Melvin Redondo–Tanis C24 Fréjus
Zakaria Aattache C24 Lille

HbNb - UML

Introduction:

The HbNb project is a simplified version of an AirBnB-like application, where users can register, list properties, manage amenities, and submit reviews. This documentation outlines the system's architecture and design, serving as a guide during the implementation process. It provides detailed breakdown of the structure, focusing on the high-level architecture, business logic, and key API interactions.

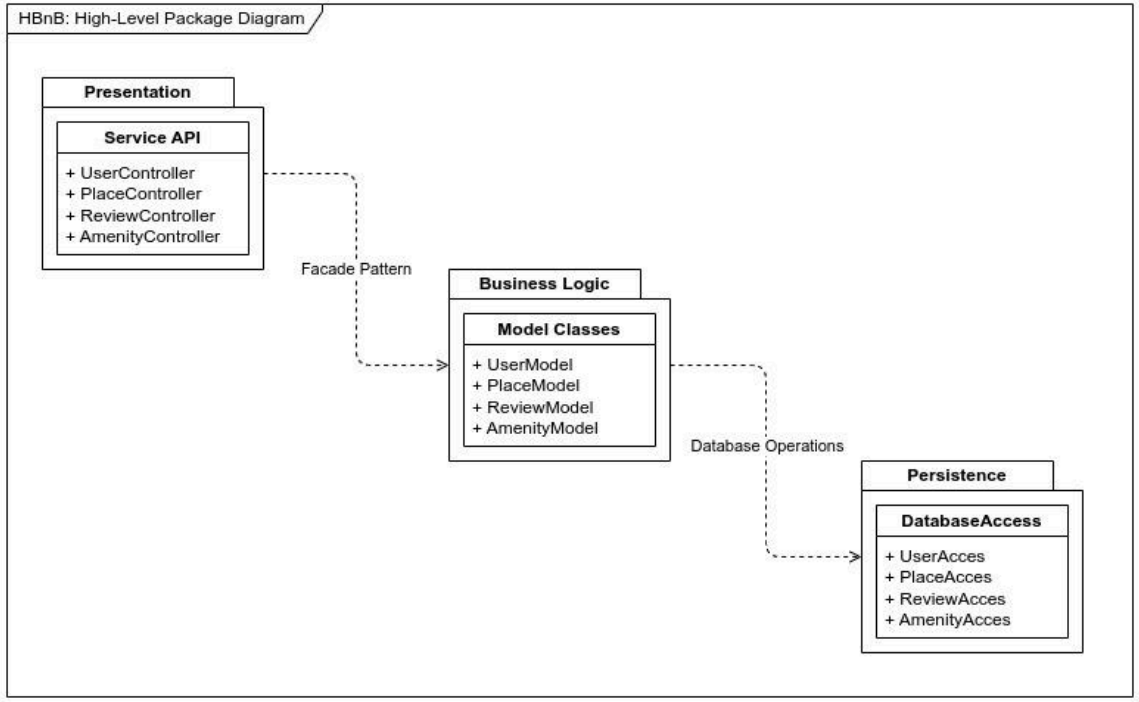
The documents is organized as follows:

- **High-Level Architecture:** Provides an overview of the application's layered architecture, demonstrating how the presentation, business logic, and persistence layers interact.*
- **Business Logic Layer:** Offers a detailed breakdown of the entities within the business logic layer; including their attributes, relationships and roles within the system.
- **API interaction flows:** Explains how the different layers communicate through the system via key API calls, using sequence diagrams to illustrate the flow of information.

Melvin Redondo–Tanis C24 Fréjus
Zakaria Aattache C24 Lille

High-level Architecture:

For this part, we will be using a High-Level Package Diagram to demonstrate how the different layers are organized. The application is structured around a three-layer architecture:



- **Presentation Layer:** This layer handles user interactions through an API. it receives requests from users and forwards them to the business logic layer. The controllers act as intermediaries that validate inputs and manages responses

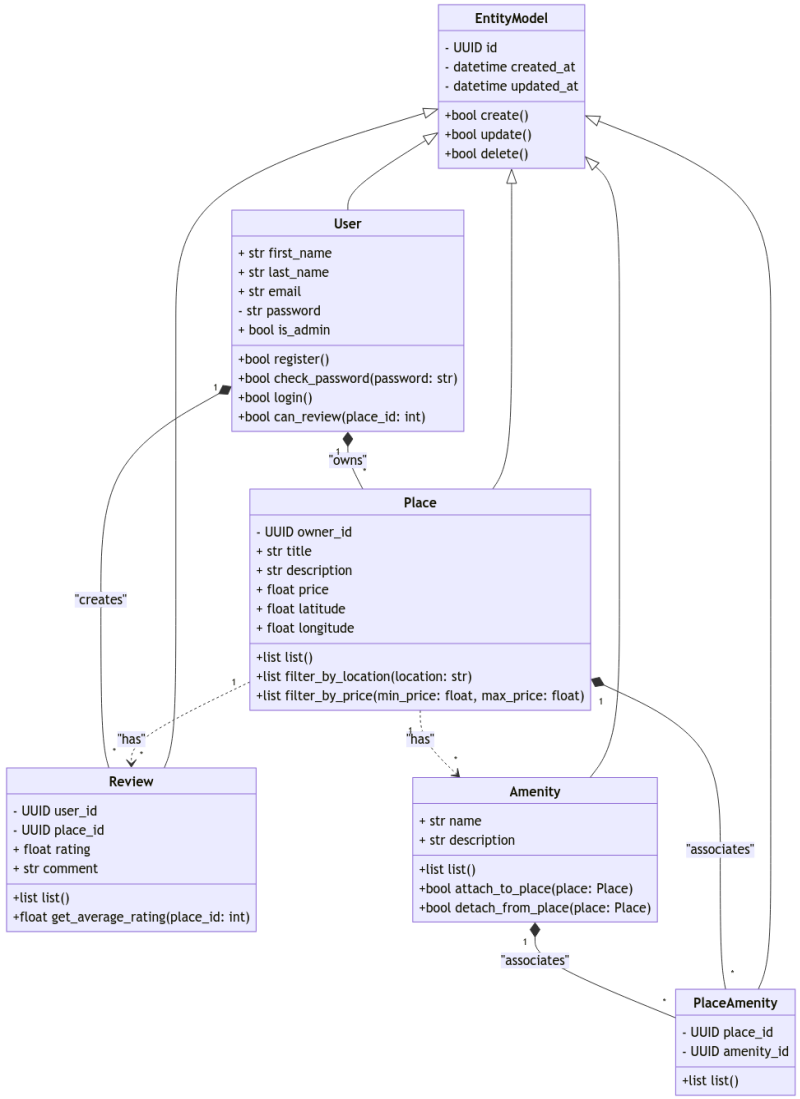
- **Business Logic Layer**: This is the core of the application, where all the business rules, validations and operations are handled. It includes services for managing users, places, reviews, and amenities. These services abstract the operations performed in the persistence layer and interact with the presentation layer through the Facade Pattern to simplify the communication.

- **Persistence Layer:** The persistence layer deals with database operations. It is responsible for storing and retrieving data for users, places, reviews, and amenities. The Database Operations act as a bridge for the business logic into Database.

Melvin Redondo–Tanis C24 Fréjus
Zakaria Aattache C24 Lille

Business Logic Layer:

The following diagram represents the Use, Place, Review, and Amenity entities. Each entity is a key part of the system, with attributes and methods that manage the business logic Operations.



- **Entity Model:** This base class includes common fields such as `id`, `created_date`, and `updated_at` to track the lifecycle of entities. It also includes generic methods such as `create()`, `update()`, and `delete()`.

- **User Entity:** Represents the users in the system, with attributes like `first_name`, `last_name`, `email`, and `password`. The `is_admin` boolean differentiates between regular users and administrators. Users can register, log in, and manage places and reviews. Key methods include:
  - `register()`: Registers a new user.
  - `check_password(password: str)`: Verifies the user's password.
  - `can_review(place_id: int)`: Determines if a user can leave a review for a specific place.

- **Place Entity**: Represents properties listed by users, with details like `title`, `description`, `price`, `latitude`, and `longitude`. Places are associated with users (owners) and can have multiple reviews and amenities. Key methods include:
  `list()`: Lists all places.
  `filter_by_location(location: str)`: Filters places by location.
  `filter_by_price(min_price: float, max_price: float)`: Filters places based on price range.

Melvin Redondo–Tanis C24 Fréjus
Zakaria Aattache C24 Lille

- **Review Entity**: Represents feedback provided by users for places. Each review includes a `rating`, `comment`, and is linked to both a user and a place. Key methods include:
  `list()`: Lists all reviews.
  `get_average_rating(place_id: int)`: Gets the average rating for a place based on all user reviews.

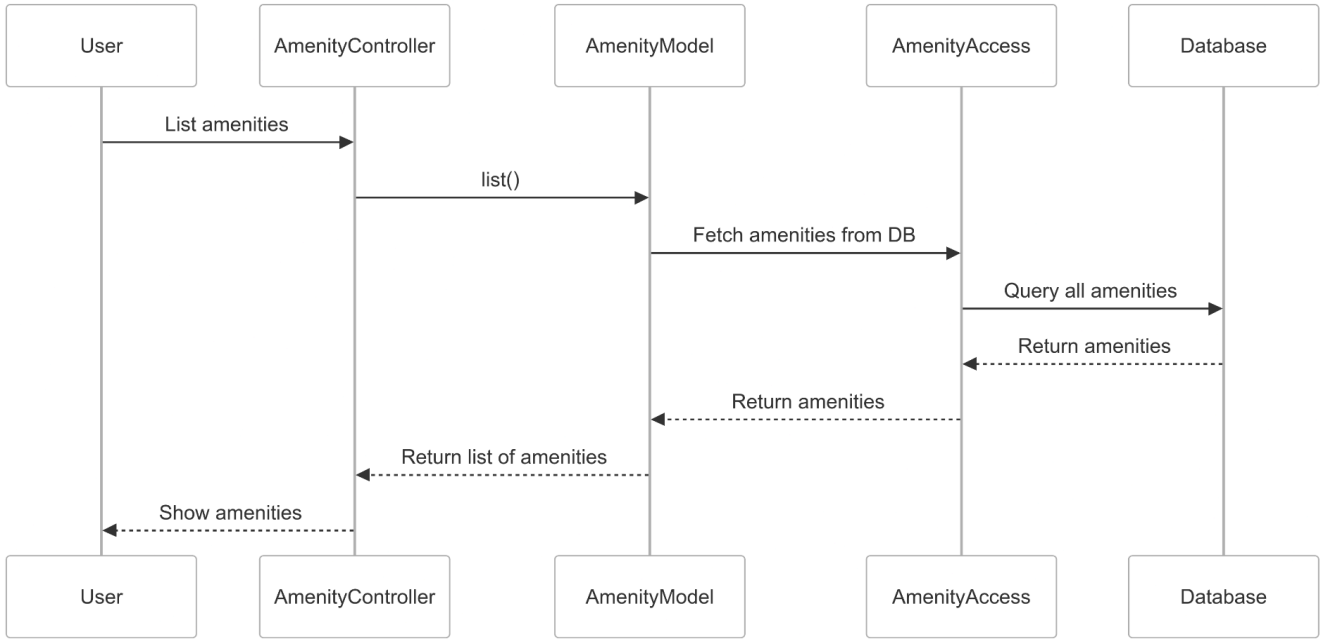
- **Amenity Entity**: Represents the amenities available at places. Each amenity has a `name` and `description`. Key methods include:
  `attach_to_place(place: Place)`: Attaches an amenity to a specific place.
  `detach_from_place(place: Place)`: Detaches an amenity from a place.

Melvin Redondo–Tanis C24 Fréjus
Zakaria Aattache C24 Lille

API Interaction Flow:

The API interaction flow outlines the flow of information between layers for four key API interaction using an sequence diagram.it shows how the application processes requests, ensures business rules are applied, and interacts with the persistence layer to retrieve or store data.

Amenity Sequence Diagram:



Melvin Redondo–Tanis C24 Fréjus
Zakaria Aattache C24 Lille

1. User Request :

The user initiates a request to list all available amenities by sending a command to the **AmenityController.**This request is captured in the form of `List amenities`.

2. AmenityController :

The **AmenityController** serves as an intermediary between the user and the business logic of the system. Upon receiving the user's request. The controller calls the `list()` method on the **AmenityModel** to fetch the list of amenities.

3. AmenityModel :

The **AmenityModel** is responsible for handling the business logic related to amenities. It process the request to fetch amenities by calling the **AmenityAcces** component using the `list()` method, which interacts with the database

4. AmenityAcces :

The **AmenityAccess** layer acts as the data access layer responsible for querying the **Database** to retrieve the requested data. Upon receiving the request from the model, it sends a `Query all amenities` command to the database. So the Database processes the query and returns the list of amenities to the **AmenityAccess** layer.

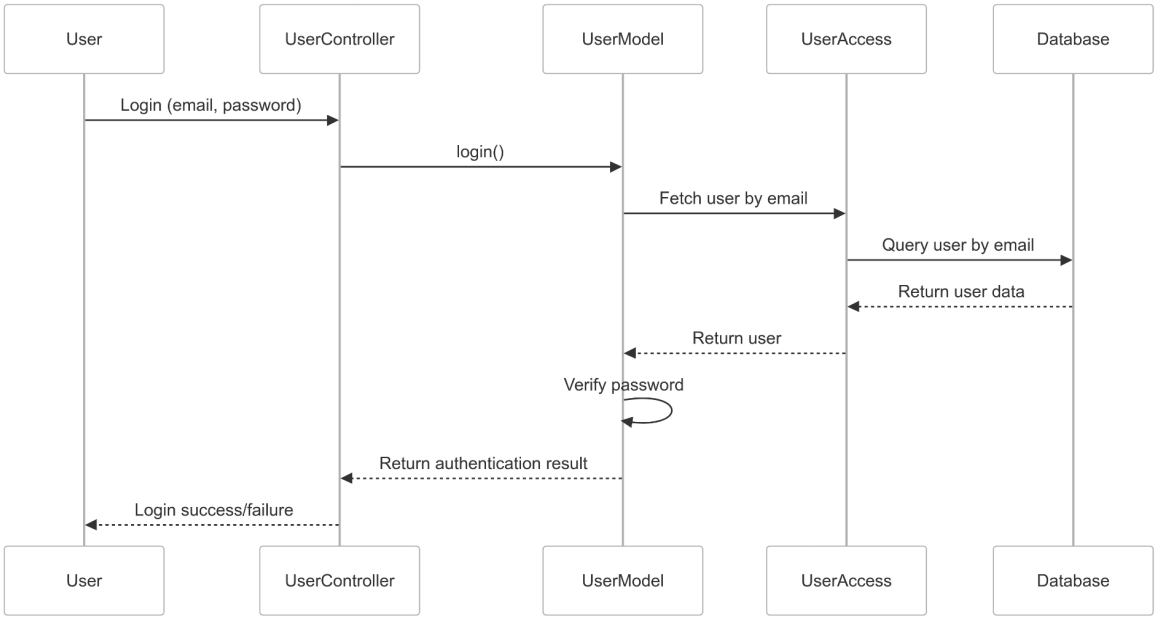
5. AmenityModel :

One the list of amenities is fetched from the Database, it is passed back through the system:

- The **AmenityAcces** returns the data to the **AmenityModel**
- The AmenityModel passes the list back to the **AmenityCrontroller**
- Finally, the **AmenityCrontroller** returns the list of amenities to the User, fulfilling the original request.

Melvin Redondo–Tanis C24 Fréjus
Zakaria Aattache C24 Lille

Login Sequence Diagram:



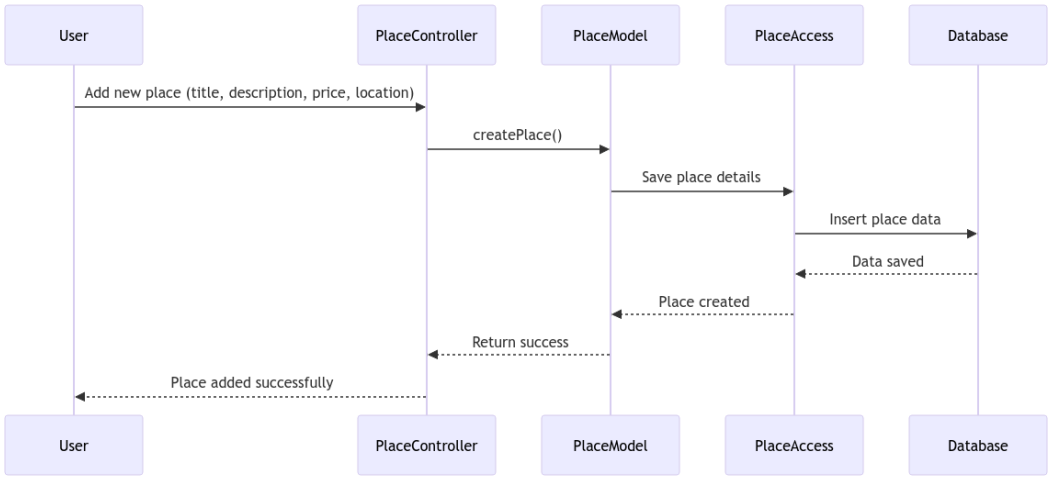
This diagram shows the login process for a user in the HBnB system. Here's a simple breakdown:

1. **Users** submit their email and password to the **UserController**.
2. **UserController** calls the **UserModel** to handle the login.
3. **UserModel** asks **UserAccess** to retrieve the user by email from the **Database**.
4. **Database** returns the user data (including the password hash) to **UserAccess**.
5. **UserModel** checks the provided password against the stored hash.
6. **UserModel** returns the authentication result (success or failure) to the **UserController**.
7. **UserController** sends the result back to the **User** (login success or failure).

This process ensures secure user authentication by validating the password with the stored data.

Melvin Redondo–Tanis C24 Fréjus
Zakaria Aattache C24 Lille

Place Sequence Diagram:



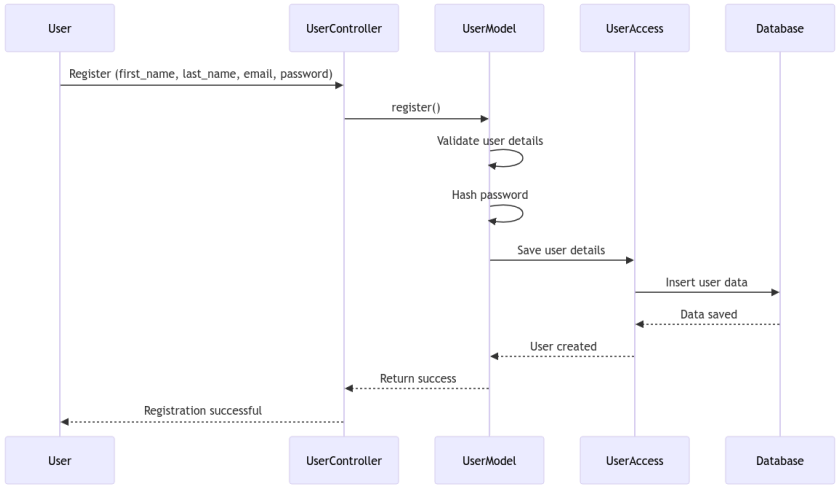
1. **User submits a request**: The user sends details for a new place (title, description, price, location) to the `PlaceController`.
2. **PlaceController calls PlaceModel**: The `PlaceController` invokes the `createPlace()` method in the `PlaceModel` to handle the request.
3. **PlaceModel calls PlaceAccess**: The `PlaceModel` requests `PlaceAccess` to save the place details.
4. **PlaceAccess interacts with the database**: `PlaceAccess` sends a request to the `Database` to insert the new place data.
5. **Database responds**: The `Database` confirms the data has been successfully saved and sends a response back to `PlaceAccess`.
6. **PlaceAccess informs PlaceModel**: Once the data is saved, `PlaceAccess` returns a confirmation to the `PlaceModel` that the place has been created.
7. **PlaceModel informs PlaceController**: The `PlaceModel` passes this success confirmation to the `PlaceController`.
8. **Response to the user**: Finally, the `PlaceController` informs the user that the new place has been added successfully.

This process describes how a new place is added in a structured way, showing how the different layers (controller, model, data access, database) interact to ensure data persistence.

Melvin Redondo–Tanis C24 Fréjus
Zakaria Aattache C24 Lille

Register Sequence Diagram:

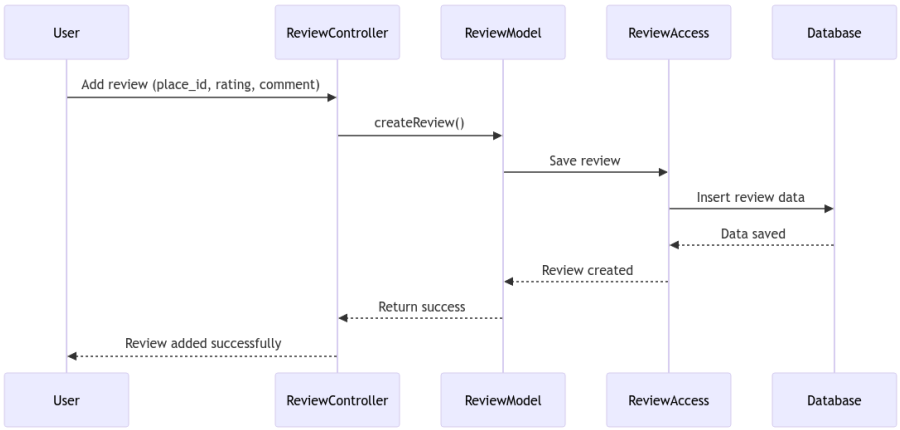
1. **User submits registration**: The user sends their details (first name, last name, email, and password) to the `UserController`.
2. **UserController calls UserModel**: The `UserController` invokes the `register()` method in the `UserModel` to process the registration.
3. **UserModel validates details**: The `UserModel` validates the user details (e.g., checking if the email is valid or if required fields are provided).
4. **Password hashing**: The `UserModel` hashes the password to securely store it in the database.
5. **UserModel requests UserAccess to save data**: The `UserModel` requests `UserAccess` to save the validated and hashed user details.
6. **UserAccess interacts with the database**: `UserAccess` sends a request to the `Database` to insert the new user data.
7. **Database confirms data save**: The `Database` confirms that the user data has been saved and sends a response back to `UserAccess`.
8. **UserAccess informs UserModel**: After saving the data, `UserAccess` returns confirmation to the `UserModel` that the user has been created.
9. **UserModel informs UserController**: The `UserModel` passes this success confirmation to the `UserController`.
10. **Response to the user**: Finally, the `UserController` informs the user that the registration was successful.



This sequence illustrates the flow for securely registering a new user and storing their data in the database, ensuring that sensitive information like passwords is hashed before storage.

Melvin Redondo–Tanis C24 Fréjus
Zakaria Aattache C24 Lille

Review Sequence Diagram:



1. **User submits a review**: The user sends details for a review, including `place_id`, rating, and comment, to the `ReviewController`.
2. **ReviewController calls ReviewModel**: The `ReviewController` invokes the `createReview()` method in the `ReviewModel` to handle the review creation.
3. **ReviewModel requests ReviewAccess to save data**: The `ReviewModel` asks `ReviewAccess` to save the review details.
4. **ReviewAccess interacts with the database**: `ReviewAccess` sends a request to the `Database` to insert the review data.
5. **Database confirms data save**: The `Database` confirms the review data has been saved successfully and sends a response back to `ReviewAccess`.
6. **ReviewAccess informs ReviewModel**: Once the review data is saved, `ReviewAccess` returns confirmation to the `ReviewModel` that the review has been created.
7. **ReviewModel informs ReviewController**: The `ReviewModel` passes this success confirmation to the `ReviewController`.
8. **Response to the user**: Finally, the `ReviewController` informs the user that the review has been added successfully.

This sequence ensures the secure handling and storage of the user's review for a specific place.