

HBnB Project – Technical Documentation

System Architecture and API Design Blueprint

SELENA GÓMEZ RUEDA

ALEXANDER ZULETA GARRO

SOFTWARE DEVELOPMENT

HOLBERTON SCHOOL

NOVEMBER - 2025

INTRODUCTION

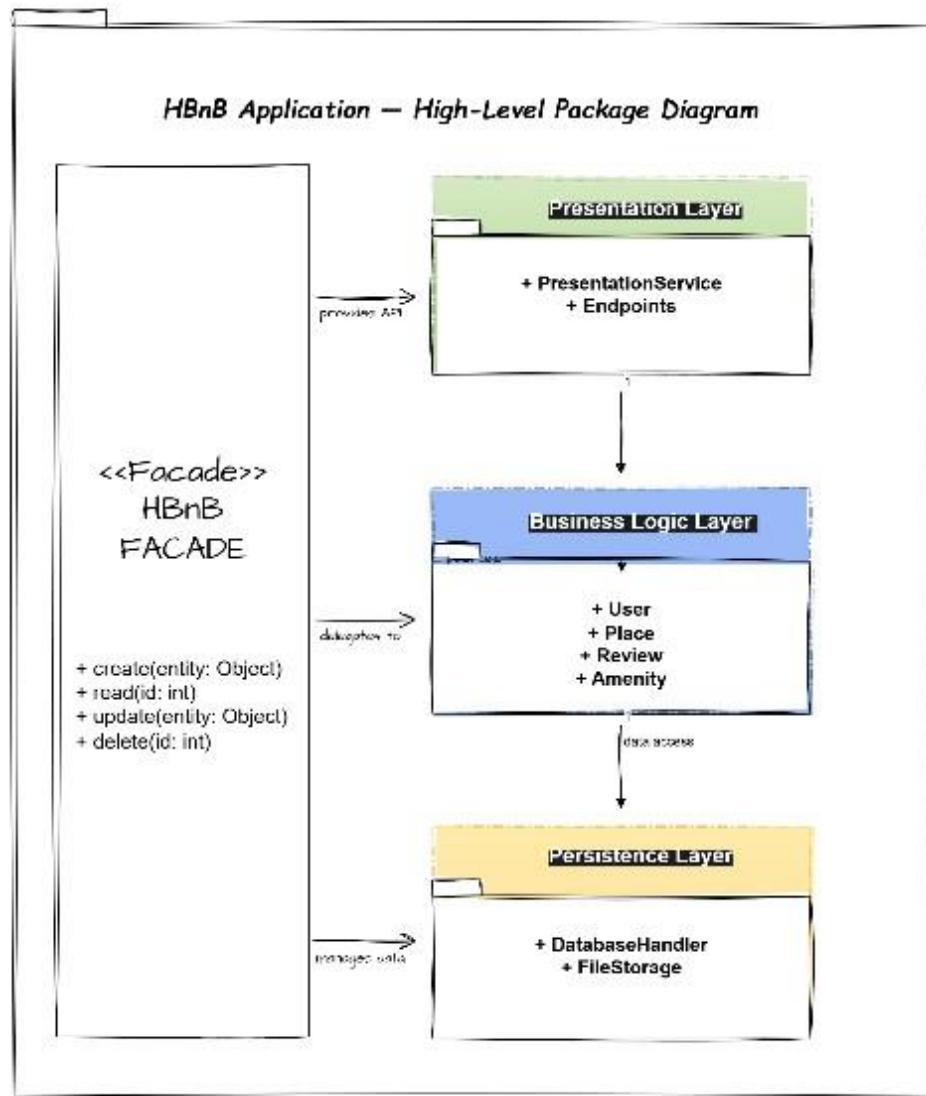
The **HBnB project** is a full-stack web application inspired by Airbnb, designed to manage accommodations, users, and bookings through a RESTful API and a layered architecture.

This technical document compiles all the **architectural and design artifacts** produced during the analysis and design phases. Its goal is to serve as a **blueprint** for developers and maintainers, guiding the implementation process and ensuring a consistent, scalable design.

This document includes:

- High-Level Architecture overview.
- Business Logic Layer class design.
- API Interaction and sequence flow diagrams.

HIGH-LEVEL PACKAGE DIAGRAM



Overview — HBnB Application High-Level Package Diagram

The diagram illustrates the **overall architecture** of the HBnB system, divided into three main layers and a central façade component. Each layer has a specific role, ensuring a clear separation of concerns and modularity.

HBnB Facade

The *Facade* acts as the **entry point** to the entire system. It provides unified methods — `create()`, `read()`, `update()`, and `delete()` — allowing other components or external systems to interact with HBnB in a simplified way. Its main goal is to **hide internal complexity** and offer a clean, consistent interface.

Presentation Layer

This layer represents the **API and user-facing services**. It includes the PresentationService and multiple endpoints that expose system functionalities.

Requests from clients pass through this layer before reaching the business logic via the facade.

Business Logic Layer

This layer defines the **core logic and behavior** of the application. It contains the main entities such as User, Place, Review, and Amenity, and is responsible for applying business rules, validations, and processing data before persistence or presentation.

Persistence Layer

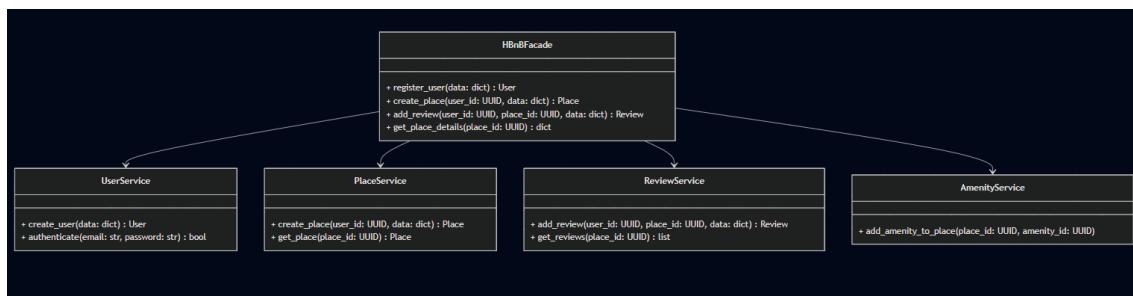
The persistence layer manages **data storage and retrieval**. It includes components like DatabaseHandler and FileStorage that ensure all information is correctly saved and accessible when needed. This layer isolates the rest of the system from low-level storage details.

Overall Flow

1. The **client** interacts with the Presentation Layer.
2. The **Facade** receives the request and delegates it appropriately.
3. The **Business Logic Layer** processes the data and applies the rules.
4. The **Persistence Layer** stores or retrieves data as needed.

This architecture promotes **scalability, maintainability, and clean separation** between presentation, logic, and data management.

DETAILED CLASS DIAGRAM FOR BUSINESS LOGIC LAYER



OVERVIEW — HBnB FACADE AND SERVICE LAYER DIAGRAM

This diagram illustrates the **interaction between the HBnB Facade and the core service components** that manage different parts of the system's functionality.

It shows how the *Facade Pattern* simplifies access to multiple independent services by providing a unified interface.

HBnBFacade

The HBnBFacade class acts as a **central coordinator** that exposes high-level methods to interact with the system without revealing internal details. It provides operations for managing users, places, reviews, and amenities:

- `register_user(data: dict): User` — Creates a new user in the system.
- `create_place(user_id: UUID, data: dict): Place` — Registers a new place for a specific user.
- `add_review(user_id: UUID, place_id: UUID, data: dict): Review` — Adds a review from a user to a place.
- `get_place_details(place_id: UUID): dict` — Retrieves detailed information about a place, including reviews and amenities.

The facade delegates these operations to the appropriate **service classes**, ensuring that clients interact with a single entry point.

UserService

Handles **user-related logic** such as registration and authentication.

- `create_user(data: dict): User` — Creates a user object and stores it.
- `authenticate(email: str, password: str): bool` — Verifies user credentials.

PlaceService

Manages **places and their data**.

- `create_place(user_id: UUID, data: dict): Place` — Registers a place associated with a user.
- `get_place(place_id: UUID): Place` — Retrieves information about a specific place.

ReviewService

Handles all **review-related actions**.

- `add_review(user_id: UUID, place_id: UUID, data: dict): Review` — Adds a new review to a place.
- `get_reviews(place_id: UUID): list` — Returns all reviews for a specific place.

AmenityService

Manages **amenities** associated with places.

- `add_amenity_to_place(place_id: UUID, amenity_id: UUID)` — Links an amenity to a given place.

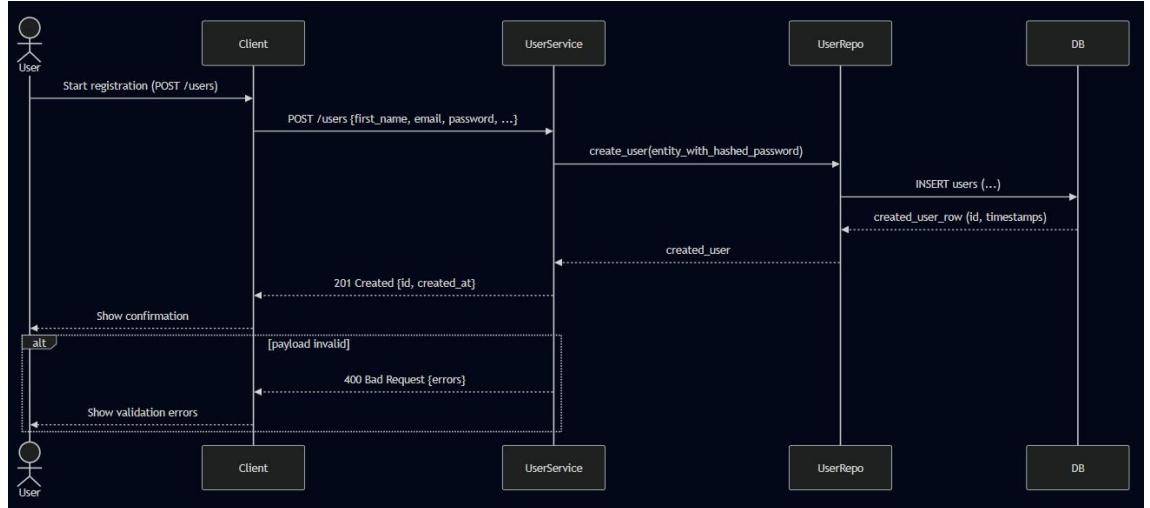
Overall Flow

1. The **client** calls a method in the HBnBFacade.
2. The facade **delegates** the request to the relevant service (User, Place, Review, or Amenity).
3. Each service handles its own logic and returns data to the facade.
4. The **Facade** then returns the processed result to the client.

This structure promotes **loose coupling, reusability, and scalability**, allowing each service to evolve independently while keeping the client interface simple.

SEQUENCE DIAGRAMS FOR API CALLS

1. **User Registration:** A user signs up for a new account.



Overview — User Registration Sequence Diagram

This diagram illustrates the **sequence of interactions** involved in the *User Registration* process within the HBnB system. It shows how different components — the client, service, repository, and database — collaborate to create a new user account.

Flow Explanation

1. **User** → **Client**
The user initiates registration through the client application (e.g., a web or mobile app).
The client prepares a POST /users request containing the registration data:
{ first_name, email, password, ... }.
2. **Client** → **UserService**
The client sends the registration request to the UserService, which is responsible for handling business logic related to users.
The service validates the input and hashes the password before creating the entity.
3. **UserService** → **UserRepo**
The UserService calls UserRepo.create_user(entity_with_hashed_password) to store the new user in the database.

4. **UserRepo** → **DB**
The repository executes an INSERT INTO users (...) operation in the database to persist the user record.
The database responds with the created record, including the generated id and timestamps.
5. **UserRepo** → **UserService** → **Client**
The repository returns the created user to the service, which then sends a 201 Created response to the client containing { id, created_at }.
6. **Client** → **User**
The client displays a **confirmation message** to the user indicating successful registration.

Alternative Flow — Error Handling

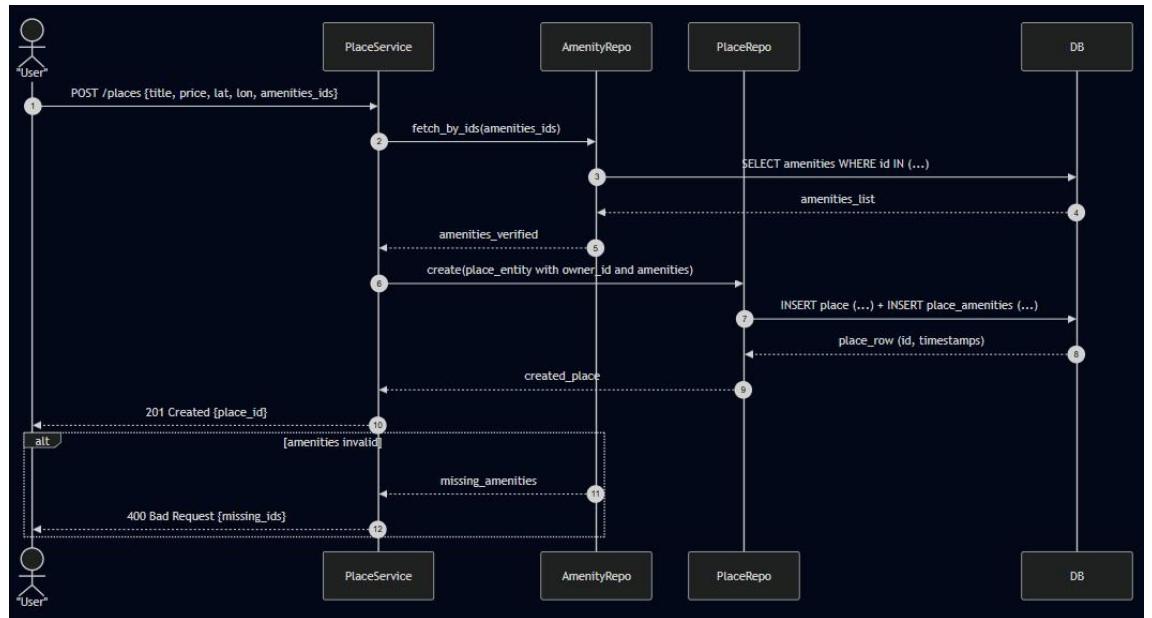
If the request payload is invalid (e.g., missing fields or invalid data):

- The UserService detects validation errors.
- The system responds with a 400 Bad Request and an { errors } object.
- The client displays **validation error messages** to guide the user.

Key Takeaways

- The diagram follows a **clear separation of concerns**:
 - UserService handles business rules.
 - UserRepo manages database operations.
 - DB stores persistent data.
- The alt block visually represents **alternative outcomes** — success or validation failure.
- This design promotes **modularity**, **data integrity**, and **error transparency** in the registration process.

2. Place Creation: A user creates a new place listing.



This sequence diagram illustrates the **creation workflow of a new Place entity** within the system.

The interaction starts when the **User** submits a `POST /places` request containing place data such as title, price, coordinates, and a list of amenity IDs.

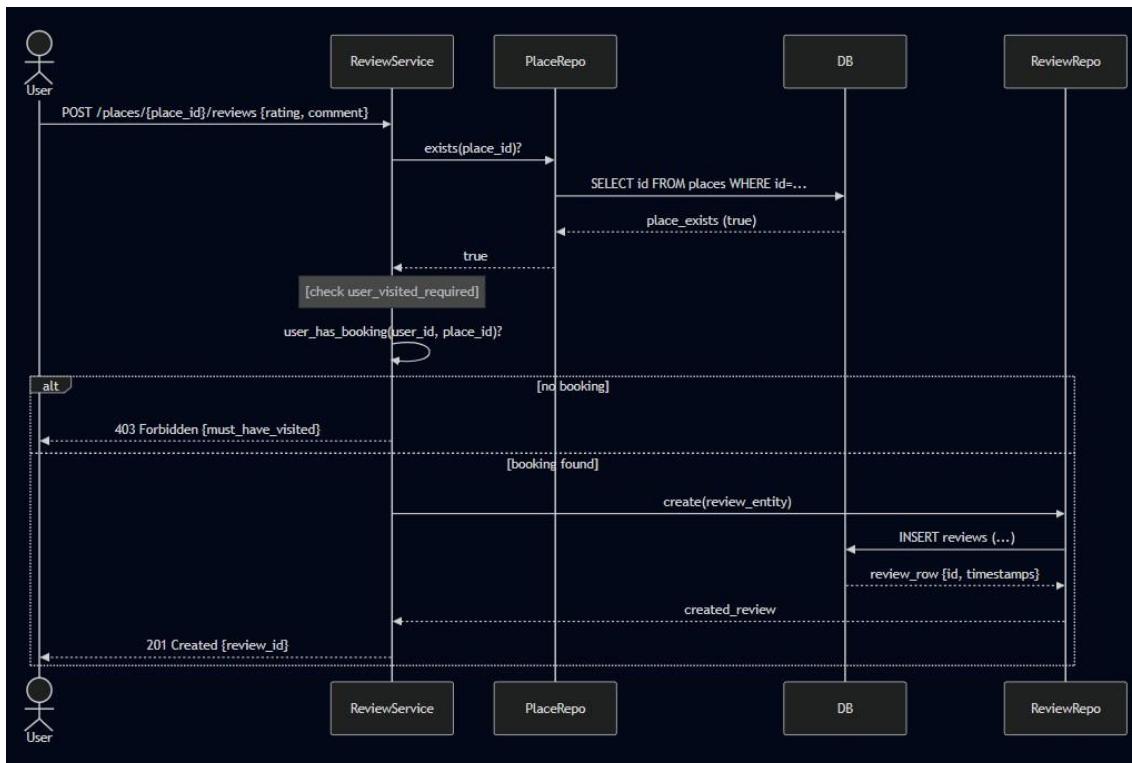
The **PlaceService** acts as the orchestrator:

1. It first validates the provided amenities by invoking **AmenityRepo.fetch_by_ids()**, which queries the database to confirm their existence.
2. If all amenities are verified, the service constructs a new **Place entity** that includes the owner ID and associated amenities.
3. The **PlaceRepo** then persists the entity by executing `INSERT` operations for both the place and place_amenities tables.
4. Upon success, the service returns a **201 Created** response containing the generated place_id.

An **alternative flow** handles invalid amenity references. If one or more amenity IDs are missing, the service returns a **400 Bad Request** with a list of invalid or missing IDs, avoiding inconsistent data creation.

This diagram highlights the internal collaboration between service and repository layers, ensuring data integrity and controlled persistence logic.

3. Review Submission: A user submits a review for a place.



This sequence diagram represents the **review submission process** for a place.

The **User** sends a POST /places/{place_id}/reviews request with a rating and comment.

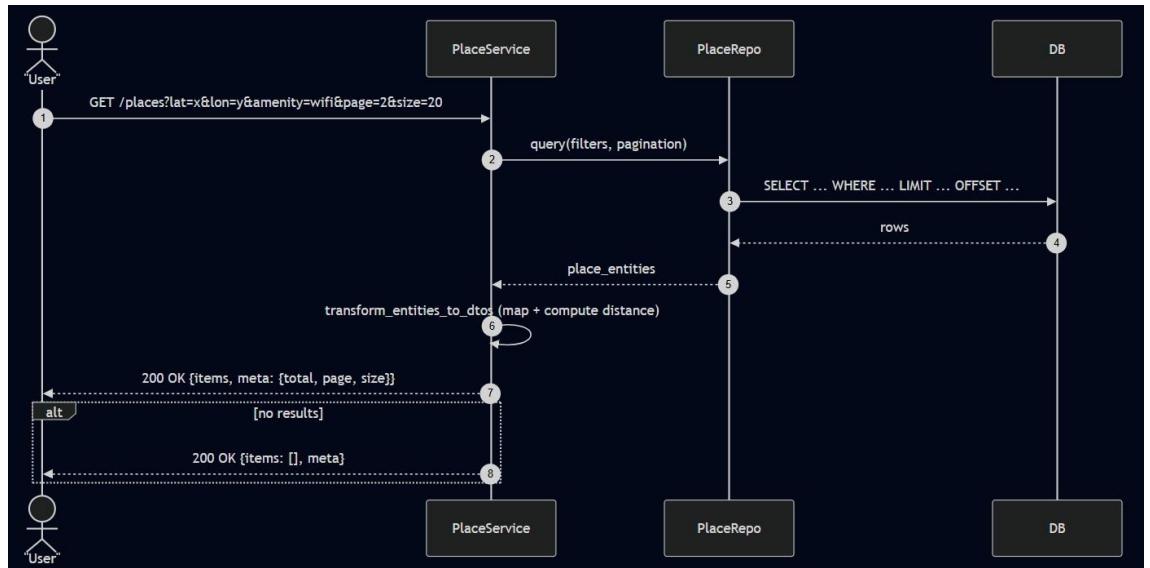
The **ReviewService** first verifies that the target place exists via the **PlaceRepo**. Then, it checks whether the user has a valid booking for that place— enforcing the “must have visited” rule.

If no booking is found, the service returns a **403 Forbidden** response.

Otherwise, it creates a **Review entity** and persists it through the **ReviewRepo**, resulting in a **201 Created** response containing the new review_id.

This flow ensures **data validity** and prevents users from reviewing places they haven’t visited.

4. **Fetching a List of Places:** A user requests a list of places based on certain criteria.



This diagram illustrates the flow for fetching a **paginated list of places** from an API, filtered by location and amenity type (e.g., wifi). It involves three main layers: **service**, **repository**, and **database**, along with the user interacting with the API.

Flow Explanation

1. The **User** sends a request:
GET /places?lat=x&lon=y&amenity=wifi&page=2&size=20.
2. **PlaceService** receives the request and builds the query with filters and pagination.
3. **PlaceRepo** executes a SQL query (SELECT ... WHERE ... LIMIT ... OFFSET ...) against the **DB**.
4. The **DB** returns the matching rows.
5. **PlaceRepo** sends the results back as place_entities.
6. **PlaceService** transforms the entities into DTOs and computes the distance.
7. If there are results, it responds with 200 OK and the payload (items, meta).
8. If no results are found, it still returns 200 OK but with an empty items array.