Project HBnB - UML

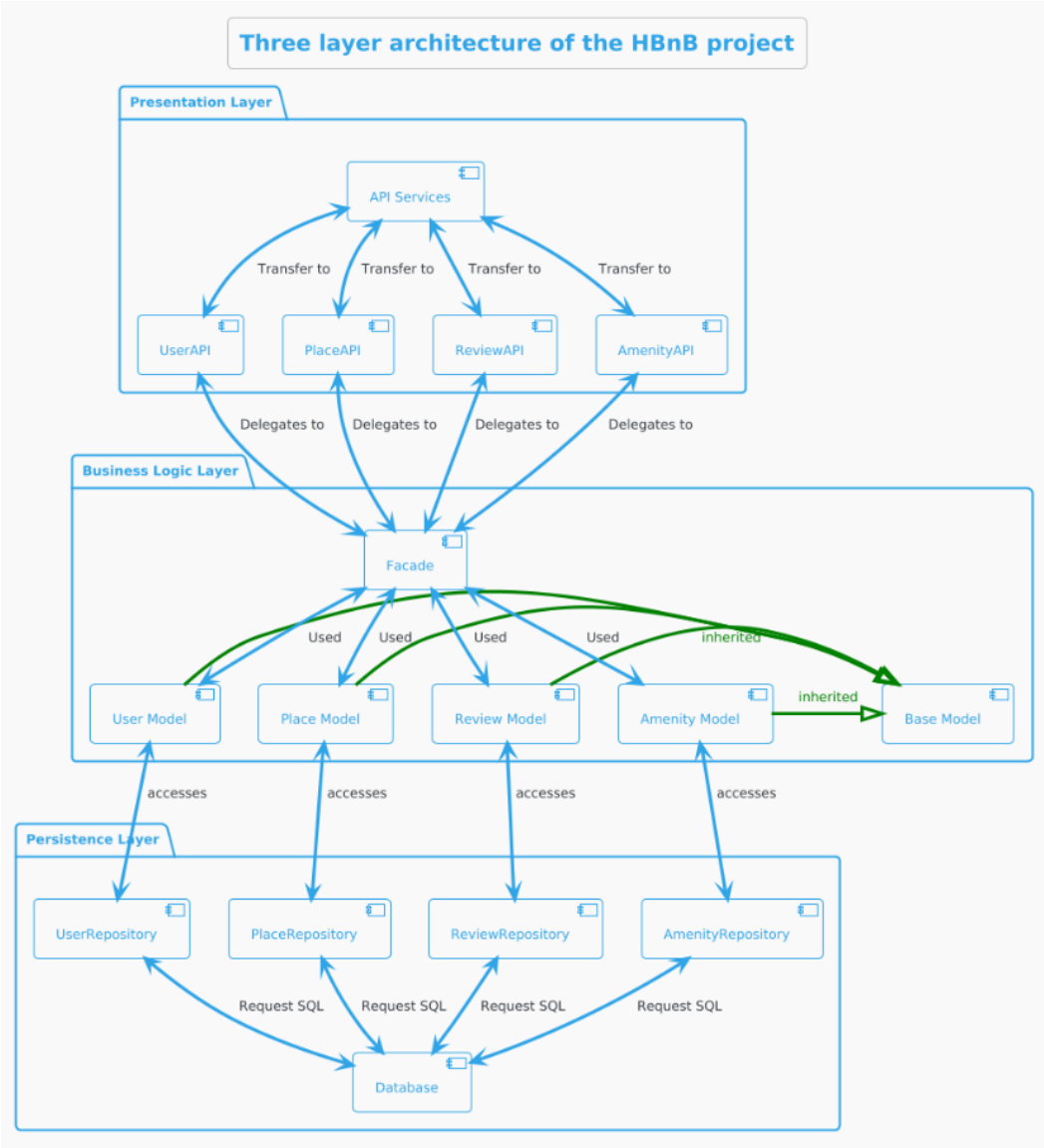
1. Introduction to the project and purpose of the three diagrams

This document constitutes the complete technical documentation for the HBnB project, developed as part of a group project. The project aims to replicate the Airbnb website.

This document presents three UML diagrams designed to provide a clear overview of the behaviour of packages, the structure of their classes, and the sequence of interactions between user actions on the website and the responses returned based on the data received.

2. High-level diagram - 3-layer architecture

The high-level package diagram illustrates the three-layer architecture for the HBnB project and the communication between these layers. This diagram highlights the organisation between the layers and the flow of data, in particular via the Pattern facade. The system is divided into three distinct layers.



2.1 Presentation layer

The presentation layer is the first layer in our three-layer architecture. It acts as the interface between the user and the system. The Services API is the main entry point to the web application. It is made up of several modules, each dedicated to a specific business resource: UserAPI for user management, PlaceAPI for accommodation, ReviewAPI for reviews, and AmenityAPI for accommodation-related equipment.

In short, it allows you to :
- Receive user requests via an API
- Verify & Validate inputs (check that the data provided by the user is correctly formed).
- Transmit the data to the business logic layer for processing.
- Return responses to the user after data processing.

2.2 Business logic layer

The second layer of the architecture contains the business logic. To structure access to the business models (UserModel, PlaceModel, ReviewModel, AmenityModel), the architecture is based on a facade.

This front-end acts as a single point of entry to the business layer: it centralises and organises calls to the various models. It hides internal implementation details, exposes a consistent interface and thus facilitates communication with the presentation layer (API). The front-end fulfils a co-ordination role: it directs each business request to the right model, applies the necessary checks, and then delegates data persistence to the lower layer.

To sum up, this makes it possible to :
- Standardize interactions with business models;
- Simplify the code on the API side, which does not need to know the internal structure of the models;
- Isolate business rules in a controlled space, facilitating maintenance, readability and unit testing.

2.3 Persistence layer

The persistence layer is the lowest layer in the three-layer architecture. It is responsible for the physical and logical management of data. It interacts directly with the storage systems, via UserRepository, PlaceRepository, etc. and the databases, to ensure that business information is retained, retrieved, modified and deleted.

The persistence layer is never invoked directly by the presentation layer. It is invoked exclusively by the business layer, via the facade.

In short, this allows you to :
- Ensure the persistence of business objects in a database or file.
- Extract data from queries emanating from the business layer.
- Apply read, write, update and delete (CRUD) operations.

3. Class diagram

3.1 Introduction

The class diagram is a type of UML diagram that describes the structure of a system by showing the different classes, their attributes, their methods and, in particular, their relationships, which can be seen using arrows.

This provides a better understanding of the direction of the classes in the business logic and their behaviour in this project.

3.2 How does the diagram fit in?

The class diagram is integrated into the architecture by detailing the structure of the business logic. It provides a better understanding of the internal workings of the application. In the overall design, it completes the overview by providing a level of detail that is essential for modelling, developing and maintaining the code.

3.3 Class logic

- **User** : Many other classes depend on this one. It concerns a user's personal and authentication information.
- **Place** : is the class that corresponds to a location with all the information about it.
- **Review** : This class represents the opinions left by users on the site.
- **Amenity** : groups together the various facilities available at a location.
- **Base model** : Many classes inherit from Base Model in order to centralise common attributes and methods, with the aim of avoiding repetition.

3.3.1 User

We decided to use the User Class as the starting point for our business logic, because it's the user who makes the site behave according to what they decide to do.

3.3.2 Base Model

In our view, the Base model class is essential because, as mentioned earlier, it is useful for centralising common attributes and methods, such as unique identifiers, or the creation and updating of certain data, thereby avoiding repetition of the code.

3.3.3 Attributs

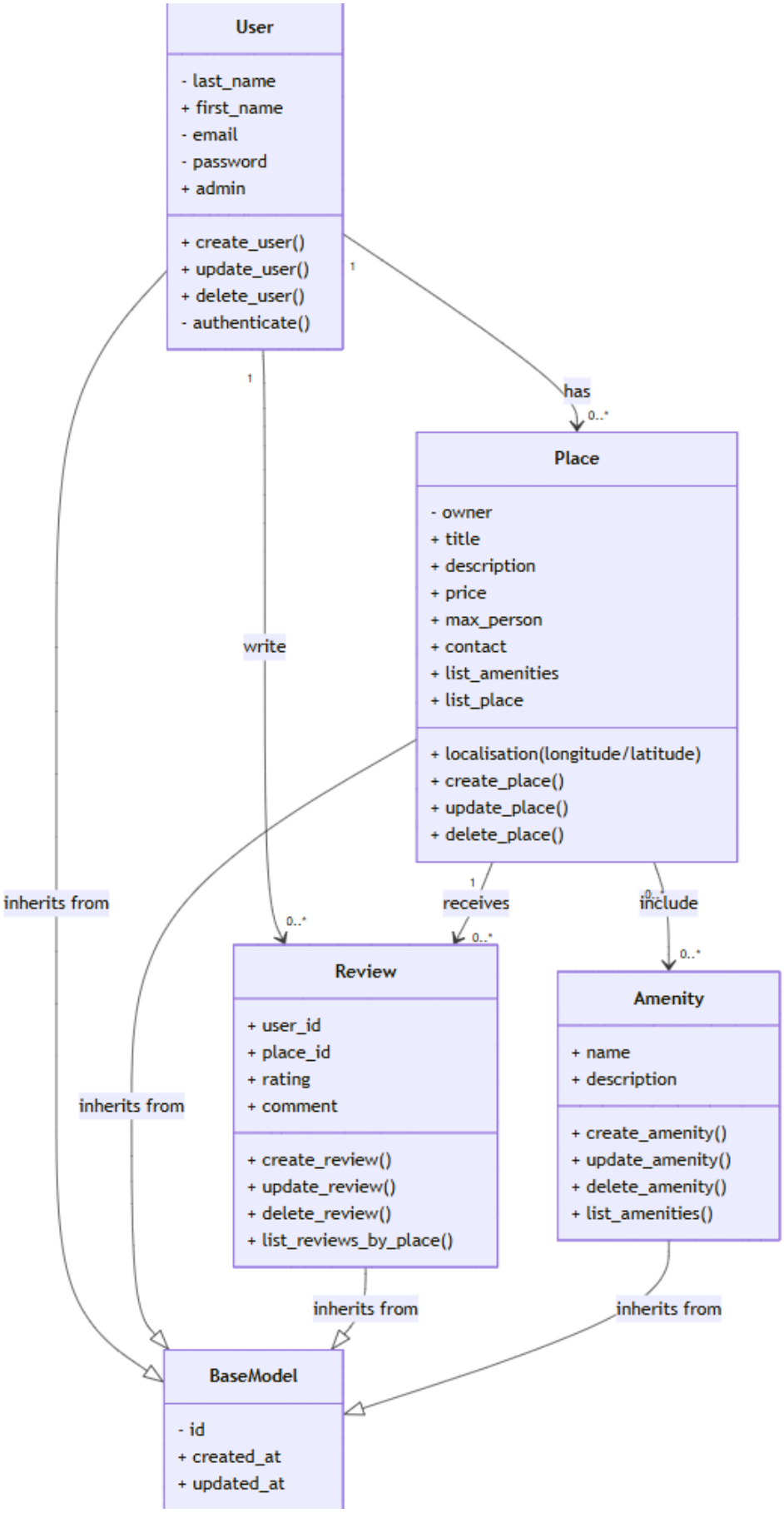
The attributes have been designed to suit their use. For example, for the user we have chosen the following attributes: surname, first name, email, password and admin management.

3.3.4 Methods

The design of the methods for each class follows the same logic. For example, for the User class, we'll use the - authenticate() method, which authenticates a user. The same applies to the other methods of the various classes.

3.3.5 Multiple associations

The Association Multiplicity represents the number of objects in a class that can be associated with an object in another class. We assign it a value according to the relationships between each class. For example, the User class (1) can have 0..* objects of the Location class, i.e. Zero, a single location or several locations. If the number is 1, then the instance of the class will only be associated with another instance, and for 0, there will be no link.

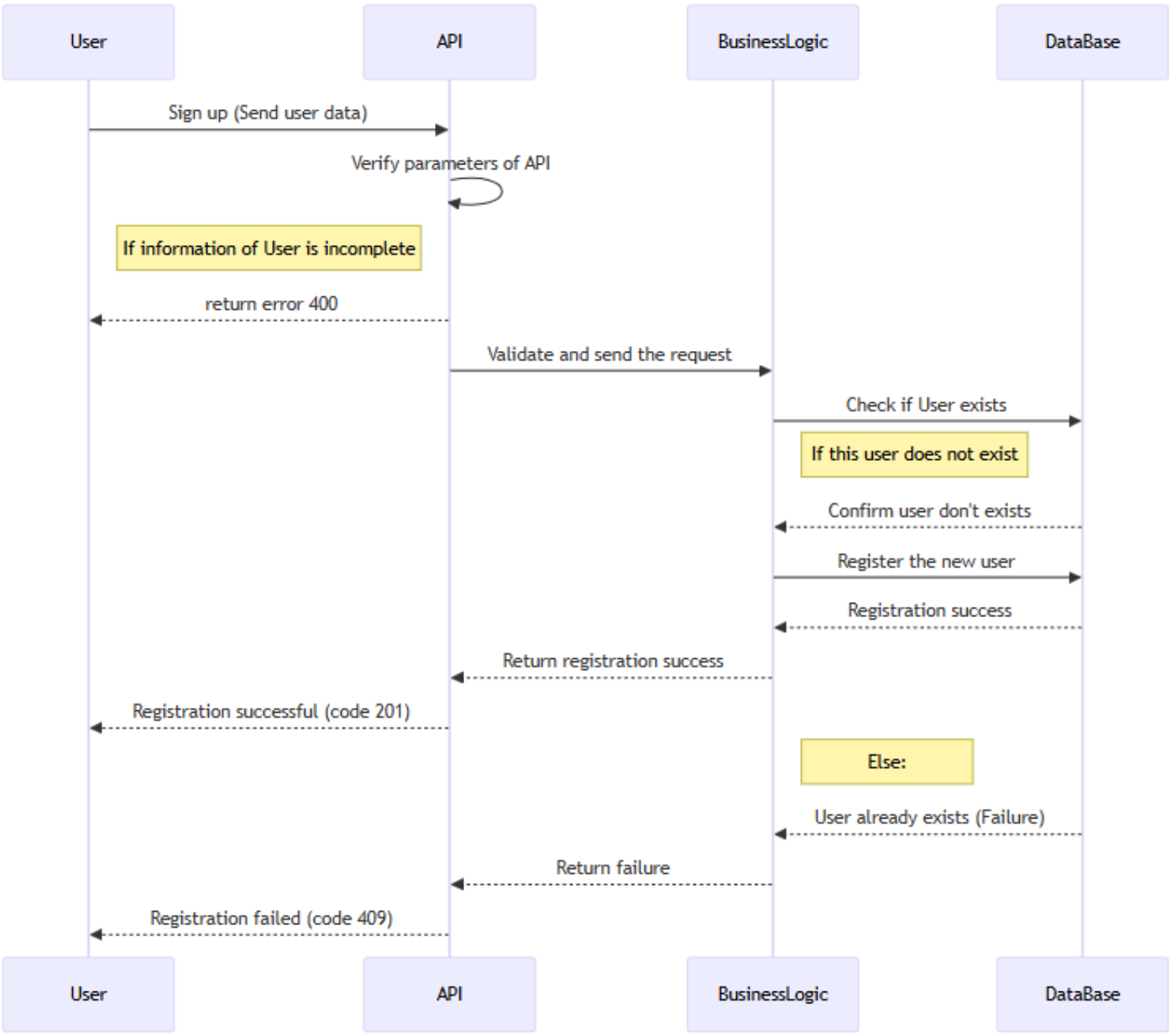


4. Sequence diagram

A sequence diagram shows the dynamic interactions between the different components of a system (API, business logic, database), step by step, during different user paths. Here are the paths developed for the needs of our project :

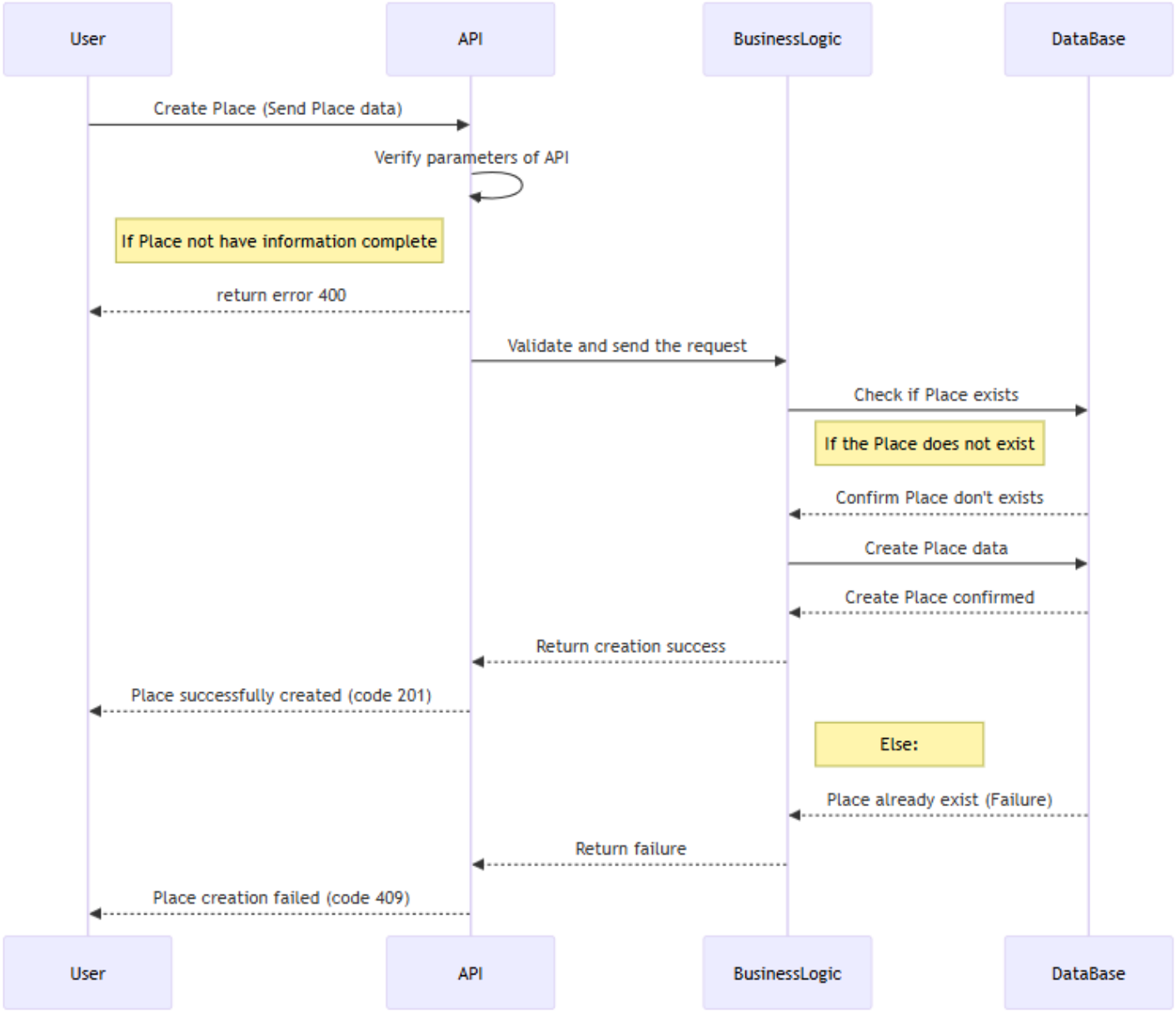
4.1 User registration

- The user sends his email address and password to the API.
- If the data is missing or invalid, the API returns a failure with an error message.
- If the data is valid, the API transmits it to the business layer.
- The business logic checks whether the user already exists by querying the database.
- If the user does not exist, the business logic asks the database to register the user and then returns a success to the API.
- If the user already exists: registration conflict, the business logic returns Failure and then to the user via the API.



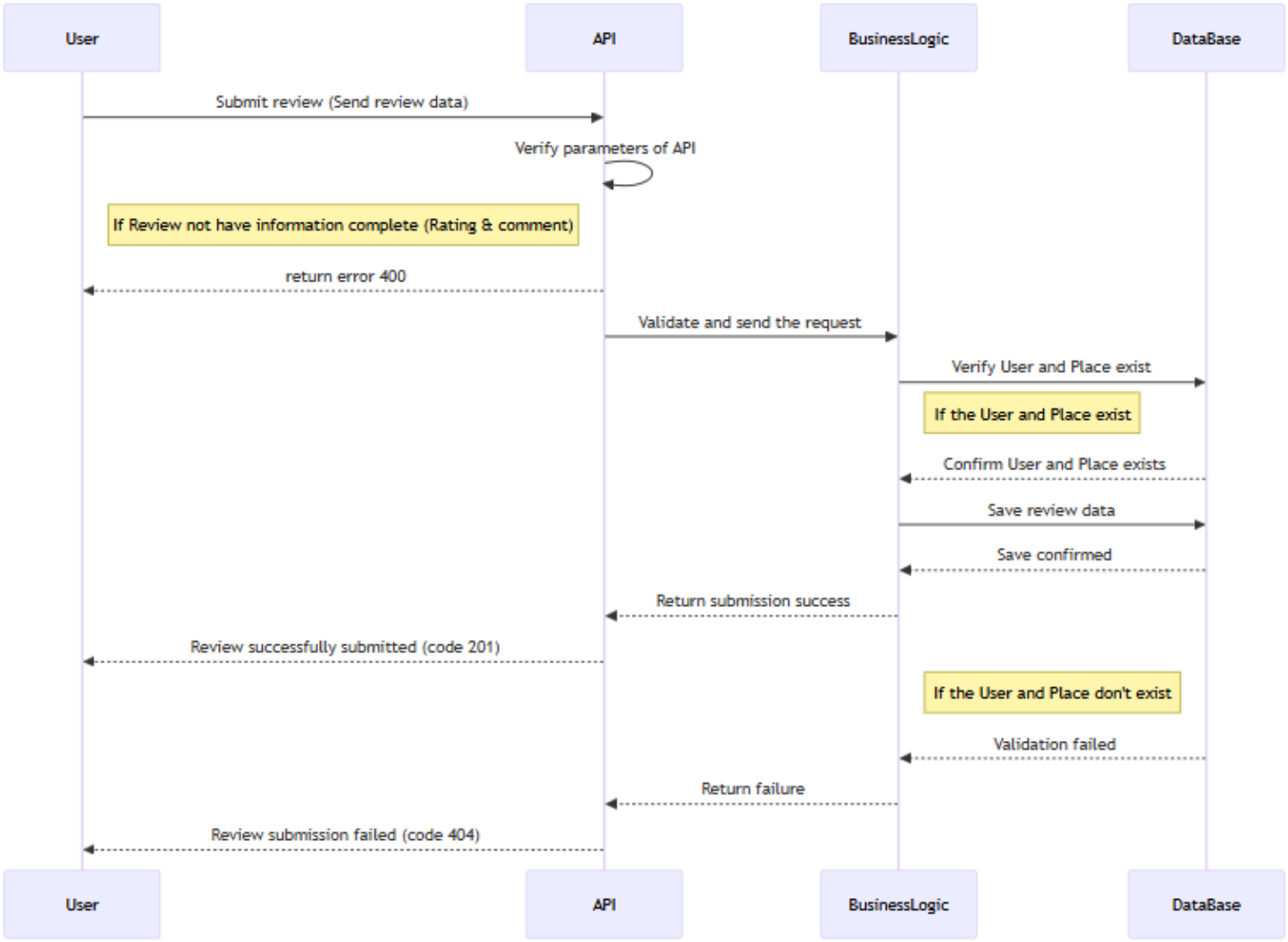
4.2 Creation of a place

- The user sends the data for their location to the API.
- If the data is incomplete, the API returns a failure.
- If the data is valid, the API sends it to the business layer.
- The business logic checks whether the location already exists:
- If it does not exist: create the location in the database, then return a Success.
- If it already exists: creation refused, return of a Failure.



4.3 Review Submission

- The user sends a review of a location via the API.
- If data is missing, the API returns Failure
- If the conditions are met, the API sends the data to the business logic :
- If the user and location exist: the notification is recorded and published, it returns Success,
- If the user or location does not exist: it returns Failure.



4.4 Fetching a List of Places

- The user sends a selection of criteria to the API to retrieve a list of locations.
- The API forwards the request to the business logic, which queries the database.
- A list of locations matching the criteria is returned (0, one, or more locations).
- The business logic returns Success to the API and then to the user.