HBnB Technical Documentation (Part 1)

📌 **Purpose:** This technical document consolidates the main architecture diagrams and explanatory notes for **HBnB (Part 1)**.
Goal: Use it as an implementation blueprint, with clear layer responsibilities, domain model relationships, and API interaction flows across the **Presentation**, **Business Logic**, and **Persistence** layers.

---

1. Introduction

HBnB is a simplified Airbnb-like application designed to manage **Users**, **Places**, **Reviews**, and **Amenities** through a REST API. In **Part 1**, the focus is on defining a clean architecture and a consistent domain model that can be implemented and extended without mixing responsibilities across layers.

Why this document exists: when multiple contributors implement endpoints and models in parallel, small inconsistencies in validation, ownership, and persistence rules can quickly cause bugs.

Scope of this document

- A **high-level packaging diagram** describing the layered architecture and connectors.
- A **Business Logic class diagram** describing the entities, shared base behavior, and relationships.
- **Sequence diagrams** for key API calls, with step-by-step explanations, Mermaid references, **control flow** (alt/opt), and **HTTP status codes**.

How to use it

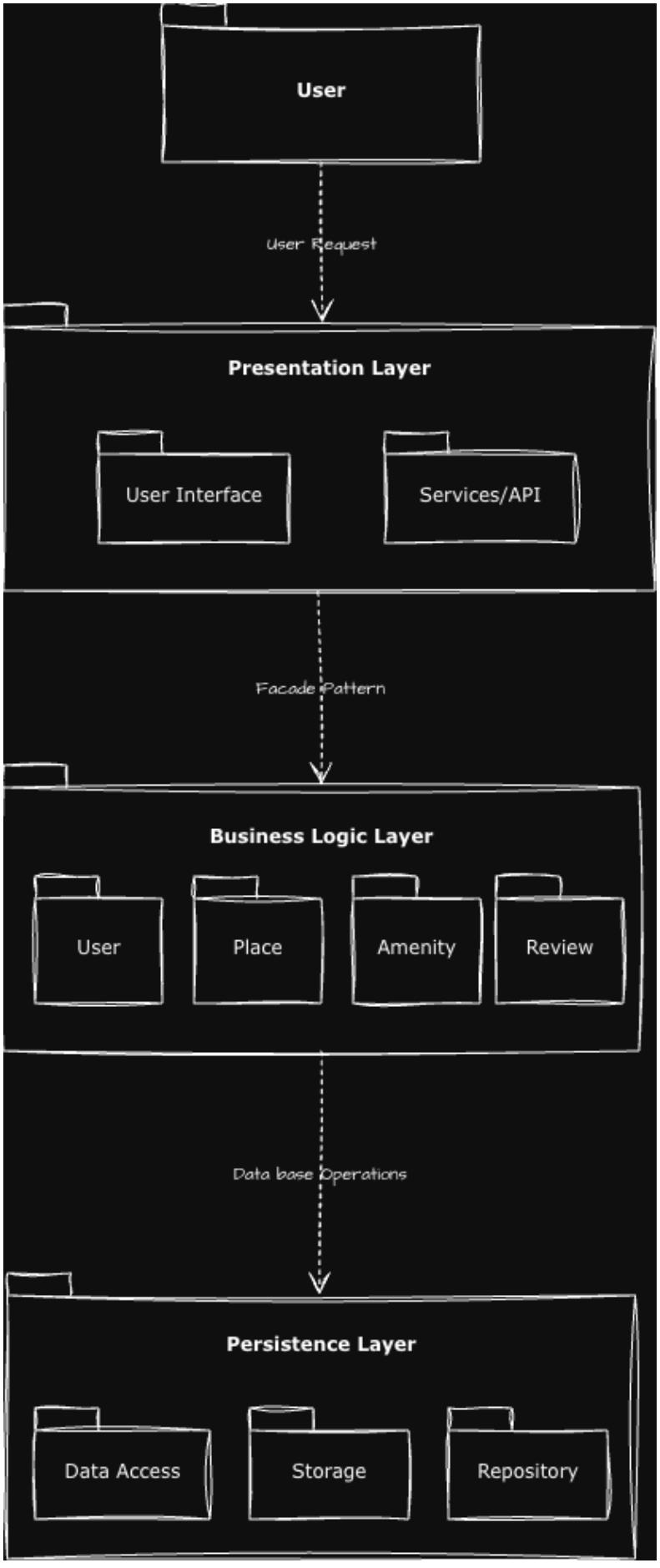
- Use the diagrams to keep **layer responsibilities** consistent (Presentation vs Business Logic vs Persistence).
- Use the class diagram to enforce **business rules** (validation, ownership, multiplicity) in the correct layer.
- Use the sequence diagrams as a reference for **expected control flow** and **error handling**.

🧠 Note: Mermaid diagrams are references for structure and behavior.

---

2. High-Level Packaging (Architecture Overview)

🧭 Reminder: this is a conceptual overview (layers + responsibilities), not a line-by-line implementation diagram.


User
User Request
Presentation Layer
User Interface
Services/API
Facade Pattern
Business Logic Layer
User
Place
Amenity
Review
Data base Operations
Persistence Layer
Data Access
Storage
Repository

2.1 Purpose of the diagram

This diagram illustrates the high-level architecture of the HBnB application. It shows the three main layers and how they interact, from the initial user request to persistence operations and back.

2.2 Key components

1. **Presentation Layer (top layer)**
    - **User Interface:** handles user interactions and displays information.
    - **Services / API:** manages HTTP requests and responses, and exposes API endpoints.
    - **Role:** entry point for client interactions, validates input at the HTTP boundary, and formats output.
2. **Facade Pattern (connector)**
    - Provides a simplified interface between the Presentation and Business Logic layers.
    - Reduces coupling by hiding the internal complexity of the business layer.
    - Offers a unified set of methods for the API to call.
3. **Business Logic Layer (middle layer)**
    - Contains the core business entities:
        - **User**
        - **Place**
        - **Amenity**
        - **Review**
    - **Role:** implements business rules, validations, and core application logic.
4. **Database Operations (connector)**
    - Represents the communication between Business Logic and Persistence.
    - Covers CRUD operations and data transactions.

5. **Persistence Layer (bottom layer)**
    - **Data Access:** provides methods to access stored data (queries, lookups, persistence helpers).
    - **Repository:** abstracts data access behind a consistent interface.
    - **Storage:** manages the underlying database.
    - **Role:** handles all data storage and retrieval operations.

2.3 Design decisions and rationale

Keywords: layered architecture, **Facade**, **Repository**, **separation of concerns**, **testability**

- **Three-layer architecture**
    - Separates concerns, improves maintainability, and supports testing and scalability.
- **Facade pattern**
    - Simplifies the interface for the API layer and reduces dependencies.
- **Repository pattern**
    - Keeps SQL out of business logic, improves modularity, and supports mocking during tests.

2.4 Why the arrows are dashed ("transparent")

The diagram uses **dashed arrows** to represent **logical flow and dependency direction**, not a literal implementation detail.

- Requests flow **top** → **down** across layers.
- Responses flow **bottom** → **up**.
- Dashed arrows avoid implying tight coupling or direct method calls between layers.

2.5 Data flow description

1. User request → user makes a request through the UI or API.
2. Presentation layer → receives and validates the request.

3. Facade → routes the request to the appropriate business logic.
4. Business logic → applies business rules and executes the use case.
5. Database operations → translates business needs into data operations.
6. Persistence layer → executes storage and retrieval.
7. Response → data flows back up to the user.

2.6 How this fits into the overall architecture

This layered architecture is meant to keep the system predictable as it grows:

- **Separation of concerns:** each layer has a single responsibility.
- **Maintainability:** changes in one layer have minimal impact on others.
- **Testability:** layers can be tested independently (mock the Facade/Repository).
- **Scalability:** layers can be scaled based on demand.
- **Security:** business logic and data access are protected behind the API boundary.
- **Flexibility:** easier to modify or replace individual components over time.

2.7 Benefits of this design

- Clear boundaries between responsibilities.
- Easier to understand and navigate the codebase.
- Uses standard patterns (Facade, Repository) familiar to most developers.
- Supports future enhancements without rewriting the entire stack.
- Reduces complexity through abstraction and consistent interfaces.

---

3. Business Logic Layer (Class Diagram)

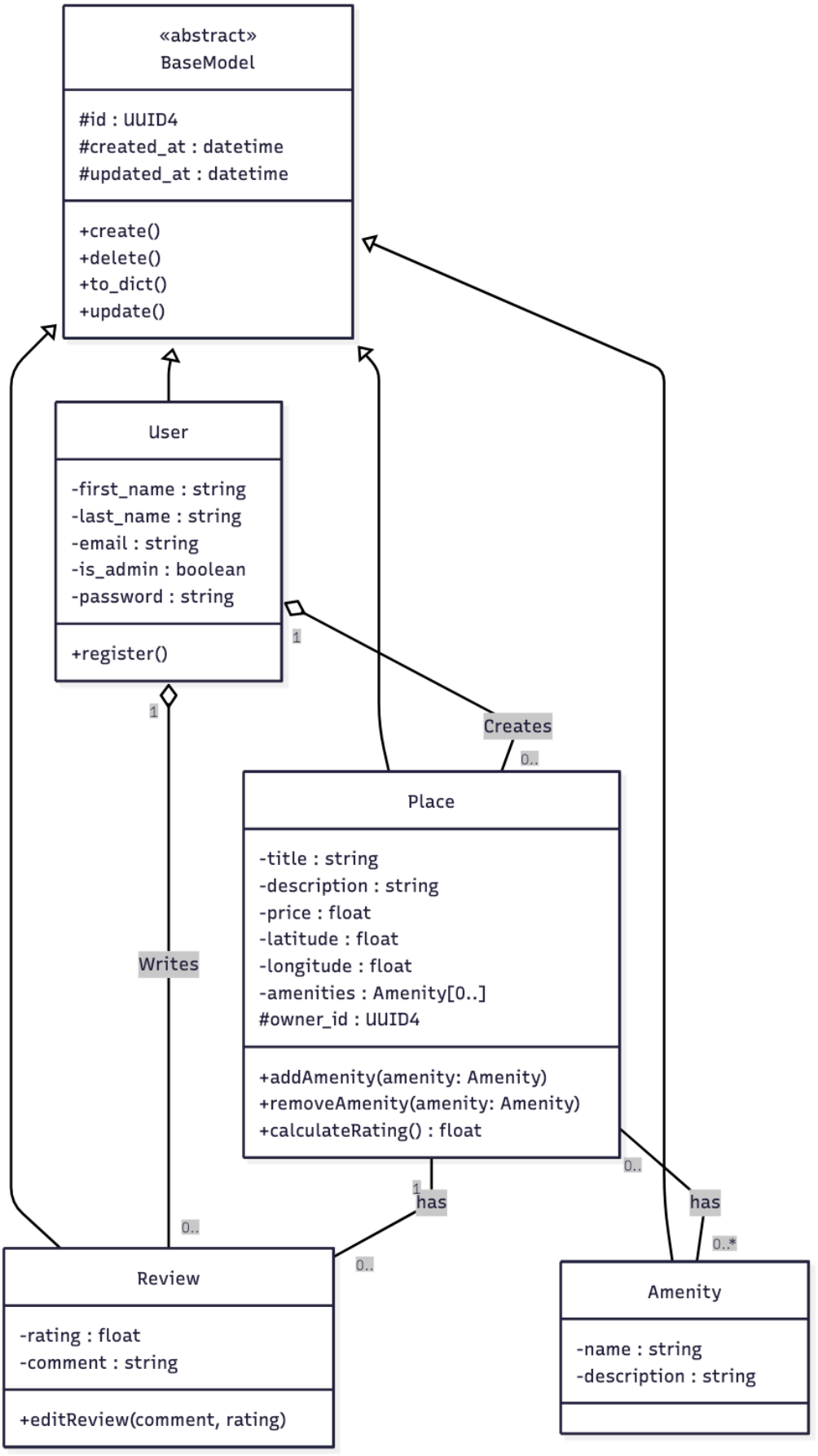
This section presents the detailed class diagram for the Business Logic Layer, explaining the entities, their attributes, methods, and relationships.

Key idea: entities inherit shared behavior from BaseModel, and relationships encode ownership and multiplicity (1, 0..*).

3.1 Class Diagram

This is a *living* diagram. We keep the first version for traceability, then we iterate as we discover improvements during implementation.

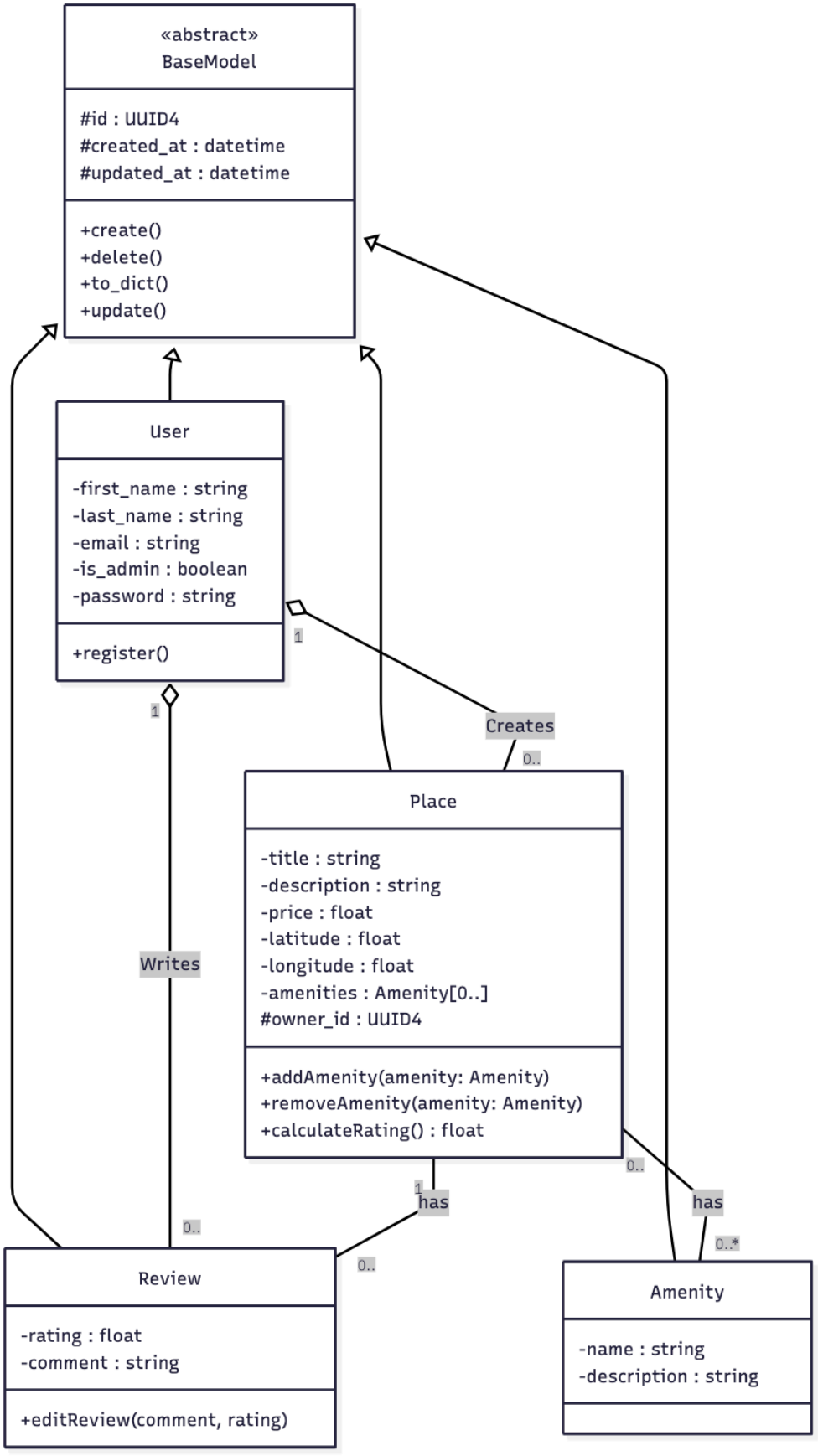
3.1.1 Version 1 (Initial draft)


«abstract»
BaseModel
#id : UUID4
#created_at : datetime
#updated_at : datetime
+create()
+delete()
+to_dict()
+update()
User
-first_name : string
-last_name : string
-email : string
-is_admin : boolean
-password : string
+register()
1
1
Creates
0..
Place
-title : string
-description : string
-price : float
-latitude : float
-longitude : float
-amenities : Amenity[0..]
#owner_id : UUID4
+addAmenity(amenity: Amenity)
+removeAmenity(amenity: Amenity)
+calculateRating() : float
Writes
0..
1
has
0..
0..
has
0..*
Review
-rating : float
-comment : string
+editReview(comment, rating)
Amenity
-name : string
-description : string

```
classDiagram
    direction TB
    class BaseModel {
        #id : UUID4
        #created_at : datetime
        #updated_at : datetime
        +create()
        +delete()
        +to_dict()
        +update()
    }
    class User {
        -first_name : string
        -last_name : string
        -email : string
        -is_admin : boolean
        -password : string
        +register()
    }
    class Place {
        -title : string
        -description : string
        -price : float
        -latitude : float
        -longitude : float
        -amenities : Amenity[0..]
        #owner_id : UUID4
        +addAmenity(amenity: Amenity)
        +removeAmenity(amenity: Amenity)
        +calculateRating() float
    }
    class Review {
        -rating : float
        -comment : string
        +editReview(comment, rating)
    }
    class Amenity {
        -name : string
```

```
        -description : string
    }
    <<abstract>> BaseModel
    BaseModel <|-- User
    BaseModel <|-- Place
    BaseModel <|-- Review
    BaseModel <|-- Amenity
    User "1" o-- "0.." Place : Creates
    User "1" o-- "0.." Review : Writes
    Place "1"-- "0.." Review : has
    Place "0.." -- "0..*" Amenity : has
```

3.1.2 Version 2 (Revised / current)


«abstract»
BaseModel
#id : UUID4
#created_at : datetime
#updated_at : datetime
+create()
+delete()
+to_dict()
+update()
User
-first_name : string
-last_name : string
-email : string
-is_admin : boolean
-password : string
+register()
Place
-title : string
-description : string
-price : float
-latitude : float
-longitude : float
-amenities : Amenity[0..]
#owner_id : UUID4
+addAmenity(amenity: Amenity)
+removeAmenity(amenity: Amenity)
+calculateRating() : float
Review
-rating : float
-comment : string
+editReview(comment, rating)
Amenity
-name : string
-description : string
1
1
Creates
0..
Writes
0..
1
has
0..
0..
has
0..*

```
classDiagram
    direction TB

    class BaseModel {
        <<abstract>>
        #id: UUID4
        #created_at: datetime
        #updated_at: datetime
        +update() void
        +to_dict() dict
    }

    class User {
        -first_name: string
        -last_name: string
        -email: string
        -is_admin: boolean
        -password: string
        +register() void
    }

    class Place {
        -title: string
        -description: string
        -price: float
        -latitude: float
        -longitude: float
        #owner_id: UUID4
        +addAmenity(amenity: Amenity) void
        +removeAmenity(amenity: Amenity) void
        +calculateRating() float
    }

    class Review {
        -rating: int
        -comment: string
        #user_id: UUID4
        #place_id: UUID4
```

```
        +editReview(comment: string, rating: int) void
    }

    class Amenity {
        -name: string
        -description: string
    }

    BaseModel <|-- User
    BaseModel <|-- Place
    BaseModel <|-- Review
    BaseModel <|-- Amenity

    User "1" o-- "0..*" Place : owns
    User "1" o-- "0..*" Review : writes
    Place "1" *-- "0..*" Review : receives
    Place "0..*" -- "0..*" Amenity : has
```

3.1.3 What changed

1. **Abstract syntax**
   - We moved `<<abstract>>` *inside* `BaseModel` because that is how Mermaid expects it.
2. `create()` / `delete()`
   - While implementing, we realized CRUD lives in the Persistence layer (repo/storage).
   - So `create()` and `delete()` do not belong in the Business Logic class diagram, and we removed them.
3. **Method signatures**
   - We added parameter and return types everywhere to make the contract clearer.
4. **Amenities redundancy**
   - The Place ↔ Amenity relationship already shows the link, so we removed the `amenities` attribute to avoid duplication.

5. **Review ownership**
    - We added `user_id` and `place_id` to `Review` so it is obvious what a review belongs to.
6. **Rating type**
    - We switched `rating` from `float` to `int` because ratings are typically 1–5.
7. **Cardinality + wording**
    - We fixed the `0..*` multiplicities and used clearer labels (`owns`, `writes`, `receives`).
8. **Lifecycle**
    - We kept Place → Review as composition to reflect cascade delete.

3.2 BaseModel (Abstract Class)

BaseModel is an abstract class (<<abstract>>). It cannot be instantiated directly and serves as a parent class. All entities inherit from it.

Protected attributes (#)

- `id` (UUID4): universal unique identifier (version 4). Guarantees uniqueness even in distributed systems.
- `created_at` (datetime): creation timestamp for traceability.
- `updated_at` (datetime): last update timestamp (updated on each save).

Public methods (+)

- `to_dict()`: serializes the object for JSON/API responses.
- `update()`: modifies attributes and refreshes `updated_at`.

> Note: persistence CRUD methods like `create()` / `delete()` are handled in the Persistence layer (Repository/Storage), so they are not part of the Business Logic diagram.

Design principle: centralizing these members in BaseModel applies DRY (Don't Repeat Yourself).

3.3 Entity Descriptions

3.3.1 User

Represents a platform user.

Attributes

- `first_name` (string): user first name.
- `last_name` (string): user last name.
- `email` (string): login identifier (unique, valid format).
- `is_admin` (boolean): access control flag.
- `password` (string): stored as a hashed value.

Method

- `register()`: handles registration logic (email uniqueness, format validation, password hashing).

3.3.2 Place

Represents an accommodation listing with geolocation data.

Attributes

- `title` (string): listing title.
- `description` (string): listing description.
- `price` (float): nightly price (must be > 0).
- `latitude` (float): GPS latitude.
- `longitude` (float): GPS longitude.
- Amenities are associated through a **many-to-many** relationship (see Place ↔ Amenity).
- `owner_id` (UUID4): reference to the owning user.

Methods

- `addAmenity()` / `removeAmenity()`: manage the many-to-many relationship with Amenity.
- `calculateRating()`: aggregates Review ratings to compute the average.

3.3.3 Review

Models a user review on a place.

Attributes

- `rating` (int): numeric score (typically 1–5).
- `comment` (string): review text.

Method

- `editReview(comment, rating)`: edits an existing review.

3.3.4 Amenity

Represents an equipment or feature (Wi-Fi, pool, parking).

Attributes

- `name` (string): amenity name.
- `description` (string): amenity description.

3.4 Relationships

3.4.1 Inheritance (Generalization)

All entities inherit from BaseModel. Each entity automatically includes the id, timestamps, and shared helper methods (like `to_dict()` and `update()`).

3.4.2 User → Place (Aggregation)

A User (1) can create 0..* Places. Aggregation shows a weak "has-a" relationship.

3.4.3 Place → Review (Composition)

A Place (1) owns 0..* Reviews. If a place is deleted, its reviews should be deleted as well (cascade delete).

3.4.4 User → Review

A User (1) can write 0..* Reviews.

3.4.5 Place ↔ Amenity (Many-to-Many)

A Place can have many amenities and an Amenity can belong to many places. In persistence, this requires a junction table (association table).

3.4.6 UML Visibility Notation

- `+` Public: accessible from any class.
- `-` Private: accessible only within the class.
- `#` Protected: accessible within the class and its subclasses.

3.4.7 Relationship notation (arrows and diamonds)

This diagram uses standard UML relationship symbols. Understanding these symbols is important because they express **ownership**, **lifecycle**, and **multiplicity**.

- **Inheritance / Generalization** (`<|--`)
  - Meaning: *is-a* relationship (child class inherits from parent).
  - In the diagram: `User`, `Place`, `Review`, and `Amenity` inherit common fields and helper methods from `BaseModel` (ex: `to_dict()`, `update()`).
- **Association** (`--`)
  - Meaning: generic *is-related-to* relationship (no ownership implied).
  - In this model: Place ↔ Amenity is many-to-many and neither side "owns" the other.
  - In the diagram: used for **Place** ↔ **Amenity**.
- **Aggregation** (`o--`, open diamond)
  - Meaning: weak *has-a* relationship (the child can exist independently).
  - In this model, think of it as "linked to" / "owned by" without strict lifecycle dependency.
  - In the diagram:
    - `User "1" o-- "0..*" Place`: a user can own many places.
    - `User "1" o-- "0..*" Review`: a user can write many reviews.
- **Composition** (`*--`, filled diamond)
  - Meaning: strong ownership (the child lifecycle depends on the parent).

    - In this model: reviews do not make sense without the place they belong to.
    - In the diagram:
        - `Place "1" *-- "0..*" Review`: deleting a place deletes its reviews too (**cascade delete**).
- **Multiplicity / Cardinality** (e.g., `"1"`, `"0..*"`)
    - Meaning: how many instances can participate in the relationship.
    - Quick read:
        - `"1"` means exactly one.
        - `"0..*"` means zero or more.
        - So `User "1" o-- "0..*" Place` = one user can have zero or many places.

3.5 Business rules implied by the model

Keywords: UUID4, uniqueness, referential integrity, cascade delete, many-to-many, junction table

These constraints are not fully expressed by UML alone, but they are implied by the attributes/methods and are typically enforced in the Business Logic layer (and sometimes also at the database level).

- **Identity and audit fields (all entities)**
    - Each entity has a unique `id` (UUID4).
    - `created_at` is set once at creation.
    - `updated_at` changes on each update.
- **User**
    - `email` must be **unique** (no two users share the same email).
    - `email` must have a **valid format**.
    - `password` must be **hashed** before storage.
    - `is_admin` controls authorization for admin-only actions.
- **Place**
    - `price` must be **> 0**.

    - `latitude` and `longitude` should be validated as real-world coordinates.
    - `owner_id` must reference an existing user (referential integrity).
- **Review**
    - `rating` should be constrained to an accepted range (for example **1–5**), and type should be consistent.
    - Business constraint commonly enforced: **one review per user per place** (prevents duplicates).
- **Relationships and persistence-level constraints**
    - **Place → Review composition** often implies **cascade delete** (deleting a place deletes its reviews).
    - **Place ↔ Amenity many-to-many** requires a **junction table** (for example `place_amenities(place_id, amenity_id)`), usually with a unique constraint on the pair.

---

4. Conventions Used in Sequence Diagrams

4.1 Layer responsibilities (quick reminder)

- **User (Client):** sends HTTP requests and receives responses.
- **API (Presentation):** request validation, auth checks (when required), routing, and HTTP status mapping.
- **Model (Business Logic):** domain rules, validations, orchestration of use cases.
- **Persistence:** database operations and transaction management.

4.2 Why SQL is not shown in sequence diagrams

Database work is represented using abstract methods (for example, `find_by_id()` or `check_duplicate_review()`) to avoid leaking storage details into the business logic representation.

4.3 Mermaid syntax cheatsheet

Syntax	Meaning	Example
`participant A`	Declare participant (an actor/service shown in the diagram: client, API, model/service, DB, etc.)	`participant User`
`A->>B`	Call (solid arrow)	`User->>API: POST /api/register`
`A-->>B`	Return (dashed arrow)	`API-->>User: HTTP 201`
`activate A` / `deactivate A`	Activation bar	`activate API`
`A->>A`	Self-call	`Model->>Model: validate()`
`alt ... else ... end`	If/else (branching)	Success vs error path
`opt ... end`	Optional block (only runs if condition is met)	Amenities provided

5. API Interaction Flow (Sequence Diagrams)

Each section below includes the Mermaid code and a step-by-step explanation of the flow.

5.1 User Registration (POST /api/register)

5.1.1 What this diagram represents

This diagram models how a new user creates an account:

- Input is validated.
- Email uniqueness is checked.
- A user record is saved using a transaction.
- The new user id is returned.

5.1.2 Participants

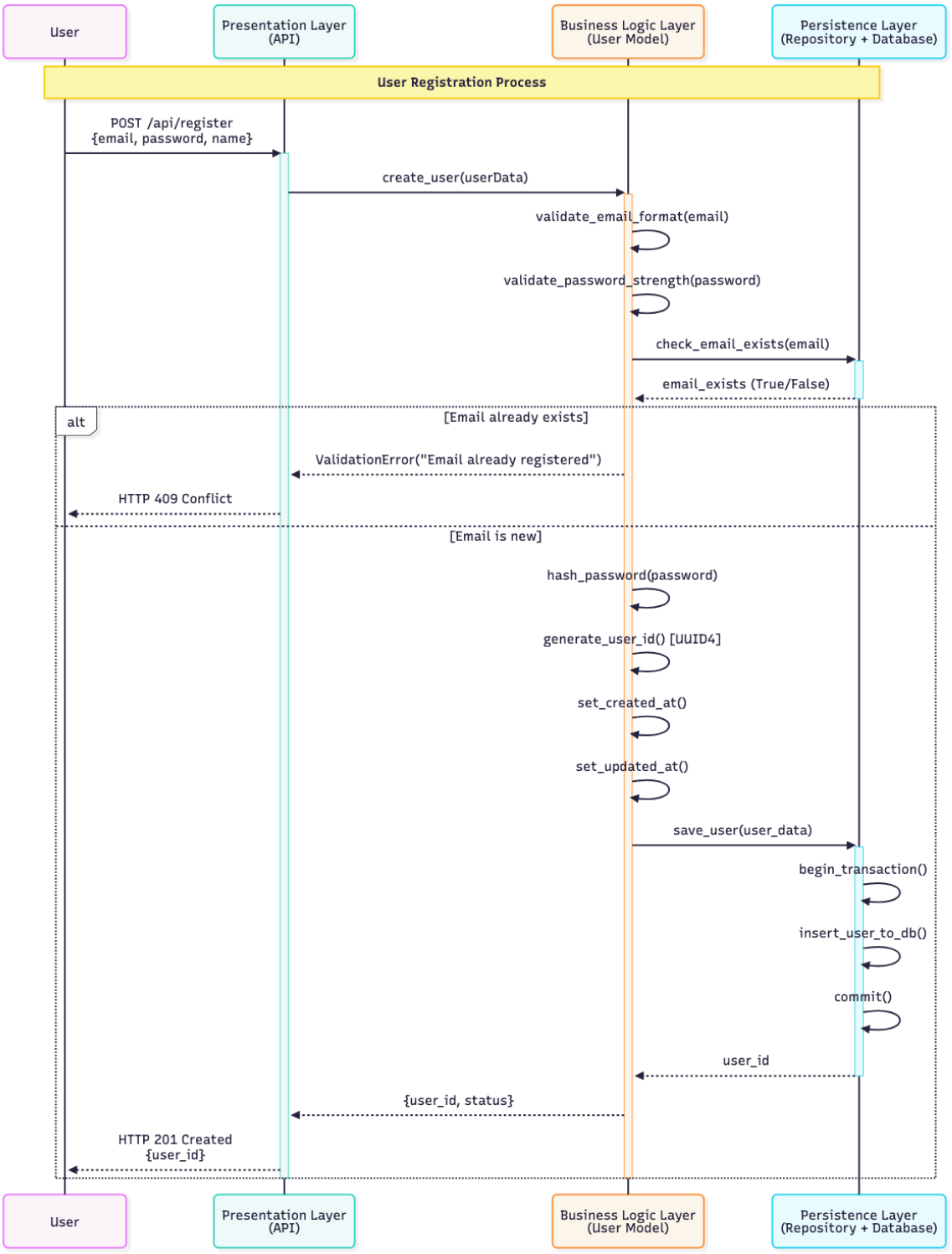
- **User**
- **API (Presentation Layer)**
- **UserModel (Business Logic Layer)**

- **Persistence (Repository + Database)**

5.1.3 Mermaid code

```
sequenceDiagram
    participant User
    participant API as Presentation Layer<br/>(API)
    participant UserModel as Business Logic Layer<br/>(User
Model)
    participant Persistence as Persistence Layer<br/>(Repos
itory + Database)

    Note over User,Persistence: User Registration Process

    User->>API: POST /api/register<br/>{email, password, na
me}
    activate API

    API->>UserModel: create_user(userData)
    activate UserModel

    UserModel->>UserModel: validate_email_format(email)
    UserModel->>UserModel: validate_password_strength(passw
ord)

    UserModel->>Persistence: check_email_exists(email)
    activate Persistence
    Persistence-->>UserModel: email_exists (True/False)
    deactivate Persistence

    alt Email already exists
        UserModel-->>API: ValidationError("Email already re
gistered")
        API-->>User: HTTP 409 Conflict

    else Email is new
        UserModel->>UserModel: hash_password(password)
        UserModel->>UserModel: generate_user_id() [UUID4]
```

```
        UserModel->>UserModel: set_created_at()
        UserModel->>UserModel: set_updated_at()

        UserModel->>Persistence: save_user(user_data)
        activate Persistence
        Persistence->>Persistence: begin_transaction()
        Persistence->>Persistence: insert_user_to_db()
        Persistence->>Persistence: commit()
        Persistence-->>UserModel: user_id
        deactivate Persistence

        UserModel-->>API: {user_id, status}
        API-->>User: HTTP 201 Created<br/>{user_id}
    end

    deactivate UserModel
    deactivate API
```



5.1.4 Step-by-step explanation

- **Client request**
  - User calls `POST /api/register` with `{email, password, name}`.
- **API boundary checks (Presentation)**

    - Check required fields.
    - Normalize input (trim email, optional lower-case).
    - If payload invalid → `400 Bad Request`.
- **Business call**
    - API calls `UserModel.create_user(userData)`.
- **Business validations (Business)**
    - Validate email format.
    - Validate password strength.
    - Optional: validate name rules.
    - If invalid → API returns `400 Bad Request`.
- **Uniqueness check (Persistence)**
    - `Persistence.check_email_exists(email)`.
    - If email already exists → `409 Conflict`.
- **Build user entity (Business)**
    - Hash password.
    - Generate UUID.
    - Set timestamps.
- **Persist (transaction) (Persistence)**
    - Begin transaction.
    - Insert user.
    - Commit.
    - On DB error → rollback → `500 Internal Server Error`.
- **Response**
    - Return `201 Created` with `{user_id}`.

Notes

- `409` = conflict with existing state (not a bad payload).
- In a real DB, email should still be unique at the DB level too.

5.2 Place Creation (POST /api/places)

5.2.1 What this diagram represents

This diagram shows how an authenticated user creates a new place:

- Token validation happens in the API layer.
- Place validation happens in the business layer.
- Persistence saves the place and optionally links amenities in one transaction.

5.2.2 Step-by-step explanation

- **Client request**
  - User calls `POST /api/places` with `{auth_token, title, description, price, latitude, longitude, amenities}`.
- **Auth gate (API)**
  - Validate token.
  - If invalid → `401 Unauthorized`.
- **Business call**
  - API calls `PlaceModel.create_place(placeData)`.
- **Business validations**
  - Validate title.
  - Validate `price > 0`.
  - Validate coordinates.
  - Optional: validate amenities list (format, no duplicates).
  - If invalid → `400 Bad Request`.
- **Build place entity**
  - Generate `place_id`.
  - Set `owner_id` from token.
  - Set timestamps.
- **Persist (transaction) (Persistence)**

- Insert place.
- If amenities provided → link via association table.
- Commit.

- **Response**
  - Return `201 Created` with `{place_id, title, price}`.

Notes

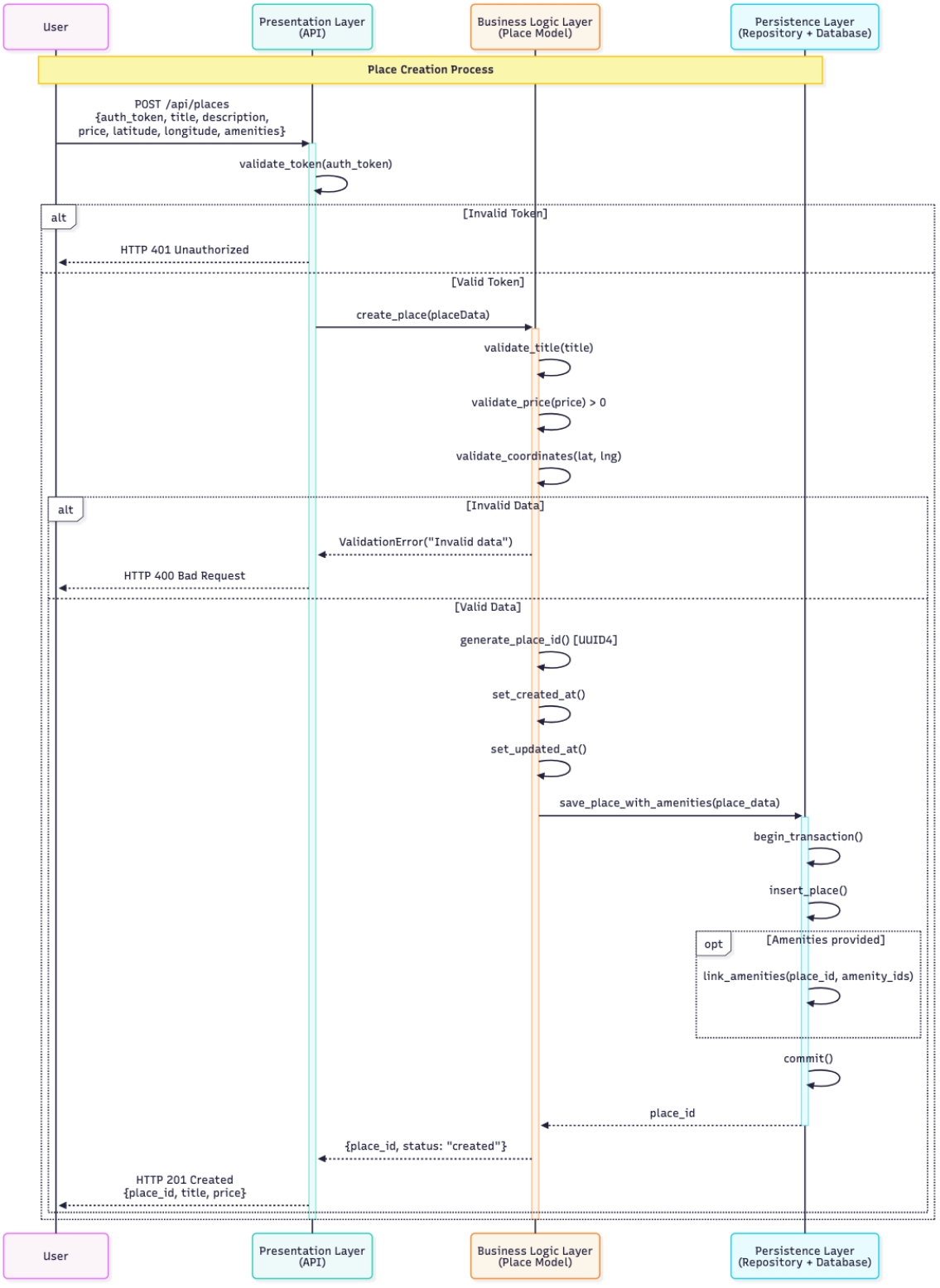
- If you validate amenity IDs, you can return `400` (invalid reference) or `404` (amenity not found) depending on your conventions.

5.2.3 Mermaid code

```
sequenceDiagram
    participant User
    participant API as Presentation Layer<br/>(API)
    participant PlaceModel as Business Logic Layer<br/>(Pla
ce Model)
    participant Persistence as Persistence Layer<br/>(Repos
itory + Database)

    Note over User,Persistence: Place Creation Process

    User->>API: POST /api/places<br/>{auth_token, title, de
scription,<br/>price, latitude, longitude, amenities}
    activate API

    API->>API: validate_token(auth_token)

    alt Invalid Token
        API-->>User: HTTP 401 Unauthorized

    else Valid Token
        API->>PlaceModel: create_place(placeData)
        activate PlaceModel

        PlaceModel->>PlaceModel: validate_title(title)
```

```
        PlaceModel->>PlaceModel: validate_price(price) > 0
        PlaceModel->>PlaceModel: validate_coordinates(lat,
lng)

        alt Invalid Data
            PlaceModel-->>API: ValidationError("Invalid dat
a")
            API-->>User: HTTP 400 Bad Request

        else Valid Data
            PlaceModel->>PlaceModel: generate_place_id() [U
UID4]
            PlaceModel->>PlaceModel: set_created_at()
            PlaceModel->>PlaceModel: set_updated_at()

            PlaceModel->>Persistence: save_place_with_ameni
ties(place_data)
            activate Persistence
            Persistence->>Persistence: begin_transaction()
            Persistence->>Persistence: insert_place()

            opt Amenities provided
                Persistence->>Persistence: link_amenities(p
lace_id, amenity_ids)
            end

            Persistence->>Persistence: commit()
            Persistence-->>PlaceModel: place_id
            deactivate Persistence

            PlaceModel-->>API: {place_id, status: "create
d"}
            API-->>User: HTTP 201 Created<br/>{place_id, ti
tle, price}
        end
    end
```

```
    deactivate PlaceModel
    deactivate API
```



5.3 Review Submission (POST /api/reviews)

5.3.1 What this diagram represents

This diagram shows review submission with the most critical transaction:

- Authentication gate.
- Input validation.
- Duplicate check.
- Transaction inserts a review and updates place average rating atomically.

5.3.2 Step-by-step explanation

- **Client request**
  - User calls `POST /api/reviews` with `{auth_token, place_id, rating, comment}`.
- **Auth gate (API)**
  - Validate token.
  - If invalid → `401 Unauthorized`.
- **Business call**
  - API calls `ReviewModel.create_review(reviewData)`.
- **Business validations**
  - Validate rating in **1–5**.
  - Validate comment rules.
  - If invalid → `400 Bad Request`.
- **Duplicate protection**
  - `Persistence.check_duplicate_review(user_id, place_id)`.
  - If already reviewed → `409 Conflict`.
- **Build review entity**
  - Generate `review_id`.
  - Set timestamps.
  - Attach `user_id` + `place_id`.

- **Persist atomically (transaction) (Persistence)**
  - Insert review.
  - Update place average rating (if you store a cached avg).
  - Commit.
  - On failure → rollback → `500 Internal Server Error`.
- **Response**
  - Return `201 Created` with `{review_id, rating}`.

Notes

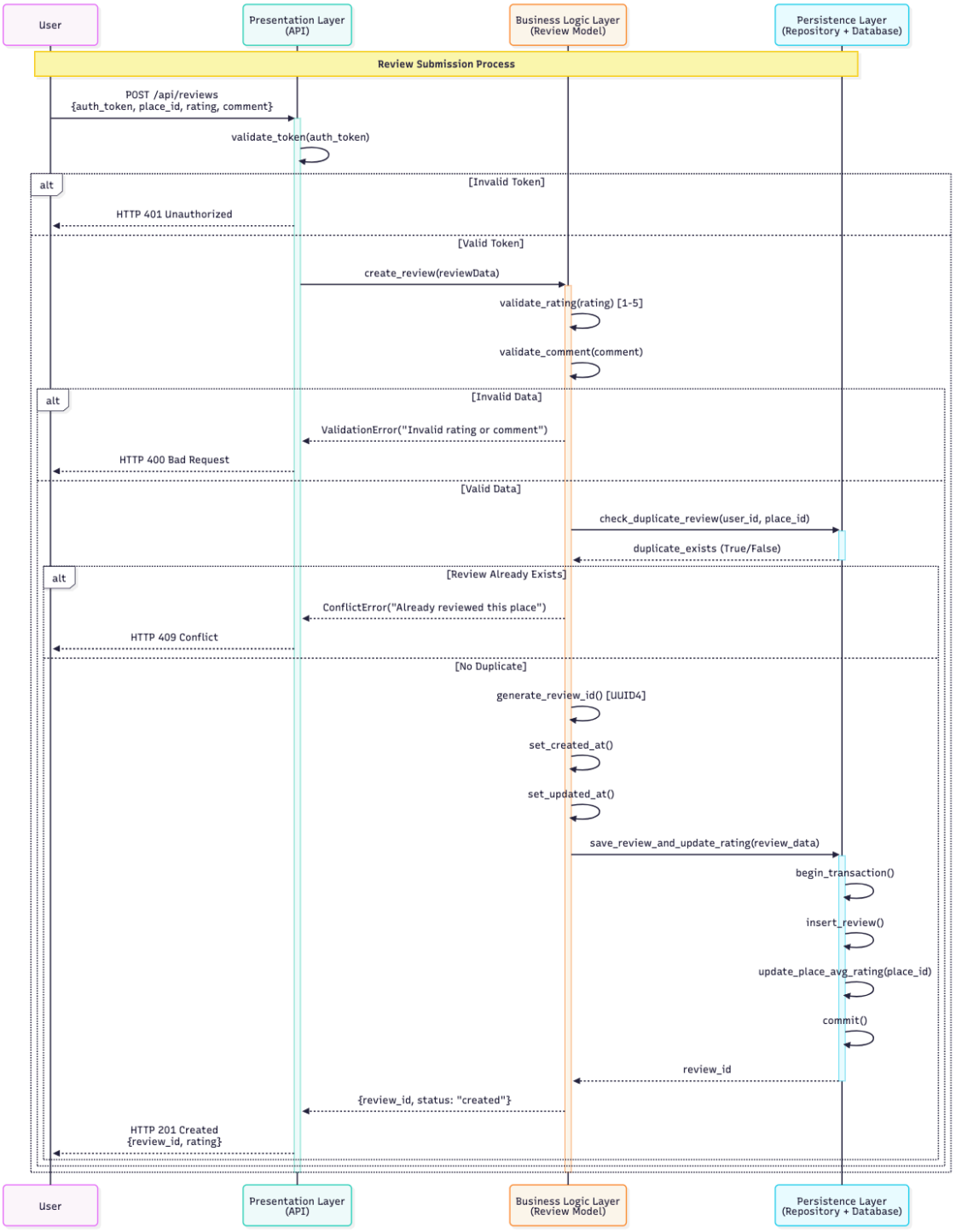
- You can also check that the place exists first; if not → `404 Not Found`.

5.3.3 Mermaid code

```
sequenceDiagram
    participant User
    participant API as Presentation Layer<br/>(API)
    participant ReviewModel as Business Logic Layer<br/>(Re
view Model)
    participant Persistence as Persistence Layer<br/>(Repos
itory + Database)

    Note over User,Persistence: Review Submission Process

    User->>API: POST /api/reviews<br/>{auth_token, place_i
d, rating, comment}
    activate API

    API->>API: validate_token(auth_token)

    alt Invalid Token
        API-->>User: HTTP 401 Unauthorized

    else Valid Token
        API->>ReviewModel: create_review(reviewData)
        activate ReviewModel
```

```
        ReviewModel->>ReviewModel: validate_rating(rating)
[1-5]
        ReviewModel->>ReviewModel: validate_comment(commen
t)

        alt Invalid Data
            ReviewModel-->>API: ValidationError("Invalid ra
ting or comment")
            API-->>User: HTTP 400 Bad Request

        else Valid Data
            ReviewModel->>Persistence: check_duplicate_revi
ew(user_id, place_id)
            activate Persistence
            Persistence-->>ReviewModel: duplicate_exists (T
rue/False)
            deactivate Persistence

            alt Review Already Exists
                ReviewModel-->>API: ConflictError("Already
reviewed this place")
                API-->>User: HTTP 409 Conflict

            else No Duplicate
                ReviewModel->>ReviewModel: generate_review_
id() [UUID4]
                ReviewModel->>ReviewModel: set_created_at()
                ReviewModel->>ReviewModel: set_updated_at()

                ReviewModel->>Persistence: save_review_and_
update_rating(review_data)
                activate Persistence
                Persistence->>Persistence: begin_transactio
n()
                Persistence->>Persistence: insert_review()
                Persistence->>Persistence: update_place_avg
_rating(place_id)
                Persistence->>Persistence: commit()
```

```
                Persistence-->>ReviewModel: review_id
                deactivate Persistence

                ReviewModel-->>API: {review_id, status: "cr
eated"}
                API-->>User: HTTP 201 Created<br/>{review_i
d, rating}
            end
        end
    end

    deactivate ReviewModel
    deactivate API
```



5.4 Fetching Places (GET endpoints)

5.4.1 What this diagram represents

This diagram groups four read-only scenarios:

- List all places.

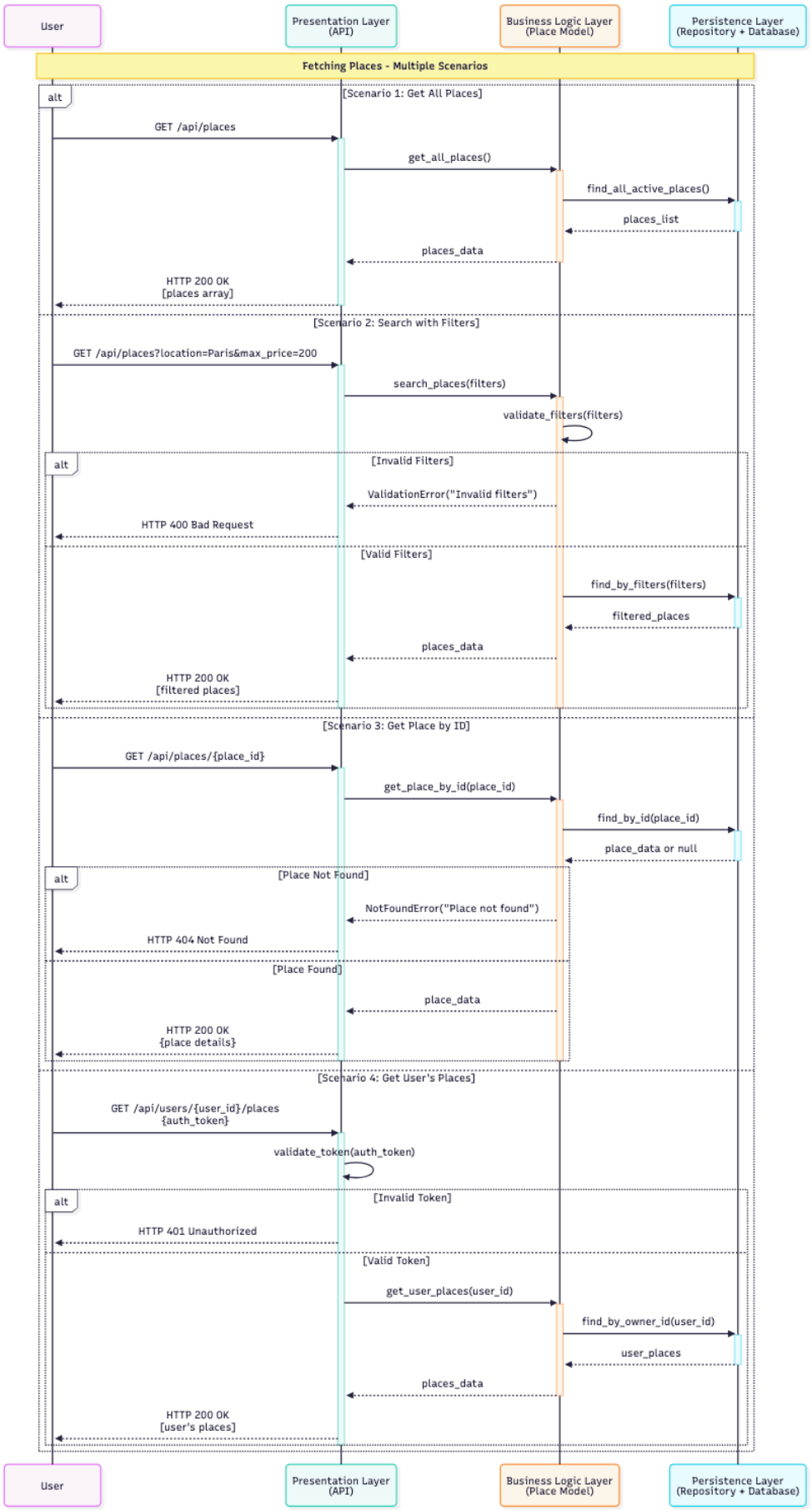
- Search with filters.
- Get a place by id.
- Get a user’s owned places (private, requires auth).

5.4.2 Mermaid code

```
sequenceDiagram
    participant User
    participant API as Presentation Layer<br/>(API)
    participant PlaceModel as Business Logic Layer<br/>(Pla
ce Model)
    participant Persistence as Persistence Layer<br/>(Repos
itory + Database)

    Note over User,Persistence: Fetching Places - Multiple
Scenarios

    alt Scenario 1: Get All Places
        User->>API: GET /api/places
        activate API
        API->>PlaceModel: get_all_places()
        activate PlaceModel
        PlaceModel->>Persistence: find_all_active_places()
        activate Persistence
        Persistence-->>PlaceModel: places_list
        deactivate Persistence
        PlaceModel-->>API: places_data
        deactivate PlaceModel
        API-->>User: HTTP 200 OK<br/>[places array]
        deactivate API

    else Scenario 2: Search with Filters
        User->>API: GET /api/places?location=Paris&max_pric
e=200
        activate API
        API->>PlaceModel: search_places(filters)
        activate PlaceModel
```

```
        PlaceModel->>PlaceModel: validate_filters(filters)

        alt Invalid Filters
            PlaceModel-->>API: ValidationError("Invalid fil
ters")
            API-->>User: HTTP 400 Bad Request

        else Valid Filters
            PlaceModel->>Persistence: find_by_filters(filte
rs)
            activate Persistence
            Persistence-->>PlaceModel: filtered_places
            deactivate Persistence
            PlaceModel-->>API: places_data
            API-->>User: HTTP 200 OK<br/>[filtered places]
        end
        deactivate PlaceModel
        deactivate API

    else Scenario 3: Get Place by ID
        User->>API: GET /api/places/{place_id}
        activate API
        API->>PlaceModel: get_place_by_id(place_id)
        activate PlaceModel
        PlaceModel->>Persistence: find_by_id(place_id)
        activate Persistence
        Persistence-->>PlaceModel: place_data or null
        deactivate Persistence

        alt Place Not Found
            PlaceModel-->>API: NotFoundError("Place not fou
nd")
            API-->>User: HTTP 404 Not Found

        else Place Found
            PlaceModel-->>API: place_data
            API-->>User: HTTP 200 OK<br/>{place details}
```

```
        end
        deactivate PlaceModel
        deactivate API

    else Scenario 4: Get User's Places
        User->>API: GET /api/users/{user_id}/places<br/>{au
th_token}
        activate API

        API->>API: validate_token(auth_token)

        alt Invalid Token
            API-->>User: HTTP 401 Unauthorized

        else Valid Token
            API->>PlaceModel: get_user_places(user_id)
            activate PlaceModel
            PlaceModel->>Persistence: find_by_owner_id(user
_id)
            activate Persistence
            Persistence-->>PlaceModel: user_places
            deactivate Persistence
            PlaceModel-->>API: places_data
            deactivate PlaceModel
            API-->>User: HTTP 200 OK<br/>[user's places]
        end
        deactivate API
    end
```


User
Presentation Layer (API)
Business Logic Layer (Place Model)
Persistence Layer (Repository + Database)
Fetching Places - Multiple Scenarios
alt
[Scenario 1: Get All Places]
GET /api/places
get_all_places()
find_all_active_places()
places_list
places_data
HTTP 200 OK
[places array]
[Scenario 2: Search with Filters]
GET /api/places?location=Paris&max_price=200
search_places(filters)
validate_filters(filters)
alt
[Invalid Filters]
ValidationError("Invalid filters")
HTTP 400 Bad Request
[Valid Filters]
find_by_filters(filters)
filtered_places
places_data
HTTP 200 OK
[filtered places]
[Scenario 3: Get Place by ID]
GET /api/places/{place_id}
get_place_by_id(place_id)
find_by_id(place_id)
place_data or null
alt
[Place Not Found]
NotFoundError("Place not found")
HTTP 404 Not Found
[Place Found]
place_data
HTTP 200 OK
{place details}
[Scenario 4: Get User's Places]
GET /api/users/{user_id}/places
{auth_token}
validate_token(auth_token)
alt
[Invalid Token]
HTTP 401 Unauthorized
[Valid Token]
get_user_places(user_id)
find_by_owner_id(user_id)
user_places
places_data
HTTP 200 OK
[user's places]
User
Presentation Layer (API)
Business Logic Layer (Place Model)
Persistence Layer (Repository + Database)

5.4.3 Step-by-step explanation

▼ Scenario 1: Get all places (public)

- User calls `GET /api/places`.
- API calls `PlaceModel.get_all_places()`.
- Business applies rules (example: only active places).
- Persistence returns the list (via `find_all_active_places()`).
- API returns `200 OK` with `[places]`.

▼ Scenario 2: Search with filters (public)

- User calls `GET /api/places?...`.
- API parses query params.
- Business validates filters.
- If filters invalid → `400 Bad Request`.
- Else Persistence returns filtered list (via `find_by_filters(filters)`).
- API returns `200 OK` with `[filtered places]`.

▼ Scenario 3: Get one place by id (public)

- User calls `GET /api/places/{place_id}`.
- API validates `{place_id}`.
- Persistence returns place or null.
- If not found → `404 Not Found`.
- Else → `200 OK` with `{place details}`.

▼ Scenario 4: Get a user's places (private)

- User calls `GET /api/users/{user_id}/places` with `auth_token`.
- API validates token.
- If token invalid → `401 Unauthorized`.
- Optional: if token user cannot access this `user_id` → `403 Forbidden`.
- Else Persistence returns owned places (via `find_by_owner_id(user_id)`).
- API returns `200 OK` with `[user's places]`.

Notes

- `400` = request format/filters invalid.
- `401` = not authenticated.
- `403` = authenticated but not allowed.
- `404` = resource not found.

Summary

✅

What the architecture enforces

- HBnB follows a **3-layer design** (**Presentation** → **Business Logic** → **Persistence**) with a **Facade** to reduce coupling.
- The **Presentation layer** is responsible for request parsing, auth checks, and mapping business errors to **HTTP status codes**.
- The **Business Logic layer** owns validation and domain rules. It should not leak SQL or storage details.
- The **Persistence layer** centralizes CRUD operations and transactions, enabling easier testing and swapping storage implementations.

What the domain model enforces

- Core entities are **User**, **Place**, **Review**, and **Amenity**, inheriting shared behavior from **BaseModel** (UUID + timestamps + common helpers).
- Relationships define ownership and expected lifecycle behavior:
  - **User** → **Place**: 1 to 0..* (creates)
  - **User** → **Review**: 1 to 0..* (writes)
  - **Place** → **Review**: 1 to 0..* (**composition**, so reviews are deleted when a place is deleted)
  - **Place** ↔ **Amenity**: 0..* to 0..* (many-to-many via a junction table)

What the API flows cover

- **POST /api/register**: validates input, checks email uniqueness, creates the user, returns `201` or an error code.
- **POST /api/places**: requires auth, validates place data, creates the place, returns `201` or an error code.
- **POST /api/reviews**: requires auth, validates the review, prevents duplicates, creates the review, returns `201` or an error code.

- **GET /api/places**: read-only flows (list, filtered search, by id, user-owned when applicable).

Project Info

📦 Team

- **Frances Palmer**
- **Sedra Ramarosaona**
- **Yohnny Marcellus**

Codebase

- Repository: `holbertonschool-hbnb`
- Directory: `part1`