

## Portal de Reservas para Espacios de Trabajo Compartido

### Descripción

#### Objetivo del Proyecto:

El objetivo principal del portal es facilitar la reserva de espacios de trabajo compartido (coworking) para usuarios que necesitan un lugar cómodo y equipado para trabajar temporalmente. El portal permitirá a los propietarios de espacios de trabajo ofrecer sus instalaciones a los usuarios, quienes podrán explorar, reservar y dejar reseñas sobre los espacios utilizados.

#### Descripción General:

El Portal de Reservas para Espacios de Trabajo Compartido es una plataforma web diseñada para conectar a personas que buscan un lugar para trabajar con propietarios de espacios de coworking. Los usuarios pueden registrarse en el portal, buscar espacios disponibles, realizar reservas en línea, y dejar reseñas sobre su experiencia. Los propietarios de los espacios pueden administrar sus ofertas, controlar la disponibilidad y gestionar las reservas realizadas por los usuarios.

#### Funcionalidades Principales:

##### 1. Registro y Autenticación de Usuarios:

- **Registro de Usuarios:** Los usuarios pueden registrarse en la plataforma proporcionando su nombre, email, contraseña, y otros datos personales.
- **Autenticación:** Los usuarios pueden iniciar sesión para acceder a sus perfiles, hacer reservas y revisar el historial de reservas.

##### 2. Gestión de Espacios de Trabajo:

- **Administración de Espacios:** Los propietarios pueden registrar sus espacios de trabajo en la plataforma, proporcionando detalles como nombre, ubicación, servicios disponibles, y tarifas.
- **Gestión de Disponibilidad:** Los propietarios pueden establecer y actualizar la disponibilidad de sus espacios para diferentes fechas y horas.

##### 3. Sistema de Reservas:

- **Reserva de Espacios:** Los usuarios pueden buscar espacios de trabajo disponibles según su ubicación y necesidades, visualizar detalles y fotos del espacio, y realizar reservas para fechas y horarios específicos.
- **Confirmación y Estado de Reservas:** Las reservas pueden tener estados como "pendiente", "confirmada", "completada" o "cancelada", según el progreso de la reserva.

##### 4. Sistema de Reseñas:

- **Calificación y Comentarios:** Después de utilizar un espacio de trabajo, los usuarios pueden dejar una reseña calificando el espacio y proporcionando comentarios sobre su experiencia.
- **Visualización de Reseñas:** Las reseñas son visibles para otros usuarios interesados en reservar el mismo espacio.

## 5. Notificaciones y Alertas:

- **Alertas de Reservas:** Los usuarios y propietarios reciben notificaciones sobre las reservas realizadas, confirmadas o canceladas.
- **Recordatorios:** Los usuarios pueden recibir recordatorios antes de su reserva para asegurar que no olviden su cita.

## Consideraciones Técnicas:

- **Backend:** El sistema estará desarrollado utilizando Node.js con un framework flask para manejar las solicitudes y MongoDB como base de datos.
- **Frontend:** La interfaz de usuario se desarrollará utilizando Angular, proporcionando una experiencia de usuario fluida y dinámica.
- **Base de Datos:** MongoDB será utilizado para almacenar los datos de usuarios, espacios de trabajo, reservas y reseñas, aprovechando su flexibilidad y escalabilidad.

## Casos de Uso:

1. Un usuario registrado busca un espacio de coworking en su ciudad, lo reserva para un día completo y recibe una confirmación instantánea de la reserva.
2. Un propietario de un espacio de coworking recibe una solicitud de reserva, la confirma, y posteriormente recibe una reseña positiva del usuario.
3. Un usuario nuevo se registra en la plataforma, explora los espacios disponibles, y se suscribe a notificaciones de disponibilidad para ser informado cuando haya un espacio que cumpla con sus necesidades.

## Usuarios Objetivo:

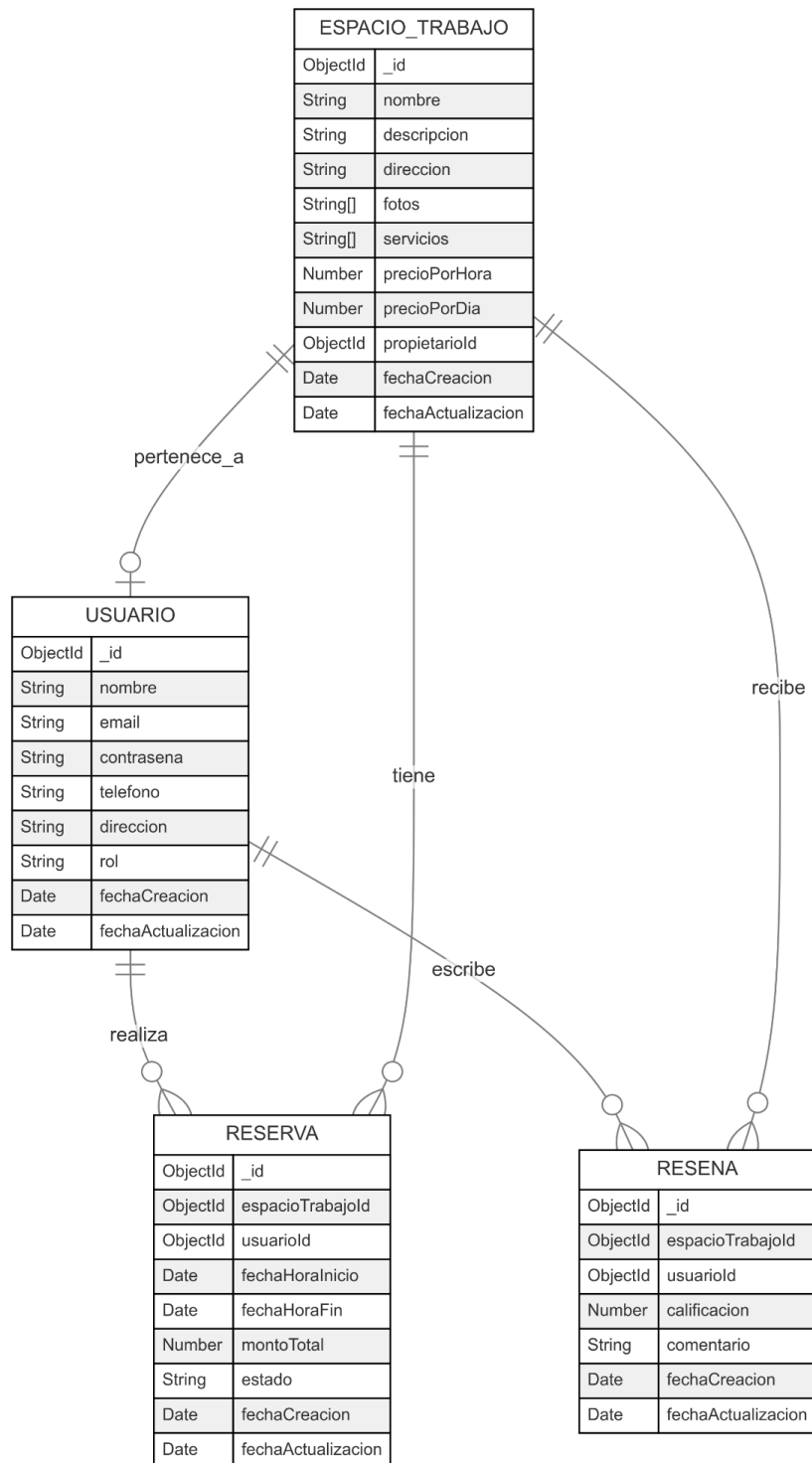
- **Freelancers y Profesionales Remotos:** Personas que necesitan un lugar temporal para trabajar con buenas instalaciones y ambiente profesional.
- **Pequeñas Empresas y Startups:** Equipos que buscan espacios flexibles para reuniones, trabajo en equipo o proyectos cortos.
- **Propietarios de Espacios de Coworking:** Dueños de instalaciones que desean maximizar la utilización de sus espacios ofreciendo a través del portal.

## Backend

Una vez analizado nuestro proyecto, podremos crear nuestro modelo Entidad Relación para nuestro sistema.

**Nota:** Aunque trabajamos con una base de datos NoSQL, quería remarcar la relación de las tablas o colecciones JSON.

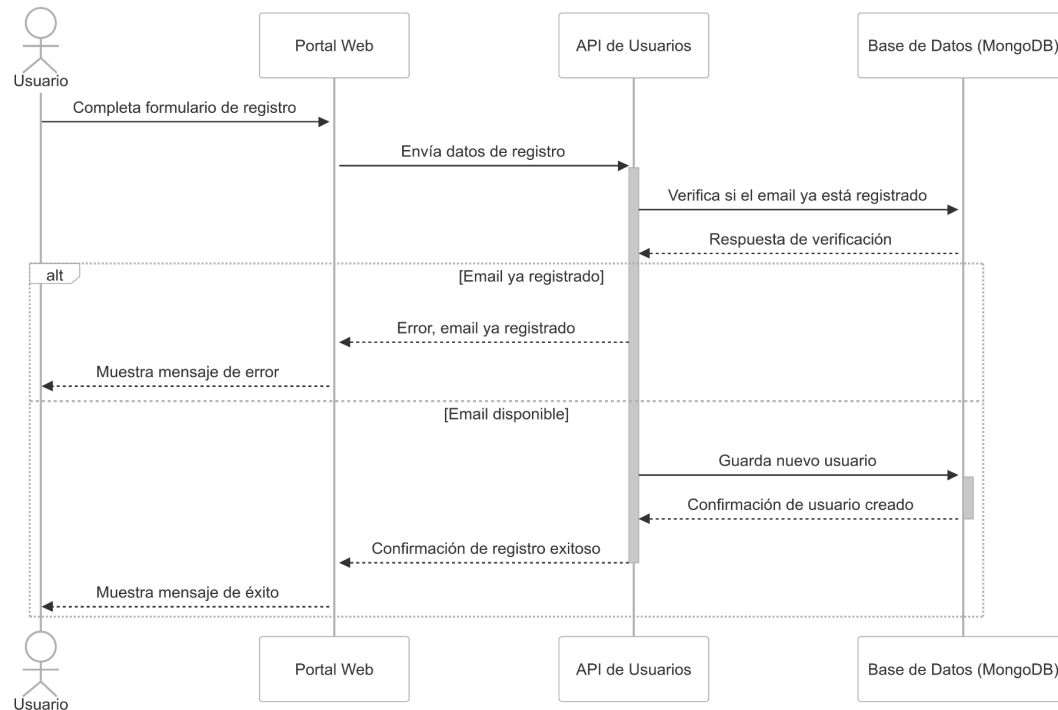
## Modelo E-R



**Figura 1:** Modelo E-R.

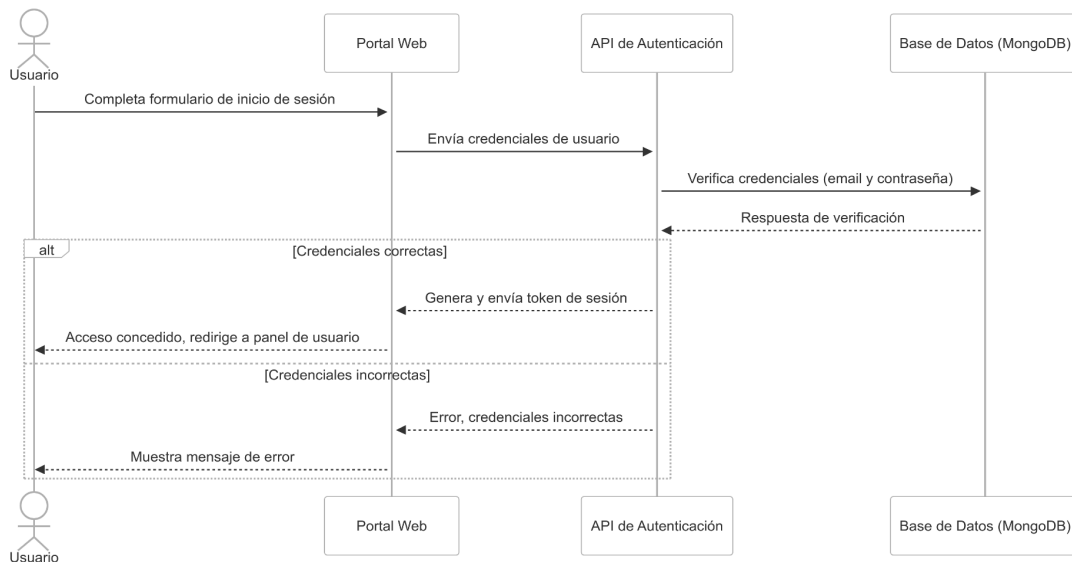
Como siguiente, crearemos nuestros diagramas de secuencia.

## Registro de Usuario



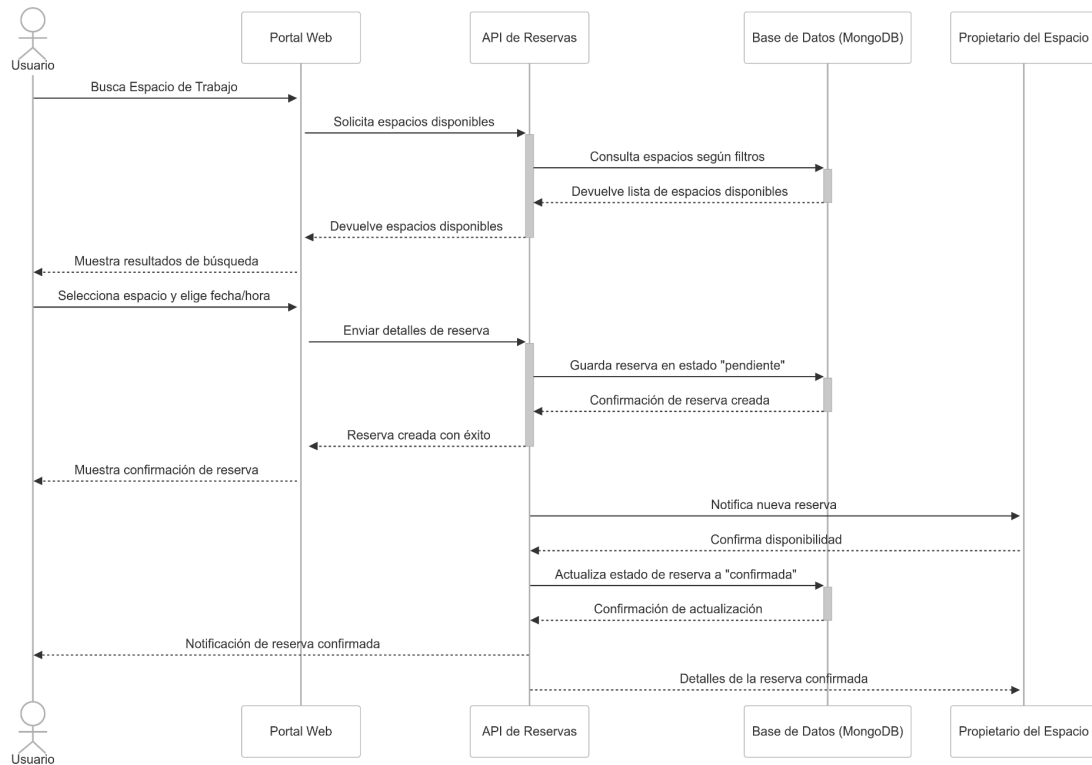
**Figura 2:** Diagrama de secuencia de Registro.

## Login



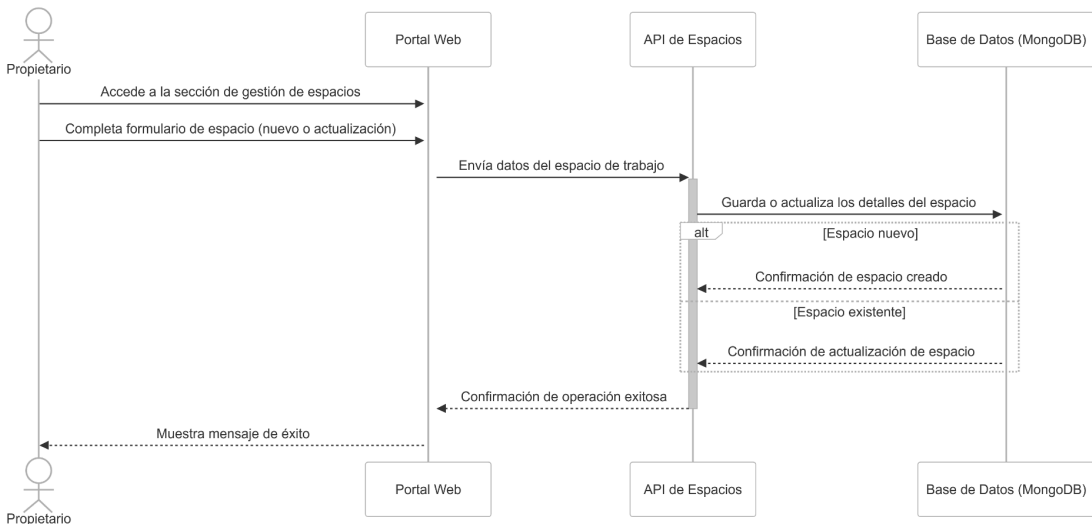
**Figura 3:** Diagrama de secuencia de Login.

## Reserva



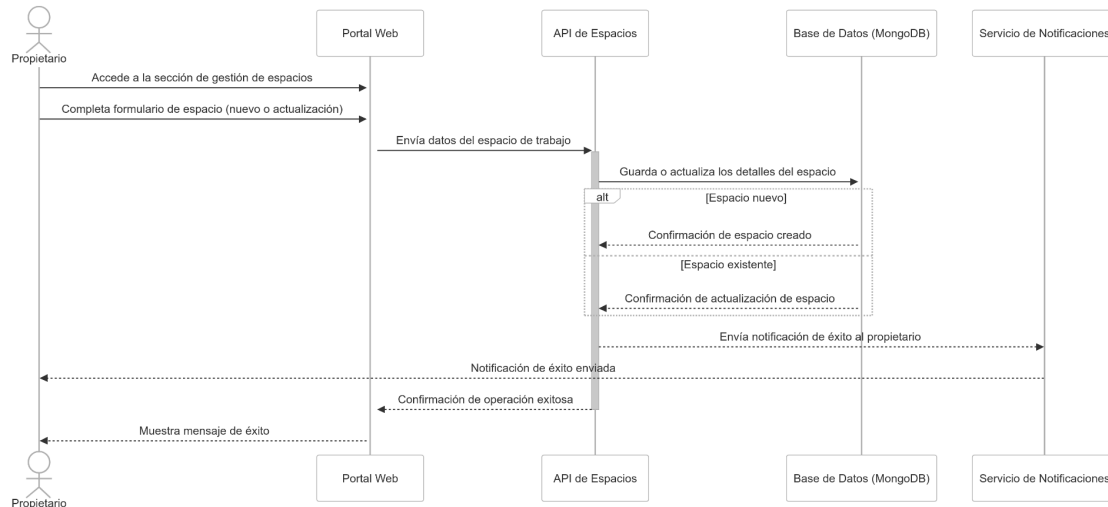
**Figura 4:** Diagrama de secuencia de Reserva.

## Gestión de Espacios



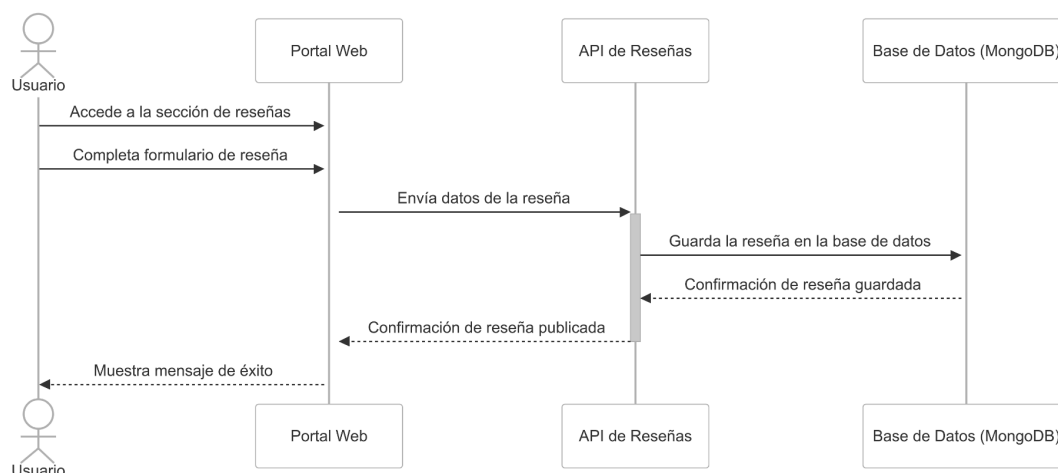
**Figura 5:** Diagrama de secuencia Gestión de Espacios.

## Gestión de Espacios con Notificaciones



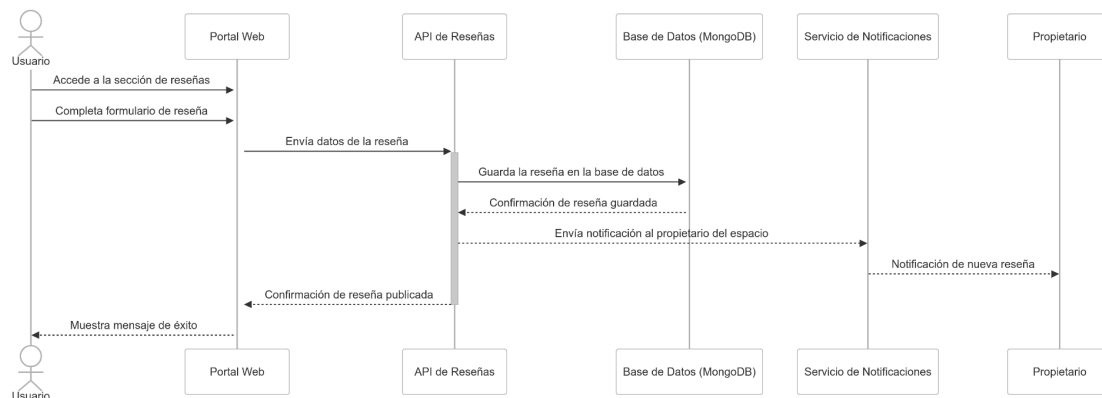
**Figura 6:** Diagrama de secuencia Gestión de Espacios con Notificaciones.

## Reseña



**Figura 7:** Diagrama de secuencia de Reseña.

## Reseñas con Notificaciones



**Figura 8:** Diagrama de secuencia de Reseña con Notificaciones.

## Diagrama de Clases

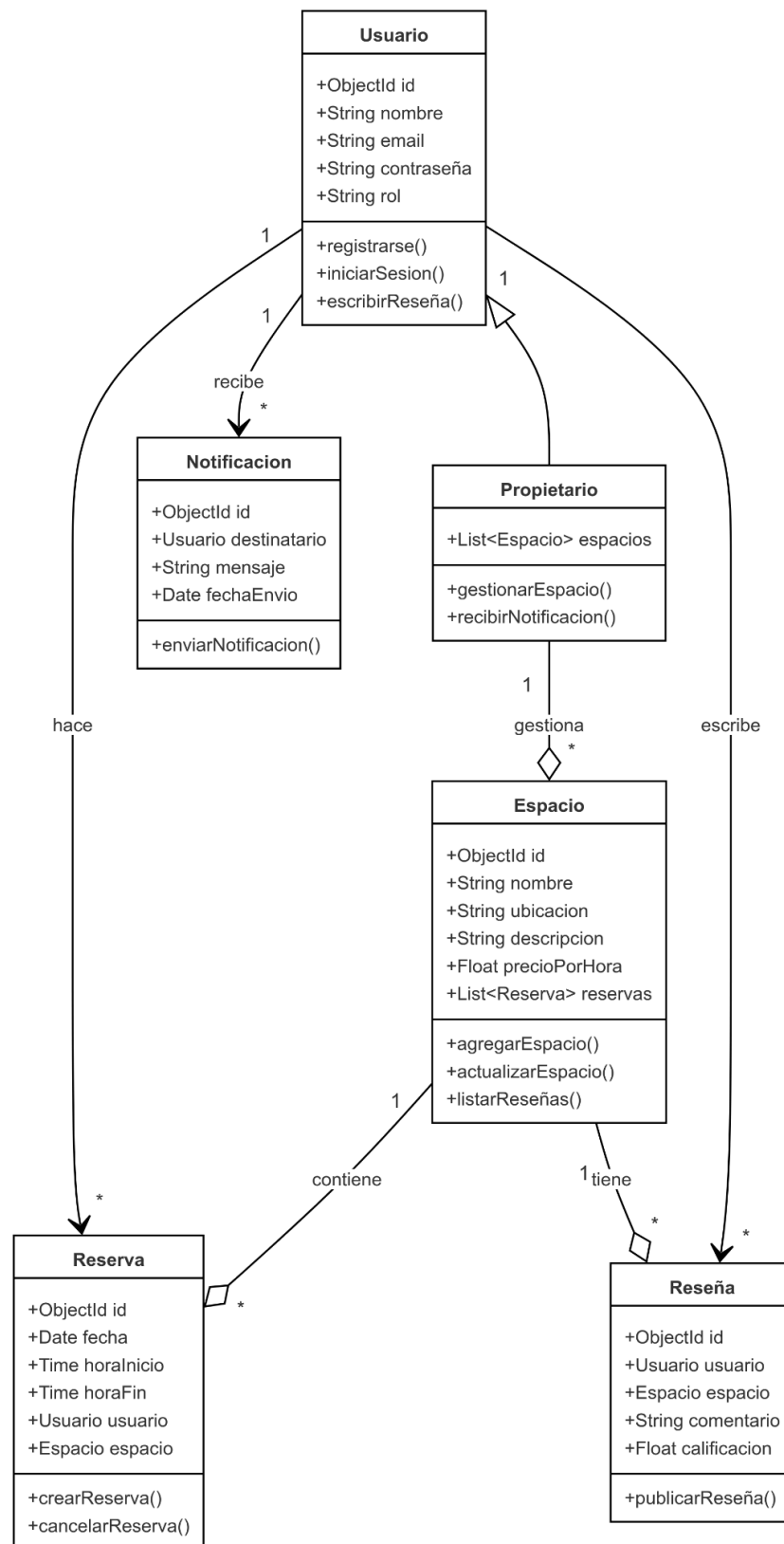
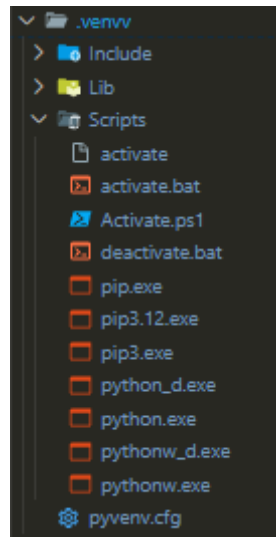


Figura 9: Diagrama de secuencia de Clases.

Una vez obtenido nuestro proyecto, crearemos nuestro Backend en el microservicio de Flask API.

Para conseguir que nuestro proyecto Flask funcione correctamente, deberemos crear un entorno virtual para su funcionamiento y ejecutar sus scripts activate.bat para instalar las dependencias que necesitaremos para su funcionamiento.



**Figura 10:** Estructura virtual python.

Ingresamos al entorno de Flask e instalamos flask, cors, jwt y pymongo. Para verificar nuestras dependencias usaremos pip freeze para ver las librerías que ocuparemos.

```
Backend > requirements.txt
1 blinker==1.8.2
2 click==8.1.7
3 colorama==0.4.6
4 dnspython==2.6.1
5 Flask==3.0.3
6 Flask-Cors==5.0.0
7 Flask-JWT-Extended==4.6.0
8 Flask-PyMongo==2.3.0
9 itsdangerous==2.2.0
10 Jinja2==3.1.4
11 MarkupSafe==2.1.5
12 PyJWT==2.9.0
13 pymongo==4.8.0
14 Werkzeug==3.0.4
```

**Figura 11:** Dependencias de librerías.

Configuraremos en MongoDB Compass la conexión por defecto, claro, ejecutamos mondongo primero para el funcionamiento.



## New Connection

Manage your connection settings

URI ⓘ

Edit Connection String 

mongodb://localhost:27017/

Name

Color

No Color ▼


☐ Favorite this connection

Favoriting a connection will pin it to the top of your list of connections


➤ Advanced Connection Options

### How do I find my connection string in Atlas?

If you have an Atlas cluster, go to the Cluster view. Click the 'Connect' button for the cluster to which you wish to connect.

[See example](#) 

### How do I format my connection string?

[See example](#) 

**Figura 12:** Comunicación Local MongoDB Compass.

Una vez conectados crearemos las colecciones de nuestra base de datos



**Figura 13:** Colecciones de la base de datos.

Aprovechamos que el mongodb genera una base de datos dinámica según las peticiones y seguiremos con Flask y su código.

Nuestro proyecto, se divide en varios archivos, como:

**config:** almacenará las configuraciones de la aplicación, las que incluimos claves secretas y comunicación a MongoDB.

**init:** Aquí configuramos nuestra aplicación Flask.

**routes:** En esta sección configuramos las rutas o Endpoints de la aplicación y sus funcionalidades. Además de las respuestas y peticiones.

**model:** Definimos las entidades mediante el uso de clases

**run:** este archivo se centrará en correr nuestra aplicación flask

**requirements:** Este archivo solo contiene datos generales sobre las dependencias que ocupamos en este proyecto.

## Conexión MongoDB

Lo primero para conectarnos a mongodb en nuestro entorno **visual studio code**, es el usar de MONGO\_URI, donde almacenará la conexión a la base de datos que usaremos y las colecciones que ésta posee.

```
Backend > config.py > Config
1 import os
2
3 class Config:
4     SECRET_KEY=os.environ.get('SECRET_KEY') or 'clave'
5     MONGO_URI = 'mongodb://localhost:27017/portal_reservas'
6     JWT_SECRET_KEY = os.environ.get('JWT_SECRET_KEY') or 'clave_jwt'
```

**Figura 14:** Comunicación Flask con MongoDB.

Para garantizar esta clase, modificamos el iniciador de Flask.

```
Backend > app > create_app.py > create_app
1 from flask import Flask
2 from flask_cors import CORS
3 from flask_pymongo import PyMongo
4 from flask_jwt_extended import JWTManager
5 from config import Config
6
7 mongo = PyMongo()
8
9 def create_app():
10     app = Flask(__name__)
11     app.config.from_object(Config)
12
13     mongo.init_app(app)
14
15     CORS(app)
16
17     jwt = JWTManager(app)
18
19     from app import routes
20     app.register_blueprint(routes.bp)
21
22     return app
```

**Figura 15:** Configuración `__init__`.

Luego modificamos nuestros router, que maneja las rutas y lógica del negocio.

```
Backend > app > routes.py > get_resenas_espacio
1 from flask import Blueprint, request, jsonify
2 from flask_jwt_extended import create_access_token, jwt_required, get_jwt_identity
3 from app import mongo
4 from bson import ObjectId
5 from werkzeug.security import generate_password_hash, check_password_hash
6
7 bp = Blueprint('routes', __name__)
8
9 # Rutas de Usuario
10 @bp.route('/usuarios', methods=['POST'])
11 def create_usuario():
12     data = request.json
13     if mongo.db.usuarios.find_one({'email': data['email']}):
14         return jsonify({"msg": "El usuario ya existe"}), 400
15
16     # Cambiamos el método de hash a pbkdf2:sha256
17     hashed_password = generate_password_hash(data['contrasena'], method='pbkdf2:sha256')
18     data['contrasena'] = hashed_password
19     mongo.db.usuarios.insert_one(data)
20     return jsonify({"msg": "Usuario creado exitosamente"}), 201
21
```

**Figura 16:** Configuración de endpoints.

**Nota:** Los endpoints trabajados se verán en nuestras pruebas Postman o si gustan, podrán revisarlos en nuestro repositorio.

Luego, de configurar lo necesario de nuestro código, será necesario validar el backend los endpoint y su base de datos (MongoDB).

Para conseguir esto, evaluaremos mediante el uso de postman las peticiones de usuario.

## Creación de usuario

```
@bp.route('/usuarios', methods=['POST'])
def create_usuario():
    data = request.json
    if mongo.db.usuarios.find_one({'email': data['email']}):
        return jsonify({"msg": "El usuario ya existe"}), 400

    # Cambiamos el método de hash a pbkdf2:sha256
    hashed_password = generate_password_hash(data['contrasena'], method='pbkdf2:sha256')
    data['contrasena'] = hashed_password
    mongo.db.usuarios.insert_one(data)
    return jsonify({"msg": "Usuario creado exitosamente"}), 201
```

Figura 17: Endpoint registro

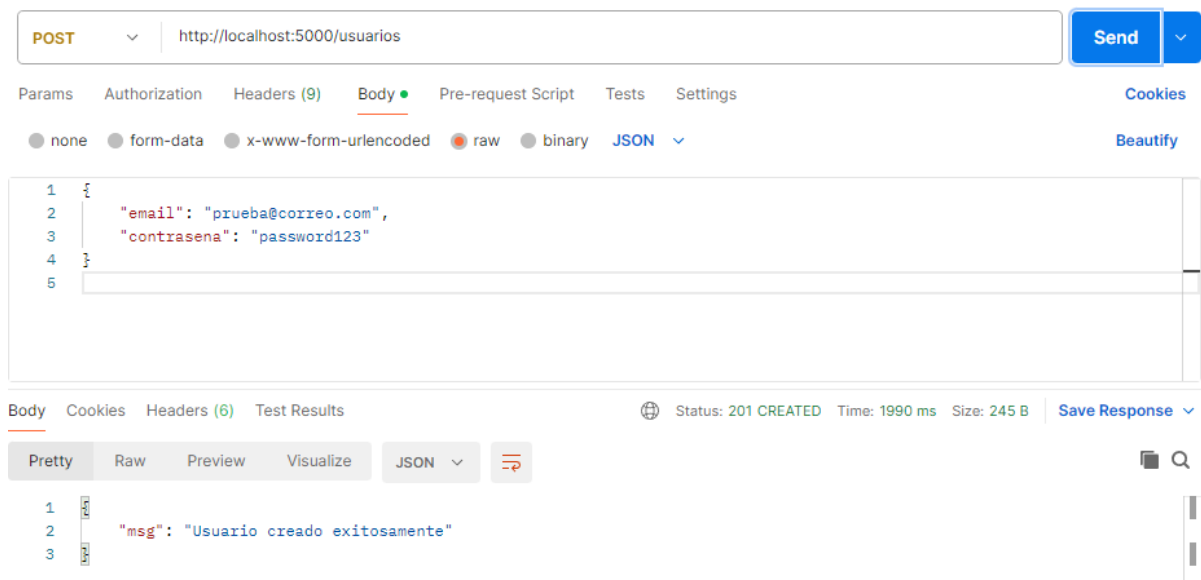


Figura 18: Creación de usuario.



Figura 19: Almacenamiento de datos usuarios.

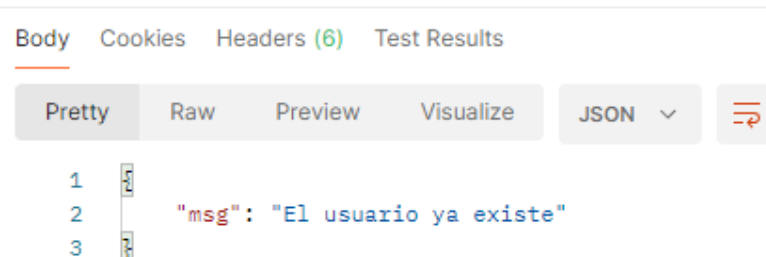


Figura 20: Usuario ya existente

## Verificación de Login

```
@bp.route('/login', methods=['POST'])
def login():
    data = request.json
    usuario = mongo.db.usuarios.find_one({'email': data['email']})
    if not usuario or not check_password_hash(usuario['contrasena'], data['contrasena']):
        return jsonify({"msg": "Credenciales incorrectas"}), 401
    access_token = create_access_token(identity=str(usuario['_id']))
    return jsonify({"msg": "Inicio de sesión exitoso", "access_token": access_token, "usuario_id": str(usuario['_id'])})
```

Figura 21: Endpoint Login

The screenshot shows a REST client interface with the following details:

- URL:** http://localhost:5000/login
- Method:** POST
- Request Body (JSON):**

```
{
  "email": "prueba@correo.com",
  "contrasena": "password123"
}
```
- Response Status:** 200 OK, Time: 800 ms, Size: 665 B
- Response Body (JSON):**

```
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmcmVzaCI6ZmFsc2UsIm1hdCI6MTcyNjc2ODcyNywiYWVhbnR5b2N0cyNyYyYyZjIiOiwiZmVzaCI6ImFjY2VzcyIsIn15IjY2ZWZmM2E2MDk1M2VmdmVzYzI0ODZjMSIsIm5iZiI6MTcyNjc2ODcyNywiY3NyZiI6ImEzNWQzYmVmLTYYMDItNDZhNy85ZjIjLjY2ZWZmM2E2MDk1M2VmdmVzYzI0ODZjMSIsImV4cCI6MTcyNjc2ODcyNyN30.RAjaQc7jP9vedrGPXqPMivCcNfdhHP_ljXuPeIkUx_A",
  "msg": "Inicio de sesión exitoso",
  "usuario id": "66ec3a60953e1460c8586c1"
}
```

Figura 22: Verificador de usuarios.

The screenshot shows a REST client interface with the following details:

- Response Body (JSON):**

```
{
  "msg": "Credenciales incorrectas"
}
```

Figura 23: Credenciales no encontradas

## Obtención de usuario por token

```

@bp.route('/perfil', methods=['GET'])
@jwt_required()
def get_perfil():
    usuario_id = get_jwt_identity()
    usuario = mongo.db.usuarios.find_one({'_id': ObjectId(usuario_id)}, {'contrasena': 0})
    if not usuario:
        return jsonify({"msg": "Usuario no encontrado"}), 404

    usuario['_id'] = str(usuario['_id'])
    return jsonify(usuario)

```

Figura 24: Perfil de usuario

http://localhost:5000/perfil

GET http://localhost:5000/perfil

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies

Headers 8 hidden

Key	Value	Bulk Edit
Authorization	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmcmVzaCI6ZmF...	
Key	Value	

Body Cookies Headers (6) Test Results Status: 200 OK Time: 48 ms Size: 269 B Save Response

Pretty Raw Preview Visualize JSON

```

1 {
2   "_id": "66ec63a60953ef460c8586c1",
3   "email": "prueba@correo.com"
4 }

```

Figura 25: Obtener usuario por token JWT y cabecera.

## Creación de Espacio

```

@bp.route('/espacios', methods=['POST'])
@jwt_required()
def create_espacio():
    data = request.json
    usuario_id = get_jwt_identity()

    espacio = {
        "nombre": data['nombre'],
        "ubicacion": data['ubicacion'],
        "descripcion": data['descripcion'],
        "precioPorHora": data['precioPorHora'],
        "usuario_id": ObjectId(usuario_id)
    }
    mongo.db.espacios.insert_one(espacio)
    return jsonify({"msg": "Espacio creado exitosamente"}), 201

```

Figura 26: Crear Espacios Disponibles

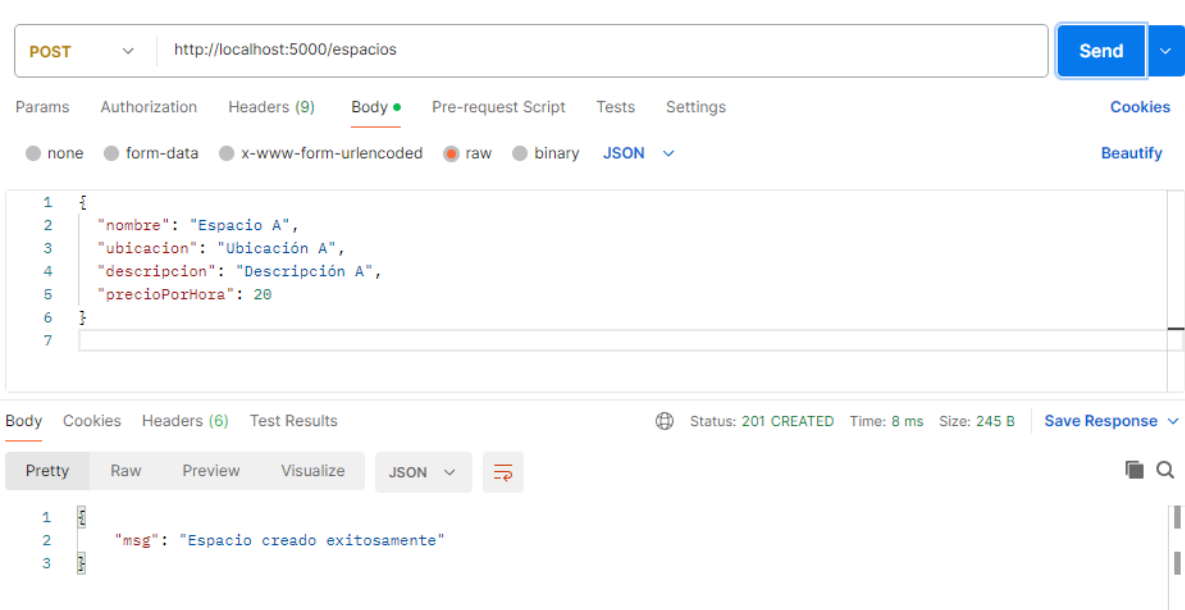


Figura 27: Creación de espacios.

```

_id: ObjectId('66ec6fad0953ef460c8586c2')
nombre : "Espacio A"
ubicacion : "Ubicación A"
descripcion : "Descripción A"
precioPorHora : 20
usuario_id : ObjectId('66ec63a60953ef460c8586c1')

```

Figura 28: Creación del registro espacios.

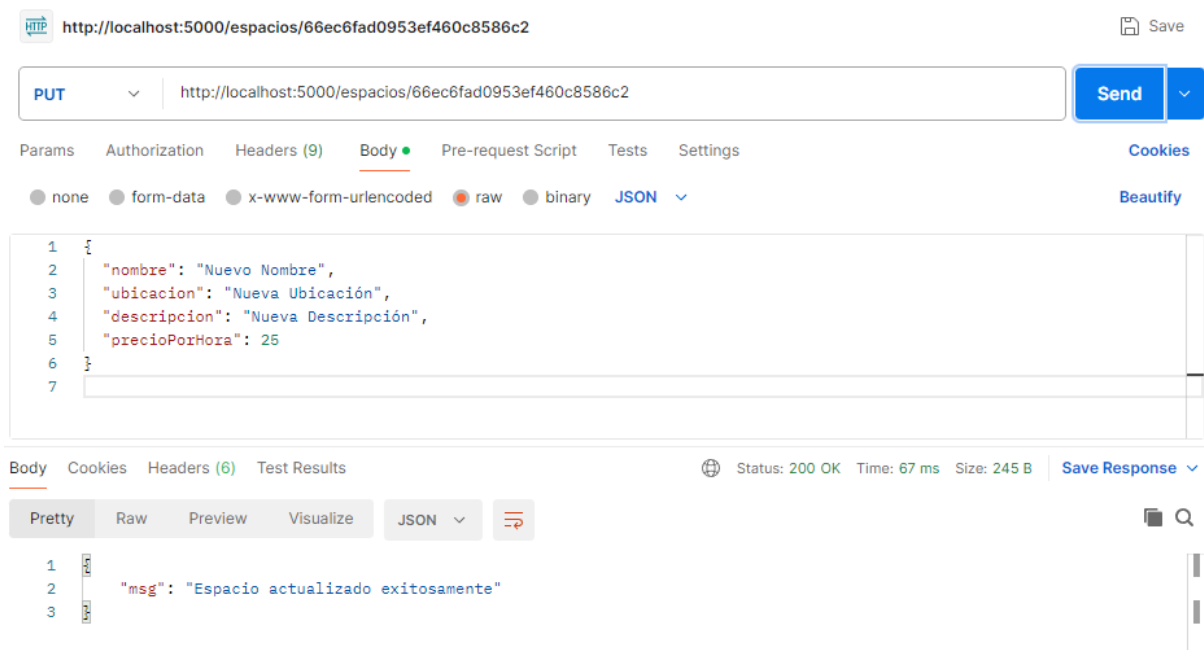
## Modificar Espacio

```

@bp.route('/espacios/<espacio_id>', methods=['PUT'])
@jwt_required()
def update_espacio(espacio_id):
    data = request.json
    result = mongo.db.espacios.update_one(
        {'_id': ObjectId(espacio_id)},
        {'$set': {
            'nombre': data['nombre'],
            'ubicacion': data['ubicacion'],
            'descripcion': data['descripcion'],
            'precioPorHora': data['precioPorHora']
        }}
    )
    if result.matched_count == 0:
        return jsonify({"msg": "Espacio no encontrado"}), 404
    return jsonify({"msg": "Espacio actualizado exitosamente"})

```

Figura 29: Actualizar Espacio



**Figura 30:** Actualización de espacio.

```

_id: ObjectId('66ec6fad0953ef460c8586c2')
nombre: "Nuevo Nombre"
ubicacion: "Nueva Ubicación"
descripcion: "Nueva Descripción"
precioPorHora: 25
usuario_id: ObjectId('66ec63a60953ef460c8586c1')

```

**Figura 31:** Registro de cambios en MongoDB

## Eliminar Espacio

```

@bp.route('/espacios/<espacio_id>', methods=['DELETE'])
@jwt_required()
def delete_espacio(espacio_id):
    result = mongo.db.espacios.delete_one({'_id': ObjectId(espacio_id)})
    if result.deleted_count == 0:
        return jsonify({"msg": "Espacio no encontrado"}), 404
    return jsonify({"msg": "Espacio eliminado exitosamente"})

```

**Figura 32:** Endpoints eliminar espacios.

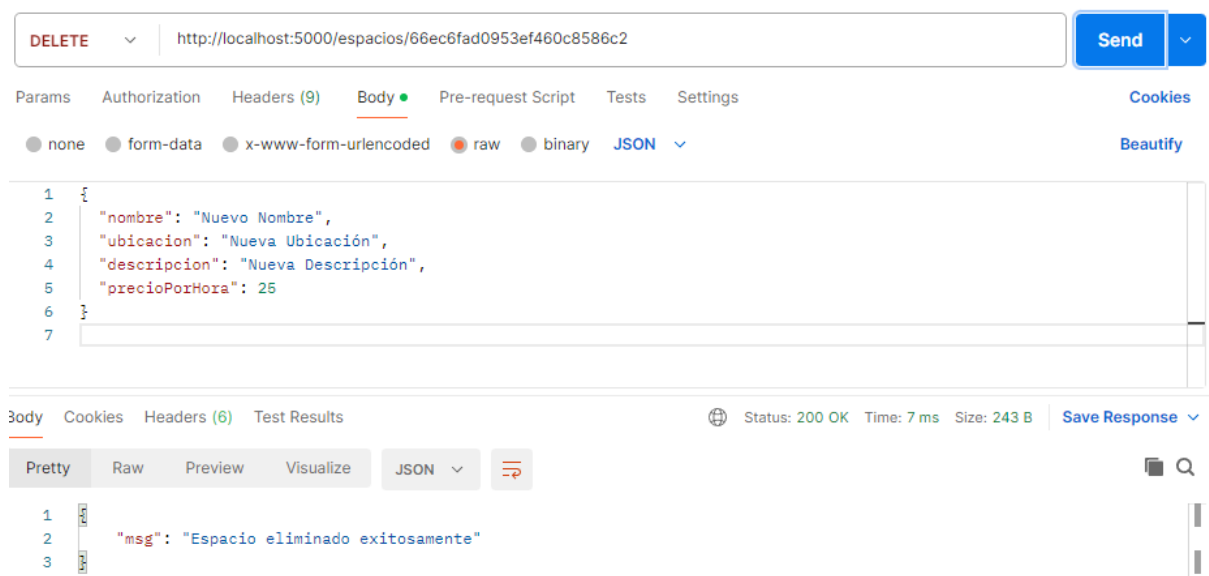


Figura 33: Eliminar registro

## Despliegue de los espacios adquiridos

```
@bp.route('/espacios', methods=['GET'])
def get_espacios():
    espacios = mongo.db.espacios.find()
    return jsonify([
        {
            "id": str(espacio['_id']),
            "nombre": espacio['nombre'],
            "ubicacion": espacio['ubicacion'],
            "descripcion": espacio['descripcion'],
            "precioPorHora": espacio['precioPorHora']
        } for espacio in espacios])
```

Figura 34: Obtener espacios de trabajo.

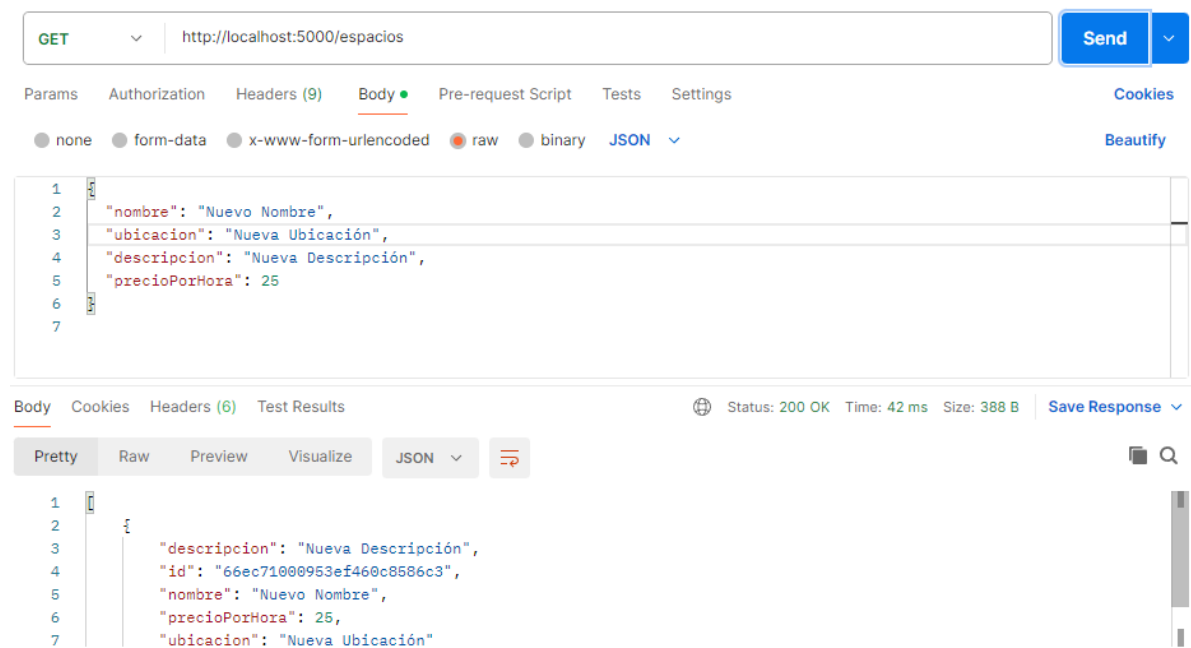


Figura 35: Obtención de espacios



## Creación de reservas

```
@bp.route('/reservas', methods=['POST'])
@jwt_required()
def create_reserva():
    data = request.json
    usuario_id = get_jwt_identity()
    espacio_nombre = data.get('espacio_nombre')
    espacio = mongo.db.espacios.find_one({'nombre': espacio_nombre})
    if not usuario_id or not espacio:
        return jsonify({"msg": "Usuario o espacio no especificado"}), 400

    reserva = {
        "fecha": data['fecha'],
        "horaInicio": data['horaInicio'],
        "horaFin": data['horaFin'],
        "usuario_id": ObjectId(usuario_id),
        "espacio_id": ObjectId(espacio['_id'])
    }
    mongo.db.reservas.insert_one(reserva)
    return jsonify({"msg": "Reserva creada exitosamente"}), 201
```

Figura 36: Crear Reserva

The screenshot shows a REST client interface with a POST request to `http://localhost:5000/reservas`. The request body is a JSON object with the following fields: `espacio_nombre` (Nuevo Nombre), `fecha` (2024-09-20), `horaInicio` (09:00), and `horaFin` (12:00). The response status is 201 CREATED, and the response body is a JSON object with the field `msg` (Reserva creada exitosamente).

POST `http://localhost:5000/reservas` Send

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary JSON Beautify

```
1 {
2   "espacio_nombre": "Nuevo Nombre",
3   "fecha": "2024-09-20",
4   "horaInicio": "09:00",
5   "horaFin": "12:00"
6 }
7
```

Body Cookies Headers (6) Test Results Status: 201 CREATED Time: 8 ms Size: 245 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "msg": "Reserva creada exitosamente"
3 }
```

Figura 37: Crear Reservas

## Reservación por usuario

```

@bp.route('/reservas/usuario/<usuario_id>', methods=['GET'])
def get_reservas_usuario(usuario_id):
    reservas = mongo.db.reservas.find({"usuario_id": ObjectId(usuario_id)})
    resultado = []

    for reserva in reservas:
        espacio = mongo.db.espacios.find_one({"_id": reserva["espacio_id"]})
        resultado.append({
            "id": str(reserva["_id"]),
            "fecha": reserva["fecha"],
            "horaInicio": reserva["horaInicio"],
            "horaFin": reserva["horaFin"],
            "espacio": {
                "id": str(espacio["_id"]),
                "nombre": espacio["nombre"],
                "ubicacion": espacio["ubicacion"],
                "descripcion": espacio["descripcion"],
                "precioPorHora": espacio["precioPorHora"]
            }
        })

    return jsonify(resultado)

```

**Figura 38 : Obtener reserva**

The screenshot shows a REST client interface with a GET request to `http://localhost:5000/reservas/usuario/66ec63a60953ef460c8586c1`. The response is a JSON object representing a reservation, displayed in a 'Pretty' format. The response status is 200 OK, with a response time of 7 ms and a size of 537 B.

```

{
  "espacio": {
    "descripcion": "Nueva Descripción",
    "id": "66ec71000953ef460c8586c3",
    "nombre": "Nuevo Nombre",
    "precioPorHora": 25,
    "ubicacion": "Nueva Ubicación"
  },
  "fecha": "2024-09-20",
  "horaFin": "12:00",
  "horaInicio": "09:00",
  "id": "66ec723a0953ef460c8586c4"
}

```

**Figura 39: Obtención de reservas.**

## Creación de Reseñas

```
# Rutas de Reseñas
@bp.route('/resenas', methods=['POST'])
@jwt_required()
def create_resena():
    data = request.json
    usuario_id = get_jwt_identity()
    espacio_nombre = data.get('espacio_nombre')
    if not espacio_nombre:
        return jsonify({"msg": "Falta el nombre del espacio"}), 400
    espacio = mongo.db.espacios.find_one({'nombre': espacio_nombre})
    if not espacio:
        return jsonify({"msg": "Espacio no encontrado"}), 404
    espacio_id = espacio['_id']
    reserva = mongo.db.reservas.find_one({
        "usuario_id": ObjectId(usuario_id),
        "espacio_id": ObjectId(espacio_id)
    })
    if not reserva:
        return jsonify({"msg": "No puedes reseñar un espacio que no has reservado"}), 403
    if 'comentario' not in data or 'calificacion' not in data:
        return jsonify({"msg": "Faltan campos en la reseña (comentario o calificación)"}), 400
    resena = {
        "comentario": data['comentario'],
        "calificacion": data['calificacion'],
        "usuario_id": ObjectId(usuario_id),
        "espacio_id": ObjectId(espacio_id)
    }
    result = mongo.db.resenas.insert_one(resena)
    if not result.inserted_id:
        return jsonify({"msg": "Error al insertar la reseña"}), 500
    return jsonify({"msg": "Reseña creada exitosamente"}), 201
```

Figura 40: Creación de reseñas

The screenshot shows a REST client interface with the following details:

- Method:** POST
- URL:** http://localhost:5000/resenas
- Body:** JSON format with the following content:

```
1 {
2   "espacio_nombre": "Nuevo Nombre",
3   "comentario": "Great place!",
4   "calificacion": 5
5 }
```
- Status:** 201 CREATED
- Time:** 8 ms
- Size:** 249 B
- Response Body:** JSON format with the following content:

```
1 {
2   "msg": "Reseña creada exitosamente"
3 }
```

Figura 41; Verificar creación

```

_id: ObjectId('66ec75ce0953ef460c8586c5')
comentario: "Great place!"
calificacion: 5
usuario_id: ObjectId('66ec63a60953ef460c8586c1')
espacio_id: ObjectId('66ec71000953ef460c8586c3')

```

**Figura 42:** Almacenador de Reseña.

### Obtener Reseña del Espacio

```

@bp.route('/resenas/espacio/<espacio_id>', methods=['GET'])
def get_resenas_espacio(espacio_id):
    resenas = mongo.db.resenas.find({"espacio_id": ObjectId(espacio_id)})
    espacio = mongo.db.espacios.find_one({"_id": ObjectId(espacio_id)}, {"nombre": 1})

    if not espacio:
        return jsonify({"msg": "Espacio no encontrado"}), 404

    resultado = []

    for resena in resenas:
        usuario = mongo.db.usuarios.find_one({"_id": resena["usuario_id"]}, {"email": 1})
        resultado.append({
            "id": str(resena['_id']),
            "comentario": resena['comentario'],
            "calificacion": resena['calificacion'],
            "usuario_email": usuario['email'],
            "nombre_espacio": espacio['nombre'] # Incluyendo el nombre del espacio
        })

    return jsonify(resultado)

```

**Figura 43:** Obtener reseña

The screenshot shows a REST client interface with a GET request to `http://localhost:5000/resenas/espacio/66ec71000953ef460c8586c3`. The response is a JSON object with the following structure:

```

{
  "calificacion": 5,
  "comentario": "Great place!",
  "id": "66ec75ce0953ef460c8586c5",
  "nombre_espacio": "Nuevo Nombre",
  "usuario_email": "prueba@correo.com"
}

```

Así comprobamos el funcionamiento de nuestros Endpoints y APIs. Solo nos faltaría validarlos mediante una interfaces para que el usuario lo consume.

### **Mejoras Futuras del Backend**

1. Para mejorar el proyecto, en el futuro se propone subir las APIs a un servidor en nube para mejorar el rendimiento y el consumo sea más sencillo.
2. Se propone que el proyecto separe los routers a futuro, para trabajarlos por servicio y solo se ocupen los endpoints necesarios.
3. Se propone desarrollar microservicios para trabajar mejor los servicios con el proyecto principal backend.
4. Validar los endpoints faltantes si se requiere.
5. Mejorar los usuario y el despliegue del perfil, para un control de Roles de usuario y además mostrar más datos a mostrar en el perfil.
6. Mejorar el despliegue de reseñas a los usuarios.