HBnB Evolution: Technical Documentation

Overview

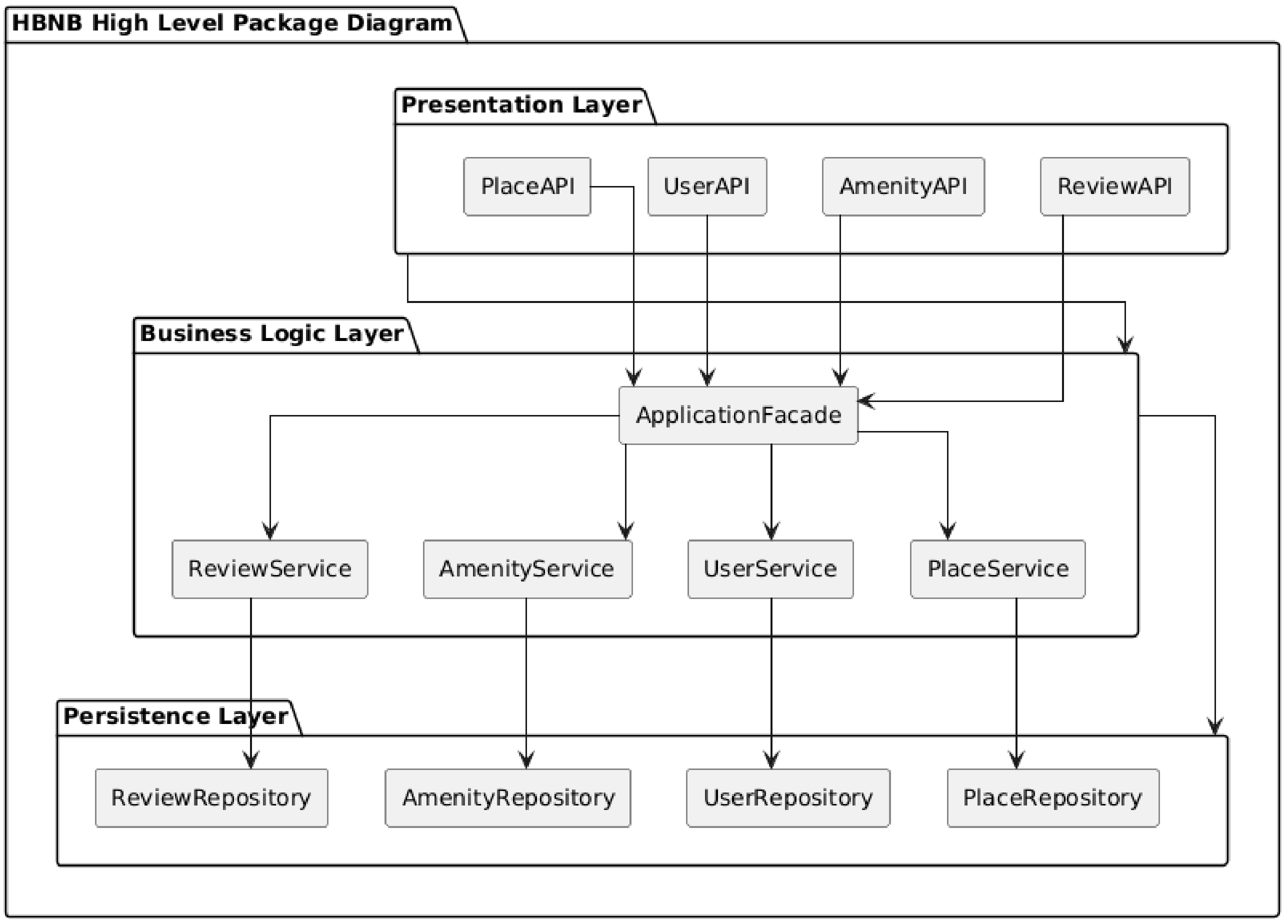
This document outlines the architecture and design of **HBnB Evolution**, a simplified AirBnB-like application. It covers the high-level package structure, detailed class design for the business logic layer, and sequence diagrams for key API operations. This serves as the foundation for development.

Architecture

HBnB Evolution uses a **layered architecture** with three primary layers:

- **Presentation Layer**: Exposes RESTful APIs for user interaction.
- **Business Logic Layer**: Manages core entities and business rules.
- **Persistence Layer**: Handles database storage and retrieval.

High-Level Package Diagram



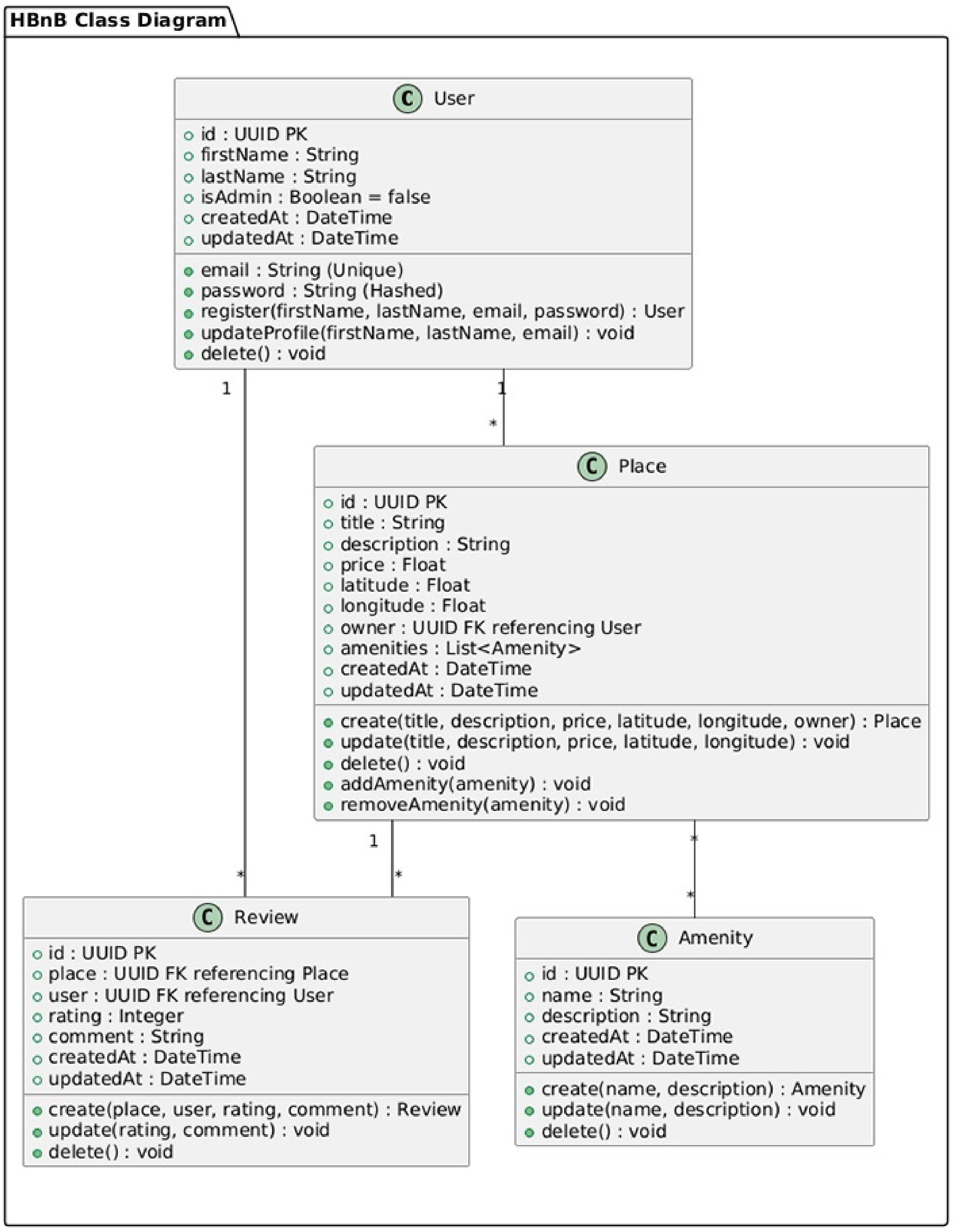
The system is organized into three packages, communicating via the **Facade pattern**:

Package	Components	Responsibilities
`presentation`	`UserAPI`, `PlaceAPI`, `ReviewAPI`, `AmenityAPI`	Handles API requests and responses.
`business`	`ApplicationFacade`, `UserService`, `PlaceService`, `ReviewService`, `AmenityService`	Processes logic and delegates to persistence.
`persistence`	`UserRepository`, `PlaceRepository`, `ReviewRepository`, `AmenityRepository`	Manages data CRUD operations.

Flow: `presentation` → `business` (via `ApplicationFacade`) → `persistence`.

Business Logic Layer

Class Diagram


HBnB Class Diagram
User
id : UUID PK
firstName : String
lastName : String
isAdmin : Boolean = false
createdAt : DateTime
updatedAt : DateTime
email : String (Unique)
password : String (Hashed)
register(firstName, lastName, email, password) : User
updateProfile(firstName, lastName, email) : void
delete() : void
1
1
*
Place
id : UUID PK
title : String
description : String
price : Float
latitude : Float
longitude : Float
owner : UUID FK referencing User
amenities : List<Amenity>
createdAt : DateTime
updatedAt : DateTime
create(title, description, price, latitude, longitude, owner) : Place
update(title, description, price, latitude, longitude) : void
delete() : void
addAmenity(amenity) : void
removeAmenity(amenity) : void
1
*
*
*
*
Review
id : UUID PK
place : UUID FK referencing Place
user : UUID FK referencing User
rating : Integer
comment : String
createdAt : DateTime
updatedAt : DateTime
create(place, user, rating, comment) : Review
update(rating, comment) : void
delete() : void
Amenity
id : UUID PK
name : String
description : String
createdAt : DateTime
updatedAt : DateTime
create(name, description) : Amenity
update(name, description) : void
delete() : void

The core entities include attributes, methods, and relationships:

User

- **Attributes**: `id: UUID`, `firstName: String`, `lastName: String`, `email: String`, `password: String`, `isAdmin: Boolean`, `createdAt: DateTime`, `updatedAt: DateTime`
- **Methods**: `register()`, `updateProfile()`, `delete()`

Place

- **Attributes**: `id: UUID`, `title: String`, `description: String`, `price: Float`, `latitude: Float`, `longitude: Float`, `owner: User`, `amenities: List<Amenity>`, `createdAt: DateTime`, `updatedAt: DateTime`
- **Methods**: `create()`, `update()`, `delete()`, `addAmenity()`, `removeAmenity()`

Review

- **Attributes**: `id: UUID`, `place: Place`, `user: User`, `rating: Integer`, `comment: String`, `createdAt: DateTime`, `updatedAt: DateTime`
- **Methods**: `create()`, `update()`, `delete()`

Amenity

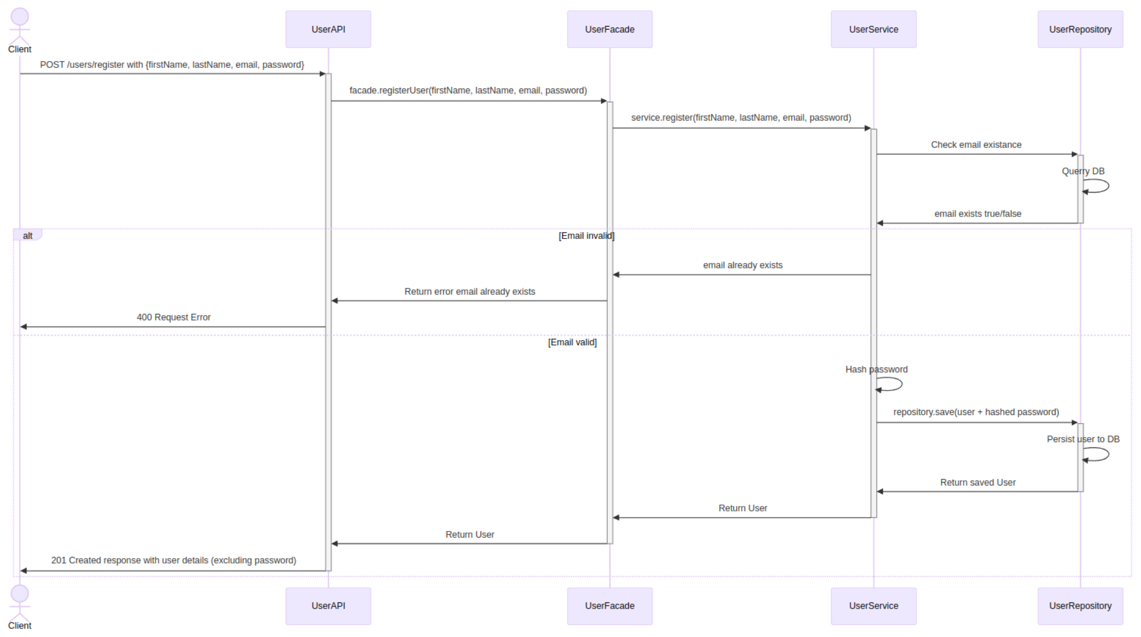
- **Attributes**: `id: UUID`, `name: String`, `description: String`, `createdAt: DateTime`, `updatedAt: DateTime`
- **Methods**: `create()`, `update()`, `delete()`

Relationships

- `User` → `Place`: One-to-Many
- `Place` ↔ `Amenity`: Many-to-Many
- `Place` → `Review`: One-to-Many
- `User` → `Review`: One-to-Many

API Sequence Diagrams

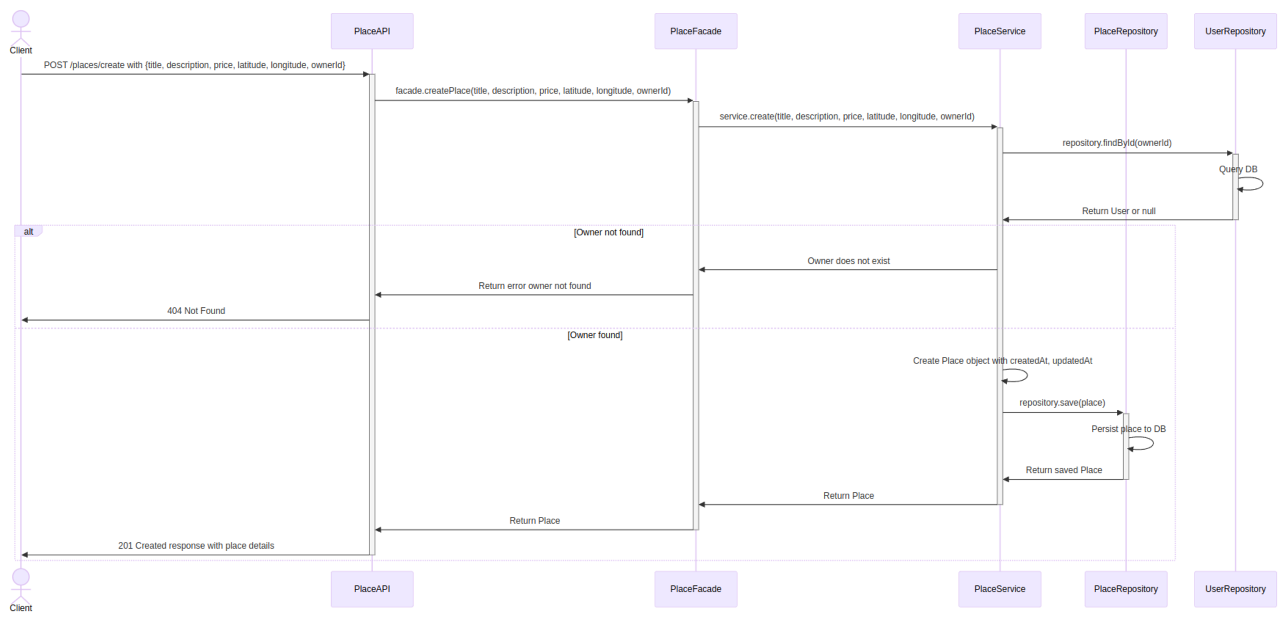
1. User Registration (`POST /users/register`)


Client
UserAPI
UserFacade
UserService
UserRepository
POST /users/register with {firstName, lastName, email, password}
facade.registerUser(firstName, lastName, email, password)
service.register(firstName, lastName, email, password)
Check email existance
Querry DB
email exists true/false
alt
[Email invalid]
email already exists
Return error email already exists
400 Request Error
[Email valid]
Hash password
repository.save(user + hashed password)
Persist user to DB
Return saved User
Return User
Return User
201 Created response with user details (excluding password)
UserAPI
UserFacade
UserService
UserRepository
Client

Participants: Client, `UserAPI`, `ApplicationFacade`, `UserService`, `UserRepository`
Steps:

1. The `Client` sends a `POST /users/register` request to the `UserAPI` with `firstName`, `lastName`, `email`, and `password`.
2. The `UserAPI` forwards the data by calling `UserFacade.registerUser()`, which then passes it to `UserService.register()` with the same details.
3. The `UserService` requests the `UserRepository` to check if the email already exists; the `UserRepository` queries the database and returns `true` if it exists or `false` if it doesn't.
4. If the email exists, the `UserService` sends an error back through the `UserFacade` and `UserAPI`, which responds to the `Client` with a `400 Request Error`. If the email is available, the process continues.
5. The `UserService` hashes the password to ensure it's stored securely.
6. The `UserService` sends the user data, including the hashed password, to `UserRepository.save()`; the `UserRepository` saves it to the database and returns the saved `User` object.
7. The `UserService` passes the saved `User` back through the `UserFacade` and `UserAPI`, which sends a `201 Created` response to the `Client` with user details, excluding the password.

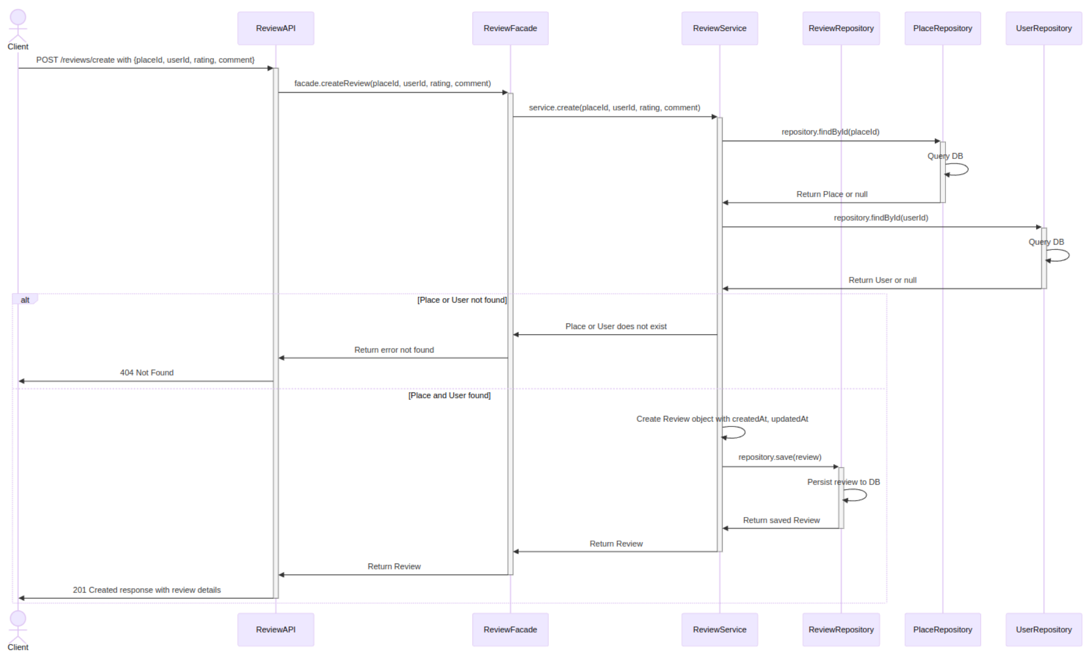
2. Place Creation (`POST /places/create`)


Client
PlaceAPI
PlaceFacade
PlaceService
PlaceRepository
UserRepository
POST /places/create with {title, description, price, latitude, longitude, ownerId}
facade.createPlace(title, description, price, latitude, longitude, ownerId)
service.create(title, description, price, latitude, longitude, ownerId)
repository.findById(ownerId)
Query DB
Return User or null
alt
[Owner not found]
Owner does not exist
Return error owner not found
404 Not Found
[Owner found]
Create Place object with createdAt, updatedAt
repository.save(place)
Persist place to DB
Return saved Place
Return Place
Return Place
201 Created response with place details
Client
PlaceAPI
PlaceFacade
PlaceService
PlaceRepository
UserRepository

Participants: Client, `PlaceAPI` , `ApplicationFacade` , `PlaceService` , `PlaceRepository` , `UserRepository`
Steps:

1. The `Client` sends a `POST /places/create` request to the `PlaceAPI` with `title` , `description` , `price` , `latitude` , `longitude` , and `ownerId` .
2. The `PlaceAPI` forwards the data by calling `PlaceFacade.createPlace()` , which then passes it to `PlaceService.create()` with the same details.
3. The `PlaceService` requests the `UserRepository` to find the user by calling `repository.findById(ownerId)` ; the `UserRepository` queries the database and returns the `User` object or `null` if not found.
4. If the owner is not found (i.e., `null`), the `PlaceService` sends an error back through the `PlaceFacade` and `PlaceAPI` , which responds to the `Client` with a `404 Not Found` . If the owner exists, the process continues.
5. The `PlaceService` creates a `Place` object, adding `createdAt` and `updatedAt` timestamps to the provided data.
6. The `PlaceService` sends the `Place` object to `PlaceRepository.save()` ; the `PlaceRepository` persists it to the database and returns the saved `Place` object.
7. The `PlaceService` passes the saved `Place` back through the `PlaceFacade` and `PlaceAPI` , which sends a `201 Created` response to the `Client` with the place details.

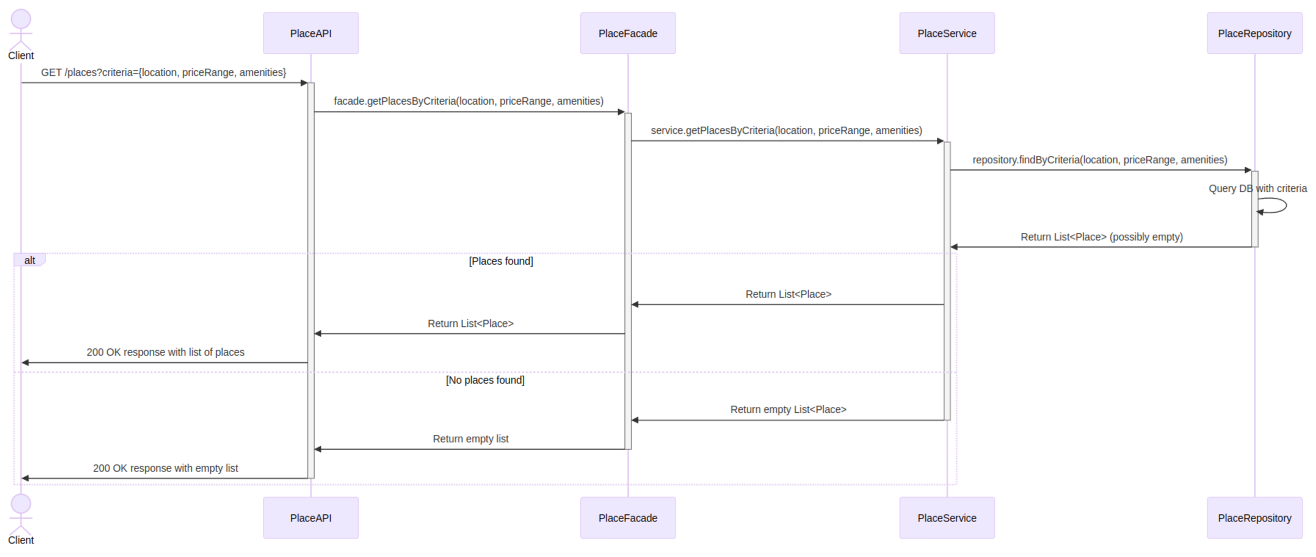
3. Review Submission (`POST /reviews/create`)



Participants: Client, `ReviewAPI`, `ApplicationFacade`, `ReviewService`, `ReviewRepository`, `PlaceRepository`, `UserRepository`
Steps:

1. The `Client` sends a `POST /reviews/create` request to the `ReviewAPI` with `placeId`, `userId`, `rating`, and `comment`.
2. The `ReviewAPI` forwards the data by calling `ReviewFacade.createReview()`, which then passes it to `ReviewService.create()` with the same details.
3. The `ReviewService` requests the `PlaceRepository` to find the place by calling `repository.findById(placeId)`; the `PlaceRepository` queries the database and returns the `Place` object or `null` if not found.
4. The `ReviewService` requests the `UserRepository` to find the user by calling `repository.findById(userId)`; the `UserRepository` queries the database and returns the `User` object or `null` if not found.
5. If either the place or user is not found (i.e., `null`), the `ReviewService` sends an error back through the `ReviewFacade` and `ReviewAPI`, which responds to the `Client` with a `404 Not Found`. If both the place and user exist, the process continues.
6. The `ReviewService` creates a `Review` object, adding `createdAt` and `updatedAt` timestamps to the provided data.
7. The `ReviewService` sends the `Review` object to `ReviewRepository.save()`; the `ReviewRepository` persists it to the database and returns the saved `Review` object.
8. The `ReviewService` passes the saved `Review` back through the `ReviewFacade` and `ReviewAPI`, which sends a `201 Created` response to the `Client` with the review details.

4. Fetch Places (`GET /places`)


Client
PlaceAPI
PlaceFacade
PlaceService
PlaceRepository
GET /places?criteria={location, priceRange, amenities}
facade.getPlacesByCriteria(location, priceRange, amenities)
service.getPlacesByCriteria(location, priceRange, amenities)
repository.findByCriteria(location, priceRange, amenities)
Query DB with criteria
Return List<Place> (possibly empty)
alt
[Places found]
Return List<Place>
Return List<Place>
200 OK response with list of places
[No places found]
Return empty List<Place>
Return empty list
200 OK response with empty list
Client
PlaceAPI
PlaceFacade
PlaceService
PlaceRepository

Participants: Client, `PlaceAPI`, `ApplicationFacade`, `PlaceService`, `PlaceRepository`
Steps:

1. The `Client` sends a `GET /places?criteria={location, priceRange, amenities}` request to the `PlaceAPI` with search criteria: `location`, `priceRange`, and `amenities`.
2. The `PlaceAPI` forwards the request by calling `PlaceFacade.getPlacesByCriteria()`, which then passes it to `PlaceService.getPlacesByCriteria()` with the same criteria.
3. The `PlaceService` requests the `PlaceRepository` to find places by calling `repository.findByCriteria(location, priceRange, amenities)`; the `PlaceRepository` queries the database using the provided criteria and returns a `List<Place>`, which may be empty.
4. If places matching the criteria are found, the `PlaceService` sends the `List<Place>` back through the `PlaceFacade` and `PlaceAPI`, which responds to the `Client` with a `200 OK` response containing the list of places. If no places are found, the `PlaceService` returns an empty `List<Place>`.
5. In the case of an empty list, the empty `List<Place>` is passed back through the `PlaceFacade` and `PlaceAPI`, which sends a `200 OK` response to the `Client` with the empty list.