HBnB project - Holberton Trimester 2

Introduction

This document serves as the technical documentation for the HBnB project, developed as part of the Holberton School curriculum. The project consists of building a web-based application inspired by AirBnB, allowing users to publish, book, and manage rental properties.

The purpose of this documentation is to provide a comprehensive and structured view of the system's software architecture. It compiles the key design diagrams - high-level package diagram, class diagram (as part of the business logic layer), and sequence diagrams - along with explanatory notes. Each section outlines the technical choices made, the responsibilities of each component, and how they interact within the system.

This document is intended to guide the implementation phases of the project. It acts as central reference to ensure development consistency, facilitate collaboration among contributors, and support future maintenance and enhancements.

Valentin DUMONT & Quentin LATASTE for Holberton School Dijon - May 2025

Summary

Valentin DUMONT & Quentin LATASTE for Holberton School Dijon - May 2025

Technical Documentation for high-level package diagram :

This diagram provides a high-level overview of the three-layer architecture implemented in our HBnB project for Holberton School.
It highlights the main components of each layer and their respective responsibilities.

The architecture is structured around the following three layers:

1. Presentation Layer

This package groups all client-facing services, including:

- The user interface
- API endpoints that facilitate communication between the client and the server

Responsibilities:

- Handle HTTP requests
- Validate input data
- Call business logic functions
- Return structured HTTP responses (typically in JSON format)

2. Business Logic Layer

This package defines the core functional rules of the application.

Responsibilities:

- Define and manipulate domain models (User, Place, etc.)
- Enforce business rules (e.g., a Place must have an associated host)
- Coordinate data flow between the presentation and persistence layers

3. Persistence Layer

This package manages the interaction with the data storage system (e.g., database or file-based storage).

Responsibilities:

- Abstract and encapsulate data access logic
- Read and write data
- Hide the details of the storage implementation
- Provide classes to persist and retrieve model instances

Use of the Facade Pattern

In this project, we implemented a Facade pattern within the business logic layer.

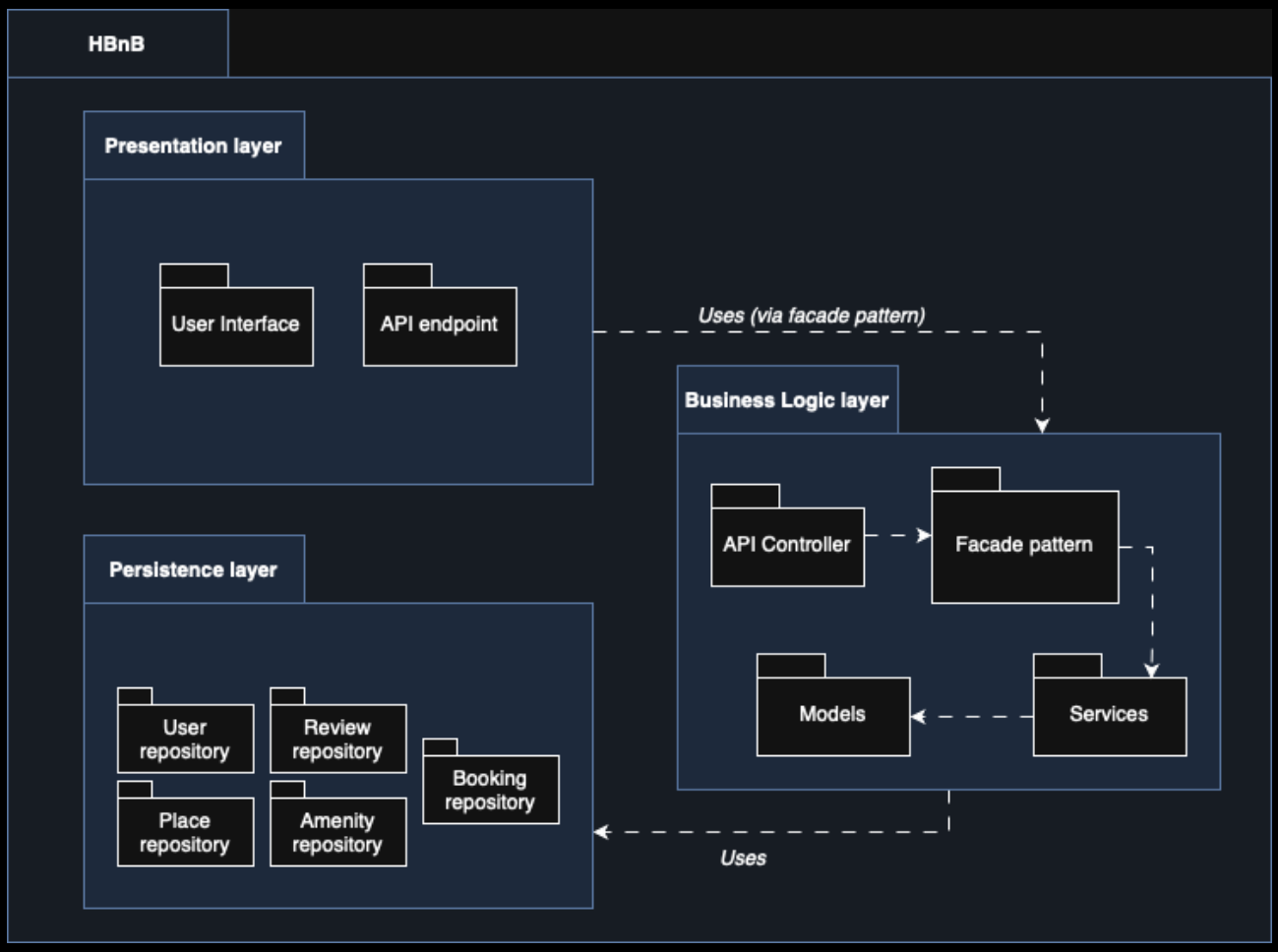
Purpose:
The facade serves as a unified interface for a group of classes or subsystems. It encapsulates the complexity of multiple interactions behind a single method call.

Valentin DUMONT & Quentin LATASTE for Holberton School Dijon - May 2025

Example Use Case:
When an API endpoint (e.g., POST /places) is called, it invokes a facade method such as create_place(). This method:

- Validates the input data
- Instantiates the necessary model objects (Place, User, etc.)
- Persists them in the database
- Returns a well-structured result to the presentation layer

This pattern allows the API layer to remain clean and simple while centralizing complex logic within the business layer.



Technical documentation for classes diagram

Overview:

HBnB is a web application inspired by Airbnb that allows users to publish, reserve, and manage rental places. The system is built around a set of classes that model the key entities of the application, their interactions, and the possible actions.

All classes share a common set of attributes:

- id: unique identifier (UUID v4)
- created_at: date and time of creation
- updated_at: date and time of last modification

Entity Descriptions:

- Facade (design pattern)

The facade is a structural design pattern used to provide a simplified, unified interface to the complex subsystem composed of multiple classes (User, Place, Photo, Review, Amenity).

Role:

- Acts as an entry point to key system features (e.g., account management, place publishing, review handling)
- Hides the internal complexity of the application's class interactions
- Facilitates easier use of the system by external modules or API layers (e.g., front-end or REST endpoints)

Responsibilities:

- Orchestrates operations such as:
- Create/update/delete a user or place
- Upload and retrieve photos
- Post or moderate reviews
- Attach amenities to places
- Ensures appropriate access control logic (based on user roles) when invoking methods from underlying entities

Example Use Case:
When a client requests to "update a place and add two new photos", the Facade handles:

- Authorization (check if the user owns the place)
- Place update
- Photo validation and storage
- Linking photos to the place

This pattern contributes to separation of concerns and makes the system easier to test, extend, and maintain.

- User (Abstract)

Represents a generic user of the system.
This is an abstract base class, inherited by both Client and Admin.

Role:
Provides the basic interface for user interactions such as profile management and place handling.

Valentin DUMONT & Quentin LATASTE for Holberton School Dijon - May 2025

- **Client** (inherits from User)

Represents a standard user of the platform.

Capabilities:
- Manage their own profile
- Add, update, and delete their own places
- Manage their profile picture
- Post reviews on places they visit
- Can book a place

- **Admin** (inherits from User)

Represents a privileged user with elevated rights.

Capabilities:
- Access and manage all user profiles and places
- Moderate or delete any content (except content from other admins)
- Shares all capabilities of a Client

- **Place**

Represents a rental property posted by a user.

Attributes:
- title, description, price, latitude, longitude

Relationships:
- Owned by a User
- Can be linked to multiple Photo, Review, and Amenity objects

- **Photo**

Handles images associated with users (profile picture) and places (galleries).

Role:
- Stores and provides access to the photo URL hosted on an external server
- Used only by User and Place entities

- **Review**

Represents a rating and comment posted by a user on a specific place.

Relationships:
- Associated with a User (author) and a Place (target)

Capabilities:
- Can be created, modified, or deleted
- Can be listed by user or by place

- **Amenity**

Represents the amenities or facilities available in a place (e.g., Wi-Fi, kitchen, pool).
Attributes:
- name, description

Role:
- Can be managed independently
- Can be associated with one or more Place entities

- **Booking**

Represents the booking possibilities for a place and allow users to create a reservation.

Role:

- Manage the place's calendar
- Allow users to place reservations
- Check the reservation status

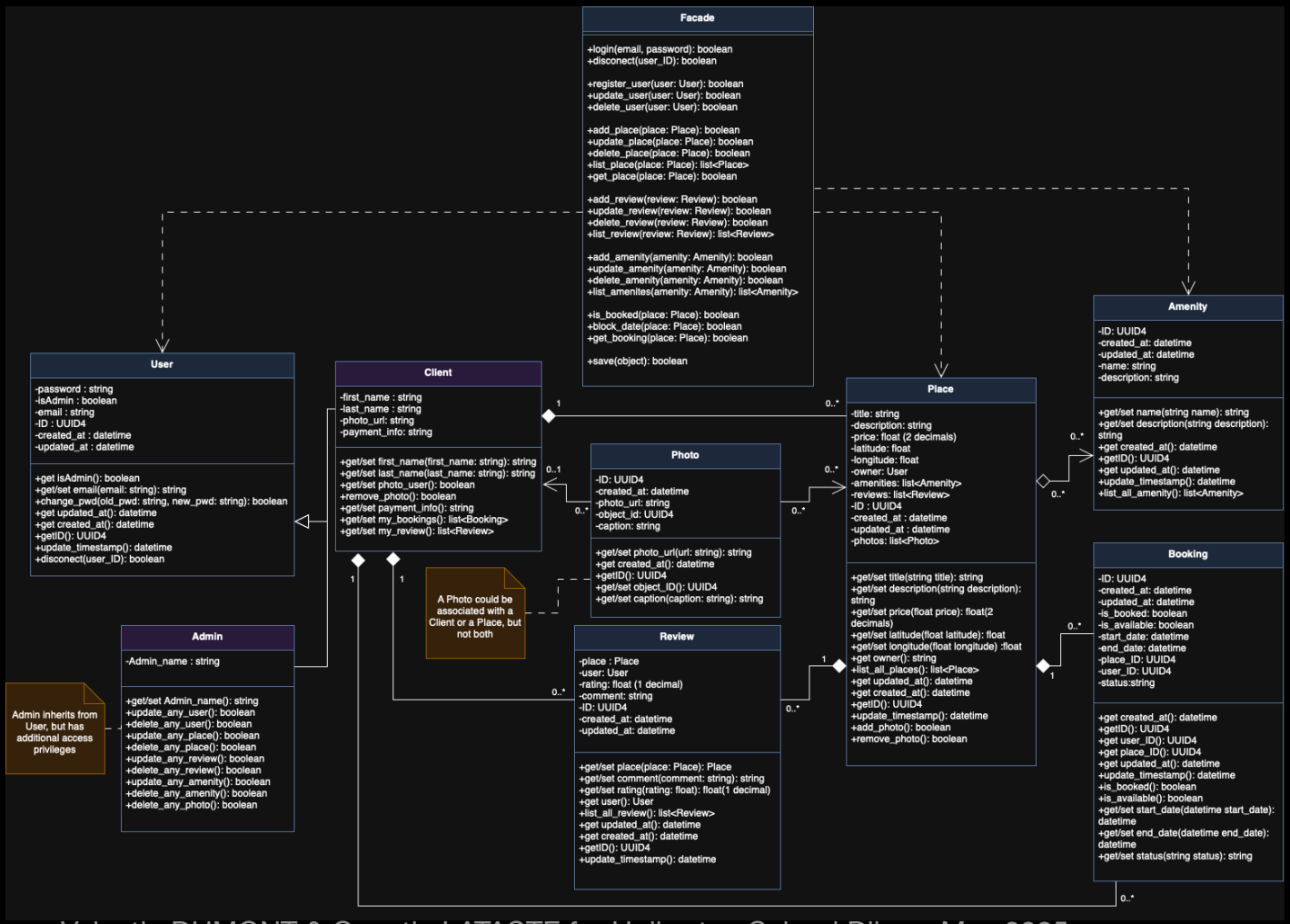
Main Relationships:

- User ⟷ Place: One-to-many (a user can own multiple places or none)
- User ⟷ Photo: Zero-or-one-to-many (for profile picture)
- Place ⟷ Photo: Many-to-many (gallery)
- Place ⟷ Review: One-to-many
- User ⟷ Review: One-to-many
- Place ⟷ Amenity: Many-to-many
- Place ⟷ Booking: One-to-many

Persistence & Access Control:

All entities are stored with temporal tracking via created_at and updated_at attributes to facilitate versioning and auditing.

The model implements a clear separation of user roles (Client vs Admin) to ensure proper rights and access control.
The inheritance structure (User → Client / Admin) is central to managing permissions and system behavior.



Sequence diagram technical documentation

Here are the four sequence diagrams for the HBnB application.

1. User Registration Workflow

The first diagram illustrates the process of creating a new user. The user initiates a pop-up window and fills out a registration form. The submitted data is sent to the API controller, which communicates with the facade. The facade then creates a new instance of the User entity and stores the data in the database via the ClientRepository.
If the operation is successful, a boolean true value is returned, and the API responds with an HTTP status code (201 Created), which is then relayed to the frontend to display a success message. In case of failure, an appropriate error code is returned (e.g., 400, 429, or 500).

2. Place Creation Workflow

The second diagram describes the process for creating a new place. The flow is similar to user registration: the frontend interacts with the API, which calls the facade, and the data is stored in the database. The key difference is that this operation uses the PlaceRepository instead of the ClientRepository.

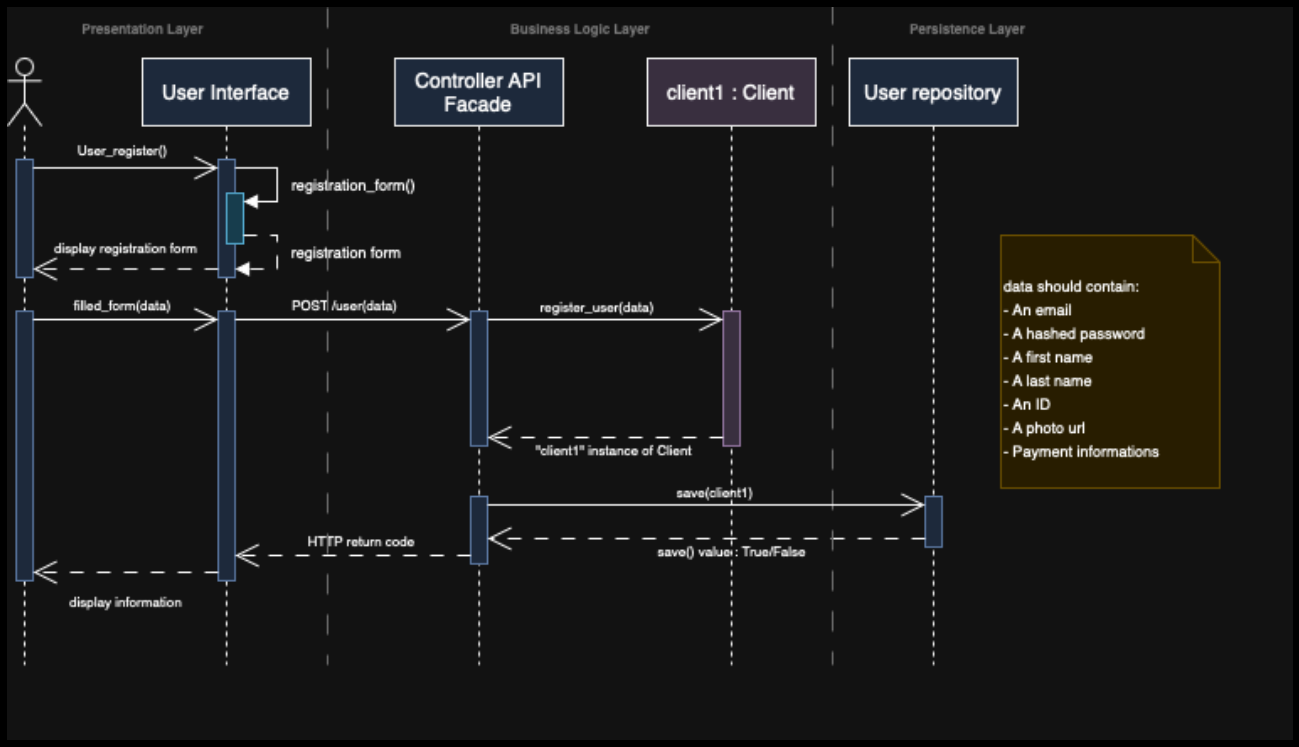
3. Review Submission Workflow

The third diagram shows how a user can submit a review for a place. First, the system verifies that the user is logged in and has previously completed a stay at the selected location. If these conditions are met, the user can submit a review by entering a comment and a rating.
The frontend sends this data to the API controller, which routes it to the facade. The facade creates a new Review object and saves it via the ReviewRepository.
If the operation is successful, the backend returns a confirmation and the frontend displays a success message. Otherwise, it returns an error code such as 400 (Bad Request) or 500 (Internal Server Error), depending on the issue.

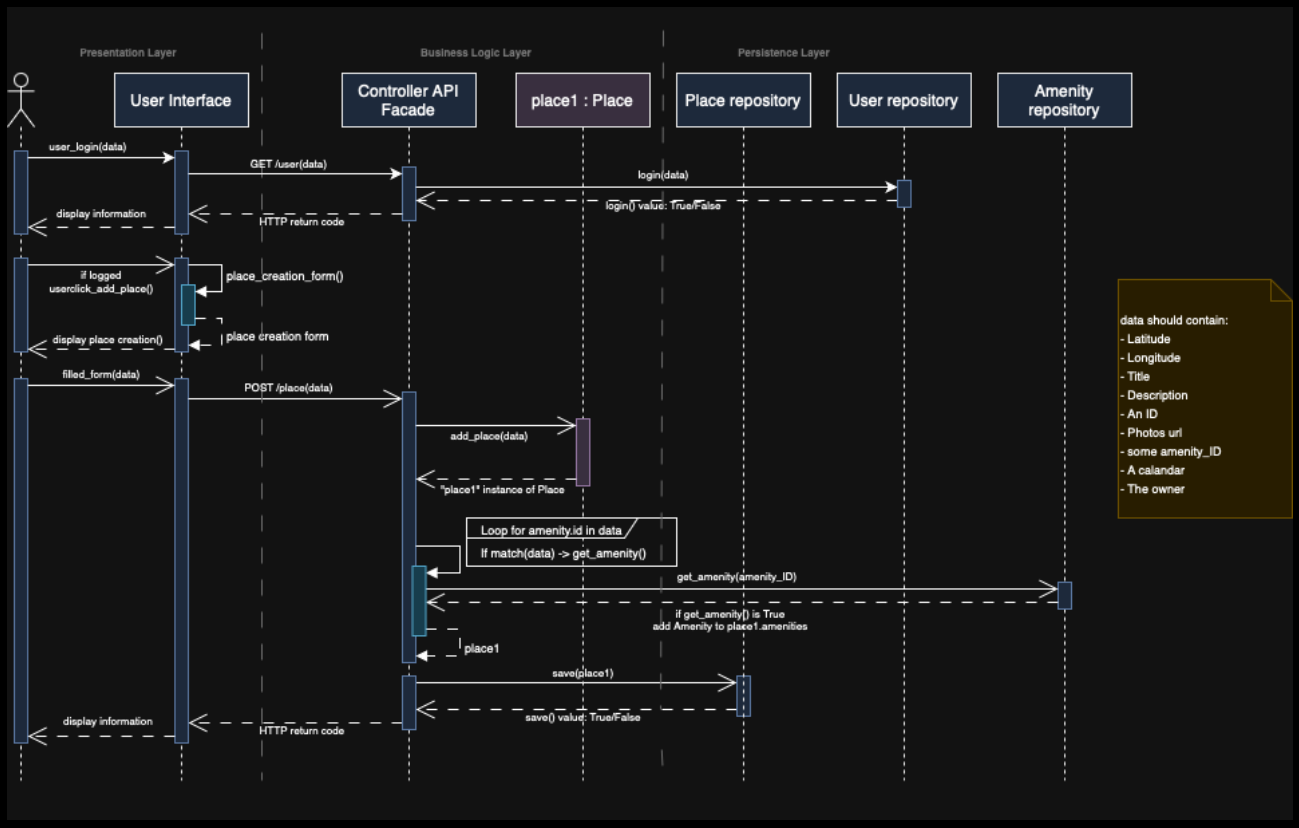
4. Fetching a list of places

The fourth diagram outlines how users can search for available places using specific criteria (e.g., location, dates, number of guests). The frontend sends a search request to the API, which forwards it to the facade. The facade then queries the PlaceRepository to retrieve matching results from the database.
The list of available places is then returned to the API and sent back to the frontend, where it’s displayed to the user. If no places match the criteria, the system can return an empty result or an informational message.

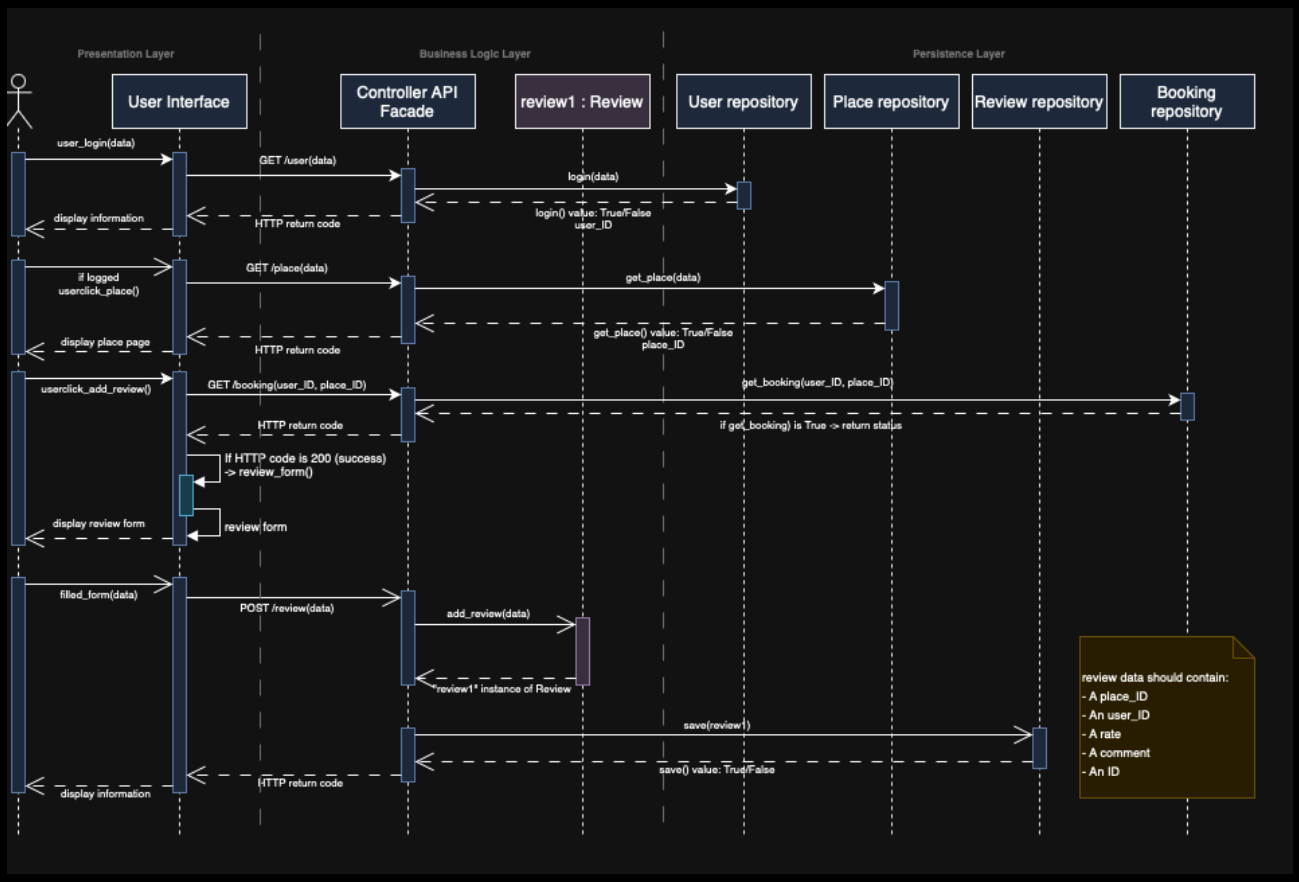
Valentin DUMONT & Quentin LATASTE for Holberton School Dijon - May 2025



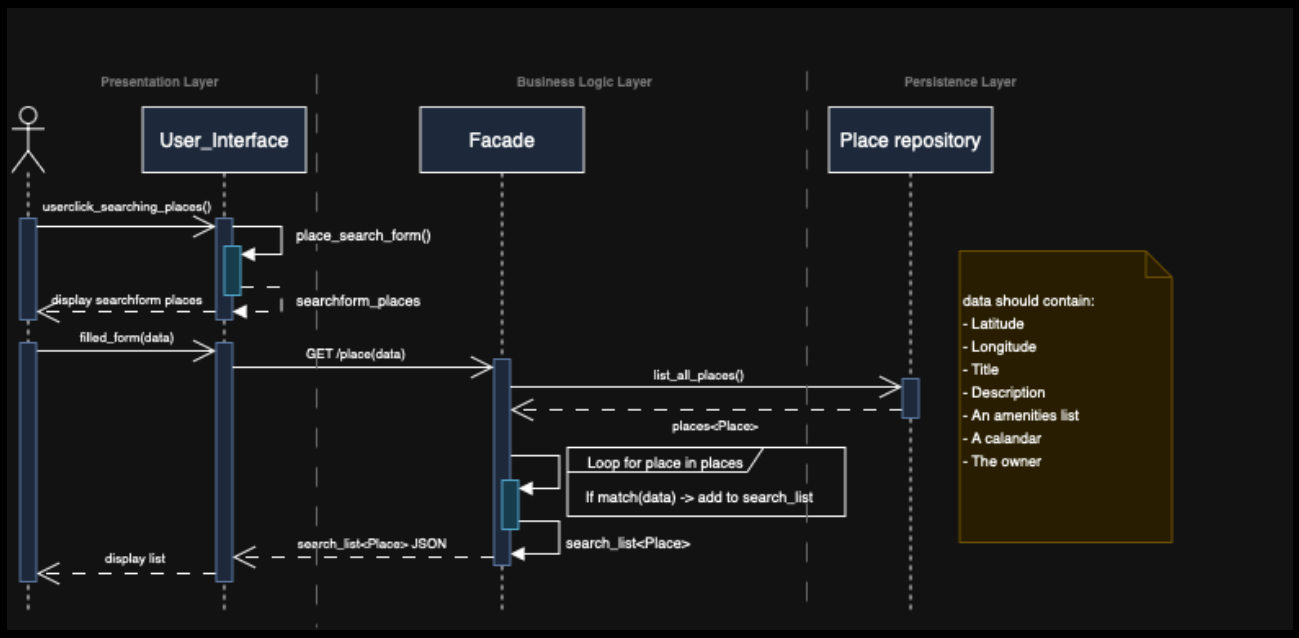
User registration sequence diagram



Place creation sequence diagram



Review creation sequence diagram



Amenities list fetching sequence diagram

Conclusion

The HBnB project is designed as a full-featured web application, leveraging a clear separation of concerns through a layered architecture. From the presentation interface to the persistence layer, each component was structured to ensure scalability, maintainability, and ease of collaboration.

This documentation aimed to provide a comprehensive overview of the system's design, covering the core entities, their interactions, and the workflows implemented via sequence diagrams. Through the use of design patterns such as Facade and a modular approach, the project demonstrates the application of software engineering principles in a real-world context.

As development continues or new features are introduced, this document should serve as a reliable reference for future contributors, helping maintain technical consistency and supporting long-term project evolution.