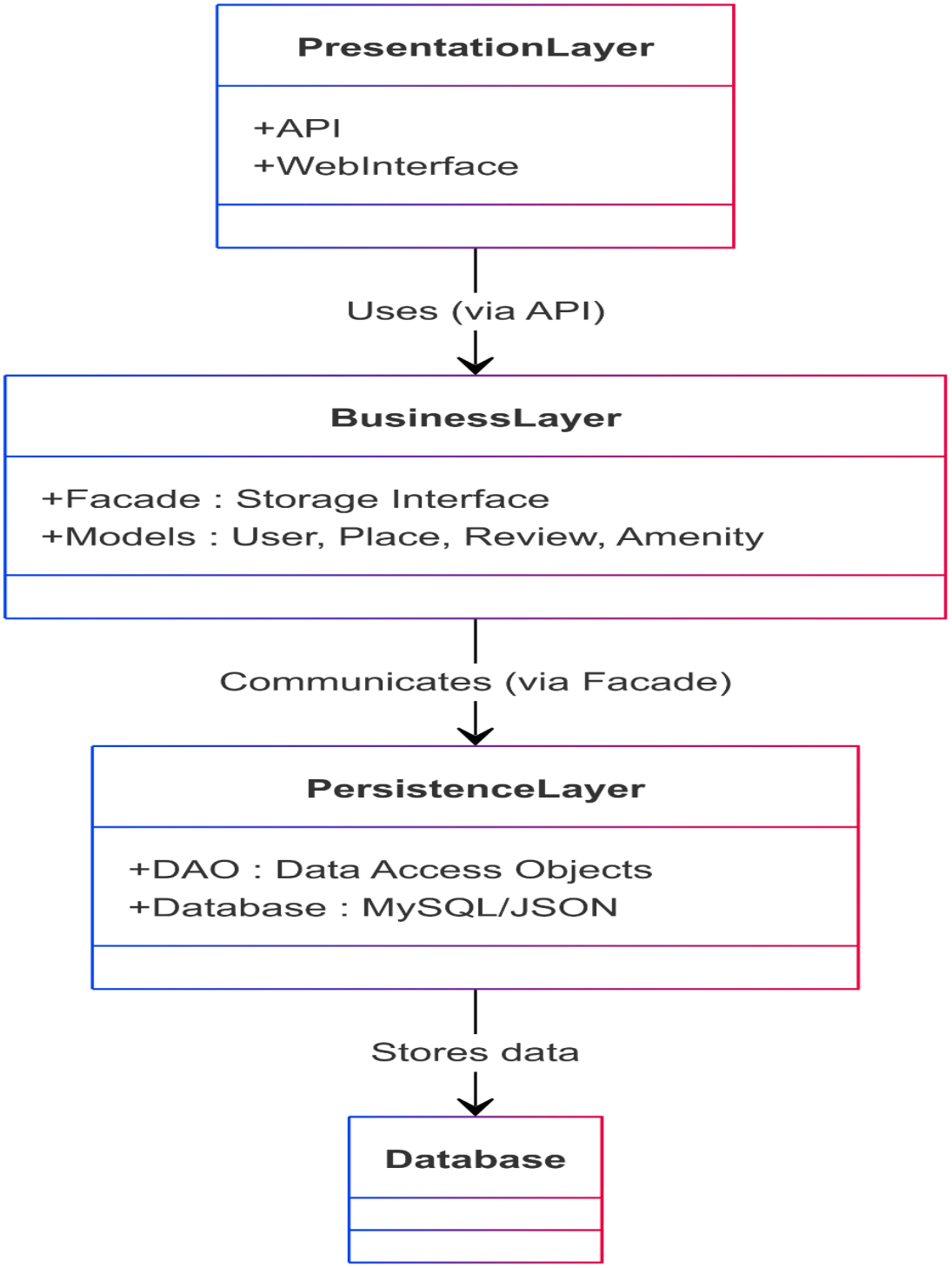
HBnB Project: Technical Documentation

Introduction

This document serves as a technical blueprint for the HBnB project. It describes the general architecture, business logic and interaction flows between components.



0. High-Level Architecture (Package Diagram):

Purpose:

To represent the layered architecture and how the application is organized.

Key Components:

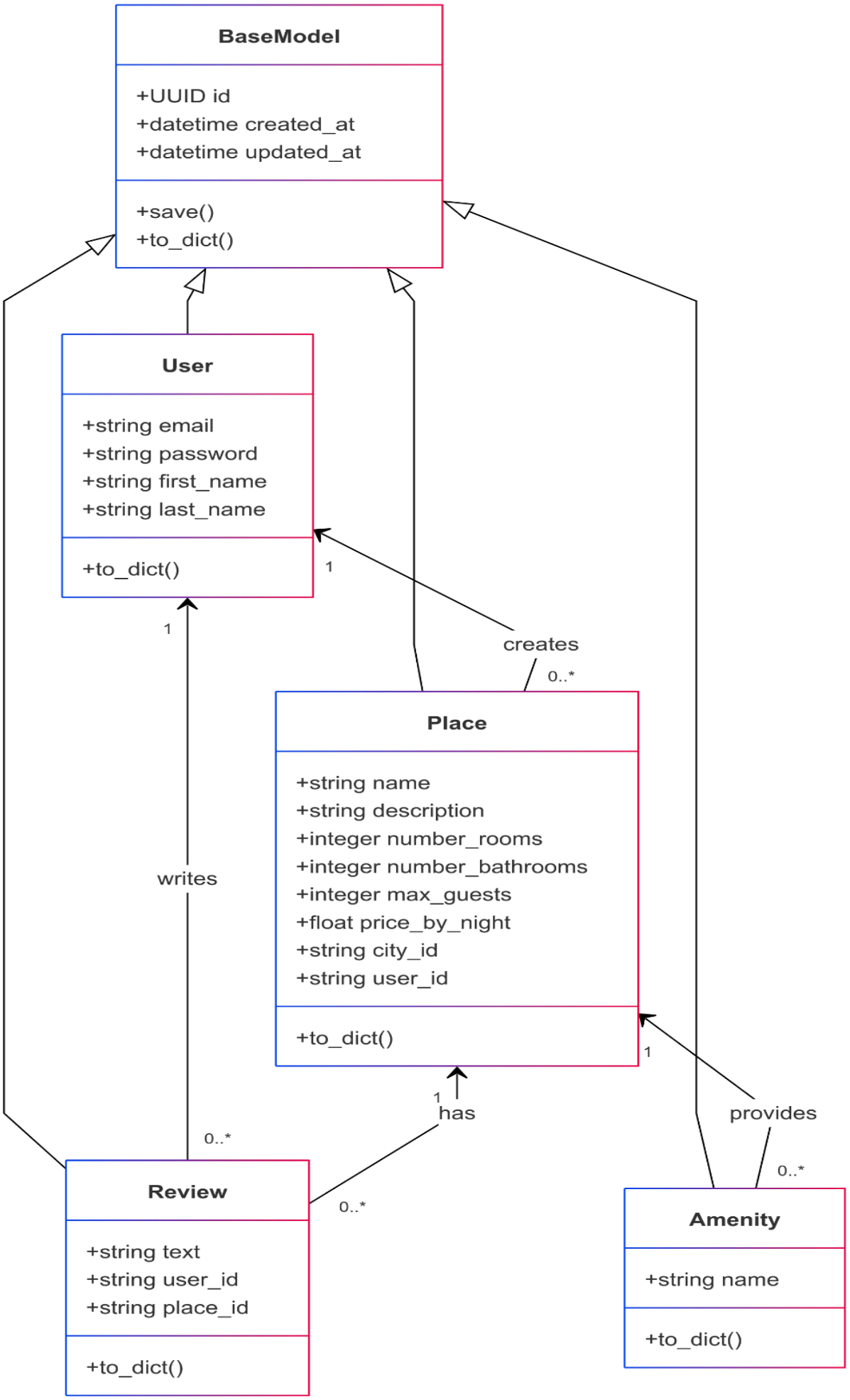
- **Presentation Layer:** Handles user interaction through the API and web interface.
- **Business Logic Layer:** Contains the models and business logic. It uses the **Facade Pattern** to interact with the persistence layer.
- **Persistence Layer:** Manages data storage using a database engine (DBStorage for MySQL or FileStorage for JSON).

Design Decisions:

- We use the **layered architecture pattern** to separate concerns.
- We implement the **Facade Pattern** to decouple the business logic from the storage system.

Relationship with Overall Architecture:

This structure enhances scalability and maintainability, allowing database or API changes without impacting other layers.


BaseModel
+UUID id
+datetime created_at
+datetime updated_at
+save()
+to_dict()
User
+string email
+string password
+string first_name
+string last_name
+to_dict()
Place
+string name
+string description
+integer number_rooms
+integer number_bathrooms
+integer max_guests
+float price_by_night
+string city_id
+string user_id
+to_dict()
Review
+string text
+string user_id
+string place_id
+to_dict()
Amenity
+string name
+to_dict()
1
creates
0..*
1
writes
0..*
1
has
0..*
1
provides
0..*

1. Business Logic Layer (Class Diagram):

Purpose:

To show the main entities (User, Place, Review, Amenity) and their relationships.

Key Components:

- BaseModel**:** A base class inherited by all entities, containing common attributes (id, created_at, updated_at).
- User**:** Represents a registered user, related to Place and Review.
- Place**:** Represents a rental property, associated with a User and multiple Amenity.
- Review**:** A review created by a User about a Place.
- Amenity**:** Services offered by a Place.

Relationships:

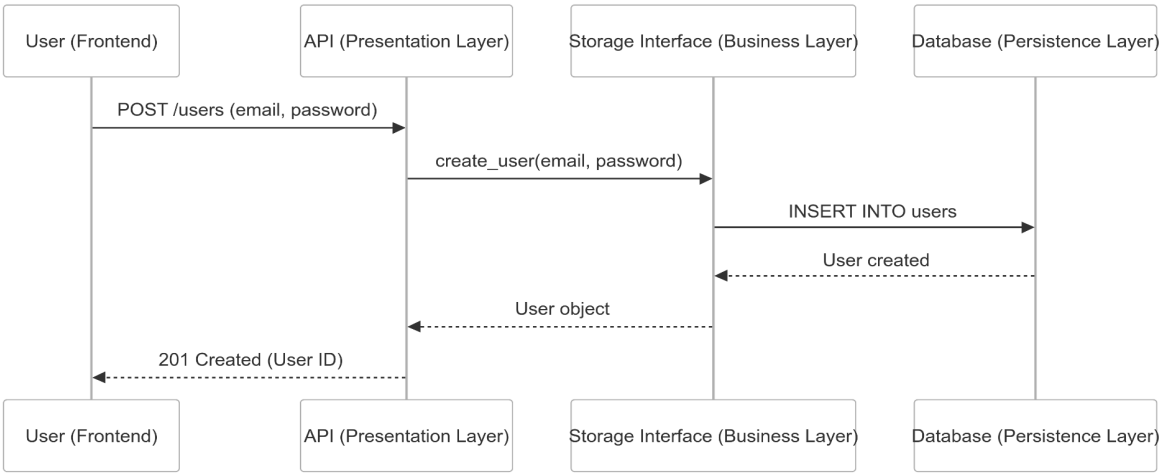
- **Inheritance (**<|--**):** All classes inherit from BaseModel.
- **Associations (**--**):**
    - A User can create multiple Place instances.
    - A User can submit multiple Review instances.
    - A Place can have multiple Review instances and multiple Amenity instances.

Design Decisions:

- **Inheritance** is used to avoid duplicate common attributes.
- **Relationships** (1-to-N and N-to-M) are clearly defined to reflect business logic.

Relationship with Overall Architecture:

This layer is the core of business logic, processing operations and validations before interacting with the database.



2. API Interaction Flow (Sequence Diagrams):

Purpose:

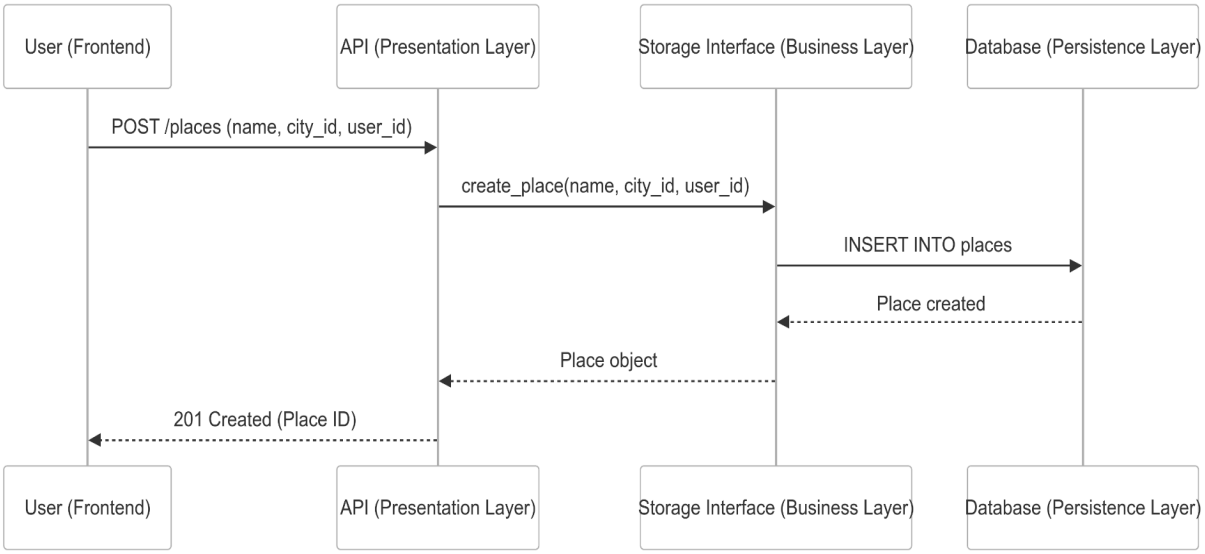
To visualize how API calls are managed and how the layers interact.

Case 1: User Registration

- The user sends a POST /users request.
- The API passes the request to the Facade (create_user).
- The Facade inserts the new user into the database.
- The response returns with the new User ID.

Design Rationale:

- Centralizing user creation through the Facade simplifies database interactions.
- Returning a 201 Created response follows REST standards.

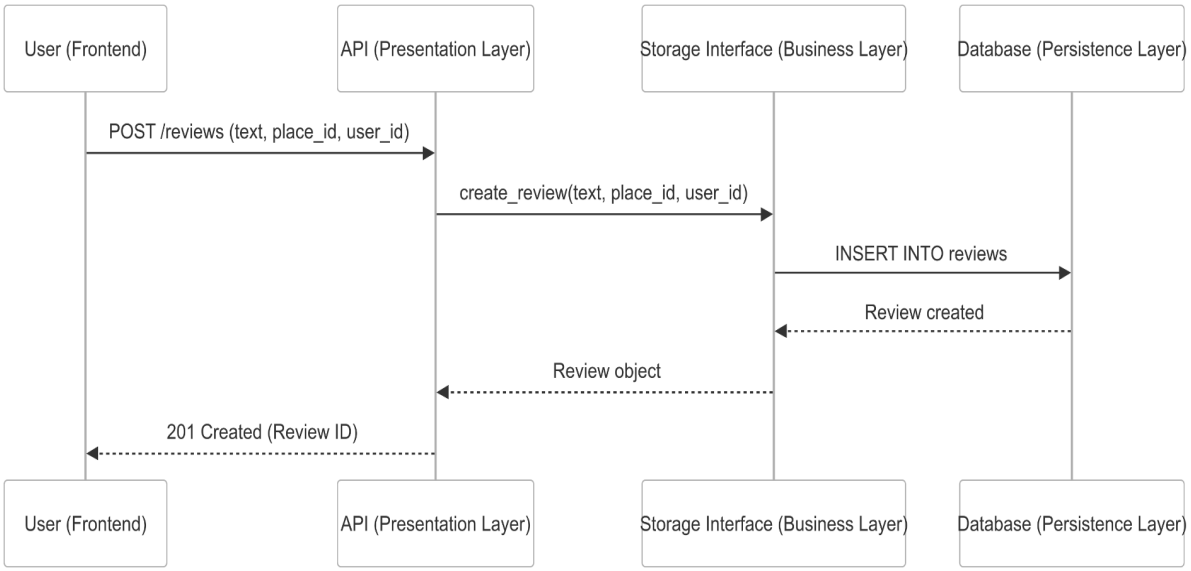


Case 2: Place Creation

- The user sends a POST /places request.
- The API delegates the creation to the Facade (create_place).
- The Facade inserts the new place into the database.
- The response returns the generated Place ID.

Design Rationale:

- A single access point (Facade) is used for place creation.
- Returning a 201 Created status aligns with REST principles.

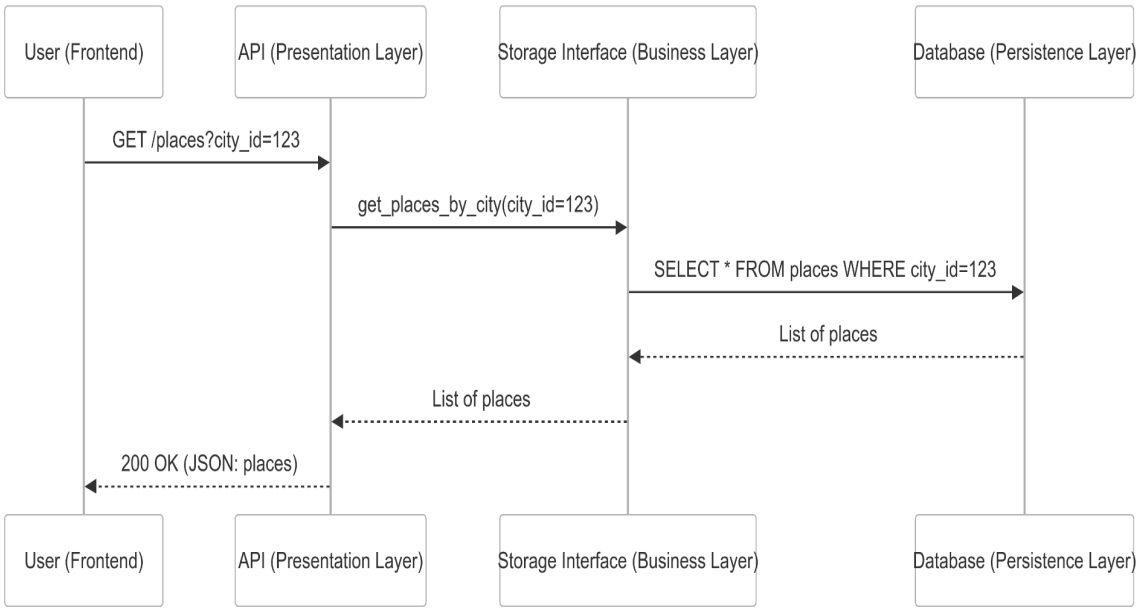


Case 3: Review Submission

- The user sends a POST /reviews request.
- The API forwards the request to the Facade (create_review).
- The Facade stores the review in the database.
- The response returns the generated Review ID.

Design Rationale:

- Centralized review creation through the Facade.
- Adherence to REST standards with appropriate status codes.



Case 4: Fetching a List of Places

- The user sends a GET /places request.
- The API queries the Facade (get_places_by_city).
- The Facade retrieves the list from the database.
- The response returns a list of places in JSON format.

Design Rationale:

- Filtering options (e.g., ?city_id=) are implemented for flexible queries.
- Returning a 200 OK status follows REST standards.

Conclusion:

- **Layered Architecture:** Enhances modularity and maintainability.
- **Facade Pattern:** Simplifies communication between business logic and storage.
- **Class Diagram:** Represents the structure and relationships of data models.
- **Sequence Diagrams:** Show the flow between client, API, business logic, and storage layers.