■

Complete Technical Documentation

HBnB Project

A comprehensive documentation covering the architecture, design patterns, and implementation details of the HBnB application — an AirBnB-inspired platform.

Project	HBnB (Holberton AirBnB Clone)
Architecture	3-Tier Model (Presentation, Business Logic, Persistence)
Authors	Blee Leny, Jourdan Auxance
Institution	Holberton School
Date	12/02/2026

Table of Contents

1. Introduction

The **HBnB** project is an application inspired by AirBnB, allowing users to register, publish places, leave reviews, and manage amenities. The architecture is based on a **3-tier model** (Presentation, Business Logic, Persistence) ensuring a clear separation of concerns.

This document gathers all the UML diagrams produced during the design phase, along with explanatory notes and a consistency analysis.

2. High-Level Architecture Diagram

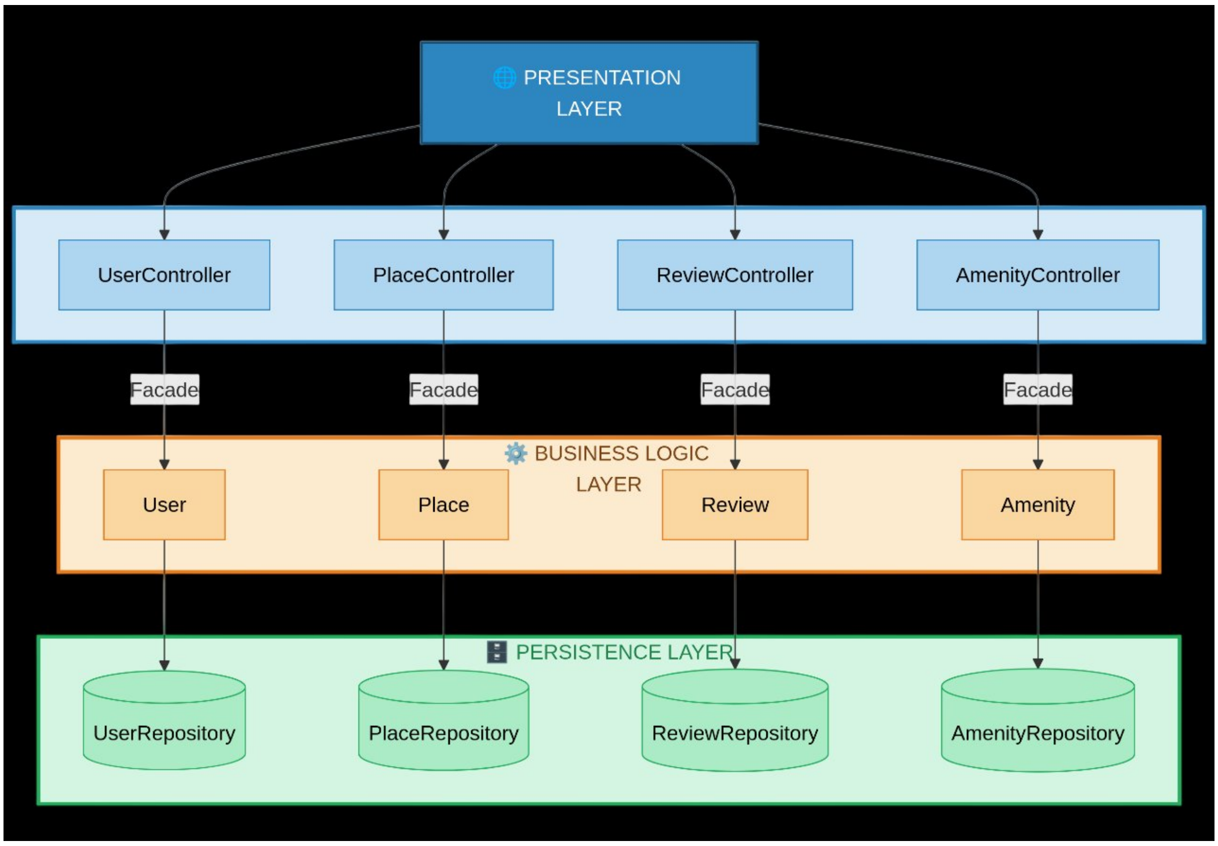

Figure 1: High-Level Package Diagram - 3-Tier Architecture

2.1 Presentation Layer

Component	Role
UserController	Exposes REST endpoints related to users (/api/users)
PlaceController	Exposes REST endpoints related to places (/api/places)
ReviewController	Exposes REST endpoints related to reviews (/api/reviews)
AmenityController	Exposes REST endpoints related to amenities (/api/amenities)

• This is the **entry point** for all HTTP requests coming from the client.

• Controllers contain **no business logic**; they immediately delegate processing to the layer below.

• Each controller communicates with the Business Logic Layer through the **Facade pattern**, ensuring loose coupling between layers.

2.2 Business Logic Layer

Component	Role
User	Handles registration logic, profile updates, and user deletion
Place	Handles creation, modification, deletion, and listing of places, as well as adding/removing amenitie
Review	Handles creation, modification, deletion, and listing of reviews by place
Amenity	Handles creation, modification, deletion, and listing of amenities

• This layer encapsulates all **business rules** (validations, constraints, data transformations).

• The business entities (User, Place, Review, Amenity) are the core domain models.

• Communication with the Persistence Layer is done through **repositories**, allowing the storage method to be changed without impacting the business logic.

2.3 Persistence Layer

Component	Role
UserRepository	CRUD operations on user data in the database
PlaceRepository	CRUD operations on place data in the database
ReviewRepository	CRUD operations on review data in the database
AmenityRepository	CRUD operations on amenity data in the database

• Responsible for **saving and retrieving** data (database, files, memory).

• Implements the **Repository Pattern**, providing an abstract interface to the Business Logic Layer.

• Allows easy future migration (e.g., switching from in-memory storage to an SQL database).

2.4 Communication Flow Between Layers

The architecture follows a strict vertical flow where each layer communicates only with its adjacent layers:

• **Client → Presentation Layer:** HTTP requests are received by REST controllers

• **Presentation → Business Logic (via Facade):** Controllers delegate to business entities using the Facade pattern

• **Business Logic → Persistence (via Repository):** Entities use repositories to persist or retrieve data

• **Response flow:** Data flows back through the same layers in reverse order

This layered approach ensures **separation of concerns**, **maintainability**, and **testability**. Each layer can be modified or replaced independently without affecting the others.

3. Class Diagram

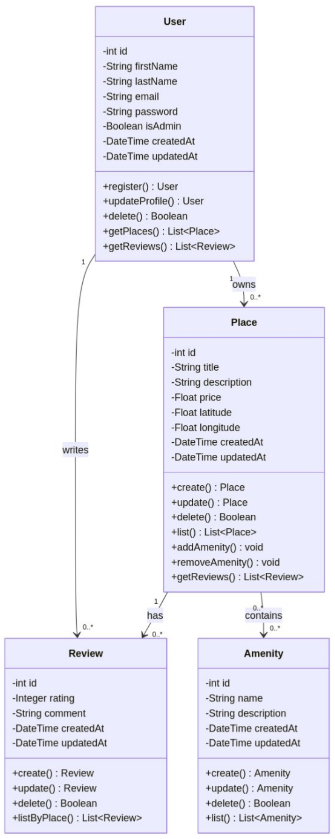

Figure 2: Class Diagram showing entities and their relationships

3.1 User Class

- **Core entity** of the system representing a registered user.
- The *isAdmin* attribute distinguishes administrators from standard users.
- *email* serves as the unique identifier for authentication.
- The *register()* method creates a new user with password hashing.
- The *getPlaces()* and *getReviews()* methods allow navigation to associated entities.

3.2 Place Class

- Represents a **place/accommodation** available for rent.
- The *latitude/longitude* coordinates enable geolocation features.
- *addAmenity()* and *removeAmenity()* manage the many-to-many relationship with amenities.
- *getReviews()* returns all reviews associated with this place.

3.3 Review Class

- Represents a **review** left by a user on a place.
- *rating* is an integer score (e.g., from 1 to 5).
- *listByPlace()* retrieves all reviews for a specific place.

3.4 Amenity Class

- Represents an **amenity/service** available at a place (WiFi, pool, parking, etc.).
- Can be shared across multiple places (many-to-many relationship).

3.5 Relationships Between Classes

Relationship	Type	Cardinality	Description
User → Place	owns	1 — 0..*	A user owns zero or more places
User → Review	writes	1 — 0..*	A user writes zero or more reviews
Place → Review	has	1 — 0..*	A place receives zero or more reviews
Place → Amenity	contains	0..* — 0..*	A place contains zero or more amenities; an amenity can belong to multipl

3.6 Common Attributes

All entities share the following attributes, suggesting the potential existence of an **abstract base class**:

- *id*: unique identifier
- *createdAt*: creation date
- *updatedAt*: last modification date

4. Sequence Diagrams

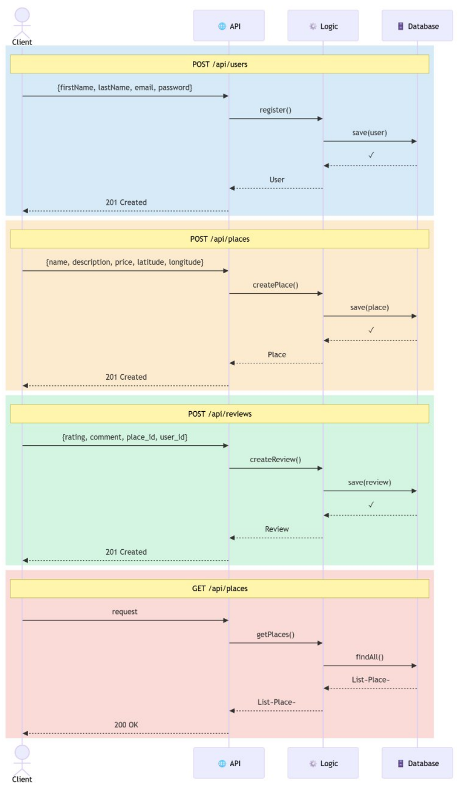

Figure 3: Sequence Diagrams for main API operations

The sequence diagrams illustrate the interactions between the **four participants** of the system for the main API calls:

Participant	Layer	Role
Client	External	The user or application sending HTTP requests
API	Presentation Layer	The REST controller that receives and responds to requests
Logic	Business Logic Layer	The service/model that applies business logic
Database	Persistence Layer	The repository that persists data

4.1 POST /api/users (User Registration)

1. The **Client** sends a POST request with registration data (firstName, lastName, email, password).
2. The **API** (UserController) receives the request and calls the *register()* method of the Logic layer.
3. The **Logic** layer (User) validates the data (unique email, password format, etc.), creates the User object, and requests persistence via *save(User)* to the Database.
4. The **Database** (UserRepository) persists the user and confirms the operation (✓).

5. The Logic layer returns the created **User** object to the API.

6. The API sends the Client a **201 Created** response with the user data.

4.2 POST /api/places (Place Creation)

1. The **Client** sends a POST request with place information (name, description, price, latitude, longitude).

2. The **API** (PlaceController) forwards to the Logic layer via *createPlace()*.

3. The **Logic** layer (Place) validates the data (positive price, valid coordinates, etc.), associates the place with the authenticated user, then saves via *save(Place)*.

4. The **Database** (PlaceRepository) persists the place and confirms (✓).

5. The **Place** object is returned through the layers.

6. The Client receives a **201 Created** response.

4.3 POST /api/reviews (Review Creation)

1. The **Client** sends a POST request with review data (rating, comment, place_id, user_id).

2. The **API** (ReviewController) forwards via *createReview()* to the Logic layer.

3. The **Logic** layer (Review) verifies that the place and user exist, the user is not reviewing their own place (optional business rule), and the rating is within the allowed range.

4. The **Database** (ReviewRepository) persists the review via *saveReview()* and confirms (✓).

5. The **Review** object is returned.

6. The Client receives a **201 Created** response.

4.4 GET /api/places (Retrieve List of Places)

1. The **Client** sends a simple GET request (no body).

2. The **API** (PlaceController) calls *getPlaces()* in the Logic layer.

3. The **Logic** layer (Place) requests retrieval of all places via *findAll()* from the Database.

4. The **Database** (PlaceRepository) returns the **List<Place>** containing all places.

5. The list is passed through the layers.

6. The Client receives a **200 OK** response with the list of places.

5. Conclusion

This technical documentation presents the complete architecture of the **HBnB** project through three complementary types of UML diagrams:

Diagram	What It Shows	Perspective
Layered Architecture	How the system is organized	Macro structure (deployment)
Class Diagram	How the data is modeled	Micro structure (domain)
Sequence Diagrams	How the components interact	Behavior (runtime)

The three diagrams are **globally consistent** with each other:

- The **4 entities** (User, Place, Review, Amenity) appear in all three views.
- The **3-tier architecture** is respected in the sequence flows (Client → API → Logic → Database).
- The **methods** defined in the class diagram correspond to the calls visible in the sequence diagrams.
- The **Facade pattern** mentioned in the architecture is effectively implemented in the inter-layer interactions.

This documentation base will serve as a **reference** for the implementation of the different layers of the application.

6. Authors

Name	GitHub Profile
Blee Leny	github.com/LenyBl
Jourdan Auxance	github.com/JAuxance

Document written as part of the HBnB project – Holberton School