HBnB Evolution

Technical Documentation

Architecture, Domain Model & API Design

Lucas LUPON
Mateo MARQUES
Holberton School

February 2026

Introduction

HBnB Evolution is a web application inspired by AirBnB, built as part of the Holberton School curriculum. The project follows a layered architecture and uses the Facade design pattern to keep a clear separation between how users interact with the system, how data is processed, and how it is stored.

This document serves as the technical blueprint for the project. It brings together the architectural decisions, domain models, and API interaction flows into a single reference designed to guide the development process. Each section includes the relevant diagrams along with explanatory notes on the design choices made.

High-Level Architecture

The HBnB application is structured around a three-layer architecture. Each layer has a specific responsibility and communicates only with the layer directly below it, which keeps the system organized and makes each part easier to work on independently.

The Presentation Layer is the top layer. It handles all communication with the user through API endpoints. Its role is to receive requests, pass them to the appropriate service, and return the response. It contains no business rules and no direct access to the database.

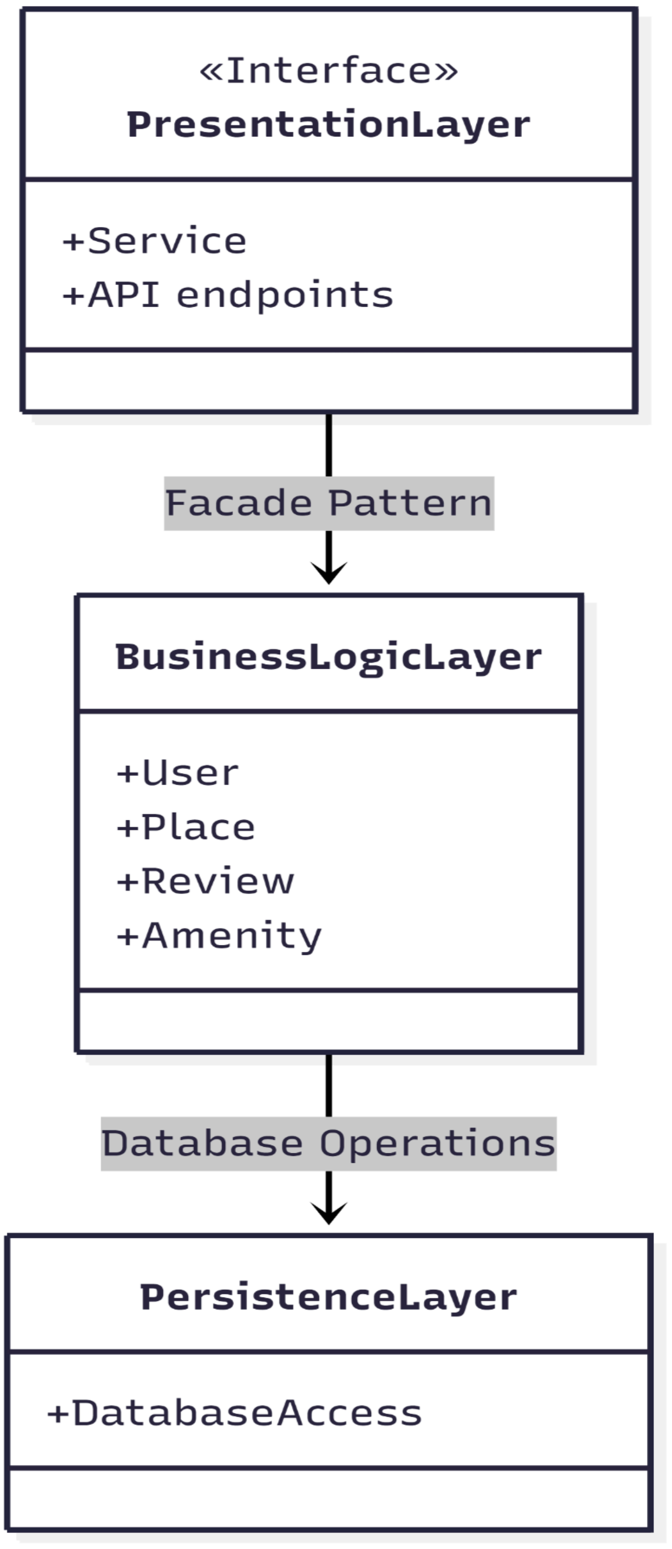
The Business Logic Layer sits in the middle. It contains the core entities of the application — User, Place, Review, and Amenity — along with all the rules that govern how they behave and interact. This is where data is validated, processed, and structured before being sent to storage.

The Persistence Layer is the bottom layer. It is responsible for storing and retrieving data through a DatabaseAccess component. By isolating all database operations in this layer, the rest of the application does not need to know how or where the data is stored, which means the storage technology can be changed without affecting the other layers.

These three layers communicate through well-defined interfaces. The Presentation Layer interacts with the Business Logic Layer through the **Facade pattern** — instead of having the API endpoints call multiple business objects directly, they go through a single, simplified interface that handles the coordination behind the scenes. The Business Logic Layer then communicates with the Persistence Layer through **Database Operations**, sending requests to save, retrieve, update, or delete data without handling any storage details itself. This separation keeps each layer focused on its own responsibility and reduces the dependencies between them.

This approach was chosen because it allows each layer to evolve independently. For example, the database technology could be replaced without rewriting any business rules, or the API could be restructured without touching the data validation logic. The Facade pattern specifically prevents the Presentation Layer from becoming tightly coupled to the internal structure of the Business Logic, which would make future changes risky and difficult to test.

The following diagram illustrates this three-layer structure and the communication paths between each layer.



Business Logic Layer

The Business Logic Layer contains the core domain entities of the HBnB application and enforces the business rules that ensure data consistency and validity.

This layer is responsible for representing the main concepts of the application (users, places, reviews, amenities), managing their relationships, and exposing domain-level behaviors (e.g., password verification, amenity management, rating validation).

The Business Logic Layer does not directly handle database operations. These are delegated to the Persistence Layer through the application architecture (facade/persistence services), ensuring a clean separation of concerns.

---

1. BaseModel (Abstract Parent Class)

Role: Common base class inherited by all domain entities. It standardizes identification and tracking metadata.

Attribute	Type	Description
id	UUID	Unique identifier for each instance
created_at	DateTime	Timestamp of object creation
updated_at	DateTime	Timestamp of last update

Method	Return	Description
save()	void	Persists changes (create/update) by delegating to persistence mechanisms
delete()	void	Removes the entity from storage (delegation to persistence)
to_dict()	dict	Serializes the entity into a dictionary format (useful for API responses / DTO mapping)

---

2. User

Role: Represents a person using the application. A user can own places and write reviews.

Attribute	Type	Description
first_name	string	User's first name

Attribute	Type	Description
last_name	string	User's last name
email	string	User's email address
password_hash	string	Secure hash of the user's password
is_admin	bool	Whether the user has admin privileges

Method	Return	Description
set_password(password)	void	Stores a secure hash instead of the raw password
check_password(password)	bool	Verifies credentials by comparing with the stored hash

Business rules / validation notes:

- **email** should be unique and correctly formatted.
- **password_hash** must never expose the raw password.
- **is_admin** enables privileged operations (authorization decisions often happen at service/facade level but this flag belongs to the domain model).

---

3. Place

Role: Represents a listing created by a user, including its description, location, and pricing data.

Attribute	Type	Description
title	string	The name/title of the place
description	string	Detailed description of the place
price	float	Price per night
latitude	float	Geographic latitude coordinate
longitude	float	Geographic longitude coordinate
owner_id	UUID	Foreign key reference to the owning User

Method	Return	Description
add_amenity(amenity_id)	void	Associates an amenity with the place
remove_amenity(amenity_id)	void	Removes an amenity association from the place

Business rules / validation notes:

- **price** should be non-negative.
- **latitude** should be within [-90, 90] and **longitude** within [-180, 180].
- **owner_id** must reference an existing User (ownership integrity rule).

---

4. Review

Role: Represents feedback written by a user about a specific place. Reviews are crucial for reputation and trust in the platform.

Attribute	Type	Description
user_id	UUID	Author of the review
place_id	UUID	Target place
rating	int	Numerical rating (1 to 5)
comment	string	Text content of the review

Method	Return	Description
is_valid_rating()	bool	Validates that the rating respects the accepted range (1–5)

Business rules / validation notes:

- A review must be linked to exactly one user and exactly one place.
- Rating validation prevents corrupted or meaningless ratings.
- Optional rule: one user may only post one review per place (can be enforced at service layer or by constraints).

---

5. Amenity

Role: Represents a feature that can be associated with a place (e.g., Wi-Fi, parking, pool).

Attribute	Type	Description
name	string	The name of the amenity
description	string	A detailed description of the amenity

Business rules / validation notes:

- Amenity names should be unique or standardized to avoid duplicates.

Relationships & Multiplicities

User ↔ Place (Ownership)

- A User (1) owns 0..* Places. Each Place belongs to exactly one User (via owner_id).
- *Why it matters:* ensures clear ownership, permissions (edit/delete), and accountability.

User ↔ Review (Authorship)

- A User (1) writes 0..* Reviews. Each Review is written by exactly one User (user_id).
- *Why it matters:* traceability of feedback and moderation workflows.

Place ↔ Review

- A Place (1) has 0..* Reviews. Each Review targets exactly one Place (place_id).
- *Why it matters:* supports aggregation (average rating, review listing) and place reputation.

Place ↔ Amenity (Association)

- A Place includes 0..* Amenities. An Amenity can be associated with 0..* Places.
- This is a many-to-many relationship (commonly implemented with a join table such as place_amenities).
- The methods `add_amenity()` and `remove_amenity()` are domain-level operations that manage this association.

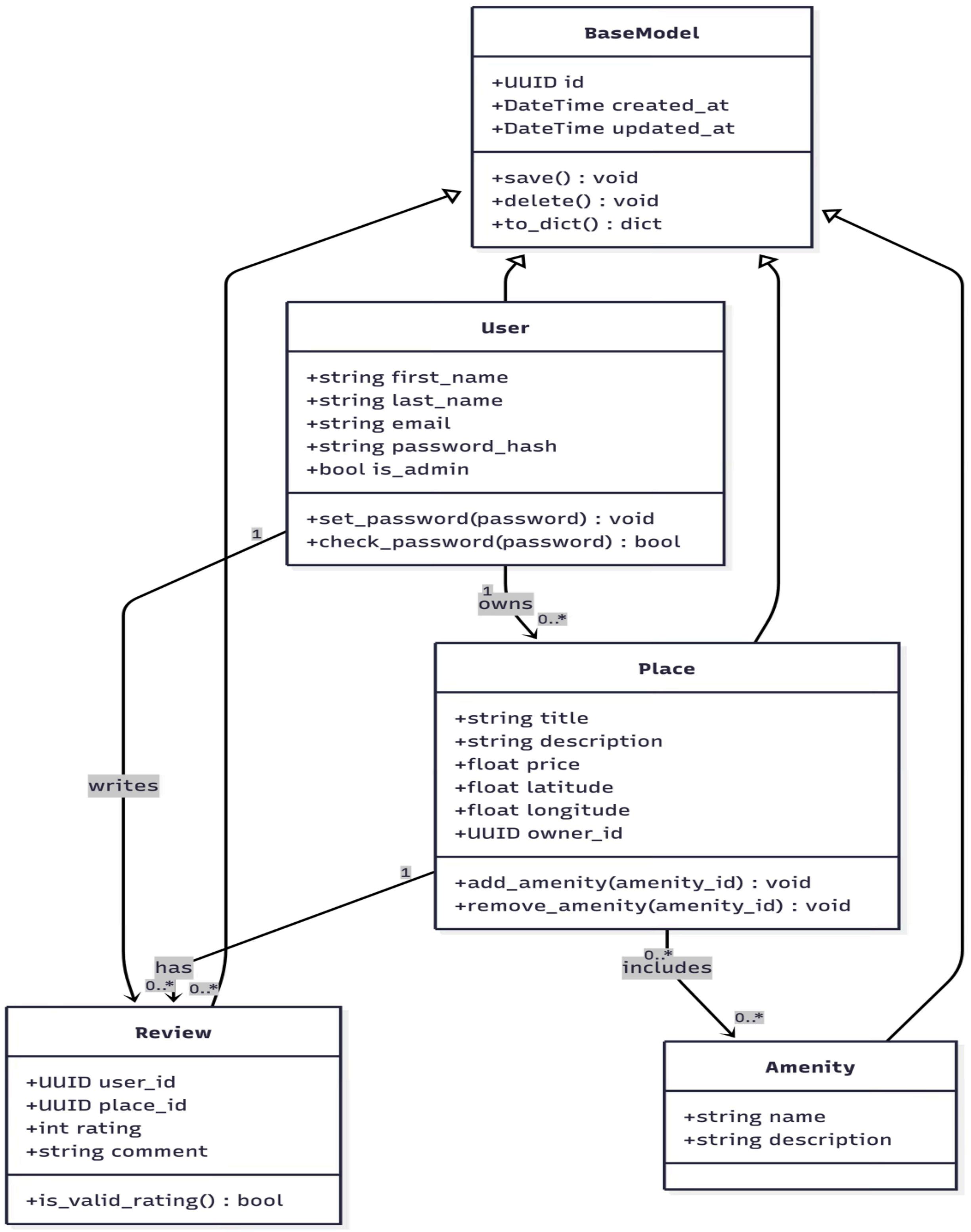
How This Fits the Overall Architecture

The Business Logic Layer is the source of truth for:

- Domain structure (entities + relationships)
- Core behaviors (password handling, rating validation, amenity linking)
- Invariants (ownership integrity, valid coordinates, valid rating bounds)

It supports the Facade/Service layer by providing clean model operations and consistent validation points, while persisting changes through the Persistence Layer.

The following diagram illustrates the complete class structure of the Business Logic Layer, showing all entities, their attributes, methods, and relationships.



API Interaction Flow

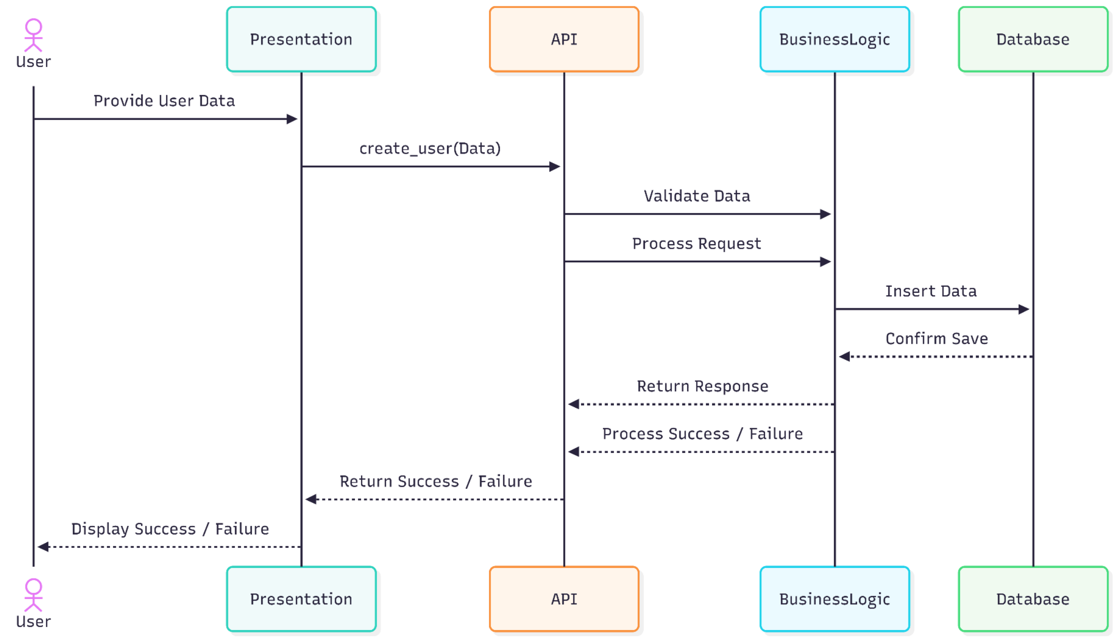
This section describes how data moves through the three-layer architecture described above when a user performs one of the four main API operations. Each sequence diagram maps directly to the layered structure — the Presentation Layer receives the request, the Business Logic Layer processes it, and the Persistence Layer stores or retrieves the data.

In all four flows, data validation is handled by the Business Logic layer rather than the API. This is a deliberate choice — it ensures that the business rules are enforced in one place regardless of how the data enters the system. If a new entry point were added in the future, such as a mobile application or a command-line tool, the same validations would still apply without having to duplicate them.

User Registration

When a user submits their registration data, the Presentation layer receives the request and calls `create_user(Data)` on the Business Logic layer through the Facade. The Business Logic layer validates the data — checking that the email format is correct, required fields are filled, and no duplicate account exists. If everything passes, it sends an insert request to the Database. The Database confirms the save, and the response travels back through each layer until the user sees a success or failure message.

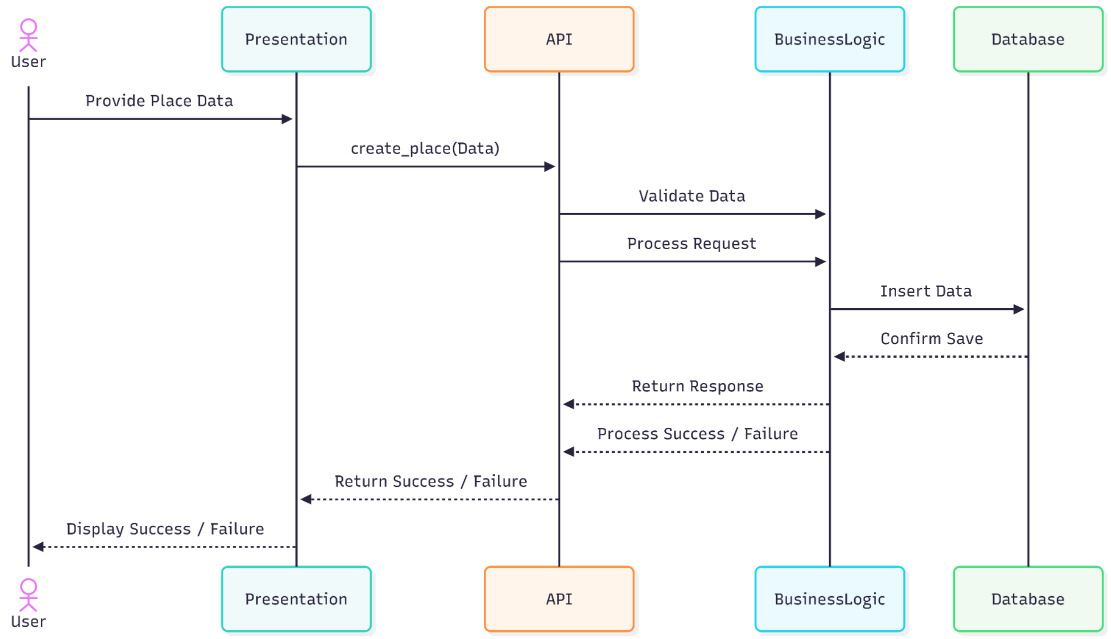
The following diagram illustrates the complete sequence of interactions for user registration, from the initial request to the final response.



Place Creation

The flow for creating a place follows the same pattern. The user provides place data, and the Presentation layer calls `create_place(Data)` through the Facade. The Business Logic layer validates the input — ensuring the price is positive, the coordinates are within valid ranges, and the owner exists in the system. Once validated, the data is inserted into the Database and the result is returned to the user.

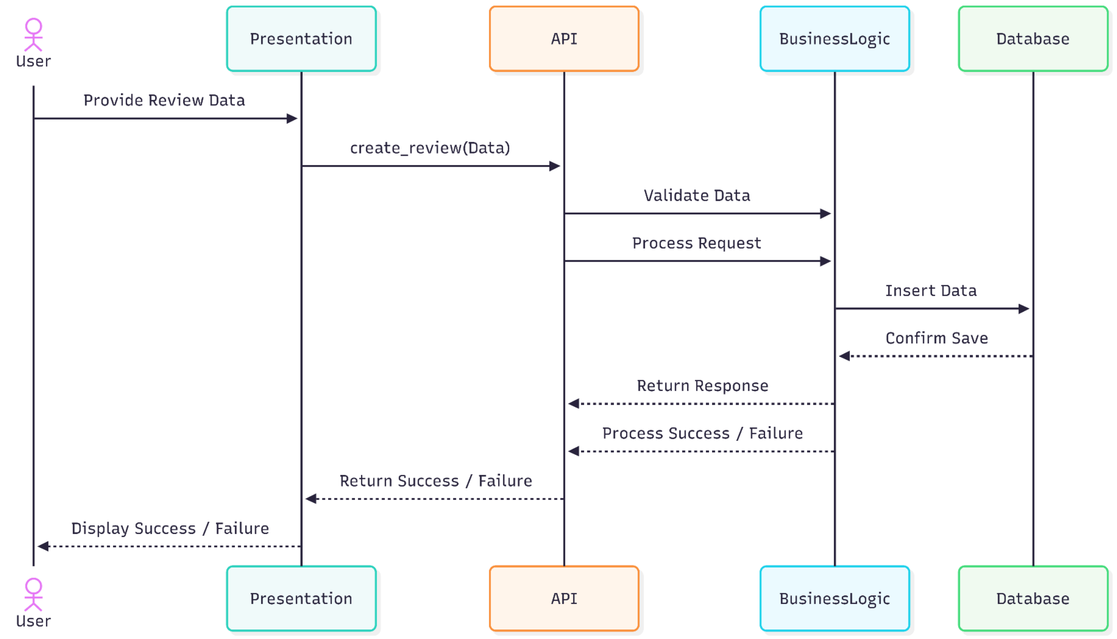
The following diagram illustrates the complete sequence of interactions for place creation.



Review Submission

When a user submits a review, the Presentation layer calls `create_review(Data)` through the Facade. The Business Logic layer checks that the rating is between 1 and 5, that both the user and the place exist, and that the user has not already reviewed that place. If valid, the review is saved to the Database and the user receives confirmation.

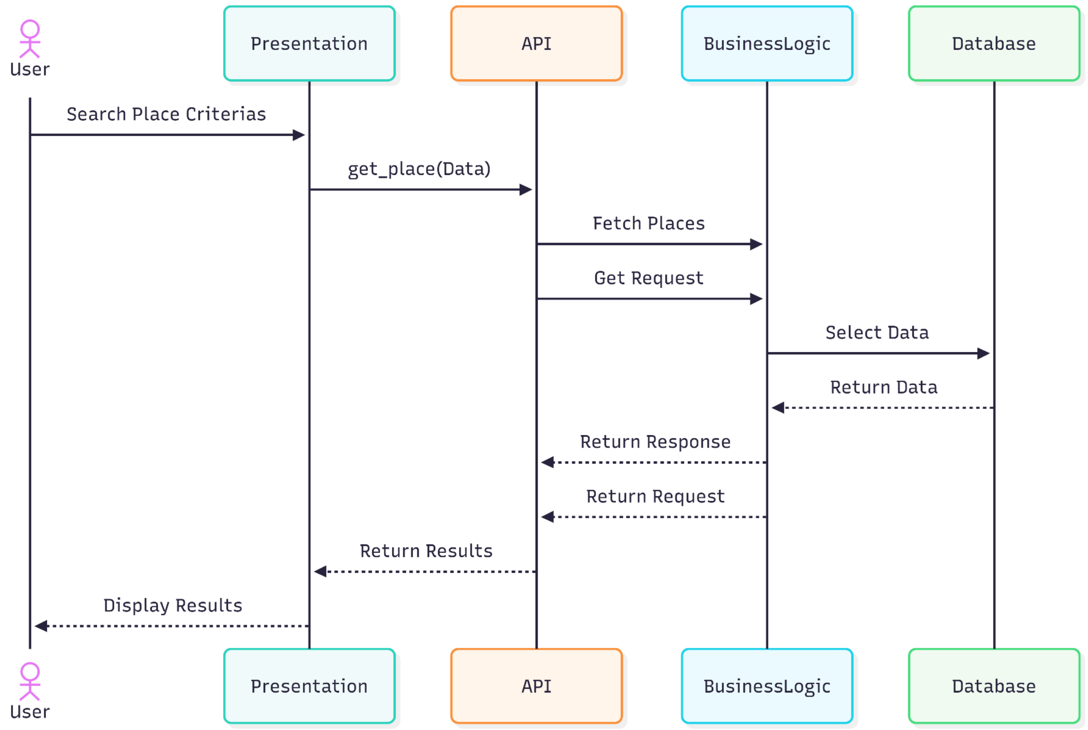
The following diagram illustrates the complete sequence of interactions for review submission.



Place Retrieval

This flow differs from the three above because it reads data instead of creating it. The user submits search criteria, and the Presentation layer calls `get_place(Data)` through the Facade. The Business Logic layer sends a select query to the Database, which returns the matching results. The data then travels back through the layers and is displayed to the user.

The following diagram illustrates the complete sequence of interactions for place retrieval, showing how it differs from the creation flows by using a select operation instead of an insert.



Conclusion

This document provides a complete overview of the HBnB Evolution application design — from its layered architecture and the Facade pattern that connects its layers, to the entities that form its Business Logic, and the step-by-step API flows that bring it all together. It will serve as the main reference throughout the implementation process and will be updated as the project evolves.