

Diplomado de actualización en nuevas tecnologías para el desarrollo de Software.

Taller Backend

Autor:

Jeason Steven Gelpud Cadena

Docente:

Vicente Aux Revelo

Universidad de Nariño

Facultad de Ingeniería

Ingeniería de Sistemas

Ipiales, Colombia

A continuación, se realiza el desarrollo del Taller Backend

Se crea una carpeta para trabajar el taller

```
jeison@LAPTOP-VFI2G2SH MINGW64 ~/documents/diplomado
$ mkdir TallerBackend
```

Iniciar el proyecto con node

```
jeison@LAPTOP-VFI2G2SH MINGW64 ~/documents/diplomado/TallerBackend
$ code .

jeison@LAPTOP-VFI2G2SH MINGW64 ~/documents/diplomado/TallerBackend
$ npm init -y
```

Crear un repositorio local para el TallerBackend:

- “TallerGit<Nombre>

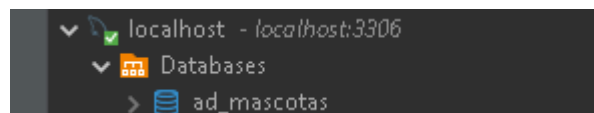
Se crea un repositorio local en el sistema de archivos con el nombre “TallerBackend”. En este repositorio, se desarrollará el código para el desarrollo del taller.

```
jeison@LAPTOP-VFI2G2SH MINGW64 ~/documents/diplomado/TallerBackend
$ git init
Initialized empty Git repository in C:/Users/jeiso/Documents/Diplomado/TallerBackend/.git/

jeison@LAPTOP-VFI2G2SH MINGW64 ~/documents/diplomado/TallerBackend (main)
$
```

1. Crear una base de datos

Se crea la base de datos llamada: **ad_mascotas**



2. Desarrollar una aplicación Backend implementada en NodeJS y ExpressJS

Para el desarrollo de este punto, lo primero es instalar las dependencias necesarias para el desarrollo del mismo.

- Instalar nodemon

```
PS C:\Users\jeiso\Documents\Diplomado\TallerBackend> npm install nodemon -D
```

- Instalar express

```
PS C:\Users\jeiso\Documents\Diplomado\TallerBackend> npm install express
```

- Instalar mysql2

```
PS C:\Users\jeiso\Documents\Diplomado\TallerBackend> npm install mysql2
```

- Instalar Sequelize

```
PS C:\Users\jeiso\Documents\Diplomado\TallerBackend> npm install sequelize
```

- Instalar cors

```
PS C:\Users\jeiso\Documents\Diplomado\TallerBackend> npm install cors
```

- Instalar bcrypt

```
PS C:\Users\jeiso\Documents\Diplomado\TallerBackend> npm install bcrypt
```

Una vez terminado el proceso de instalación revisamos las dependencias en el archivo package.js

```
},  
"dependencies": {  
  "bcrypt": "^5.1.1",  
  "cors": "^2.8.5",  
  "express": "^4.21.0",  
  "mysql2": "^3.11.3",  
  "sequelize": "^6.37.3"  
}  
}
```

- Tareas asociadas al registro y administración de las mascotas de la empresa *AdopMascotas*

Para este punto se considera que la empresa *AdopMascotas* contara con inicialmente con 4 tablas, considerando la ampliación de la empresa por ende una proyección en cuanto al crecimiento de la cantidad de tablas en la base de datos **ad_mascotas**:

- Mascotas
- Usuarios
- Solicitudes
- Adopciones

Modelos

Para realizar el proceso de creación de las tablas la base de datos, crea una carpeta llamada **modelos**, en el cual se realiza el modelo de cada una de las tablas utilizando a **Sequelize**, que es un **ORM (Object-Relational Mapping)** para Node.js que facilita la interacción con bases de datos.

Se crea la carpeta **modelos**

```
> node_modules
└─ src
   ├── database
   └── modelos
```

Ingresamos en la carpeta creada llamada **modelos** y creamos un archivo llamado `masctasModelo.js` la cual contine todos los atributos necesarios para nuestra tabla que se definió como “mascotas”

```
> modelos > JS mascotaModelo.js > Mascotas > espec
import Sequelize from "sequelize";
import { db } from "../database/conexion.js";

const Mascotas = db.define("mascotas", {
  id: {
    type: Sequelize.INTEGER,
    allowNull: false,
    autoIncrement: true,
    primaryKey: true
  },
  nombre: {
    type: Sequelize.STRING,
    allowNull: false
  },
  especie: {
    type: Sequelize.STRING,
    allowNull: false
  },
  raza: {
    type: Sequelize.STRING,
    allowNull: true
  },
  edad: {
    type: Sequelize.INTEGER,
    allowNull: false
  },
  descripcion: {
    type: Sequelize.TEXT,
    allowNull: true
  },
  estado_adopcion: {
    type: Sequelize.BOOLEAN,
    defaultValue: false
  },
  fecha_ingreso: {
    type: Sequelize.DATE,
    allowNull: false,
    defaultValue: Sequelize.NOW
  },
  imagen_url: {
    type: Sequelize.STRING,
    allowNull: true
  }
});

export { Mascotas };
```

Realizamos el mismo proceso para las tablas de: **Usuarios**, **Solicitudes**, **Adopciones**. Cabe aclarar que cada una de las tablas contiene sus correspondientes atributos necesarios para la recepción de la información necesaria para el funcionamiento de nuestro backend.

Tabla usuarios:

```
modelos > JS userModel.js > Usuarios > password
import Sequelize from "sequelize";
import { db } from "../database/conexion.js";

const Usuarios = db.define("usuarios", {
  id: {
    type: Sequelize.INTEGER,
    allowNull: false,
    autoIncrement: true,
    primaryKey: true
  },
  nombre: {
    type: Sequelize.STRING,
    allowNull: false
  },
  email: {
    type: Sequelize.STRING,
    allowNull: false,
    unique: true
  },
  password: {
    type: Sequelize.STRING,
    allowNull: false
  },
  telefono: {
    type: Sequelize.STRING,
    allowNull: true
  },
  direccion: {
    type: Sequelize.STRING,
    allowNull: true
  },
  fecha_registro: {
    type: Sequelize.DATE,
    allowNull: false,
    defaultValue: Sequelize.NOW
  }
});

export { Usuarios };
```

Tablas Aopciones:

```
modelos > JS adoptionModel.js > ...
import Sequelize from "sequelize";
import { db } from "../database/conexion.js";

const Adopciones = db.define("adopciones", {
  id: {
    type: Sequelize.INTEGER,
    allowNull: false,
    autoIncrement: true,
    primaryKey: true
  },
  usuarioId: {
    type: Sequelize.INTEGER,
    allowNull: false,
    references: {
      model: 'usuarios',
      key: 'id'
    }
  },
  mascotaId: {
    type: Sequelize.INTEGER,
    allowNull: false,
    references: {
      model: 'mascotas',
      key: 'id'
    }
  },
  fechaAdopcion: {
    type: Sequelize.DATE,
    allowNull: false,
    defaultValue: Sequelize.NOW
  }
});

export { Adopciones };
```

Tabla solicitudes:

```
modelos > JS requestModel.js > Solicitudes > estado
import Sequelize from "sequelize";
import { db } from "../database/conexion.js";

const solicitudes = db.define("solicitudes", {
  id: {
    type: Sequelize.INTEGER,
    allowNull: false,
    autoIncrement: true,
    primaryKey: true
  },
  usuarioId: {
    type: Sequelize.INTEGER,
    allowNull: false,
    references: {
      model: 'usuarios',
      key: 'id'
    }
  },
  mascotaId: {
    type: Sequelize.INTEGER,
    allowNull: false,
    references: {
      model: 'mascotas',
      key: 'id'
    }
  },
  fechaSolicitud: {
    type: Sequelize.DATE,
    allowNull: false,
    defaultValue: Sequelize.NOW
  },
  estado: {
    type: Sequelize.STRING,
    allowNull: false,
    defaultValue: 'pendiente' // 'pendiente', 'aprobada', 'rechazada'
  }
});

export { solicitudes };
```

A continuación se realiza el controlador para cada uno de los modelos que se crearon anteriormente, para poder utilizar esos modelos en los controladores se importa cada uno de los modelos en los controladores gracias a que se implementó **export** con el fin de que haya modularidad y sobre todo una buena organización .

Controladores

Controlador de usuarios:

Función crear: En esta captura inicial se realiza la función crear para registrar una nuevo, con la validación que los campos nombre, email y password estén presentes en la solicitud. Si falta alguno, devuelve un error 400.

```
const crear = async (req, res) => {  
  const { nombre, email, password, telefono, direccion } = req.body;  
  
  // Validar datos  
  if (!nombre || !email || !password) {  
    return res.status(400).json({ mensaje: "Nombre, email y contraseña son obligatorios." });  
  }  
}
```

Se utiliza bcrypt para encriptar la contraseña antes de guardarla en la base de datos, lo que es una buena práctica para la seguridad.

```
const hashedPassword = await bcrypt.hash(password, 10);
```

Es importante resaltar que se utiliza **await** lo cual indica que la ejecución debe esperar a que la operación de creación sea completada antes de continuar. Esto es importante porque las operaciones de base de datos son asíncronas.

Luego se preparan los datos creando un objeto que contiene los datos necesarios para crear una nuevo usuario.

```
const nuevoUsuario = await Usuarios.create({  
  nombre,  
  email,  
  password: hashedPassword,  
  telefono,  
  direccion  
});
```

Cuando el registro es exitoso, devuelve un mensaje de éxito junto con los detalles del nuevo usuario, de lo contrario captura cualquier error que ocurra durante la creación del usuario y se devuelve un mensaje.

```
res.status(201).json({ mensaje: "Usuario creado con éxito", usuario: nuevoUsuario });  
} catch (err) {  
  res.status(500).json({ mensaje: `Error al crear el usuario: ${err}` });  
}
```

Función BuscarId: La función buscarId busca un usuario en la base de datos utilizando el ID proporcionado en la solicitud. Si el ID es válido y se encuentra el usuario, devuelve la información del usuario. Si el ID está vacío o no se encuentra el usuario, devuelve un mensaje de error correspondiente. Si ocurre un error durante la búsqueda, también se maneja y se devuelve un mensaje de error.

```
const buscarId = async (req, res) => {  
  const { id } = req.params;  
  
  if (!id) {  
    return res.status(400).json({ mensaje: "El ID no puede estar vacío." });  
  }  
  
  try {  
    const usuario = await Usuarios.findById(id);  
    if (!usuario) {  
      return res.status(404).json({ mensaje: "Usuario no encontrado." });  
    }  
    res.status(200).json(usuario);  
  } catch (err) {  
    res.status(500).json({ mensaje: `Error al buscar el usuario: ${err}` });  
  }  
};
```

Función Buscar La función buscar obtiene todos los usuarios de la base de datos. Si la operación es exitosa, devuelve la lista de usuarios con un código de estado 200. Si ocurre un error durante la búsqueda, se captura y se devuelve un mensaje de error con un código de estado 500.

```
const buscar = async (req, res) => {  
  try {  
    const usuarios = await Usuarios.findAll();  
    res.status(200).json(usuarios);  
  } catch (err) {  
    res.status(500).json({ mensaje: `Error al buscar usuarios: ${err}` });  
  }  
};
```

Función Actualizar:

La función actualizar se encarga de modificar los datos de un usuario en la base de datos. Primero, extrae el id del usuario y los nuevos datos (nombre, email, teléfono, dirección) de la solicitud. Luego, valida que el id esté presente y que al menos uno de los campos a actualizar no esté vacío; si no se cumplen estas condiciones, devuelve un error 400. A continuación, utiliza Usuarios.update() para realizar la actualización en la base de datos, y si es exitosa, envía un mensaje de éxito con un código 200. En caso de error durante la

operación, captura el problema y responde con un mensaje de error y un código 500, asegurando así que se gestionen adecuadamente tanto las validaciones como los errores.

```
const actualizar = async (req, res) => {
  const { id } = req.params;
  const { nombre, email, telefono, direccion } = req.body;

  if (!id || (!nombre && !email && !telefono && !direccion)) {
    return res.status(400).json({ mensaje: "Datos insuficientes para actualizar." });
  }

  try {
    await Usuarios.update({ nombre, email, telefono, direccion }, { where: { id } });
    res.status(200).json({ mensaje: "Usuario actualizado con éxito." });
  } catch (err) {
    res.status(500).json({ mensaje: `Error al actualizar el usuario: ${err}` });
  }
};
```

Función eliminar: La función `eliminar` se encarga de eliminar un usuario de la base de datos. Primero, extrae el `id` del usuario desde los parámetros de la solicitud y verifica que no esté vacío; si lo está, devuelve un error 400. Luego, intenta eliminar el usuario utilizando `Usuarios.destroy()` y, si la operación no encuentra ningún registro para eliminar, responde con un error 404 indicando que el usuario no fue encontrado. Si la eliminación se realiza con éxito, envía un mensaje de confirmación con un código 200. En caso de que ocurra un error durante el proceso, captura el problema y devuelve un mensaje de error con un código 500, asegurando así una correcta gestión de las operaciones y los errores.

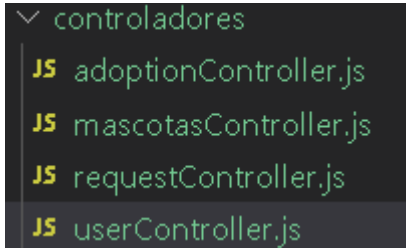
```
const eliminar = async (req, res) => {
  const { id } = req.params;

  if (!id) {
    return res.status(400).json({ mensaje: "El ID no puede estar vacío." });
  }

  try {
    const resultado = await Usuarios.destroy({ where: { id } });
    if (!resultado) {
      return res.status(404).json({ mensaje: "Usuario no encontrado." });
    }
    res.status(200).json({ mensaje: `Usuario con ID ${id} eliminado con éxito.` });
  } catch (err) {
    res.status(500).json({ mensaje: `Error al eliminar el usuario: ${err}` });
  }
};
```


Controladores de Mascotas, Solicitudes, adopciones

Una vez terminado el controlador para usuarios, procedemos a la creación de los controladores para las mascotas, adopciones y las solicitudes, ya que cada uno cuenta con su CRUD para el manejo de la información y el procedimiento de creación de los controladores es el mismo se adjunta la captura de la carpeta controladores el cual contiene los controladores ya realizados.



Rutas

Se procede a crear una carpeta llamada **rutas**, en esta se definen los enrutadores de Express para el manejo de solicitudes HTTP con las operaciones relacionadas a los usuarios.

En esta carpeta se crea el archivo **user.js**, en el cual se realiza lo siguiente:

Importación de Módulos: Se importan las dependencias necesarias, incluyendo Express y las funciones del controlador de usuarios (crear, buscar, buscarId, actualizar, eliminar).

```
import express from "express";
import {crear, buscar, buscarId, actualizar, eliminar} from "../controladores/userController.js";
```

Definición del Router: Se crea un objeto `users` utilizando `express.Router()`, que se encargará de gestionar las rutas relacionadas con los usuarios.

```
const users = express.Router();
```

Rutas Definidas:

- **GET /:** Responde con un mensaje de saludo cuando se accede a la raíz del enrutador.

```
users.get('/', (req, res) => {
  res.send('Hola Sitio usuario');
});
```

- **POST /crear:** Invoca la función `crear` para procesar la creación de un nuevo usuario.

```
users.post('/crear', (req, res) => {  
  //res.send('Crear usuario');  
  crear(req,res);  
});
```

- **GET /buscar:** Invoca la función `buscar` para obtener todos los usuarios.

```
users.get('/buscar', (req, res) => {  
  //res.send('Buscar usuario');  
  buscar(req,res);  
});
```

- **GET /buscarId/:id:** Usa `buscarId` para obtener un usuario específico basado en el ID proporcionado en la URL.

```
users.get('/buscarId/:id', (req, res) => {  
  //res.send('Buscar usuario');  
  buscarId(req,res);  
});
```

- **PUT /actualizar/:id:** Llama a la función `actualizar` para modificar un usuario existente según el ID.

```
users.put('/actualizar/:id', (req, res) => {  
  //res.send('Actualizar usuario');  
  actualizar(req,res);  
});
```

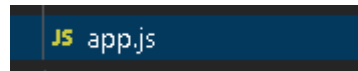
- **DELETE /eliminar/:id:** Llama a la función `eliminar` para borrar un usuario utilizando el ID.

```
users.delete('/eliminar/:id', (req, res) => {  
  //res.send('eliminar usuario');  
  eliminar(req,res);  
});
```

- **Exportación del Router:** Finalmente, se exporta el objeto `users` para que pueda ser utilizado en

```
export {users}
```

Ahora se configura un servidor Express para manejar un sistema de adopción de mascotas en el archivo *app.js*



En este archivo se realiza lo siguiente:

Importaciones: Se importan los módulos necesarios, incluyendo Express, CORS, la conexión a la base de datos y los enrutadores para diferentes recursos (mascotas, adopciones, empresa, solicitudes y usuarios).

```
import express from "express";
import './modelos/relationsModelo.js';
import { routerMascotas } from './rutas/mascotasRouter.js';
import { adoptions } from './rutas/adoptions.js';
import { company } from './rutas/company.js';
import { requestsM } from './rutas/requestsM.js';
import { users } from './rutas/users.js';
import { db } from './database/conexion.js';
import cors from "cors";
```

Instancia de Express: Se crea una instancia de la aplicación Express llamada app.

```
//Crear instancia de Express
const app = express();
```

Configuración de CORS: Se habilita CORS para permitir solicitudes desde diferentes orígenes, lo que es útil para aplicaciones web.

```
//Cors
app.use(cors());
```

Middleware para JSON: Se configura el middleware `express.json()` para que la aplicación pueda recibir y procesar solicitudes JSON.

```
//Middleware para JSON
app.use(express.json());
```

Conexión a la Base de Datos: Se verifica la conexión a la base de datos usando `db.authenticate()`. Si la conexión es exitosa, se imprime un mensaje en la consola; de lo contrario, se maneja el error.

```
//Verificar Conexion Base Datos
db.authenticate().then(()=>{
  console.log(`Conexion a Base de datos correcta`);
}).catch(err=>{
  console.log(`Conexion a Base de datos incorrecta ${err}`);
});
```

Definición de Rutas:

Se establece una ruta raíz (/) que responde con un mensaje de bienvenida.

```
app.get('/', (req, res) => {
  res.send('Bienvenido al sitio de adopción de mascotas');
});
```

Se integran diferentes enrutadores para gestionar las operaciones CRUD de cada recurso (mascotas, adopciones, empresa, solicitudes y usuarios) a través de las rutas correspondientes.

```
app.use("/mascotas",routerMascotas);
app.use("/adopciones",adoptions);
app.use("/empresa",company);
app.use("/solicitudes",requestsM);
app.use("/usuarios",users);
```

Puerto del Servidor: Se define el puerto en el que se ejecutará el servidor (4000).

Sincronización con la Base de Datos: Se utiliza db.sync() para sincronizar el modelo de datos con la base de datos. Si es exitoso, se inicia el servidor y se imprime un mensaje en la consola indicando que el servidor está en funcionamiento; si hay un error, se maneja adecuadamente.

```
db.sync().then(()=>{
  //Abri servicio e iniciar el Servidor
  app.listen(PORT,()=>{
    console.log(`Servidor Inicializado en el puerto ${PORT}`);
  })
}).catch(err=>{
  console.log(`Error al Sincronizar base de datos ${err}`);
});
```

Se inicia el se inicia el servicio con el siguiente comando:

Una vez iniciado ya se crea las tablas en nuestra base de datos, con los modelos que se creó para cada una de ellas.

3. Realizar verificación de las diferentes operaciones

Usuarios

- **POST:** Se utiliza para enviar datos al servidor, en este caso para crear un nuevo usuario el cual se ha creado de manera correcta.

Base de datos : ad mascotas

	id	nombre	email	password	telefono	direccion	fecha_registro	crea
ar	1	Julian	julian@hotmail	\$2b\$10\$8qsal7KetL53iKizmaJ5ek3NYZ.7ekNtNrpWYyG2UDs...	NULL	NULL	2024-09-30 16:33:52	2024

id	nombre	email	password	telefono	direccion	fecha_registro
1	Julian	julian@hotmail	\$2b\$10\$8qsaI7KetL53jKizmavJ5ek3NYZ.7eKntNrpWyG2UDs...	NULL	NULL	2024-09-30 16:33:52
2	Stiventiven	Stiven@hotmail	\$2b\$10\$bh7heZMz2oSABNu2n2l.kuWZTg2j.PdGlrH47e3CDK/...	NULL	NULL	2024-09-30 16:37:36

Usuarios/buscar

GET: Este método se utiliza para solicitar datos del servidor. En este caso, se está buscando información sobre los usuarios en el endpoint /usuarios/buscar, el cual se ha ejecutado con éxito.

GET	http://127.0.0.1:4000/usuarios/buscar	Send	Status: 200 OK	Size: 566 Bytes	Time: 48 ms
Query	Headers ²	Auth	Body	Tests	Pre Run
Query Parameters			Response	Headers ⁸	Cookies
<input type="checkbox"/> parameter			Results		
value			Docs		
			4 "nombre": "Julian",		
			5 "email": "julian@hotmail",		
			6 "password": "\$2b\$10\$8qsaI7KetL53jKizmavJ5ek3NYZ.7eKntNrpWyG2UDs32AvNLWxZW",		
			7 "telefono": null,		
			8 "direccion": null,		
			9 "fecha_registro": "2024-09-30T16:33:52.000Z",		
			10 "createdAt": "2024-09-30T16:33:52.000Z",		
			11 "updatedAt": "2024-09-30T16:33:52.000Z",		
			12 },		
			13 {		
			14 "id": 2,		
			15 "nombre": "Stiventiven",		
			16 "email": "Stiven@hotmail",		

Usuario/actualizar

PUT: Este método se utiliza para actualizar un recurso existente. En este caso, se está actualizando un usuario con el ID 2, la ejecución de muestra con éxito.

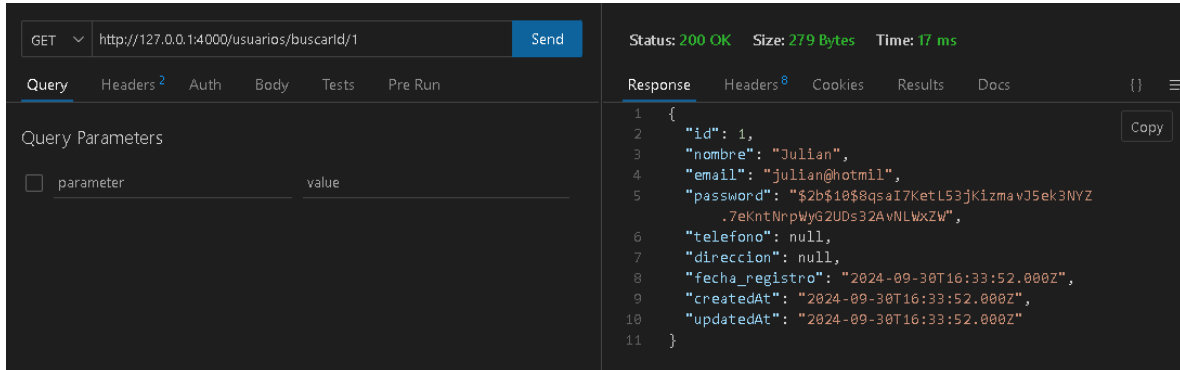
PUT	http://127.0.0.1:4000/usuarios/actualizar/2	Send	Status: 200 OK	Size: 45 Bytes	Time: 73 ms
Query	Headers ²	Auth	Body ¹	Tests	Pre Run
JSON	XML	Text	Form	Form-encode	GraphQL
JSON Content			Binary		
Format			Response		
1 {			1 {		
2 "nombre": "Jeason",			2 "mensaje": "Usuario actualizado con éxito."		
3 "email": "Stiven@hotmail",			3 }		
4 "password": "0000"					
5 }					
6 }					
7 }					
8 }					

Base de datos : ad_mascotas

id	nombre	email	password	telefono	direccion	fecha_registro
1	Julian	julian@hotmail	\$2b\$10\$8qsaI7KetL53jKizmavJ5ek3NYZ.7eKntNrpWyG2UDs...	NULL	NULL	2024-09-30 16:33:52
2	Jeason	Stiven@hotmail	\$2b\$10\$bh7heZMz2oSABNu2n2l.kuWZTg2j.PdGlrH47e3CDK/...	NULL	NULL	2024-09-30 16:37:36

Usuario/buscarId

GET: Este método se utiliza para solicitar datos del servidor. En este caso, se está buscando información sobre un usuario específico. El cual muestra una ejecución con éxito



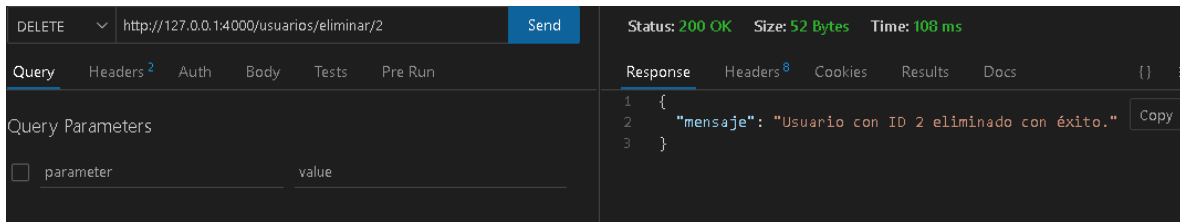
The screenshot shows a REST client interface with the following details:

- Method:** GET
- URL:** http://127.0.0.1:4000/usuarios/buscarId/1
- Status:** 200 OK
- Size:** 279 Bytes
- Time:** 17 ms
- Response Body (JSON):**

```
{
  "id": 1,
  "nombre": "Julian",
  "email": "julian@hotmail",
  "password": "$2b$10$8qsa17KetL53jKizmavJ5ek3NYZ.7eKntNrpWyG2UDs32AvNLwxZW",
  "telefono": null,
  "direccion": null,
  "fecha_registro": "2024-09-30T16:33:52.000Z",
  "createdAt": "2024-09-30T16:33:52.000Z",
  "updatedAt": "2024-09-30T16:33:52.000Z"
}
```

Usuario/eliminar

DELETE: Este método se utiliza para eliminar un recurso. Aquí se está eliminando el usuario con el ID 2.



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

- Method:** DELETE
- URL:** http://127.0.0.1:4000/usuarios/eliminar/2
- Status:** 200 OK
- Size:** 52 Bytes
- Time:** 108 ms
- Response Body (JSON):**

```
{
  "mensaje": "Usuario con ID 2 eliminado con éxito."
}
```

La implantación y su descripción para cada uno de los verbos de cada tabla es el mismo. Por ende, se procede colocar las capturas de la verificación de cada una.

Solicitudes

Solicitudes/crear

POST

http://127.0.0.1:4000/solicitudes/crear

Send

Status: 201 Created

Size: 233 Bytes

Time: 86 ms

Query

Headers²

Auth

Body¹

Tests

Pre Run

JSON

XML

Text

Form

Form-encode

GraphQL

Binary

JSON Content

Format

1

2

3

4

5

6

7

8

1

2

3

4

5

6

7

8

9

10

11

12

```
{
  "usuarioId": 1,
  "mascotaId": 2
}
```

```
{
  "mensaje": "Solicitud creada con \u00e9xito",
  "solicitud": {
    "fechaSolicitud": "2024-09-30T16:53:31.496Z",
    "estado": "pendiente",
    "id": 1,
    "usuarioId": 1,
    "mascotaId": 2,
    "updatedAt": "2024-09-30T16:53:31.499Z",
    "createdAt": "2024-09-30T16:53:31.499Z"
  }
}
```

Solicitudes/buscar

GET

http://127.0.0.1:4000/solicitudes/buscar

Send

Status: 200 OK

Size: 217 Bytes

Time: 19 ms

Query

Headers²

Auth

Body

Tests

Pre Run

Query Parameters

☐

parameter

value

1

2

3

4

5

6

7

8

9

10

11

12

13

1

2

3

4

5

6

7

8

9

10

11

12

13

GET

http://127.0.0.1:4000/solicitudes/buscarId/1

Send

Status: 200 OK

Size: 215 Bytes

Time: 18 ms

Query

Headers²

Auth

Body

Tests

Pre Run

Query Parameters

☐

parameter

value

1

2

3

4

5

6

7

8

9

10

11

1

2

3

4

5

6

7

8

9

10

11

Solicitudes/actualizar/1

PUT

http://127.0.0.1:4000/solicitudes/actualizar/1

Send

Status: 200 OK

Size: 47 Bytes

Time: 128 ms

Query

Headers²

Auth

Body¹

Tests

Pre Run

JSON

XML

Text

Form

Form-encode

GraphQL

Binary

JSON Content

Format

1 {

2 "usuarioId":1,

3 "mascotaId":2,

4 "estado": "rechazada"

5

6

7 }

Response

Headers⁸

Cookies

Results

Docs

1 {

2 "mensaje": "Solicitud actualizada con éxito."

3 }

id	usuarioId	mascotaId	fechaSolicitud	estado	createdAt	updatedAt	id_usuario	id_mascota
1	1	2	2024-09-30 16:53:31	aprobada	2024-09-30 16:53:31	2024-09-30 17:01:04	NULL	NULL

id	usuarioId	mascotaId	fechaSolicitud	estado	createdAt	updatedAt	id_usuario	id_mascota
1	1	2	2024-09-30 16:53:31	rechazada	2024-09-30 16:53:31	2024-09-30 17:02:39	NULL	NULL

Solicitudes/eliminar

DELETE

http://127.0.0.1:4000/solicitudes/eliminar/1

Send

Status: 200 OK

Size: 54 Bytes

Time: 91 ms

Query

Headers²

Auth

Body

Tests

Pre Run

Query Parameters

☐ parameter

value

Response

Headers⁸

Cookies

Results

Docs

1 {

2 "mensaje": "Solicitud con ID 1 eliminada con éxito."

3 }

Adopciones

Adopciones/crear

POST

http://127.0.0.1:4000/adopciones/crear

Send

Status: 201 Created

Size: 210 Bytes

Time: 286 ms

Query

Headers²

Auth

Body¹

Tests

Pre Run

JSON

XML

Text

Form

Form-encode

GraphQL

Binary

JSON Content

Format

1 {

2 "usuarioId":1,

3 "mascotaId":2

4

5

6

7

8 }

Response

Headers⁸

Cookies

Results

Docs

1 {

2 "mensaje": "Adopción creada con éxito",

3 "adopcion": {

4 "fechaAdopcion": "2024-09-30T17:21:32.906Z",

5 "id": 1,

6 "usuarioId": 1,

7 "mascotaId": 2,

8 "updatedAt": "2024-09-30T17:21:32.909Z",

9 "createdAt": "2024-09-30T17:21:32.909Z"

10 }

11 }

Adopciones/buscar

GEThttp://127.0.0.1:4000/adopciones/buscarSend

QueryHeadersAuthBodyTestsPre Run

Query Parameters

☐

parametervalue

Status: 200 OKSize: 195 BytesTime: 54 ms

ResponseHeadersCookiesResultsDocs

```
1 [
2   {
3     "id": 1,
4     "usuarioId": 1,
5     "mascotaId": 2,
6     "fechaAdopcion": "2024-09-30T17:21:32.000Z",
7     "createdAt": "2024-09-30T17:21:32.000Z",
8     "updatedAt": "2024-09-30T17:21:32.000Z",
9     "id_usuario": null,
10    "id_mascota": null
11  }
12 ]
```

Adopciones/actualizar

PUThttp://127.0.0.1:4000/adopciones/actualizar/1Send

QueryHeadersAuthBodyTestsPre Run

JSONContentFormat

```
1 {
2   "usuarioId":1,
3   "mascotaId":2
4 }
5
6 
```

Status: 200 OKSize: 47 BytesTime: 55 ms

ResponseHeadersCookiesResultsDocs

```
1 {
2   "mensaje": "Adopci3n actualizada con 3xito."
3 }
```

Adopciones/buscar

GEThttp://127.0.0.1:4000/adopciones/buscarSend

QueryHeadersAuthBodyTestsPre Run

Query Parameters

☐

parametervalue

Status: 200 OKSize: 389 BytesTime: 14 ms

ResponseHeadersCookiesResultsDocs

```
1 [
2   {
3     "id": 1,
4     "usuarioId": 1,
5     "mascotaId": 2,
6     "fechaAdopcion": "2024-09-30T17:21:32.000Z",
7     "createdAt": "2024-09-30T17:21:32.000Z",
8     "updatedAt": "2024-09-30T18:16:00.000Z",
9     "id_usuario": null,
10    "id_mascota": null
11  },
12  {
13    "id": 2,
14    "usuarioId": 1,
```

Adopciones/eliminar/1

DELETEhttp://127.0.0.1:4000/adopciones/eliminar/1Send

QueryHeadersAuthBodyTestsPre Run

Query Parameters

☐

parametervalue

Status: 200 OKSize: 54 BytesTime: 45 ms

ResponseHeadersCookiesResultsDocs

```
1 {
2   "mensaje": "Adopci3n con ID 1 eliminada con 3xito."
3 }
```

	id	usuarioId	mascotaId	fechaAdopcion	createdAt	updatedAt	id_usuario	id_mascota
	2	1	2	2024-09-30 18:13:59	2024-09-30 18:13:59	2024-09-30 18:13:59	NULL	NULL

Mascotas

Mascotas/crear

The screenshot shows a REST client interface with a POST request to `http://127.0.0.1:4000/mascotas/crear`. The request body is a JSON object: `{ "nombre": "Fideo", "especie": "Perro", "edad": 6 }`. The response status is `201 Created` with a size of `51 Bytes` and a time of `220 ms`. The response body is a JSON object: `{ "mensaje": "Registro de Mascota Creado con Éxito" }`.

```
POST http://127.0.0.1:4000/mascotas/crear
```

JSON Content

```
{  "nombre": "Fideo",  "especie": "Perro",  "edad": 6}
```

Status: 201 Created Size: 51 Bytes Time: 220 ms

Response

```
{  "mensaje": "Registro de Mascota Creado con Éxito"}
```

Mascotas/buascar

The screenshot shows a REST client interface with a GET request to `http://127.0.0.1:4000/mascotas/buascar`. The response status is `200 OK` with a size of `248 Bytes` and a time of `26 ms`. The response body is a JSON object: `{ "id": 1, "nombre": "Fideo", "especie": "Perro", "raza": null, "edad": 6, "descripcion": null, "estado_adopcion": false, "fecha_ingreso": "2024-09-30T14:59:19.000Z", "imagen_url": null }`.

```
GET http://127.0.0.1:4000/mascotas/buascar
```

Query Parameters

parameter	value
-----------	-------

Status: 200 OK Size: 248 Bytes Time: 26 ms

Response

```
{  "id": 1,  "nombre": "Fideo",  "especie": "Perro",  "raza": null,  "edad": 6,  "descripcion": null,  "estado_adopcion": false,  "fecha_ingreso": "2024-09-30T14:59:19.000Z",  "imagen_url": null}
```

Mascotas/Actualizar/2

New Request

ActivityCollectionsEnv

filter activity

PUTjust now

GET127.0.0.1:4000/m...13 mins ago

POST127.0.0.1:4000/...5 mins ago

PUT

http://127.0.0.1:4000/mascotas/actualizar/2

Send

QueryHeaders²AuthBody¹TestsPre Run

JSONXMLTextFormForm-encodeGraphQLBi

1{

2"nombre": "Max",

3"especie": "Gato",

4"edad": 6

5

6}

Status: 200 OKSize: 34 BytesTime: 81 msResponse

1{

2"mensaje": "Registro Actualizado"

3}

Base de datos: ad_mascotas

id	nombre	especie	raza	edad	descripcion	estado_adopcion	fecha_ingreso	imagen
1	Fideo	Perro	NULL	6	NULL	0	2024-09-30 14:59:19	NULL
2	Max	Gato	NULL	6	NULL	0	2024-09-30 15:12:48	NULL

id	nombre	especie	raza	edad	descripcion	estado_adopcion	fecha_ingreso	imagen
1	Fideo	Perro	NULL	6	NULL	0	2024-09-30 14:59:19	NULL
2	Max	Perro	NULL	6	NULL	0	2024-09-30 15:12:48	NULL

Mascotas/buscarId/2

The screenshot shows a REST client interface with a 'New Request' button and a dropdown menu. The request is a GET to 'http://127.0.0.1:4000/mascotas/buscarId/2'. The 'Query' tab is active, showing 'Query Parameters' with a table for parameter and value. The status bar indicates 'Status: 200 OK', 'Size: 243 Bytes', and 'Time: 24 ms'. The response is a JSON object with the following details:

parameter	value
id	2
nombre	Max
especie	Gato
raza	null
edad	6
descripcion	null
estado_adopcion	false
fecha_ingreso	2024-09-30T15:12:48.000Z
imagen_url	null

Mascotas/eliminar/1

The screenshot shows a REST client interface with a 'New Request' button and a dropdown menu. The request is a DELETE to 'http://127.0.0.1:4000/mascotas/eliminar/1'. The 'Query' tab is active, showing 'Query Parameters' with a table for parameter and value. The status bar indicates 'Status: 200 OK', 'Size: 55 Bytes', and 'Time: 330 ms'. The response is a JSON object with the following details:

parameter	value
mensaje	Registro con ID 1 Eliminado Correctamente

	id	nombre	especie	raza	edad	descripcion	estado
<input type="checkbox"/>	2	Max	Gato	NULL	6	NULL	