HBnB - Documentación Técnica

Introducción

Este documento presenta la documentación técnica de **HBnB**, un proyecto inspirado en el modelo de AirBnB.

Tenemos varios diagramas UML que describen la estructura y el flujo de la aplicación, incluyendo:

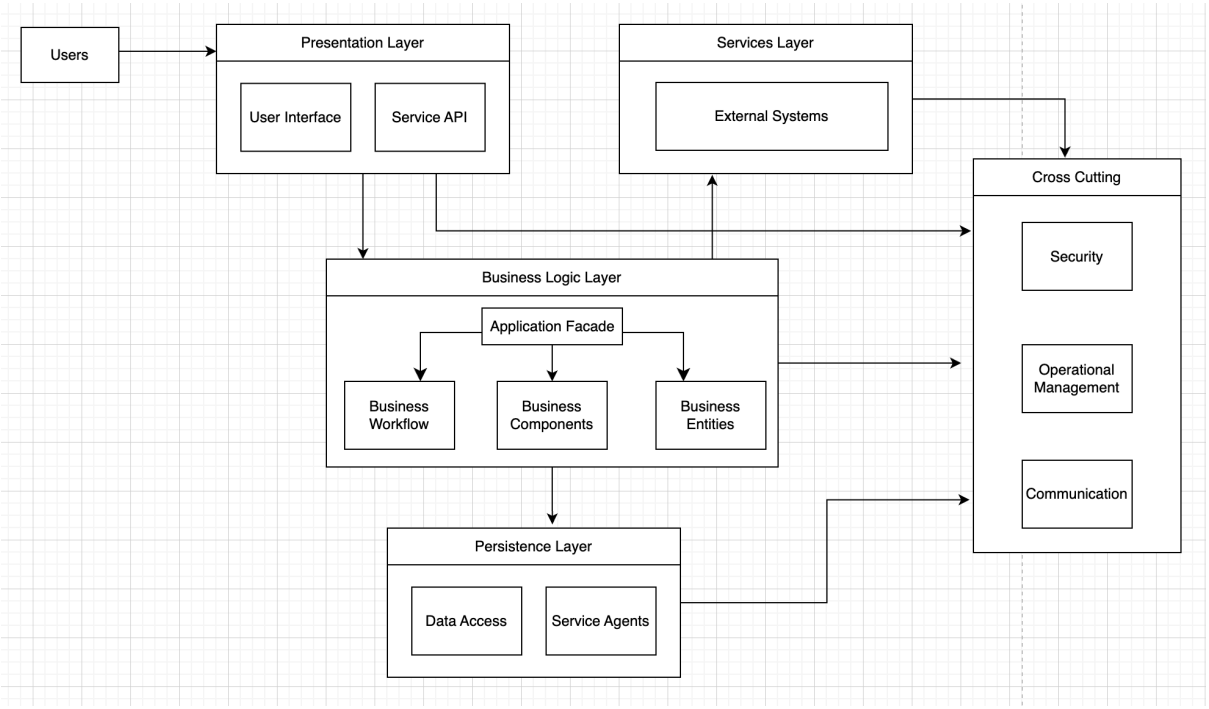
- **Diagrama de paquetes** para representar la arquitectura general.
- **Diagrama de clases** con las entidades y sus relaciones.
- **Diagramas de secuencia** que muestran cómo interactúan los diferentes componentes del sistema.

---

1. Arquitectura General (Diagrama de Paquetes)

La aplicación sigue una **arquitectura en capas**, dividiéndose en:

- **Presentation Layer:** Maneja las solicitudes de los usuarios a través de una API.
- **Business Logic Layer:** Contiene las reglas y la lógica.
- **Persistence Layer:** Responsable del almacenamiento y recuperación de datos



1. El usuario interactúa con la aplicación a través de la Interfaz de Usuario (**User Interface**). Actualmente, como no tenemos frontend, las peticiones se hacen directamente a la **Service API**, que actúa como intermediaria entre el usuario y la aplicación mediante los endpoints.

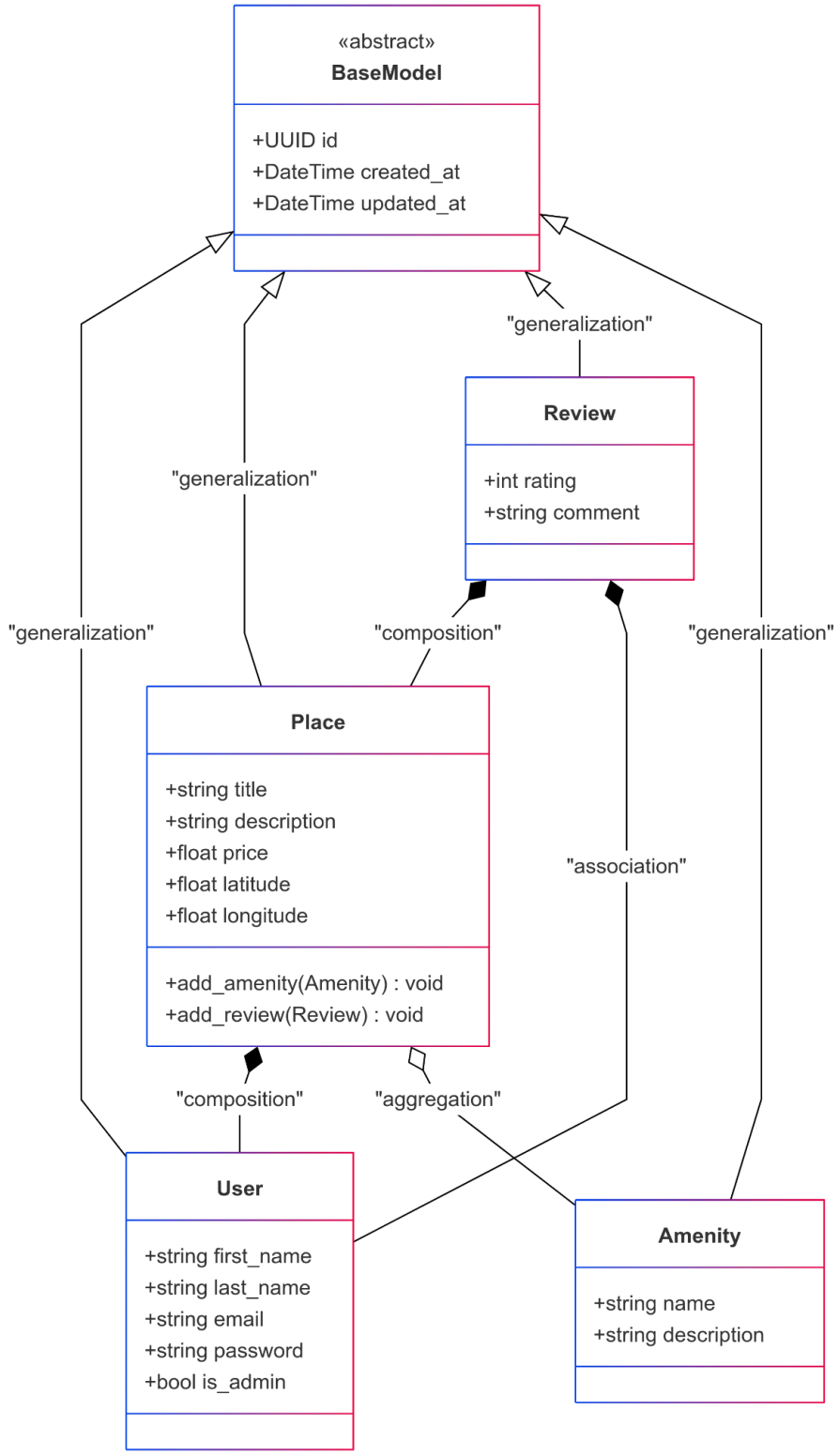
2. La **Presentation Layer** se comunica con la **Business Logic Layer** usando el **Facade Pattern**. Básicamente, la Presentation Layer no se preocupa por los detalles internos, solo envía la solicitud y espera el resultado. En la **Business Logic Layer** se manejan todas las reglas y la lógica de la aplicación. Para esto, se usa **Application Facade**, evitando que la Presentation Layer acceda directamente a los modelos o la base de datos. En pocas palabras, la **Application Facade** actúa como intermediaria, llamando a los procesos internos cuando es necesario. Dentro de la **Business Logic Layer**, tenemos:
    - **Application Facade**: Intermediario, evita que la capa de presentación interactúe con los componentes internos.
    - **Business Workflow**: Define los procesos y la lógica (ejemplo: verificar si un usuario ya existe antes de registrarlo).
    - **Business Components**: Manejan las operaciones específicas de cada entidad.
    - **Business Entities**: Representan los modelos de datos (*User, Place, Review, Amenity*).

3. La **Business Logic Layer** accede a la **Persistence Layer** para leer o escribir datos, ya que no se interactúa directamente con la base de datos desde la API. Dentro de esta capa, tenemos:
    - **Data Access**: Maneja las consultas y operaciones en la base de datos.
    - **Service Agents**: Interactúan con servicios externos si la aplicación lo requiere.

4. La **Services Layer** permite la integración con sistemas externos. Es clave porque algunas funciones requieren conectarse con otros servicios, como **Google Maps** para ubicaciones o plataformas de pago. La **Services Layer** gestiona estas conexiones sin exponerlas a la lógica interna.

5. **Cross Cutting Layer:** Maneja aspectos transversales que afectan a todas las capas, como:
    - **Security**: Autenticación, control de acceso, cifrado de datos.
    - **Operational Management**: Monitoreo del sistema y registros.
    - **Communication**: Mensajes y notificaciones (por ejemplo, enviarle un email al usuario cuando se registra).

2. Diagrama de Clases (Business Logic Layer)



-> Un usuario está representado por la clase **User**. Tiene atributos como **id, first_name, last_name, email, password** y **is_admin**. Puede interactuar con **Places** y **Reviews**.

- Tiene una **relación de Asociación** con **Review**, ya que un usuario puede crear múltiples reseñas para diferentes lugares.
- Está relacionado por **Composición** con **Place**, porque un usuario puede ser propietario de varios lugares.

-> Un lugar está representado por la clase **Place**. Tiene atributos como i**d, title, description, price, latitude** y **longitude**. También cuenta con dos funciones:

- **add_amenity(Amenity)**: Agrega una amenidad a un lugar.
- **add_review(Review)**: Se encarga de agregar una reseña al lugar.

Las relaciones que tiene son:

- **Composición con Review**: Un lugar puede tener varias reseñas, pero si se elimina, sus reseñas asociadas también desaparecen.
- **Agregación con Amenity**: Un lugar puede tener múltiples amenities, pero estas pueden existir de manera independiente. Es decir, dos lugares distintos pueden compartir la misma amenity.

-> Las reseñas están representadas por la clase **Review**. Tienen atributos como **id, rating, comment, created_at** y **updated_at.**

- Tiene una relación de **Composición con Place**, ya que una reseña pertenece a un lugar específico y, si el lugar se elimina, sus reseñas dejan de mostrarse,.
- Está relacionado por **Asociación con User**, porque un usuario puede crear múltiples reseñas.

-> Una comodidad está representada por la clase **Amenity**. Tiene atributos como **id, name, description, created_at** y **updated_at.**

- Tiene una relación de **Agregación con Place**, ya que, un lugar puede tener múltiples amenities, y un mismo amenity puede estar asociada a varios lugares diferentes.

3. Diagramas de Secuencia (Flujo de API Calls)

Los diagramas de secuencia son un tipo de diagrama UML que sirven para representar la interacción entre objetos en un sistema, muestran en qué orden ocurren los eventos y cómo se envían los mensajes entre ellos, tanto de error como de validación.

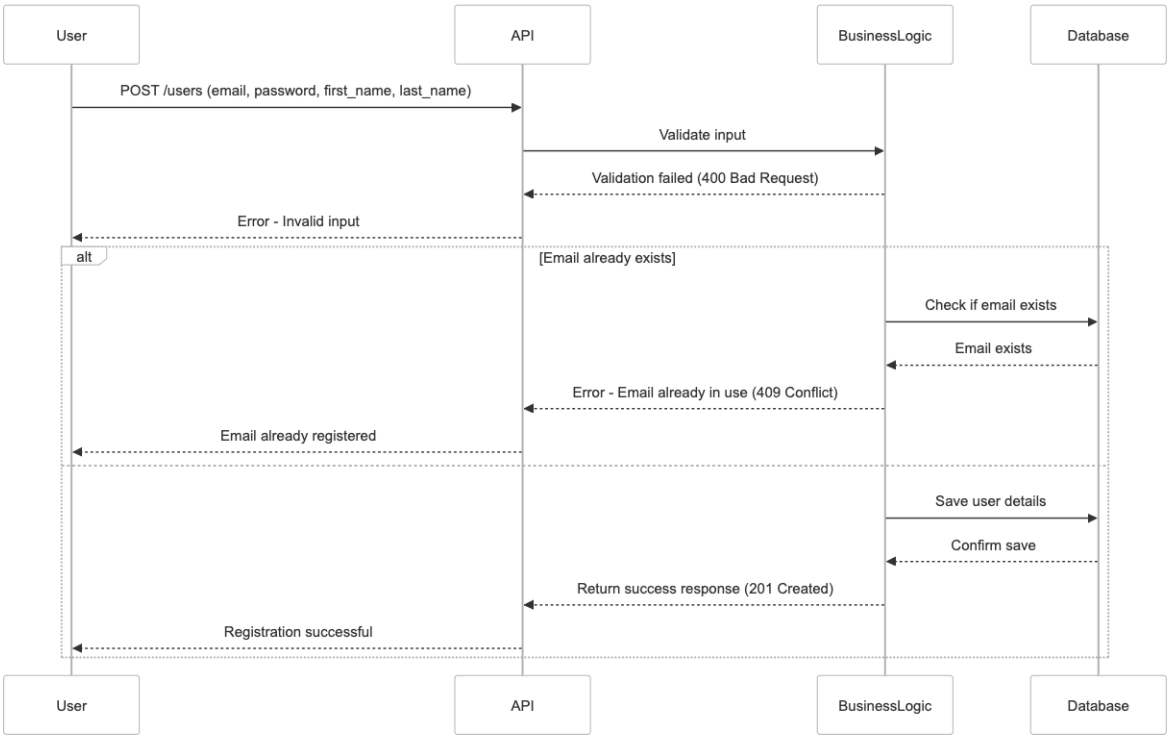
Componentes principales de un diagrama de secuencia:

Objetos:Representan las entidades que interactúan en el sistema (pueden ser usuarios, sistemas externos o componentes internos).

- **Líneas de vida**: Son líneas verticales que muestran el tiempo de existencia de un objeto.
- **Mensajes**: Representan la comunicación entre los objetos a través de líneas con flechas.
- **Activación**: Barras en las líneas de vida que indican cuándo un objeto está realizando una acción.
- **Condiciones y bucles**: Se pueden agregar condiciones (`alt`, `opt`) y bucles (`loop`) para representar lógica condicional o repetitiva.

3.1 Registro de Usuario

Este diagrama describe el flujo de registro de un usuario en la aplicación, incluyendo validaciones y posibles errores.



1. El usuario envía una solicitud POST a la API con los datos: **email, password, first_name** y **last_name**.

2. La API envía la solicitud a la capa BusinessLogic para validar los datos:

   Si los datos son inválidos, la API responde con **400 Bad Request**.

   - El código de estado **HTTP 400** indica que la solicitud es incorrecta debido a un error del cliente, como datos faltantes o con formato incorrecto. En este caso, significa que el usuario no ingresó toda la información requerida o la ingresó de manera incorrecta.
   - Cuando esto ocurre, el proceso de registro se detiene y el usuario recibe un mensaje de error.

3. La BusinessLogic verifica si el email ya está registrado en la base de datos:

   Si el email ya existe, se devuelve **409 Conflict**.

   - El código de estado **HTTP 409** indica que hay un conflicto con el estado actual del recurso. En este caso, significa que el email ingresado ya está registrado en la base de datos.
   - El usuario recibe un mensaje de error indicando que ya existe una cuenta con ese correo electrónico.

   Si el email no está registrado, el proceso continúa y los datos del usuario se guardan en la base de datos.

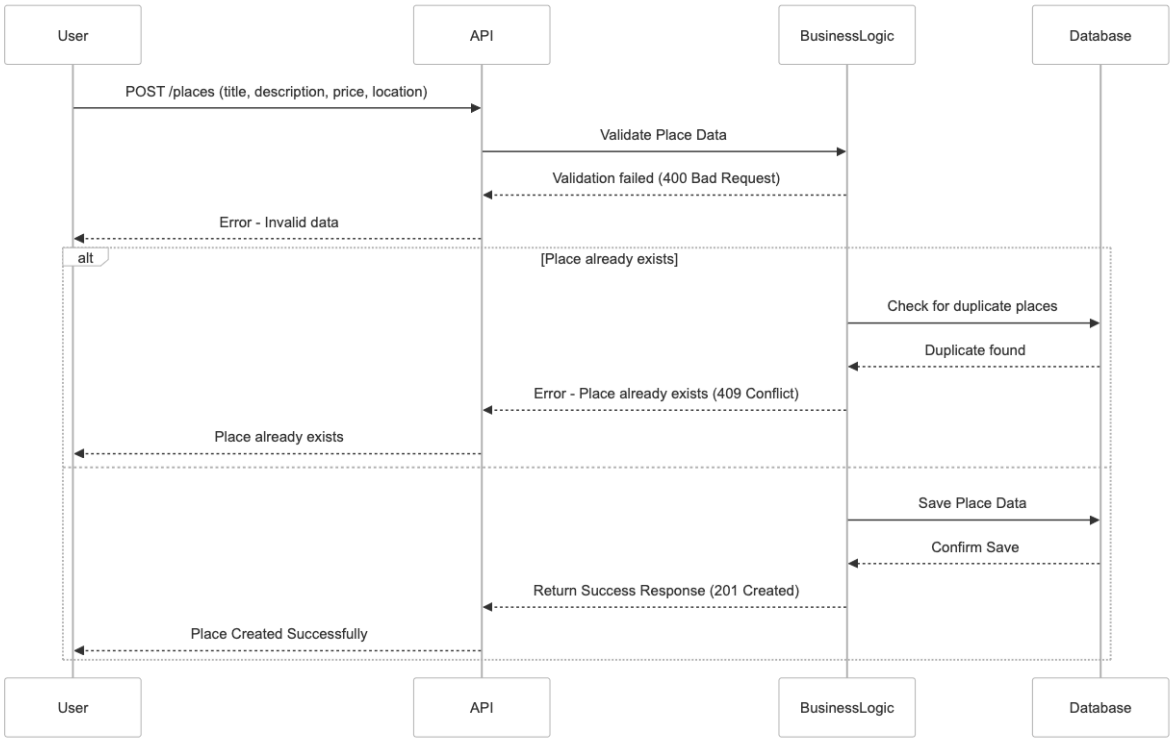
4. La base de datos confirma que los datos fueron almacenados correctamente y responde a la BusinessLogic.

5. La BusinessLogic envía una respuesta exitosa a la API, indicando que el usuario ha sido creado correctamente (**201 Created**).
   - El código de estado **HTTP 201** indica que la solicitud fue exitosa y que se ha creado un nuevo recurso en la base de datos.

6. La API responde al usuario con un mensaje de éxito, confirmando el registro.

3.2 Creación de un Lugar

Describimos el flujo de creación de un lugar (Place), asegurándonos que no se creen lugares duplicados y validando los datos.



1. El usuario envía una solicitud **POST** con los datos del lugar (**title**, **description**, **price**, **location**).

2. La **API** envía la solicitud a la capa **BusinessLogic** para validar los datos.
    - Si los datos son inválidos, la API responde con **400 Bad Request**.
      (El código de estado **HTTP 400** indica que la solicitud es incorrecta debido a un error del cliente, como datos faltantes o con formato incorrecto)
      Luego, el proceso se detiene.

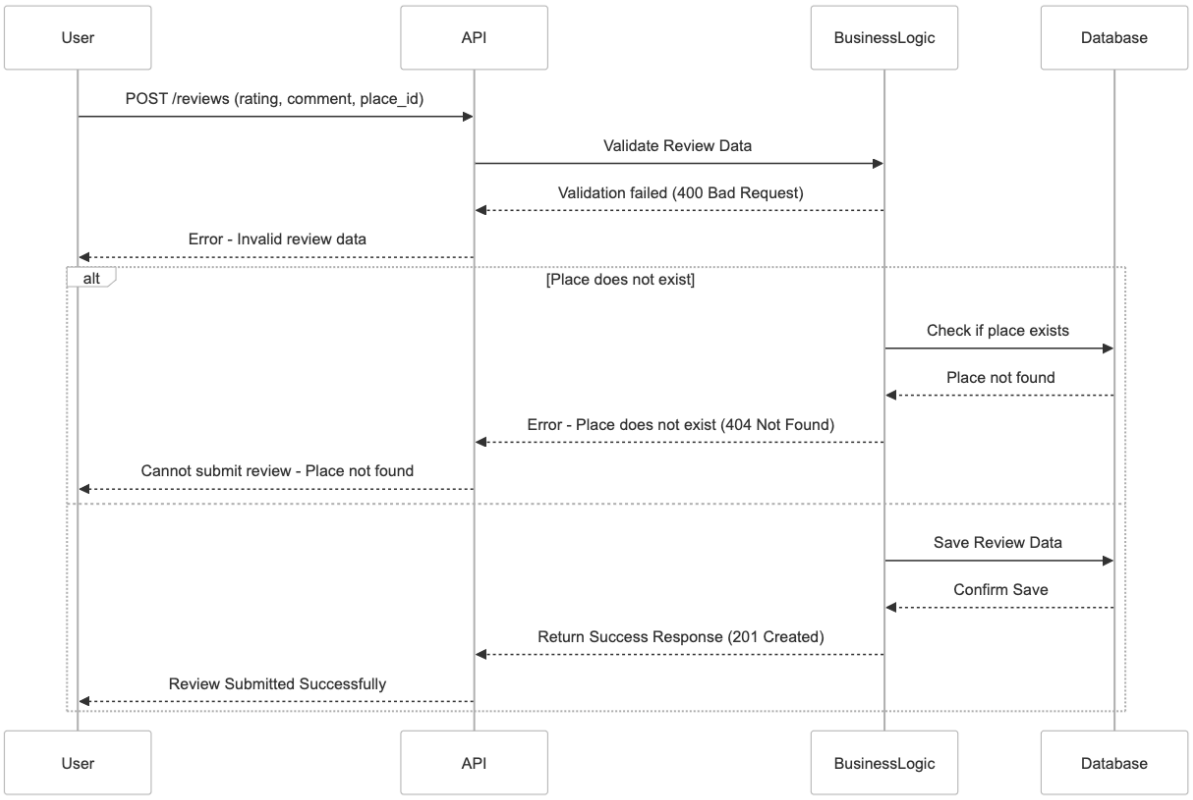
3. La **BusinessLogic** verifica en la base de datos si el lugar ya existe.
    - Si el lugar ya está registrado, responde con **409 Conflict**.
      (El código de estado **HTTP 409** indica que hay un conflicto con el estado actual del recurso. En este caso, significa que el lugar ya está registrado en la base de datos. El usuario recibe un mensaje de error indicando que el lugar ya existe.)
    - Si el lugar no existe, se guarda en la base de datos.

4. La **base de datos** confirma que los datos fueron guardados correctamente.

5. La **BusinessLogic** informa a la API que la creación fue exitosa y responde con **201 Created**. (El código de estado **HTTP 201** indica que la solicitud fue exitosa y que se ha creado un nuevo recurso en la base de datos.)
   Finalmente, la API envía al usuario un mensaje de éxito.

3.3 En Reseñas

En el siguiente esquema vamos a ver cómo un usuario puede dejar una reseña (Review) en un lugar, verificando que el lugar exista.



1. El usuario envía una solicitud POST con los datos de la reseña: **rating, comment** y **place_id.**

2. La API reenvía la solicitud a BusinessLogic para validar los datos.
    - Si los datos son inválidos, la API responde con 400 Bad Request y se detiene el proceso.

3. La BusinessLogic verifica en la base de datos si el lugar (place_id) existe.
    - Si el lugar no existe, responde con 404 Not Found. (El código de estado HTTP 404 indica que el servidor no puede encontrar el recurso solicitado. En este caso, significa que el **`place_id`** ingresado no corresponde a ningún lugar registrado en la base de datos.)
      Luego, el usuario recibe un error.
    - Si el lugar existe, se guarda la reseña en la base de datos.

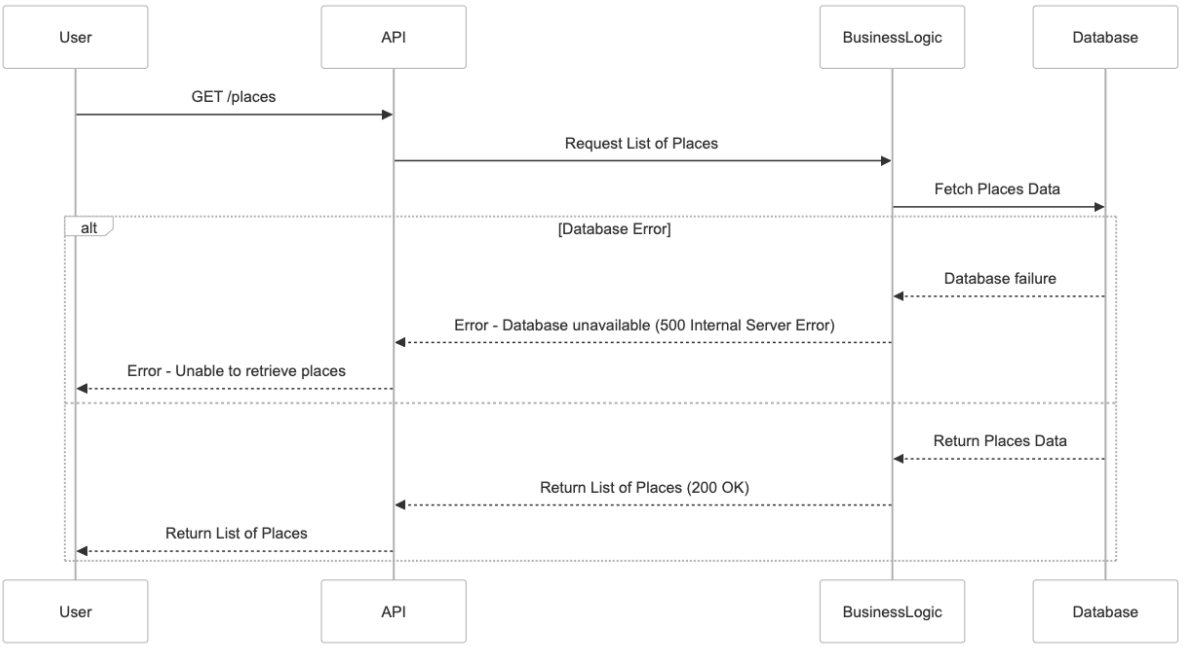
4. La base de datos confirma que la reseña fue guardada correctamente.

5. La BusinessLogic informa a la API que la reseña fue creada exitosamente y responde con 201 Created.

6. La API responde al usuario con un mensaje de éxito.

3.4 Obtención de una Lista de Lugares

En el siguiente esquema vemos como hace un usuario para ver una lista de lugares disponibles.



1. El usuario envía una solicitud **GET** para obtener los lugares.

2. La **API** solicita los datos a **BusinessLogic**.

3. **BusinessLogic** consulta la base de datos para obtener los lugares.
    - Si ocurre un error en la base de datos, responde con **500 Internal Server Error**.(Este código indica que el servidor encontró una condición inesperada que le impidió completar la solicitud.)

4. La **base de datos** devuelve la lista de lugares.

5. La **API** responde al usuario con la lista y el código **200 OK**. (Este código indica que la solicitud se completó con éxito y los datos están disponibles.)

Autores:

- **Nazarena Aranda**
- **Ignacio Devita**