Technical Documentation — HBnB

Contents: diagrams (package, class, sequence), explanatory notes, design decisions, and delivery instructions.

Introduction

This document consolidates the diagrams and explanatory notes required to understand the architecture and behavior of the **HBnB** application. It serves as a technical blueprint for implementation and maintenance. The document covers:

- Package diagram (high-level view)
- Detailed class diagram for the Business Logic layer
- Sequence diagrams for four key API calls
- Explanatory notes and design decisions

---

1. High-Level Architecture (Package Diagram)

The system is structured into three main layers: **Presentation**, **Business Logic**, **Persistence**. Below is a Mermaid representation of the package/architecture:

```
classDiagram
    class PresentationLayer {
        <>
        +Services
        +API Endpoints
    }
    class BusinessLogicLayer {
        <>
        +User
        +Place
        +Review
        +Amenity
    }
    class PersistenceLayer {
        <>
        +Database access objects
        +Repositories
    }
    PresentationLayer --> BusinessLogicLayer : Facade Pattern
    BusinessLogicLayer --> PersistenceLayer : CRUD operations
```

Purpose of the diagram: to show separation of responsibilities and dependency flow (Presentation depends on Business Logic, Business Logic depends on Persistence). **Design decisions**:

- Strict separation: controllers never access the database directly.
- Services / Use Cases centralize business logic, improving testability.

- Repositories encapsulate data access (ORM/SQL), allowing flexibility and easier testing.
- Create, Read, Update, Delete operations

---

2. Class Diagram — Business Logic Layer

The diagram below describes the main entities and their relationships: `BaseModel`, `User`, `Place`, `Review`, `City`, `Amenity`, along with the services and repositories.

```
classDiagram
    class UserEntity {
        -ID: UUID4
        -Admin: bool
        -first name: str
        -last name: str
        -email: str
        -password: str
        +register()
        +update()
        +deleted()
    }
    UserEntity "1" <|-- "0..*" Owner
    class Owner {
        +listedplace()
    }
    Owner "1" *-- "0..*" PlaceEntity : create
    class PlaceEntity {
        -ID: UUID4
        -title: str
        -description: str
        -price: float
        -latitude: float
        -longitude: float
        +create()
        +update()
        +delete()
    }
    PlaceEntity "1" *-- "0..*" AmenityEntity
    class AmenityEntity {
        -ID: UUID4
        -name: str
        -description: str
        +create()
        +update()
        +delete()
    }
    UserEntity "1"<|-- "2" Administrator : Valentin Loïc
    Administrator -- UserClient
    Administrator -- Owner
    Administrator -- ReviewEntity
    Administrator -- PlaceEntity
    Administrator -- AmenityEntity
```

```
    class Administrator {
        +modify*()
    }
    UserEntity "1" <|-- "0..*" UserClient
    class UserClient {
        -cardinfo: str
    }
    UserClient "1" *-- "0..1"ReviewEntity : write
    ReviewEntity -- "1" PlaceEntity
    class ReviewEntity {
        +id: UUID
        +rating: int
        +comment: str
        +created_at: datetime
        +updated_at: datetime
        +create()
        +delete()
    }
```

Explanation:

- `BaseModel` provides common fields and behavior (id, timestamps, basic CRUD).
- Entities `User`, `Place`, `Review` inherit from `BaseModel`.
- Services (`UserService`, `PlaceService`, `ReviewService`) orchestrate business logic: validations, rules, transactions.
- Repositories encapsulate access to the database. **Design choices**:
- Services maintain lightweight controllers (single responsibility).
- Repositories return domain objects (`Place`, `User`, etc.) rather than raw dicts for consistency.
- Domain logic (e.g., `calculate_rating`) resides in services or models depending on complexity.

---

3. Sequence Diagrams for API Interaction Flow

The following diagrams show the interaction flow between User (client), API (controllers), Business Logic (services/models), and Persistence (repositories/database).

3.1 User Registration

```
sequenceDiagram
participant User
participant API
participant UserEntity
participant Database
User->>API: POST /register (user info)
API->>UserEntity: validate() & create()
UserEntity->>Database: save()
Database-->>UserEntity: confirm save
UserEntity-->>API: return created user
API-->>User: return success + user ID
```

Notes:

- Check email uniqueness (`find_by_email`) before creating.
- Hash password in `UserService` before saving.
- Return token (JWT) if required by the spec.

3.2 Place Creation

```
sequenceDiagram
participant Owner
participant API
participant PlaceEntity
participant Database
Owner->>API: POST /places (place info)
API->>PlaceEntity: create() & validate()
PlaceEntity->>Database: save()
Database-->>PlaceEntity: confirm save
PlaceEntity-->>API: return place ID
API-->>Owner: return success + place info
```

Notes:

- Validate ownership (the user can create a place).
- Handle relationships (city_id, amenities) in the transaction.

3.3 Review Submission

```
sequenceDiagram
participant UserClient
participant API
participant ReviewEntity
participant PlaceEntity
participant Database
UserClient->>API: POST /places/:id/reviews (review info)
API->>ReviewEntity: create() & validate()
ReviewEntity->>PlaceEntity: link review to place
ReviewEntity->>Database: save()
Database-->>ReviewEntity: confirm save
ReviewEntity-->>API: return review ID
API-->>UserClient: return success + review info
```

Notes:

- Ensure user is allowed to post a review (e.g., only after booking).
- After saving, recalculate and update the place’s average rating.

3.4 Fetch Places

```
sequenceDiagram
participant User
participant API
participant PlaceEntity
participant Database
User->>API: GET /places?filters
API->>PlaceEntity: fetchList(filters)
PlaceEntity->>Database: query(filters)
Database-->>PlaceEntity: return list of places
PlaceEntity-->>API: return places
API-->>User: return list of places
```

Notes:

- Pagination and limits recommended (`page`, `per_page`).
- Sanitize/filter inputs (prevent injection).
- Option: cache results for frequent queries.

---

4. Explanatory Notes and Design Decisions

4.1 Layered separation

- **Presentation (API)**: routes, auth, request/response handling.
- **Business Logic (Services/Models)**: rules, validations, orchestrations.
- **Persistence (Repositories/Database)**: transactions and data access. Reason: maintainability, testability, easier evolution.

4.2 Transactions and consistency

- Multi-table operations (e.g., place + amenities) must run in atomic transactions.
- Services coordinate transactions (or delegate to ORM).

4.3 Validation and security

- Validate at API (format) and Service (business rules).
- Password hashing with bcrypt/argon2.
- JWT/sessions for authentication; verify roles/scopes.

4.4 Performance

- Use pagination, indexing on frequent queries (city_id, price).
- Possible caching with Redis.
