Part 1: Technical Documentation

Overview

This directory contains the complete technical documentation for the HBnB Evolution project's architecture and design. Part 1 focuses on establishing a solid foundation through comprehensive UML modeling and architectural planning.

📋 Contents

Task 0: High-Level Package Diagram

File: `package_diagram.md`
Responsible: Shaden Khaled Almansour
Description: Illustrates the three-layer architecture (Presentation, Business Logic, Persistence) and demonstrates how the Facade pattern facilitates communication between layers.

Task 1: Detailed Class Diagram

File: `class_diagram.md`
Responsible: Tariq Rashed Almutairi
Description: Comprehensive class diagram for the Business Logic layer, depicting all entities (User, Place, Review, Amenity), their attributes, methods, and relationships following SOLID principles.

Task 2: Sequence Diagrams for API Calls

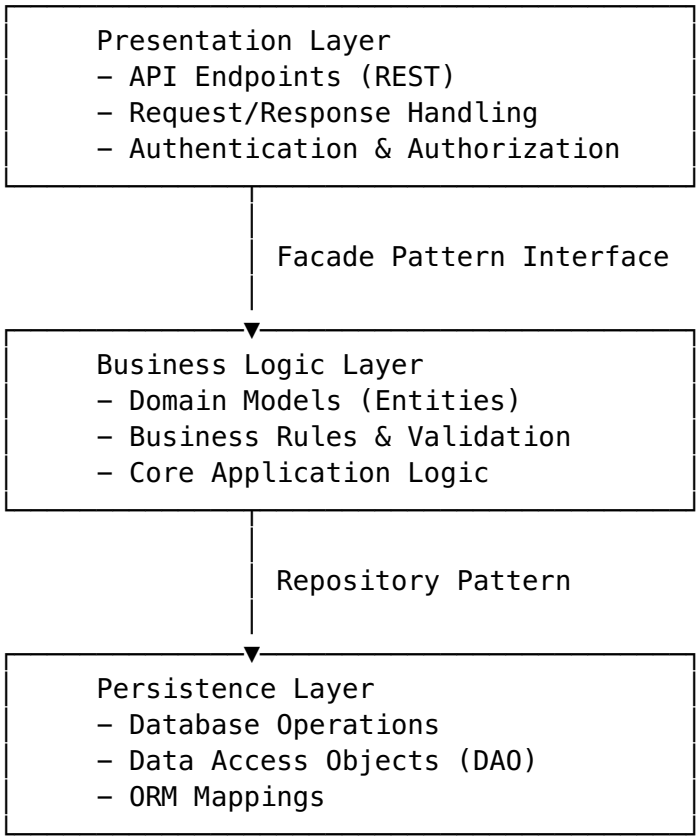
File: `sequence_diagrams.md`
Responsible: Nora Mohammed Alsakran
Description: Four detailed sequence diagrams showing the complete interaction flow for critical API operations: - User Registration - Place Creation - Review Submission - Fetching List of Places

Task 3: Compiled Technical Documentation

File: `technical_documentation.pdf`
Responsible: Team Collaboration
Description: Professional PDF compilation of all diagrams and technical specifications with comprehensive explanatory notes.

🏗️ Architecture Overview

Three-Layer Architecture



Design Patterns

1. **Facade Pattern**: Provides a unified interface to the Business Logic layer
2. **Repository Pattern**: Abstracts data persistence operations
3. **MVC Pattern**: Separates concerns in the presentation layer

🎯 Core Business Entities

User Entity

- Manages user accounts and authentication
- Supports both regular users and administrators

Place Entity

- Represents property listings with location and pricing
- Supports multiple amenities

Review Entity

- User-generated feedback with ratings (1-5 scale)

Amenity Entity

- Reusable features for places (WiFi, Pool, etc.)

🔄 Entity Relationships

- **User** → **Place**: One-to-Many
- **User** → **Review**: One-to-Many
- **Place** → **Review**: One-to-Many
- **Place** 🔲 **Amenity**: Many-to-Many

📐 UML Standards

All diagrams follow: - UML 2.5 Notation - Mermaid.js Syntax (GitHub rendering) - Professional documentation standards

👥 Team Contact

Name	Role	Email
Tariq Rashed Almutairi	Project Lead & Class Diagram	Tariq@hostworksa.com
Shaden Khaled Almansour	Package Architecture	shadeenn1424@gmail.com
Nora Mohammed Alsakran	Sequence Diagrams	NoraAlsakran1122@gmail.com

Organization: Holberton School Saudi Arabia
Location: Riyadh, Saudi Arabia

Last Updated: December 2025
Version: 1.0
Status: In Development

Task 0: High-Level Package Diagram

Overview

This diagram illustrates the three-layer architecture of the HBnB Evolution application, demonstrating the separation of concerns and communication patterns between layers using the Facade design pattern.

Architecture Layers

1. Presentation Layer (Services & API)

- **Responsibility**: Handles all user interactions and HTTP requests
- **Components**: RESTful API endpoints, Request/Response handlers
- **Communication**: Interacts with Business Logic via Facade pattern

2. Business Logic Layer (Models)

- **Responsibility**: Contains core business rules and domain models
- **Components**: Entity models (User, Place, Review, Amenity), Business rules validation
- **Communication**: Receives requests from Presentation, communicates with Persistence

3. Persistence Layer (Database)

- **Responsibility**: Manages data storage and retrieval
- **Components**: Database operations, Data access objects, ORM mappings
- **Communication**: Provides data to Business Logic layer

Facade Pattern Implementation

The Facade pattern provides a unified interface that: - Simplifies communication between layers - Reduces coupling and dependencies - Centralizes business logic access - Improves maintainability and testability

Package Diagram

```
graph TB
    subgraph Presentation["🎨 Presentation Layer"]
        API[API Services]
        Routes[Route Handlers]
        Controllers[Controllers]
    end
```

```
    subgraph Business["⚙️ Business Logic Layer"]
        Facade[Facade Pattern Interface]
        Models[Domain Models]
        User[User Model]
        Place[Place Model]
        Review[Review Model]
        Amenity[Amenity Model]
    end

    subgraph Persistence["💾 Persistence Layer"]
        Repository[Repository Pattern]
        Database[(Database)]
        ORM[ORM/Data Mappers]
    end

    %% Connections
    API --> Facade
    Routes --> Facade
    Controllers --> Facade

    Facade --> Models
    Models --> User
    Models --> Place
    Models --> Review
    Models --> Amenity

    User --> Repository
    Place --> Repository
    Review --> Repository
    Amenity --> Repository

    Repository --> ORM
    ORM --> Database

    %% Styling
    classDef presentationStyle fill:#e1f5ff,stroke:#01579b,stroke-
width:2px
    classDef businessStyle fill:#f3e5f5,stroke:#4a148c,stroke-
width:2px
    classDef persistenceStyle fill:#e8f5e9,stroke:#1b5e20,stroke-
width:2px

    class API,Routes,Controllers presentationStyle
    class Facade,Models,User,Place,Review,Amenity businessStyle
    class Repository,Database,ORM persistenceStyle
```

Alternative Detailed Package Diagram

```
classDiagram
    namespace PresentationLayer {
        class APIServices {
            <<Interface>>
```

```
        +handle_request()
        +send_response()
    }
    class Controllers {
        +UserController
        +PlaceController
        +ReviewController
        +AmenityController
    }
}

namespace BusinessLogicLayer {
    class Facade {
        <<Interface>>
        +create_user()
        +create_place()
        +create_review()
        +create_amenity()
        +get_entity()
        +update_entity()
        +delete_entity()
    }
    class Models {
        +User
        +Place
        +Review
        +Amenity
    }
}

namespace PersistenceLayer {
    class Repository {
        <<Interface>>
        +save()
        +find()
        +update()
        +delete()
    }
    class DatabaseAccess {
        +execute_query()
        +manage_transactions()
    }
}

APIServices --> Facade : Uses
Controllers --> Facade : Uses
Facade --> Models : Manages
Models --> Repository : Persisted by
Repository --> DatabaseAccess : Uses
```

Key Benefits of This Architecture

Separation of Concerns

- Each layer has a single, well-defined responsibility
- Changes in one layer don't ripple through others
- Easier to test and maintain

Facade Pattern Advantages

- **Simplified Interface**: Single entry point for business logic
- **Decoupling**: Presentation layer doesn't know about persistence details
- **Flexibility**: Easy to swap implementations without affecting clients
- **Centralized Logic**: Business rules are consolidated in one place

Scalability

- Layers can be scaled independently
- Easy to add new features without disrupting existing code
- Clear boundaries for team collaboration

Communication Flow Example

1. **Client Request** → API Services (Presentation Layer)
2. **API Services** → Facade (Business Logic Layer)
3. **Facade** → Models (Business Logic Layer)
4. **Models** → Repository (Persistence Layer)
5. **Repository** → Database (Persistence Layer)
6. **Response flows back** through the same path

Implementation Notes

- All inter-layer communication goes through well-defined interfaces
- The Facade acts as a gatekeeper, enforcing business rules before data operations
- Persistence layer is completely isolated from presentation concerns
- This architecture supports easy migration to microservices if needed

---

Created by: Shaden Khaled Almansour
Project: HBnB Evolution - Part 1
Date: December 2025

Task 1: Detailed Class Diagram for Business Logic Layer

Overview

This diagram represents the complete Business Logic layer of the HBnB Evolution application, showcasing all entities, their attributes, methods, and relationships according to SOLID principles and object-oriented design best practices.

Business Rules Summary

Entity Requirements

- **Unique Identification**: All entities use UUID4 for unique IDs
- **Audit Trail**: All entities track creation and update timestamps
- **Data Integrity**: Relationships enforce referential integrity
- **Validation**: All entities validate data before persistence

Detailed Class Diagram

```
classDiagram
    %% Base Entity Class
    class BaseEntity {
        <<abstract>>
        #UUID id
        #DateTime created_at
        #DateTime updated_at
        +save() void
        +update() void
        +delete() void
        #validate() bool
    }

    %% User Entity
    class User {
        -String first_name
        -String last_name
        -String email
        -String password_hash
        -Boolean is_admin
        -List~Place~ owned_places
        -List~Review~ reviews
        +__init__(first_name, last_name, email, password)
        +register() User
        +update_profile(data) void
        +authenticate(password) bool
        +hash_password(password) String
```

```
        +add_place(place) void
        +add_review(review) void
        +get_owned_places() List~Place~
        +get_reviews() List~Review~
        +is_administrator() bool
        +to_dict() dict
    }

    %% Place Entity
    class Place {
        -String title
        -String description
        -Decimal price
        -Float latitude
        -Float longitude
        -UUID owner_id
        -User owner
        -List~Amenity~ amenities
        -List~Review~ reviews
        +__init__(title, description, price, latitude, longitude,
owner)
        +add_amenity(amenity) void
        +remove_amenity(amenity) void
        +get_amenities() List~Amenity~
        +add_review(review) void
        +get_reviews() List~Review~
        +calculate_average_rating() float
        +update_details(data) void
        +validate_coordinates() bool
        +validate_price() bool
        +to_dict() dict
    }

    %% Review Entity
    class Review {
        -UUID place_id
        -UUID user_id
        -Integer rating
        -String comment
        -Place place
        -User user
        +__init__(place, user, rating, comment)
        +validate_rating() bool
        +update_review(rating, comment) void
        +get_place() Place
        +get_user() User
        +to_dict() dict
    }

    %% Amenity Entity
    class Amenity {
        -String name
        -String description
        -List~Place~ places
```

```
        +__init__(name, description)
        +update_details(data) void
        +get_associated_places() List~Place~
        +to_dict() dict
    }

    %% Place-Amenity Association Class
    class PlaceAmenity {
        -UUID place_id
        -UUID amenity_id
        -DateTime assigned_at
        +assign() void
        +unassign() void
    }

    %% Relationships
    BaseEntity <|-- User : inherits
    BaseEntity <|-- Place : inherits
    BaseEntity <|-- Review : inherits
    BaseEntity <|-- Amenity : inherits

    User "1" --> "0..*" Place : owns
    User "1" --> "0..*" Review : writes
    Place "1" --> "0..*" Review : has
    Place "0..*" --> "0..*" Amenity : features
    Place "1" --> "0..*" PlaceAmenity : through
    Amenity "1" --> "0..*" PlaceAmenity : through

    %% Notes
    note for User "Handles authentication,\nauthorization,
and\nuser management"
    note for Place "Central entity for\nproperty listings"
    note for Review "Maintains referential\nintegrity with User
and Place"
    note for Amenity "Reusable features\nshared across places"
```

Alternative Simplified View

```
classDiagram
    class User {
        +UUID id
        +String first_name
        +String last_name
        +String email
        +String password_hash
        +Boolean is_admin
        +DateTime created_at
        +DateTime updated_at
        +register()
        +update_profile()
        +authenticate()
    }
```

```
class Place {
    +UUID id
    +String title
    +String description
    +Decimal price
    +Float latitude
    +Float longitude
    +UUID owner_id
    +DateTime created_at
    +DateTime updated_at
    +add_amenity()
    +remove_amenity()
    +calculate_average_rating()
}

class Review {
    +UUID id
    +UUID place_id
    +UUID user_id
    +Integer rating
    +String comment
    +DateTime created_at
    +DateTime updated_at
    +validate_rating()
    +update_review()
}

class Amenity {
    +UUID id
    +String name
    +String description
    +DateTime created_at
    +DateTime updated_at
    +update_details()
}

User "1" -- "0..*" Place : owns >
User "1" -- "0..*" Review : writes >
Place "1" -- "0..*" Review : receives >
Place "0..*" -- "0..*" Amenity : features >
```

Detailed Entity Specifications

1. User Entity

Attributes

Attribute	Type	Constraints	Description
id	UUID4		Unique identifier

Attribute	Type	Constraints	Description
		Primary Key, Unique, Auto-generated	
first_name	String(50)	Not Null, Min 2 chars	User’s first name
last_name	String(50)	Not Null, Min 2 chars	User’s last name
email	String(120)	Unique, Not Null, Email format	User’s email address
password_hash	String(128)	Not Null	Hashed password (bcrypt)
is_admin	Boolean	Default: False	Administrative privileges flag
created_at	DateTime	Auto-generated	Account creation timestamp
updated_at	DateTime	Auto-updated	Last modification timestamp

Methods

- `register()`: Creates new user account with validation
- `update_profile(data)`: Updates user information
- `authenticate(password)`: Verifies password against hash
- `hash_password(password)`: Generates bcrypt hash
- `add_place(place)`: Associates owned place
- `add_review(review)`: Associates written review
- `to_dict()`: Serializes to dictionary

Business Rules

- Email must be unique across all users
- Password must be hashed before storage (never store plain text)
- Users can own multiple places
- Users can write multiple reviews
- Admin status affects system permissions

---

2. Place Entity

Attributes

Attribute	Type	Constraints	Description
id	UUID4	Primary Key, Unique, Auto-generated	Unique identifier
title	String(100)	Not Null, Min 5 chars	Place title/name
description	Text	Not Null, Max 1000 chars	Detailed description

Attribute	Type	Constraints	Description
price	Decimal(10,2)	Not Null, > 0	Price per night
latitude	Float	Not Null, Range: -90 to 90	Geographic latitude
longitude	Float	Not Null, Range: -180 to 180	Geographic longitude
owner_id	UUID4	Foreign Key → User.id, Not Null	Reference to owner
created_at	DateTime	Auto-generated	Listing creation timestamp
updated_at	DateTime	Auto-updated	Last modification timestamp

Methods

- `add_amenity(amenity)`: Associates amenity with place
- `remove_amenity(amenity)`: Removes amenity association
- `get_amenities()`: Returns list of amenities
- `add_review(review)`: Associates new review
- `calculate_average_rating()`: Computes average from all reviews
- `validate_coordinates()`: Ensures valid lat/long values
- `validate_price()`: Ensures price is positive
- `to_dict()`: Serializes to dictionary

Business Rules

- Each place must have exactly one owner
- Price must be positive decimal value
- Coordinates must be valid geographic values
- Places can have zero or more amenities
- Places can have zero or more reviews
- Average rating is calculated from all reviews

---

3. Review Entity

Attributes

Attribute	Type	Constraints	Description
id	UUID4	Primary Key, Unique, Auto-generated	Unique identifier
place_id	UUID4	Foreign Key → Place.id, Not Null	Reference to reviewed place
user_id	UUID4	Foreign Key → User.id, Not Null	Reference to reviewer
rating	Integer	Not Null, Range: 1-5	Star rating
comment	Text		Review text

Attribute	Type	Constraints	Description
		Not Null, Max 500 chars	
created_at	DateTime	Auto-generated	Review submission timestamp
updated_at	DateTime	Auto-updated	Last modification timestamp

Methods

- `validate_rating()`: Ensures rating is between 1-5
- `update_review(rating, comment)`: Updates existing review
- `get_place()`: Returns associated place object
- `get_user()`: Returns associated user object
- `to_dict()`: Serializes to dictionary

Business Rules

- Each review is associated with exactly one place
- Each review is written by exactly one user
- Rating must be integer between 1 and 5 (inclusive)
- Users can write multiple reviews (different places)
- One user can review the same place multiple times (update existing)

---

4. Amenity Entity

Attributes

Attribute	Type	Constraints	Description
id	UUID4	Primary Key, Unique, Auto-generated	Unique identifier
name	String(50)	Not Null, Unique	Amenity name
description	Text	Optional, Max 200 chars	Amenity description
created_at	DateTime	Auto-generated	Creation timestamp
updated_at	DateTime	Auto-updated	Last modification timestamp

Methods

- `update_details(data)`: Updates amenity information
- `get_associated_places()`: Returns all places with this amenity
- `to_dict()`: Serializes to dictionary

Business Rules

- Amenity names must be unique
- Amenities are reusable across multiple places

- Deleting an amenity removes associations but not places
- Multiple places can share the same amenity

Relationships Explained

1. User → Place (One-to-Many)

- **Type**: Composition/Ownership
- **Multiplicity**: 1 User can own 0..* Places
- **Implementation**: Place.owner_id references User.id
- **Cascade**: Deleting user may delete their places (business decision)

2. User → Review (One-to-Many)

- **Type**: Association
- **Multiplicity**: 1 User can write 0..* Reviews
- **Implementation**: Review.user_id references User.id
- **Cascade**: Deleting user typically deletes their reviews

3. Place → Review (One-to-Many)

- **Type**: Association
- **Multiplicity**: 1 Place can have 0..* Reviews
- **Implementation**: Review.place_id references Place.id
- **Cascade**: Deleting place deletes all its reviews

4. Place ? Amenity (Many-to-Many)

- **Type**: Association
- **Multiplicity**: Many Places can have Many Amenities
- **Implementation**: PlaceAmenity junction table
- **Cascade**: Deleting place removes associations; deleting amenity removes associations

---

SOLID Principles Implementation

Single Responsibility Principle (SRP)

- Each entity class has one primary responsibility
- User: Manages user data and authentication
- Place: Manages property listings
- Review: Manages ratings and feedback
- Amenity: Manages reusable features

Open/Closed Principle (OCP)

- BaseEntity provides extensible foundation
- New entity types can extend BaseEntity without modification
- Methods can be overridden for specialized behavior

Liskov Substitution Principle (LSP)

- All entities can be treated as BaseEntity
- Derived classes maintain base class contracts
- Polymorphic operations work on any entity type

Interface Segregation Principle (ISP)

- Entities implement only needed methods
- No entity is forced to depend on unused methods
- Clean, focused interfaces

Dependency Inversion Principle (DIP)

- Entities depend on abstractions (BaseEntity)
- High-level business logic doesn't depend on low-level details
- Repository pattern will provide data access abstraction

Design Patterns Used

1. Template Method Pattern

- BaseEntity provides common structure
- Derived classes implement specific validation

2. Repository Pattern (Implied)

- Each entity will have corresponding repository
- Abstracts data access from business logic

3. Factory Pattern (Future)

- Entity creation can be centralized
- Ensures proper initialization and validation

Validation Rules

User Validation

- Email format: RFC 5322 compliant
- Password: Min 8 chars, contains uppercase, lowercase, number
- Names: Alphabetic characters only, 2–50 chars

Place Validation

- Title: 5–100 characters
- Description: Max 1000 characters
- Price: Positive decimal, max 2 decimal places
- Latitude: -90.0 to 90.0
- Longitude: -180.0 to 180.0

Review Validation

- Rating: Integer 1–5 only
- Comment: Max 500 characters, **not** empty
- User cannot review own place

Amenity Validation

- Name: Unique, 3–50 characters
- Description: Max 200 characters

Example Usage Scenarios

Creating a User and Place

```
# Create user
user = User(
    first_name="Tariq",
    last_name="Almutairi",
    email="tariq@hostworksa.com",
    password="SecurePass123"
)
user.register()

# Create place
place = Place(
    title="Luxury Villa in Riyadh",
    description="Beautiful 5-bedroom villa in diplomatic quarter",
    price=1500.00,
    latitude=24.7136,
    longitude=46.6753,
```

```
    owner=user
)
place.save()

# Add amenities
wifi = Amenity(name="WiFi", description="High-speed internet")
pool = Amenity(name="Pool", description="Outdoor swimming pool")
place.add_amenity(wifi)
place.add_amenity(pool)
```

Creating a Review

```
# Another user reviews the place
reviewer = User.get_by_email("norah@example.sa")
review = Review(
    place=place,
    user=reviewer,
    rating=5,
    comment="Amazing property! Highly recommend."
)
review.save()

# Calculate average rating
avg_rating = place.calculate_average_rating()  # Returns 5.0
```

Future Enhancements

Phase 2 Considerations

- Add Booking entity (User books Place for dates)
- Add Payment entity (Track transactions)
- Add Location entity (City, Country hierarchy)
- Add Image entity (Multiple images per place)

Scalability Features

- Implement caching for frequently accessed data
- Add search/filter capabilities
- Implement soft delete (mark as deleted vs. actual deletion)
- Add versioning for entity changes

Created by: Tariq Rashed Almutairi
Organization: Holberton School Saudi Arabia
Project: HBnB Evolution - Part 1
Date: December 2025
Version: 1.0

Task 2: Sequence Diagrams for API Calls

Overview

This document presents sequence diagrams for four critical API operations in the HBnB Evolution application. Each diagram illustrates the complete interaction flow from the client request through all three architectural layers (Presentation, Business Logic, Persistence) and back.

API Call 1: User Registration

Description

A new user registers for an account in the system. This process involves validation, password hashing, and database persistence.

Sequence Diagram

```
sequenceDiagram
    actor Client
    participant API as API Layer
    participant Facade as Business Facade
    participant User as User Model
    participant Validator as Validation Service
    participant Hash as Password Hash Service
    participant Repo as User Repository
    participant DB as Database

    Client->>+API: POST /api/v1/users/register
    Note over Client,API: {first_name, last_name,<br/>email,
password}

    API->>+Facade: register_user(user_data)
    Facade->>+Validator: validate_email(email)
    Validator->>Validator: check_email_format()
    Validator-->>-Facade: validation_result

    alt Email Invalid
        Facade-->>API: ValidationError
        API-->>Client: 400 Bad Request
    else Email Valid
        Facade->>+Validator: check_email_unique(email)
        Validator->>+Repo: find_by_email(email)
        Repo->>+DB: SELECT * FROM users WHERE email=?
        DB-->>-Repo: null (not found)
        Repo-->>-Validator: not_exists
        Validator-->>-Facade: unique_confirmed
```

```
        Facade->>+Hash: hash_password(password)
        Hash->>Hash: bcrypt.hash()
        Hash-->>-Facade: password_hash

        Facade->>+User: new User(data, password_hash)
        User->>User: validate_attributes()
        User->>User: set_id(UUID4)
        User->>User: set_timestamps()
        User-->>-Facade: user_instance

        Facade->>+Repo: save(user_instance)
        Repo->>+DB: INSERT INTO users VALUES(...)
        DB-->>-Repo: success
        Repo-->>-Facade: user_saved

        Facade-->>-API: user_object
        API-->>-Client: 201 Created
        Note over Client,API: {id, email, first_name,<br/
>last_name, created_at}
    end
```

Flow Description

1. **Client Request**: Client sends POST request with user registration data
2. **API Reception**: API layer receives and forwards to Business Facade
3. **Email Validation**:
    - Check email format (RFC 5322)
    - Verify email uniqueness in database
4. **Password Security**: Hash password using bcrypt
5. **User Creation**: Instantiate User model with validated data
6. **Persistence**: Save user to database via repository
7. **Response**: Return created user data (excluding password)

Error Scenarios

- Invalid email format → 400 Bad Request
- Duplicate email → 409 Conflict
- Weak password → 400 Bad Request
- Database error → 500 Internal Server Error

API Call 2: Place Creation

Description

An authenticated user creates a new place listing. This requires authentication, validation of coordinates and price, and amenity associations.

Sequence Diagram

```
sequenceDiagram
    actor Client
    participant API as API Layer
    participant Auth as Auth Middleware
    participant Facade as Business Facade
    participant Place as Place Model
    participant User as User Model
    participant Amenity as Amenity Model
    participant PlaceRepo as Place Repository
    participant AmenityRepo as Amenity Repository
    participant DB as Database

    Client->>+API: POST /api/v1/places
    Note over Client,API: Authorization: Bearer token<br/>{title,
description, price,<br/>lat, lng, amenity_ids}

    API->>+Auth: validate_token(token)
    Auth->>Auth: decode_jwt(token)
    Auth->>+DB: verify_user_exists(user_id)
    DB-->>-Auth: user_found
    Auth-->>-API: user_authenticated(user_id)

    alt Authentication Failed
        API-->>Client: 401 Unauthorized
    else Authenticated
        API->>+Facade: create_place(place_data, user_id)

        Facade->>+User: get_user(user_id)
        Facade->>+PlaceRepo: find_user(user_id)
        PlaceRepo->>+DB: SELECT * FROM users WHERE id=?
        DB-->>-PlaceRepo: user_data
        PlaceRepo-->>-Facade: user_object
        Facade-->>-User: user_instance

        Facade->>+Place: new Place(data, owner)
        Place->>Place: validate_coordinates()
        Place->>Place: validate_price()
        Place->>Place: set_id(UUID4)
        Place->>Place: set_timestamps()
        Place-->>-Facade: place_instance

        loop For each amenity_id
            Facade->>+AmenityRepo: find_amenity(amenity_id)
            AmenityRepo->>+DB: SELECT * FROM amenities WHERE id=?
            DB-->>-AmenityRepo: amenity_data
            AmenityRepo-->>-Facade: amenity_object
            Facade->>Place: add_amenity(amenity_object)
        end

        Facade->>+PlaceRepo: save(place_instance)
        PlaceRepo->>+DB: BEGIN TRANSACTION
```

```
        DB-->>-PlaceRepo: transaction_started
        PlaceRepo->>+DB: INSERT INTO places VALUES(...)
        DB-->>-PlaceRepo: place_inserted
        PlaceRepo->>+DB: INSERT INTO place_amenities VALUES(...)
        DB-->>-PlaceRepo: associations_created
        PlaceRepo->>+DB: COMMIT
        DB-->>-PlaceRepo: committed
        PlaceRepo-->>-Facade: place_saved

        Facade-->>-API: place_object
        API-->>-Client: 201 Created
        Note over Client,API: {id, title, price, owner_id,<br/
>amenities[], created_at}
    end
```

Flow Description

1. **Authentication**: Validate JWT token and user existence
2. **User Verification**: Retrieve user object from database
3. **Place Validation**:
    - Validate geographic coordinates (-90/90, -180/180)
    - Validate price (positive decimal)
    - Validate required fields
4. **Amenity Association**:
    - Verify each amenity exists
    - Create many-to-many associations
5. **Transactional Save**:
    - Begin database transaction
    - Insert place record
    - Insert amenity associations
    - Commit transaction
6. **Response**: Return created place with amenities

Error Scenarios

- Invalid/expired token → 401 Unauthorized
- User not found → 404 Not Found
- Invalid coordinates → 400 Bad Request
- Negative price → 400 Bad Request
- Amenity not found → 404 Not Found
- Database constraint violation → 409 Conflict

API Call 3: Review Submission

Description

An authenticated user submits a review for a place they visited. The system validates the rating, ensures the place exists, and prevents users from reviewing their own properties.

Sequence Diagram

```
sequenceDiagram
    actor Client
    participant API as API Layer
    participant Auth as Auth Middleware
    participant Facade as Business Facade
    participant Review as Review Model
    participant Place as Place Model
    participant Validator as Validation Service
    participant ReviewRepo as Review Repository
    participant PlaceRepo as Place Repository
    participant DB as Database

    Client->>+API: POST /api/v1/places/{place_id}/reviews
    Note over Client,API: Authorization: Bearer token<br/>{rating,
comment}

    API->>+Auth: validate_token(token)
    Auth-->>-API: user_id

    alt Not Authenticated
        API-->>Client: 401 Unauthorized
    else Authenticated
        API->>+Facade: create_review(place_id, user_id,
review_data)

        Facade->>+PlaceRepo: find_place(place_id)
        PlaceRepo->>+DB: SELECT * FROM places WHERE id=?
        DB-->>-PlaceRepo: place_data
        PlaceRepo-->>-Facade: place_object

        alt Place Not Found
            Facade-->>API: NotFoundError
            API-->>Client: 404 Not Found
        else Place Exists
            Facade->>+Validator: validate_not_owner(user_id,
place.owner_id)
            Validator-->>-Facade: validation_result

            alt User is Owner
                Facade-->>API: ForbiddenError
                API-->>Client: 403 Forbidden<br/>"Cannot review
own place"
            else User Not Owner
                Facade->>+Review: new Review(place, user, data)
                Review->>Review: validate_rating(1-5)
                Review->>Review: validate_comment_length()
                Review->>Review: set_id(UUID4)
                Review->>Review: set_timestamps()
                Review-->>-Facade: review_instance

                Facade->>+ReviewRepo: save(review_instance)
```

```
                ReviewRepo->>+DB: INSERT INTO reviews VALUES(...)
                DB-->>-ReviewRepo: review_inserted
                ReviewRepo-->>-Facade: review_saved

                Facade->>Place: add_review(review_instance)
                Facade->>Place: recalculate_average_rating()

                Facade->>+PlaceRepo: update(place)
                PlaceRepo->>+DB: UPDATE places SET avg_rating=?
WHERE id=?
                DB-->>-PlaceRepo: updated
                PlaceRepo-->>-Facade: place_updated

                Facade-->>-API: review_object
                API-->>-Client: 201 Created
                Note over Client,API: {id, place_id, user_id,<br/
>rating, comment, created_at}
            end
        end
    end
```

Flow Description

1. **Authentication**: Validate user token
2. **Place Verification**: Ensure place exists in database
3. **Ownership Check**: Prevent users from reviewing own places
4. **Review Validation**:
   - Rating must be 1-5
   - Comment within length limits
5. **Review Creation**: Instantiate and validate review
6. **Rating Update**: Recalculate place's average rating
7. **Persistence**: Save review and update place statistics
8. **Response**: Return created review

Business Rules

- Users cannot review their own places
- Rating must be integer between 1 and 5
- Comment is required and limited to 500 characters
- One user can submit multiple reviews for same place (updates existing)

Error Scenarios

- Unauthenticated → 401 Unauthorized
- Place not found → 404 Not Found
- User reviewing own place → 403 Forbidden
- Invalid rating → 400 Bad Request
- Comment too long → 400 Bad Request

API Call 4: Fetching List of Places

Description

Retrieve a paginated and filtered list of places based on search criteria such as location, price range, and amenities.

Sequence Diagram

```
sequenceDiagram
    actor Client
    participant API as API Layer
    participant Facade as Business Facade
    participant Filter as Filter Service
    participant PlaceRepo as Place Repository
    participant Cache as Cache Layer
    participant DB as Database

    Client->>+API: GET /api/v1/places?
lat=24.7&lng=46.6&radius=10&min_price=500&max_price=2000&amenities=

    API->>+Facade: get_places(filters, pagination)
    Facade->>+Filter: build_query(filters)
    Filter->>Filter: validate_coordinates()
    Filter->>Filter: validate_price_range()
    Filter->>Filter: parse_amenity_ids()
    Filter-->>-Facade: query_params

    Facade->>+Cache: check_cache(query_hash)
    Cache-->>-Facade: cache_miss

    alt Cache Hit
        Cache-->>Facade: cached_results
        Facade-->>API: places_list
    else Cache Miss
        Facade->>+PlaceRepo: find_with_filters(query_params,
pagination)
        PlaceRepo->>PlaceRepo: build_sql_query()

        PlaceRepo->>+DB: SELECT p.* FROM places p<br/>JOIN
place_amenities pa ON p.id = pa.place_id<br/>WHERE distance(p.lat,
p.lng, ?, ?) <= ?<br/>AND p.price BETWEEN ? AND ?<br/>AND
pa.amenity_id IN (?)<br/>GROUP BY p.id<br/>LIMIT ? OFFSET ?
        DB-->>-PlaceRepo: result_set

        PlaceRepo->>PlaceRepo: map_to_objects()

        loop For each place
            PlaceRepo->>+DB: SELECT a.* FROM amenities a<br/>JOIN
place_amenities pa<br/>WHERE pa.place_id = ?
            DB-->>-PlaceRepo: amenities_list
            PlaceRepo->>PlaceRepo: attach_amenities(place)
```

```
        end

        PlaceRepo->>+DB: SELECT COUNT(*) FROM places p<br/>WHERE
[same conditions]
        DB-->>-PlaceRepo: total_count

        PlaceRepo-->>-Facade: places_list, total_count

        Facade->>+Cache: store_cache(query_hash, results, ttl=300)
        Cache-->>-Facade: cached

        Facade->>Facade: build_pagination_metadata()
        Facade-->>-API: {places[], pagination{}}
    end

    API-->>-Client: 200 OK
    Note over Client,API: {<br/>  data: [{place1},
{place2}...],<br/>  pagination: {page, limit, total},<br/>
filters_applied: {...}<br/>}
```

Flow Description

1. **Request Parsing**: Extract and validate query parameters
2. **Filter Validation**:
   - Validate coordinates and radius
   - Validate price range
   - Parse amenity filters
3. **Cache Check**: Look for cached results
4. **Database Query** (if cache miss):
   - Build complex SQL with joins
   - Apply geographic distance calculation
   - Apply price filters
   - Filter by amenities (many-to-many)
   - Apply pagination
5. **Eager Loading**: Fetch associated amenities for each place
6. **Count Query**: Get total matching records for pagination
7. **Cache Storage**: Store results for future requests
8. **Response**: Return paginated results with metadata

Query Parameters

Parameter	Type	Description	Default
lat	Float	Center latitude	Required
lng	Float	Center longitude	Required
radius	Integer	Search radius (km)	10
min_price	Decimal	Minimum price	0
max_price	Decimal	Maximum price	No limit
amenities	String	Comma-separated IDs	None
page	Integer	Page number	1

Parameter	Type	Description	Default
limit	Integer	Results per page	20

Optimization Strategies

1. **Caching**: 5-minute TTL for search results
2. **Database Indexes**:
   - Geographic index on (latitude, longitude)
   - Index on price
   - Composite index on (created_at, price)
3. **Eager Loading**: Fetch amenities in batch to avoid N+1 queries
4. **Query Optimization**: Use EXISTS instead of IN for large amenity lists

Response Structure


```
{
  "data": [
    {
      "id": "uuid-here",
      "title": "Luxury Villa",
      "price": 1500.00,
      "latitude": 24.7136,
      "longitude": 46.6753,
      "amenities": [
        {"id": "uuid", "name": "WiFi"},
        {"id": "uuid", "name": "Pool"}
      ],
      "average_rating": 4.5,
      "owner": {
        "id": "uuid",
        "first_name": "Tariq"
      }
    }
  ],
  "pagination": {
    "page": 1,
    "limit": 20,
    "total_results": 45,
    "total_pages": 3
  },
  "filters_applied": {
    "location": {"lat": 24.7, "lng": 46.6, "radius": 10},
    "price_range": {"min": 500, "max": 2000},
    "amenities": ["wifi", "pool"]
  }
}
```

Summary of Interactions

Common Patterns Across All APIs

1. **Authentication Layer**:
   - All modify operations require authentication
   - Read operations may be public or authenticated
2. **Validation Layer**:
   - Input validation at API layer (format, types)
   - Business validation at Facade layer (rules, constraints)
   - Database validation (constraints, foreign keys)
3. **Error Handling**:
   - Consistent HTTP status codes
   - Structured error responses
   - Proper exception propagation
4. **Transaction Management**:
   - ACID compliance for write operations
   - Rollback on failures
   - Consistent state maintenance
5. **Response Format**:
   - Consistent JSON structure
   - Include relevant related data
   - Exclude sensitive information (passwords)

Performance Considerations

Database Optimization

- Use connection pooling
- Implement prepared statements
- Apply proper indexing strategy
- Use EXPLAIN ANALYZE for query optimization

Caching Strategy

- Cache frequently accessed data (place listings)
- Invalidate cache on updates
- Use distributed cache (Redis) for scalability

Asynchronous Operations

- Send email notifications asynchronously
- Process analytics in background
- Generate reports offline

Security Measures

Authentication

- JWT tokens with expiration
- Token refresh mechanism
- Secure password hashing (bcrypt, cost factor 12)

Authorization

- Role-based access control
- Resource ownership verification
- Admin privilege checking

Input Sanitization

- SQL injection prevention (parameterized queries)
- XSS protection (output encoding)
- CSRF token validation

Rate Limiting

- Per-user API rate limits
- IP-based throttling
- Exponential backoff for failed attempts

Created by: Norah & Team
Project: HBnB Evolution - Part 1
Date: December 2025
Version: 1.0