DOCUMENTACIÓN API

Paquetes instalados

-express->Framework para crear servidores HTTP, rutas, middleware y controladores, este lo usamos para alojar la API en un servidor express local.

-dotnev->Para cargar variables de entorno, en este caso nuestro archivo .env.

-sequilize->Para manejar modelos, relaciones y consultas SQL.

-sqlite3->Driver que permite a sequilize conectarse a SQLite.

-swagger->Para generar la documentación de la API.

-zod->Para validar datos de entrada con schemas

Creación de modelos

-Para la creación de los modelos usamos los DataType que nos da la librería de sequilize para establecer los tipos de las columnas que tendrá la base de datos, además del sequilize de la base de datos para darle el esquema a la propia base de datos.

Modelo User

```
import { DataTypes } from "sequelize";//Esto nos permitirá establecer los tipos de las columnas
const sequelize = require("../service/db.js");//Es la conexión a la base de datos que ya fue configurada


const User = sequelize.define("User", {//Establezco que quiero un modelo llamado USER
    nombre: {//Con un atributo nombre
        type: DataTypes.STRING,//de tipo string
        allowNull: false//que no pueda ser nullable
    },
    email: {//con un atributo email
        type: DataTypes.STRING,//de tipo string
        unique: true,//que sea unico, es decir, que no se pueda repitir
        allowNull: false//que sea nullable
    },
    activo: {//con una propiedad para declarar si se encuentra activo
        type: DataTypes.BOOLEAN,//establecer que es de tipo booleano
        defaultValue: true//que tenga un valor por defecto
    }
}, {
    tableName: "user"//El nombre de  la tabla
});

module.exports = User;
```

Modelo Book

```
import { DataTypes } from "sequelize";//Esto nos permitirá establecer los tipos de las columnas
const sequelize = require("../service/db.js");//Es la conexión a la base de datos que ya fue configurada


const Book = sequelize.define("book", {//Establezco que quiero un modelo llamado book
    titulo: {//con un atributo book
        type: DataTypes.STRING,//de tipo string
        allowNull: false//que no permite que sea nullable
    },
    autor: {//Con un atributo autor
        type: DataTypes.STRING,//de tipo string
        allowNull: false//que no permite que sea nullable
    },
    stock: {//Con un atributo stock
        type: DataTypes.INTEGER,//que quiero que sea de tipo entero
        allowNull: false//que no permite que sea nullable
    }
}, {
    tableName: "book"//estableciendo que el nombre en la base de datos sea book
});

module.exports = Book;
```

Modelo Loan

```
import { DataTypes } from "sequelize";//Esto nos permitirá establecer los tipos de las columnas
const sequelize = require("../service/db.js");//Es la conexión a la base de datos que ya fue configurada

const Loan = sequelize.define("loan", {//Quiero un modelo llamado Loan
    fechaPrestamo: {//Con un atributo fechaPrestamo
        type: DataTypes.DATE,//de tipo fecha
        defaultValue: DataTypes.NOW//con valor por defecto la fecha del momento en el que se crea
    },
    fechaDevolucionPrevista: {//con un atributo fechaDevolucionPrevista
        type: DataTypes.DATE,//de tipo fecha
        allowNull: false//que no sea nullable
    },
    fechaDevolucionReal: {//con un atributo fechaDevolucionReal
        type: DataTypes.DATE,//de tipo fecha
        allowNull: true//`que no sea nullable
    }
}, {
    tableName: "loan"//que  en la base de datos la tabla se llamo loan
});

module.exports = Loan;
```

Relaciones de los modelos

-Al igual que se hace en un proyecto MVC de .NET se deben de declarar las diferentes relaciones que existen entre las entidades, es decir, tenemos que importar los diferentes schemas declarados (modelos) y establecer las relaciones entre ellos:

- HasMany:Crea una relación 1:N
- belongsTo:Añadel la FK a Loan

```
const User = require("./User.js");//Obtengo el modelo de user
const Book = require("./Book.js");//Obtengo el modelo de book
const Loan = require("./Loan.js");//Obtengo el modelo de loan

User.hasMany(Loan, { foreignKey: 'usuarioId' });//Relacion un usuario puede
tener muchos prestamos
Loan.belongsTo(User, { foreignKey: 'usuarioId' });//Cada prestamo pertenece
a un usuario
Book.hasMany(Loan, { foreignKey: 'libroId' });//Relacion un libro puede
tener muchos prestamos
Loan.belongsTo(Book, { foreignKey: 'libroId' });//Cada prestamo pertenece a
un libro

export { User, Book, Loan };
```

Creación de servicios

-En los servicios se usan los modelos para aplicar la lógica de negocio, este orquesta operaciones complejas y esconde los detalles de la base de datos del resto de la aplicación.

Cuando obtenemos la data en algunos de los métodos de los servicios se refiere al objeto que va a recibir y si lo recibe como Partial<T> significa que podrá ser opcional esos datos.

BookService

```
import { Book, Loan } from "../models";
// Obtenemos los modelos de libros y préstamos para trabajar con la base de datos



// Método en el que creamos un libro
export const createBook = async (data: {
    titulo: string;
    autor: string;
    stock: number;
}) => {
    // Creamos un nuevo registro de libro con los datos recibidos
    return Book.create(data);
};



// Método en el que obtenemos todos los libros
export const getBooks = async () => {
    // Devuelve todos los libros registrados en la base de datos
    return Book.findAll();
};



// Método en el que obtenemos un libro por su id
export const getBookById = async (id: number) => {
    // Busca un libro por su clave primaria (id)
    return Book.findByPk(id);
```

```
};



// Método en el que actualizamos un libro concreto
export const updateBook = async (
    id: number,
    data: Partial<{ titulo: string; autor: string; stock: number }>
) => {
    // Buscamos el libro por su id
    const book = await Book.findByPk(id);

    // Si no existe, lanzamos un error
    if (!book) throw new Error("Libro no encontrado");

    // Actualizamos los campos permitidos con los datos recibidos
    await book.update(data);

    // Devolvemos el libro actualiza    do
    return book;
};



// Método en el que borramos un libro concreto
export const deleteBook = async (id: number) => {
    // Buscamos el libro por su id
    const book = await Book.findByPk(id);

    // Si no existe, lanzamos un error
    if (!book) throw new Error("Libro no encontrado");

    // No podemos borrar un libro que tenga préstamos activos
    const activeLoans = await Loan.count({
        where: { libroId: id, fechaDevolucionReal: null },
        // Contamos cuántos préstamos activos existen para este libro
    });

    // Si hay préstamos activos, no permitimos eliminarlo
    if (activeLoans > 0) {
        throw new Error("No se puede eliminar un libro con préstamos
activos");
    }
```

```
    // Si no tiene préstamos activos, eliminamos el libro
    await book.destroy();
};
```

LoanService

```
import { Book, Loan, User } from "../models";
// Importamos los modelos Book, Loan y User para trabajar con la base de
datos



// Método que crea un préstamo nuevo
export const createLoan = async (data: {
    usuarioId: number;
    libroId: number;
    diasPrestamo: number;
}) => {
    // Desestructuramos los datos recibidos del cuerpo de la petición
    const { usuarioId, libroId, diasPrestamo } = data;

    // Buscamos al usuario que realiza el préstamo
    const usuario = await User.findByPk(usuarioId);
    // Si el usuario no existe o está inactivo, lanzamos un error
    if (!usuario || !usuario.getDataValue("activo")) {
        throw new Error("Usuario no válido o inactivo");
    }

    // Buscamos el libro que se quiere prestar
    const libro = await Book.findByPk(libroId);
    // Si el libro no existe, lanzamos un error
    if (!libro) throw new Error("Libro no encontrado");

    // Obtenemos el stock actual del libro
    const stock = libro.getDataValue("stock");
    // Si no hay ejemplares disponibles, no permitimos el préstamo
    if (stock <= 0) {
        throw new Error("No hay ejemplares disponibles");
    }

    // Calculamos la fecha de inicio del préstamo (fecha actual)
    const fechaPrestamo = new Date();
    // Inicializamos la fecha prevista de devolución
    const fechaDevolucionPrevista = new Date();
    // Sumamos los días de préstamo a la fecha actual para calcular la
devolución prevista
```

```
    fechaDevolucionPrevista.setDate(fechaPrestamo.getDate() +
diasPrestamo);

    // Creamos el registro del préstamo en la base de datos
    const prestamo = await Loan.create({
        usuarioId,
        libroId,
        fechaPrestamo,
        fechaDevolucionPrevista,
    });

    // Actualizamos el stock del libro restando una unidad
    await libro.update({ stock: stock - 1 });

    // Devolvemos el préstamo creado
    return prestamo;
};



// Método que devuelve todos los préstamos
export const getLoans = async () => {
    // Busca todos los préstamos incluyendo la información del usuario y
del libro asociado
    return Loan.findAll({ include: [User, Book] });
};



// Método que devuelve un préstamo concreto por su id
export const getLoanById = async (id: number) => {
    // Busca un préstamo por su clave primaria e incluye usuario y libro
relacionados
    return Loan.findByPk(id, { include: [User, Book] });
};



// Método que permite actualizar un préstamo
export const updateLoan = async (
    id: number,
    data: Partial<{ fechaDevolucionPrevista: Date; fechaDevolucionReal:
Date }>
```

```
) => {
    // Buscamos el préstamo por su id
    const loan = await Loan.findByPk(id);
    // Si no existe, lanzamos un error
    if (!loan) throw new Error("Préstamo no encontrado");

    // Actualizamos los campos permitidos con los datos recibidos
    await loan.update(data);
    // Devolvemos el préstamo ya actualizado
    return loan;
};



// Método que permite borrar un préstamo
export const deleteLoan = async (id: number) => {
    // Buscamos el préstamo por su id
    const loan = await Loan.findByPk(id);
    // Si no existe, lanzamos un error
    if (!loan) throw new Error("Préstamo no encontrado");

    // Si el préstamo aún no fue devuelto, debemos devolver el ejemplar al
stock
    if (!loan.getDataValue("fechaDevolucionReal")) {
        // Obtenemos el id del libro asociado al préstamo
        const libroId = loan.getDataValue("libroId");
        // Buscamos el libro en la base de datos
        const libro = await Book.findByPk(libroId);
        // Si el libro existe, incrementamos su stock en 1
        if (libro) {
            const stock = libro.getDataValue("stock");
            await libro.update({ stock: stock + 1 });
        }
    }

    // Eliminamos el préstamo de la base de datos
    await loan.destroy();
};



// Método que permite marcar un préstamo como devuelto
export const returnLoan = async (id: number) => {
```

```
    // Buscamos el préstamo por su id
    const loan = await Loan.findByPk(id);
    // Si no existe, lanzamos un error
    if (!loan) throw new Error("Préstamo no encontrado");

    // Si ya tiene fecha de devolución real, no podemos devolverlo de nuevo
    if (loan.getDataValue("fechaDevolucionReal")) {
        throw new Error("El préstamo ya fue devuelto");
    }

    // Establecemos la fecha de devolución real como la fecha actual
    loan.set("fechaDevolucionReal", new Date());
    // Guardamos los cambios en la base de datos
    await loan.save();

    // Recuperamos el id del libro asociado al préstamo
    const libroId = loan.getDataValue("libroId");
    // Buscamos el libro correspondiente
    const libro = await Book.findByPk(libroId);
    // Si el libro existe, incrementamos el stock en 1
    if (libro) {
        const stock = libro.getDataValue("stock");
        await libro.update({ stock: stock + 1 });
    }

    // Devolvemos el préstamo actualizado
    return loan;
};



// Método para obtener los préstamos activos
export const getActiveLoans = async () => {
    // Busca todos los préstamos que aún no tienen fecha de devolución real
    return Loan.findAll({
        where: { fechaDevolucionReal: null },
        include: [User, Book],
        // Incluye también los datos del usuario y del libro asociado
    });
};
```

```
// Método para obtener el historial completo de préstamos
export const getLoanHistory = async () => {
    // Devuelve todos los préstamos, sin filtrar, incluyendo usuario y
libro
    return Loan.findAll({
        include: [User, Book],
    });
};
```

UserService

```
import { User } from "../models";//Importamos el modelo de usuario para trabajar con la base de datos


//Crear usuario
export const createUser = async (data: {
    nombre: string;
    email: string;
    activo?: boolean;
}) => {
    return User.create(data);//Crea un nuevo registro de usuario con los datos recibidos
};


//Obtener todos los usuarios
export const getUsers = async () => {
    return User.findAll();//Devuelve todos los usuarios registrados en la base de datos
};


//Obtener usuario por ID
export const getUserById = async (id: number) => {
    return User.findByPk(id);//Busca un usuario por su clave primaria (id)
};


//Actualizar usuario
export const updateUser = async (
    id: number,
    data: Partial<{ nombre: string; email: string; activo: boolean }>
) => {
    const user = await User.findByPk(id);//Buscamos el usuario por id

    if (!user) throw new Error("Usuario no encontrado");// Si no existe lanzamos un error

    await user.update(data);//Si existe, actualizamos sus datos
```

```
    return user;//Devolvemos el usuario actualizado
};


//Eliminar usuario
export const deleteUser = async (id: number) => {
    const user = await User.findByPk(id); // Buscamos el usuario por id

    if (!user) throw new Error("Usuario no encontrado"); // Si no existe
lanzamos un error

    await user.destroy();// Eliminamos el usuario de la base de datos
};
```

Validations

-¿Por qué validations se encuentran en service? Esto se debe a que es un middleware de validación que usa Zod para validar el cuerpo de las peticiones antes de que lleguen al controlador, es decir, es un filtro que revisa los datos antes de que entren en tu lógica de negocio, es una función reutilizable.

```
import { Request, Response, NextFunction } from "express";
// Importamos los tipos de Express para tipar correctamente req, res y next

import { ZodSchema } from "zod";
// Importamos el tipo de esquema de Zod para poder recibir cualquier
esquema válido


// Middleware genérico de validación
export const validate =
    (schema: ZodSchema) => // Recibe un esquema de Zod como parámetro
        (req: Request, res: Response, next: NextFunction): void => {

            const result = schema.safeParse(req.body);
            // Valida el cuerpo de la petición contra el esquema recibido

            if (!result.success) {
                const msg = result.error.issues.map((i) =>
i.message).join(", ");
                // Si la validación falla, recogemos todos los mensajes de
error
                // y los convertimos en un solo string

                res.status(400).json({ error: msg });
                // Devolvemos respuesta 400 con los mensajes de error

                return;
                // Cortamos la ejecución del middleware
            }

            (req as any).validated = result.data;
            // Si la validación es correcta, guardamos los datos validados
            // dentro de la request para poder usarlos en el controlador

            next();
```

```
        // Continuamos con el siguiente middleware o controlador
    };
```

Schemas

-Pasa lo mismo que con validations, son schemas de validación usando Zod, porque necesitamos validar los datos que llegan desde el cliente antes que entren en tu lógica de negocio.

```
import { z } from "zod";


//Validaciones para los usuarios
export const userCreateSchema = z.object({
    nombre: z.string().min(1, "El nombre es obligatorio"),//Debe de ser una string y no puede estar vacio
    email: z.string().email("Email no válido"),//Debe de tener formato de email valido
    activo: z.boolean().optional(), //Campo opcional
});


export const userUpdateSchema = z.object({
    nombre: z.string().min(1).optional(),//Puede venir o no, pero si viene tiene que ser una string y no puede estar vacio
    email: z.string().email().optional(),//Puede venir o no, pero si viene tienen que ser una cadena y con formato de email
    activo: z.boolean().optional(),//Campo opcional
});

//Validaciones para los libros
export const bookCreateSchema = z.object({
    titulo: z.string().min(1, "El título es obligatorio"),//El titulo es obligatorio y no puede estar vacio
    autor: z.string().min(1, "El autor es obligatorio"),//El autor es obligatorio y no puede estar vacio
    stock: z.number().int().nonnegative("El stock no puede ser negativo"),//El stock debe de ser un enter positivo y ademas obligatorio
});

export const bookUpdateSchema = z.object({
    titulo: z.string().min(1).optional(),//El titulo puede venir o no, pero si viene debe de ser de tipo string y ademas que no venga vacio
    autor: z.string().min(1).optional(),//El autor puede venir o no, pero si viene debe de ser de tipo string y ademas que no venga vacio
    stock: z.number().int().nonnegative().optional(),//El stock debe de ser
```

```
un enter positivo y ademas puede venir o no
});


//Validaciones para prestamos
export const loanCreateSchema = z.object({
    usuarioId: z.number().int().positive("usuarioId inválido"),//El id del
usuario debe de ser obligatorio ademas de ser un numero positivo
    libroId: z.number().int().positive("libroId inválido"),//El id del
libro debe de ser obligatorio ademas de ser un numero positivo
    diasPrestamo: z.number().int().positive("diasPrestamo
inválido"),//Cuantos dias de prestamo va a tener, debe de ser obligatorio y
ademas un numero entero
});

export const loanUpdateSchema = z.object({
    fechaDevolucionPrevista: z.coerce.date().optional(),//debera de ser una
fecha y ademas opcional
    fechaDevolucionReal: z.coerce.date().optional(),//debera de ser una
fecha y ademas opcional
});
```

Creación de controladores

-El controlador es aquella capa que recibe la petición HTTP, extrae los datos de la request y llama al servicio correspondiente con la petición, devolviendo finalmente la respuesta al cliente, debemos de tener en cuenta estos tres parámetros en los controladores:

- Request:Es un objeto que contiene toda la información que llega de fuera, es decir, la petición.
- Response:Es la respuesta que va a devolver el servidor al cliente.
- NextFunction:Es una función que express te da para pasar la ejecución al siguiente middleware para enviar errores al middleware de manejo de errores.

BookController

```
import { Request, Response, NextFunction } from "express";//Importamos los tipos de Express para tipar correctamente los controladores
import * as bookService from "../service/bookService";//Importamos el servicio de libros que contiene la lógica de negocio

export const createBook = async (req: Request, res: Response, next: NextFunction) => {//Controlador que crea un libro nuevo
    try {
        const data = (req as any).validated ?? req.body;//Obtenemos los datos validados o el body si no hay validación
        const book = await bookService.createBook(data);//Creamos el libro usando el servicio
        res.status(201).json(book);//Respondemos con el libro creado y código 201
    } catch (err) {
        next(err);//Pasamos el error al middleware de manejo de errores
    }
};

export const getBooks = async (_req: Request, res: Response, next: NextFunction) => {//Controlador que devuelve todos los libros
    try {
        const books = await bookService.getBooks();//Obtenemos todos los libros desde el servicio
        res.json(books);//Respondemos con la lista de libros
    } catch (err) {
```

```
        next(err);//Pasamos el error al middleware de manejo de errores
    }
};

export const getBook = async (req: Request, res: Response, next:
NextFunction) => {//Controlador que devuelve un libro por id
    try {
        const id = Number(req.params.id);//Convertimos el parámetro id a
número
        const book = await bookService.getBookById(id);//Buscamos el libro
por id
        if (!book) return res.status(404).json({ error: "Libro no
encontrado" });//Si no existe, devolvemos 404
        res.json(book);//Si existe, lo devolvemos
    } catch (err) {
        next(err);//Pasamos el error al middleware de manejo de errores
    }
};

export const updateBook = async (req: Request, res: Response, next:
NextFunction) => {//Controlador que actualiza un libro
    try {
        const id = Number(req.params.id);//Obtenemos el id del libro a
actualizar
        const data = (req as any).validated ?? req.body;//Obtenemos los
datos validados o el body
        const book = await bookService.updateBook(id, data);//Actualizamos
el libro mediante el servicio
        res.json(book);//Devolvemos el libro actualizado
    } catch (err) {
        next(err);//Pasamos el error al middleware de manejo de errores
    }
};

export const deleteBook = async (req: Request, res: Response, next:
NextFunction) => {//Controlador que elimina un libro
    try {
        const id = Number(req.params.id);//Obtenemos el id del libro a
eliminar
        await bookService.deleteBook(id);//Eliminamos el libro mediante el
servicio
        res.status(204).send();//Respondemos con 204 (sin contenido)
    } catch (err) {
```

```
        next(err);//Pasamos el error al middleware de manejo de errores
    }
};
```

LoanController

```
import { Request, Response, NextFunction } from "express";//Importamos los tipos de Express para tipar correctamente los controladores
import * as loanService from "../service/loanService";//Importamos el servicio de préstamos que contiene la lógica de negocio

export const createLoan = async (req: Request, res: Response, next: NextFunction) => {//Controlador que crea un préstamo nuevo
    try {
        const data = (req as any).validated ?? req.body;//Obtenemos los datos validados o el body si no hay validación
        const loan = await loanService.createLoan(data);//Creamos el préstamo usando el servicio
        res.status(201).json(loan);//Respondemos con el préstamo creado y código 201
    } catch (err) {
        next(err);//Pasamos el error al middleware de manejo de errores
    }
};

export const getLoans = async (_req: Request, res: Response, next: NextFunction) => {//Controlador que devuelve todos los préstamos
    try {
        const loans = await loanService.getLoans();//Obtenemos todos los préstamos desde el servicio
        res.json(loans);//Respondemos con la lista de préstamos
    } catch (err) {
        next(err);//Pasamos el error al middleware de manejo de errores
    }
};

export const getLoan = async (req: Request, res: Response, next: NextFunction) => {//Controlador que devuelve un préstamo por id
    try {
        const id = Number(req.params.id);//Convertimos el parámetro id a número
        const loan = await loanService.getLoanById(id);//Buscamos el préstamo por id
        if (!loan) return res.status(404).json({ error: "Préstamo no encontrado" });//Si no existe, devolvemos 404
        res.json(loan);//Si existe, lo devolvemos
```

```
    } catch (err) {
        next(err);//Pasamos el error al middleware de manejo de errores
    }
};

export const updateLoan = async (req: Request, res: Response, next:
NextFunction) => {//Controlador que actualiza un préstamo
    try {
        const id = Number(req.params.id);//Obtenemos el id del préstamo a
actualizar
        const data = (req as any).validated ?? req.body;//Obtenemos los
datos validados o el body
        const loan = await loanService.updateLoan(id, data);//Actualizamos
el préstamo mediante el servicio
        res.json(loan);//Devolvemos el préstamo actualizado
    } catch (err) {
        next(err);//Pasamos el error al middleware de manejo de errores
    }
};

export const deleteLoan = async (req: Request, res: Response, next:
NextFunction) => {//Controlador que elimina un préstamo
    try {
        const id = Number(req.params.id);//Obtenemos el id del préstamo a
eliminar
        await loanService.deleteLoan(id);//Eliminamos el préstamo mediante
el servicio
        res.status(204).send();//Respondemos con 204 (sin contenido)
    } catch (err) {
        next(err);//Pasamos el error al middleware de manejo de errores
    }
};

export const returnLoan = async (req: Request, res: Response, next:
NextFunction) => {//Controlador que marca un préstamo como devuelto
    try {
        const id = Number(req.params.id);//Obtenemos el id del préstamo
        const loan = await loanService.returnLoan(id);//Marcamos el
préstamo como devuelto mediante el servicio
        res.json(loan);//Devolvemos el préstamo actualizado
    } catch (err) {
        next(err);//Pasamos el error al middleware de manejo de errores
    }
```

```
};

export const getActiveLoans = async (_req: Request, res: Response, next:
NextFunction) => {//Controlador que devuelve los préstamos activos
    try {
        const loans = await loanService.getActiveLoans();//Obtenemos los
préstamos activos desde el servicio
        res.json(loans);//Respondemos con la lista de préstamos activos
    } catch (err) {
        next(err);//Pasamos el error al middleware de manejo de errores
    }
};

export const getLoanHistory = async (_req: Request, res: Response, next:
NextFunction) => {//Controlador que devuelve el historial completo de
préstamos
    try {
        const loans = await loanService.getLoanHistory();//Obtenemos todos
los préstamos desde el servicio
        res.json(loans);//Respondemos con el historial completo
    } catch (err) {
        next(err);//Pasamos el error al middleware de manejo de errores
    }
};
```

UserController

```
import { Request, Response, NextFunction } from "express";//Importamos los tipos de Express para tipar correctamente los controladores
import * as userService from "../service/userService";//Importamos el servicio de usuarios que contiene la lógica de negocio

export const createUser = async (req: Request, res: Response, next: NextFunction) => {//Controlador que crea un usuario nuevo
    try {
        const data = (req as any).validated ?? req.body;//Obtenemos los datos validados o el body si no hay validación
        const user = await userService.createUser(data);//Creamos el usuario usando el servicio
        res.status(201).json(user);//Respondemos con el usuario creado y código 201
    } catch (err) {
        next(err);//Pasamos el error al middleware de manejo de errores
    }
};

export const getUsers = async (_req: Request, res: Response, next: NextFunction) => {//Controlador que devuelve todos los usuarios
    try {
        const users = await userService.getUsers();//Obtenemos todos los usuarios desde el servicio
        res.json(users);//Respondemos con la lista de usuarios
    } catch (err) {
        next(err);//Pasamos el error al middleware de manejo de errores
    }
};

export const getUser = async (req: Request, res: Response, next: NextFunction) => {//Controlador que devuelve un usuario por id
    try {
        const id = Number(req.params.id);//Convertimos el parámetro id a número
        const user = await userService.getUserById(id);//Buscamos el usuario por id
        if (!user) return res.status(404).json({ error: "Usuario no encontrado" });//Si no existe, devolvemos 404
        res.json(user);//Si existe, lo devolvemos
```

```
    } catch (err) {
        next(err);//Pasamos el error al middleware de manejo de errores
    }
};

export const updateUser = async (req: Request, res: Response, next:
NextFunction) => {//Controlador que actualiza un usuario
    try {
        const id = Number(req.params.id);//Obtenemos el id del usuario a
actualizar
        const data = (req as any).validated ?? req.body;//Obtenemos los
datos validados o el body
        const user = await userService.updateUser(id, data);//Actualizamos
el usuario mediante el servicio
        res.json(user);//Devolvemos el usuario actualizado
    } catch (err) {
        next(err);//Pasamos el error al middleware de manejo de errores
    }
};

export const deleteUser = async (req: Request, res: Response, next:
NextFunction) => {//Controlador que elimina un usuario
    try {
        const id = Number(req.params.id);//Obtenemos el id del usuario a
eliminar
        await userService.deleteUser(id);//Eliminamos el usuario mediante
el servicio
        res.status(204).send();//Respondemos con 204 (sin contenido)
    } catch (err) {
        next(err);//Pasamos el error al middleware de manejo de errores
    }
};
```

Routes

-Los routes son la capa de la aplicación que define qué URL existe, que método HTTP usa, que controlador se ejecuta y además que middlewares se aplican además de indicarselo al Swagger para generar la documentación.

BookRoutes

```
import { Router } from "express";
import {
    createBook,
    getBooks,
    getBook,
    updateBook,
    deleteBook,
} from "../controllers/bookController";
import { validate } from "../service/validate";
import { bookCreateSchema, bookUpdateSchema } from "../service/schemas";

const router = Router();

/**
 * @swagger
 * tags:
 *   name: Books
 *   description: Operaciones CRUD para libros
 */

/**
 * @swagger
 * /api/books:
 *   get:
 *     summary: Obtener todos los libros
 *     tags: [Books]
 *     responses:
 *       200:
 *         description: Lista de libros
 *   post:
 *     summary: Crear un nuevo libro
 *     tags: [Books]
```

```
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             type: object
 *             properties:
 *               titulo:
 *                 type: string
 *               autor:
 *                 type: string
 *               stock:
 *                 type: number
 *     responses:
 *       201:
 *         description: Libro creado correctamente
 */
router.post("/", validate(bookCreateSchema), createBook);
router.get("/", getBooks);

/**
 * @swagger
 * /api/books/{id}:
 *   get:
 *     summary: Obtener un libro por ID
 *     tags: [Books]
 *     parameters:
 *       - in: path
 *         name: id
 *         required: true
 *         schema:
 *           type: integer
 *     responses:
 *       200:
 *         description: Libro encontrado
 *       404:
 *         description: Libro no encontrado
 *
 *   put:
 *     summary: Actualizar un libro
 *     tags: [Books]
 *     parameters:
 *       - in: path
```

```
 *         name: id
 *         required: true
 *         schema:
 *           type: integer
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             type: object
 *             properties:
 *               titulo:
 *                 type: string
 *               autor:
 *                 type: string
 *               stock:
 *                 type: number
 *     responses:
 *       200:
 *         description: Libro actualizado
 *
 *   delete:
 *     summary: Eliminar un libro
 *     tags: [Books]
 *     parameters:
 *       - in: path
 *         name: id
 *         required: true
 *         schema:
 *           type: integer
 *     responses:
 *       204:
 *         description: Libro eliminado
 *       400:
 *         description: No se puede eliminar un libro con préstamos activos
 */
router.get("/:id", getBook);
router.put("/:id", validate(bookUpdateSchema), updateBook);
router.delete("/:id", deleteBook);

export default router;
```

LoanRoutes

```
import { Router } from "express";
import {
    createLoan,
    getLoans,
    getLoan,
    updateLoan,
    deleteLoan,
    returnLoan,
    getActiveLoans,
    getLoanHistory,
} from "../controllers/loanController";
import { validate } from "../service/validate";
import { loanCreateSchema, loanUpdateSchema } from "../service/schemas";

const router = Router();

/**
 * @swagger
 * tags:
 *   name: Loans
 *   description: Operaciones CRUD y gestión de préstamos
 */

/**
 * @swagger
 * /api/loans:
 *   get:
 *     summary: Obtener todos los préstamos
 *     tags: [Loans]
 *     responses:
 *       200:
 *         description: Lista de préstamos
 *
 *   post:
 *     summary: Crear un préstamo (reduce stock del libro)
 *     tags: [Loans]
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
```

```
 *           schema:
 *             type: object
 *             properties:
 *               usuarioId:
 *                 type: integer
 *               libroId:
 *                 type: integer
 *               diasPrestamo:
 *                 type: integer
 *     responses:
 *       201:
 *         description: Préstamo creado correctamente
 */
router.post("/", validate(loanCreateSchema), createLoan);
router.get("/", getLoans);

/**
 * @swagger
 * /api/loans/{id}:
 *   get:
 *     summary: Obtener un préstamo por ID
 *     tags: [Loans]
 *     parameters:
 *       - in: path
 *         name: id
 *         required: true
 *         schema:
 *           type: integer
 *     responses:
 *       200:
 *         description: Préstamo encontrado
 *       404:
 *         description: Préstamo no encontrado
 *
 *   put:
 *     summary: Actualizar un préstamo
 *     tags: [Loans]
 *     parameters:
 *       - in: path
 *         name: id
 *         required: true
 *         schema:
 *           type: integer
```

```
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             type: object
 *             properties:
 *               fechaDevolucionPrevista:
 *                 type: string
 *                 format: date
 *               fechaDevolucionReal:
 *                 type: string
 *                 format: date
 *     responses:
 *       200:
 *         description: Préstamo actualizado
 *
 *   delete:
 *     summary: Eliminar un préstamo (si está activo, aumenta stock)
 *     tags: [Loans]
 *     parameters:
 *       - in: path
 *         name: id
 *         required: true
 *         schema:
 *           type: integer
 *     responses:
 *       204:
 *         description: Préstamo eliminado
 */
router.get("/:id", getLoan);
router.put("/:id", validate(loanUpdateSchema), updateLoan);
router.delete("/:id", deleteLoan);

/**
 * @swagger
 * /api/loans/{id}/devolver:
 *   post:
 *     summary: Devolver un préstamo (aumenta stock del libro)
 *     tags: [Loans]
 *     parameters:
 *       - in: path
 *         name: id
```

```
 *         required: true
 *         schema:
 *           type: integer
 *     responses:
 *       200:
 *         description: Préstamo devuelto correctamente
 */
router.post("/:id/devolver", returnLoan);

/**
 * @swagger
 * /api/loans/estado/activos:
 *   get:
 *     summary: Obtener préstamos activos (no devueltos)
 *     tags: [Loans]
 *     responses:
 *       200:
 *         description: Lista de préstamos activos
 */
router.get("/estado/activos", getActiveLoans);

/**
 * @swagger
 * /api/loans/estado/historial:
 *   get:
 *     summary: Obtener historial completo de préstamos
 *     tags: [Loans]
 *     responses:
 *       200:
 *         description: Historial de préstamos
 */
router.get("/estado/historial", getLoanHistory);

export default router;
```

UserRoutes

```
import { Router } from "express";
import {
    createUser,
    getUsers,
    getUser,
    updateUser,
    deleteUser,
} from "../controllers/userController";
import { validate } from "../service/validate";
import { userCreateSchema, userUpdateSchema } from "../service/schemas";

const router = Router();

/**
 * @swagger
 * tags:
 *   name: Users
 *   description: Operaciones CRUD para usuarios
 */

/**
 * @swagger
 * /api/users:
 *   get:
 *     summary: Obtener todos los usuarios
 *     tags: [Users]
 *     responses:
 *       200:
 *         description: Lista de usuarios
 *
 *   post:
 *     summary: Crear un nuevo usuario
 *     tags: [Users]
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             type: object
 *             properties:
 *               nombre:
```

```
 *                 type: string
 *               email:
 *                 type: string
 *               activo:
 *                 type: boolean
 *     responses:
 *       201:
 *         description: Usuario creado correctamente
 */
router.post("/", validate(userCreateSchema), createUser);
router.get("/", getUsers);

/**
 * @swagger
 * /api/users/{id}:
 *   get:
 *     summary: Obtener un usuario por ID
 *     tags: [Users]
 *     parameters:
 *       - in: path
 *         name: id
 *         required: true
 *         schema:
 *           type: integer
 *     responses:
 *       200:
 *         description: Usuario encontrado
 *       404:
 *         description: Usuario no encontrado
 *
 *   put:
 *     summary: Actualizar un usuario
 *     tags: [Users]
 *     parameters:
 *       - in: path
 *         name: id
 *         required: true
 *         schema:
 *           type: integer
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
```

```
 *           schema:
 *             type: object
 *             properties:
 *               nombre:
 *                 type: string
 *               email:
 *                 type: string
 *               activo:
 *                 type: boolean
 *     responses:
 *       200:
 *         description: Usuario actualizado
 *
 *   delete:
 *     summary: Eliminar un usuario
 *     tags: [Users]
 *     parameters:
 *       - in: path
 *         name: id
 *         required: true
 *         schema:
 *           type: integer
 *     responses:
 *       204:
 *         description: Usuario eliminado
 */
router.get("/:id", getUser);
router.put("/:id", validate(userUpdateSchema), updateUser);
router.delete("/:id", deleteUser);

export default router;
```

Server

-El archivo que podemos definir como el main, este carga la configuración, inicializa el express, configura swagger, registra las rutas, conecta con la base de datos, sincroniza los modelos, ejecuta el seed inicial y arranca el servidor, es decir, es el motor que pone en marcha la API:

```
require("dotenv").config();// Carga las variables de entorno desde el
archivo .env

const express = require("express");//Importa el servidor Express para
manejar rutas y peticiones HTTP
const sequelize = require('./service/db');//Importamos la conexión
configurada a nuestra base de datos SQLite

import { Book, User, Loan } from "./models/index.js";//Importamos los
modelos y sus relaciones para que Sequelize los registre
import swaggerUi from "swagger-ui-express";//Importa Swagger UI para
mostrar la documentación en una interfaz web
import swaggerJsdoc from "swagger-jsdoc";//Genera la especificación OpenAPI
a partir de opciones y comentarios
import { swaggerOptions } from "./swagger/swagger";//Importa la
configuración de Swagger (título, versión, rutas, etc.)
import { runSeedIfEmpty } from "./seed/autoSeed";//Función que inserta
datos iniciales si la base de datos está vacía
import userRoutes from "./routes/userRoutes";//Importa las rutas
relacionadas con usuarios
import bookRoutes from "./routes/bookRoutes";//Importa las rutas
relacionadas con libros
import loanRoutes from "./routes/loanRoutes";//Importa las rutas
relacionadas con préstamos



const app = express();//Inicializa el servidor Express

const swaggerSpec = swaggerJsdoc(swaggerOptions);//Genera la documentación
Swagger a partir de las opciones configuradas

app.use("/api-docs", swaggerUi.serve,
swaggerUi.setup(swaggerSpec));//Habilita la ruta /api-docs para visualizar
la documentación de la API
```

```
app.use(express.json());//Permite que Express reciba y procese JSON en las peticiones



app.use("/api/books", bookRoutes);//Registra las rutas de libros bajo el prefijo /api/books
app.use("/api/users", userRoutes);//Registra las rutas de usuarios bajo el prefijo /api/users
app.use("/api/loans", loanRoutes);//Registra las rutas de préstamos bajo el prefijo /api/loans



const PORT = process.env.port || 4000;//Define el puerto del servidor, usando .env o 4000 por defecto



const start = async () => {//Función principal que inicia la aplicación

    try {
        await sequelize.authenticate();//Verifica la conexión con la base de datos
        console.log("Conexión a SQLite OK");//Mensaje de confirmación en consola

        await sequelize.sync({ alter: true });//Sincroniza los modelos con la base de datos (crea o actualiza tablas)
        await runSeedIfEmpty();//Inserta datos iniciales si la base está vacía
        console.log("Modelos sincronizados");//Confirma que las tablas están listas

        app.listen(PORT, () => {
            //Inicia el servidor en el puerto definido

            console.log(`Servidor corriendo en el puerto ${PORT}
`);//Mensaje indicando que el servidor está activo
        });

    } catch (error) {
        console.error("Error iniciando la app:");//Muestra un mensaje si
```

```
ocurre un error al iniciar
        process.exit(1);//Finaliza la ejecución de la aplicación
    }
};



start();//Ejecuta la función principal para iniciar el servidor
```