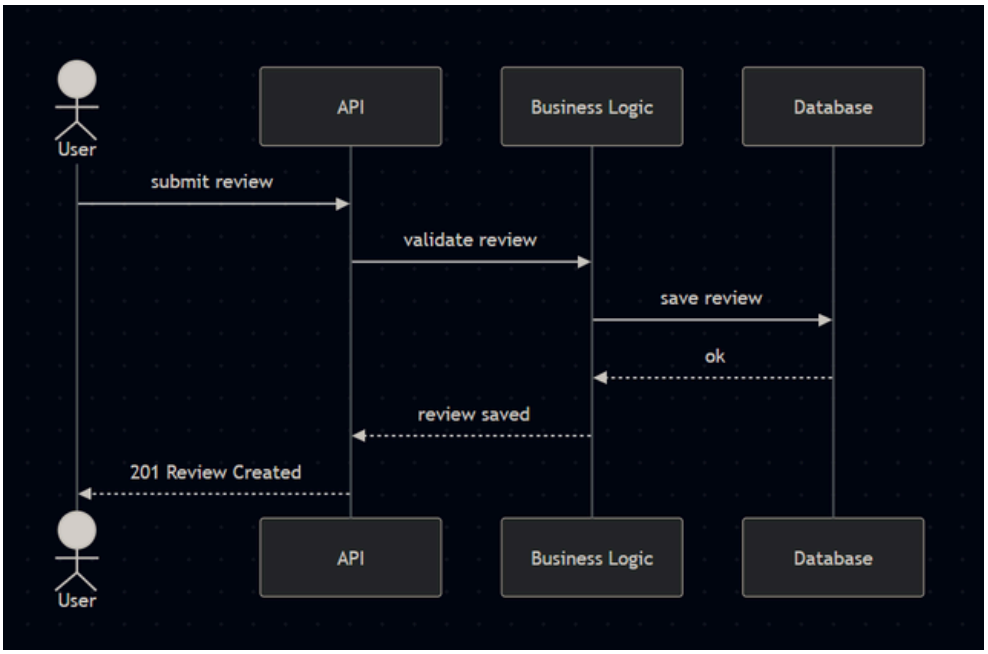
🔍Diagram Explanation

1. The User submits a review. new place.
2. The API layer forwards request to Business Logic.
3. Business Logic:

- Validates rating (1–5)
- Verifies user exists
- Verifies place exists
- Ensures user is NOT reviewing their own property

4. The review is saved in the database.
5. The system returns **201 Review Created**

All Places diagram



📌 Objective

This sequence describes how users retrieve available properties.

🔍Diagram Explanation

1. The User sends a GET request.
2. The API forwards filters (if any) to Business Logic.
3. Business Logic:

- Builds search filters
- Applies optional criteria (price, location, amenities)

4. The database executes the query.
5. Results are returned as a list.
6. The API responds with **200 OK**