Readme.MD

The purpose of this HBNB project is to lay the groundwork for the future development of an application similar to the Airbnb platform.

In this readme, we will be presenting various diagrams that represent its functionality.

These are:

The High-Level Package Diagram:

It provides a preview of how different components of the application are organised and how they interact with one another.

The Class Diagram:

It offers a clear and detailed image of the core of the Business Logic Layer, focusing on key entities: Base, User, Place, Review, Amenities.

The Sequence Diagram:

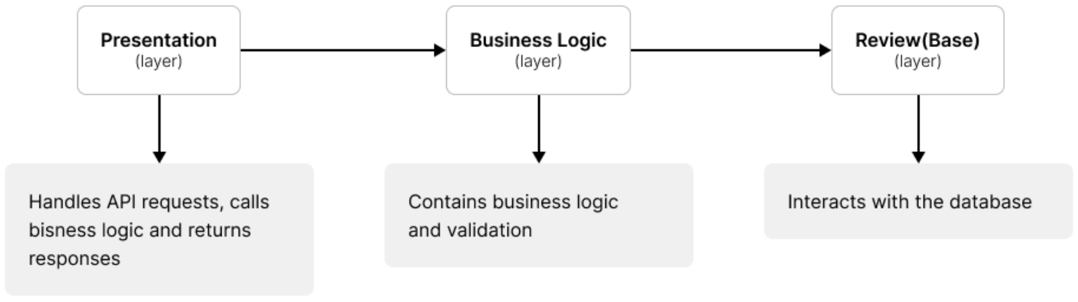
It helps to visualise how different components of the system interact across the various layers, showing step by step the process by which API requests are handled.

The focus of this project is primarily based on the adoption of new tools and concepts, as well as the planning of the conceptual structure of what will eventually be a platform similar to Airbnb.

We began by learning completely new concepts that will be utilised later, such as the Facade pattern, the Persistence Layer, APIs, endpoints, etc. We also learned how to structure a project using tools like Figma, LucidApp, and Notion.

Subsequently, we started conceptualising the structure, defining how we believe the different layers interact with one another, in order to begin outlining the various diagrams that comprise this project. These were created in LucidApp and then transferred to Figma for licensing reasons.

High-level package diagram


Presentation
(layer)
Business Logic
(layer)
Review(Base)
(layer)
Handles API requests, calls bisness logic and returns responses
Contains business logic and validation
Interacts with the database

This **High-Level Package Diagram** represents a layered architecture for our project based on Airbnb.

The Presentation Layer is the user interface or the API that handles external requests. It is responsible for receiving requests, validating them, and forwarding them to the Business Logic Layer. It also receives responses from the Business Logic Layer and returns them in an appropriate format (JSON, HTML, etc.).

The Business Logic Layer contains the core logic of the application, such as business rules and validations. It processes the requests received from the Presentation Layer and communicates with the Data Layer to obtain or update information.

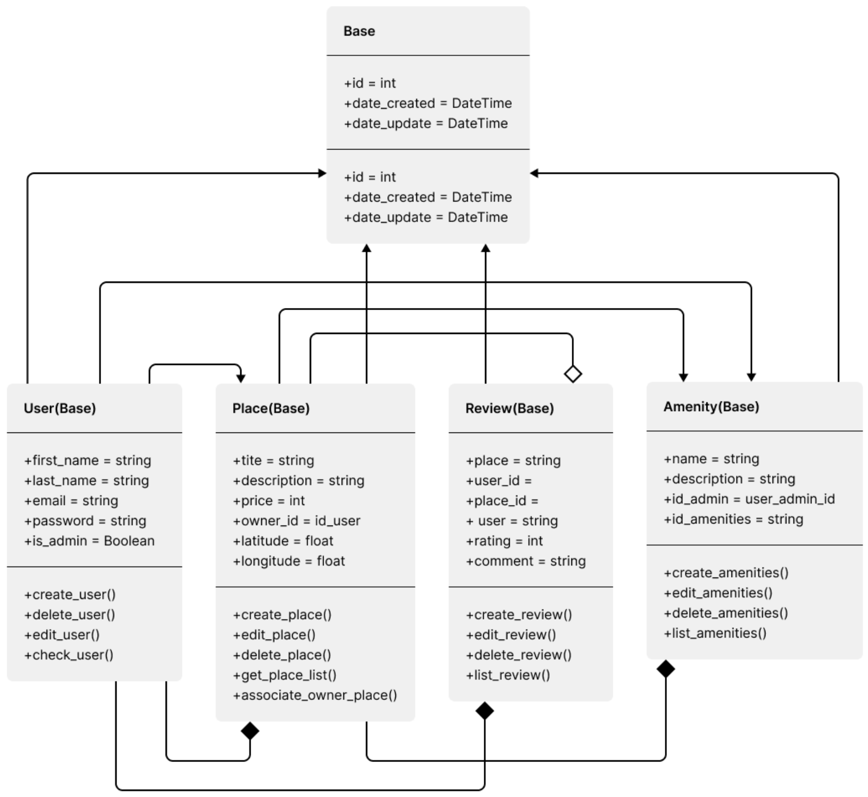
The Review (Base) Layer is the persistence layer, responsible for interacting with the database. It performs operations such as creating, reading, updating, and deleting (CRUD) records and may include ORM (Object-Relational Mapping) to facilitate data manipulation.

Typical Workflow:

1. A user makes a request (for example, to view reviews of an accommodation).
2. The **Presentation Layer** handles the request and forwards it to the **Business Logic Layer**.
3. The **Business Logic Layer** validates the request, applies business rules, and queries the **Review Layer** to obtain the data.
4. The **Review Layer** retrieves the data from the database and sends it to the **Business Logic Layer**.
5. The **Business Logic Layer** processes the data and sends it to the **Presentation Layer**.
6. The **Presentation Layer** returns the response to the user.

A positive aspect of this type of modular architecture is that it helps to keep the code organised, facilitates scalability, and allows for the reuse of components

Class Diagram



This class diagram was designed with the aim of achieving the right and necessary abstraction of classes, in order to avoid unnecessary code repetition and the creation of superfluous classes. This will allow us to reconcile the concepts learned based on the patterns suggested so far, with the goal of achieving scalability and code readability, which are among the main objectives of this project. We believe that without a good class diagram, it would be unfeasible for the group to begin coding this project. These diagrams lay the foundation for our project.

In this diagram, five entities are observed.

A **Base** class with attributes and methods that will be inherited by all classes, ensuring that every time an object of another class is instantiated, it has a unique ID, a creation date, and a modification date. Additionally, there are four more classes: **User ()**, **Place()**, **Review()**, and **Amenities()**.

User (): This represents the user, who has a unique ID and also a boolean that verifies whether they are an admin (able to delete users) or not. It follows the CRUD pattern. It interacts directly with all the classes, and most depend on it for their existence. A **User ()** can create and modify **Places()**, add **Amenities()**, and make **Reviews()**. If the **User ()** is an admin, they can also delete them.

Place() is a class created by a **User ()** where the desired properties will be instantiated. It must be associated with the ID number of the **User ()**, making that **User ()** the owner. Each instance of **Place()** has its own ID.

Place() depends on **User ()** but not on **Review()** or **Amenities()**; both classes may or may not be part of **Place()**.

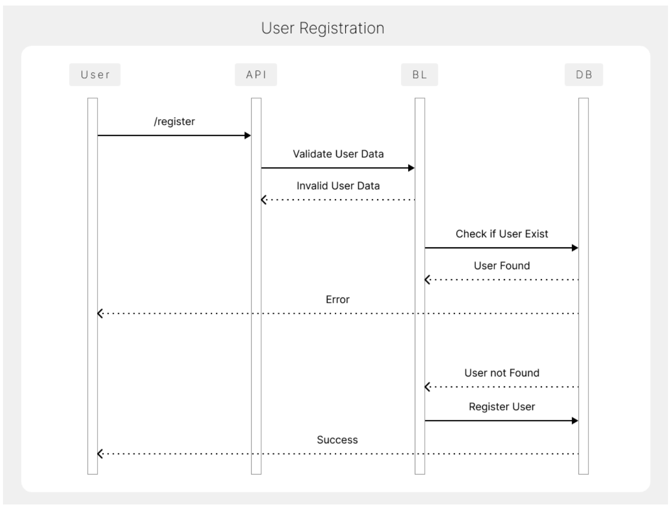
Review() is a class that directly depends on **User ()** and **Place()**. A **User ()** can create a review as long as a **Place()** exists.

Review() stores the ID of the **User ()** and the ID of the **Place()**, along with the comment and the rating. Additionally, the **Review()** also has its own ID.

Amenities() are extra features that a **Place()** may or may not have. These are added or created by the **User ()**.

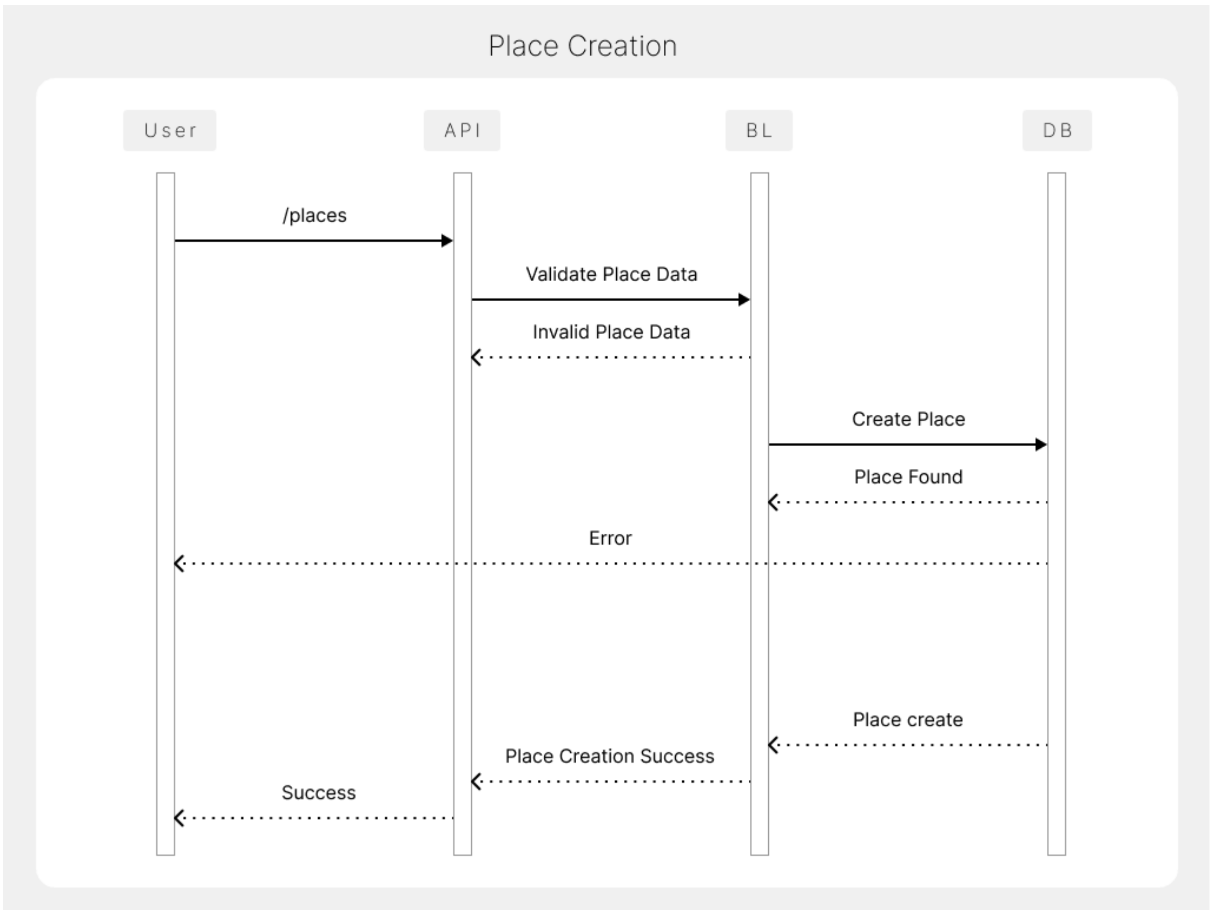
Amenities() stores a list of amenities that the owner **User ()** can add and modify for their property, but only an admin **User ()** can modify **Amenities()**.

Sequence Diagram

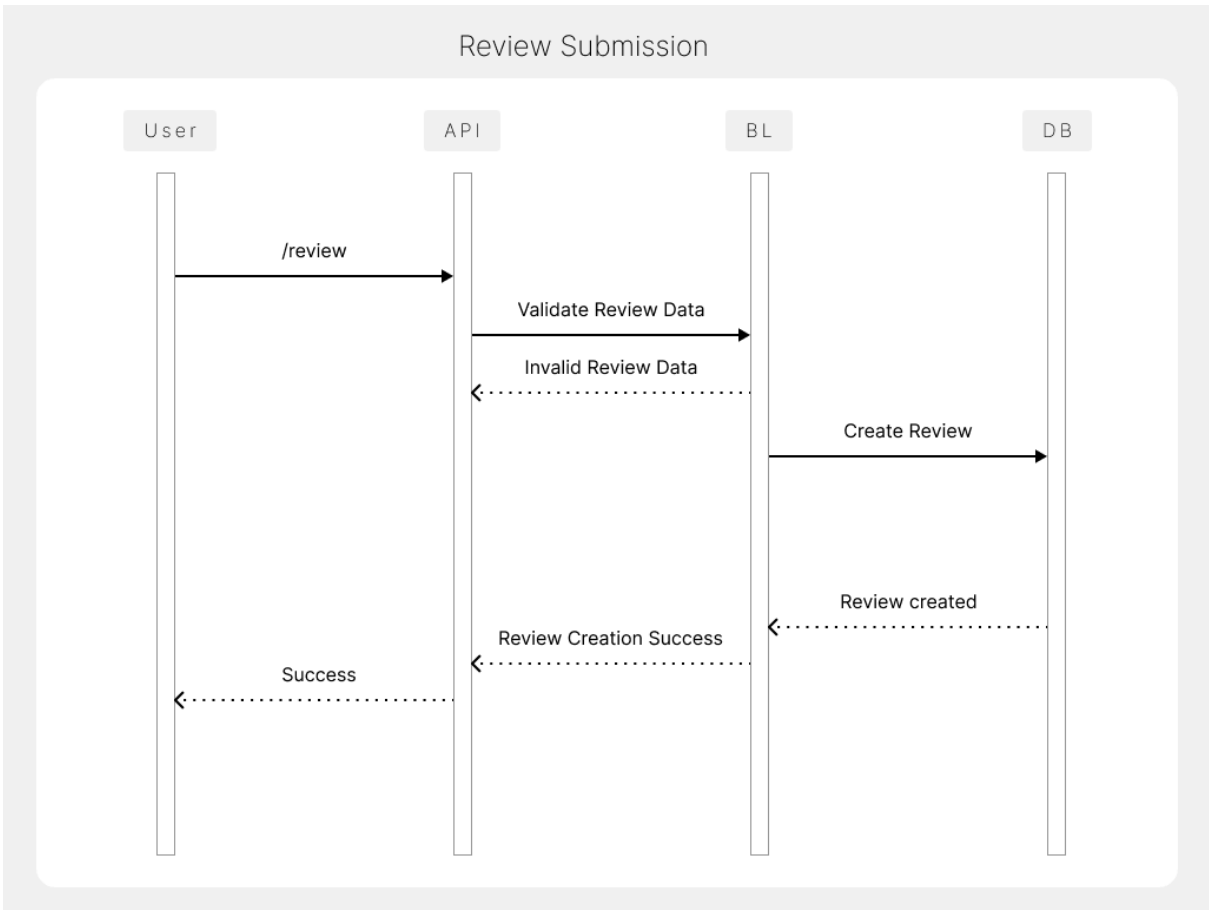


This example uses a scenario in which a user wishes to register. In this case, the User interacts with the visual layer, generating an input for registration through the API. This input is sent to the **Business Logic Layer**, which performs checks to determine if, for example, the email is valid based on its format or if the password meets the required criteria. If these conditions are not met, an error is returned.

If the requirements are satisfied, the **Business Logic Layer** queries the **Database** to verify whether the user is already registered. If the user is found, an error is returned; if not, the user is registered in the **Database**.



In this example, we consider a scenario in which an **Owner** creates a **Place**. Similarly, the system verifies that the details of the place are valid. If the details are invalid, an error is returned. If the data is validated, the system proceeds to check the **Database** to see if the **Place** already exists. If it does exist, an error is returned; if it does not, the **Place** is created, and a success message is returned.

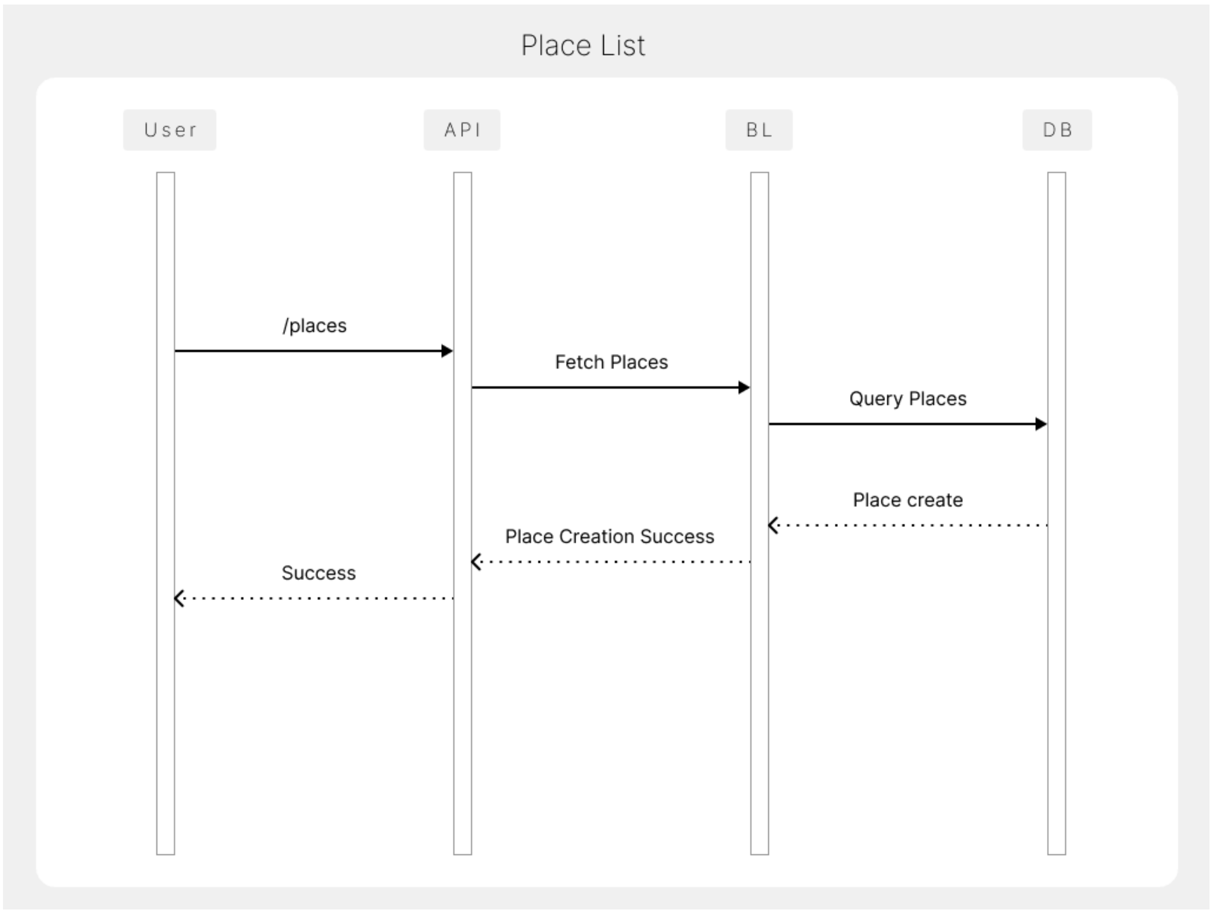


Este diagrama representa la interacción de una API en un sistema de **tres capas** cuando un usuario envía una **Review**.

This diagram represents the interaction of an API in a **three-layer** system when a user submits a **Review**.

The user sends a review through the system's interface, and the API receives the request at the endpoint `/review`, forwarding it to the Business Logic Layer (BL) for validation. In the BL, the process `Validate Review Data` is executed, meaning that the business logic checks whether the review contains correct data. If the data is incorrect, a message of `Invalid Review Data` is returned, notifying the user of the error and halting the process.

If the data is valid, the process `Create Review` is executed, which involves saving the review in the database. The database stores the review and confirms the operation with `Review created`. The BL processes the response and sends it back to the API, which then informs the **User ** with `Review Creation Success`, indicating that the review was created successfully.



In this final example, we observe a scenario in which the **User ** requests a list of **Places**. The API receives the request at the endpoint `/places` and is responsible for redirecting the request to the Business Logic Layer.

The Business Logic Layer executes the action `Fetch Places` and then sends a query (`Query Places`) to the database to retrieve the information.

The database processes the query and returns the requested data. The Business Logic Layer receives the data and organises it if necessary. It then sends the data back to the API, which responds to the User with the list of places obtained from the database.