INFORME DESARROLLO DE UNA API REST FULL

AUTOR:

JAVIER NICOLAS SALAS

YESI ESTEBAN PANTOJA CUELLAR

ENTREGADO A:

ING. BRAYAN IGNACIO ARCOS BURBANO

INSTITUTO TECNOLÓGICO DEL PUTUMAYO

TECNOLOGÍA EN DESARROLLO DE SOFTWARE

QUINTO SEMESTRE

PROGRAMACIÓN BACKEND

MOCOA – PUTUMAYO

2024

Contenido

Repositorio De GitHub:

Resumen Ejecutivo

Este informe ofrece una visión detallada del proceso de desarrollo de una API RESTful destinada a la gestión integral de un complejo deportivo. En él, se aborda el diseño de la API, la estructura de los endpoints, las consultas realizadas y el proceso de identificación de recursos y relaciones entre ellos. Además, se incluye el modelo de arquitectura de la API, las metodologías HTTP empleadas y el proceso de autenticación y autorización, proporcionando una explicación lógica y coherente de cada etapa del desarrollo realizado. Este documento busca servir como una guía clara y comprensible para quienes deseen entender el funcionamiento y la organización de la API implementada.

Introducción

Contexto y Motivación

El principal motivador para nuestro equipo es el avance en el desarrollo de una API RESTful para un proyecto personal que estamos llevando a cabo. Nuestro objetivo es aplicar los conocimientos adquiridos en el desarrollo de servicios web y crear una solución de alta calidad que nos permita avanzar de manera significativa en nuestros proyectos. Este proceso no solo nos brinda la oportunidad de consolidar nuestras habilidades técnicas, sino que también nos inspira a superar desafíos y a innovar en la gestión de datos y la interacción con sistemas externos para lograr resultados óptimos.

Alcance del Informe

El informe cubre los siguientes aspectos del desarrollo de una API RESTful:
• Diseño y arquitectura de la API.
• Implementación de endpoints y métodos HTTP.
• Gestión de recursos, relaciones y autenticación.
• Proceso de pruebas y validación de la API.

Objetivos del Proyecto

El objetivo principal de este proyecto es crear una plataforma versátil y segura que permita:

1. Gestionar usuarios con operaciones CRUD completas.
2. Almacenar y manipular información detallada de personas.
3. Implementar un sistema de roles flexible y potente.
4. Manejar datos adicionales como tipos de identificación y direcciones.

Componentes Principales

El sistema se dividirá en los siguientes componentes clave:

1. Gestión de Usuarios:

- Implementación de CRUD completo para la tabla de usuarios.
- Manejo de credenciales (correo y contraseña).

2. Gestión de Personas:

- Almacenamiento de información personal vinculada a los usuarios.
- CRUD completo para la tabla de 'people'.

3. Sistema de Roles:

- CRUD completo para la tabla de roles.
- Funcionalidades adicionales como cálculo de total de roles y búsqueda por nombre.

4. Gestión de Roles de Usuario:

- Implementación de la tabla 'userRoles' con CRUD completo.
- Endpoints para actualizar múltiples roles de usuarios y obtener información específica.

5. Funcionalidades Adicionales:

- CRUD para tipos de identificación y direcciones.
- Endpoints especializados para consultas y análisis de datos.

Desarrollo del Informe

Estructura del Desarrollo

El desarrollo se organizará en módulos, cada uno centrado en una entidad o funcionalidad específica. Se seguirá un enfoque de desarrollo incremental, comenzando con las funcionalidades básicas de CRUD y avanzando hacia operaciones más complejas y especializadas.

Consideraciones Técnicas

- Se implementarán validaciones robustas en todas las operaciones CRUD.
- Se prestará especial atención a la seguridad en el manejo de datos sensibles de usuarios.
- Se optimizarán las consultas para un rendimiento eficiente, especialmente en operaciones que involucren múltiples tablas.

Este proyecto no solo proporcionará una base sólida para la gestión de usuarios y roles, sino que también ofrecerá funcionalidades avanzadas para el análisis y manipulación de datos relacionados con las personas y sus características.

A medida que avancemos en el desarrollo, cada componente se construirá sobre los anteriores, asegurando una integración fluida y una funcionalidad coherente en todo el sistema.

Primeros pasos

La URL base para todos los endpoints de la API es: http://localhost:3000/api/

Crear la tabla de usuarios y operaciones crud

1. Crear el Modelo de User

Paso 1

Primero, define el modelo de usuario en TypeScript. Esto representa la estructura de la tabla en la base de datos.

Archivo Typescript (.ts)

```
// src/models/userModel.ts

import { Model, DataTypes } from 'sequelize';
import sequelize from '../config/database';
import IdentificationType from './identificationTypeModel';
import Address from './addressModel';
```

```
class User extends Model {
  public id!: number;
  public email!: string;
  public password!: string;
```

```
  // Timestamps
  public readonly createdAt!: Date;
  public readonly updatedAt!: Date;
}
```

```
User.init(
  {
    id: {
      type: DataTypes.INTEGER.UNSIGNED,
      autoIncrement: true,
      primaryKey: true,
    },
    email: {
```

```
            type: DataTypes.STRING,
            unique: true,
            allowNull: false,
        },
        password: {
            type: DataTypes.STRING,
            allowNull: false,
        },
    },
    {
        tableName: 'users',
        sequelize, // passing the `sequelize` instance is required,
        timestamps: true, // Habilitar timestamps automáticos
    }
);
```

```
// Modelo People
class People extends Model {
    public id!: number;
```

```
    public userId!: number;
    public names!: string;
    public lastNames!: string;
    public firstName!: string;
    public lastName!: string;
    public birthDate!: Date;
    public identificationTypeId!: number;
    public identificationNumber!: string;
```

```
    // Timestamps
    public readonly createdAt!: Date;
    public readonly updatedAt!: Date;
}
```

```
People.init(
    {
        id: {
            type: DataTypes.INTEGER.UNSIGNED,
            autoIncrement: true,
            primaryKey: true,
        },
        userId: {
            type: DataTypes.INTEGER.UNSIGNED,
            allowNull: false,
            references: {
                model: User, // Hace referencia al modelo `User`
                key: 'id', // La columna `id` de `User`
            },
        },
```

```
      names: {
        type: DataTypes.STRING,
        allowNull: false,
      },
      lastNames: {
        type: DataTypes.STRING,
        allowNull: false,
      },
      firstName: {
        type: DataTypes.STRING,
        allowNull: false,
      },
      lastName: {
        type: DataTypes.STRING,
        allowNull: false,
      },
      birthDate: {
        type: DataTypes.DATE,
        allowNull: false,
      },
      identificationTypeId: {
        type: DataTypes.INTEGER.UNSIGNED,
        allowNull: false,
        references: {
          model: IdentificationType,
          key: 'id',
        },
      },
      identificationNumber: {
        type: DataTypes.STRING,
        allowNull: false,
      },
    },
    {
      tableName: 'people',
      sequelize, // passing the `sequelize` instance is required,
      timestamps: true, // Habilitar timestamps automáticos
    }
);
```

```
// Relaciones
People.belongsTo(User, { foreignKey: 'userId', as: 'user' });
People.belongsTo(IdentificationType, { foreignKey: 'identificationTypeId', as:
'identificationType' });
People.hasMany(Address, { foreignKey: 'personId', as: 'addresses' });
```

```
export { User, People };
```

Paso 2: Crear los Endpoints en el Controlador

Crea el controlador de usuarios con las operaciones CRUD necesarias.

Crear Usuario

Archivo Typescript (.ts)

```
// src/controllers/createUser.ts

import { Request, Response } from 'express';
import { CreateUserSql } from '../infra/createUser';

const createUserSql = new CreateUserSql();

export const createUser = async (req: Request, res: Response): Promise<void> => {
    try {
        const user = await createUserSql.createUserSql(req.body);
        res.status(201).json({ user });
    } catch (error) {
        res.status(500).json({ error: 'Error creating user' });
    }
};
```

Service:

```
// src/services/createUserService.ts

import { User } from '../models/userModel';

export const createUserService = async (data: Partial<User>): Promise<User> => {
    // Validación de los datos
    if (!data.email || !data.password) {
        throw new Error('Email and password are required');
    }
    // Creación del usuario
    const user = await User.create(data);
    return user;
};
```

Eliminar Usuario

Archivo Typescript (.ts)

```
// src/controllers/deleteUser.ts

import { Request, Response } from 'express';
import { User } from '../models/userModel';
```

```
export const deleteUser = async (req: Request, res: Response): Promise<void> => {
  try {
    const user = await User.findByPk(req.params.id);
    if (user) {
      await user.destroy();
      res.status(200).json({ message: 'User deleted successfully' });
    } else {
      res.status(404).json({ error: 'User not found' });
    }
  } catch (error) {
    res.status(500).json({ error: 'Error deleting user' });
  }
};
```

Service:

```
// src/services/deleteUserService.ts

import { User } from '../models/userModel';
```

```
export const deleteUserService = async (id: number): Promise<boolean> => {
  // Obtener el usuario existente
  const user = await User.findByPk(id);
  if (user) {
    // Eliminar el usuario
    await user.destroy();
    return true;
  }
  return false;
};
```

Obtener Usuario

Archivo Typescript (.ts)

```
// src/controllers/getUser.ts

import { Request, Response } from 'express';
import { User } from '../models/userModel';
```

```
export const getUser = async (req: Request, res: Response): Promise<void> => {
  try {
    const user = await User.findByPk(req.params.id);
    if (user) {
      res.status(200).json({ user });
    } else {
      res.status(404).json({ error: 'User not found' });
    }
  } catch (error) {
```

```
        res.status(500).json({ error: 'Error fetching user' });
    }
};
```

Service:

```
// src/services/getUserService.ts

import { User } from '../models/userModel';
```

```
export const getUserService = async (id: number): Promise<User | null> => {
    // Obtener un usuario por ID
    const user = await User.findByPk(id);
    return user;
};
```

Obtener usuarios

Archivo Typescript (.ts)

```
// src/controllers/getUsers.ts

import { Request, Response } from 'express';
import { User } from '../models/userModel';
```

```
export const getUsers = async (_: Request, res: Response): Promise<void> => {
    try {
        const users = await User.findAll();
        res.status(200).json({ users });
    } catch (error) {
        res.status(500).json({ error: 'Error fetching users' });
    }
};
```

Service:

```
// src/services/getUsersService.ts

import { User } from '../models/userModel';
```

```
export const getUsersService = async (): Promise<User[]> => {
    // Obtener todos los usuarios
    const users = await User.findAll();
    return users;
};
```

Actualizar Usuario

Archivo Typescript (.ts)

```
// src/controllers/updateUser.ts

import { Request, Response } from 'express';
import { User } from '../models/userModel';

export const updateUser = async (req: Request, res: Response): Promise<void> => {
    try {
        const user = await User.findByPk(req.params.id);
        if (user) {
            await user.update(req.body);
            res.status(200).json({ user });
        } else {
            res.status(404).json({ error: 'User not found' });
        }
    } catch (error) {
        res.status(500).json({ error: 'Error updating user' });
    }
};
```

Service:

```
// src/services/loginUserService.ts

import { User } from '../models/userModel';
import bcrypt from 'bcrypt';
import jwt from 'jsonwebtoken';

export const loginUserService = async ({ email, password }: { email: string; password: string }):
Promise<string | null> => {
    try {
        const user = await User.findOne({ where: { email } });
        if (!user) {
            throw new Error('User not found');
        }

        const isMatch = await bcrypt.compare(password, user.password);
        if (!isMatch) {
            throw new Error('Invalid password');
        }

        const token = jwt.sign({ id: user.id }, process.env.JWT_SECRET || 'secretKey', { expiresIn:
'1h' });
        return token;
    } catch (error) {
        console.error('Error logging in user:', error);
```

```
        return null;
    }
};
```

Rutas:

```
import { Router } from 'express';
import { createUser } from '../controllers/createUser';
import { getUsers } from '../controllers/getUsers';
import { getUser } from '../controllers/getUser';
import { updateUser } from '../controllers/updateUser';
import { deleteUser } from '../controllers/deleteUser';

const router: Router = Router();
```

```
router.post('/users', createUser);
router.get('/users', getUsers);
router.get('/users/:id', getUser);
router.put('/users/:id', updateUser);
router.delete('/users/:id', deleteUser);
```

```
export default router;
```

Funciones:

- **createUser.ts**: Permite crear un nuevo usuario en la base de datos.
- **getUsers.ts**: Recupera todos los usuarios registrados.
- **getUser.ts**: Recupera un usuario específico mediante su ID.
- **updateUser.ts**: Actualiza la información de un usuario existente.
- **deleteUser.ts:** Eliminar usuario de la base de datos.

Crear la Tabla de Personas y Operaciones CRUD

2. Crear el Modelo de People

Paso 1

Define el modelo de persona en TypeScript. Esto representa la estructura de la tabla en la base de datos.

Archivo Typescript (.ts)

```
// src/models/peopleModel.ts

import { Model, DataTypes } from 'sequelize';
import sequelize from '../config/database';
import { User } from './userModel';
import IdentificationType from './identificationTypeModel';
import Address from './addressModel';
```

```
// Modelo People
class People extends Model {
  public id!: number;
```

```
  public userId: number;
```

```
  public names!: string;
```

```
  public lastNames!: string;
```

```
  public firstName!: string;
  public lastName!: string;
  public birthDate!: Date;
  public identificationTypeId!: number;
  public identificationNumber!: string;
```

```
  // Relaciones
  public identificationType?: IdentificationType | null;
  public readonly createdAt!: Date;
```

```
  public readonly updatedAt!: Date;
}
```

```
People.init(
  {
    id: {
      type: DataTypes.INTEGER.UNSIGNED,
      autoIncrement: true,
```

```
        primaryKey: true,
    },
    userId: {
        type: DataTypes.INTEGER.UNSIGNED,
        allowNull: false,
        references: {
            model: User, // Hace referencia al modelo `User`
            key: 'id', // La columna `id` de `User`
        },
    },
    names: {
        type: DataTypes.STRING,
        allowNull: false,
    },
    lastNames: {
        type: DataTypes.STRING,
        allowNull: false,
    },
    firstName: {
        type: DataTypes.STRING,
        allowNull: false,
    },
    lastName: {
        type: DataTypes.STRING,
        allowNull: false,
    },
    birthDate: {
        type: DataTypes.DATE,
        allowNull: false,
    },
    identificationTypeId: {
        type: DataTypes.INTEGER.UNSIGNED,
        allowNull: false,
        references: {
            model: IdentificationType, // Referencia a IdentificationType
            key: 'id',
        },
    },
    identificationNumber: {
        type: DataTypes.STRING,
        allowNull: false,
    },
},
{
    tableName: 'people',
    sequelize, // passing the `sequelize` instance is required,
    timestamps: true, // Habilitar timestamps automáticos
},
```

```
);
```

```
// Relaciones
People.belongsTo(User, { foreignKey: 'userId', as: 'user' });
People.belongsTo(IdentificationType, { foreignKey: 'identificationTypeId', as:
'identificationType' });
People.hasMany(Address, { foreignKey: 'personId', as: 'addresses' });
```

```
export default People;
```

Paso 2: Crear los Endpoints en el Controlador People

Crea el controlador de personas con las operaciones CRUD necesarias.

Crear Persona
Archivo Typescript (.ts)

```
// src/controllers/createPerson.ts

import { Request, Response } from 'express';
import People from '../models/peopleModel';
```

```
export const createPerson = async (req: Request, res: Response): Promise<void> => {
    try {
        const { firstName, lastName, birthDate, identificationTypeId, identificationNumber } =
req.body;
        const person = await People.create({ firstName, lastName, birthDate, identificationTypeId,
identificationNumber });
        res.status(201).json({ person });
    } catch (error) {
        res.status(500).json({ error: 'Error creating person' });
    }
};
```

Service:

```
// src/services/createPersonService.ts

import People from '../models/peopleModel';
```

```
export const createPersonService = async (firstName: string, lastName: string, birthDate: Date,
identificationTypeId: number, identificationNumber: string): Promise<People> => {
    const person = await People.create({ firstName, lastName, birthDate, identificationTypeId,
identificationNumber });
    return person;
};
```

Obtener Todas las Personas
Archivo Typescript (.ts)

```
// src/controllers/getPeople.ts

import { Request, Response } from 'express';
import People from '../models/peopleModel';
```

```
export const getPeople = async (_: Request, res: Response): Promise<void> => {
    try {
        const people = await People.findAll();
        res.status(200).json({ people });
    } catch (error) {
        res.status(500).json({ error: 'Error fetching people' });
    }
};
```

Service:

```
// src/services/getPeopleService.ts

import People from '../models/peopleModel';
```

```
// Función existente
export const getPeopleService = async (): Promise<People[]> => {
    const people = await People.findAll();
    return people;
};
```

```
// Función adicional para obtener una persona por ID
export const getPersonService = async (id: number): Promise<People | null> => {
    const person = await People.findByPk(id);
    return person;
};
```

```
// Función adicional para crear una persona
export const createPersonService = async (firstName: string, lastName: string, birthDate: Date,
identificationTypeId: number, identificationNumber: string): Promise<People> => {
    const person = await People.create({ firstName, lastName, birthDate, identificationTypeId,
identificationNumber });
    return person;
};
```

```
// Función adicional para actualizar una persona
export const updatePersonService = async (id: number, firstName: string, lastName: string,
birthDate: Date, identificationTypeId: number, identificationNumber: string): Promise<People |
null> => {
    const person = await People.findByPk(id);
    if (person) {
        await person.update({ firstName, lastName, birthDate, identificationTypeId,
identificationNumber });
        return person;
    }
```

```
    return null;
};
```

```
// Función adicional para eliminar una persona
export const deletePersonService = async (id: number): Promise<void> => {
    const person = await People.findByPk(id);
    if (person) {
        await person.destroy();
    }
};
```

Obtener Persona por ID
Archivo Typescript (.ts)

```
// src/controllers/getPerson.ts

import { Request, Response } from 'express';
import People from '../models/peopleModel';
```

```
export const getPerson = async (req: Request, res: Response): Promise<void> => {
    try {
        const person = await People.findByPk(req.params.id);
        if (person) {
            res.status(200).json({ person });
        } else {
            res.status(404).json({ error: 'Person not found' });
        }
    } catch (error) {
        res.status(500).json({ error: 'Error fetching person' });
    }
};
```

Service:

```
// src/services/deletePersonService.ts

import People from '../models/peopleModel';
```

```
export const deletePersonService = async (id: number): Promise<void> => {
    const person = await People.findByPk(id);
    if (person) {
        await person.destroy();
    }
};
```

Actualizar Persona

Archivo Typescript (.ts)

```
// src/controllers/updatePerson.ts

import { Request, Response } from 'express';
import People from '../models/peopleModel';
```

```
export const updatePerson = async (req: Request, res: Response): Promise<void> => {
    try {
        const id = parseInt(req.params.id, 10);
        const { firstName, lastName, birthDate, identificationTypeId, identificationNumber } =
req.body;
        const person = await People.findByPk(id);
        if (person) {
            await person.update({ firstName, lastName, birthDate, identificationTypeId,
identificationNumber });
            res.status(200).json({ person });
        } else {
            res.status(404).json({ error: 'Person not found' });
        }
    } catch (error) {
        res.status(500).json({ error: 'Error updating person' });
    }
};
```

Service:

```
// src/services/updatePersonService.ts

import People from '../models/peopleModel';
```

```
export const updatePersonService = async (id: number, firstName: string, lastName: string,
birthDate: Date, identificationTypeId: number, identificationNumber: string): Promise<People |
null> => {
    const person = await People.findByPk(id);
    if (person) {
        await person.update({ firstName, lastName, birthDate, identificationTypeId,
identificationNumber });
        return person;
    }
    return null;
};
```

Eliminar Persona
Archivo Typescript (.ts)

```
// src/controllers/deletePerson.ts
```

```
import { Request, Response } from 'express';
import People from '../models/peopleModel';
```

```
export const deletePerson = async (req: Request, res: Response): Promise<void> => {
    try {
        const id = parseInt(req.params.id, 10);
        const person = await People.findByPk(id);
        if (person) {
            await person.destroy();
            res.status(204).send();
        } else {
            res.status(404).json({ error: 'Person not found' });
        }
    } catch (error) {
        res.status(500).json({ error: 'Error deleting person' });
    }
};
```

Service:

```
// src/services/deletePersonService.ts

import People from '../models/peopleModel';
```

```
export const deletePersonService = async (id: number): Promise<void> => {
    const person = await People.findByPk(id);
    if (person) {
        await person.destroy();
    }
};
```

Obtener Información Completa del Usuario

Archivo Typescript (.ts)

```
// src/controllers/getUserFullInfo.ts

import { Request, Response } from 'express';
import { getUserFullInfoService } from '../services/getUserFullInfoService';
```

```
export const getUserFullInfo = async (req: Request, res: Response): Promise<void> => {
    try {
        const person = await getUserFullInfoService(parseInt(req.params.id, 10));
```

```
        if (person) {
            res.status(200).json({ person });
        } else {
            res.status(404).json({ error: 'Person not found' });
        }
```

```
    } catch (error) {
        res.status(500).json({ error: 'Error fetching user information' });
    }
};
```

Service:

```
// src/services/getUserFullInfoService.ts

import People from '../models/peopleModel';
import { User } from '../models/userModel';
import IdentificationType from '../models/identificationTypeModel';
import Address from '../models/addressModel';
```

```
export const getUserFullInfoService = async (id: number) => {
    const person = await People.findOne({
        where: { id },
        include: [
            { model: User, as: 'user' },
            { model: IdentificationType, as: 'identificationType' },
            { model: Address, as: 'addresses' }
        ]
    });
    return person;
};
```

Obtener Total de Personas

Archivo Typescript (.ts)

```
// src/controllers/getTotalPeople.ts

import { Request, Response } from 'express';
import { getTotalPeopleService } from '../services/getTotalPeopleService';
```

```
export const getTotalPeople = async (_: Request, res: Response): Promise<void> => {
    try {
        const data = await getTotalPeopleService();
        res.status(200).json(data);
    } catch (error) {
        res.status(500).json({ error: 'Error fetching total people' });
    }
};
```

Service:

```
// src/services/getTotalPeopleService.ts

import People from '../models/peopleModel';
```

```
export const getTotalPeopleService = async () => {
  const people = await People.findAll({
    attributes: ['firstName', 'lastName'],
  });
  const total = people.length;
  const names = people.map(person => `${person.firstName} ${person.lastName}`);
  return { total, names };
};
```

Obtener Personas por Criterios

Archivo Typescript (.ts)

```
// src/controllers/getPeopleByCriteria.ts

import { Request, Response } from 'express';
import { getPeopleByCriteriaService } from '../services/getPeopleByCriteriaService';
```

```
export const getPeopleByCriteria = async (req: Request, res: Response): Promise<void> => {
  try {
    const name = req.query.name ? String(req.query.name) : undefined;
    const city = req.query.city ? String(req.query.city) : undefined;
```

```
    const people = await getPeopleByCriteriaService({ name, city });
    res.status(200).json({ people });
  } catch (error) {
    res.status(500).json({ error: 'Error fetching people by criteria' });
  }
};
```

Service:

```
// src/services/getPeopleByCriteriaService.ts

import { Op } from 'sequelize';
import People from '../models/peopleModel';
import Address from '../models/addressModel';
```

```
interface Criteria {
  name?: string;
  city?: string;
}
```

```
export const getPeopleByCriteriaService = async (criteria: Criteria) => {
  const { name, city } = criteria;
```

```
  const people = await People.findAll({
    where: {
      ...(name && {
```

```
            [Op.or]: [
              { firstName: { [Op.like]: `%${name}%` } },
              { lastName: { [Op.like]: `%${name}%` } },
            ]
          }),
          ...(city && {
            '$addresses.city$': city
          })
        },
        include: [
          {
            model: Address,
            as: 'addresses',
            attributes: ['city']
          }
        ]
    });
```

```
    return people;
};
```

Obtener Personas por Ciudad

Archivo Typescript (.ts)

```
// src/controllers/getPeopleByCity.ts

import { Request, Response } from 'express';
import { getPeopleByCityService } from '../services/getPeopleByCityService';
```

```
export const getPeopleByCity = async (req: Request, res: Response): Promise<void> => {
    try {
        const city = req.query.city ? String(req.query.city) : undefined;
```

```
        if (!city) {
            res.status(400).json({ error: 'City parameter is required' });
            return;
        }
```

```
        const people = await getPeopleByCityService(city);
        res.status(200).json({ people });
    } catch (error) {
        res.status(500).json({ error: 'Error fetching people by city' });
    }
};
```

Service:

```
// src/services/getPeopleByCityService.ts

import People from '../models/peopleModel';
import Address from '../models/addressModel';

export const getPeopleByCityService = async (city: string) => {
    const people = await People.findAll({
        include: [
            {
                model: Address,
                as: 'addresses',
                where: { city },
            }
        ]
    });

    return people;
};
```

Actualizar Dirección y Analizar Datos

Archivo Typescript (.ts)

```
// src/controllers/updatePersonAddressAndAnalyze.ts

import { Request, Response } from 'express';
import { updatePersonAddressAndAnalyzeService } from
'../services/updatePersonAddressAndAnalyzeService';

export const updatePersonAddressAndAnalyze = async (req: Request, res: Response):
Promise<void> => {
    try {
        const id = parseInt(req.params.id, 10);
        const { street, city, state, zipCode, country } = req.body;

        if (!id || !street || !city || !state || !zipCode || !country) {
            res.status(400).json({ error: 'All address fields are required' });
            return;
        }

        const data = await updatePersonAddressAndAnalyzeService(id, { street, city, state, zipCode,
country });
        res.status(200).json(data);
    } catch (error) {
        res.status(500).json({ error: 'Error updating address and analyzing data' });
    }
};
```

Service:

```
// src/services/updatePersonAddressAndAnalyzeService.ts

import People from '../models/peopleModel';
import Address from '../models/addressModel';

interface AddressData {
    street: string;
    city: string;
    state: string;
    zipCode: string;
    country: string;
}

export const updatePersonAddressAndAnalyzeService = async (id: number, addressData:
AddressData) => {
    const person = await People.findByPk(id);

    if (!person) {
        throw new Error('Person not found');
    }

    // Actualizar dirección
    await Address.update(addressData, {
        where: { personId: id }
    });

    // Análisis de datos: contar cuántas personas viven en la misma ciudad
    const count = await Address.count({
        where: { city: addressData.city }
    });

    return { message: 'Address updated successfully', peopleInSameCity: count };
};
```

Obtener Personas en Ciudad

Archivo Typescript (.ts)

```
// src/controllers/getPeopleInCity.ts

import { Request, Response } from 'express';
import { getPeopleInCityService } from '../services/getPeopleInCityService';

export const getPeopleInCity = async (req: Request, res: Response): Promise<void> => {
    try {
```

```
    const city = req.query.city ? String(req.query.city) : undefined;

    if (!city) {
      res.status(400).json({ error: 'City parameter is required' });
      return;
    }

    const people = await getPeopleInCityService(city);
    res.status(200).json({ people });
  } catch (error) {
    res.status(500).json({ error: 'Error fetching people in city' });
  }
};
```

Service:

```
// src/services/getPeopleInCityService.ts

import People from '../models/peopleModel';
import Address from '../models/addressModel';

export const getPeopleInCityService = async (city: string) => {
  const people = await People.findAll({
    include: [
      {
        model: Address,
        as: 'addresses',
        where: { city }
      }
    ]
  });

  return people;
};
```

Buscar Persona por Identificación

Archivo Typescript (.ts)

```
// src/controllers/getPersonById.ts

import { Request, Response } from 'express';
import { getPersonByIdService } from '../services/getPersonByIdService';

export const getPersonById = async (req: Request, res: Response): Promise<void> => {
  try {
    const { identificationTypeId, identificationNumber } = req.query;

    if (!identificationTypeId || !identificationNumber) {
```

```
            res.status(400).json({ error: 'Identification type and number are required' });
            return;
        }

        const person = await getPersonByIdService(Number(identificationTypeId),
String(identificationNumber));
        if (person) {
            res.status(200).json({ person });
        } else {
            res.status(404).json({ error: 'Person not found' });
        }
    } catch (error) {
        res.status(500).json({ error: 'Error fetching person by identification' });
    }
};
```

Service:

```
// src/services/getPersonByIdService.ts

import People from '../models/peopleModel';

export const getPersonByIdService = async (identificationTypeId: number, identificationNumber:
string) => {
    const person = await People.findOne({
        where: {
            identificationTypeId,
            identificationNumber
        }
    });

    return person;
};
```

Obtener Estadísticas de Personas

Archivo Typescript (.ts)

```
// src/controllers/getPeopleStatistics.ts

import { Request, Response } from 'express';
import { getPeopleStatisticsService } from '../services/getPeopleStatisticsService';

export const getPeopleStatistics = async (_: Request, res: Response): Promise<void> => {
    try {
        const stats = await getPeopleStatisticsService();
        res.status(200).json(stats);
    } catch (error) {
        res.status(500).json({ error: 'Error fetching people statistics' });
```

```
    }
};
```

Service:

```
// src/services/getPeopleStatisticsService.ts

import People from '../models/peopleModel';
import { Op } from 'sequelize';

export const getPeopleStatisticsService = async () => {
    const currentDate = new Date();
    const adultAge = new Date(currentDate.setFullYear(currentDate.getFullYear() - 18));

    const total = await People.count();
    const adults = await People.count({ where: { birthDate: { [Op.lte]: adultAge } } });
    const minors = total - adults;

    return {
        total,
        adults,
        minors
    };
};
```

Obtener Total de Identificaciones

Archivo Typescript (.ts)

```
// src/controllers/getTotalIdentifications.ts

import { Request, Response } from 'express';
import { getTotalIdentificationsService } from '../services/getTotalIdentificationsService';

export const getTotalIdentifications = async (_: Request, res: Response): Promise<void> => {
    try {
        const data = await getTotalIdentificationsService();
        res.status(200).json(data);
    } catch (error) {
        res.status(500).json({ error: 'Error fetching total identifications' });
    }
};
```

Service:

```
// src/services/getTotalIdentificationsService.ts

import { fn, col } from 'sequelize';
import People from '../models/peopleModel';
```

```
import IdentificationType from '../models/identificationTypeModel';
```

```
export const getTotalIdentificationsService = async () => {
    const identifications = await People.findAll({
        attributes: [
            'identificationTypeId',
            [fn('COUNT', col('identificationTypeId')), 'total'], // Alias para contar las identificaciones
        ],
        group: ['identificationTypeId'],
        include: [
            {
                model: IdentificationType, // Modelo relacionado
                as: 'identificationType', // Alias usado en el belongsTo
                attributes: ['id', 'name'], // Atributos deseados
            },
        ],
    });
```

```
    // Mapear los resultados para devolverlos en el formato requerido
    const result = identifications.map(identification => {
        const identificationType = identification.get('identificationType') as IdentificationType | null;
// Castear a IdentificationType
        return {
            type: identificationType?.name || null, // Acceder a `name`
            total: identification.get('total'),
        };
    });
```

```
    return result;
};
```

Rutas:

```
import { Router } from 'express';
import { createPerson } from '../controllers/createPerson';
import { getPeople } from '../controllers/getPeople';
import { getPerson } from '../controllers/getPerson';
import { updatePerson } from '../controllers/updatePerson';
import { deletePerson } from '../controllers/deletePerson';
import { getUserFullInfo } from '../controllers/getUserFullInfo';
import { getTotalPeople } from '../controllers/getTotalPeople';
import { getPeopleByCriteria } from '../controllers/getPeopleByCriteria';
import { getPeopleByCity } from '../controllers/getPeopleByCity';
import { updatePersonAddressAndAnalyze } from
'../controllers/updatePersonAddressAndAnalyze';
import { getPeopleInCity } from '../controllers/getPeopleInCity';
import { getPersonById } from '../controllers/getPersonById';
import { getPeopleStatistics } from '../controllers/getPeopleStatistics';
import { getTotalIdentifications } from '../controllers/getTotalIdentifications';
```

```
const router: Router = Router();
```

```
router.post('/people', createPerson);
router.get('/people', getPeople);
router.get('/people/:id', getPerson);
router.put('/people/:id', updatePerson);
router.delete('/people/:id', deletePerson);
router.get('/userFullInfo/:id', getUserFullInfo);
router.get('/totalPeople', getTotalPeople);
router.get('/peopleByCriteria', getPeopleByCriteria);
router.get('/peopleByCity', getPeopleByCity);
router.put('/updatePersonAddress/:id', updatePersonAddressAndAnalyze);
router.get('/peopleInCity', getPeopleInCity);
router.get('/personById', getPersonById);
router.get('/peopleStatistics', getPeopleStatistics);
router.get('/totalIdentifications', getTotalIdentifications);
```

```
export default router;
```

Funciones:

- **createPerson.ts**: Permite crear un nuevo registro de persona en la base de datos.
- **getPeople.ts**: Recupera todos los registros de personas.
- **getPerson.ts**: Recupera una persona específica mediante su ID.
- **updatePerson.ts**: Actualiza la información de una persona existente.
- **deletePerson.ts:** Elimina un registro de persona de la base de datos.
- **getUserFullInfo.ts**: Recupera la información completa de un usuario, incluyendo detalles de identificación y dirección.
- **getTotalPeople.ts**: Calcula el total de personas registradas.
- **getPeopleByCriteria.ts**: Recupera personas que coinciden con criterios de búsqueda específicos.
- **getPeopleByCity.ts**: Recupera personas que viven en una ciudad específica.
- **updatePersonAddressAndAnalyze.ts**: Actualiza la dirección de una persona y realiza un análisis de datos.
- **getPeopleInCity.ts**: Recupera personas que viven en una ciudad específica.
- **getPersonById.ts**: Busca una persona mediante su tipo y número de identificación.
- **getPeopleStatistics.ts**: Recupera estadísticas sobre las personas registradas.
- **getTotalIdentifications.ts**: Calcula el total de identificaciones en registros.

Crear el Modelo de Rol

Paso 1.

Define el modelo de rol en TypeScript. Esto representa la estructura de la tabla en la base de datos.

Archivo Typescript (.ts)

```
// src/models/roleModel.ts

import { Model, DataTypes } from 'sequelize';
import sequelize from '../config/database';

class Role extends Model {
  public id!: number;
  public name!: string;

  // Timestamps
  public readonly createdAt!: Date;
  public readonly updatedAt!: Date;
}

// Definición del modelo 'Role'
Role.init(
  {
    id: {
      // Identificador único del rol
      type: DataTypes.INTEGER.UNSIGNED,
      autoIncrement: true,
      primaryKey: true,
    },
    name: {
      // Nombre del rol
      type: DataTypes.STRING,
      allowNull: false,
      unique: true,
    },
  },
  {
    tableName: 'roles',
    sequelize, // Pasar la instancia de `sequelize` es requerido
    timestamps: true, // Habilitar timestamps automáticos
  }
);

export default Role;
```

Paso 2: Crear los Endpoints en el Controlador

Crea el controlador de roles con las operaciones CRUD necesarias.

Crear Rol

Archivo Typescript (.ts)

```
// src/controllers/createRole.ts

import { Request, Response } from 'express';
import Role from '../models/roleModel';
```

```
// Controlador para crear un nuevo rol
export const createRole = async (req: Request, res: Response): Promise<void> => {
    try {
        const { name } = req.body; // Obtener el nombre del rol desde el cuerpo de la solicitud
        const role = await Role.create({ name }); // Crear el rol en la base de datos
        res.status(201).json({ role }); // Responder con el rol creado
    } catch (error) {
        res.status(500).json({ error: 'Error creando el rol' }); // Manejo de errores
    }
};
```

Obtener Todos los Roles

Archivo Typescript (.ts)

```
// src/controllers/getRoles.ts

import { Request, Response } from 'express';
import Role from '../models/roleModel';
```

```
// Controlador para obtener todos los roles
export const getRoles = async (_: Request, res: Response): Promise<void> => {
    try {
        const roles = await Role.findAll(); // Obtener todos los roles de la base de datos
        res.status(200).json({ roles }); // Responder con la lista de roles
    } catch (error) {
        res.status(500).json({ error: 'Error obteniendo los roles' }); // Manejo de errores
    }
};
```

Obtener Rol por Id

Archivo Typescript (.ts)

```
// src/controllers/getRole.ts

import { Request, Response } from 'express';
import Role from '../models/roleModel';
```

```
// Controlador para obtener un rol por su ID
export const getRole = async (req: Request, res: Response): Promise<void> => {
    try {
        const role = await Role.findByPk(req.params.id); // Obtener el rol por su ID de la base de datos
        if (role) {
            res.status(200).json({ role }); // Responder con el rol encontrado
        } else {
            res.status(404).json({ error: 'Rol no encontrado' }); // Responder si el rol no existe
        }
    } catch (error) {
        res.status(500).json({ error: 'Error obteniendo el rol' }); // Manejo de errores
    }
};
```

Actualizar Rol

Archivo Typescript (.ts)

```
// src/controllers/updateRole.ts

import { Request, Response } from 'express';
import Role from '../models/roleModel';
```

```
// Controlador para actualizar un rol por su ID
export const updateRole = async (req: Request, res: Response): Promise<void> => {
    try {
        const role = await Role.findByPk(req.params.id); // Obtener el rol por su ID de la base de datos
        if (role) {
            await role.update(req.body); // Actualizar el rol con los datos proporcionados en el cuerpo de la solicitud
            res.status(200).json({ role }); // Responder con el rol actualizado
        } else {
            res.status(404).json({ error: 'Rol no encontrado' }); // Responder si el rol no existe
        }
    } catch (error) {
        res.status(500).json({ error: 'Error actualizando el rol' }); // Manejo de errores
    }
};
```

Eliminar Rol

Archivo Typescript (.ts)

```
// src/controllers/deleteRole.ts

import { Request, Response } from 'express';
```

```
import Role from '../models/roleModel';
```

```
// Controlador para eliminar un rol por su ID
export const deleteRole = async (req: Request, res: Response): Promise<void> => {
    try {
        const role = await Role.findByPk(req.params.id); // Obtener el rol por su ID de la base de datos
        if (role) {
            await role.destroy(); // Eliminar el rol de la base de datos
            res.status(200).json({ message: 'Rol eliminado exitosamente' }); // Responder con éxito
        } else {
            res.status(404).json({ error: 'Rol no encontrado' }); // Responder si el rol no existe
        }
    } catch (error) {
        res.status(500).json({ error: 'Error eliminando el rol' }); // Manejo de errores
    }
};
```

Endpoints Adicionales

Calcular Total de Roles

Archivo Typescript (.ts)

```
/ src/controllers/countRoles.ts

import { Request, Response } from 'express';
import Role from '../models/roleModel';
```

```
// Controlador para contar la cantidad total de roles
export const countRoles = async (_: Request, res: Response): Promise<void> => {
    try {
        const count = await Role.count(); // Contar la cantidad total de roles en la base de datos
        res.status(200).json({ count }); // Responder con la cantidad total de roles
    } catch (error) {
        res.status(500).json({ error: 'Error contando los roles' }); // Manejo de errores
    }
};
```

Buscar Rol por Nombre

Archivo Typescript (.ts)

```
// src/controllers/getRoleByName.ts

import { Request, Response } from 'express';
import Role from '../models/roleModel';
```

```
// Controlador para buscar un rol por su nombre
export const getRoleByName = async (req: Request, res: Response): Promise<void> => {
    try {
        const { name } = req.params; // Obtener el nombre del rol desde los parámetros de la
solicitud
        const role = await Role.findOne({ where: { name } }); // Buscar el rol en la base de datos
        if (role) {
            res.status(200).json({ role }); // Responder con el rol encontrado
        } else {
            res.status(404).json({ error: 'Rol no encontrado' }); // Responder si el rol no existe
        }
    } catch (error) {
        res.status(500).json({ error: 'Error buscando el rol' }); // Manejo de errores
    }
};
```

Ruta:

```
import { Router } from 'express';
import { createRole } from '../controllers/createRole';
import { getRoles } from '../controllers/getRoles';
import { getRole } from '../controllers/getRole';
import { updateRole } from '../controllers/updateRole';
import { deleteRole } from '../controllers/deleteRole';
import { countRoles } from '../controllers/countRoles';
import { getRoleByName } from '../controllers/getRoleByName';

const router: Router = Router();
```

```
router.post('/roles', createRole);
router.get('/roles', getRoles);
router.get('/roles/:id', getRole);
router.put('/roles/:id', updateRole);
router.delete('/roles/:id', deleteRole);
router.get('/roles/count', countRoles);
router.get('/roles/name/:name', getRoleByName);
```

```
export default router;
```

Funciones:

- **createRole.ts:** Permite crear un nuevo rol en la base de datos.
- **getRoles.ts:** Recupera todos los roles registrados.
- **getRole.ts:** Recupera un rol específico mediante su ID.
- **updateRole.ts:** Actualiza la información de un rol existente.
- **deleteRole.ts:** Elimina un rol de la base de datos.
- **countRoles.ts:** Calcula el total de roles registrados.
- **getRoleByName.ts:** Busca un rol por su nombre.

Modelo de IdentificationType

Definimos el modelo de IdentificationType para estructurar los datos.

```
// src/models/identificationTypeModel.ts

import { Model, DataTypes } from 'sequelize';
import sequelize from '../config/database';

class IdentificationType extends Model {
  public id!: number;
  public name!: string;

  // Timestamps
  public readonly createdAt!: Date;
  public readonly updatedAt!: Date;
}

IdentificationType.init(
  {
    id: {
      type: DataTypes.INTEGER.UNSIGNED,
      autoIncrement: true,
      primaryKey: true,
    },
    name: {
      type: DataTypes.STRING,
      allowNull: false,
    },
  },
  {
    tableName: 'identificationTypes',
    sequelize,
    timestamps: true,
  }
);

export default IdentificationType;
```

1) Endpoints del Controlador para IdentificationType

- Crear Tipo de Identificación
- Archivo: createIdentificationType.ts
- Ruta: POST /identificationTypes
- Descripción: Permite crear un nuevo tipo de identificación en la base de datos.

```
// src/controllers/createIdentificationType.ts

import { Request, Response } from 'express';
import IdentificationType from '../models/identificationTypeModel';
```

```
export const createIdentificationType = async (req: Request, res: Response): Promise<void> => {
    try {
        const { name } = req.body;
        const identificationType = await IdentificationType.create({ name });
        res.status(201).json({ identificationType });
    } catch (error) {
        res.status(500).json({ error: 'Error creating identification type' });
    }
};
```

2) Obtener Todos los Tipos de Identificación

Archivo: getIdentificationTypes.ts

Ruta: GET /identificationTypes

Descripción: Recupera todos los tipos de identificación registrados.

```
// src/controllers/getIdentificationTypes.ts

import { Request, Response } from 'express';
import IdentificationType from '../models/identificationTypeModel';
```

```
export const getIdentificationTypes = async (_: Request, res: Response): Promise<void> => {
    try {
        const identificationTypes = await IdentificationType.findAll();
        res.status(200).json({ identificationTypes });
    } catch (error) {
        res.status(500).json({ error: 'Error fetching identification types' });
    }
};
```

3) Obtener Tipo de Identificación por ID

Archivo: getIdentificationType.ts

Ruta: GET /identificationTypes/:id

Descripción: Recupera un tipo de identificación específico mediante su ID.

```
// src/controllers/getIdentificationType.ts
```

```
import { Request, Response } from 'express';
import IdentificationType from '../models/identificationTypeModel';
```

```
export const getIdentificationType = async (req: Request, res: Response): Promise<void> => {
    try {
        const id = parseInt(req.params.id, 10);
        const identificationType = await IdentificationType.findByPk(id);
        if (identificationType) {
            res.status(200).json({ identificationType });
        } else {
            res.status(404).json({ error: 'Identification type not found' });
        }
    } catch (error) {
        res.status(500).json({ error: 'Error fetching identification type' });
    }
};
```

4) Actualizar Tipo de Identificación

Archivo: updateIdentificationType.ts

Ruta: PUT /identificationTypes/:id

Descripción: Actualiza la información de un tipo de identificación existente.

```
// src/controllers/updateIdentificationType.ts

import { Request, Response } from 'express';
import IdentificationType from '../models/identificationTypeModel';
```

```
export const updateIdentificationType = async (req: Request, res: Response): Promise<void> => {
    try {
        const id = parseInt(req.params.id, 10);
        const { name } = req.body;
        const identificationType = await IdentificationType.findByPk(id);
        if (identificationType) {
            await identificationType.update({ name });
            res.status(200).json({ identificationType });
        } else {
            res.status(404).json({ error: 'Identification type not found' });
        }
    } catch (error) {
        res.status(500).json({ error: 'Error updating identification type' });
    }
};
```

5) Eliminar Tipo de Identificación

Archivo: deleteIdentificationType.ts

Ruta: DELETE /identificationTypes/:id

Descripción: Elimina un tipo de identificación de la base de datos.

```
// src/controllers/deleteIdentificationType.ts

import { Request, Response } from 'express';
import IdentificationType from '../models/identificationTypeModel';
```

```
export const deleteIdentificationType = async (req: Request, res: Response): Promise<void> => {
    try {
        const id = parseInt(req.params.id, 10);
        const identificationType = await IdentificationType.findByPk(id);
        if (identificationType) {
            await identificationType.destroy();
            res.status(204).send();
        } else {
            res.status(404).json({ error: 'Identification type not found' });
        }
    } catch (error) {
        res.status(500).json({ error: 'Error deleting identification type' });
    }
};
```

Modelo de Address

Definimos el modelo de Address para estructurar los datos de las direcciones.

```
import { Model, DataTypes } from 'sequelize';
import sequelize from 'src/config/database'; // Ruta absoluta
import People from './peopleModel';

class Address extends Model {
    public id!: number;
    public street!: string;
    public city!: string;
    public state!: string;
    public zipCode!: string;
    public country!: string;
    public personId!: number;
```

```
    // Timestamps
    public readonly createdAt!: Date;
    public readonly updatedAt!: Date;
}
```

```
Address.init(
  {
    id: {
      type: DataTypes.INTEGER.UNSIGNED,
      autoIncrement: true,
      primaryKey: true,
    },
    street: {
      type: DataTypes.STRING,
      allowNull: false,
    },
    city: {
      type: DataTypes.STRING,
      allowNull: false,
    },
    state: {
      type: DataTypes.STRING,
      allowNull: false,
    },
    zipCode: {
      type: DataTypes.STRING,
      allowNull: false,
    },
    country: {
      type: DataTypes.STRING,
      allowNull: false,
    },
    personId: {
      type: DataTypes.INTEGER.UNSIGNED,
      allowNull: false,
      references: {
        model: People, // Referencia al modelo People
        key: 'id',
      },
    },
  },
  {
    tableName: 'addresses',
    sequelize,
    timestamps: true,
  }
);
```

```
export default Address;
```

Endpoints del Controlador para Address

Crear Dirección

Archivo: createAddress.ts

Ruta: POST /addresses

Descripción: Permite crear una nueva dirección en la base de datos.

```
// src/controllers/deleteAddress.ts

import { Request, Response } from 'express';
import Address from '../models/addressModel';
```

```
export const deleteAddress = async (req: Request, res: Response): Promise<void> => {
    try {
        const id = parseInt(req.params.id, 10);
        const address = await Address.findByPk(id);
        if (address) {
            await address.destroy();
            res.status(204).send();
        } else {
            res.status(404).json({ error: 'Address not found' });
        }
    } catch (error) {
        res.status(500).json({ error: 'Error deleting address' });
    }
};
```

Obtener Todas las Direcciones

Archivo: getAddresses.ts

Ruta: GET /addresses

Descripción: Recupera todas las direcciones registradas.

```
// src/controllers/getAddresses.ts

import { Request, Response } from 'express';
import Address from 'src/models/addressModel';
```

```
export const getAddresses = async (_: Request, res: Response): Promise<void> => {
    try {
        const addresses = await Address.findAll();
        res.status(200).json({ addresses });
    } catch (error) {
        res.status(500).json({ error: 'Error fetching addresses' });
    }
};
```

Obtener Dirección por ID

Archivo: getAddress.ts

Ruta: GET /addresses/:id

Descripción: Recupera una dirección específica mediante su ID.

```
// src/controllers/getAddress.ts

import { Request, Response } from 'express';
import Address from '../models/addressModel';
```

```
export const getAddress = async (req: Request, res: Response): Promise<void> => {
    try {
        const id = parseInt(req.params.id, 10);
        const address = await Address.findByPk(id);
        if (address) {
            res.status(200).json({ address });
        } else {
            res.status(404).json({ error: 'Address not found' });
        }
    } catch (error) {
        res.status(500).json({ error: 'Error fetching address' });
    }
};
```

Endpoints del Controlador para Address (Continuación)

Actualizar Dirección

Archivo: updateAddress.ts

Ruta: PUT /addresses/:id

Descripción: Actualiza la información de una dirección existente.

```
// src/controllers/updateAddress.ts

import { Request, Response } from 'express';
import Address from '../models/addressModel';
```

```
export const updateAddress = async (req: Request, res: Response): Promise<void> => {
    try {
        const id = parseInt(req.params.id, 10);
        const { street, city, state, zipCode, country, personId } = req.body;
        const address = await Address.findByPk(id);
```

```
        if (address) {
            await address.update({ street, city, state, zipCode, country, personId });
            res.status(200).json({ address });
        } else {
            res.status(404).json({ error: 'Address not found' });
        }
    } catch (error) {
        res.status(500).json({ error: 'Error updating address' });
    }
};
```

Eliminar Dirección

Archivo: deleteAddress.ts

Ruta: DELETE /addresses/:id

Descripción: Elimina una dirección de la base de datos.

```
// src/controllers/deleteAddress.ts

import { Request, Response } from 'express';
import Address from '../models/addressModel';
```

```
export const deleteAddress = async (req: Request, res: Response): Promise<void> => {
    try {
        const id = parseInt(req.params.id, 10);
        const address = await Address.findByPk(id);
        if (address) {
            await address.destroy();
            res.status(204).send();
        } else {
            res.status(404).json({ error: 'Address not found' });
        }
    } catch (error) {
        res.status(500).json({ error: 'Error deleting address' });
    }
};
```

Rutas:

```
import { Router } from 'express';
```

```
import { createAddress } from '../controllers/createAddress';
import { getAddresses } from '../controllers/getAddresses';
import { getAddress } from '../controllers/getAddress';
import { updateAddress } from '../controllers/updateAddress';
import { deleteAddress } from '../controllers/deleteAddress';
```

```
const router: Router = Router();

router.post('/addresses', createAddress);
router.get('/addresses', getAddresses);
router.get('/addresses/:id', getAddress);
router.put('/addresses/:id', updateAddress);
router.delete('/addresses/:id', deleteAddress);

export default router;
```

Crear Modelos y Relaciones

Objetivo:

Establecer los modelos y relaciones entre `IdentificationType`, `Addresses`, `People`, y otras entidades relevantes para asegurar la integridad y coherencia de los datos en la base de datos.

Modelos:

1. IdentificationType:

- Un tipo de identificación que puede ser utilizado por una persona para identificarse, como pasaporte, licencia de conducir, etc.

2. Address:

- Una dirección física que está asociada a una persona, incluyendo detalles como calle, ciudad, estado, código postal y país.

3. People:

- Información detallada de una persona, incluyendo su nombre, apellido, fecha de nacimiento, tipo de identificación, número de identificación y dirección.

Relaciones:

- People - IdentificationType:

- Relación: Un `People` tiene un `IdentificationType`.

- Tipo de Relación: `belongsTo` y `hasMany`.

- Descripción: Cada persona tiene un tipo de identificación específico. Esto implica que un `People` está relacionado con un único `IdentificationType`, y un `IdentificationType` puede estar relacionado con muchas `People`.

- People - Address:

- Relación: Un `People` tiene una `Address`.

- Tipo de Relación: `belongsTo` y `hasMany`.

- Descripción: Cada persona tiene una dirección específica. Un `People` está relacionado con una única `Address`, y una `Address` puede estar relacionada con muchas `People`.

Implementación sin Código:

1. Definición de Modelos:

- Empieza por definir los modelos de `IdentificationType`, `Address` y `People` en tu ORM (Object-Relational Mapping), como Sequelize. Estos modelos representan la estructura de tus tablas en la base de datos, incluyendo los campos y sus tipos de datos.

2. Establecer Relaciones:

- Después de definir los modelos, establece las relaciones entre ellos. Utiliza métodos como `belongsTo` para indicar que una entidad está relacionada con otra mediante una clave foránea. Utiliza `hasMany` para indicar que una entidad puede estar relacionada con muchas otras entidades.

3. Sincronización de la Base de Datos:

- Sincroniza la base de datos para asegurarte de que las tablas y sus relaciones se crean correctamente. Esto incluye ejecutar migraciones si es necesario.

4. Verificación y Validación:

- Verifica que los datos insertados cumplan con las restricciones y relaciones definidas. Realiza consultas para asegurarte de que los datos relacionados se recuperan correctamente y que no hay inconsistencias.

Ejemplo Práctico:

- Escenario: Supongamos que necesitamos almacenar y gestionar información sobre personas, sus tipos de identificación y sus direcciones.

1. Crear el Modelo de `IdentificationType`:

- Define un modelo con campos como `id`, `type`, y `description`. Asegúrate de que `type` sea único y no esté vacío.

2. Crear el Modelo de `Address`:

- Define un modelo con campos como `id`, `street`, `city`, `state`, `zipCode`, `country`, y `phoneNumber`. Valida que el `zipCode` y `phoneNumber` tengan el formato correcto.

3. Crear el Modelo de `People`:

- Define un modelo con campos como `id`, `firstName`, `lastName`, `birthDate`, `identificationTypeId`, `identificationNumber`, `addressId`, y `email`. Asegúrate de que `identificationTypeId` y `addressId` sean claves foráneas válidas.

4. Establecer Relaciones:

- Configura que `People` pertenece a `IdentificationType` y `Address` mediante claves foráneas `identificationTypeId` y `addressId`.

- Configura que `IdentificationType` y `Address` tienen muchas `People`.

5. Sincronizar y Verificar:

- Sincroniza la base de datos para crear las tablas y relaciones.

- Inserta datos de prueba y realiza consultas para verificar que las relaciones funcionan correctamente y que los datos son consistentes.

Y asi hemos establecido los modelos y relaciones necesarios para `IdentificationType`, `Address` y `People`, asegurando que los datos en la base de datos sean consistentes y coherentes. Este paso es crucial para la integridad de la información y para permitir consultas eficientes y precisas.

Completar Tablas y Relacionar Datos (6.2)

Objetivo:

El objetivo principal de este subpunto es asegurar que las tablas de la base de datos estén completas y que los datos entre las tablas IdentificationType, Addresses y People estén correctamente relacionados. Esto es crucial para mantener la integridad y coherencia de los datos.

Pasos para Completar Tablas y Relacionar Datos

1. Agregar Campos Faltantes

Para asegurar que todas las tablas tengan la información necesaria, debemos agregar cualquier campo adicional que falte. Esto incluye definir campos adicionales en los modelos para capturar información relevante.

IdentificationType:

-Agregar un campo description para proporcionar más contexto sobre el tipo de identificación.

Address:

-Agregar un campo phoneNumber para capturar el número de teléfono asociado con la dirección.

People:

-Agregar un campo email para capturar la dirección de correo electrónico de la persona.

2. Establecer Claves Foráneas y Relaciones

Es esencial configurar las relaciones entre las tablas para asegurar que los datos estén correctamente relacionados y mantener la integridad referencial.

People - IdentificationType:

-Cada persona debe estar asociada con un tipo de identificación válido.

-Configura identificationTypeId como clave foránea en People.

-Define la relación con belongsTo en People y hasMany en IdentificationType.

People - Address:

-Cada persona debe estar asociada con una dirección válida.

-Configura addressId como clave foránea en People.

-Define la relación con belongsTo en People y hasMany en Address.

3. Sincronización de la Base de Datos

Asegura que todas las tablas y relaciones estén correctamente sincronizadas en la base de datos. Esto puede implicar la creación de migraciones y la ejecución de scripts para sincronizar el esquema de la base de datos.

Ejemplo Práctico

Sin Código:

Definir los Modelos:

-Comienza por definir o actualizar los modelos en tu ORM para incluir cualquier campo adicional necesario.

-Para IdentificationType, asegúrate de incluir un campo description y que type sea único y no vacío.

-Para Address, incluye un campo phoneNumber con las validaciones necesarias para el formato.

-Para People, asegúrate de que incluye un campo email con la validación de formato de correo electrónico.

Establecer Relaciones:

-Configura People para que pertenezca a IdentificationType y Address, estableciendo identificationTypeId y addressId como claves foráneas.

-Configura IdentificationType para que tenga muchas People.

-Configura Address para que tenga muchas People.

Sincronizar la Base de Datos:

-Ejecuta las migraciones necesarias para crear o actualizar las tablas y relaciones en la base de datos.

-Verifica que las tablas estén correctamente relacionadas mediante consultas y verificaciones manuales.

Ejemplo Práctico en Código:

Para aquellos interesados en cómo se vería esto en código (sin incluir fragmentos de código en la respuesta):

- Modelo de IdentificationType: Incluir el campo description y asegurar la unicidad del campo type.
- Modelo de Address: Incluir el campo phoneNumber con validación.
- Modelo de People: Incluir el campo email y configurar las relaciones.
- Sincronización: Asegurar la integridad referencial mediante claves foráneas.

Asi hemos de completar las tablas y establecer relaciones correctas ya que es un paso crucial para asegurar que la base de datos funcione de manera eficiente y coherente. Esto garantiza que los datos estén correctamente relacionados y que podamos realizar consultas precisas y efectivas.

Validar Consistencia de Datos (6.3)

Objetivo:

El objetivo es asegurar que los datos en la base de datos sean consistentes y cumplan con todas las restricciones y relaciones definidas entre las tablas IdentificationType, Addresses, y People.

Pasos para Validar la Consistencia de Datos

1. Validaciones en el Modelo

Implementaremos validaciones en los modelos para asegurarnos de que los datos ingresados sean consistentes y cumplan con las restricciones definidas.

- IdentificationType: Asegura que cada tipo de identificación tenga un valor único y significativo. Por ejemplo, el campo type no debe estar vacío y debe contener solo caracteres alfanuméricos.
- Address: Verifica que cada dirección tenga campos como calle, ciudad, estado, código postal y país correctamente llenados. Utiliza validaciones para asegurar que el código postal tenga el formato correcto y que el número de teléfono también cumpla con los estándares esperados.
- People: Cada persona debe tener un nombre, apellido, fecha de nacimiento, tipo de identificación válido, número de identificación único y dirección. Valida que el correo electrónico sea válido y no esté vacío.

2. Establecer Claves Foráneas y Relaciones

Es crucial configurar las relaciones entre las tablas para asegurar que los datos estén correctamente relacionados y mantener la integridad referencial.

- **People - IdentificationType:**

- Cada persona debe estar asociada con un tipo de identificación válido.

-Configura identificationTypeId como clave foránea en People.

-Define la relación con belongsTo en People y hasMany en IdentificationType.

- **People - Address:**

-Cada persona debe estar asociada con una dirección válida.

-Configura addressId como clave foránea en People.

-Define la relación con belongsTo en People y hasMany en Address.

3. Sincronización de la Base de Datos

Asegura que todas las tablas y relaciones estén correctamente sincronizadas en la base de datos. Esto puede implicar la creación de migraciones y la ejecución de scripts para sincronizar el esquema de la base de datos.

4. Verificación de Integridad Referencial

Para garantizar la integridad referencial, configura claves foráneas y valida que las relaciones entre las tablas sean coherentes. Esto incluye:

IdentificationType: Cada entrada en People debe referirse a una entrada válida en IdentificationType.

Address: Cada entrada en People debe referirse a una entrada válida en Address.

Ejemplos de Uso

Al crear un Registro Completo de People con Relaciones:

-Insertamos una nueva persona con los datos correctos, incluyendo el tipo de identificación y la dirección. Asegúrando de que estos estén validados y relacionados correctamente.

Consultar una Persona con sus Relaciones:

-Recupera una persona específica e incluye sus relaciones con IdentificationType y Address. Verificamos que estas relaciones sean válidas y que no haya inconsistencias.

Endpoints para Validación de Consistencia

Endpoint para Validar Personas con Relaciones Correctas:

-Creamos un endpoint que recupera todas las personas, incluye sus tipos de identificación y direcciones, y verifica que todos los datos sean consistentes. Esto devuelve un informe detallado de cualquier inconsistencia encontrada para que pueda ser corregida.

Agregar Funciones Avanzadas:

Nuestro objetivo aquí fue extender la funcionalidad básica del CRUD para incluir operaciones más avanzadas que añadan valor y mejoren la usabilidad de nuestra aplicación.

Filtrado y Búsqueda Avanzada:

Implementamos funcionalidades para buscar personas basándose en criterios específicos, como nombre, ciudad o tipo de identificación. Esto permite a los usuarios encontrar información más rápidamente y de manera más eficiente.

Estadísticas y Reportes:

Desarrollamos funcionalidades que permiten generar estadísticas y reportes, como el total de personas registradas, el total de identificaciones, y otros datos relevantes. Esto proporciona a los usuarios una visión general y detallada del estado de los datos.

Validaciones y Notificaciones:

Incluimos validaciones adicionales y notificaciones para alertar a los usuarios en caso de inconsistencias o errores en los datos. Esto mejora la calidad de los datos y ayuda a mantener la integridad del sistema.

Mejoras en la Seguridad y Autenticación:

Implementamos mejoras en la seguridad, como autenticación y autorización, para asegurar que solo usuarios autorizados puedan acceder y modificar los datos. Esto protege la información sensible y asegura que el sistema sea utilizado de manera segura.

Implementación de Filtrado y Búsqueda Avanzada

Desarrollamos funcionalidades que permiten a los usuarios realizar búsquedas avanzadas y filtrar resultados basados en criterios específicos. Esto mejora la eficiencia y efectividad al encontrar información relevante.

- Filtrar por Nombre y Ciudad: Implementamos opciones de filtrado que permiten a los usuarios buscar personas por nombre y ciudad, facilitando la localización rápida de datos específicos.
- Buscar por Tipo de Identificación: Añadimos funcionalidades para buscar personas basadas en su tipo de identificación. Esto es útil para organizaciones que necesiten clasificar o agrupar personas según su tipo de documento.

Generación de Estadísticas y Reportes

Para proporcionar una visión más amplia del estado de los datos, creamos funcionalidades que generan estadísticas y reportes detallados

- Total de Personas Registradas: Desarrollamos funcionalidades para calcular y mostrar el número total de personas registradas en el sistema. Esto ayuda a tener una visión general del volumen de datos manejados.
- Estadísticas por Ciudad: Implementamos reportes que muestran estadísticas detalladas de personas agrupadas por ciudad, permitiendo identificar patrones y tendencias demográficas.

Validaciones y Notificaciones

Para asegurar la calidad de los datos, incluimos validaciones adicionales y notificaciones que alertan a los usuarios sobre posibles errores o inconsistencias en los datos.

- Validación de Datos de Entrada: Añadimos validaciones en el frontend y backend para asegurar que los datos ingresados sean correctos y cumplan con los formatos esperados.
- Notificaciones de Errores: Implementamos un sistema de notificaciones que alerta a los usuarios cuando se detectan inconsistencias o errores en los datos. Esto mejora la integridad de los datos y facilita su corrección.

Mejoras en la Seguridad y Autenticación

Para proteger la información sensible y asegurar que solo usuarios autorizados puedan acceder y modificar los datos, implementamos mejoras en la seguridad del sistema.

- Autenticación de Usuarios: Desarrollamos un sistema de autenticación robusto que verifica la identidad de los usuarios antes de permitir el acceso a la información.
- Autorización Basada en Roles: Implementamos un sistema de autorización que asigna permisos específicos a los usuarios según su rol. Esto asegura que solo los usuarios con los permisos adecuados puedan realizar ciertas acciones.

Beneficios de Usar Swagger

Facilidad de Uso:

-Swagger proporciona una interfaz visual interactiva que facilita a los desarrolladores probar y entender los endpoints de la API.

Documentación Automatizada:

-Las anotaciones en el código generan automáticamente la documentación, asegurando que esté siempre actualizada con los cambios en la API.

-Acceso Rápido a la Información:

-Los desarrolladores pueden ver rápidamente qué endpoints existen, qué parámetros aceptan y qué respuestas devuelven, lo que facilita la integración y el uso de la API.

Mejora en la Colaboración:

-Con una documentación clara y accesible, es más fácil para los equipos de desarrollo colaborar y entender cómo interactuar con la API.

Conclusiones

El desarrollo de esta API RESTful representó un desafío significativo, pero sumamente enriquecedor para nuestro equipo. A lo largo de este proyecto, enfrentamos y superamos diversos retos, aplicando las mejores prácticas y enfoques para lograr soluciones eficientes y bien estructuradas. Este proceso no solo nos permitió consolidar nuestros conocimientos, sino que también contribuyó al avance considerable de nuestro desarrollo profesional.

El proyecto nos brindó una valiosa experiencia en la creación y gestión de APIs RESTful, estableciendo una base sólida sobre la cual podemos construir y expandir en el futuro. Estamos convencidos de que los aprendizajes adquiridos y las habilidades desarrolladas durante este proyecto nos preparan adecuadamente para enfrentar futuros desafíos en el campo del desarrollo de servicios web. Esto nos permitirá avanzar en nuestro desarrollo y llevar a cabo proyectos de mayor envergadura con éxito, especialmente en relación con nuestros futuros proyectos de grado y los proyectos personales en los que se basa este desarrollo.

Recomendaciones

Por el momento, se requiere un uso básico de la API, lo que implica la descarga de los scripts y la configuración necesarios para su funcionamiento y pruebas. Los detalles específicos se encuentran en el repositorio de GitHub correspondiente. Para asegurar un funcionamiento adecuado, es importante descargar la última versión disponible de la API y los scripts asociados, garantizando así que el sistema opere de manera efectiva y eficiente.

Referencias