Nest js Primeros pasos

Nest JS Primeros Pasos

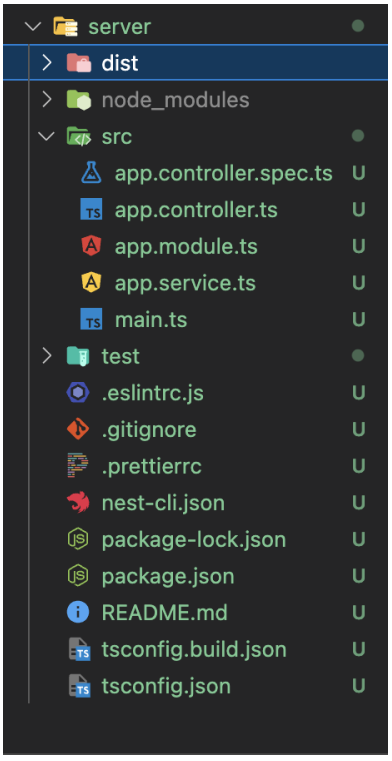
> 💡 **Implemente la siguiente guia teniendo en cuenta sus bases de datos y colecciones**.

Creación del proyecto

Para la creacion de un proyecto Nest js iniciaremos con los siguietes comandos en la terminal.

```
$ nest new server
$ cd server
$ npm run start:dev
```

En este punto debemos encontrar una serie de carpetas con el siguiente aspecto.



Deberiamos poder acceder al host local desde el navegador a traves de http://localhost:3000 y ver un mensaje de “Hello World!”

Hello World!

A continuacion se explica el funcionamiento de la palicacion a traves de los controladores y los servicios en NestJS

Servicios y Controladores

Los controladores y servicios en NestJS son elementos fundamentales para crear aplicaciones web.

Controladores:
Los controladores son responsables de manejar las solicitudes HTTP y de definir las rutas de la aplicación. Cada controlador se encarga de un conjunto específico de rutas y se asocia con una clase decorada con el decorador `@Controller()`. Dentro de los controladores, se definen los métodos que manejan las solicitudes y las respuestas HTTP, utilizando decoradores como `@Get()`, `@Post()`, `@Put()`, `@Delete()`, entre otros. Estos métodos pueden retornar datos, renderizar vistas o redireccionar a otras rutas.

Servicios:
Los servicios en NestJS son clases que contienen la lógica de negocio de una aplicación. Estos servicios pueden ser inyectados en los controladores u otros servicios mediante el mecanismo de inyección de dependencias. Para definir un servicio, se utiliza el decorador `@Injectable()`. Los servicios pueden tener métodos que realizan operaciones específicas y se encargan de interactuar con la capa de datos, como por ejemplo, realizar consultas a una base de datos.

En resumen, los controladores se encargan de manejar las solicitudes HTTP y definir las rutas de la aplicación, mientras que los servicios contienen la lógica de negocio y se encargan de realizar operaciones específicas. La separación de responsabilidades entre controladores y servicios permite crear aplicaciones más organizadas y mantenibles.

Archivo app.service.ts

```
import { Injectable } from '@nestjs/common';

@Injectable()
export class AppService {
  getHello(): string {
    return 'Hello World!';
  }
}
```

El código anterior muestra un archivo llamado `app.service.ts` en un proyecto de Nest JS. En este archivo, se define una clase llamada `AppService`. Esta clase está

decorada con `@Injectable()`, lo que significa que puede ser inyectada como una dependencia en otros componentes de la aplicación.

Dentro de la clase `AppService`, hay un método llamado `getHello()`. Este método devuelve un mensaje de tipo `string`, en este caso, "Hello World!".

El propósito de este servicio es proporcionar la lógica de negocio relacionada con una funcionalidad específica de la aplicación. En este caso, el método `getHello()` simplemente devuelve un saludo estático, pero en una aplicación real, este servicio podría realizar operaciones más complejas, como acceder a una base de datos o interactuar con otros servicios.

Al separar la lógica de negocio en servicios, se sigue el principio de "separación de preocupaciones", lo que facilita el mantenimiento y la escalabilidad de la aplicación. Los servicios pueden ser inyectados en controladores u otros servicios, lo que permite reutilizar y compartir la funcionalidad en diferentes partes de la aplicación.

Es importante destacar que este es solo un ejemplo básico de un servicio en Nest JS, y en una aplicación real, se pueden tener múltiples servicios con diferentes métodos y funcionalidades.

Archivo app.controller.ts

```
import { Controller, Get } from '@nestjs/common';
import { AppService } from './app.service';

@Controller()
export class AppController {
  constructor(private readonly appService: AppService) {}

  @Get()
  getHello(): string {
    return this.appService.getHello();
  }
}
```

El código anterior muestra un archivo llamado `app.controller.ts` en un proyecto de Nest JS. En este archivo, se define una clase llamada `AppController`. Esta clase está decorada con `@Controller()`, lo que indica que es un controlador.

En el constructor del controlador, se inyecta una instancia del servicio `AppService` utilizando la sintaxis `private readonly appService: AppService`. Esto permite que el

controlador utilice los métodos y la lógica definida en el servicio.

Dentro de la clase `AppController`, hay un método llamado `getHello()`, decorado con `@Get()`. Este método maneja las solicitudes HTTP GET a la ruta raíz (`/`) de la aplicación. Cuando se realiza una solicitud GET a la ruta raíz, este método invoca al método `getHello()` del servicio `AppService` y devuelve el resultado.

En resumen, el controlador `AppController` maneja las solicitudes GET a la ruta raíz y utiliza el servicio `AppService` para obtener el mensaje "Hello World!" que se devuelve como respuesta.

Creardo controladores y servicios de nuestra App

Para generar los archivos `song.controller.ts` y `song.service.ts` en Nest a través de la consola o bash, puedes utilizar los siguientes comandos, pero es recomendable que se genere primero el módulo para que este se agregue automáticamente a la aplicación, si deseas hacerlo de esta manera por favor remítete a la siguiente seccion donde se genera el **Módulo Song**:

1. Para generar el archivo `song.controller.ts`:

```
$ nest generate controller song
```

Este comando generará automáticamente el archivo `song.controller.ts` en la carpeta correspondiente de tu proyecto Nest.

1. Para generar el archivo `song.service.ts`:

```
$ nest generate service song
```

Este comando generará automáticamente el archivo `song.service.ts` en la carpeta correspondiente de tu proyecto Nest.

Al ejecutar estos comandos, asegúrate de estar ubicado en la raíz de tu proyecto Nest en la terminal o bash.

Archivo song.controller.ts

```
import { Controller, Get, Post, Body } from '@nestjs/common';
import { SongService } from './song.service';
import { CreateSongDto } from './dto/create-song.dto';

@Controller('song')
export class SongController {
  constructor(private readonly songService: SongService) {}

  @Get()
  findAll(): string {
    return this.songService.findAll();
  }

  @Post()
  create(@Body() createSongDto: CreateSongDto): string {
    return this.songService.create(createSongDto);
  }
}
```

El código anterior muestra un archivo llamado `song.controller.ts` en un proyecto de Nest JS. En este archivo, se define una clase llamada `SongController`. Esta clase está decorada con `@Controller('song')`, lo que indica que el controlador maneja las solicitudes relacionadas con la ruta '/song'.

En el constructor del controlador, se inyecta una instancia del servicio `SongService` utilizando la sintaxis `private readonly songService: SongService`. Esto permite que el controlador utilice los métodos y la lógica definida en el servicio.

Dentro de la clase `SongController`, hay dos métodos decorados con `@Get()` y `@Post()` respectivamente.

El método `findAll()` está decorado con `@Get()`. Este método maneja las solicitudes HTTP GET a la ruta '/song'. Cuando se realiza una solicitud GET a esta ruta, el método invoca al método `findAll()` del servicio `SongService` y devuelve el resultado.

El método `create()` está decorado con `@Post()`. Este método maneja las solicitudes HTTP POST a la ruta '/song'. Cuando se realiza una solicitud POST a esta ruta, el método recibe los datos enviados en el cuerpo de la solicitud (`createSongDto`) y los pasa al método `create()` del servicio `SongService`. El resultado de este método se devuelve como respuesta.

Archivo song.service.ts

```
import { Injectable } from '@nestjs/common';
import { CreateSongDto } from './dto/create-song.dto';
```

```
@Injectable()
export class SongService {
  private songs: any[] = [];

  findAll(): string {
    return 'Get all songs';
  }

  create(createSongDto: CreateSongDto): string {
    this.songs.push(createSongDto);
    return 'Song created successfully';
  }
}
```

El código anterior muestra un archivo llamado `song.service.ts` en un proyecto de Nest JS. En este archivo, se define una clase llamada `SongService`. Esta clase está decorada con `@Injectable()`, lo que significa que puede ser inyectada como una dependencia en otros componentes de la aplicación.

Dentro de la clase `SongService`, se declara una propiedad privada `songs` que es una matriz vacía inicialmente. Esta propiedad se utiliza para almacenar las canciones.

El método `findAll()` devuelve un mensaje de tipo `string` que indica que se están obteniendo todas las canciones.

El método `create()` recibe un objeto `createSongDto` del tipo `CreateSongDto` y lo agrega a la matriz `songs`. Luego, devuelve un mensaje de tipo `string` que indica que la canción se creó exitosamente.

En resumen, el controlador `SongController` maneja las solicitudes GET y POST a la ruta '/song' y utiliza el servicio `SongService` para obtener todas las canciones y crear una nueva canción respectivamente.

Vinculación con el resto de la aplicación

Guía para crear `songModule`

A continuación, se detallan los pasos para crear el módulo `songModule` en tu aplicación de Nest JS, así como vincular el controlador y el servicio de `song` y conectar `songModule` con el resto de la aplicación.

Paso 1: Generar el módulo y los archivos necesarios

En la terminal, ejecuta el siguiente comando para generar el módulo `songModule` y los archivos `song.controller.ts` y `song.service.ts`:

```
$ nest generate module song
$ nest generate controller song
$ nest generate service song
```

Esto generará automáticamente los archivos necesarios en la estructura de carpetas de tu proyecto Nest.

Paso 2: Vincular el controlador y el servicio de `song`

Abre el archivo `song.controller.ts` y actualiza el contenido con el siguiente código:

```
import {
  Controller,
  Get,
  Post,
  Body,
  Param,
  Put,
  Delete,
  Patch,
} from '@nestjs/common';
import { SongService } from './song.service';

@Controller('songs')
export class SongController {
  constructor(private readonly songService: SongService) {}
  // find one by id findOne(id)
  @Get(':id')
  findOne(@Param('id') id: number): string {
    return this.songService.findOne(id);
  }

  @Get()
  findAll(): string {
    return this.songService.findAll();
  }

  @Post()
  create(@Body() createSongDto): string {
    return this.songService.create(createSongDto);
  }

  @Patch(':id')
  update(@Param('id') id: number, @Body() updateSong): string {
```

```
    return this.songService.update(id, updateSong);
  }

  @Delete(':id')
  delete(@Param('id') id: number): string {
    return this.songService.delete(id);
  }
}
```

Luego, abre el archivo `song.service.ts` y actualiza el contenido con el siguiente código:

```
import { Injectable } from '@nestjs/common';

@Injectable()
export class SongService {
  private songs: any[] = [];

  findOne(id: number): string {
    return `Get song with id ${id}`;
  }

  findAll(): string {
    return 'Get all songs';
  }

  create(createSongDto): string {
    this.songs.push(createSongDto);
    return 'Song created successfully';
  }

  update(id, updateSong): string {
    return 'Song updated successfully';
  }

  delete(id): string {
    return 'Song deleted successfully';
  }
}
```

Paso 3: Vincular `songModule` al resto de la aplicación

Abre el archivo `app.module.ts` y actualiza el contenido para importar y agregar `songModule` a los módulos importados:

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { SongModule } from './song/song.module'; // Agrega esta línea
```

```
@Module({
  imports: [SongModule], // Agrega SongModule aquí
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

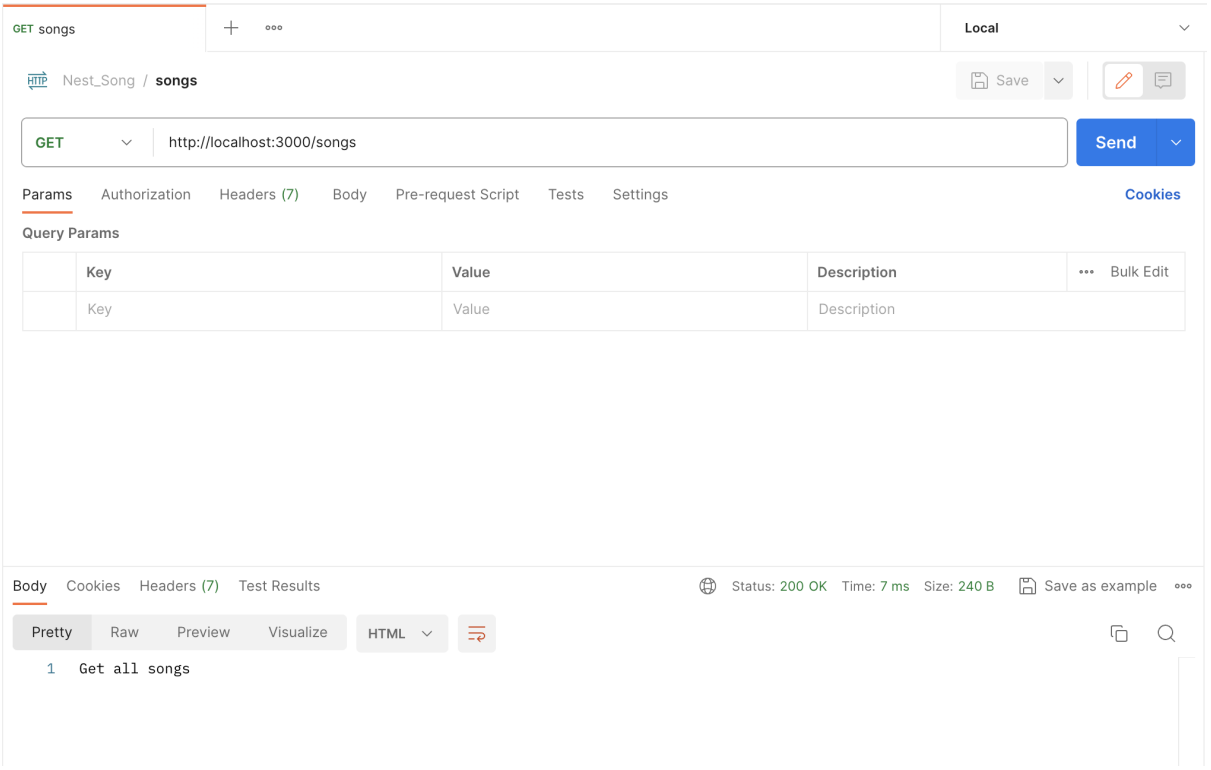
Con estos pasos, has creado el módulo `songModule`, vinculado el controlador y el servicio de `song`, y conectado `songModule` al resto de la aplicación.

Recuerda que este es solo un ejemplo básico para crear y vincular módulos en Nest JS. En una aplicación real, puedes tener más funcionalidades y configuraciones específicas según tus necesidades.

¡Ahora estás listo para seguir construyendo tu aplicación de Nest JS con el módulo `songModule`!

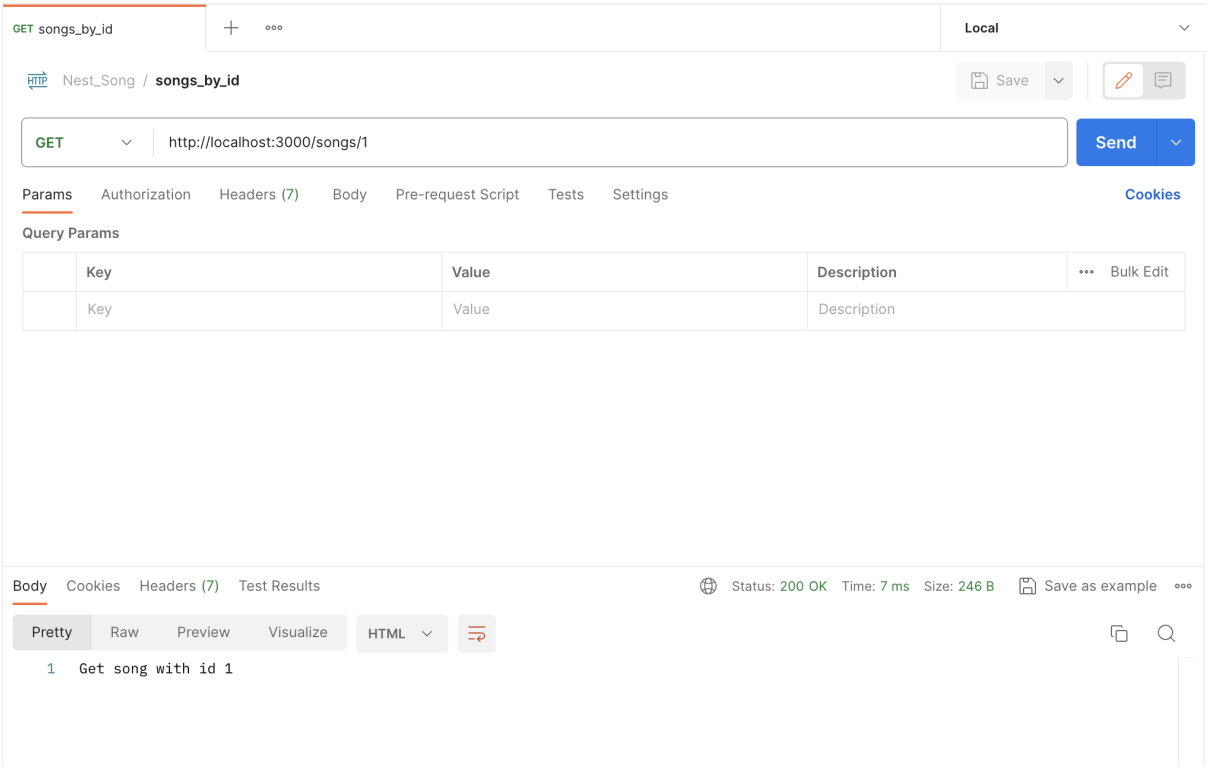
Pruebas de rutas con Postman

Ruta para obtener todas las canciones
GET /songs



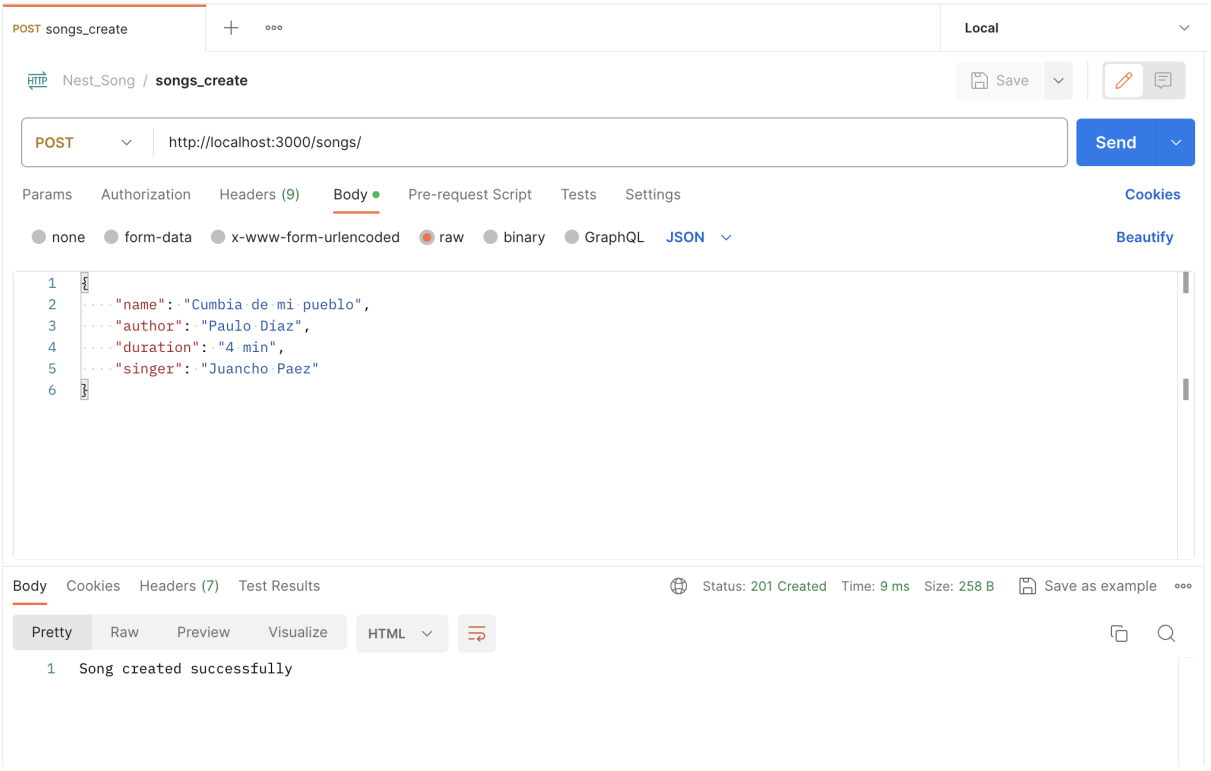
Ruta para obtener una canción por su id

GET /songs/:id



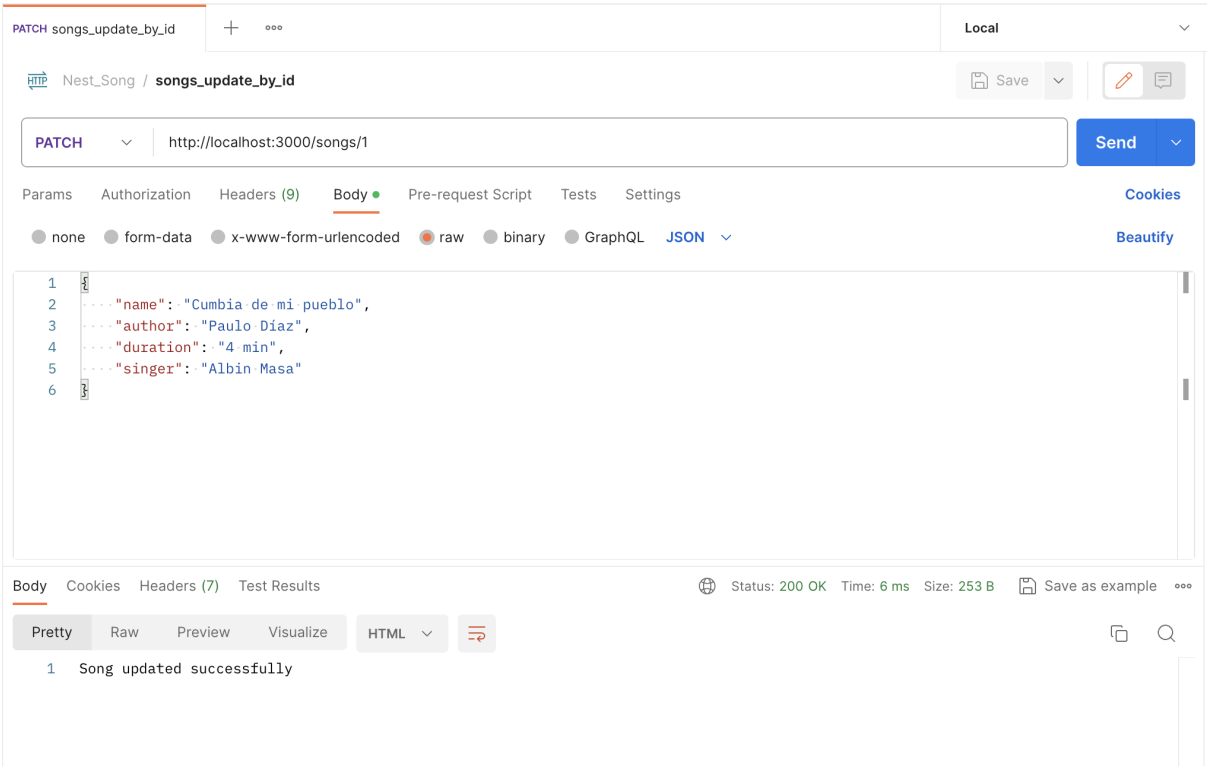
Ruta para agregar una canción

POST /songs



Ruta para actualizar una canción

PATCH /users/:id



Ruta para eliminar una canción por su id

DELETE /songs/:id