Architecture – Entity Overview (Part 1 – Task 3)

1. Introduction

This document constitutes the deliverable for **Part 1** of the **HBnB Evolution** project, a web application inspired by platforms such as AirBnB.
The goal of this project is to strengthen skills in software architecture, object-oriented modeling, REST API design, and the application of design patterns such as the **Façade Pattern**.

In this first phase, the focus is on the **technical design** of the application through various UML modeling tools and a **three-layered architecture** (Presentation / Business Logic / Persistence).
This separation ensures better maintainability, scalability, and code reusability, while enforcing a strict division of responsibilities.

🎯 **Objectives of Part 1**:

- Define the **overall software architecture** of the application.
- Identify the **main business entities**, their relationships, and responsibilities.
- Model the **API interaction flows** using sequence diagrams.
- Formalize the use of **architectural patterns** (such as the Façade Pattern) in a clean and consistent logic.
- Justify design choices based on **software engineering best practices**.

🧱 **Document Structure**:

This deliverable is composed of three main sections:

1. **High-Level Architecture**
   Presentation of the overall architecture of HBnB Evolution, organized into layered structure and integrating the Façade Pattern.
2. **Business Logic Layer – Class Diagram**
   Modeling of the main business entities (**User**, **Place**, **Review**, **Amenity**) and their common superclass (**BaseModel**). Description of attributes, methods, relationships, and object-oriented design principles.
3. **API Interaction Flow – Sequence Diagrams**
   Analysis of four representative use cases of the application (creating a user, a place, a review, and searching for places). Each case is broken down step-by-step, respecting the **Presentation** → **Business** → **Persistence** data flow.

Through these various representations, this document aims to demonstrate the **overall consistency of the software design**, while serving as a **solid foundation for the technical implementation** in the next phases of the project.

2. High Level Architecture

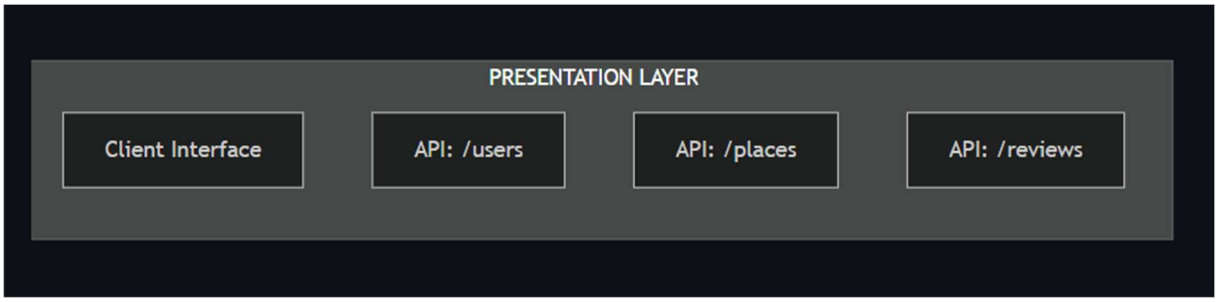
This UML diagram represents the software architecture of the HBnB Evolution application, inspired by an AirBnB-like model.

It adopts a three-layered structure following a classic layered architecture:

- Presentation Layer
- Business Logic Layer
- Persistence Layer

It also highlights the use of the Façade Pattern, in accordance with the project requirements.

🧱 1. Presentation Layer

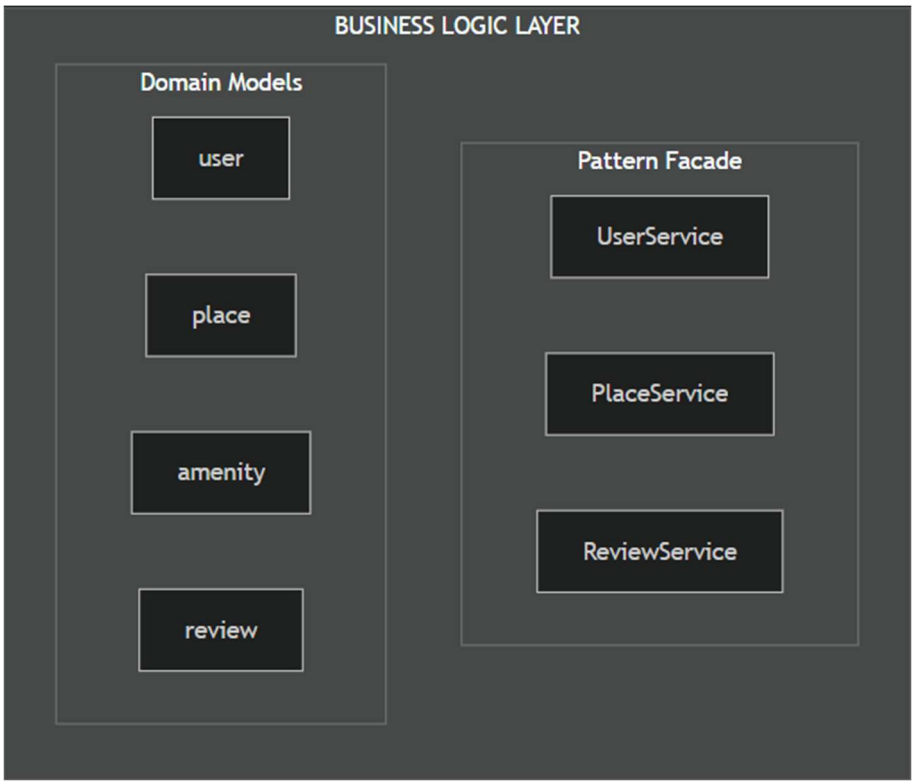


This layer corresponds to the client interaction interface (browser, mobile app, or tools like Postman).
It contains:

- Client Interface: the user entry point
- API with main routes:
    - /users → user creation
    - /places → place creation
    - /reviews → review submission

✅ The presentation layer never communicates directly with the business models: it always goes through a façade.

🧠 2. Business Logic Layer

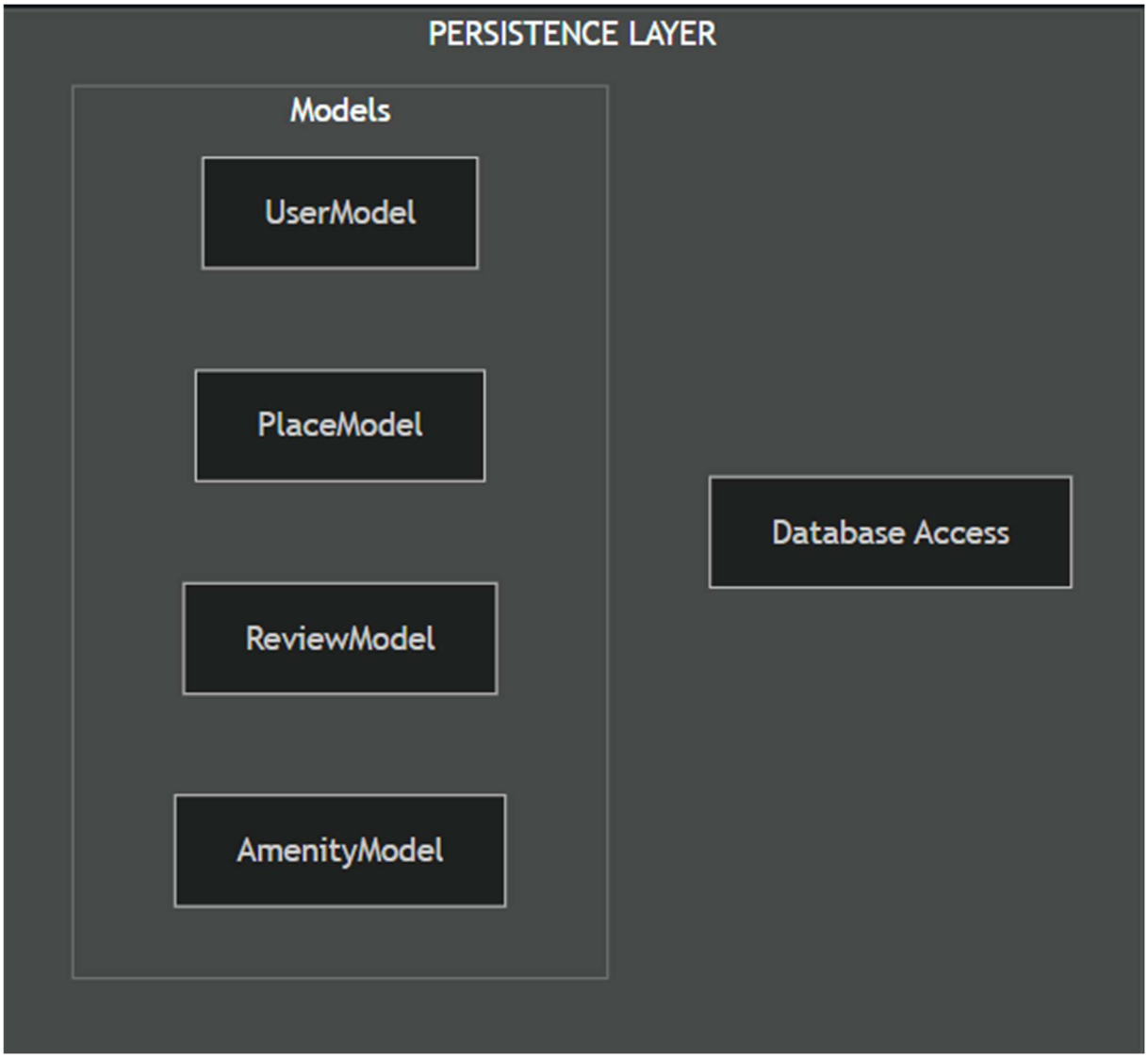


This layer contains the application’s business logic.
It is divided into two sub-parts:

- Domain Models:
    - User
    - Place
    - Review
    - Amenity
- Façade Pattern:
    - UserService
    - PlaceService
    - ReviewService

Each service represents a unique entry point for the business rules of a given entity. This ensures a clear separation between presentation and business logic, and centralizes validations and internal processing.

🗂️ 3. Persistence Layer



This layer is responsible for managing persisted data.
It contains:

- Database Access: generic functions (save(), update(), query())
- Models:

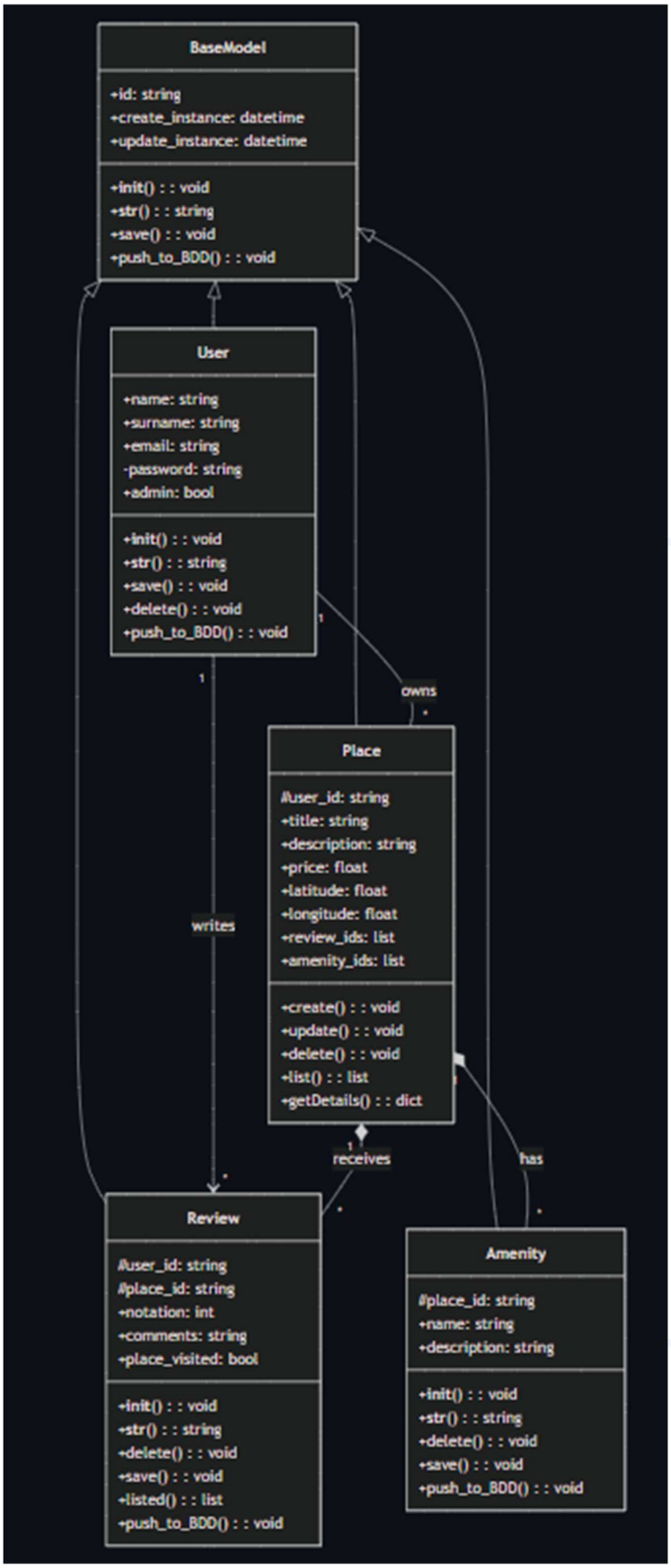
- UserModel
- PlaceModel
- ReviewModel
- AmenityModel

The models represent the entities in the database and are used exclusively by the business services.

🔁 Relations between layers

From	To	Action
Presentation Layer	Business Logic Layer	Uses the façade pattern
Business Logic Layer	Persistence Layer	Calls the persistent models

3. Business Logic Layer



This class diagram represents the main business entities of our HBnB application, along with their relationships, attributes, and methods.
It extends the package diagram validated in Task 0 and serves as the foundation for designing the business models.

🧱 General Structure

We have identified 4 main entities:

- User: the platform's users
- Place: the rental properties offered
- Review: the reviews left by users
- Amenity: the services associated with a property

All these entities inherit from a common superclass: BaseModel.

🧩 Class Details

🟦 BaseModel

Common parent class containing

shared attributes and methods.
Attributes:

- id: string – unique identifier
- create_instance: datetime – creation date
- update_instance: datetime – last update date

Methods :

- __init__() : void
- __str__() : string
- save() : void
- push_to_BDD() : void

🟩 User (inherits from BaseModel)
Represents a registered user.

Attributes:

- name: string
- surname: string
- email: string
- -password: string (protected field)
- admin: bool

Methods:

- init()
- str()
- save()
- delete()
- push_to_DB()

🟧 Place (inherits from BaseModel)

Represents a rental property.

Attributes:

- #user_id: string
- title: string
- description: string
- price: float
- latitude: float
- longitude: float
- review_ids: list<string>
- amenity_ids: list<string>

Methods:

- create()
- update()
- delete()
- list(): list<Place>
- getDetails(): dict

🟥 Review (inherits from BaseModel)

Represents a review written by a user about a property.

Attributes:

- #user_id: string

- #place_id: string
- rating: int
- comments: string
- place_visited: bool
  Methods:
- init()
- str()
- delete()
- save()
- listed(): list<Review>
- push_to_DB()

🟪 Amenity (inherits from BaseModel)

Represents a service offered in a property.

Attributes:

- #place_id: string
- name: string
- description: string

Methods:

- init()
- str()
- delete()
- save()
- push_to_DB()

🔗 Relationships between entities

Source Class	Relation	Target Class	Description
User	1 → *	Place	A user can offer multiple places
User	1 → *	Review	A user can write multiple reviews
Place	1 *→ *	Review	A place can receive multiple reviews
Place	1 *→ *	Amenity	A place can offer multiple services

🔸 → represents a strong association (aggregation): linked entities are interdependent.

🛠️ Design Choices

- Factorization via BaseModel to centralize identifier and metadata management.
- Encapsulation of links (user_id, place_id) using protected attributes (#) to promote consistency.
- Multiple relationships managed simply via lists of identifiers (review_ids, amenity_ids).
- Explicit CRUD methods: each class includes clear business functions aligned with persistence logic.

🧭 Conclusion

This class diagram formalizes the foundation of our HBnB application with a coherent and maintainable object-oriented approach.
It will serve as a direct reference for writing business models, implementing the API, and managing relationships between entities.

4. API Interaction Flow

This document accompanies the sequence diagrams produced for Task 2 of Part 1 of the HBnB Evolution project. It aims to explain the selected use cases, the architectural decisions made, and how our diagrams reflect the logical organization of the application.

Objectives:

- Justify the selection of the represented use cases.
- Demonstrate alignment with our 3-layer architecture (Presentation / Business Logic / Persistence).
- Provide context for reading the separate Mermaid (.md) files.

Architecture Overview
All diagrams were created in line with our software architecture, structured into three layers:

- **Presentation Layer:** Handles incoming API requests and exposes routes (e.g., /places, /reviews, etc.).
- **Business Logic Layer:** Contains business rules, validations, and calls to services like UserService, ReviewService, etc.
- **Persistence Layer:** Interacts directly with the database through models (e.g., UserModel, PlaceModel, etc.).

📚 How to Read a Sequence Diagram

A UML sequence diagram represents the flow of messages between system components in a given scenario.

Key elements to identify:

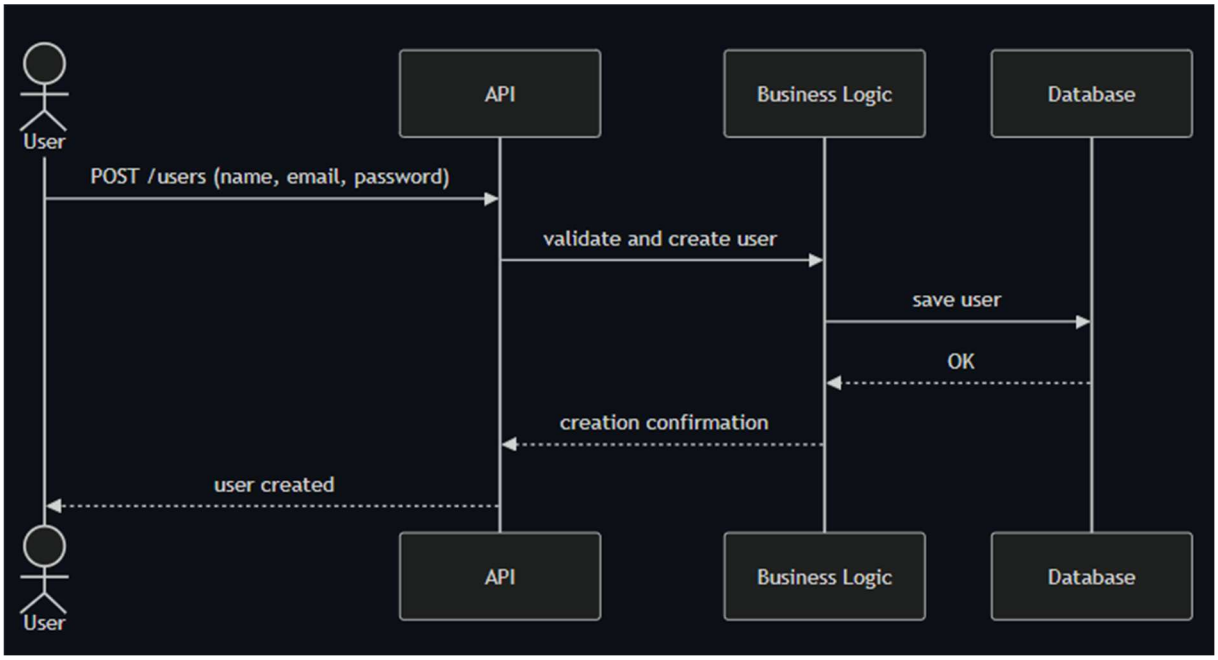
Element	Role
Participants	Entities interacting (user, service, model, etc.)
Arrows (→)	Function calls or messages sent from one actor to another
Arrows (-->)	Responses or return values
Vertical order	Time flows downward: the lower an action, the later it occurs
Inter-layer messages	Respect architecture flow: Presentation → Business Logic → Persistence

Cas d’usage représentés

Nous avons choisi de représenter quatre cas d’usage parmi les plus représentatifs de l’application :

1. Create User

This use case represents the creation of a new user via the /users endpoint.
It illustrates smooth data flow between layers, from the API to the final database storage.

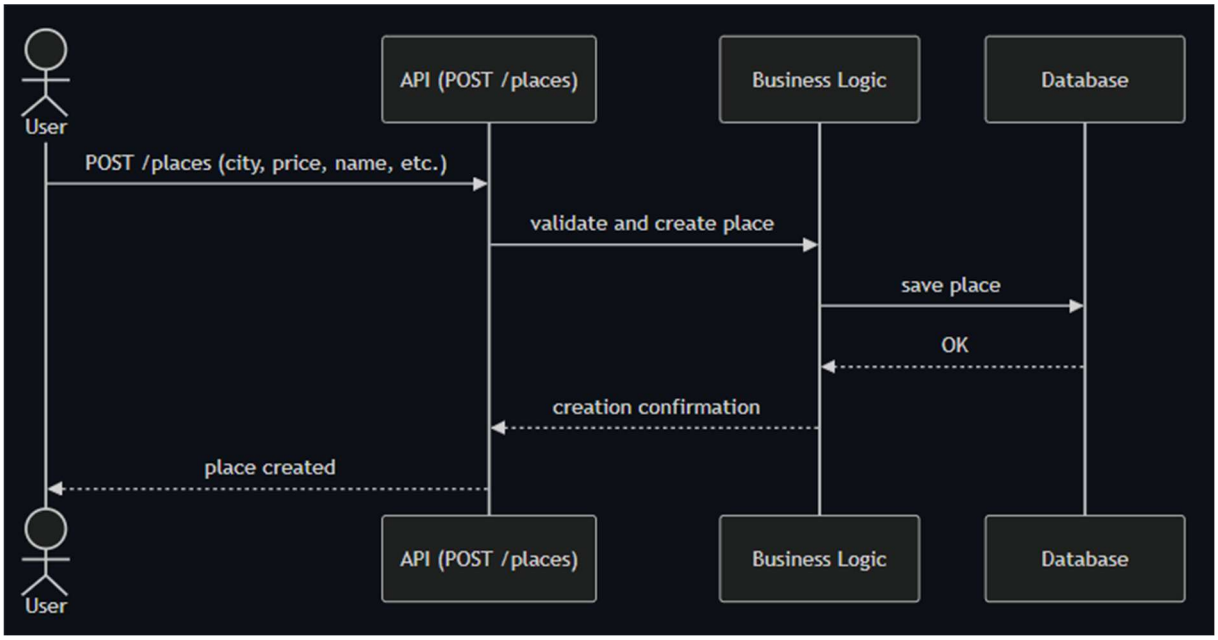


🧭 Step-by-Step Overview:

Step	Description
1. POST /users (name, email, password)	The user fills out a form with personal information.
2. validate & create user	The API forwards the data to the business layer.
3. save user	The Business Layer validates the data and saves the user.
4. API Response	Upon successful creation, a response is returned to the user.

2. Create a Place

This sequence shows how the /places creation route works.
It includes business logic validation (e.g., required fields) before saving to the database.



🧭 Step-by-Step Overview:

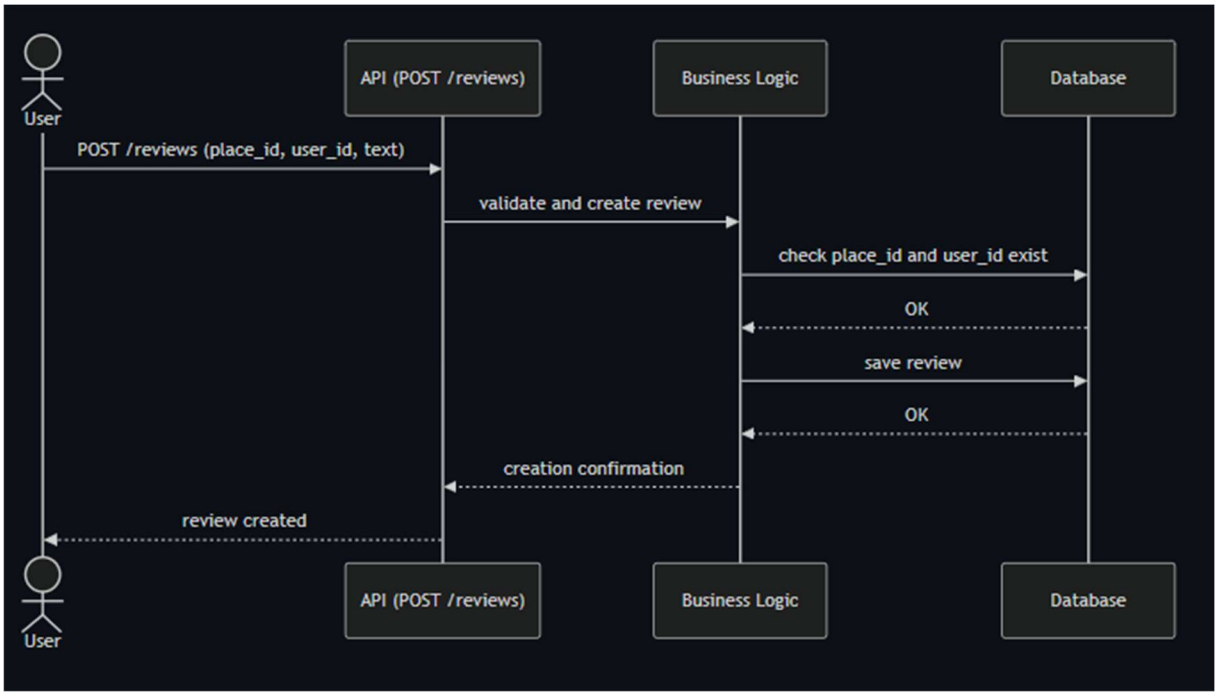
Step	Description
1. POST /places (city, price, name, etc.)	The user submits information for a new place.
2. validate and create place	The API sends the data to the business layer.
3. save place	The Business Layer checks the fields, then saves the place.
4. API Response	Upon success, the response is returned to the user.

3. Create a Review

This use case is slightly more complex: creating a review requires two validations:

- The place must exist.
- The author (user) must exist.

This interaction is a good example of coordination between services (PlaceService, UserService) and business validation before persisting an entity.

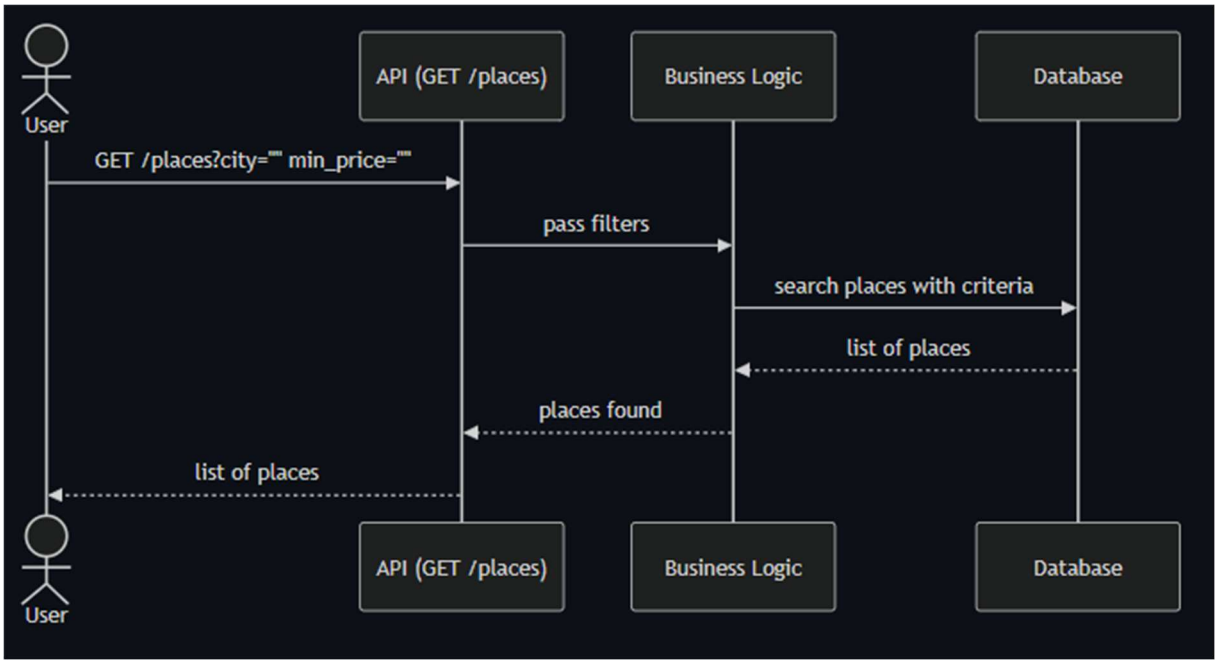


🧭 Step-by-Step Overview:

Step	Description
1. POST /reviews (place_id, user_id, text)	The user writes a review for a place.
2. validate and create review	The API forwards the data to the business layer.
3. check place_id and user_id exist	The system verifies that both the place and the author exist.
4. save review	If everything is valid, the review is saved.
5. API Response	A confirmation is returned to the user.

4. List Available Places

Here, we observe the behavior of a GET-type route with filters (e.g., city, min_price, etc.).
This diagram illustrates how filters are processed by the Business Logic Layer and passed to the Persistence Layer.



🧭 Step-by-Step Overview:

Step	Description
1. GET /places?city="" min_price=""	The user searches using filters like city or price.
2. pass filters	The API forwards the filters to the business layer.
3. search places with criteria	The system queries the database for matching results.
4. API Response	The list is returned to the user in JSON format.

5. Conclusion

This document provides a comprehensive summary of the software design carried out for Part 1 of the HBnB Evolution project. Based on a structured three-layer architecture, it demonstrates our ability to model a complex web application in a clear, maintainable manner, while adhering to best practices in software development.

The use of UML diagrams (architecture, class, sequence) has allowed us to:

- **Formalize** the relationships between business entities, highlighting the consistency of attributes, methods, and inheritance.
- **Clarify** the communication flows between the different layers of the application, strictly following the Presentation → Business Logic → Persistence structure.
- **Illustrate** concrete use cases, enabling us to better anticipate real-world interaction scenarios with the application.
- **Implement** appropriate design patterns, particularly the Façade Pattern, ensuring a clean interface between business logic and the presentation layer.

Beyond simple modeling, this work serves as a technical foundation for the upcoming implementation of the application by providing a global view of the components and their interactions. It will also facilitate collaboration among team members by establishing a shared language around the logical structures of the application.