# Guía de conocimiento para la implementación de soluciones informáticas en MongoDB y NestJS

**Deiber Santiago Cardenas Matamoros**
Ingeniería de Sistemas
Universidad Santo Tomás
Noviembre 2023
Tunja, Colombia
deiber.cardenas@usantoto.edu.co

Director
**Luz Elena Gutiérrez López**

# Contenido

## I. INTRODUCCIÓN

En la era actual, donde la tecnología evoluciona a un ritmo vertiginoso, es esencial que los profesionales del desarrollo de software estén equipados con las herramientas y conocimientos necesarios para enfrentar los desafíos del mundo digital. En este contexto, el presente trabajo de grado surge como una respuesta a la creciente demanda de capacitación especializada en el desarrollo de aplicaciones backend, específicamente haciendo uso de tecnologías de vanguardia como MongoDB y NestJS.

El objetivo general de este proyecto es desarrollar una guía de conocimiento tecnológico que funcione como un protocolo integral para la formación de recursos humanos interesados en la creación de aplicaciones con persistencia en MongoDB y empleando el marco de trabajo modular NestJS en el backend. Este enfoque no solo busca abordar la necesidad de habilidades específicas en el ámbito tecnológico, sino también proporcionar un camino estructurado y evolutivo para la adquisición de estos conocimientos.

Para alcanzar este propósito general, se han establecido objetivos específicos que guiarán el desarrollo y la implementación de la guía de conocimiento:

1. Diseñar la Estructura del Protocolo: A través de una exhaustiva revisión del estado del arte en las tecnologías NestJS y MongoDB, se pretende diseñar una estructura de protocolo que aborde los conocimientos mínimos requeridos y su evolución a lo largo del proceso de formación. Este objetivo garantiza que la guía proporcione una base sólida y actualizada para los usuarios.

2. Construir la Guía de Conocimiento: Se buscará la creación de una guía detallada que respalde los procesos de capacitación en MongoDB y NestJS. Esta guía estará diseñada de manera que oriente al lector en el proceso completo de construcción de una aplicación backend, proporcionando ejemplos y buenas prácticas.

3. Implementar la Guía Tecnológica con Ejemplos Funcionales: Además de la teoría, se implementará la guía tecnológica en un entorno práctico. Esto incluirá el desarrollo de una aplicación frontend que permita al lector verificar y aplicar los conocimientos adquiridos a través de ejemplos funcionales durante su proceso de entrenamiento.

4. Verificar con el Semillero Veritate Software: La validación práctica de la guía se llevará a cabo con la colaboración del semillero Veritate Software. Este paso crucial asegurará que la guía sea efectiva y aplicable en un entorno educativo real, permitiendo a los estudiantes del semillero construir aplicaciones basadas en el conocimiento presentado.

A medida que avanzamos en este trabajo, nos sumergiremos en los detalles de cada uno de estos objetivos, destacando su importancia en la formación de profesionales capacitados y listos para enfrentar los retos del desarrollo de software en un mundo digital en constante cambio.

## II. RECURSOS ÚTILES

- MongoDB University es una plataforma en línea que ofrece cursos y recursos gratuitos de aprendizaje relacionados con MongoDB: https://learn.mongodb.com/
- Documentación oficial de MongoDB: https://www.mongodb.com/docs
- Canal oficial de MongoDB en YouTube, es un canal bastante activo, en donde se pueden encontrar tutoriales y charlas de interés sobre MongoDB: https://youtube.com/@MongoDB
- NodeSchool.io es una plataforma en línea que proporciona tutoriales interactivos y recursos de aprendizaje para programadores interesados en aprender Node.js y JavaScript: https://nodeschool.io/
- Página oficial de TypeScript: https://www.typescriptlang.org/
- Página oficial de Express: https://expressjs.com/
- Frameworks o marcos de trabajo que están basados en Express: https://expressjs.com/en/resources/frameworks.html
- Documentación oficial de Nestjs: https://docs.nestjs.com/

## III. GLOSARIO

**MongoDB**: Una base de datos NoSQL orientada a documentos que almacena datos en formato BSON (Binary JSON).

**Documento**: Un objeto o entidad almacenada en MongoDB. Un documento consiste en un conjunto de pares clave-valor, similar a un objeto JSON.

**Colección**: Una agrupación lógica de documentos en MongoDB. Es similar a una tabla en una base de datos relacional.

**Documento anidado**: Un documento dentro de otro documento en MongoDB. Permite representar relaciones o estructuras complejas de datos.

**ID de documento**: Un identificador único asignado a cada documento en una colección de MongoDB.

**Schema**: Un esquema o modelo que define la estructura de un documento en una colección. Puede ser utilizado para validar y gestionar la consistencia de los datos.

**Mongoose**: Una biblioteca de modelado de documentos de MongoDB para Node.js. Proporciona una capa de abstracción y características adicionales para interactuar con MongoDB.

**Nestjs**: Un framework de desarrollo Backend basado en Express que utiliza TypeScript para crear aplicaciones escalables y modularizadas.

**Controlador**: Un componente en Nestjs que maneja las solicitudes HTTP y contiene la lógica de negocio para una ruta específica.

**Servicio**: Un componente en Nestjs que encapsula la lógica de negocio y se utiliza para compartir funcionalidad entre diferentes controladores.

**Middleware**: Una función o componente en Nestjs que se ejecuta antes o después del manejo de una solicitud HTTP. Se utiliza para realizar tareas como la autenticación, validación de datos, etc.

**Decorador**: Una característica de TypeScript utilizada en Nestjs para agregar metadatos o funcionalidad adicional a clases, métodos o propiedades.

**DTO (Data Transfer Object)**: Un objeto utilizado para transferir datos entre diferentes componentes de una aplicación. Ayuda a definir y validar la estructura de los datos en la comunicación entre el cliente y el servidor.

**Repositorio**: Un componente utilizado en Nestjs para interactuar con la base de datos. Proporciona métodos para realizar operaciones de lectura, escritura, actualización y eliminación en una colección.

**Indexación**: El proceso de crear índices en los campos de una colección de MongoDB. Mejora la eficiencia de las consultas al permitir búsquedas rápidas y ordenadas.

**Agregación**: Una operación avanzada en MongoDB que permite procesar y combinar datos de múltiples documentos en una colección mediante una serie de etapas de procesamiento.

**Consulta**: Una solicitud para recuperar datos de una colección de MongoDB que cumpla ciertos criterios. Se utiliza para filtrar y ordenar los documentos.

**Validación**: El proceso de asegurarse de que los datos ingresados cumplan con ciertas reglas o restricciones definidas en el esquema o modelo.

**Índice**: Una estructura de datos utilizada para mejorar la velocidad de búsqueda y recuperación de datos en una base. En MongoDB, los índices pueden crearse en uno o varios campos de una colección.
**Autenticación**: El proceso de verificar la identidad de un usuario o sistema. En MongoDB y NestJS, se pueden utilizar diferentes métodos de autenticación, como tokens de acceso, JWT (JSON Web Tokens), autenticación basada en cookies, etc.

**Autorización**: El proceso de otorgar o denegar acceso a determinados recursos o funcionalidades de una aplicación basándose en los permisos y roles asignados a un usuario.

**Desarrollo local**: El entorno de desarrollo en el que se trabaja en una máquina local antes de implementar una aplicación en un entorno de producción.

**Cache**: Un mecanismo de almacenamiento temporal de datos para mejorar el rendimiento de la aplicación. En Nestjs, se pueden utilizar diferentes estrategias de caché, como el almacenamiento en memoria o el uso de sistemas de caché externos como Redis.

**Eventos**: En Nestjs, los eventos son señales o notificaciones que se emiten y se pueden escuchar en diferentes partes de la aplicación. Ayudan a establecer una comunicación y coordinación entre diferentes módulos o componentes de la aplicación.

**Validación de datos**: El proceso de verificar la integridad y consistencia de los datos ingresados por los usuarios. Puede incluir la validación de formatos, la comprobación de valores mínimos y máximos, la verificación de restricciones y más.

**Escalabilidad**: La capacidad de una aplicación para manejar un aumento en la carga de trabajo sin degradar el rendimiento. MongoDB y Nestjs proporcionan opciones y técnicas para escalar vertical y horizontalmente una aplicación.

**Logging**: El proceso de registrar eventos y mensajes relevantes durante la ejecución de una aplicación. El registro es útil para el monitoreo, depuración y análisis de problemas en la aplicación.

**API RESTful**: Una arquitectura de diseño de API que se basa en los principios de REST (Representational State Transfer). Las API RESTful utilizan métodos HTTP como GET, POST, PUT y DELETE para manipular recursos.

**Seguridad**: La protección de una aplicación contra amenazas y vulnerabilidades. Esto implica la implementación de prácticas y medidas de seguridad, como autenticación, autorización, encriptación de datos, protección contra ataques, etc.

**Escalamiento horizontal**: El proceso de agregar más instancias de una aplicación para distribuir la carga de trabajo y mejorar la escalabilidad. En MongoDB, esto puede incluir la creación de réplicas y la distribución de datos en clústeres.

**Patrón de diseño**: Una solución probada y comúnmente aceptada para un problema recurrente en el desarrollo de software. Algunos patrones de diseño comunes en NestJS incluyen el patrón de Inyección de Dependencias (Dependency Injection), el patrón de Middleware y el patrón Controlador.

**Endpoint:** Es un punto final de comunicación en un servicio web o API. Representa una URL específica a la que se puede acceder para realizar operaciones como obtener datos, enviar información o realizar acciones específicas dentro de una aplicación web.

**Scope (Ámbito):** En el contexto del desarrollo de aplicaciones, el "scope" se refiere al alcance o la extensión de acceso que tiene un token de autenticación o una variable dentro de un programa. Define qué recursos o datos pueden ser accedidos y qué acciones pueden ser realizadas por un usuario, un proceso o una función específica en un contexto determinado.

**Gateway:** Un "gateway" es un componente que actúa como un punto de entrada centralizado para dirigir el tráfico entre diferentes sistemas, servicios o redes. En el contexto de aplicaciones web, un gateway puede ser utilizado para dirigir las solicitudes de los clientes a los servicios correspondientes, aplicar políticas de seguridad, realizar transformaciones en las solicitudes o respuestas, y controlar el flujo de datos entre diferentes partes de una aplicación distribuida.

## IV. CONTEXTO DE LAS HERRAMIENTAS

### *A. MongoDB [1]*

**Origen**: MongoDB surgió como una solución a las limitaciones de las bases de datos relacionales en el manejo de datos no estructurados o semiestructurados. Su desarrollo se basó en la idea de documentos almacenados en formato BSON (Binary JSON), lo que permite una representación ágil y flexible de los datos. Fundamentado en el lenguaje de programación C++, MongoDB se ha convertido en una de las bases de datos NoSQL más populares, ofreciendo versiones tanto en código abierto como en versiones comerciales más avanzadas.

**Características Clave**:

1. Flexibilidad: La estructura de documentos JSON/BSON permite cambios dinámicos en la estructura de los datos.
2. Escalabilidad: Ofrece escalabilidad horizontal a través de la distribución de datos en clústeres.
3. Alto Rendimiento: Diseñado para operaciones de lectura/escritura rápidas y eficientes.
4. Consultas Avanzadas: Admite consultas complejas y permite índices flexibles para optimizar el rendimiento de las consultas.

**Objetivo**: MongoDB se destaca por su capacidad para manejar aplicaciones modernas y dinámicas, desde redes sociales hasta aplicaciones empresariales. Es idóneo para entornos donde la estructura de los datos puede evolucionar con el tiempo o cuando se trata con grandes volúmenes de datos no uniformes, como datos geoespaciales o registros de eventos.

### *B. NestJS [2]*

**Origen**: NestJS se basa en principios sólidos provenientes de Angular y está construido sobre Node.js. Utiliza TypeScript como lenguaje principal, lo que aporta beneficios en cuanto a tipado estático y ofrece una sintaxis más robusta para el desarrollo de aplicaciones backend. Inspirado en Angular, NestJS adopta conceptos como módulos, inyección de dependencias y decoradores para simplificar y estructurar el código.

**Características Clave**:

1. Arquitectura Modular: Divide la aplicación en módulos para organizar y escalar el código de manera eficiente.
2. Tipado Estático: Beneficios de TypeScript para detectar errores durante la fase de desarrollo.
3. Soporte a Express: Integración transparente con Express para aprovechar su potencial en el manejo de peticiones HTTP.
4. Inyección de Dependencias: Facilita la gestión de componentes y la reutilización de código.

**Objetivo**: NestJS se posiciona como una herramienta para la construcción de servidores escalables y mantenibles, ideal para aplicaciones backend robustas y API RESTful. Su enfoque modular y su integración con TypeScript proporcionan una estructura ordenada y una experiencia de desarrollo más predecible para los desarrolladores Node.js.

## V. NEST [2]

Nest (NestJS) se presenta como un **marco de trabajo** (***framework***) que permite la construcción de aplicaciones web del lado del servidor con Node.js de manera eficiente y escalable. Se destaca por su uso de TypeScript, aunque también permite a los desarrolladores trabajar con JavaScript. Además, combina elementos de Programación Orientada a Objetos (OOP), Programación Funcional (FP) y Programación Funcional Reactiva (FRP).

En cuanto a su implementación, Nest aprovecha marcos de trabajo de servidores HTTP sólidos como Express (el predeterminado) y, en caso de ser necesario, se puede configurar para utilizar **Fastify**.

La característica distintiva de Nest radica en su nivel de abstracción superior a los marcos de trabajo comunes de Node.js (Express/Fastify), al tiempo que proporciona acceso directo a sus APIs para los desarrolladores. Esta libertad brinda la posibilidad de utilizar una amplia variedad de módulos de terceros disponibles para la plataforma subyacente.

*A. Filosofía*

En los últimos años, gracias a Node.js, JavaScript se ha consolidado como el lenguaje dominante en la web tanto para aplicaciones front-end como back-end. Esto ha dado lugar al surgimiento de proyectos destacados como **Angular**, **React** y **Vue**, los cuales mejoran la productividad de los desarrolladores y posibilitan la creación de aplicaciones front-end rápidas, testeables y extensibles. No obstante, a pesar de la existencia de numerosas bibliotecas, herramientas y utilidades de calidad para Node.js (y JavaScript en el lado del servidor), ninguna de ellas aborda de manera efectiva el problema fundamental: la arquitectura.

Nest proporciona una arquitectura de aplicación lista para usar, que permite a los desarrolladores y equipos crear aplicaciones altamente testeables, escalables, con un bajo acoplamiento y de fácil mantenimiento. Esta arquitectura se inspira en gran medida en **Angular**.

*B. Primeros pasos [3]*

En esta serie de secciones, se abordarán los conceptos fundamentales de Nest. Con el propósito de adquirir conocimiento sobre los elementos esenciales de las aplicaciones de Nest, se desarrollará una aplicación CRUD básica que incorpora diversas funcionalidades de nivel introductorio.

*1) Lenguaje*

El equipo de Nest tiene un fuerte afecto por TypeScript, pero su amor más profundo está en Node.js. Por este motivo, Nest ofrece compatibilidad tanto con TypeScript como con JavaScript puro. Nest aprovecha las características más recientes del lenguaje, lo que significa que, para utilizarlo con JavaScript convencional, se requiere el uso de un compilador como Babel.

*2) Prerrequisitos*

Es importante verificar que el sistema operativo tenga Node.js instalado, preferiblemente en su versión 18 o superior.

*3) Configuración*

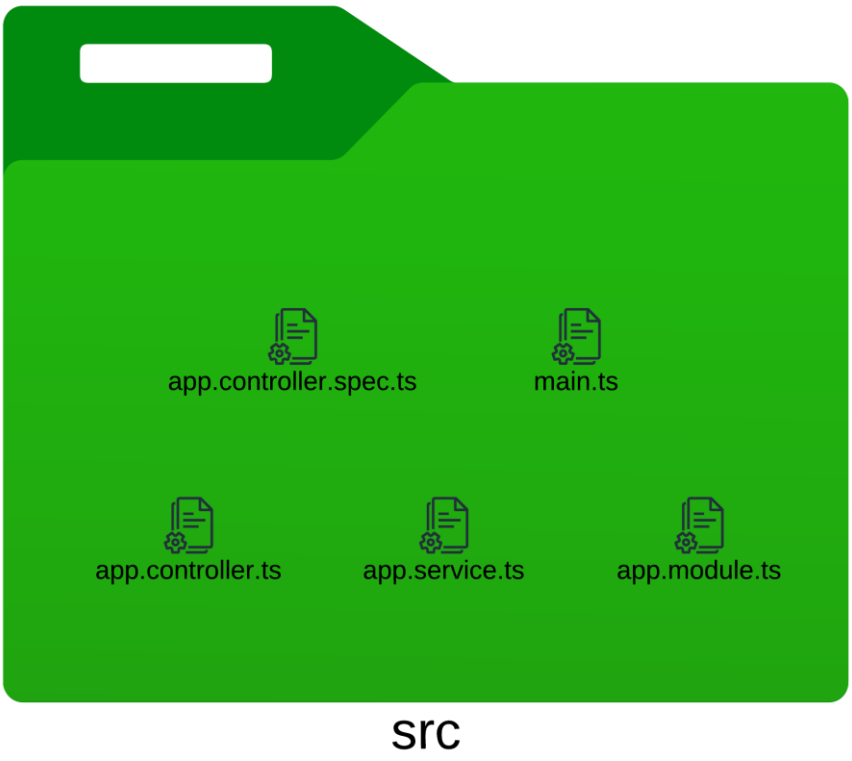
Iniciar un proyecto nuevo es un proceso relativamente simple cuando se utiliza la interfaz de línea de comandos (CLI) de Nest. Si se cuenta con npm instalado, es posible generar un proyecto de Nest completamente nuevo mediante los siguientes comandos, ejecutados en la terminal del sistema operativo:

```
$ npm i -g @nestjs/cli
$ nest new project-name
```

CONSEJO
Si se desea iniciar un nuevo proyecto con un conjunto más riguroso de características de TypeScript, se puede agregar la opción **--strict** al comando **nest new**.

Se creará el directorio "project-name", se instalarán los módulos de Node y algunos otros archivos básicos, y se creará un directorio "src/" que se llenará con varios archivos fundamentales.

*Imagen 1 Estructura base de un proyecto en NestJS*




*Fuente: Autor*

He aquí un breve resumen de esos archivos básicos:

*Tabla 1 Descripción básica de los archivos en un proyecto NestJS*

| | |
|---|---|
| app.controller.ts | Un controlador básico con una sola ruta. |
| app.controller.spec.ts | Las pruebas unitarias para el controlador. |
| app.module.ts | El módulo raíz de la aplicación. |
| app.service.ts | Un servicio básico con un único método. |
| main.ts | El archivo de entrada de la aplicación que utiliza la función central NestFactory para crear una instancia de aplicación Nest. |

*Fuente: Autor*

Dentro del archivo main.ts, se encuentra una función asincrónica encargada de iniciar la aplicación:

main.ts

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  await app.listen(3000);
}
bootstrap();
```

Para instanciar una aplicación de Nest, se emplea la clase principal llamada **NestFactory**. Esta clase presenta métodos estáticos que facilitan la creación de una instancia de la aplicación. El método **create()** retorna un objeto de la aplicación que cumple con la interfaz **INestApplication**. Este objeto proporciona un conjunto de

métodos que serán detallados en secciones posteriores. En el ejemplo mostrado en **main.ts**, simplemente se inicia un oyente HTTP, permitiendo que la aplicación espere las solicitudes HTTP entrantes.

Es relevante mencionar que un proyecto generado con la CLI de Nest establece una estructura inicial que fomenta la convención de alojar cada módulo en su propio directorio.

CONSEJO
Por defecto, en caso de que surja algún error durante la creación de la aplicación, esta se cerrará con el código 1. Si se prefiere que en su lugar genere una excepción, es posible desactivar la opción **abortOnError**, como se muestra en este ejemplo: **NestFactory.create(AppModule, { abortOnError: false })**.

*4) Plataforma HTTP*

La meta de Nest es ser un framework independiente de la plataforma. Esta independencia de la plataforma permite la creación de componentes lógicos reutilizables que los desarrolladores pueden utilizar en diversas aplicaciones. Desde un punto técnico, Nest es compatible con cualquier framework de HTTP de Node, una vez que se configure un adaptador. De manera nativa, Nest soporta dos plataformas de HTTP: express y fastify. Los usuarios pueden optar por la que mejor se ajuste a sus requerimientos.

*Tabla 2 Plataformas HTTP soportadas por NestJS*

| | |
|---|---|
| platform-express | Express es un conocido framework web minimalista para Node.js. Es una biblioteca probada en producción y con numerosos recursos desarrollados por la comunidad. El paquete **@nestjs/platform-express** se utiliza de forma predeterminada. Muchos usuarios se benefician ampliamente de Express y no necesitan realizar ninguna acción adicional para habilitarlo. |
| platform-fastify | Fastify es un marco de trabajo de alto rendimiento y bajo consumo, centrado en proporcionar la máxima eficiencia y velocidad. Lea cómo utilizarlo aquí. |

*Fuente: Autor*

Independientemente de la plataforma que se emplee, cada una presenta su propia interfaz de aplicación. Estas interfaces se conocen como **NestExpressApplication** y **NestFastifyApplication**, de acuerdo con su respectiva plataforma.

Cuando se proporciona un tipo al método **NestFactory.create()**, como se ilustra en el ejemplo a continuación, el objeto de la aplicación contendrá métodos diseñados exclusivamente para esa plataforma particular. No obstante, no es necesario especificar un tipo a menos que se tenga la intención de acceder a la API subyacente de la plataforma.

```
const app = await NestFactory.create<NestExpressApplication>(AppModule);
```

*5) Ejecutar la aplicación*

Después de finalizar la instalación, se puede emplear el siguiente comando en la terminal del sistema operativo para iniciar la aplicación y estar atento a las solicitudes HTTP entrantes.

```
$ npm run start
```

CONSEJO
Con el objetivo de aumentar la velocidad del proceso de desarrollo, logrando que las compilaciones sean hasta 20 veces más rápidas, se puede optar por el constructor SWC al incluir la bandera **-b swc** en el script de inicio. Esto se logra de la siguiente manera: **npm run start -- -b swc**.

Este comando da comienzo a la aplicación, activando el servidor HTTP para escuchar en el puerto especificado en el archivo **src/main.ts**. Una vez que la aplicación se encuentra en ejecución, se puede abrir el navegador y acceder a **http://localhost:3000/** . En ese enlace, debería visualizarse el mensaje **"Hello, World"**.

Para estar al tanto de las modificaciones en los archivos, se puede ejecutar el siguiente comando para iniciar la aplicación:

```
$ npm run start:dev
```

Este comando estará atento a los cambios en los archivos, realizando la re-compilación automática y recargando el servidor de manera automática.

*6) Linting y formato*

La CLI proporciona su mejor esfuerzo para estructurar un flujo de trabajo de desarrollo confiable a gran escala. Por lo tanto, un proyecto de Nest generado incluye un linter de código y un formateador preinstalado (respectivamente ESLint y prettier).

CONSEJO
¿No estás seguro acerca del papel de los formateadores en comparación con los linters? Aprende la diferencia aquí.

Para garantizar la máxima estabilidad y capacidad de extensión, se utilizan los paquetes base de ESLint y prettier en la línea de comandos. Esta configuración permite una integración eficaz con el IDE a través de extensiones oficiales.

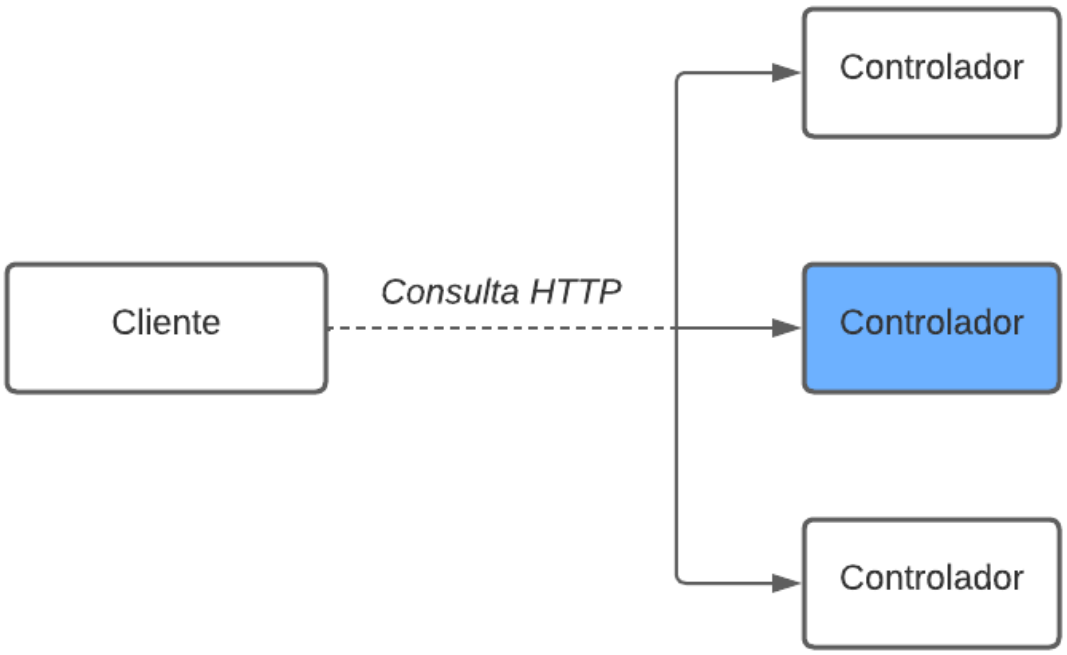
Para entornos sin interfaz gráfica donde un IDE no es relevante (Integración Continua, ganchos de Git, etc.), un proyecto de Nest incluye scripts npm listos para usar.

```
# Lint and autofix with eslint
$ npm run lint

# Format with prettier
$ npm run format
```

*C. Controladores [4]*

Los controladores tienen la responsabilidad de manejar las solicitudes y enviar las respuestas correspondientes al cliente.

*Imagen 2 Diagrama de funcionamiento de un controlador*




*Fuente: Autor*

El propósito de un controlador es recibir solicitudes específicas para la aplicación. El mecanismo de enrutamiento determina qué controlador recibe cada solicitud. Con frecuencia, cada controlador cuenta con múltiples rutas y diferentes rutas pueden llevar a cabo diversas acciones.

Para crear un controlador básico, se emplean clases y decoradores. Los decoradores vinculan las clases con metadatos necesarios y permiten que Nest genere un mapa de enrutamiento que asocia las solicitudes con los controladores correspondientes.

*1) Enrutamiento*

En el siguiente ejemplo, se emplea el decorador **@Controller()**, el cual resulta esencial para la definición de un controlador básico. Además, se define un prefijo de ruta opcional **cats**. La utilización de un prefijo de ruta en el decorador **@Controller()** permite la agrupación sencilla de un conjunto de rutas relacionadas y la reducción de la redundancia en el código. Por ejemplo, es posible optar por agrupar un conjunto de rutas que se encarguen de las interacciones con una entidad de **cat** bajo la ruta **/cats**. En este caso, el prefijo de ruta **cats** se especifica en el decorador **@Controller()** para evitar la repetición de esa parte de la ruta en cada una de las rutas subyacentes del archivo.

cats.controller.ts

```
import { Controller, Get } from '@nestjs/common';

@Controller('cats')
export class CatsController {
  @Get()
  findAll(): string {
    return 'This action returns all cats';
  }
}
```

CONSEJO
Para crear un controlador utilizando la interfaz de línea de comandos (CLI), simplemente ejecute el comando: **nest g controller [nombre-del-controlador]**.

El uso del decorador del método de solicitud HTTP **@Get()** antes del método **findAll()** obliga a Nest a crear un controlador destinado a un endpoint específico para las solicitudes HTTP. Este endpoint corresponde con el método de solicitud HTTP (en este caso, GET) y la ruta asociada a él. La ruta de un controlador se determina al combinar el prefijo (si está presente) definido para el controlador y cualquier ruta especificada en el decorador del método. Dado que se ha definido un prefijo para cada ruta (**cats**) y no se ha incluido información de la ruta en el decorador, Nest dirigirá las solicitudes **GET /cats** a este controlador. Como se ha mencionado, la ruta engloba tanto el prefijo opcional del controlador como cualquier cadena de ruta especificada en el decorador del método de la solicitud. Por ejemplo, si se tuviera un prefijo de ruta **cats** junto al decorador **@Get('breed')**, esto resultaría en la asignación de la ruta para solicitudes como **GET /cats/breed**.

En el ejemplo previo, al efectuar una solicitud GET en este endpoint, Nest direcciona la solicitud hacia el método **findAll()**, el cual ha sido definido por el usuario. Cabe resaltar que el nombre del método seleccionado es completamente libre. Aunque es necesario declarar un método al cual se vaya a conectar con la ruta, el nombre específico del método no reviste importancia alguna para Nest.

El método en cuestión retornará un código de estado 200 junto con la respuesta correspondiente, que en esta instancia se limita a ser una cadena de texto. ¿Cuál es la razón de esto? Para aclararlo, en primer lugar, se presentará el concepto de que Nest emplea dos alternativas distintas para gestionar las respuestas:

*Tabla 3 Alternativas para la gestión de respuestas en NestJS*

| | |
|---|---|
| Estándar (recomendado) | Al hacer uso de este método incorporado, cuando un controlador de solicitud retorna un objeto o una matriz en el contexto de JavaScript, se llevará a cabo automáticamente la serialización a formato JSON. No obstante, cuando se retorna un tipo primitivo en JavaScript, como una cadena, un número o un valor booleano, Nest enviará el valor sin efectuar ningún intento de serialización. Esta sencillez facilita la gestión de las respuestas, ya que basta con retornar el valor deseado, y Nest se encargará del resto.<br><br>Adicionalmente, por defecto, el código de estado de la respuesta siempre será 200, a excepción de las solicitudes **POST**, las cuales emplearán el código 201. Si se desea modificar este comportamiento, es posible hacerlo con facilidad añadiendo el decorador **@HttpCode(...)** a nivel de controlador (consultar los códigos de estado). |
| Librería especifica | Es posible emplear el objeto de respuesta propio de la biblioteca, como Express, el cual se puede inyectar en el método del controlador a través del uso del decorador **@Res()** (por ejemplo, en el método **findAll(@Res() response)**). Al adoptar este enfoque, se adquiere la capacidad de utilizar los métodos nativos proporcionados por ese objeto para gestionar las respuestas. Por ejemplo, cuando se utiliza Express, es factible construir respuestas mediante código, como **response.status(200).send()**. |

*Fuente: Autor*

ADVERTENCIA
Nest identifica cuando el controlador emplea tanto **@Res()** como **@Next()**, lo que sugiere la elección de una opción específica de la biblioteca. Si se utilizan ambos enfoques de manera simultánea, el enfoque estándar se desactivará automáticamente únicamente para esta ruta en particular y no funcionará de acuerdo con las expectativas habituales. Para poder utilizar ambas aproximaciones al mismo tiempo, por ejemplo, inyectando el objeto de respuesta para configurar exclusivamente cookies o encabezados y dejando que el resto sea manejado por el marco de trabajo, es necesario establecer la opción **passthrough** en **true** mediante el decorador **@Res({ passthrough: true })**.

*2) Objeto de la solicitud*

En muchas ocasiones, los controladores requieren acceso a la información de la solicitud realizada por el cliente. Nest facilita el acceso al objeto **request** de la plataforma subyacente, que por defecto es Express. Para obtener este objeto **request**, se puede indicar a Nest, que lo inyecte al agregar el decorador **@Req()** en el método del controlador.

cats.controller.ts

```
import { Controller, Get, Req } from '@nestjs/common';
import { Request } from 'express';

@Controller('cats')
export class CatsController {
  @Get()
  findAll(@Req() request: Request): string {
    return 'This action returns all cats';
  }
}
```

CONSEJO
Para aprovechar las definiciones de tipos de Express (como en el ejemplo del parámetro **request: Request** anterior), instale el paquete **@types/express**.

El objeto **request** representa la solicitud HTTP y contiene propiedades que abarcan la cadena de consulta de la solicitud, los parámetros, las cabeceras HTTP y el cuerpo (puedes encontrar más información aquí). En la mayoría de las situaciones, no es preciso extraer manualmente estas propiedades. En su lugar, se puede aprovechar los decoradores específicos, como **@Body()** o **@Query()**, que se encuentran disponibles de forma predeterminada. A continuación, se presenta una lista de los decoradores que se proporcionan y los objetos sin modificaciones de la plataforma que representan.

*Tabla 4 Decoradores y su equivalencia con los objetos nativos de las plataformas HTTP*

| | |
|---|---|
| @Request(), @Req() | req |
| @Response(), @Res()* | res |
| @Next() | next |
| @Session() | req.session |
| @Param(key?: string) | req.params / req.params[key] |
| @Body(key?: string) | req.body / req.body[key] |
| @Query(key?: string) | req.query / req.query[key] |
| @Headers(name?: string) | req.headers / req.headers[name] |
| @Ip() | req.ip |
| @HostParam() | req.hosts |

*Fuente: Autor*

Con el fin de garantizar la compatibilidad con las definiciones de tipos en las plataformas subyacentes de HTTP, como Express y Fastify, Nest pone a disposición los decoradores **@Res()** y **@Response()**. Cabe destacar que **@Res()** simplemente funciona como un alias de **@Response()**. Ambos decoradores exponen directamente la interfaz del objeto de respuesta nativo de la plataforma subyacente. Al emplearlos, también se requiere la importación de las definiciones de tipos correspondientes a la biblioteca subyacente (por ejemplo, **@types/express**) para aprovechar al máximo sus capacidades. Es importante tener en cuenta que al inyectar **@Res()** o **@Response()** en un método del controlador, se coloca a Nest en un modo específico de biblioteca para ese controlador, lo que implica la responsabilidad de gestionar la respuesta. En consecuencia, es necesario emitir algún tipo de respuesta efectuando una llamada en el objeto de respuesta (por ejemplo, **res.json(...) o res.send(...)**), o el servidor HTTP quedará inactivo.

*3) Recursos*

En el apartado previo, se estableció un endpoint para obtener la información del recurso **cats** utilizando la ruta **GET**. Normalmente, también será necesario ofrecer un endpoint que permita la creación de nuevos registros. Para abordar esta necesidad, es necesario crear el controlador **POST**:

cats.controller.ts

```
import { Controller, Get, Post } from '@nestjs/common';

@Controller('cats')
export class CatsController {
  @Post()
  create(): string {
    return 'This action adds a new cat';
  }

  @Get()
  findAll(): string {
    return 'This action returns all cats';
  }
}
```

Es así de simple. Nest proporciona decoradores para todos los métodos HTTP estándar: **@Get(), @Post(), @Put(), @Delete(), @Patch(), @Options() y @Head()**. Además, **@All()** define un endpoint que maneja todos ellos.

*4) Comodines de enrutamiento*

También se admiten rutas basadas en patrones. Por ejemplo, el asterisco se utiliza como comodín y coincidirá con cualquier combinación de caracteres.

```
@Get('ab*cd')
findAll() {
  return 'This route uses a wildcard';
}
```

La ruta **"ab*cd"** coincidirá con **abcd**, **ab_cd**, **abecd** y así sucesivamente. Los caracteres: **?**, **+**, *****, y **()** pueden utilizarse en una ruta, y son subconjuntos de sus contrapartes en expresiones regulares. El guion ( **-** ) y el punto ( **.** ) se interpretan de forma literal en rutas basadas en cadenas de texto.

ADVERTENCIA
Un comodín en medio de la ruta solo es compatible con Express.

*5) Código de estado*

Como se explicó previamente, el código de estado de respuesta, de manera predeterminada, es 200, a excepción de las solicitudes POST, que utilizan el código 201. Modificar este comportamiento es sencillo, ya que se puede hacer mediante la adición del decorador **@HttpCode(...)** en el nivel del controlador.

```
@Post()
@HttpCode(204)
create() {
  return 'This action adds a new cat';
}
```

CONSEJO
Importe **HttpCode** desde el paquete **@nestjs/common**.

En muchas ocasiones, el código de estado de la respuesta no es fijo, sino que está sujeto a varios factores. En situaciones así, es posible emplear un objeto de respuesta particular de la biblioteca (que se inyecta a través de **@Res()**) o, en caso de producirse un error, lanzar una excepción.

*6) Encabezados*

Para indicar un encabezado de respuesta personalizado, es factible recurrir tanto al uso del decorador **@Header()** como al empleo de un objeto de respuesta perteneciente a la biblioteca en cuestión (y efectuar la llamada directa a **res.header()**).

```
@Post()
@Header('Cache-Control', 'none')
create() {
  return 'This action adds a new cat';
}
```

**CONSEJO**
Importe **Header** desde el paquete **@nestjs/common**.

*7) Redirección*

Con el propósito de redireccionar una respuesta hacia una URL específica, es posible emplear un decorador **@Redirect()** o hacer uso de un objeto de respuesta particular relacionado con la biblioteca (y realizar una llamada directa a **res.redirect()**).

El decorador **@Redirect()** acepta dos argumentos, **url** y **statusCode**, los cuales son opcionales. Si no se proporciona el **statusCode**, el valor predeterminado es 302 (Encontrado).

```
@Get()
@Redirect('https://nestjs.com', 301)
```

CONSEJO
En ocasiones, puede resultar necesario establecer el código de estado HTTP o la URL de redirección de forma dinámica. Esto se puede lograr retornando un objeto que cumpla con la interfaz **HttpRedirectResponse** de **@nestjs/common**.

Los valores devueltos anularán cualquier argumento pasado al decorador **@Redirect()**. Por ejemplo:

```
@Get('docs')
@Redirect('https://docs.nestjs.com', 302)
getDocs(@Query('version') version) {
  if (version && version === '5') {
    return { url: 'https://docs.nestjs.com/v5/' };
  }
}
```

*8) Parámetros de enrutamiento*

Cuando es necesario recibir datos dinámicos como parte de una solicitud, las rutas con enfoque estático no resultan apropiadas. Un ejemplo de esto es cuando se necesita obtener información sobre un **cat** específico con una solicitud como **GET /cats/1**, donde **"1"** representa el **ID** del **cat**. Para definir rutas que acepten parámetros dinámicos, se pueden incorporar tokens de parámetros de ruta en la misma ruta, lo que permite capturar el valor de la variable en esa posición de la URL de la solicitud. El uso de un parámetro de ruta en el siguiente ejemplo con el decorador **@Get()** ilustra esta funcionalidad. Los parámetros de ruta declarados de esta manera permiten el acceso mediante el decorador **@Param()**, el cual se agrega a la firma del método.

CONSEJO
Las rutas con parámetros deben declararse después de cualquier ruta estática. Esto evita que las rutas parametrizadas intercepten el tráfico destinado a las rutas estáticas.

```
@Get(':id')
findOne(@Param() params: any): string {
  console.log(params.id);
  return `This action returns a #${params.id} cat`;
}
```

El decorador **@Param()** se utiliza para anotar un parámetro del método, como se muestra en el ejemplo previo utilizando **params**. Esta anotación permite que los parámetros de la ruta estén disponibles como propiedades dentro del cuerpo del método, utilizando el decorador del parámetro. En el código anterior, se ejemplifica el acceso al parámetro **id** mediante la referencia a **params.id**. También es posible transmitir un token de parámetro específico al decorador y luego referirse al parámetro de la ruta directamente por su nombre en el cuerpo del método.

CONSEJO
Importe **Param** desde el paquete **@nestjs/common**.

```
@Get(':id')
findOne(@Param('id') id: string): string {
  return `This action returns a #${id} cat`;
}
```

*9) Enrutamiento de subdominios*

El decorador **@Controller** puede tomar una opción de **host** para requerir que el **host HTTP** de las solicitudes entrantes coincida con un valor específico.

```
@Controller({ host: 'admin.example.com' })
export class AdminController {
  @Get()
  index(): string {
    return 'Admin page';
  }
}
```

ADVERTENCIA
Dado que Fastify no admite enrutadores anidados, al utilizar el enrutamiento de subdominio, se debe utilizar el adaptador (predeterminado) de Express en su lugar.

De manera análoga a cómo funcionan las rutas, la opción **hosts** tiene la capacidad de emplear tokens para capturar valores dinámicos en posiciones específicas del nombre del **host**. La utilización de un token de parámetro de **host** se ilustra en el ejemplo del decorador **@Controller()** que se presenta a continuación. Los parámetros de **host** declarados de esta manera se vuelven accesibles a través del decorador **@HostParam()**, que se debe incorporar en la firma del método.

```
@Controller({ host: ':account.example.com' })
export class AccountController {
  @Get()
  getInfo(@HostParam('account') account: string) {
    return account;
  }
}
```

*10) Scope – Ámbito*

Para individuos que proceden de diversas disciplinas en programación, puede resultar sorprendente descubrir que en Nest, la mayoría de los elementos se comparten entre las solicitudes entrantes. Esto abarca un conjunto de conexiones a bases de datos, servicios que son singleton y mantienen un estado global, entre otros. Es importante recordar que Node.js no sigue el modelo de procesamiento de solicitudes y respuestas en múltiples subprocesos sin estado, en el que cada solicitud se aborda en un subproceso independiente. Por lo tanto, el uso de instancias singleton es completamente seguro para las aplicaciones.

No obstante, existen situaciones específicas en las que se busca una duración de la solicitud basada en el controlador, como en el caso de implementaciones de almacenamiento en caché por solicitud en aplicaciones GraphQL, el seguimiento de solicitudes o la gestión de múltiples inquilinos. Se puede obtener información sobre cómo controlar estos ámbitos en el siguiente recurso.

*11) Asincronía*

El JavaScript moderno extrae los datos en su mayoría de forma asincrónica. Por eso, Nest admite y funciona bien con funciones asíncronas. Cada función asincrónica debe retornar una **Promesa** (Promise), lo que implica la capacidad de proporcionar un valor pospuesto que Nest podrá resolver de forma automática. A continuación, se ilustra un ejemplo de este concepto:

cats.controller.ts

```
@Get()
async findAll(): Promise<any[]> {
  return [];
}
```

El código anterior es completamente válido. Además, los controladores de rutas de Nest son aún más potentes, ya que pueden devolver flujos observables de RxJS. Nest se suscribirá automáticamente a la fuente subyacente y tomará el último valor emitido (una vez que se complete el flujo).

cats.controller.ts

```
@Get()
findAll(): Observable<any[]> {
  return of([]);
}
```

Ambas metodologías previas son efectivas y se puede optar por la que mejor se ajuste a los requisitos del desarrollador.

*12) Solicitud de Cargas Utiles*

El controlador de ruta POST en el ejemplo previo no recibía ningún parámetro del cliente. Para resolver esta situación se incorpora el decorador **@Body()** en este contexto.

Sin embargo, en primer lugar, si se está trabajando con TypeScript, es necesario definir el esquema del objeto o DTO (Objeto de Transferencia de Datos). Un DTO es un objeto que describe cómo los datos se enviarán a través de la red. Esto se puede lograr mediante interfaces de TypeScript o clases simples. Se recomienda utilizar clases en este contexto. ¿Cuál es la razón detrás de esta elección? Las clases forman parte del estándar JavaScript ES6 y, por lo tanto, se mantienen como entidades reales en el JavaScript compilado. Por otro lado, las interfaces de TypeScript se eliminan durante el proceso de transpilación, lo que impide que Nest pueda hacer referencia a ellas en el tiempo de ejecución. Este detalle cobra relevancia, ya que características como los Pipes ofrecen posibilidades adicionales cuando tienen acceso al metatipo de la variable en tiempo de ejecución.

La clase **CreateCatDto** se define de la siguiente manera:

create-cat.dto.ts

```
export class CreateCatDto {
  name: string;
  age: number;
  breed: string;
}
```

Esta clase cuenta únicamente con tres propiedades fundamentales. A partir de este punto, se puede emplear el DTO en el **CatsController**.

cats.controller.ts

```
@Post()
async create(@Body() createCatDto: CreateCatDto) {
  return 'This action adds a new cat';
}
```

CONSEJO

El **ValidationPipe** filtra propiedades no autorizadas para su recepción por el controlador, al mantener una lista blanca de propiedades permitidas. Cualquier propiedad que no se encuentre en esta lista será excluida automáticamente del objeto resultante. Por ejemplo, en el caso de **CreateCatDto**, se especifican las propiedades **name**, **age** y **breed** en la lista blanca. Puede encontrar información adicional siga el siguiente enlace.

*13) Manejo de errores*

Hay un capítulo aparte sobre cómo manejar errores (es decir, trabajar con excepciones) aquí.

*14) Muestra completa de un ejemplo*

A continuación, se presenta un ejemplo que emplea diversos decoradores disponibles para la creación de un controlador elemental. Este controlador ofrece un par de funciones que permiten acceder y modificar datos internos.

cats.controller.ts

```
import { Controller, Get, Query, Post, Body, Put, Param, Delete } from '@nestjs/common';
import { CreateCatDto, UpdateCatDto, ListAllEntities } from './dto';

@Controller('cats')
export class CatsController {
  @Post()
  create(@Body() createCatDto: CreateCatDto) {
    return 'This action adds a new cat';
  }

  @Get()
  findAll(@Query() query: ListAllEntities) {
    return `This action returns all cats (limit: ${query.limit} items)`;
  }

  @Get(':id')
  findOne(@Param('id') id: string) {
    return `This action returns a #${id} cat`;
  }

  @Put(':id')
  update(@Param('id') id: string, @Body() updateCatDto: UpdateCatDto) {
    return `This action updates a #${id} cat`;
  }

  @Delete(':id')
  remove(@Param('id') id: string) {
    return `This action removes a #${id} cat`;
  }
}
```

CONSEJO

El CLI de Nest proporciona un generador, que genera automáticamente todo el código inicial, esto es de gran ayuda ya que evita tener que hacer todo el código de manera manual, y simplifica la experiencia de desarrollo. Para obtener más información sobre esta característica siga este enlace.

*15) Puesta en marcha*

Una vez que el controlador previamente definido esté completamente configurado, Nest aún no tiene conocimiento de la existencia de **CatsController**. Como resultado, no generará una instancia de esta clase.

Los controladores siempre se asocian a un módulo, por lo tanto, se incluye el array de controladores dentro del decorador **@Module()**. Dado que aún no se ha especificado ningún otro módulo aparte del módulo raíz **AppModule**, se empleará para registrar **CatsController:**

app.module.ts

```
import { Module } from '@nestjs/common';
import { CatsController } from './cats/cats.controller';

@Module({
  controllers: [CatsController],
})
export class AppModule {}
```

Los metadatos se han vinculado a la clase del módulo mediante el uso del decorador **@Module()**, permitiendo a Nest identificar con facilidad qué controladores deben ser incorporados.

*16) Enfoque para bibliotecas específicas*

Hasta este punto, se ha abordado la manera convencional en que Nest maneja las respuestas. La segunda alternativa para manipular la respuesta consiste en utilizar un objeto de respuesta particular de la biblioteca. Para inyectar dicho objeto de respuesta, se requiere emplear el decorador **@Res()**. Para ilustrar las distinciones, se procede a modificar el **CatsController** de la siguiente manera:

```
import { Controller, Get, Post, Res, HttpStatus } from '@nestjs/common';
import { Response } from 'express';

@Controller('cats')
export class CatsController {
  @Post()
  create(@Res() res: Response) {
    res.status(HttpStatus.CREATED).send();
  }

  @Get()
  findAll(@Res() res: Response) {
     res.status(HttpStatus.OK).json([]);
  }
}
```

Aunque este enfoque resulta funcional y, de hecho, brinda mayor flexibilidad en ciertos aspectos al otorgar un control total sobre el objeto de respuesta, como la manipulación de encabezados y características específicas de la biblioteca, es importante usarlo con precaución. En general, este enfoque carece de la claridad que ofrece el método estándar de Nest y presenta algunas desventajas significativas. La principal desventaja radica en que el código se vuelve dependiente de la plataforma, ya que las bibliotecas subyacentes pueden proporcionar APIs diferentes en el objeto de respuesta, lo que también dificulta la realización de pruebas, ya que se debe simular el objeto de respuesta, entre otros.

Además, en el ejemplo anterior, se pierde la compatibilidad con las características de Nest que se basan en el manejo convencional de respuestas de Nest, como los Interceptors y los decoradores **@HttpCode()** / **@Header()**. Para resolver este problema, se puede habilitar la opción **passthrough** estableciéndola en **true**, de la siguiente manera:

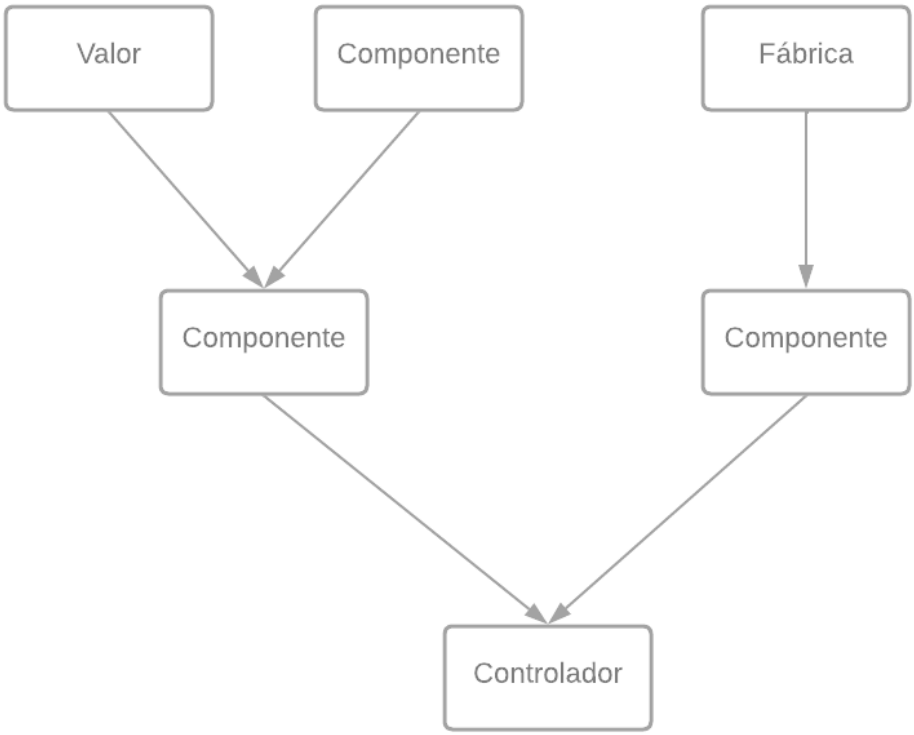
```
@Get()
findAll(@Res({ passthrough: true }) res: Response) {
  res.status(HttpStatus.OK);
  return [];
}
```

Ahora es posible interactuar con el objeto de respuesta nativo, como la configuración de cookies o encabezados en función de condiciones específicas, mientras se delega el resto de la funcionalidad al marco de trabajo.

*D. Proveedores [5]*

Los proveedores representan un concepto fundamental en Nest. Varias de las clases básicas de Nest pueden desempeñar el papel de proveedor, como servicios, repositorios, fábricas, ayudantes, entre otros. La idea principal de un proveedor es que puede ser inyectado como una dependencia. Esto implica que los objetos pueden establecer diversas relaciones entre sí, y la responsabilidad de "conectar" instancias de objetos puede ser en gran medida delegada al sistema en tiempo de ejecución de Nest.

*Imagen 3 Diagrama de funcionamiento de un proveedor*




*Fuente: Autor*

En el capítulo anterior, se construyó un **CatsController** simple. Los controladores deben encargarse de las solicitudes HTTP y delegar tareas más complejas a los proveedores. Los proveedores son clases simples de JavaScript que se declaran en un módulo.

CONSEJO
Dado que Nest permite la posibilidad de diseñar y organizar dependencias de una manera más orientada a objetos, se recomienda encarecidamente seguir los principios SOLID.

*1) Servicios*

Se puede empezar creando un **CatsService** básico. Dicho servicio tendrá la responsabilidad de gestionar el almacenamiento y la recuperación de datos, y su propósito principal es ser empleado por el **CatsController**, lo que lo convierte en una elección adecuada para ser establecido como un proveedor.

cats.service.ts

```
import { Injectable } from '@nestjs/common';
import { Cat } from './interfaces/cat.interface';

@Injectable()
export class CatsService {
  private readonly cats: Cat[] = [];

  create(cat: Cat) {
    this.cats.push(cat);
  }

  findAll(): Cat[] {
    return this.cats;
  }
}
```

CONSEJO
Para crear un servicio utilizando la CLI, simplemente ejecuta el comando: **nest g service cats**

El **CatsService** es una clase elemental que consta de una propiedad y dos funciones. La novedad reside en el uso del decorador **@Injectable()**. Esta anotación especifica que **CatsService** es una clase que el contenedor IoC de Nest puede administrar. Cabe destacar que este ejemplo también incorpora una interfaz llamada **Cat**, la cual probablemente tiene la siguiente estructura:

interfaces/cat.interface.ts

```
export interface Cat {
  name: string;
  age: number;
  breed: string;
}
```

Una vez que se dispone de un servicio para la recuperación de **cats**, es apropiado emplearla en el **CatsController**:

cats.controller.ts

```
import { Controller, Get, Post, Body } from '@nestjs/common';
import { CreateCatDto } from './dto/create-cat.dto';
import { CatsService } from './cats.service';
import { Cat } from './interfaces/cat.interface';

@Controller('cats')
export class CatsController {
  constructor(private catsService: CatsService) {}

  @Post()
  async create(@Body() createCatDto: CreateCatDto) {
    this.catsService.create(createCatDto);
  }

  @Get()
  async findAll(): Promise<Cat[]> {
    return this.catsService.findAll();
  }
}
```

El CatsService es inyectado mediante el constructor de la clase, y se puede notar la utilización de la sintaxis **private**. Esta notación permite la declaración e inicialización simultánea de la propiedad **catsService** en el mismo contexto.

*2) Inyección de dependencias*

Nest se fundamenta en el patrón de diseño SOLID de Inyección de Dependencias. Se recomienda leer el artículo sobre este concepto en la documentación oficial de Angular.

En el contexto de Nest, aprovechando las capacidades de TypeScript, la gestión de las dependencias se vuelve sumamente sencilla, ya que se resuelven según el tipo. En el ejemplo que se muestra a continuación, Nest se encargará de resolver **catsService**, creando y entregando una instancia de **CatsService** (o, en el caso normal de un singleton, proporcionando la instancia existente si ya ha sido solicitada en otro lugar). Luego, esta dependencia se resuelve y pasa al constructor del controlador (o se asigna a la propiedad correspondiente):

```
constructor(private catsService: CatsService) {}
```

*3) Scope – Ámbito*

Normalmente, los proveedores tienen un ciclo de vida ("ámbito") sincronizado con el ciclo de vida de la aplicación. Cuando la aplicación se inicia, es necesario resolver todas las dependencias y, por lo tanto, se debe instanciar cada proveedor. De manera similar, cuando la aplicación se apaga, cada proveedor será destruido. Sin embargo, existen formas de hacer que el ciclo de vida de su proveedor sea también de ámbito por solicitud. Puede obtener más información sobre estas técnicas siga este enlace.

*4) Proveedores personalizados*

Nest tiene un contenedor de inversión de control ("IoC") incorporado que resuelve las relaciones entre proveedores. Esta característica subyace de la inyección de dependencias descrita anteriormente, pero es, de hecho, mucho más poderosa de lo que se ha descrito hasta ahora. Hay varias formas de definir un proveedor: se puede utilizar valores simples, clases y fábricas tanto asíncronas como síncronas. Se proporcionan más ejemplos aquí.

*5) Proveedores opcionales*

En algunas situaciones, puede ocurrir que existan dependencias que no necesariamente deben resolverse de manera obligatoria. Por ejemplo, una clase podría depender de un objeto de configuración, pero en caso de que no se proporcione dicho objeto, deberían utilizarse los valores predeterminados. En esta circunstancia, la dependencia se vuelve voluntaria, ya que la ausencia del proveedor de configuración no provocaría errores.

Para señalar que un proveedor es opcional, se utiliza el decorador **@Optional()** en la declaración del constructor.

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

@Injectable()
export class HttpService<T> {
  constructor(@Optional() @Inject('HTTP_OPTIONS') private httpClient: T) {}
}
```

Es importante notar que en el caso anterior se está empleando un proveedor personalizado, y es por esta razón que se ha incorporado el token personalizado **HTTP_OPTIONS**. Los ejemplos previos han ilustrado la inyección de dependencias basada en el constructor, en la que se establece una dependencia a través de una clase en el constructor. Para obtener información adicional sobre proveedores personalizados y los tokens relacionados, se recomienda consultar esta fuente.

*6) Inyección basada en la propiedad*

La técnica que se ha empleado hasta este punto se conoce como inyección basada en el constructor, dado que los proveedores se introducen mediante el constructor. No obstante, en situaciones muy concretas, la inyección basada en propiedades puede ser beneficiosa. Por ejemplo, cuando una clase de alto nivel depende de uno o varios proveedores, pasarlos de forma repetitiva hacia arriba a través de llamadas a **super()** en las subclases desde el constructor puede resultar bastante laborioso. Para evitar esta complejidad, es posible utilizar el decorador **@Inject()** a nivel de propiedad.

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

@Injectable()
```

```
export class HttpService<T> {
  @Inject('HTTP_OPTIONS')
  private readonly httpClient: T;
}
```

ADVERTENCIA
En caso de que una clase no tenga una herencia de otra clase, se recomienda dar preferencia a la inyección basada en el constructor.

*7) Registro de proveedores*

Una vez que se ha establecido un proveedor (**CatsService**) y se ha creado un componente que consume ese servicio (**CatsController**), es necesario registrar el servicio en Nest para habilitar la inyección. Este proceso se lleva a cabo modificando el archivo de módulo (**app.module.ts)** y añadiendo el servicio al conjunto de proveedores dentro del decorador **@Module()**.

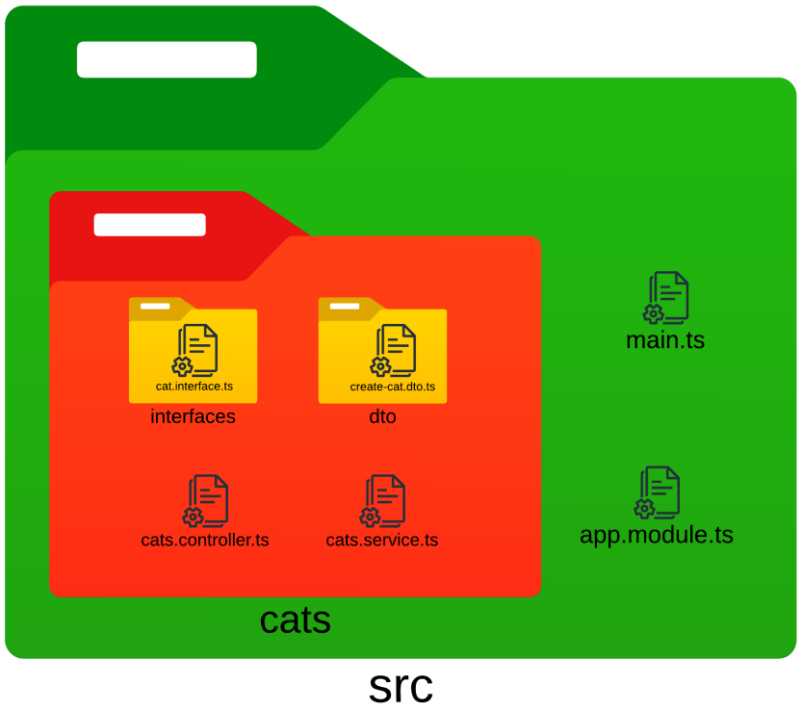
app.module.ts

```
import { Module } from '@nestjs/common';
import { CatsController } from './cats/cats.controller';
import { CatsService } from './cats/cats.service';

@Module({
  controllers: [CatsController],
  providers: [CatsService],
})
export class AppModule {}
```

Nest ahora podrá resolver las dependencias de la clase **CatsController**.

Así es como debería lucir ahora la estructura de directorios:

*Imagen 4 Estructura de archivos y carpetas del proyecto Cats*




*Fuente: Autor*

*8) Uso Manual*

Hasta el momento, se ha explorado cómo Nest se encarga automáticamente de la mayoría de los aspectos relacionados con la resolución de dependencias. Sin embargo, en situaciones específicas, es posible que surja la necesidad de salir del sistema integrado de Inyección de Dependencias para recuperar o crear proveedores de forma manual. A continuación, se presentan breves acotaciones sobre los temas mencionados.

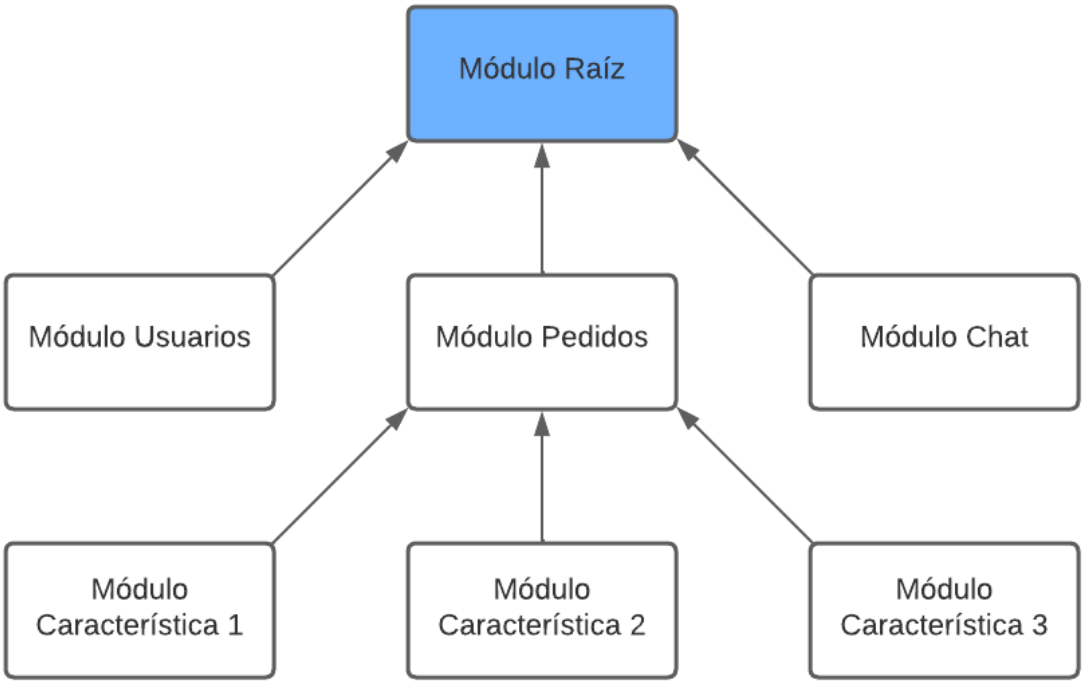
Para obtener instancias existentes o crear proveedores de manera dinámica, es posible emplear la referencia al módulo.

Si se desea obtener proveedores dentro de la función **bootstrap()** (por ejemplo, en aplicaciones independientes que no cuenten con controladores o cuando sea necesario utilizar un servicio de configuración durante el inicio), se recomienda consultar la sección sobre Aplicaciones independientes.

*E. Módulos [6]*

Un módulo es una clase anotada con un decorador **@Module()**. El decorador **@Module()** proporciona metadatos que Nest utiliza para organizar la estructura de la aplicación.

*Imagen 5 Diagrama de funcionamiento de los módulos*




*Fuente: Autor*

Cada aplicación cuenta con al menos un módulo, conocido como módulo raíz. Este módulo raíz es el punto de partida utilizado por Nest para construir el gráfico de la aplicación, una estructura interna de datos que se encarga de resolver las relaciones y dependencias entre módulos y proveedores. Aunque teóricamente las aplicaciones muy pequeñas podrían tener solo el módulo raíz, esto no es lo habitual. Es importante destacar que se recomienda fuertemente utilizar módulos como una manera efectiva de organizar los componentes. Por lo tanto, en la mayoría de las aplicaciones, la arquitectura resultante se compone de múltiples módulos, cada uno de ellos encapsulando un conjunto de capacidades estrechamente relacionadas.

El decorador **@Module()** toma un solo objeto cuyas propiedades describen el módulo:

*Tabla 5 Propiedades del decorador @Module*

| | |
|---|---|
| Providers | Los proveedores que serán instanciados por el inyector de Nest y que pueden ser compartidos al menos dentro de este módulo. |
| Controllers | El conjunto de controladores definidos en este módulo que deben ser instanciados. |
| Imports | La lista de módulos importados que exportan los proveedores que son necesarios en este módulo. |
| exports | El subconjunto de proveedores que son proporcionados por este módulo y deberían estar disponibles en otros módulos que los importen. Se puede usar tanto el proveedor en sí mismo como solo su token (valor proporcionado). |

*Fuente: Autor*

De manera predeterminada, los módulos encapsulan proveedores. Esto implica que no se puede inyectar proveedores que no formen parte directa del módulo actual ni que sean exportados desde los módulos importados. Por consiguiente, se puede considerar que los proveedores exportados desde un módulo representan la interfaz pública o API del módulo.

*1) Módulo de Características*

El **CatsController** y **CatsService** pertenecen al mismo dominio de la aplicación. Dado que están estrechamente relacionados, tiene sentido trasladarlos a un módulo de características. Un módulo de características simplemente organiza el código relevante para una característica específica, manteniendo el código organizado y estableciendo límites claros. Esto ayuda a gestionar la complejidad y desarrollar con principios SOLID, especialmente a medida que el tamaño de la aplicación y/o del equipo de desarrollo crece.

Para demostrar esto, se crea el **CatsModule**.

cats/cats.module.ts

```
import { Module } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Module({
  controllers: [CatsController],
  providers: [CatsService],
})
export class CatsModule {}
```

CONSEJO
Para crear un módulo utilizando la CLI, simplemente ejecute el comando: **nest g module cats**

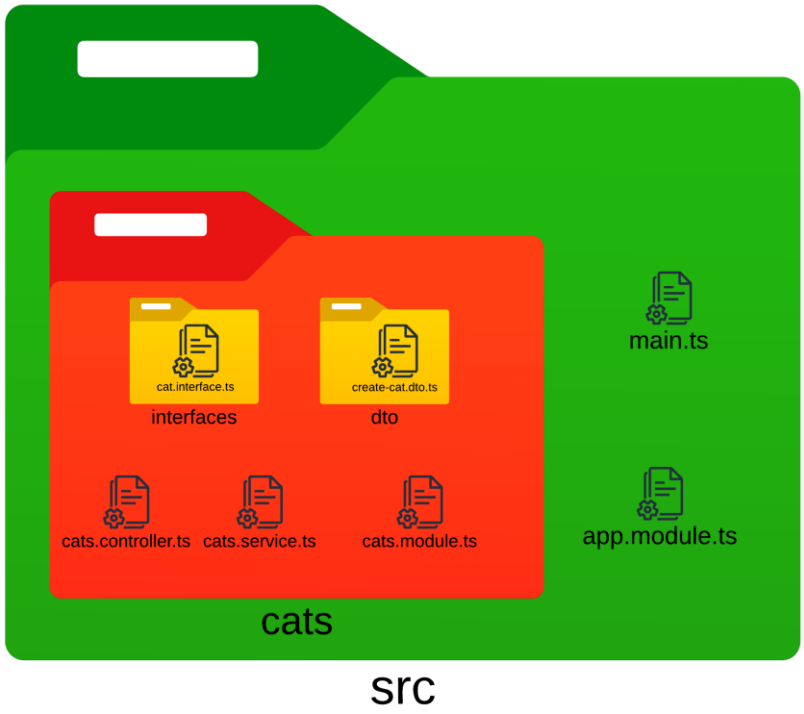
En el párrafo anterior, se describió la creación del **CatsModule** en el archivo **cats.module.ts** y la reubicación de todos los elementos relacionados con dicho módulo en la carpeta **cats**. El paso final consiste en la importación de este módulo en el módulo principal (**AppModule**), el cual se encuentra definido en el archivo **app.module.ts**.

app.module.ts

```
import { Module } from '@nestjs/common';
import { CatsModule } from './cats/cats.module';

@Module({
  imports: [CatsModule],
})
export class AppModule {}
```

Así es como se ve ahora la estructura de archivos:

*Imagen 6 Estructura de carpetas y archivos del proyecto Cats*

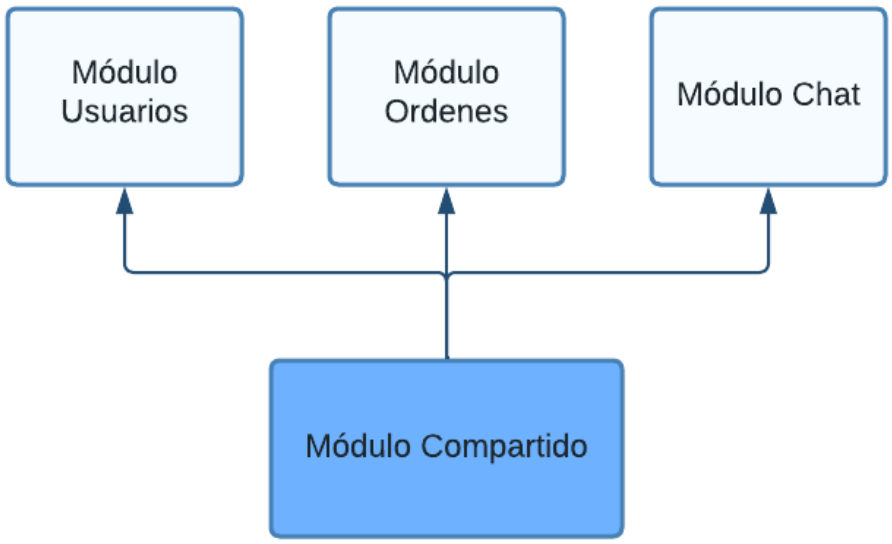



*Fuente: Autor*

*2) Módulos compartidos*

En Nest, los módulos son singleton de forma predeterminada, por lo tanto, se puede compartir la misma instancia de cualquier proveedor entre múltiples módulos sin esfuerzo.

*Imagen 7 Diagrama de funcionamiento de los módulos compartidos*




*Fuente: Autor*

Cada módulo se convierte automáticamente en un módulo compartido y, una vez creado, se vuelve apto para su reutilización por parte de otros módulos. Suponga que se desea compartir una instancia del proveedor **CatsService** entre múltiples módulos adicionales. Para lograrlo, el primer paso consiste en exportar el proveedor **CatsService**, incorporándolo en el conjunto de elementos exportados del módulo, como se ilustra a continuación:

cats.module.ts

```
import { Module } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Module({
  controllers: [CatsController],
  providers: [CatsService],
  exports: [CatsService]
})
export class CatsModule {}
```

Ahora, cualquier módulo que importe el **CatsModule** tendrá acceso al **CatsService** y compartirá la misma instancia con los demás módulos que también lo importen.

*3) Reexportación de módulos*

Como se vio anteriormente, los módulos pueden exportar sus proveedores internos. Además, pueden volver a exportar módulos que importen. En el siguiente ejemplo, el **CommonModule** es importado en el **CoreModule** y también exportado desde él, volviéndolo disponible para otros módulos que lo importen.

```
@Module({
  imports: [CommonModule],
  exports: [CommonModule],
})
export class CoreModule {}
```

*4) Inyección de dependencias*

Una clase de un módulo también puede inyectar proveedores (por ejemplo, para propósitos de configuración):

cats.module.ts

```
import { Module } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Module({
  controllers: [CatsController],
  providers: [CatsService],
})
export class CatsModule {
  constructor(private catsService: CatsService) {}
}
```

Sin embargo, las clases de un módulo en sí no pueden ser inyectadas como proveedores debido a una dependencia circular.

*5) Módulos globales*

Cuando se requiere importar el mismo conjunto de módulos repetidamente en varios lugares, esta tarea puede resultar en una labor monótona. A diferencia de Nest, en Angular, los proveedores se registran en un ámbito global y, una vez definidos, se encuentran accesibles en todos los rincones de la aplicación. Sin embargo, en el caso de Nest, se encapsulan los proveedores dentro del alcance específico de un módulo, lo que significa que no es posible emplear los proveedores de un módulo en otro lugar sin previamente importar el módulo que los contiene.

Si se busca ofrecer un conjunto de proveedores que deban estar disponibles de manera predeterminada en todos los rincones de la aplicación (como asistentes, conexiones a bases de datos, entre otros), se puede lograr convirtiendo el módulo en global mediante el decorador **@Global()**.

```
import { Module, Global } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Global()
@Module({
  controllers: [CatsController],
  providers: [CatsService],
  exports: [CatsService],
})
export class CatsModule {}
```

El decorador **@Global()** hace que el módulo tenga un ámbito global. Los módulos globales deben registrarse solo una vez, generalmente en el módulo raíz o núcleo. En el ejemplo anterior, el proveedor **CatsService** estará disponible en todas partes, y los módulos que deseen inyectar el servicio no necesitarán importar el **CatsModule** en su arreglo de importaciones.

CONSEJO
Hacer todo global no es una buena decisión de diseño. Los módulos globales están disponibles para reducir la cantidad de código redundante. Por lo general, el arreglo de importaciones es la forma preferida de hacer que la API del módulo esté disponible para sus consumidores.

*6) Módulos dinámicos*

El sistema de módulos de Nest cuenta con una característica de gran utilidad conocida como módulos dinámicos. Esta funcionalidad permite a los usuarios crear módulos personalizados de manera sencilla, que pueden registrar y configurar proveedores de forma dinámica. Los módulos dinámicos se examinan en profundidad en este enlace. En esta sección, se proporciona un resumen breve con el objetivo de concluir la introducción a los módulos.

A continuación, se presenta un ejemplo de un módulo dinámico para un **DatabaseModule**:

```
import { Module, DynamicModule } from '@nestjs/common';
import { createDatabaseProviders } from './database.providers';
import { Connection } from './connection.provider';

@Module({
  providers: [Connection],
})
export class DatabaseModule {
  static forRoot(entities = [], options?): DynamicModule {
    const providers = createDatabaseProviders(options, entities);
    return {
      module: DatabaseModule,
      providers: providers,
      exports: providers,
    };
  }
}
```

CONSEJO
El método **forRoot()** puede devolver un módulo dinámico de manera síncrona o asíncrona (es decir, a través de una **Promesa**).

Este módulo establece el proveedor **Connection** como predeterminado, según se especifica en los metadatos del decorador **@Module()**. Además, en función de las entidades y opciones proporcionadas mediante el método

**forRoot()**, expone un conjunto de proveedores adicionales, como los repositorios. Es importante destacar que las propiedades que provienen de este módulo dinámico se integran con los metadatos base del módulo definidos en el decorador **@Module()**, extendiéndolos en lugar de reemplazarlos. De esta manera, tanto el proveedor **Connection** declarado estáticamente como los proveedores de repositorios generados dinámicamente son exportados por el módulo.

Si se desea registrar un módulo dinámico en el ámbito global, simplemente se debe configurar la propiedad **global** en **true**.

```
{
  global: true,
  module: DatabaseModule,
  providers: providers,
  exports: providers,
}
```

ADVERTENCIA
Como se mencionó anteriormente, hacer todo global no es una buena decisión de diseño.

El **DatabaseModule** se puede importar y configurar de la siguiente manera:

```
import { Module } from '@nestjs/common';
import { DatabaseModule } from './database/database.module';
import { User } from './users/entities/user.entity';

@Module({
  imports: [DatabaseModule.forRoot([User])],
})
export class AppModule {}
```

En caso de que se quiera reexportar un módulo dinámico, se tiene la opción de no incluir la llamada al método **forRoot()** dentro de la lista de elementos a exportar.

```
import { Module } from '@nestjs/common';
import { DatabaseModule } from './database/database.module';
import { User } from './users/entities/user.entity';

@Module({
  imports: [DatabaseModule.forRoot([User])],
  exports: [DatabaseModule],
})
export class AppModule {}
```

El capítulo de módulos dinámicos aborda este tema con mayor detalle e incluye un ejemplo funcional.

*F. Middleware [7]*

El middleware es una función que se ejecuta antes de que se active el controlador de una ruta específica. Estas funciones de middleware tienen acceso a los objetos de solicitud y respuesta, así como a la función **next()**, que se utiliza en el ciclo de solicitud-respuesta de la aplicación. Por lo general, la siguiente función del middleware se representa mediante una variable llamada **next**.

*Imagen 8 Diagrama de funcionamiento de un Middleware*


*Fuente: Autor*

El middleware de Nest, por defecto, es equivalente al middleware de Express. La siguiente descripción de la documentación oficial de Express describe las capacidades del middleware:

Las funciones middleware pueden realizar las siguientes tareas:

- Ejecutar cualquier código.
- Realizar cambios en los objetos de solicitud y respuesta.
- Finalizar el ciclo de solicitud-respuesta.
- Llamar a la siguiente función de middleware en la pila.

Si la función middleware actual no finaliza el ciclo de solicitud-respuesta, debe llamar a **next()** para pasar el control a la siguiente función de middleware. De lo contrario, la solicitud quedará pendiente.

En Nest, es posible desarrollar middlewares personalizados tanto en forma de función o como una clase con el decorador **@Injectable()**. Para que una clase sea considerada middleware, debe implementar la interfaz **NestMiddleware**, mientras que, en el caso de una función, no existen requisitos específicos. Ahora, es momento de crear una característica de middleware básica utilizando la técnica basada en clases.

ADVERTENCIA
Express y Fastify manejan el middleware de manera diferente y proporcionan diferentes métodos. Lea más al respecto en este enlace.

logger.middleware.ts

```
import { Injectable, NestMiddleware } from '@nestjs/common';
import { Request, Response, NextFunction } from 'express';

@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  use(req: Request, res: Response, next: NextFunction) {
    console.log('Request...');
    next();
  }
}
```

*1) Inyección de dependencias*

El middleware de Nest admite por completo la Inyección de Dependencias. Al igual que con los proveedores y los controladores, pueden inyectar dependencias que están disponibles en el mismo módulo. Como es habitual, esto se hace a través del constructor.

*2) Utilizando middleware*

El decorador **@Module()** no es el lugar adecuado para el middleware; en su lugar, se configura utilizando el método **configure()** dentro de la clase del módulo. Para los módulos que desean incorporar middlewares, deben asegurarse de implementar la interfaz **NestModule**. A continuación, se configura el **LoggerMiddleware** en el **AppModule**.

app.module.ts

```
import { Module, NestModule, MiddlewareConsumer } from '@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { CatsModule } from './cats/cats.module';

@Module({
  imports: [CatsModule],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes('cats');
  }
}
```

En la instancia anterior, se ha establecido la configuración del **LoggerMiddleware** para los controladores de la ruta **/cats** que previamente se habían definido en el **CatsController**. Además, es factible limitar la aplicación del middleware a un método de solicitud en particular, al proporcionar un objeto que incluye tanto la ruta como el método de solicitud requeridos al utilizar el método **forRoutes()** durante la configuración del middleware. En el siguiente ejemplo, es relevante notar que se importa el enum **RequestMethod** para hacer referencia al tipo de método de solicitud deseado.

app.module.ts

```
import { Module, NestModule, RequestMethod, MiddlewareConsumer } from '@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { CatsModule } from './cats/cats.module';

@Module({
  imports: [CatsModule],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes({ path: 'cats', method: RequestMethod.GET });
  }
}
```

CONSEJO
El método **configure()** se puede hacer asincrónico utilizando **async/await** (por ejemplo, al esperar la finalización de una operación asincrónica dentro del cuerpo del método **configure()**).

ADVERTENCIA
Cuando se utiliza el adaptador de Express en Nest, la aplicación de Nest registrará por defecto **json** y **urlencoded** del paquete **body-parser**. Esto significa que, si se desea personalizar ese middleware a través de **MiddlewareConsumer**, se debe desactivar el middleware global configurando la bandera **bodyParser** en **false** al crear la aplicación con **NestFactory.create()**.

*3) Comodines de ruta*

También se admiten rutas basadas en patrones. Por ejemplo, el asterisco se utiliza como comodín y coincidirá con cualquier combinación de caracteres:

```
forRoutes({ path: 'ab*cd', method: RequestMethod.ALL });
```

La ruta '**ab*cd**' coincidirá con **abcd, ab_cd, abecd** y así sucesivamente. Los caracteres: ?, +, *, y **()** se pueden usar en una ruta, y son subconjuntos de sus contrapartes en expresiones regulares. El guion (-) y el punto (.) se interpretan literalmente en las rutas basadas en cadenas.

ADVERTENCIA
La biblioteca fastify utiliza la última versión del paquete **path-to-regexp**, que ya no admite asteriscos como comodines *. En su lugar, se debe utilizar parámetros (por ejemplo, (.*), :splat*).

*4) Consumir middlewares*

El **MiddlewareConsumer**, es una clase de apoyo, ofrece una serie de métodos integrados destinados a facilitar la administración del middleware. Todos estos métodos se pueden encadenar, para tener una estructura de código más legible. El método **forRoutes()** es versátil, ya que puede aceptar una sola cadena, múltiples cadenas, un objeto **RouteInfo**, una clase del controlador e incluso varias clases del controlador. En la mayoría de los casos, es probable que simplemente se pase una lista de los controladores separados por comas. A continuación, se presenta un ejemplo que utiliza un solo controlador:

app.module.ts

```
import { Module, NestModule, MiddlewareConsumer } from '@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { CatsModule } from './cats/cats.module';
import { CatsController } from './cats/cats.controller';

@Module({
  imports: [CatsModule],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes(CatsController);
  }
}
```

CONSEJO
El método **apply()** puede tomar uno o varios argumentos para especificar varios middlewares.

*5) Rutas excluidas*

En ciertas situaciones, puede resultar necesario excluir rutas del middleware. Esta tarea se puede llevar a cabo de manera sencilla utilizando el método **exclude()**. Dicho método tiene la capacidad de recibir una sola cadena, múltiples cadenas o un objeto **RouteInfo** que especifica las rutas que se deben excluir, tal como se ilustra a continuación:

```
consumer
  .apply(LoggerMiddleware)
  .exclude(
    { path: 'cats', method: RequestMethod.GET },
    { path: 'cats', method: RequestMethod.POST },
    'cats/(.*)',
  )
  .forRoutes(CatsController);
```

CONSEJO
El método **exclude()** admite comodines utilizando el paquete path-to-regexp.

Con el ejemplo anterior, el **LoggerMiddleware** se vinculará a todas las rutas definidas dentro del **CatsController**, excepto las tres que se pasaron al método **exclude()**.

*6) Middleware funcional*

La clase **LoggerMiddleware** que se ha estado utilizando es bastante sencilla. Carece de atributos, métodos adicionales o dependencias. ¿Por qué no simplemente definirla como una función simple en lugar de una clase? De hecho, se puede hacer. Este tipo de middleware se conoce como middleware funcional. Para transformar el middleware de registro de un enfoque basado en clases a un middleware funcional, se hace lo siguiente:

logger.middleware.ts

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

export function logger(req: Request, res: Response, next: NextFunction) {
  console.log(`Request...`);
  next();
};
```

Y utilizarlo dentro del AppModule:

app.module.ts

```
consumer
  .apply(logger)
  .forRoutes(CatsController);
```

CONSEJO
Se sugiere optar por la forma más simple de middleware funcional siempre que se prescinda de cualquier dependencia en el middleware.

*7) Múltiples middlewares*

Como se mencionó anteriormente, para unir múltiples middlewares que se ejecutan secuencialmente, simplemente hay que proporcionar una lista separada por comas dentro del método **apply()**.

```
consumer.apply(cors(), helmet(), logger).forRoutes(CatsController);
```

*8) Middleware global*

Si se desea asociar un middleware a todas las rutas registradas simultáneamente, se puede emplear el método **use()** que se ofrece a través de la instancia de **INestApplication**.

main.ts

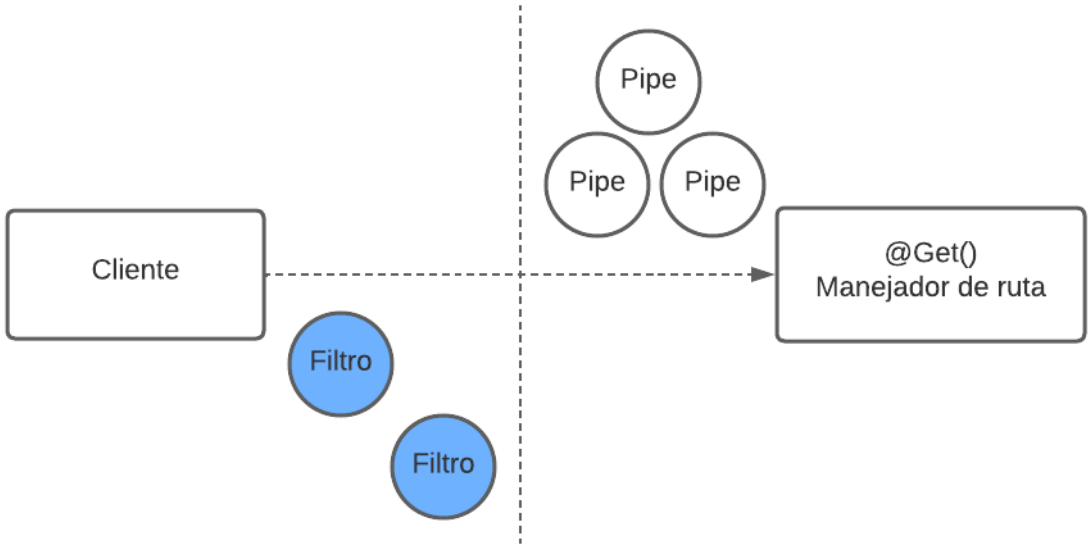
```
const app = await NestFactory.create(AppModule);
app.use(logger);
await app.listen(3000);
```

CONSEJO
No se encuentra habilitado el acceso al contenedor de inyección de dependencias en un middleware global. En su lugar, se puede emplear un middleware funcional a través de la función **app.use()**. Otra opción posible es utilizar un middleware basado en una clase y consumirlo mediante **.forRoutes('*')** dentro del AppModule (o cualquier otro módulo).

*G. Filtros de excepción [8]*

Nest posee una capa de excepciones integrada que asume la responsabilidad de procesar todas las excepciones no controladas dentro de una aplicación. Cuando el código de una aplicación no logra manejar una excepción, esta capa se encarga de capturarla y enviar de manera automática una respuesta apropiada y amigable al usuario.

*Imagen 9 Diagrama de funcionamiento de los filtros de excepción*




*Fuente: Autor*

De manera predeterminada, esta función es llevada a cabo por un filtro global de excepciones incorporado, que se encarga de manejar excepciones del tipo **HttpException** (y sus subclases). Cuando se encuentra una excepción no reconocida (que no sea ni **HttpException** ni una clase que herede de **HttpException**), el filtro de excepciones incorporado genera la siguiente respuesta JSON por defecto:

```
{
  "statusCode": 500,
  "message": "Internal server error"
}
```

CONSEJO
El filtro global de excepciones admite parcialmente la biblioteca **http-errors**. Básicamente, cualquier excepción lanzada que contenga las propiedades **statusCode** y **message** se llenará correctamente y se enviará como respuesta (en lugar de la **InternalServerErrorException** predeterminada para excepciones no reconocidas).

*1) Lanzar excepciones estándar*

Nest ofrece una clase integrada llamada **HttpException** que se encuentra disponible en el paquete **@nestjs/common**. En aplicaciones comunes que utilizan API-REST/GraphQL, se recomienda enviar objetos de respuesta HTTP estándar cuando se encuentran en ciertas situaciones de error.

Por ejemplo, en el **CatsController**, hay un método llamado **findAll()** que corresponde a una ruta **GET**. Si, por alguna razón, este controlador de ruta genera una excepción, se puede resolver de la siguiente manera:

cats.controller.ts

```
@Get()
async findAll() {
  throw new HttpException('Forbidden', HttpStatus.FORBIDDEN);
}
```

CONSEJO
Aquí se utiliza el **HttpStatus**. Este es un enum auxiliar importado del paquete **@nestjs/common**.

Cuando el cliente llama a este endpoint, la respuesta se ve de la siguiente manera:

```
{
  "statusCode": 403,
  "message": "Forbidden"
}
```

El constructor de **HttpException** toma dos argumentos requeridos que determinan la respuesta:

- El argumento **response** define el cuerpo de la respuesta en formato JSON. Puede ser una cadena de texto o un objeto.
- El argumento **status** define el código de estado HTTP.

Por defecto, el cuerpo de la respuesta en formato JSON contiene dos propiedades:

- **statusCode**: se establece por defecto en el código de estado HTTP proporcionado en el argumento de status.
- **message**: una breve descripción del error HTTP basada en el código de estado.

Para modificar solo la porción del mensaje que se encuentra en el cuerpo de la respuesta en formato JSON, se debe suministrar una cadena en el parámetro de respuesta. Para reemplazar por completo el contenido del cuerpo de la respuesta en formato JSON, se debe proporcionar un objeto en el parámetro de respuesta. Nest procederá a serializar el objeto y lo enviará como el nuevo contenido del cuerpo de la respuesta en formato JSON.

El constructor tiene un segundo argumento denominado **status,** el cual requiere un código de estado HTTP válido. La recomendación principal es utilizar el enum **HttpStatus** que se importa desde el paquete **@nestjs/common**.

Además, existe un tercer argumento en el constructor (opcional) denominado **options**, que puede utilizarse para proporcionar una causa de error. Este objeto de causa no se incluye en la serialización del objeto de respuesta, pero puede resultar útil para fines de registro al ofrecer información relevante sobre el error interno que desencadenó la **HttpException**.

A continuación, se presenta un ejemplo en el que se reemplaza la totalidad del contenido de la respuesta y se proporciona una causa de error como ilustración:

cats.controller.ts

```
@Get()
async findAll() {
  try {
    await this.service.findAll()
  } catch (error) {
    throw new HttpException({
      status: HttpStatus.FORBIDDEN,
      error: 'This is a custom message',
    }, HttpStatus.FORBIDDEN, {
      cause: error
    });
  }
}
```

Utilizando lo anterior, así es como se vería la respuesta:

```
{
  "status": 403,
  "error": "This is a custom message"
}
```

*2) Excepciones personalizadas*

En la mayoría de las situaciones, no será necesario desarrollar excepciones personalizadas, y se puede recurrir a las excepciones HTTP incorporadas de Nest, como se explica en la sección posterior. Sin embargo, si se requiere crear excepciones personalizadas, se aconseja seguir una buena práctica que implica establecer una jerarquía de excepciones personalizadas, en la que las excepciones que se creen hereden de la clase base **HttpException**. Al adoptar este enfoque, Nest identificará las excepciones personalizadas y se encargará de forma automática de las respuestas de error. A continuación, se muestra cómo implementar una excepción personalizada de este tipo:

forbidden.exception.ts

```
export class ForbiddenException extends HttpException {
  constructor() {
    super('Forbidden', HttpStatus.FORBIDDEN);
  }
}
```

Dado que **ForbiddenException** es una extensión de la clase base **HttpException**, se integrará perfectamente con el manejador de excepciones predefinido. Por lo tanto, es posible emplearlo dentro del método **findAll()**.

cats.controller.ts

```
@Get()
async findAll() {
  throw new ForbiddenException();
}
```

*3) Excepciones HTTP integradas*

Nest proporciona un conjunto de excepciones estándar que heredan de la base HttpException. Estas están expuestas desde el paquete **@nestjs/common** y representan muchas de las excepciones HTTP más comunes:

- BadRequestException
- UnauthorizedException
- NotFoundException
- ForbiddenException
- NotAcceptableException
- RequestTimeoutException
- ConflictException
- GoneException
- HttpVersionNotSupportedException
- PayloadTooLargeException
- UnsupportedMediaTypeException
- UnprocessableEntityException
- InternalServerErrorException
- NotImplementedException
- ImATeapotException
- MethodNotAllowedException
- BadGatewayException
- ServiceUnavailableException
- GatewayTimeoutException
- PreconditionFailedException

Todas las excepciones incorporadas pueden proporcionar tanto una causa de error como una descripción de error utilizando el parámetro options:

```
throw new BadRequestException('Something bad happened', { cause: new Error(), description:
'Some error description' })
```

Usando lo anterior, así es como se vería la respuesta:

```
{
  "message": "Something bad happened",
  "error": "Some error description",
  "statusCode": 400,
}
```

*4) Filtros de excepción*

Aunque el filtro de excepciones base (incorporado) puede ocuparse automáticamente de varias situaciones, algunos usuarios pueden preferir tener un control total sobre la capa de excepciones. Por ejemplo, en ciertas circunstancias, podría ser necesario añadir registros o emplear un esquema JSON diferente basado en factores dinámicos. Los filtros de excepciones se han creado con este objetivo en mente. Permiten a los usuarios tener un control preciso sobre el flujo de excepciones y el contenido de la respuesta que se envía al cliente.

Se propone crear un filtro de excepciones que tendrá la responsabilidad de detectar excepciones que pertenezcan a la clase **HttpException** y aplicar una lógica de respuesta personalizada a dichas excepciones. Para llevar a cabo esta tarea, se requerirá el acceso a los objetos subyacentes de la solicitud (**Request**) y de la respuesta (**Response**) de la plataforma subyacente. Se accederá al objeto **Request** con el fin de extraer la URL original y registrar esta información. Además, se utilizará el objeto **Response** para tomar el control directo de la respuesta que se envía, empleando el método **response.json()**.

http-exception.filter.ts

```
import { ExceptionFilter, Catch, ArgumentsHost, HttpException } from '@nestjs/common';
import { Request, Response } from 'express';

@Catch(HttpException)
export class HttpExceptionFilter implements ExceptionFilter {
  catch(exception: HttpException, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();
    const request = ctx.getRequest<Request>();
    const status = exception.getStatus();

    response
      .status(status)
      .json({
        statusCode: status,
        timestamp: new Date().toISOString(),
        path: request.url,
      });
  }
}
```

CONSEJO

Todos los filtros de excepciones deben implementar la interfaz genérica **ExceptionFilter<T>**. Esto obliga al desarrollador a proporcionar el método **catch(exception: T, host: ArgumentsHost)**. En donde T indica el tipo de excepción.

ADVERTENCIA

Si se esta utilizando **@nestjs/platform-fastify**, se puede usar **response.send()** en lugar de **response.json()**. Se debe importar los tipos correctos desde **fastify**.

El decorador **@Catch(HttpException)** agrega los metadatos requeridos al filtro de excepciones, especificando a Nest que este filtro específico se encarga de detectar excepciones del tipo **HttpException** y nada más. El decorador **@Catch()** puede recibir un solo parámetro o una lista de parámetros separados por comas. Esto posibilita la configuración del filtro para gestionar varios tipos de excepciones al mismo tiempo.

*5) Argumentos host*

Analizando los parámetros del método **catch()**, se encuentra que el primer parámetro, denominado **exception**, representa el objeto de excepción que está siendo procesado en ese momento. Por otro lado, el segundo parámetro, **host**, es una instancia de **ArgumentsHost**. **ArgumentsHost** es un recurso de utilidad sumamente potente, el cual se explorará con mayor profundidad en la sección referente al contexto de ejecución*. En el fragmento de código proporcionado en la sección anterior, se utiliza **ArgumentsHost** para obtener una referencia a los objetos **Request** y **Response** que fueron originalmente transmitidos al controlador de solicitud inicial (es decir, al controlador donde se originó la excepción). En esta muestra de código, se hacen uso de algunos métodos auxiliares disponibles en **ArgumentsHost** para obtener los objetos **Request** y **Response** deseados. Para obtener información adicional acerca de **ArgumentsHost**, siga el siguiente enlace.

La justificación de este grado de abstracción radica en que **ArgumentsHost** opera en múltiples contextos, incluyendo el contexto del servidor HTTP en el que se encuentra actualmente en uso, así como en Microservicios y WebSockets. En el capítulo relacionado con el contexto de ejecución, se abordará la forma en que podemos acceder a los argumentos subyacentes apropiados para cualquier contexto de ejecución, aprovechando la versatilidad de ArgumentsHost y sus funciones auxiliares. Esto habilita la posibilidad de desarrollar filtros de excepción de naturaleza genérica que sean efectivos en todos los escenarios.

*6) Filtros obligatorios*

Se procede a asociar el recién creado filtro **HttpExceptionFilter** con la función **create()** del controlador **CatsController**.

cats.controller.ts

```
@Post()
@UseFilters(new HttpExceptionFilter())
async create(@Body() createCatDto: CreateCatDto) {
  throw new ForbiddenException();
}
```

CONSEJO
El decorador **@UseFilters()** se importa desde el paquete **@nestjs/common**.

En este contexto, se empleó el decorador **@UseFilters()**. Similar al decorador **@Catch()**, este puede recibir uno o una lista de filtros separados por comas. En este caso específico, se ha creado una instancia de **HttpExceptionFilter**. Como alternativa, es posible proporcionar la clase (en lugar de una instancia), lo cual delega la responsabilidad de la creación al framework y permite la inyección de dependencias.

cats.controller.ts

```
@Post()
@UseFilters(HttpExceptionFilter)
async create(@Body() createCatDto: CreateCatDto) {
  throw new ForbiddenException();
}
```

CONSEJO
Se recomienda, en la medida de lo posible, emplear clases para aplicar filtros en lugar de instancias individuales. Esta práctica contribuye a la eficiencia en el uso de la memoria, ya que Nest puede reutilizar sin dificultad instancias de la misma clase en todo el módulo.

En el caso anterior, el filtro de excepciones **HttpExceptionFilter** se aplica exclusivamente al manejador de la ruta **create()**, limitando su efecto al ámbito del método. Los filtros de excepción pueden tener diversos ámbitos, tales como ámbito al método del controlador, al controlador en su totalidad o a nivel global.

Para ilustrar, si se deseara configurar un filtro con ámbito al controlador, se seguiría el siguiente procedimiento:

cats.controller.ts

```
@UseFilters(new HttpExceptionFilter())
export class CatsController {}
```

Esta configuración establece el filtro de excepciones **HttpExceptionFilter** para cada controlador de rutas específico que esté definido dentro del **CatsController**.

Para implementar un filtro con ámbito global, se seguiría el procedimiento a continuación:

main.ts

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalFilters(new HttpExceptionFilter());
  await app.listen(3000);
}
bootstrap();
```

ADVERTENCIA
El método **useGlobalFilters()** no configura filtros para puertas de enlace o aplicaciones híbridas.

Los filtros de ámbito global son aplicables a toda la aplicación, afectando a cada controlador y a todos los manejadores de rutas por igual. En lo que respecta a la inyección de dependencias, los filtros globales que se registran desde fuera de cualquier módulo (como se muestra en el ejemplo anterior con **useGlobalFilters()**) no tienen la capacidad de inyectar dependencias, dado que esta operación se realiza fuera del contexto de cualquier módulo.

Para resolver esta limitación, se puede registrar un filtro de ámbito global directamente desde cualquier módulo utilizando la siguiente estructura:

app.module.ts

```
import { Module } from '@nestjs/common';
import { APP_FILTER } from '@nestjs/core';

@Module({
  providers: [
    {
      provide: APP_FILTER,
      useClass: HttpExceptionFilter,
    },
  ],
})
export class AppModule {}
```

CONSEJO
Al emplear este método para llevar a cabo la inyección de dependencias en el filtro, es importante tener en cuenta que, sin importar el módulo en el que se aplique este enfoque, el filtro en sí es global en su ámbito. La ubicación adecuada para realizar esta acción es en el módulo donde se define el filtro (por ejemplo, el **HttpExceptionFilter** en el caso previo). Es importante destacar que **useClass** no constituye la única opción para gestionar el registro personalizado de proveedores. Puede obtener información adicional al respecto en este enlace.

Se puede incorporar cuantos filtros se requieran mediante este método; basta con incluirlos en el conjunto de proveedores.

*7) Capturarlo todo*

Si se desean capturar todas las excepciones no gestionadas, sin importar su tipo, se debe dejar la lista de parámetros del decorador **@Catch()** en blanco, de la siguiente manera: **@Catch()**.

En el ejemplo que se presenta a continuación, el código es independiente de la plataforma, ya que se vale del adaptador HTTP para enviar la respuesta y no interactúa directamente con ninguno de los objetos específicos de la plataforma, como **Request** y **Response**:

```
import {
  ExceptionFilter,
  Catch,
  ArgumentsHost,
  HttpException,
  HttpStatus,
} from '@nestjs/common';
import { HttpAdapterHost } from '@nestjs/core';

@Catch()
export class AllExceptionsFilter implements ExceptionFilter {
  constructor(private readonly httpAdapterHost: HttpAdapterHost) {}

  catch(exception: unknown, host: ArgumentsHost): void {
    // In certain situations `httpAdapter` might not be available in the
    // constructor method, thus we should resolve it here.
    const { httpAdapter } = this.httpAdapterHost;

    const ctx = host.switchToHttp();

    const httpStatus =
      exception instanceof HttpException
        ? exception.getStatus()
        : HttpStatus.INTERNAL_SERVER_ERROR;

    const responseBody = {
      statusCode: httpStatus,
      timestamp: new Date().toISOString(),
      path: httpAdapter.getRequestUrl(ctx.getRequest()),
    };

    httpAdapter.reply(ctx.getResponse(), responseBody, httpStatus);
  }
}
```

ADVERTENCIA

Es importante tener en cuenta que al combinar un filtro que atrapa cualquier tipo de excepción, con un filtro que está asociado a un tipo específico, se debe asegurar que el filtro de "Captura todo" sea declarado primero. Esto es esencial para permitir que el filtro específico pueda manejar adecuadamente el tipo particular de excepción al que está vinculado.

*8) Herencia*

Por lo general, se desarrollarán filtros de excepción completamente adaptados para satisfacer las necesidades específicas de la aplicación. No obstante, en ocasiones, puede surgir situaciones en las que se desee simplemente ampliar el filtro global de excepciones predefinido y modificar su comportamiento en función de ciertos factores.

Para delegar el procesamiento de excepciones al filtro base, es necesario extender la clase **BaseExceptionFilter** y utilizar el método **catch()** heredado.

all-exceptions.filter.ts

```
import { Catch, ArgumentsHost } from '@nestjs/common';
import { BaseExceptionFilter } from '@nestjs/core';

@Catch()
export class AllExceptionsFilter extends BaseExceptionFilter {
  catch(exception: unknown, host: ArgumentsHost) {
    super.catch(exception, host);
  }
}
```

ADVERTENCIA
Los filtros de ámbito al método y de ámbito al controlador que se extienden de **BaseExceptionFilter** no deben instanciarse con **new**. En su lugar, permita que el marco de trabajo los instancie automáticamente.

La implementación anterior se presenta como una estructura que ejemplifica el enfoque. En su propia implementación del filtro de excepción extendido, se debe incorporar la lógica de negocio personalizada, como la gestión de múltiples condiciones.

Los filtros globales tienen la habilidad de ampliar el filtro base, y esto se puede realizar de dos maneras distintas.

El primer enfoque implica la inclusión de la referencia del **HttpAdapter** al instanciar el filtro global personalizado:

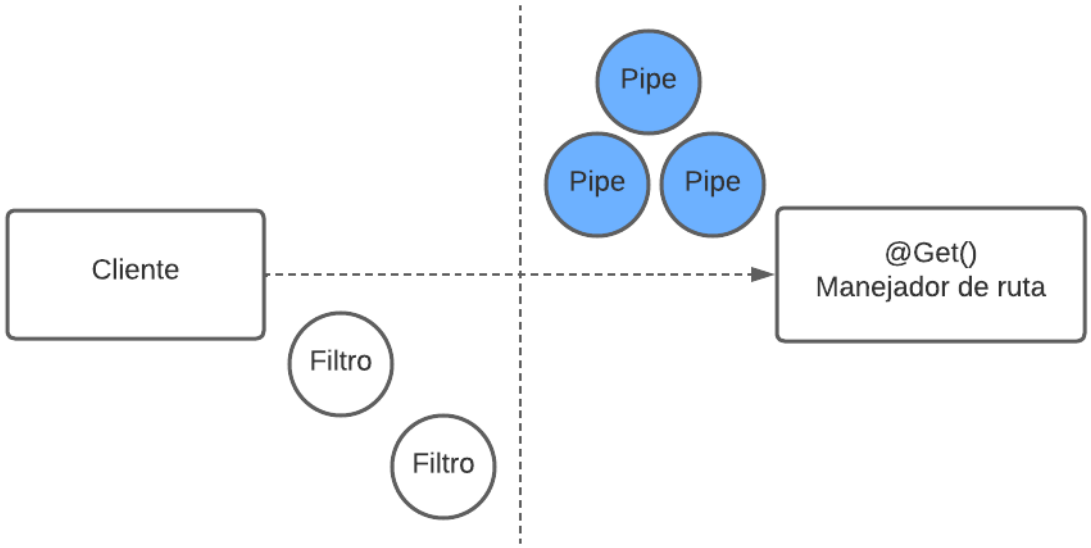
```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  const { httpAdapter } = app.get(HttpAdapterHost);
  app.useGlobalFilters(new AllExceptionsFilter(httpAdapter));

  await app.listen(3000);
}
bootstrap();
```

El segundo método es utilizar el token **APP_FILTER** como se muestra aquí.

*H. Pipes [9]*

Un pipe es una clase anotada con el decorador **@Injectable()**, que implementa la interfaz **PipeTransform**.

*Imagen 10 Diagrama de funcionamiento de los pipes*




*Fuente: Autor*

Los pipes tienen dos usos comunes:

1. Transformación: convertir los datos de entrada en el formato deseado (por ejemplo, de *string* a *number*).
2. Validación: evaluar los datos de entrada y, si son válidos, pasarlos sin cambios; de lo contrario, generar una excepción.

En ambos casos, los pipes operan en los argumentos que son procesados por un controlador. Nest inserta un pipe justo antes de llamar a un método, y el pipe recibe los argumentos destinados a dicho método para operar sobre ellos. Cualquier operación de transformación o validación ocurre en ese momento, y luego el controlador es invocado con los argumentos potencialmente transformados.

Nest incorpora diversos pipes predefinidos que se pueden emplear directamente, pero también es posible desarrollar pipes personalizados según las necesidades específicas. En esta sección, se introducirán los pipes integrados y se explicará cómo se pueden enlazar con los controladores de rutas. Posteriormente, se analizarán varios ejemplos de pipes personalizados para ilustrar cómo crear uno desde cero.

CONSEJO
Los pipes se ejecutan en el ámbito de las excepciones, lo que significa que, si un pipe genera una excepción, esta será manejada por la capa de excepciones, incluyendo el filtro global de excepciones y cualquier filtro de excepciones aplicado al contexto actual. Por lo tanto, es importante comprender que cuando una excepción se lanza en un pipe, ningún método del controlador se ejecutará después de eso. Esta práctica es valiosa para validar los datos que ingresan a la aplicación desde fuentes externas de los límites del sistema.

*1) Pipes integrados*

Nest incluye nueve pipes disponibles de forma predeterminada [10]:

- ValidationPipe
- ParseIntPipe
- ParseFloatPipe
- ParseBoolPipe

- ParseArrayPipe
- ParseUUIDPipe
- ParseEnumPipe
- DefaultValuePipe
- ParseFilePipe

Los pipes mencionados se encuentran disponibles en el paquete **@nestjs/common**.

A modo de ilustración, considere el uso de **ParseIntPipe** como ejemplo. Este caso ejemplifica una transformación, donde el pipe asegura que un parámetro del método del controlador se convierta en un número entero de JavaScript, o en su defecto, arroja una excepción si la conversión no es exitosa. En las próximas secciones, se presentará una implementación sencilla y personalizada de **ParseIntPipe**. Las técnicas descritas a continuación son aplicables también a otros pipes de transformación integrados, como **ParseBoolPipe**, **ParseFloatPipe**, **ParseEnumPipe**, **ParseArrayPipe** y **ParseUUIDPipe**, a los que nos referiremos genéricamente como **pipes Parse*** en esta sección.

*2) Uniendo pipes*

Para emplear un pipe, es necesario vincular una instancia de la clase del pipe al contexto correspondiente. En el caso del **ParseIntPipe**, se busca asociar dicho pipe a un método del manejador de rutas particular y garantizar que se ejecute previamente a la invocación del método. Esta acción se lleva a cabo mediante la siguiente estructura, que se denomina vinculación del pipe al nivel del parámetro del método.

```
@Get(':id')
async findOne(@Param('id', ParseIntPipe) id: number) {
  return this.catsService.findOne(id);
}
```

Esta medida garantiza que una de las siguientes dos condiciones se cumpla: primero, el parámetro proporcionado al método **findOne()** es un número, conforme a lo anticipado en la llamada a **this.catsService.findOne()**, o segundo, se produce una excepción antes de que el controlador de la ruta sea invocado.

Por ejemplo, en el caso de que la ruta sea nombrada de la siguiente manera:

```
GET localhost:3000/abc
```

Nest lanzará una excepción como ésta:

```
{
  "statusCode": 400,
  "message": "Validation failed (numeric string is expected)",
  "error": "Bad Request"
}
```

La excepción evitará que se ejecute el contenido del método **findOne()**.

En el ejemplo previo, se ha proporcionado una clase (**ParseIntPipe**) en lugar de una instancia, trasladando la responsabilidad de la instanciación al marco trabajo, lo cual habilita la inyección de dependencias. Del mismo modo que ocurre con los pipes y guards, existe la alternativa de pasar una instancia directamente. La elección de pasar una instancia en lugar de una clase es beneficiosa cuando se pretende personalizar el comportamiento del pipe integrado al incluir opciones específicas.

```
@Get(':id')
async findOne(
  @Param('id', new ParseIntPipe({ errorHttpStatusCode: HttpStatus.NOT_ACCEPTABLE }))
  id: number,
) {
  return this.catsService.findOne(id);
}
```

La vinculación de los otros pipes de transformación (todos los pipes Parse*) funciona de manera similar. Estos pipes funcionan en el ámbito de la validación de los parámetros de la ruta, los parámetros de la cadena de consulta y los valores del cuerpo de la solicitud.

Por ejemplo, con un parámetro como cadena de consulta:

```
@Get()
async findOne(@Query('id', ParseIntPipe) id: number) {
  return this.catsService.findOne(id);
}
```

A continuación, se presenta un ejemplo de la utilización del **ParseUUIDPipe**, el cual se emplea para analizar un parámetro en formato de cadena de texto y comprobar su validez como un **UUID**.

```
@Get(':uuid')
async findOne(@Param('uuid', new ParseUUIDPipe()) uuid: string) {
  return this.catsService.findOne(uuid);
}
```

CONSEJO
Cuando se hace uso de la función **ParseUUIDPipe()**, se efectúa el análisis de UUID en su versión 3, 4 o 5. No obstante, si se requiere únicamente una versión particular de UUID, es posible especificar dicha versión mediante las opciones proporcionadas por el pipe.

En entregas previas, se han presentado ejemplos de cómo enlazar las clases de pipes incorporadas dentro de la familia Parse*. No obstante, el proceso de vinculación de los pipes de validación se diferencia ligeramente; este tema se abordará en el próximo apartado.

CONSEJO
Además, consulte las técnicas de validación para ver ejemplos exhaustivos de pipes de validación.

*3) Pipes personalizados*

Como se ha explicado, es factible crear pipes personalizadas según las necesidades del desarrollador. A pesar de que Nest dispone de **ParseIntPipe** y **ValidationPipe** integrados de alta calidad, se explorara cómo crear versiones personalizadas sencillas de ambos desde cero, con el propósito de entender el proceso de desarrollo de pipes personalizados.

Para configurar un **ValidationPipe** este debe recibir un valor de entrada y, de manera inmediata, devolver ese mismo valor, funcionando esencialmente como una función de identificación.

validation.pipe.ts

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

@Injectable()
export class ValidationPipe implements PipeTransform {
  transform(value: any, metadata: ArgumentMetadata) {
    return value;
  }
}
```

CONSEJO
**PipeTransform<T, R>** es una interfaz genérica que debe ser implementada por cualquier pipe. La interfaz genérica utiliza **T** para indicar el tipo del valor de entrada y **R** para indicar el tipo de retorno del método **transform()**.

Cada pipe debe implementar el método **transform()** para cumplir con el contrato de la interfaz **PipeTransform**. Este método tiene dos parámetros: **valor** y **metadata**.

El parámetro valor es el argumento del método actualmente procesado (antes de que sea recibido por el método de manejo de rutas), y la metadata es la metainformación del argumento del método actualmente procesado. El objeto de metadatos tiene las siguientes propiedades:

```
export interface ArgumentMetadata {
  type: 'body' | 'query' | 'param' | 'custom';
  metatype?: Type<unknown>;
  data?: string;
}
```

Estas propiedades describen el argumento procesado actualmente.

*Tabla 6 Propiedades para la descripción de argumentos de un pipe personalizado*

| | |
|---|---|
| type | Indica si el argumento es un cuerpo **@Body()**, una consulta **@Query()**, un parámetro **@Param()** o un parámetro personalizado (para leer más al respecto siga el enlace). |
| metatype | Proporciona el metatipo del argumento, por ejemplo, String. **Nota**: el valor es indefinido si se omite una declaración de tipo en la declaración del método para el manejo de rutas o si se utiliza JavaScript estándar. |
| data | La cadena de texto que pasada al decorador, por ejemplo, **@Body('cadena')**. El valor es indefinido si se deja el paréntesis del decorador vacío. |

*Fuente: Autor*

ADVERTENCIA
Las interfaces de TypeScript desaparecen durante la transpilación. Por lo tanto, si el tipo de un parámetro del método se declara como interfaz en lugar de una clase, el valor del metatipo será **Object**.

*4) Validación basada en esquemas*

Para mejorar la utilidad del pipe de validación, se debe examinar con mayor detalle el método **create()** del **CatsController**, en el que se debería garantizar que el objeto del cuerpo de la publicación sea válido antes de intentar ejecutar el método del servicio.

```
@Post()
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
```

Este sería el aspecto del parámetro **createCatDto** en el que su tipo se define como **CreateCatDto:**

```
create-cat.dto.ts
export class CreateCatDto {
  name: string;
  age: number;
  breed: string;
}
```

Se desea garantizar que cualquier solicitud que llegue al método **create()** contenga un cuerpo válido. Por lo tanto, es necesario verificar los tres componentes del objeto **createCatDto**. Aunque se podría realizar esta validación dentro del manejador de rutas, esa no es la opción óptima, ya que infringiría el principio de responsabilidad única (SRP).

Se podría optar por un enfoque alternativo que implica la creación de una clase de validación y delegar la tarea a dicha clase. Sin embargo, la desventaja de este enfoque radica en la necesidad de invocar este validador al inicio de cada método.

¿Crear un middleware de validación? Aunque esta opción podría resultar efectiva, lamentablemente, no es factible desarrollar un middleware genérico que funcione en todos los contextos de la aplicación. Esto se debe a que el middleware carece de conocimiento sobre el contexto de ejecución, incluyendo el manejador que se invocará y sus respectivos parámetros. Claro, este es precisamente el escenario para el cual se diseñan los pipes. Por lo tanto, se puede continuar perfeccionando el pipe de validación.

*5) Validación del esquema de objetos*

Se pueden encontrar múltiples enfoques para llevar a cabo la validación de objetos de forma eficiente y siguiendo el principio de DRY (Don't Repeat Yourself). Una estrategia habitual implica la utilización de la validación basada en esquemas. Se prosigue a explorar y experimentar con dicho enfoque.

La biblioteca Zod facilita la creación de esquemas de manera simple, a través de una API legible. Un pipe de validación se desarrollará haciendo uso de estos esquemas proporcionados por Zod.

La primera tarea consiste en la instalación del paquete:

```
$ npm install --save zod
```

En el ejemplo de código que se presenta a continuación, se muestra la creación de una clase simple que recibe un esquema como parámetro en su constructor. Luego, se utiliza el método **schema.parse()** para llevar a cabo la validación de un argumento de entrada utilizando el esquema proporcionado.

Tal como se indicó previamente, un pipe de validación puede devolver el valor sin modificar o generar una excepción, dependiendo de las circunstancias.

En la siguiente sección, se abordará cómo suministrar el esquema apropiado para un método de un controlador específico, haciendo uso del decorador **@UsePipes()**. Esta práctica facilita la reutilización del pipe de validación en diversos contextos, conforme a la intención inicial.

```
import { PipeTransform, ArgumentMetadata, BadRequestException } from '@nestjs/common';
import { ZodObject } from 'zod';

export class ZodValidationPipe implements PipeTransform {
  constructor(private schema: ZodObject<any>) {}

  transform(value: unknown, metadata: ArgumentMetadata) {
    try {
      this.schema.parse(value);
    } catch (error) {
      throw new BadRequestException('Validation failed');
    }
    return value;
  }
}
```

*6) Unir pipes de validación*

Para unir pipes de validación, como se demostró anteriormente con los pipes de transformación (como **ParseIntPipe** y otros pipes **Parse***), el proceso es igual de sencillo.

En este contexto, la vinculación de pipes de validación se realiza a nivel de la llamada del método correspondiente. En el ejemplo actual, los pasos a seguir para utilizar **ZodValidationPipe** son los siguientes:

1. Se debe crear una instancia de **ZodValidationPipe**.
2. Se debe proporcionar el esquema Zod en el constructor de la clase del pipe.
3. Por último, se vincula el pipe al método deseado.

A continuación, se muestra un ejemplo de un esquema Zod:

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

export const createCatSchema = z
  .object({
    name: z.string(),
    age: z.number(),
    breed: z.string(),
  })
  .required();

export type CreateCatDto = z.infer<typeof createCatSchema>;
```

Este procedimiento se lleva a cabo empleando el decorador **@UsePipes()**, tal como se ilustra en el siguiente fragmento de código:

cats.controller.ts

```
@Post()
@UsePipes(new ZodValidationPipe(createCatSchema))
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
```

CONSEJO
El decorador **@UsePipes()** se importa desde el paquete **@nestjs/common**.

ADVERTENCIA
La biblioteca zod requiere que la configuración **strictNullChecks** esté habilitada en el archivo tsconfig.json.

*7) Validador de clases*

ADVERTENCIA
Las técnicas en esta sección requieren TypeScript y no están disponibles si la aplicación está escrita en JavaScript estándar.

Se presenta una alternativa de implementación para validar datos. Nest es compatible con la biblioteca **class-validator**, la cual ofrece la posibilidad de utilizar la validación basada en decoradores. Esta forma de validación es altamente eficaz, especialmente cuando se utiliza en conjunto con las funciones de los Pipes de Nest, que permiten acceder al metatipo de la propiedad que se está procesando. Antes de iniciar, es necesario llevar a cabo la instalación de los paquetes necesarios.

```
$ npm i --save class-validator class-transformer
```

Cuando se hayan realizado las instalaciones necesarias, será posible incorporar ciertos decoradores a la clase **CreateCatDto**. En esta situación, se aprecia una ventaja considerable de este enfoque, ya que la clase **CreateCatDto** continúa siendo la fuente primordial para el objeto del cuerpo de la solicitud POST, evitando la necesidad de crear una clase de validación independiente.

create-cat.dto.ts

```
import { IsString, IsInt } from 'class-validator';

export class CreateCatDto {
  @IsString()
  name: string;

  @IsInt()
  age: number;

  @IsString()
  breed: string;
}
```

CONSEJO
Lea más sobre los decoradores de class-validator aquí.

En este momento, es factible desarrollar una clase **ValidationPipe** que haga uso de estas notaciones.

validation.pipe.ts

```
import { PipeTransform, Injectable, ArgumentMetadata, BadRequestException } from
'@nestjs/common';
import { validate } from 'class-validator';
import { plainToInstance } from 'class-transformer';

@Injectable()
export class ValidationPipe implements PipeTransform<any> {
  async transform(value: any, { metatype }: ArgumentMetadata) {
    if (!metatype || !this.toValidate(metatype)) {
      return value;
    }
    const object = plainToInstance(metatype, value);
    const errors = await validate(object);
    if (errors.length > 0) {
      throw new BadRequestException('Validation failed');
    }
    return value;
  }

  private toValidate(metatype: Function): boolean {
    const types: Function[] = [String, Boolean, Number, Array, Object];
    return !types.includes(metatype);
  }
}
```

CONSEJO
Como recordatorio, no es necesario construir un pipe de validación genérico, ya que Nest proporciona el **ValidationPipe** de forma predeterminada. El **ValidationPipe** integrado ofrece más opciones que el ejemplo que se construyó en esta sección, que se mantuvo básico con el fin de ilustrar la mecánica de un pipe personalizado. Para más detalles, junto con muchos ejemplos, siga el enlace.

AVISO
Se utilizó la biblioteca class-transformer anteriormente, que está hecha por el mismo autor que la biblioteca **class-validator**, y como resultado, funcionan muy bien juntas.

Se hará un análisis de este código. En primer lugar, se debe tener en cuenta que el método **transform()** se ha marcado como asíncrono, lo cual es posible debido a la capacidad de Nest de admitir tanto pipes sincrónicos como asincrónicos. La asincronía se emplea en este método debido a que algunas de las validaciones del **class-validator** pueden requerir operaciones asincrónicas, haciendo uso de **Promesas**.

A continuación, se observa que se utiliza la desestructuración para extraer el campo **metatype** (obteniendo solamente esta parte de un **ArgumentMetadata**) y asignarlo al parámetro metatype. Esta es una forma abreviada de obtener todo el **ArgumentMetadata** y, en lugar de ello, se opta por una declaración adicional que asigne la variable metatype.

Luego, se presta atención a la función auxiliar **toValidate()**. Su función es evitar el proceso de validación cuando el argumento actual que se está procesando es un tipo nativo de JavaScript. Esto se debe a que los tipos nativos no pueden contar con decoradores de validación adjuntos, por lo que no es necesario ejecutarlos a través del proceso de validación.

A continuación, se emplea la función **plainToInstance()** proveniente de **class-transformer** para llevar a cabo la transformación del objeto de argumento de JavaScript plano a un objeto tipado. Esta transformación es necesaria para realizar la validación. La razón detrás de esta necesidad radica en que el objeto correspondiente al cuerpo de la solicitud POST entrante, al ser de-serializado desde la solicitud de red, carece de cualquier información de tipo. Esto se debe al funcionamiento de la plataforma subyacente, como Express. Para que **Class-validator** pueda hacer uso de los decoradores de validación definidos previamente en el DTO, es necesario realizar esta transformación, de manera que el cuerpo entrante pueda ser tratado como un objeto debidamente decorado en lugar de ser considerado un objeto plano.

Por último, como se mencionó anteriormente, dado que se trata de un pipe de validación, este puede retornar el valor sin cambios o generar una excepción.

El último paso consiste en la vinculación del **ValidationPipe**. Los pipes pueden tener un ámbito de parámetro, método, controlador o global. En un ejemplo previo, con el pipe de validación basado en Joi, se pudo apreciar cómo vincular el pipe a nivel de método. En el siguiente ejemplo, se vinculará la instancia del pipe al decorador **@Body()** en el manejador de ruta, permitiendo que nuestro pipe sea invocado para validar el cuerpo de la solicitud POST.

cats.controller.ts

```
@Post()
async create(
  @Body(new ValidationPipe()) createCatDto: CreateCatDto,
) {
  this.catsService.create(createCatDto);
}
```

Los pipes de ámbito de parámetro son útiles cuando la lógica de validación se refiere solo a un parámetro especificado.

*8) Pipes de ámbito global*

Debido a que se diseñó el **ValidationPipe** con el propósito de que fuera lo más versátil posible, es factible maximizar su utilidad al configurarlo como un pipe de ámbito global, permitiendo que se aplique a cada manejador de ruta en la totalidad de la aplicación.

main.ts

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe());
  await app.listen(3000);
}
bootstrap();
```

AVISO

En el caso de las aplicaciones híbridas, el método **useGlobalPipes()** no configura los pipes para **gateways** y **microservicios**. Para aplicaciones de microservicio "estándar" (no híbridas), **useGlobalPipes()** define los pipes de forma global.

Los pipes globales se emplean en la totalidad de la aplicación, abarcando todos los controladores y manejadores de rutas.

Es importante destacar que en lo que respecta a la inyección de dependencias, los pipes globales que se registran desde fuera de cualquier módulo, como se mostró en el ejemplo anterior con el uso de **useGlobalPipes()**, no tienen la capacidad de inyectar dependencias, ya que la vinculación se realiza en un contexto externo a cualquier módulo específico. Para resolver esta limitación, es posible configurar un pipe global directamente desde cualquier módulo, utilizando la siguiente estructura:

app.module.ts

```
import { Module } from '@nestjs/common';
import { APP_PIPE } from '@nestjs/core';

@Module({
  providers: [
    {
      provide: APP_PIPE,
      useClass: ValidationPipe,
    },
  ],
})
export class AppModule {}
```

CONSEJO
Al aplicar esta estrategia para llevar a cabo la inyección de dependencias en el pipe, es esencial recordar que, sin importar el módulo en el que se implemente esta estructura, el pipe se considera global. ¿Dónde se debería llevar a cabo esta configuración? Seleccione el módulo en el cual se define el pipe (como en el caso del **ValidationPipe** del ejemplo anterior). Además, es importante destacar que el uso de **useClass** no es la única manera de gestionar el registro de proveedores personalizados. Puede obtener información adicional sobre este tema en el siguiente enlace.

*9) El ValidationPipe integrado*

Es importante destacar que no es necesario desarrollar un pipe de validación genérico de forma personal, ya que Nest ofrece el **ValidationPipe** de manera predeterminada. El **ValidationPipe** incorporado proporciona una variedad de opciones más amplias en comparación con el ejemplo que se ha construido en esta sección, el cual se mantuvo sencillo con el propósito de ilustrar la mecánica de un pipe personalizado. Puedes acceder a todos los detalles, así como numerosos ejemplos, en este recurso.

*10) Caso práctico de transformación*

La validación no constituye el único escenario de uso para los pipes personalizados. Tal como se mencionó al comienzo de esta sección, un pipe también puede desempeñar la función de transformar los datos de entrada al formato deseado. Esto se logra gracias a que el valor devuelto por la función **transform()** reemplaza por completo el valor anterior del argumento.

¿En qué situaciones resulta útil este enfoque? Es relevante considerar que, en ocasiones, los datos enviados por el cliente requieren ciertas modificaciones, como la conversión de una cadena de texto a un valor entero, antes de poder ser adecuadamente procesados por el método del manejador de rutas. Además, podría suceder que falten algunos datos obligatorios, y se desee aplicar valores predeterminados. Los pipes de transformación son capaces de llevar a cabo estas funciones al introducir una función de procesamiento entre la solicitud del cliente y el manejador de solicitud.

A continuación, se presenta un ejemplo sencillo de un **ParseIntPipe**, diseñado para convertir una cadena de texto en un valor entero. (Es relevante destacar que, como se mencionó previamente, Nest incluye un

**ParseIntPipe** incorporado con características más avanzadas. La presente implementación se ofrece como un ejemplo básico de un pipe de transformación personalizado).

parse-int.pipe.ts

```
import { PipeTransform, Injectable, ArgumentMetadata, BadRequestException } from
'@nestjs/common';

@Injectable()
export class ParseIntPipe implements PipeTransform<string, number> {
  transform(value: string, metadata: ArgumentMetadata): number {
    const val = parseInt(value, 10);
    if (isNaN(val)) {
      throw new BadRequestException('Validation failed');
    }
    return val;
  }
}
```

Posteriormente, es posible asociar este pipe con el parámetro elegido, como se ilustra a continuación:

```
@Get(':id')
async findOne(@Param('id', new ParseIntPipe()) id) {
  return this.catsService.findOne(id);
}
```

Otro caso útil de transformación sería seleccionar una entidad de usuario existente en la base de datos utilizando un ID proporcionado en la solicitud:

```
@Get(':id')
findOne(@Param('id', UserByIdPipe) userEntity: UserEntity) {
  return userEntity;
}
```

La creación de este pipe se deja como tarea al lector. Sin embargo, es importante tener en cuenta que, al igual que cualquier otro pipe de transformación, este pipe recibe un valor de entrada (un ID) y produce un valor de salida (un objeto **UserEntity**). Esta práctica puede contribuir a hacer el código más declarativo y mantener el principio DRY (Don't Repeat Yourself) al volver abstracto el código común fuera del manejador y centralizarlo en un pipe compartido.

*11) Proporcionar valores por defecto*

Los pipes **Parse*** requieren que el valor de un parámetro esté explícitamente definido. En caso de recibir valores nulos o indefinidos, generan una excepción. Para habilitar la gestión de valores faltantes en los parámetros de la cadena de consulta de un endpoint, es necesario suministrar un valor predeterminado que se inyectará antes de que los pipes **Parse*** procesen estos valores. El **DefaultValuePipe** cumple con este propósito. Para su implementación, simplemente se instancia un **DefaultValuePipe** en el decorador **@Query()** antes del pipe **Parse*** correspondiente, como se muestra en el siguiente ejemplo:

```
@Get()
async findAll(
  @Query('activeOnly', new DefaultValuePipe(false), ParseBoolPipe) activeOnly: boolean,
  @Query('page', new DefaultValuePipe(0), ParseIntPipe) page: number,
) {
  return this.catsService.findAll({ activeOnly, page });
}
```

## *I. Guards [11]*

Un guard es una clase anotada con el decorador **@Injectable()**, que implementa la interfaz **CanActivate**.

*Imagen 11 Diagrama de funcionamiento de los guards*


*Fuente: Autor*

Los guards desempeñan una función única al determinar si una solicitud específica será manejada por el controlador, dependiendo de ciertas condiciones (como permisos, roles, ACL, etc.) que existen durante la ejecución. Esto se conoce comúnmente como autorización. En las aplicaciones tradicionales de Express, la autorización (junto con su contraparte, la autenticación) generalmente se han manejado mediante el uso de middlewares. El middleware es una opción adecuada para la autenticación, ya que tareas como la validación de tokens y la adición de propiedades al objeto de solicitud no están fuertemente vinculadas a un contexto de ruta específico (y sus metadatos).

Sin embargo, el middleware tiene limitaciones debido a su naturaleza. No tiene conocimiento sobre qué controlador se ejecutará después de llamar a la función **next()**. Además, los guards tienen acceso a la instancia de **ExecutionContext** y, por lo tanto, saben exactamente qué se ejecutará a continuación. Están diseñados, al igual que los filtros de excepción, los pipes y los interceptores, para interponer lógica de procesamiento en el momento preciso del ciclo de solicitud-respuesta y hacerlo de manera declarativa. Esto ayuda a mantener el código DRY (Don't Repeat Yourself) y declarativo.

CONSEJO
Los guards se ejecutan después de todos los middlewares, pero antes de cualquier interceptor o pipe.

### *1) Guard de autorización*

Como se ha explicado anteriormente, los Guards son ideales para implementar la autorización, ya que ciertas rutas deben ser accesibles solo cuando el solicitante, generalmente un usuario autenticado específico, cuenta con los permisos necesarios. El **AuthGuard** que se construirá a continuación se basa en la premisa de que se encuentra un usuario autenticado (lo que implica que un token se adjunta a las cabeceras de la solicitud). Este guard se encargará de extraer y validar el token, utilizando la información obtenida para determinar si la solicitud puede continuar o debe ser denegada.

auth.guard.ts

```
import { Injectable, CanActivate, ExecutionContext } from '@nestjs/common';
import { Observable } from 'rxjs';

@Injectable()
export class AuthGuard implements CanActivate {
  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {
    const request = context.switchToHttp().getRequest();
    return validateRequest(request);
  }
}
```

CONSEJO
Para obtener una muestra práctica de cómo implementar un sistema de autenticación en una aplicación, se sugiere revisar este enlace. De igual manera, si se desea obtener un ejemplo de autorización más avanzado, se puede consultar este enlace.

La lógica dentro de la función **validateRequest()** puede ser tan simple o sofisticada como sea necesario. El punto principal de este ejemplo es mostrar cómo encajan los guards en el ciclo de solicitud-respuesta.

Cada guard debe implementar una función **canActivate()**. Esta función debe devolver un valor booleano, indicando si la solicitud actual está permitida o no. Puede devolver la respuesta de forma síncrona o asíncrona (a través de una **Promesa** o un **Observable**). Nest utiliza el valor de retorno para controlar la siguiente acción:

- Si devuelve **true**, se procesará la solicitud.
- Si devuelve **false**, Nest denegará la solicitud.

*2) Contexto de ejecución*

La función **canActivate()** acepta un único argumento, que es una instancia de **ExecutionContext**. **ExecutionContext** hereda de **ArgumentsHost**, un concepto que hemos explorado anteriormente en la sección dedicada a los filtros de excepción. En el ejemplo anterior, utilizamos los mismos métodos auxiliares definidos en **ArgumentsHost**, que ya habíamos empleado previamente, para obtener una referencia al objeto **Request**.

Dado que **ExecutionContext** extiende **ArgumentsHost**, también incluye una serie de nuevos métodos auxiliares que proporcionan detalles adicionales acerca del proceso de ejecución en curso. Estos detalles pueden resultar útiles para la creación de guards más versátiles, capaces de funcionar en diversos controladores, métodos y contextos de ejecución. Se puede obtener información adicional sobre **ExecutionContext** en este enlace.

*3) Autentificación basada en roles*

Se planea la creación de un guard más efectivo que limitará el acceso exclusivamente a aquellos usuarios que cuenten con un rol específico. Inicialmente, se partirá de una plantilla de guard simple, la cual se optimizará en las siguientes secciones. Actualmente, se permitirá que todas las solicitudes pasen sin limitaciones.

roles.guard.ts

```
import { Injectable, CanActivate, ExecutionContext } from '@nestjs/common';
import { Observable } from 'rxjs';

@Injectable()
export class RolesGuard implements CanActivate {
  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {
    return true;
  }
}
```

*4) Uniendo Guards*

De manera similar a los pipes y los filtros de excepción, se pueden definir guards con ámbito de controlador, ámbito de método o ámbito global. A continuación, se muestra la configuración de un guard de ámbito de controlador utilizando el decorador **@UseGuards()**. Este decorador tiene la capacidad de recibir un solo argumento o una lista de argumentos separados por comas, lo que facilita la aplicación de un conjunto apropiado de guards mediante una única declaración.

```
@Controller('cats')
@UseGuards(RolesGuard)
export class CatsController {}
```

CONSEJO

El decorador **@UseGuards()** se importa desde el paquete **@nestjs/common**.

En el ejemplo anterior, se proporcionó la clase **RolesGuard** en lugar de una instancia, lo que delega la responsabilidad de la creación al marco de trabajo y habilita la inyección de dependencias. Similar a lo que se hace con los pipes y los filtros de excepción, también es posible pasar una instancia en su lugar:

```
@Controller('cats')
@UseGuards(new RolesGuard())
export class CatsController {}
```

La estructura previa vincula el guard a todos los métodos declarados dentro de ese controlador. Si la intención es que el guard se aplique únicamente a un método en particular, se emplea el decorador **@UseGuards()** a nivel del método correspondiente.

Si se pretende configurar un guard de ámbito global, se recurre al método **useGlobalGuards()** de la instancia de la aplicación Nest.

```
const app = await NestFactory.create(AppModule);
app.useGlobalGuards(new RolesGuard());
```

AVISO
En el caso de las aplicaciones híbridas, el método **useGlobalGuards()** no configura los guards para microservicios y gateways de forma predeterminada (consultar aplicación híbrida para obtener información sobre cómo cambiar este comportamiento). Para aplicaciones de microservicios "estándar" (no híbridas), **useGlobalGuards()** define los guards de forma global.

Los guards globales se pueden aplicar en toda la aplicación, abarcando cada controlador y manejador de rutas. En lo que respecta a la inyección de dependencias, los guards globales que se registran desde fuera de cualquier módulo (mediante **useGlobalGuards()**, como se mostró en el ejemplo previo) carecen de la capacidad de inyectar dependencias, dado que esta acción se realiza fuera del contexto de cualquier módulo. Para solucionar este inconveniente, es posible configurar un guard directamente desde cualquier módulo mediante la siguiente estructura:

app.module.ts

```
import { Module } from '@nestjs/common';
import { APP_GUARD } from '@nestjs/core';

@Module({
  providers: [
    {
      provide: APP_GUARD,
      useClass: RolesGuard,
    },
  ],
})
export class AppModule {}
```

CONSEJO
Al aplicar este método para inyectar dependencias en el guard, es fundamental tener en cuenta que, sin importar el módulo donde se implemente esta configuración, el guard se comporta como global. La elección del módulo en el que se define el guard (como en el caso de **RolesGuard** en el ejemplo anterior) es crucial. Cabe señalar que **useClass** no constituye la única manera de gestionar el registro de proveedores personalizados. Para obtener información adicional al respecto, se sugiere explorar más detalles en este enlace.

*5) Configurando roles por manejador*

El **RolesGuard** actualmente en uso demuestra funcionar, no obstante, su nivel de inteligencia aún resulta limitado. En la actualidad, no se aprovecha la característica más esencial de los guards, que es el contexto de ejecución. El guard aún carece de conocimiento sobre los roles disponibles y cuáles roles son permitidos para cada manejador en particular. Por ejemplo, el **CatsController** podría poseer distintos esquemas de permisos para rutas diferentes. Algunas de estas rutas podrían estar restringidas únicamente para usuarios con rol de administrador, mientras que otras podrían estar disponibles para cualquier usuario. La pregunta que se plantea es: ¿Cómo establecer una relación flexible y reutilizable entre los roles y las rutas?

Aquí es donde entra en juego el metadato personalizado (más información aquí). Nest proporciona la capacidad de adjuntar metadatos personalizados a los manejadores de rutas a través de decoradores creados ya sea mediante el método estático **Reflector#createDecorator** o el decorador incorporado **@SetMetadata()**.

Como ilustración, se podría diseñar un decorador denominado **@Roles()** mediante el empleo del método **Reflector#createDecorator**, el cual vinculará la información de los metadatos al manejador correspondiente. El **Reflector** es una funcionalidad incorporada en el marco de trabajo por defecto y se encuentra accesible a través del paquete **@nestjs/core**.

roles.decorator.ts

```
import { Reflector } from '@nestjs/core';

export const Roles = Reflector.createDecorator<string[]>();
```

En este contexto, el decorador **Roles** se define como una función que recibe un único argumento de tipo **string[]**.

Para aplicar este decorador, basta con etiquetarlo en el manejador correspondiente:

cats.controller.ts

```
@Post()
@Roles(['admin'])
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
```

En este caso, se ha asociado el metadato proporcionado por el decorador **Roles** al método **create()**, especificando que solamente los usuarios que posean el rol de administrador tienen permiso para acceder a esta ruta.

Otra opción que se plantea es emplear el decorador incorporado **@SetMetadata** en lugar de utilizar el método **Reflector#createDecorator**. Puede encontrar información adicional al respecto en este enlace.

*6) Ponerlo todo junto*

En este momento, se debe regresar y establecer una conexión entre el concepto mencionado y el **RolesGuard**. En el estado actual, el guard en cuestión simplemente devuelve **true** sin importar la situación, lo que conlleva a que todas las solicitudes sean autorizadas sin restricciones. La intención es que el valor de retorno se vuelva condicional, tomando como base la comparación entre los roles asignados al usuario actual y los roles necesarios para la ruta específica que se está procesando en ese instante. Para acceder a los roles relacionados con la ruta, es necesario utilizar nuevamente la clase auxiliar **Reflector** de la siguiente manera:

roles.guard.ts

```
import { Injectable, CanActivate, ExecutionContext } from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { Roles } from './roles.decorator';

@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const roles = this.reflector.get(Roles, context.getHandler());
    if (!roles) {
      return true;
    }
    const request = context.switchToHttp().getRequest();
    const user = request.user;
    return matchRoles(roles, user.roles);
  }
}
```

CONSEJO
En el contexto de Node.js, se suele realizar la práctica de incluir al usuario autorizado en el objeto de solicitud (request). Por lo tanto, en el ejemplo de código previo, se parte del supuesto de que **request.user** contiene la instancia del usuario y los roles que tiene permitidos. Para obtener más detalles acerca de este asunto, se recomienda consultar el siguiente enlace.

ADVERTENCIA
La lógica dentro de la función **matchRoles()** puede ser tan simple o sofisticada como sea necesaria. El objetivo principal de este ejemplo es mostrar cómo encajan los guards en el ciclo de solicitud-respuesta.

Se recomienda revisar el siguiente recurso titulado "Reflection and metadata" que aborda el tema del uso adecuado de **Reflector**.

En caso de que un usuario con privilegios limitados solicite un punto de acceso, Nest proporciona de forma automática la siguiente respuesta:

```
{
  "statusCode": 403,
  "message": "Forbidden resource",
  "error": "Forbidden"
}
```

Es importante tener en cuenta que, en el proceso interno, cuando un guard retorna **false**, el marco de trabajo genera una **ForbiddenException**. Si se requiere emitir una respuesta de error personalizada, es necesario lanzar una excepción específica de forma manual, como se ilustra en el siguiente ejemplo:

```
throw new UnauthorizedException();
```

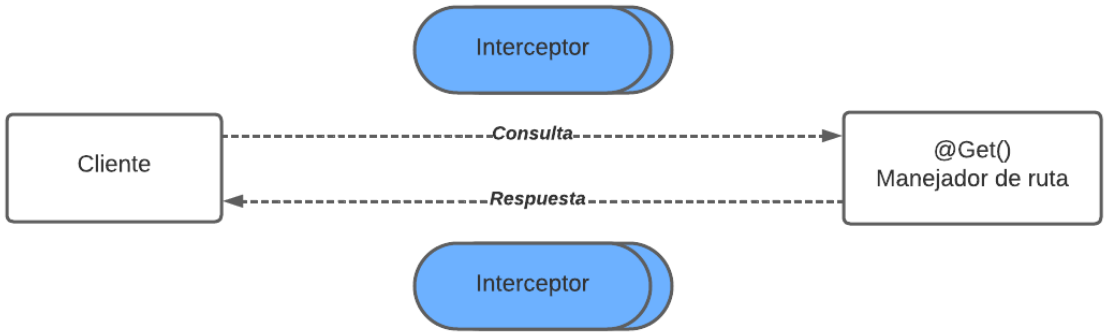
Cualquier excepción lanzada por un guard será manejada por la capa de excepciones (filtro global de excepciones y cualquier filtro de excepciones que se aplique al contexto actual).

CONSEJO
Si se busca un ejemplo del mundo real sobre cómo implementar la autorización, consulte este enlace.

*J. Interceptors [12]*

Un interceptor es una clase anotada con el decorador **@Injectable()** e implementa la interfaz **NestInterceptor**.

*Imagen 12 Diagrama de funcionamiento de los interceptores*




*Fuente: Autor*

Los interceptores tienen un conjunto de capacidades útiles que están inspiradas en la técnica de Programación Orientada a Aspectos (**AOP**, por sus siglas en inglés). Lo que permite:

- Vincular lógica adicional antes/después de la ejecución de un método.
- Transformar el resultado devuelto por una función.
- Transformar la excepción lanzada por una función.
- Extender el comportamiento básico de una función.
- Sobrescribir completamente una función dependiendo de condiciones específicas (por ejemplo, con fines de almacenamiento en caché).

*1) Conceptos Básicos*

Cada interceptor implementa el método **intercept()**, el cual recibe dos parámetros. El primero es la instancia de **ExecutionContext**, que es precisamente el mismo objeto que se utiliza para los guards. **ExecutionContext** hereda de **ArgumentsHost**, que ya se exploró en la sección sobre filtros de excepciones. En esa sección, se detalló que **ArgumentsHost** actúa como un envoltorio que contiene los argumentos pasados al controlador original y que incluye distintos conjuntos de argumentos dependiendo del tipo de aplicación.

*2) Contexto de ejecución*

Al extender **ArgumentsHost**, **ExecutionContext** también agrega varios métodos auxiliares nuevos que proporcionan detalles adicionales sobre el proceso de ejecución actual. Estos detalles pueden ser útiles para construir interceptores más genéricos que puedan funcionar en un amplio conjunto de controladores, métodos y contextos de ejecución. Para más información sobre **ExecutionContext** siga este enlace.

*3) Manejador de llamadas*

El segundo parámetro corresponde a un **CallHandler**. La interfaz **CallHandler** contiene el método **handle()**, el cual está a disposición para ser empleado en el momento en que el interceptor lo requiera, con el propósito de invocar el método del controlador de la ruta. Es importante destacar que si no se hace la llamada al método **handle()** dentro de la implementación del método **intercept()**, el método del controlador de la ruta no se ejecutará en ninguna circunstancia.

Este enfoque implica que el método **intercept()** esencialmente engloba el ciclo de solicitud-respuesta. Como consecuencia, es posible incorporar lógica personalizada tanto antes como después de la ejecución del controlador de la ruta final. Es importante destacar que, si bien se puede escribir código en el método **intercept()** que se ejecuta antes de la llamada a **handle()**, ¿cómo se puede influir en lo que ocurre posteriormente? La respuesta a esto radica en el hecho de que el método **handle()** devuelve un **Observable**, lo que permite hacer uso de potentes operadores de RxJS para efectuar manipulaciones adicionales en la respuesta. En términos de la Programación Orientada a Aspectos, la invocación del controlador de ruta (es decir, la llamada a **handle()**) se considera un "Punto de Corte", lo que significa que es el punto en el que se integra la lógica adicional.

Como ejemplo, en una solicitud entrante de tipo POST con la ruta **/cats,** esta solicitud está dirigida al controlador **create()** que se encuentra definido en el **CatsController**. Si se invoca un interceptor y este no realiza la llamada al método **handle()** en ningún punto del proceso, en consecuencia, el método **create()** no se llevará a cabo. Sin embargo, una vez que se efectúa la llamada a **handle()** (y se ha retornado su **Observable**), el controlador **create()** se activará. Posteriormente, una vez que se recibe el ciclo de respuesta a través del **Observable**, es factible ejecutar operaciones adicionales en ese ciclo y proporcionar un resultado final al solicitante.

*4) Interceptación de aspectos*

El primer escenario que se explorará consiste en emplear un interceptor para registrar las acciones del usuario, por ejemplo, la gestión de llamadas de usuario, el envío de eventos de manera asincrónica o el cálculo de una marca de tiempo. A continuación, se presenta un ejemplo básico de un **LoggingInterceptor**:

logging.interceptor.ts

```
import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from
'@nestjs/common';
import { Observable } from 'rxjs';
import { tap } from 'rxjs/operators';

@Injectable()
export class LoggingInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    console.log('Before...');

    const now = Date.now();
    return next
      .handle()
      .pipe(
        tap(() => console.log(`After... ${Date.now() - now}ms`)),
      );
  }
}
```

CONSEJO
**NestInterceptor<T, R>** es una interfaz genérica en la que **T** indica el tipo de un **Observable<T>** (que admite la transmisión de respuestas) y **R** es el tipo del valor envuelto por **Observable<R>**.

AVISO
Los interceptores, al igual que los controladores, proveedores, guards, y demás, pueden inyectar dependencias a través de su constructor.

Dado que el método **handle()** produce un **Observable** de **RxJS**, se cuenta con una amplia variedad de operadores disponibles para realizar modificaciones en el flujo de datos. En el ejemplo previo, se hizo uso del operador **tap()**, el cual ejecuta una función de registro de forma anónima al concluir el flujo del observable, ya sea de manera regular o excepcional, sin intervenir en el ciclo de respuesta de otra forma.

*5) Uniendo interceptores*

Para establecer la configuración del interceptor, se emplea el decorador **@UseInterceptors()**, el cual proviene del paquete **@nestjs/common**. De manera similar a los pipes y guards, los interceptores pueden ser aplicados con ámbito a nivel de controlador, a nivel de método o de manera global.

cats.controller.ts

```
@UseInterceptors(LoggingInterceptor)
export class CatsController {}
```

CONSEJO

El decorador **@UseInterceptors()** se importa del paquete **@nestjs/common**.

Siguiendo la estructura mencionada anteriormente, cada controlador de ruta establecido en **CatsController** hará uso de **LoggingInterceptor**. En caso de que se realice una solicitud al endpoint **GET /cats**, se visualizará el siguiente registro en la salida estándar:

```
Before...
After... 1ms
```

Es importante notar que se proporciona el tipo **LoggingInterceptor** (en lugar de una instancia), de modo que la creación de la instancia se delega al marco de trabajo, lo que permite la inyección de dependencias. Similar a lo que ocurre con los pipes, los guards y los filtros de excepciones, también existe la opción de pasar una instancia en su lugar:

cats.controller.ts

```
@UseInterceptors(new LoggingInterceptor())
export class CatsController {}
```

Como se indicó previamente, la estructura mencionada vincula el interceptor a cada manejador especificado en este controlador. En caso de que se desee limitar la aplicación del interceptor a un solo método, basta con aplicar el decorador a nivel del método correspondiente.

Para establecer un interceptor de ámbito global, se hace uso del método **useGlobalInterceptors()** de la instancia de la aplicación Nest:

```
const app = await NestFactory.create(AppModule);
app.useGlobalInterceptors(new LoggingInterceptor());
```

Los interceptores globales tienen un ámbito que abarca toda la aplicación, afectando a cada controlador y manejador de ruta por igual. En cuanto a la inyección de dependencias, es importante destacar que los interceptores globales registrados desde fuera de cualquier módulo, como se ilustra en el ejemplo previo mediante **useGlobalInterceptors()**, no tienen la capacidad de inyectar dependencias, ya que esto ocurre fuera del contexto de cualquier módulo. Para abordar esta limitación, es posible configurar un interceptor directamente desde cualquier módulo empleando la siguiente estructura:

app.module.ts

```
import { Module } from '@nestjs/common';
import { APP_INTERCEPTOR } from '@nestjs/core';

@Module({
  providers: [
    {
      provide: APP_INTERCEPTOR,
      useClass: LoggingInterceptor,
    },
  ],
})
export class AppModule {}
```

CONSEJO
Es importante tener en cuenta que, al aplicar este método para la inyección de dependencias en un interceptor, sin importar el módulo donde se utilice esta configuración, el interceptor mantendrá un ámbito global. La elección del lugar donde se debe realizar esto recae en el módulo donde el interceptor esté definido, como en el caso del **LoggingInterceptor** en el ejemplo previo. Cabe destacar que **useClass** no constituye la única opción para el registro personalizado de proveedores, por lo que es recomendable leer la información adicional al respecto en el siguiente enlace.

*6) Mapa de respuestas*

Como se ha mencionado previamente, se tiene constancia de que el método **handle()** produce un **Observable**. En este flujo se encuentra el valor retornado por el controlador de la ruta, lo que brinda la capacidad de realizar modificaciones de forma sencilla mediante el uso del operador **map()** de **RxJS**.

ADVERTENCIA
La función de mapeo de respuesta no funciona con la estrategia de respuesta específica de la biblioteca (usar directamente el objeto **@Res()** está prohibido).

Se busca crear un **TransformInterceptor** que realizará cambios mínimos en cada respuesta, con el propósito de ejemplificar el proceso. Este interceptor empleará el operador **map()** de **RxJS** para asignar el objeto de respuesta a la propiedad **data** de un objeto recién generado, y luego enviará el nuevo objeto al cliente como resultado.

transform.interceptor.ts

```
import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from
'@nestjs/common';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';

export interface Response<T> {
  data: T;
}

@Injectable()
export class TransformInterceptor<T> implements NestInterceptor<T, Response<T>> {
  intercept(context: ExecutionContext, next: CallHandler): Observable<Response<T>> {
    return next.handle().pipe(map(data => ({ data })));
  }
}
```

CONSEJO
Los interceptores de Nest funcionan con métodos **intercept()** tanto sincrónicos como asincrónicos. Se puede cambiar el método a **async** de ser necesario.

Con la construcción anterior, cuando alguien llama al endpoint **GET /cats**, la respuesta se vería como la siguiente (asumiendo que el controlador de la ruta devuelve una matriz vacía []):

```
{
  "data": []
}
```

Los interceptores desempeñan un papel significativo en la creación de soluciones que son reutilizables y aplicables a los requisitos de toda la aplicación. Por ejemplo, al tener la necesidad de transformar cada instancia de un valor nulo en una cadena de texto vacía. Esto se puede lograr con una sola línea de código y al asociar el interceptor de manera global, se garantiza que se aplique automáticamente en todos los controladores registrados.

```
import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from '@nestjs/common';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';

@Injectable()
export class ExcludeNullInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    return next
      .handle()
      .pipe(map(value => value === null ? '' : value ));
  }
}
```

*7) Mapeo de excepciones*

Otro caso de uso interesante es aprovechar el operador **catchError()** de **RxJS** para anular las excepciones lanzadas:

errors.interceptor.ts

```
import {
  Injectable,
  NestInterceptor,
  ExecutionContext,
  BadGatewayException,
  CallHandler,
} from '@nestjs/common';
import { Observable, throwError } from 'rxjs';
import { catchError } from 'rxjs/operators';

@Injectable()
export class ErrorsInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    return next
      .handle()
      .pipe(
        catchError(err => throwError(() => new BadGatewayException())),
      );
  }
}
```

*8) Anulación de flujos*

Existen diversas razones por las cuales a veces se prefiere evitar completamente la llamada al controlador y, en su lugar, optar por devolver un valor diferente. Un caso evidente es la implementación de una caché con el fin de mejorar el tiempo de respuesta. A continuación, se presentará un ejemplo sencillo de un interceptor de caché que proporciona su respuesta directamente desde una caché. En un escenario realista, se deben considerar otros factores como el tiempo de vida (TTL), la expiración de la caché, el tamaño de la caché, entre otros, pero estos aspectos se encuentran más allá del alcance de esta discusión. Aquí, se ofrece un ejemplo básico que ilustra el concepto fundamental.

cache.interceptor.ts

```
import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from
'@nestjs/common';
import { Observable, of } from 'rxjs';

@Injectable()
export class CacheInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    const isCached = true;
    if (isCached) {
      return of([]);
    }
    return next.handle();
  }
}
```

El **CacheInterceptor** que se ha implementado posee una variable denominada **isCached** y una respuesta vacía codificada como []. Es crucial destacar que en este caso se está retornando un nuevo flujo de datos, el cual es creado mediante el uso del operador **of()** de **RxJS**, lo que implica que el controlador de la ruta no será invocado en absoluto. Cuando alguien realiza una solicitud a un endpoint que emplea el **CacheInterceptor**, la respuesta, en este caso un array vacío, se devuelve de manera inmediata. Para crear una solución que sea más genérica, se puede aprovechar el **Reflector** y desarrollar un decorador personalizado.

*9) Mas operadores*

La capacidad de controlar el flujo mediante operadores de **RxJS** proporciona numerosas habilidades. Se puede contemplar otro ejemplo típico. Suponga que una persona desea gestionar los retrasos en las solicitudes de ruta. Cuando el endpoint no proporciona ningún resultado después de un tiempo determinado, se desea concluir con una respuesta de error. La siguiente estructura permite lograr esto:

timeout.interceptor.ts

```
import { Injectable, NestInterceptor, ExecutionContext, CallHandler,
RequestTimeoutException } from '@nestjs/common';
import { Observable, throwError, TimeoutError } from 'rxjs';
import { catchError, timeout } from 'rxjs/operators';

@Injectable()
export class TimeoutInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    return next.handle().pipe(
      timeout(5000),
      catchError(err => {
        if (err instanceof TimeoutError) {
          return throwError(() => new RequestTimeoutException());
        }
        return throwError(() => err);
      }),
    );
  };
};
```

Tras transcurrir 5 segundos, la solicitud será interrumpida en su procesamiento. Asimismo, es posible incorporar una lógica específica antes de generar una **RequestTimeoutException**, como, por ejemplo, la liberación de recursos.

*K. Decoradores personalizados [13]*

Nest se basa en un elemento del lenguaje denominado **decoradores**. Los decoradores representan un concepto ampliamente reconocido en numerosos lenguajes de programación, aunque en el ámbito de JavaScript, aún se consideran relativamente recientes. Para adquirir una comprensión más profunda de su funcionamiento, se sugiere la lectura de este artículo. A continuación, se proporciona una definición concisa:

> Un decorador ES2016 es una expresión que devuelve una función y puede tomar como argumentos un objetivo (target), un nombre y un descriptor de propiedad. Se aplica prefijándolo con el carácter @ y colocándolo en la parte superior de lo que está intentando decorar. Los decoradores se pueden definir para una clase, un método o una propiedad.

*1) Decoradores de parámetros*

Nest ofrece un conjunto de prácticos decoradores de parámetros que se pueden emplear en conjunto con los controladores de rutas HTTP. A continuación, se enumera una lista de los decoradores disponibles y los objetos básicos de Express (o Fastify) que representan [14]:

*Tabla 7 Lista de decoradores y su equivalencia para las plataformas HTTP*

| | |
|---|---|
| @Request(), @Req() | req |
| @Response(), @Res() | res |
| @Next() | next |
| @Session() | req.session |
| @Param(param?: string) | req.params / req.params[param] |
| @Body(param?: string) | req.body / req.body[param] |
| @Query(param?: string) | req.query / req.query[param] |
| @Headers(param?: string) | req.headers / req.headers[param] |
| @Ip() | req.ip |
| @HostParam() | req.hosts |

*Fuente: Autor*

Asimismo, es posible diseñar decoradores personalizados según las necesidades individuales. ¿Cuál es la utilidad de esto? En el entorno de Node.js, es una costumbre habitual agregar propiedades al objeto de solicitud y, posteriormente, extraerlas manualmente en cada controlador de ruta, recurriendo a código como el que se muestra a continuación:

```
const user = req.user;
```

Con el objetivo de mejorar la legibilidad y la claridad del código, es posible desarrollar un decorador llamado **@User()** y emplearlo de manera repetida en todos los controladores.

user.decorator.ts

```
import { createParamDecorator, ExecutionContext } from '@nestjs/common';

export const User = createParamDecorator(
  (data: unknown, ctx: ExecutionContext) => {
    const request = ctx.switchToHttp().getRequest();
    return request.user;
  },
);
```

Posteriormente, es factible utilizarlo de manera sencilla cuando se adapte a las necesidades específicas.

```
@Get()
async findOne(@User() user: UserEntity) {
  console.log(user);
}
```

*2) Transmisión de datos*

Cuando el decorador se ve influenciado por condiciones específicas, es posible emplear el parámetro **data** para proporcionar un argumento a la función de la fábrica del decorador. Un escenario típico de uso es la creación de un decorador personalizado encargado de extraer propiedades del objeto de solicitud según una clave determinada. Para ilustrar, si se considera una capa de autenticación que valida las solicitudes y añade una entidad de usuario al objeto de solicitud, la entidad de usuario en el contexto de una solicitud autenticada podría tener la siguiente estructura:

```
{
  "id": 101,
  "firstName": "Alan",
  "lastName": "Turing",
  "email": "alan@email.com",
  "roles": ["admin"]
}
```

Se propone la creación de un decorador que, al recibir un nombre de propiedad como referencia, entregará el valor correspondiente si está presente (en caso contrario, devolverá **undefined**). Esto será válido tanto si la propiedad no existe en el objeto de usuario como si dicho objeto aún no se ha creado.

user.decorator.ts

```
import { createParamDecorator, ExecutionContext } from '@nestjs/common';

export const User = createParamDecorator(
  (data: string, ctx: ExecutionContext) => {
    const request = ctx.switchToHttp().getRequest();
    const user = request.user;

    return data ? user?.[data] : user;
  },
);
```

De esta manera, se muestra cómo se puede acceder a una propiedad específica mediante el uso del decorador **@User()** dentro del controlador:

```
@Get()
async findOne(@User('firstName') firstName: string) {
  console.log(`Hello ${firstName}`);
}
```

Se puede usar este mismo decorador con diferentes claves para acceder a diferentes propiedades. Si el objeto del usuario es profundo o complejo, esto puede facilitar la implementación de los manejadores de solicitudes y hacer que el código sea más legible.

CONSEJO
Para aquellos que utilizan TypeScript, es importante destacar que **createParamDecorator<T>()** es un tipo genérico. Esto implica que se tiene la posibilidad de garantizar de manera explícita la seguridad de tipos, como, por ejemplo, utilizando **createParamDecorator<string>((data, ctx) => ...)**. Otra alternativa sería definir el tipo de parámetro directamente en la función de la fábrica, por ejemplo, **createParamDecorator((data: string, ctx) => ...)**. En caso de no especificar ninguno de los dos, el tipo de **data** se establecerá como **any**.

*3) Trabajando con pipes*

Nest aborda los decoradores de parámetros que han sido personalizados de la misma manera que los decoradores incorporados, tales como **@Body()**, **@Param()** y **@Query()**. Esto implica que los pipes también se ejecutarán para los parámetros que han sido anotados de forma personalizada (como se muestra en los ejemplos, con el argumento **user**). Además, existe la opción de aplicar el pipe directamente al decorador personalizado.

```
@Get()
async findOne(
  @User(new ValidationPipe({ validateCustomDecorators: true }))
  user: UserEntity,
) {
  console.log(user);
}
```

CONSEJO
Se debe tener en cuenta que la opción **validateCustomDecorators** debe establecerse en **true**. **ValidationPipe** no valida por defecto los argumentos anotados con los decoradores personalizados.

*4) Composición de decoradores*

Nest ofrece una utilidad que permite la composición de varios decoradores. Por ejemplo, si alguien desea unificar todos los decoradores relacionados con la autenticación en un solo decorador, esta tarea se puede lograr mediante la siguiente construcción:

auth.decorator.ts

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

export function Auth(...roles: Role[]) {
  return applyDecorators(
    SetMetadata('roles', roles),
    UseGuards(AuthGuard, RolesGuard),
    ApiBearerAuth(),
    ApiUnauthorizedResponse({ description: 'Unauthorized' }),
  );
}
```

El decorador personalizado **@Auth()** se puede emplear de la siguiente forma:

```
@Get('users')
@Auth('admin')
findAllUsers() {}
```

Esto tiene el efecto de aplicar los cuatro decoradores con una sola declaración.

ADVERTENCIA
El decorador **@ApiHideProperty()** del paquete **@nestjs/swagger** no es componible y no funcionará correctamente con la función **applyDecorators**.

## VI. MONGODB

MongoDB es una base de datos NoSQL orientada a documentos, de código abierto y escrita en C++. MongoDB provee una alta disponibilidad, un escalado automático, y un alto rendimiento. En las siguientes secciones se presentan las características de MongoDB.

### *A. MongoDB Características [15]*

*1) Modelo Documental*

Se ha diseñado MongoDB con la productividad y la flexibilidad de los desarrolladores en mente. Esta base de datos se orienta hacia documentos, lo que implica que los datos se almacenan en forma de documentos y estos documentos se agrupan en colecciones. El enfoque de documento resulta mucho más natural para que los desarrolladores trabajen con él, ya que los documentos son autosuficientes y pueden manejarse como objetos. De esta manera, los desarrolladores pueden enfocarse en los datos que desean almacenar y procesar, en lugar de preocuparse por cómo fragmentar los datos en diferentes tablas inflexibles.

Las colecciones son conjuntos donde se agrupan los documentos, pero no se requiere que los documentos dentro de una misma colección tengan un conjunto idéntico de campos. Esto se refiere a lo que se denomina un "esquema flexible". Esta adaptabilidad brinda a los desarrolladores la posibilidad de realizar iteraciones de manera más rápida y transferir datos entre esquemas distintos sin que se produzca un tiempo de inactividad. No obstante, si se desea establecer un esquema concreto en un momento específico, es posible lograrlo mediante la aplicación de reglas de validación en las colecciones.

En MongoDB, los documentos se guardan en formato BSON, que es una codificación en binario de JSON. Esto implica que los datos se almacenan en un formato binario, notablemente más rápido que JSON, y facilita el almacenamiento de información binaria, como imágenes, videos y otros datos similares. A pesar de que BSON es un formato binario codificado, se simplifica su manejo gracias al controlador específico de MongoDB disponible para el lenguaje de programación que se esté utilizando.

Para profundizar en el conocimiento del modelo de documentos, se recomienda revisar el artículo relacionado a bases de datos de documentos.

*2) Sharding – Fragmentación*

El proceso de "sharding" implica la división de conjuntos de datos extensos en varios fragmentos distribuidos, lo que permite una distribución y ejecución más eficiente de consultas que podrían ser complicadas y problemáticas en conjuntos de datos considerablemente grandes. Sin la implementación del "sharding", escalar una aplicación web en crecimiento con millones de usuarios diarios resultaría casi imposible.

La capacidad de escalabilidad horizontal proporcionada por el "sharding" en MongoDB se traduce en una mejora significativa. La escalabilidad horizontal implica que cada fragmento en un clúster contiene una porción del conjunto de datos, operando de manera independiente como una base de datos en sí misma. La unión de los datos de estos fragmentos distribuidos da lugar a una base de datos integral que está mucho mejor equipada para satisfacer las demandas de una aplicación popular en crecimiento sin experimentar interrupciones.

En un entorno de "sharding", se utiliza un componente de bajo consumo de recursos conocido como "mongos" para gestionar todas las operaciones del cliente. El "mongos" tiene la capacidad de redirigir consultas al fragmento apropiado, basándose en la clave de fragmentación. Además, una implementación adecuada de "sharding" desempeña un papel crucial en lograr un equilibrio de carga más efectivo.

Se recomienda consultar el artículo especializado sobre la técnica de "Sharding" en bases de datos, con el fin de obtener información adicional acerca de las diversas estructuras de fragmentación y los desafíos que abordan.

*3) Replicación*

Si los datos se encuentran exclusivamente en un solo servidor, se enfrentan a diversos posibles puntos de falla, como bloqueos del servidor, cortes de servicio o incluso fallos en el hardware. Cualquiera de estos incidentes podría resultar en una situación en la que acceder a los datos resultaría prácticamente imposible.

La replicación proporciona una solución para contrarrestar estas debilidades al implementar varios servidores de respaldo en situaciones de desastre y para realizar copias de seguridad. La escalabilidad horizontal mediante la utilización de múltiples servidores aumenta considerablemente la disponibilidad de datos, su confiabilidad y su capacidad de resistencia a fallos. En algunos casos, la replicación puede contribuir a distribuir la carga de lectura hacia los miembros secundarios del conjunto de réplicas mediante preferencias de lectura.

En MongoDB, se emplean conjuntos de réplicas con el propósito mencionado. En este sistema, un servidor o nodo principal se encarga de recibir y ejecutar todas las operaciones de escritura, transmitiendo esas mismas operaciones a los servidores secundarios para la replicación de los datos. Si el servidor principal llegara a sufrir una falla crítica, cualquiera de los servidores secundarios puede ser seleccionado para asumir el rol de nuevo nodo principal. Una vez que el antiguo nodo principal vuelve a estar en funcionamiento, lo hace desempeñando la función de servidor secundario para el nuevo nodo principal.

La plataforma de servicio de base de datos (DBaaS) MongoDB Atlas requiere, como mínimo, conjuntos de réplicas conformados por tres miembros. Estos conjuntos pueden extenderse a través de diversas regiones geográficas o incluso utilizar múltiples proveedores de servicios en la nube, según la preferencia del usuario.

Se recomienda consultar el artículo dedicado a la Replicación para obtener una comprensión más profunda de cómo se lleva a cabo el proceso de replicación en MongoDB.

*4) Autenticación*

La autenticación se convierte en un elemento de seguridad de gran importancia en MongoDB, ya que asegura que únicamente los usuarios con permisos adecuados puedan ingresar a la base de datos. La ausencia de autenticación permitiría que cualquier persona acceda a los datos sin restricciones.

MongoDB pone a disposición de los usuarios diversos métodos de autenticación para acceder a la base de datos, siendo el más usual el Mecanismo de Autenticación de Respuesta a Desafío con Sal (SCRAM), que se establece como opción por defecto. Al emplear SCRAM, se exige que el usuario suministre una base de datos de autenticación junto con un nombre de usuario y una contraseña.

Para acceder a detalles adicionales acerca de SCRAM y los demás métodos de autenticación que se encuentran disponibles, se recomienda consultar el artículo sobre la Autenticación en MongoDB.

*5) Desencadenadores de bases de datos*

En MongoDB Atlas, los desencadenadores de bases de datos representan una herramienta de gran potencia que facilita la ejecución de código en respuesta a eventos específicos que suceden en la base de datos. Por ejemplo, es factible emplear desencadenadores para ejecutar un guion cuando se lleva a cabo la inserción, actualización o eliminación de un documento. Además, estos desencadenadores se pueden programar para ejecutarse en momentos particulares.

MongoDB Atlas facilita la creación y administración de desencadenadores de una manera sencilla y de fácil comprensión. La interfaz de usuario de Atlas te permite tener control sobre los desencadenadores.

Los desencadenadores de bases de datos representan una valiosa herramienta para llevar a cabo auditorías, mantener la coherencia y la integridad de los datos, y ejecutar procesos de eventos más elaborados. Para obtener información detallada acerca de los distintos tipos de desencadenadores y su aplicación, se recomienda consultar el artículo específico sobre Desencadenadores de Bases de Datos.

*6) Datos de series temporales*

Con mayor frecuencia, los datos de series temporales provienen de dispositivos, como sensores, que registran información en el transcurso del tiempo. Estos datos se almacenan en una colección de documentos, en la que cada documento incluye una marca de tiempo y un valor correspondiente. MongoDB ofrece diversas funcionalidades diseñadas para facilitar la gestión de datos de series temporales.

Las colecciones nativas de series temporales en MongoDB han sido concebidas para optimizar la eficiencia en el almacenamiento y presentar un buen rendimiento en el contexto de secuencias de mediciones. Se proporcionan diversos parámetros que permiten gestionar el almacenamiento de datos de series temporales, como la granularidad (el lapso entre las mediciones) y el límite para la expiración de datos antiguos.

Para acceder a información adicional sobre las colecciones nativas de series temporales y otras funcionalidades de MongoDB que simplifican la manipulación de datos de series temporales, se sugiere revisar el artículo dedicado a Datos de Series Temporales.

*7) Consultas Ad-hoc*

Cuando se crea la estructura de una base de datos, no es factible anticipar todas las consultas que los usuarios finales llevarán a cabo. Una consulta "ad hoc" es una instrucción de breve duración cuyo resultado está vinculado a una variable particular. Cada vez que se ejecuta una consulta de este tipo, el resultado puede variar en función de las variables involucradas.

Mejorar la gestión de las consultas "ad hoc" puede tener un impacto significativo, especialmente en escenarios de gran escala donde se deben considerar una gran cantidad de variables. Esto explica por qué MongoDB, una base de datos orientada a documentos con un esquema adaptable se destaca como la elección preferida en la nube para aplicaciones empresariales que requieren análisis en tiempo real. Gracias a su compatibilidad con consultas "ad hoc" que permiten a los desarrolladores actualizar estas consultas en tiempo real, se pueden lograr mejoras sustanciales en el rendimiento.

MongoDB ofrece soporte para la consulta de campos, búsquedas geoespaciales y la búsqueda de expresiones regulares. Estas operaciones pueden recuperar información específica de los campos y considerar funciones personalizadas gracias a los índices, los documentos BSON y el Lenguaje de Consulta de MongoDB (MQL). Además, MongoDB brinda compatibilidad con la realización de agregaciones mediante el Marco de Agregación.

Para obtener una comprensión más profunda acerca de las funcionalidades de análisis de MongoDB, se recomienda revisar el artículo especializado en Análisis en Tiempo Real.

*8) Indexación*

El problema primordial que enfrentan con frecuencia los equipos de soporte técnico es la cuestión de la indexación. Cuando se implementa de manera adecuada, los índices están destinados a potenciar la velocidad y el rendimiento de las búsquedas. La falta de definir índices apropiados suele resultar en una serie de problemas relacionados con la accesibilidad, como dificultades en la ejecución de consultas y desequilibrios en la carga.

En ausencia de los índices apropiados, una base de datos se ve en la necesidad de revisar cada documento individualmente para identificar aquellos que se ajustan a la declaración de consulta. No obstante, cuando se dispone de un índice adecuado para cada consulta, el servidor puede ejecutar las solicitudes de los usuarios de manera eficiente. MongoDB pone a disposición una variada selección de índices y características que incluyen órdenes de clasificación específicas del idioma, lo que permite respaldar patrones de acceso complejos a los conjuntos de datos.

Cabe resaltar que los índices de MongoDB pueden ser generados según la necesidad para ajustarse a los patrones de consulta en tiempo real que experimentan cambios continuos, así como a las necesidades específicas de la aplicación. Además, tienen la flexibilidad de ser declarados en cualquier campo dentro de los documentos, incluso aquellos que se encuentran anidados en arreglos.

Se recomienda consultar el artículo titulado "Guía de Buenas Prácticas en Rendimiento: Indexación" con el fin de obtener una comprensión más detallada acerca de las distintas categorías de índices y su aplicación efectiva.

*9) Balanceo de carga*

Al final del día, el equilibrio de carga óptimo sigue siendo uno de los desafíos más importantes en la gestión de bases de datos a gran escala para aplicaciones empresariales en crecimiento. Distribuir adecuadamente millones de solicitudes de clientes entre cientos o miles de servidores puede marcar una diferencia perceptible (y muy apreciada) en el rendimiento.

Afortunadamente, a través de funcionalidades de escalabilidad horizontal como la replicación y la técnica de "sharding", MongoDB proporciona soporte para el equilibrio de carga a gran escala. La plataforma es capaz de gestionar numerosas solicitudes simultáneas de lectura y escritura para los mismos datos, empleando un sólido control de concurrencia y protocolos de bloqueo que garantizan la coherencia de la información. No es preciso incorporar un equilibrador de carga externo; MongoDB asegura que cada usuario disfrute de una vista coherente y una experiencia de alta calidad al acceder a los datos necesarios.

Para aquellos interesados en comprender el funcionamiento del equilibrio de carga en un conjunto fragmentado, se recomienda visitar la sección titulada "Equilibrador de Clúster Fragmentado" en la Documentación de MongoDB.

*B. Escenarios de uso*

*1) Inteligencia Artificial [16]*

El uso de MongoDB en la Inteligencia Artificial (IA) es esencial para aprovechar todo el potencial de los Modelos de Lenguaje a Gran Escala (LLMs) en el mundo empresarial. La llegada de ChatGPT en noviembre de 2022 marcó un hito en la aplicación de la IA generativa, demostrando su capacidad para automatizar una amplia variedad de tareas, desde la generación de texto de alta calidad hasta la predicción de movimientos en los mercados financieros.

Para aprovechar al máximo esta tecnología, las organizaciones necesitan entrenar y guiar a los LLMs con sus propios datos, lo que les proporciona un contexto único y valioso. Este enfoque, conocido como "Generación Mejorada con Recuperación" (RAG), implica utilizar datos propietarios y datos públicos más actualizados para dar a los LLMs la información necesaria para generar resultados confiables y precisos.

La transformación de datos en vectores es un paso clave en este proceso. Los vectores son representaciones numéricas multidimensionales de los datos que capturan patrones, relaciones y estructuras. Esto permite a las aplicaciones comprender las similitudes y relaciones entre diferentes conjuntos de datos, lo que antes era difícil de lograr con datos desestructurados. Una vez que los datos se han convertido en vectores, se almacenan e indexan en una base de datos de vectores, como MongoDB Atlas Vector Search.

La búsqueda de vectores mediante algoritmos de vecino más cercano aproximado (ANN) permite realizar búsquedas semánticas contextualmente conscientes, que pueden inferir significado y propósito a partir de los términos de búsqueda de los usuarios. Esto tiene un gran impacto en diversas aplicaciones de la IA, como el procesamiento de lenguaje natural (NLP), visión por computadora y generación de contenido.

En el caso de NLP, los LLMs pueden aprovechar el contexto proporcionado por los datos vectorizados para tareas como chatbots, preguntas y respuestas, resúmenes de texto y análisis de sentimientos. En el ámbito de la visión por computadora, se pueden utilizar para la clasificación de imágenes y la detección de objetos. Además, la generación de contenido se beneficia de esta tecnología al crear documentación basada en texto, páginas web optimizadas para SEO, código de programación e incluso la conversión de texto en imágenes o videos.

MongoDB juega un papel fundamental en este proceso al proporcionar una plataforma para la gestión de datos, búsqueda de vectores y recuperación de datos contextuales. Esto permite a las organizaciones aprovechar sus datos de manera eficiente para impulsar nuevas aplicaciones y experiencias, lo que a su vez mejora la eficiencia y la capacidad de respuesta en diversas áreas de la IA.

*2) Computación periférica – edge computing [17]*

MongoDB Atlas para el edge computing es una solución diseñada para simplificar la gestión de datos generados en diversos puntos del "edge", que incluyen dispositivos IoT, centros de datos locales y la nube. El edge computing, implica procesar datos más cerca de los usuarios finales, ofrece ventajas significativas, pero también presenta desafíos debido a la complejidad de la red, la gestión de volúmenes de datos y las preocupaciones de seguridad. Estos desafíos pueden ser costosos y difíciles de abordar.

MongoDB Atlas simplifica estas tareas manuales al permitir que MongoDB se ejecute en una variedad de infraestructuras edge, desde servidores gestionados de manera local, hasta implementaciones en la nube ofrecidas por proveedores importantes como AWS, Google Cloud y Microsoft Azure. Los datos fluyen de manera fluida y se mantienen sincronizados en todas las fuentes, garantizando la entrega de datos en tiempo real con una latencia mínima.

Con MongoDB Atlas, las organizaciones pueden utilizar una interfaz unificada para ofrecer una experiencia de desarrollo coherente desde el edge hasta la nube y todo lo que hay entre ellos. Esto reduce significativamente la complejidad de construir aplicaciones y arquitecturas en el edge.

Algunas de las capacidades clave de MongoDB Atlas para el Edge incluyen:

1. Ejecutar MongoDB en diversas infraestructuras de edge para alta confiabilidad y baja latencia: Posibilidad de ejecutar aplicaciones en MongoDB en una amplia variedad de infraestructuras, desde servidores autoadministrados en ubicaciones remotas hasta infraestructuras gestionadas por proveedores de nube como AWS, Google Cloud y Microsoft Azure.

2. Ejecutar aplicaciones en ubicaciones con conectividad de red intermitente: Con el uso de Atlas Edge Server y Atlas Device Sync para proporcionar una capa de sincronización de datos local para aplicaciones que se ejecutan en quioscos o dispositivos móviles e IoT. Esto mejora las experiencias de aplicaciones sin conexión y previene la pérdida de datos.
3. Construir y desplegar aplicaciones de edge computing potenciadas por IA: Posibilidad de utilizar los datos almacenados en MongoDB Atlas para potenciar aplicaciones de edge computing con funcionalidad de IA, incluso en situaciones de red no disponible. Esto incluye la generación de incrustaciones y búsqueda de vectores con Atlas Vector Search.
4. Almacenar y procesar datos en tiempo real y por lotes de dispositivos IoT para convertirlos en acciones: Utilización de MongoDB Atlas Stream Processing para ingerir y procesar datos de alto volumen y alta velocidad de millones de dispositivos IoT. Estos datos se pueden utilizar en casos de uso como mantenimiento predictivo y detección de anomalías.
5. Garantizar la seguridad de las aplicaciones de edge para la privacidad de datos y el cumplimiento: MongoDB Atlas ofrece capacidades de seguridad integradas, incluyendo cifrado de datos en reposo, en dispositivos y en tránsito, así como gestión de acceso basada en roles.

Algunas organizaciones líderes ya están aprovechando MongoDB Atlas para el Edge en sus operaciones. Como ejemplo se incluyen Cathay Pacific, que digitalizó su proceso de operaciones de vuelo, y Cloneable, que utiliza la tecnología para implementar aplicaciones de IA en dispositivos diversos.

*3) Pagos [18]*

La industria de pagos está experimentando una disrupción masiva, impulsada por las crecientes expectativas de los consumidores y los cambios en las regulaciones. Los consumidores esperan un proceso de pago rápido y sencillo, así como acceso inmediato a la información de sus cuentas y servicios. Mientras tanto, los comerciantes demandan liquidaciones rápidas y servicios de valor agregado, incluyendo información sobre los clientes.

Las regulaciones sobre privacidad de datos, gobernanza de datos y los tiempos de respuesta de pagos están agregando volatilidad a un entorno ya incierto. Además, nuevos actores, incluyendo grandes tecnológicas como Apple, Facebook y Amazon, se están ingresando a la industria de pagos y compiten con ventajas tecnológicas y recursos financieros significativos.

Las organizaciones de la industria de pagos a menudo se ven limitadas por tecnologías heredadas que no han evolucionado en décadas y no pueden satisfacer las expectativas cambiantes de los consumidores y las regulaciones.

Para mantenerse competitivas, muchas empresas en la industria de pagos necesitan actualizar su arquitectura de datos, en particular sus bases de datos. Una base de datos moderna y robusta es esencial para habilitar nuevas aplicaciones y servicios que agreguen valor al negocio. La base de datos adecuada para la industria de pagos debe ser lo suficientemente flexible para adaptarse a los requisitos técnicos cambiantes y ofrecer un rendimiento y disponibilidad sólidos. También debe permitir la generación de nuevas ideas sobre los clientes que ayuden a desarrollar servicios diferenciadores. Para competir eficazmente en este entorno, las empresas deben superar las limitaciones de sus tecnologías heredadas, adoptar una arquitectura de datos moderna y aprovechar bases de datos como MongoDB para impulsar la innovación y cumplir con las demandas cambiantes de la industria.

*4) Vista Simple [19]*

Las aplicaciones de vista simple buscan crear una vista única de cualquier conjunto de datos, integrando información dispersa de múltiples fuentes en un repositorio central. Esto es esencial en diversas industrias, como servicios financieros, gobierno y venta al por menor, donde se requiere una visión completa y en tiempo real de los datos para la toma de decisiones.

El desafío en la construcción de aplicaciones de vista simple radica en que los datos suelen estar en sistemas aislados que no se comunican entre sí, y tienen diferentes estructuras y tipos. MongoDB facilita esta tarea al permitir la incorporación de cualquier tipo de datos sin necesidad de definir un esquema rígido. Utiliza un modelo de documentos en formato JSON que admite diversos tipos de datos (números, cadenas de texto, datos binarios, matrices) y no requiere un esquema estricto.

La flexibilidad de MongoDB se extiende a los esquemas dinámicos, lo que significa que puede iterar sobre el esquema sin tener que replantearlo por completo. Esto es esencial cuando se deben incorporar nuevos tipos de

datos o cambiar la estructura de los documentos. Además, los documentos de MongoDB pueden variar en su estructura, lo que permite manejar datos heterogéneos.

La capacidad de consulta expresiva, la indexación y las capacidades de agregación de MongoDB hacen posible buscar y filtrar los datos de acuerdo con las necesidades comerciales, sin importar cómo se deba acceder a ellos. En resumen, MongoDB simplifica la construcción de aplicaciones de vista simple al ofrecer una solución flexible y escalable para integrar datos de múltiples fuentes y crear una vista única de cualquier conjunto de información.

*5) Personalización [20]*

La personalización es esencial para brindar experiencias únicas a los clientes en tiempo real. MongoDB permite crear motores de personalización que se adaptan a las experiencias en línea para los clientes en función de perfiles de comportamiento y demográficos, interacciones históricas y preferencias. Estos motores reemplazan o se integran con sistemas heredados de gestión de datos de clientes y tienen aplicaciones en diversas industrias, como servicios financieros, gobierno, alta tecnología y ventas al por menor.

Los desafíos que enfrentan otras bases de datos en este contexto incluyen esquemas rígidos que dificultan la incorporación de nuevos atributos y características, problemas de escalabilidad al analizar el comportamiento del cliente en tiempo real y largos tiempos de implementación debido a la necesidad de mantener datos redundantes en múltiples sistemas.

MongoDB aborda estos desafíos al ofrecer una solución que permite almacenar cualquier tipo de datos, realizar análisis en tiempo real y cambiar el esquema de manera incremental.

- Esquemas Dinámicos: El modelo de documentos JSON de MongoDB facilita el almacenamiento de cualquier tipo de dato, incluidas estructuras de datos complejas y datos geoespaciales. Además, permite modificar el esquema sin afectar el rendimiento o requerir que la base de datos se detenga.
- Análisis en Tiempo Real: MongoDB proporciona un lenguaje de consulta expresivo, índices secundarios, geoespaciales y de texto, un marco de agregación y MapReduce que permiten ejecutar análisis rápidos en datos con múltiples estructuras dentro de la base de datos. Esto permite una personalización rápida y escalable.
- Menor Costo y Complejidad: Los esquemas dinámicos y el modelo de documentos flexibles de MongoDB facilitan la integración de datos de clientes diversificados y su uso en múltiples aplicaciones sin necesidad de mantener sistemas de identidad separados para cada una. Los documentos de MongoDB pueden variar en estructura, lo que simplifica la incorporación de datos nuevos según sea necesario.

*6) Catálogos [21]*

La gestión de catálogos es esencial para empresas que desean crear aplicaciones móviles exitosas que escalen para millones de usuarios. Los clientes esperan una experiencia constante en sus dispositivos móviles, pero las bases de datos relacionales tradicionales pueden dificultar la iteración constante y no están diseñadas para soportar grandes volúmenes de usuarios.

Un catálogo en este contexto almacena listas de SKU, operaciones F/X, equipos y cualquier activo o entidad, junto con los metadatos asociados. Esto va más allá de los catálogos de pedidos por correo o los catálogos de productos en sitios web de comercio electrónico. Ejemplos de aplicaciones incluyen la gestión de operaciones financieras, activos gubernamentales, metadatos para entidades en la industria de alta tecnología y la gestión de catálogos de productos en el sector minorista.

Los desafíos que enfrentan otras bases de datos incluyen esquemas rígidos que dificultan la incorporación de nuevos elementos y atributos, datos heterogéneos que provienen de diferentes tipos de activos y compromisos en las funcionalidades, como búsqueda, navegación y análisis de datos. MongoDB resuelve estos desafíos al ofrecer una solución que permite almacenar cualquier tipo de dato, recuperarlos y modificar el esquema de manera incremental.

- Documentos: El modelo de documentos JSON de MongoDB permite almacenar diferentes activos con atributos diversos en un solo lugar, lo que facilita la representación de relaciones complejas y jerárquicas.
- Esquemas Dinámicos: Los esquemas en MongoDB son autosuficientes y permiten agregar nuevos productos y características sin afectar el rendimiento ni detener la base de datos.

- Lenguaje de Consulta de MongoDB: MongoDB ofrece un lenguaje de consulta expresivo, indexación, búsqueda de texto y geoespacial, así como análisis, lo que brinda acceso flexible a los datos, independientemente de cómo se necesite encontrarlos.

*7) Gestión de contenido [22]*

La gestión de contenidos ha evolucionado desde la época en la que el contenido web estaba compuesto principalmente por texto estático. Hoy en día, se requiere una variedad abrumadora de contenido, que incluye texto, audio, video, imágenes y redes sociales, para captar la atención de los usuarios. Sin embargo, agregar nuevo contenido o características a una base de datos relacional existente puede ser complicado y costoso, ya que a menudo afecta el rendimiento o requiere que la base de datos esté fuera de línea.

Los sistemas de gestión de contenidos almacenan y sirven activos de información y metadatos asociados para diversas aplicaciones, como sitios web, publicaciones en línea y archivos. Ejemplos de aplicaciones que incluyen la gestión de investigaciones de equidad en servicios financieros, la publicación de archivos gubernamentales en línea, la consolidación de servicios y activos multimedia en tecnología de alta gama y la creación de listados de productos en el sector minorista que incluyen videos y demostraciones en vivo.

Los desafíos que enfrentan otras bases de datos incluyen esquemas rígidos que dificultan la incorporación de nuevos tipos de contenido y funciones, el crecimiento masivo que puede atraer millones de usuarios y la complejidad de usar múltiples bases de datos y sistemas de archivos.

MongoDB resuelve estos desafíos al ofrecer una solución que permite almacenar y servir cualquier tipo de contenido, construir nuevas características e incorporar cualquier tipo de dato en una sola base de datos.

- Nuevos tipos de datos: El modelo de documentos JSON de MongoDB y su rico lenguaje de consulta facilitan el almacenamiento y la búsqueda de diferentes tipos de contenido con atributos diversos en un solo lugar, lo que también permite representar relaciones complejas y jerárquicas.
- Audiencia global: MongoDB permite la distribución automática de datos a través de servidores de bajo costo dentro y entre centros de datos, lo que lo hace adecuado para admitir una amplia audiencia global, con la capacidad de escalar a miles de nodos y petabytes de datos sin necesidad de capas de partición y caché personalizadas.
- Arquitectura simplificada: MongoDB GridFS permite traer activos de contenido directamente a la base de datos, eliminando dependencias de sistemas de archivos y capas de caché separadas, y permitiendo un modelo de seguridad, alta disponibilidad y escalabilidad similar al del resto de los datos.

*8) Modernización del Mainframe [23]*

La Modernización de Mainframes es un proceso fundamental para las organizaciones que buscan adaptarse a la era digital y aprovechar las ventajas tecnológicas emergentes. Un mainframe es una computadora de gran escala y de propósito general que ha sido un pilar en las empresas durante décadas. A pesar de su confiabilidad y robustez, los mainframes a menudo presentan desafíos en términos de escalabilidad, agilidad y costos operativos, lo que dificulta la implementación de nuevas iniciativas digitales y el cumplimiento de las crecientes demandas de los clientes.

Los principales motivos para considerar la modernización de mainframes son los siguientes:

1. **Reducción de costos operativos**: Los mainframes pueden ser costosos de operar, especialmente cuando se trata de gastos de energía y licencias de software. Al migrar tareas de procesamiento a plataformas más económicas como MongoDB, las organizaciones pueden reducir drásticamente sus costos operativos, liberando recursos financieros para otras iniciativas.
2. **Soporte para nuevas iniciativas digitales**: La modernización de mainframes permite a las organizaciones habilitar nuevos canales digitales, como aplicaciones móviles y servicios en línea, para interactuar con los clientes. Esto impulsa la innovación y permite una respuesta más rápida a las tendencias del mercado.
3. **Mejora de la experiencia del cliente**: Al proporcionar un acceso más rápido y fácil a los datos y servicios, las organizaciones pueden mejorar significativamente la experiencia del cliente. Esto incluye la capacidad de ofrecer respuestas en tiempo real, personalización y una vista de 360 grados del cliente.
4. **Escalabilidad ágil**: MongoDB permite una escalabilidad sencilla y asequible. A medida que las demandas aumentan, las organizaciones pueden agregar capacidad sin necesidad de adquirir costosos mainframes adicionales. Esto permite una mayor agilidad empresarial y la capacidad de manejar picos de tráfico inesperados.

5. **Flexibilidad en desarrollo y despliegue de aplicaciones**: Las bases de datos tradicionales utilizadas en mainframes a menudo tienen esquemas rígidos que dificultan la adaptabilidad a las necesidades cambiantes. MongoDB, con su esquema dinámico, permite a los desarrolladores adaptar rápidamente las aplicaciones a nuevas necesidades sin tiempos de inactividad ni afectar el rendimiento.
6. **Menor riesgo y tiempo de comercialización más rápido**: La modernización de mainframes a plataformas como MongoDB puede realizarse con menos riesgo y en un tiempo más corto en comparación con las migraciones completas. Esto significa que las organizaciones pueden lanzar nuevas aplicaciones y servicios más rápido y con menos interrupciones.

*9) Gaming[24]*

MongoDB es una elección ideal para la industria de los videojuegos debido a su flexibilidad y capacidad para abordar los desafíos específicos de este sector. Los videojuegos modernos requieren un enfoque de bases de datos que se adapte a las necesidades de desarrollo rápido, alcance global, escala masiva y disponibilidad constante. MongoDB aborda estos aspectos de la siguiente manera:

1. **Desarrollo Rápido**: Los videojuegos evolucionan constantemente con nuevas características y contenido. MongoDB permite a los desarrolladores cambiar la estructura de los datos tan rápido como cambian el juego. Su esquema flexible y almacenamiento de datos en formato JSON facilita la adaptación a las necesidades cambiantes.
2. **Alcance Global**: Los jugadores se encuentran en todo el mundo, y la baja latencia es crucial. MongoDB Atlas es un servicio de base de datos gestionado que se distribuye globalmente en múltiples regiones, lo que garantiza una baja latencia para los usuarios. Además, ofrece redundancia geográfica para mantener la alta disponibilidad de los datos.
3. **Escala Masiva**: Los videojuegos exitosos a menudo experimentan un crecimiento inesperado en la base de usuarios. MongoDB permite escalar vertical y horizontalmente, desde configuraciones gratuitas hasta clústeres que pueden manejar las cargas de trabajo más exigentes.
4. **Siempre en Línea**: Las interrupciones y retrasos son inaceptables en el mundo de los videojuegos. MongoDB ofrece una base de datos resiliente que se adapta a las demandas de disponibilidad constante de los juegos y sus jugadores.

*C. Operaciones CRUD [25]*

Sin importar la razón por la que se esté utilizando un servidor de MongoDB, es necesario realizar operaciones CRUD en él. Los métodos básicos para interactuar con un servidor MongoDB se llaman operaciones CRUD. CRUD significa Crear, Leer, Actualizar y Eliminar. Estos métodos CRUD son las principales formas en que se gestionan los datos en las bases de datos.

En esta sección, se cubre la definición de CRUD. Luego se examina cómo ejecutar operaciones CRUD en MongoDB utilizando el Lenguaje de Consulta de MongoDB (MQL).

En esta guía instructiva, se aborda:

- La naturaleza de CRUD en MongoDB.
- La ejecución de operaciones de creación.
- La realización de operaciones de lectura.
- El procedimiento para llevar a cabo operaciones de actualización.
- La implementación de operaciones de eliminación.

Previo a adentrarse en la manipulación de datos mediante operaciones CRUD en MongoDB, se reserva un momento para establecer los cimientos y definir el concepto de CRUD.

*1) ¿Qué es CRUD en MongoDB?*

Las operaciones CRUD delinean las convenciones de una interfaz de usuario que posibilita a los usuarios visualizar, buscar y modificar segmentos de la base de datos.

Para modificar documentos en MongoDB, se realiza una conexión al servidor, se consultan los documentos pertinentes y luego se ajustan las propiedades de configuración antes de remitir los datos nuevamente a la base de datos para su actualización. CRUD se centra en los datos y se adhiere a los verbos de acción HTTP para su estandarización.

En lo que respecta a las operaciones individuales de CRUD:
- La operación de Crear se emplea para introducir nuevos documentos en la base de datos de MongoDB.
- La operación de Leer se utiliza para consultar un documento en la base de datos.
- La operación de Actualizar se emplea para modificar documentos existentes en la base de datos.
- La operación de Eliminar se utiliza para suprimir documentos en la base de datos.

### *2) Cómo realizar operaciones CRUD*

Una vez que las operaciones CRUD en MongoDB han sido delineadas, se tiene la oportunidad de examinar la ejecución de cada operación individual y la manipulación de documentos en una base de datos de MongoDB. Se va a explorar los procesos de creación, lectura, actualización y eliminación de documentos, abordando cada operación de manera secuencial.

### *3) Operaciones para crear*

En el contexto de las operaciones CRUD en MongoDB, cuando se ejecuta la operación de creación y la colección especificada no existe, se creará la colección. Es importante destacar que las operaciones de creación en MongoDB se enfocan en una sola colección y no en múltiples colecciones. Además, las operaciones de inserción en MongoDB son atómicas a nivel de un solo documento.

MongoDB ofrece dos operaciones de creación distintas que permiten insertar documentos en una colección:

- db.collection.insertOne()
- db.collection.insertMany()

#### *a) insertOne()*

Tal como sugiere su denominación, **insertOne()** posibilita la inserción de un solo documento en la colección. En este caso específico, se trabajará con una colección denominada **RecordsDB**. La inserción de un único registro en la colección se lleva a cabo mediante la invocación del método **insertOne()** en **RecordsDB**. A continuación, se suministra la información que se desea insertar, presentada en forma de pares clave-valor, siguiendo el esquema establecido.

```
db.RecordsDB.insertOne({
    name: "Marsh",
    age: "6 years",
    species: "Dog",
    ownerAddress: "380 W. Fir Ave",
    chipped: true
})
```

En caso de que la operación de creación sea exitosa, se genera un nuevo documento. La función proporcionará un objeto en el cual **acknowledged** tiene el valor **true** y **insertID** corresponde al nuevo **ObjectId** creado.

```
> db.RecordsDB.insertOne({
... name: "Marsh",
... age: "6 years",
... species: "Dog",
... ownerAddress: "380 W. Fir Ave",
... chipped: true
... })
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5fd989674e6b9ceb8665c57d")
}
```

*b) insertMany()*

La inserción de múltiples elementos simultáneamente se logra al invocar el método **insertMany()** en la colección seleccionada. En esta situación, se transmiten varios elementos a la colección designada (**RecordsDB**), separándolos por comas. Dentro de los paréntesis, se emplean corchetes para indicar que se está enviando una lista de múltiples entradas, un enfoque comúnmente denominado como método anidado.

```
db.RecordsDB.insertMany([{
    name: "Marsh",
    age: "6 years",
    species: "Dog",
    ownerAddress: "380 W. Fir Ave",
    chipped: true},
    {name: "Kitana",
    age: "4 years",
    species: "Cat",
    ownerAddress: "521 E. Cortland",
    chipped: true}])
```

```
db.RecordsDB.insertMany([{ name: "Marsh", age: "6 years", species: "Dog",
ownerAddress: "380 W. Fir Ave", chipped: true}, {name: "Kitana", age: "4 years",
species: "Cat", ownerAddress: "521 E. Cortland", chipped: true}])
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("5fd98ea9ce6e8850d88270b4"),
                ObjectId("5fd98ea9ce6e8850d88270b5")
        ]
}
```

*4) Operaciones de lectura*

Las operaciones de lectura posibilitan la aplicación de filtros y criterios de consulta particulares para especificar qué documentos se desean obtener. La documentación de MongoDB ofrece detalles adicionales sobre los filtros de consulta disponibles. Asimismo, es posible utilizar modificadores de consulta para alterar la cantidad de resultados devueltos.

MongoDB dispone de dos métodos para la lectura de documentos de una colección:

- db.collection.find()
- db.collection.findOne()

*a) find()*

Para recuperar todos los documentos de una colección, basta con emplear el método **find()** en la colección pertinente. Al ejecutar únicamente el método **find()** sin argumentos, se obtendrán todos los registros presentes en la colección en ese momento.

```
db.RecordsDB.find()
```

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years",
"species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "3 years", "species"
: "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "8 years",
"species" : "Dog", "ownerAddress" : "900 W. Wood Way", "chipped" : true }
```

En este contexto, se observa que cada registro cuenta con un **ObjectId** asociado, el cual está mapeado a la clave **_id**.

Si se busca mayor especificidad en una operación de lectura para encontrar una subsección particular de los registros, es posible utilizar los criterios de filtrado mencionados previamente para indicar qué resultados se deben recuperar. Una de las estrategias más habituales para filtrar los resultados es realizar la búsqueda por valor.

```
db.RecordsDB.find({"species":"Cat"})
```

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

*b) findOne()*

Para adquirir un documento que cumpla con los criterios de búsqueda, basta con utilizar el método **findOne()** en la colección específica. En caso de que varios documentos satisfagan la consulta, este método devuelve el primer documento según el orden natural que refleja la disposición de los documentos en el disco. Si ningún documento cumple con los criterios de búsqueda, la función devuelve **null**. La sintaxis de la función es la siguiente.

```
db.{collection}.findOne({query}, {projection})
```

Consideremos, a modo de ejemplo, la colección denominada RecordsDB.

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "8 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years", "species"
: "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "3 years", "species"
: "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "8 years",
"species" : "Dog", "ownerAddress" : "900 W. Wood Way", "chipped" : true }
```

Y se realiza la ejecución de la siguiente línea de código:

```
db.RecordsDB.find({"age":"8 years"})
```

El resultado obtenido sería el siguiente:

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "8 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

Nótese que, a pesar de que dos documentos cumplen con los criterios de búsqueda, únicamente se devuelve el primer documento que satisface la condición de búsqueda.

*5) Operaciones de actualización*

De manera similar a las operaciones de creación, las operaciones de actualización se aplican a una sola colección y tienen un alcance atómico a nivel de un solo documento. Una operación de actualización utiliza filtros y criterios para seleccionar los documentos que se desean actualizar.

Es fundamental tener precaución al llevar a cabo actualizaciones de documentos, dado que estas modificaciones son permanentes y no pueden revertirse. Esta consideración también es válida para las operaciones de eliminación.

En el marco de las operaciones CRUD en MongoDB, existen tres métodos distintos para la actualización de documentos:

- db.collection.updateOne()
- db.collection.updateMany()
- db.collection.replaceOne()

*a) updateOne()*

Es factible modificar un registro existente mediante una operación de actualización, cambiando un solo documento en el proceso. Para llevar a cabo esta tarea, se emplea el método **updateOne()** en una colección específica, en este caso, **RecordsDB**. Al actualizar un documento, se suministran dos argumentos al método: un filtro de actualización y una acción de actualización.

El filtro de actualización establece qué elementos se desean actualizar, mientras que la acción de actualización indica cómo realizar dicha actualización. En primer lugar, se proporciona el filtro de actualización. Posteriormente, se utiliza la clave **$set** para indicar los campos que se desean actualizar, presentándolos como un valor. Este método actualizará el primer registro que cumpla con el filtro suministrado.

```
db.RecordsDB.updateOne({name: "Marsh"}, {$set:{ownerAddress: "451 W. Coffee St. A204"}})
```

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

```
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years",
"species" : "Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
```

*b) updateMany()*

La función **updateMany()** posibilita la actualización de varios elementos al transmitir una lista de elementos, de manera análoga a la inserción de múltiples elementos. Esta operación de actualización sigue la misma sintaxis que la actualización de un solo documento.

```
db.RecordsDB.updateMany({species:"Dog"}, {$set: {age: "5"}})
```

```
{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }
```

```
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species" :
"Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" :
"Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "5", "species"
: "Dog", "ownerAddress" : "900 W. Wood Way", "chipped" : true }
```

*c) replaceOne()*

El uso del método **replaceOne()** se destina a la sustitución de un solo documento en la colección señalada. **replaceOne()** reemplaza la totalidad del documento, implicando que los campos presentes en el documento anterior pero ausentes en el nuevo se perderán en el proceso.

```
db.RecordsDB.replaceOne({name: "Kevin"}, {name: "Maki"})
```

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

```
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species" :
"Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" :
"Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Maki" }
```

*6) Operaciones de eliminación*

Las operaciones de eliminación se aplican a una sola colección, al igual que las operaciones de actualización y creación. Estas operaciones también poseen un alcance atómico a nivel de un solo documento. Al realizar operaciones de eliminación, es posible suministrar filtros y criterios para especificar qué documentos se desean eliminar de la colección. Las opciones de filtro se rigen por la misma sintaxis utilizada en las operaciones de lectura.

En MongoDB, existen dos métodos diferentes para eliminar registros de una colección

- db.collection.deleteOne()
- db.collection.deleteMany()

*a) deleteOne()*

El método **deleteOne()** se emplea para suprimir un documento de una colección determinada en el servidor de MongoDB. Se utiliza un criterio de filtro para precisar el elemento que se eliminará, eliminando el primer registro que concuerde con el filtro suministrado.

```
db.RecordsDB.deleteOne({name:"Maki"})
```

```
{ "acknowledged" : true, "deletedCount" : 1 }
```

```
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species" :
"Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" :
"Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
```

*b) deleteMany()*

**deleteMany()** se emplea para la eliminación de múltiples documentos dentro de una colección específica mediante una única operación de eliminación. El método recibe una lista como argumento, y cada elemento de la lista se describe con criterios de filtro, de manera similar a cómo se hace en **deleteOne()**.

```
db.RecordsDB.deleteMany({species:"Dog"})
```

```
{ "acknowledged" : true, "deletedCount" : 2 }
```

```
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

*D. Mongoose [26]*

*1) ¿Qué es Mongoose?*

Aquellos que se adentran en el aprendizaje de MongoDB a menudo lo hacen a través de la reconocida biblioteca Mongoose. Esta última es definida como un "elegante modelado de objetos MongoDB diseñado para Node.js".

La biblioteca Mongoose, siendo un ODM (Object Data Modeling) para MongoDB, no es indispensable para obtener una experiencia destacada con MongoDB, aunque hay desarrolladores que la favorecen. Entre los

desarrolladores de Node.js, es común la elección de Mongoose para facilitar tareas como el modelado de datos, la aplicación de esquemas, la validación de modelos y la manipulación general de datos.

*2) ¿Por qué Mongoose?*

De manera predeterminada, MongoDB presenta un modelo de datos adaptable, lo que facilita la modificación y actualización de bases de datos en el futuro. Sin embargo, dado que muchos desarrolladores están familiarizados con esquemas más estrictos, Mongoose introduce un enfoque semirrígido. Al utilizar Mongoose, los desarrolladores se ven obligados a definir tanto un esquema (Schema) como un modelo (Model) desde el inicio.

*3) ¿Qué es un esquema?*

La estructura de los documentos en una colección se define mediante un esquema. En el caso de Mongoose, este esquema se asigna directamente a una colección específica en MongoDB.

```
const blog = new Schema({
  title: String,
  slug: String,
  published: Boolean,
  author: String,
  content: String,
  tags: [String],
  createdAt: Date,
  updatedAt: Date,
  comments: [{
    user: String,
    content: String,
    votes: Number
  }]
});
```

A través de los esquemas, se realiza la definición de cada campo junto con su tipo de datos correspondiente. Los tipos de datos permitidos son:

- String
- Number
- Date
- Buffer
- Boolean
- Mixed
- ObjectId
- Array
- Decimal128
- Map

*4) ¿Qué es un modelo?*

Los modelos aplican el esquema a cada documento dentro de su colección respectiva. Están encargados de gestionar todas las operaciones relacionadas con los documentos, como la creación, lectura, actualización y eliminación (CRUD).

AVISO
Es relevante destacar que el primer argumento proporcionado al modelo debe corresponder a la forma singular del nombre de la colección. Mongoose realiza automáticamente la transformación a la forma plural, convierte todo a minúsculas y emplea ese resultado como el nombre de la colección en la base de datos.

```
const Blog = mongoose.model('Blog', blog);
```

En este ejemplo, Blog se traduce a la colección **blogs**.

*5) Configuración del entorno*

Se procederá a la configuración del entorno. Se parte de la suposición de que el usuario ya cuenta con la instalación de Node.js. A continuación, se ejecutarán los siguientes comandos desde la terminal del sistema operativo para iniciar el proceso:

```
mkdir mongodb-mongoose
cd mongodb-mongoose
npm init -y
npm i mongoose
npm i -D nodemon
code .
```

Se generará el directorio del proyecto, realizará la inicialización, instalará los paquetes necesarios y abrirá el proyecto en VS Code. Se incorporará un script al archivo **package.json** para la ejecución del proyecto. Además, se optará por el uso de módulos ES en lugar de CommonJS, y se incluirá la especificación del tipo de módulo. Esta elección posibilitará el uso del **await** a nivel superior.

```
...
  "scripts": {
    "dev": "nodemon index.js"
  },
  "type": "module",
...
```

*6) Conexión a MongoDB*

A continuación, se procederá a la creación del archivo **index.js**, donde se empleará Mongoose para establecer la conexión con MongoDB.

```
import mongoose from 'mongoose'

mongoose.connect("mongodb+srv://<username>:<password>@cluster0.eyhty.mongodb.net/myFirstDatabase?retryWrites=true&w=majority")
```

Se tiene la opción de conectarse a una instancia local de MongoDB, pero en este escrito se optará por utilizar un clúster gratuito de MongoDB Atlas. En caso de no contar con una cuenta, el proceso de registro para obtener un clúster gratuito de MongoDB Atlas se encuentra disponible aquí.
Si aún no ha configurado su clúster, se recomienda seguir la guía proporcionada para la creación del mismo.
Una vez que haya creado su clúster, será necesario sustituir la cadena de conexión anterior por la suministrada por su clúster, asegurándose de incluir el nombre de usuario y contraseña.

AVISO
La cadena de conexión obtenida del panel de MongoDB Atlas estará vinculada a la base de datos denominada myFirstDatabase. Es necesario modificar este valor según la preferencia del usuario para asignar el nombre deseado a su propia base de datos.

*7) Crear un esquema y un modelo*

Antes de realizar cualquier acción con la conexión, es necesario que el usuario cree tanto un esquema como un modelo. Se recomienda, en lo posible, generar un archivo específico para el esquema y para el modelo. Por lo tanto, se procederá a crear una nueva estructura de carpetas y archivos bajo la ruta **model/Blog.js**.

```
import mongoose from 'mongoose';
const { Schema, model } = mongoose;

const blogSchema = new Schema({
  title: String,
  slug: String,
  published: Boolean,
  author: String,
  content: String,
  tags: [String],
  createdAt: Date,
  updatedAt: Date,
  comments: [{
    user: String,
    content: String,
    votes: Number
  }]
});

const Blog = model('Blog', blogSchema);
export default Blog;
```

*8) Inserción de datos // método 1*

Ahora que el primer modelo y esquema han sido configurados, es posible iniciar el proceso de inserción de datos en la base. Regresando al archivo **index.js**, se procederá a agregar un nuevo artículo al blog.

```
import mongoose from 'mongoose';
import Blog from './model/Blog.js';

mongoose.connect("mongodb+srv://mongo:mongo@cluster0.eyhty.mongodb.net/myFirstDatabase?retry
Writes=true&w=majority")

// Create a new blog post object
const article = new Blog({
  title: 'Awesome Post!',
  slug: 'awesome-post',
  published: true,
  content: 'This is the best post ever',
  tags: ['featured', 'announcement'],
});

// Insert the article in our MongoDB database
await article.save();
```

En un primer paso, se requiere importar el modelo Blog previamente creado. Posteriormente, se genera una nueva instancia de objeto de blog, y a continuación, se emplea el método **save()** para llevar a cabo su inserción en la base de datos de MongoDB. Después de esta operación, se añadirá información adicional para registrar el contenido actual de la base de datos. Esta acción se realizará mediante el uso del método **findOne()**.

```
// Find a single blog post
const firstArticle = await Blog.findOne({});
console.log(firstArticle);
```

Es momento de poner en marcha el código:

```
npm run dev
```

El documento que se ha insertado debería visualizarse en la terminal.

AVISO
Dado que se emplea nodemon en este proyecto, al guardar un archivo, el código se ejecutará nuevamente. Si se desea insertar múltiples documentos, basta con continuar guardando.

*9) Inserción de datos // método 2*

En la instancia previa, se empleó el método **save()** de Mongoose para insertar el documento en la base de datos, lo cual involucra dos pasos: la creación de la instancia del objeto y su posterior guardado. Como alternativa, es posible realizar ambas acciones en una sola etapa mediante el uso del método **create()** de Mongoose.

```
// Create a new blog post and insert into database
const article = await Blog.create({
  title: 'Awesome Post!',
  slug: 'awesome-post',
  published: true,
  content: 'This is the best post ever',
  tags: ['featured', 'announcement'],
});

console.log(article);
```

Este enfoque resulta considerablemente más efectivo, ya que no solo permite la inserción del documento, sino que también proporciona la devolución del documento junto con su **_id** al realizar el registro en la consola.

*10) Actualizar datos*

La actualización de datos se simplifica mediante Mongoose. Siguiendo la extensión del ejemplo previo, se procederá a modificar el título del artículo.

```
article.title = "The Most Awesomest Post!!";
await article.save();
console.log(article);
```

La posibilidad de editar el objeto local directamente y luego emplear el método **save()** para reflejar la actualización en la base de datos brinda una facilidad notable. Es difícil imaginar un proceso más sencillo que este.

*11) Encontrar datos*

Es imperativo garantizar que se esté actualizando el documento preciso. Para lograrlo, se empleará un método exclusivo de Mongoose, **findById()**, con el fin de obtener el documento mediante su **ObjectId**.

```
const article = await Blog.findById("62472b6ce09e8b77266d6b1b").exec();
console.log(article);
```

AVISO
Se destaca el uso de la función **exec()** de Mongoose. Aunque técnicamente es opcional, su implementación devuelve una promesa. Según la experiencia, se recomienda emplear esta función para prevenir posibles complicaciones. Para obtener información adicional, se sugiere consultar esta nota en la documentación de Mongoose acerca de las promesas.

Existen numerosas opciones de consulta disponibles en Mongoose. Se invita a explorar la lista completa de estas consultas.

*12) Proyección de campos de documentos*

Siguiendo la misma línea que el controlador convencional de MongoDB para Node.js, es factible seleccionar únicamente los campos necesarios. En este caso, se buscarán exclusivamente los campos de **title**, **slug** y **content**.

```
const article = await Blog.findById("62472b6ce09e8b77266d6b1b", "title slug
content").exec();
console.log(article);
```

El segundo parámetro puede adoptar la forma de **Object|String|Array<String>**, permitiendo así especificar los campos que se desean proyectar. En esta situación, se opta por utilizar un **String**.

*13) Borrar datos*

De manera similar al controlador convencional de MongoDB para Node.js, se cuentan con los métodos deleteOne() y deleteMany().

```
const blog = await Blog.deleteOne({ author: "Jesse Hall" })
console.log(blog)

const blog = await Blog.deleteMany({ author: "Jesse Hall" })
console.log(blog)
```

*14) Validación*

Es relevante notar que los documentos que se han insertado hasta el momento no incluyen información sobre el autor, fechas o comentarios. Aunque se ha delineado la apariencia deseada para la estructura del documento, aún no se ha especificado qué campos son esenciales. En este punto, se ha permitido la omisión de cualquier campo. Ahora, es pertinente definir algunos campos obligatorios en el esquema Blog.js.

```
const blogSchema = new Schema({
  title:  {
    type: String,
    required: true,
  },
  slug:  {
    type: String,
    required: true,
    lowercase: true,
  },
  published: {
    type: Boolean,
    default: false,
  },
  author: {
    type: String,
    required: true,
  },
  content: String,
  tags: [String],
  createdAt: {
    type: Date,
    default: () => Date.now(),
    immutable: true,
  },
  updatedAt: Date,
  comments: [{
    user: String,
    content: String,
    votes: Number
  }]
});
```

Al incorporar validación en un campo, se suministra un objeto como su valor.

CONSEJO
**value: String** es el mismo el valor que: **{type: String}**.

Existen diversos métodos de validación disponibles para su utilización. La propiedad **required** se puede establecer como **true** en cualquier campo que se deba ser obligatorio. En el caso del **slug**, se busca que la cadena de texto esté siempre en minúsculas, para ello, se puede fijar **lowercase** en **true**. De esta manera, la entrada del **slug** se convertirá a minúsculas antes de almacenar el documento en la base de datos. Respecto a la fecha de creación, se puede definir el valor predeterminado mediante una función flecha. Además, si se desea que esta fecha sea inmutable y no pueda modificarse más adelante, se puede configurar **immutable** como **true**.

AVISO
Los validadores sólo se ejecutan en los métodos crear o guardar.

*15) Otros métodos útiles*

Mongoose incorpora numerosos métodos convencionales de MongoDB y añade varios métodos auxiliares que se han abstraído de los métodos regulares de MongoDB. A continuación, se describirán solo algunos de ellos.

**exists()**

El método **exists()** devuelve **null** o el **ObjectId** de un documento que cumple con la consulta suministrada.

```
const blog = await Blog.exists({ author: "Jesse Hall" })
console.log(blog)
```

**where()**

Mongoose presenta su propio enfoque para la consulta de datos. El método **where()** posibilita la concatenación y construcción de consultas de manera efectiva.

```
// Instead of using a standard find method
const blogFind = await Blog.findOne({ author: "Jesse Hall" });

// Use the equivalent where() method
const blogWhere = await Blog.where("author").equals("Jesse Hall");
console.log(blogWhere)
```

Cualquiera de estas alternativas es válida. Se puede optar por aquella que resulte más intuitiva. Además, es posible encadenar múltiples instancias del método **where()** para abarcar incluso consultas más complejas. En el caso de que sea necesario incorporar la proyección al emplear el método **where()**, se debe concatenar el método **select()** después de la consulta.

```
const blog = await Blog.where("author").equals("Jesse hall").select("title author")
console.log(blog)
```

*16) Esquemas múltiples*

Es fundamental comprender las opciones disponibles al modelar datos. Para aquellos familiarizados con entornos de bases de datos relacionales, la práctica común implica tener tablas distintas para los datos relacionados. En MongoDB, en términos generales, se aconseja almacenar juntos los datos que se acceden de manera conjunta. Se recomienda planificar esta estructura con antelación, si es posible, y anidar datos dentro del mismo esquema cuando resulte lógico. En situaciones que demanden esquemas separados, Mongoose facilita considerablemente esta tarea.

Para ilustrar cómo se pueden utilizar varios esquemas en conjunto, se creará un nuevo esquema en un archivo independiente llamado **User.js** en la carpeta **model**.

```
import mongoose from 'mongoose';
const {Schema, model} = mongoose;

const userSchema = new Schema({
  name: {
    type: String,
    required: true,
  },
  email: {
    type: String,
    minLength: 10,
    required: true,
    lowercase: true
  },
});

const User = model('User', userSchema);
export default User;
```

En el caso del correo electrónico, se emplea una propiedad recién introducida, **minLength**, con el propósito de especificar la longitud mínima de caracteres necesaria para esta cadena de texto. A continuación, se mencionará este nuevo modelo de usuario en el esquema del blog, tanto para el autor como para **comments.user**.

```
import mongoose from 'mongoose';
const { Schema, SchemaTypes, model } = mongoose;

const blogSchema = new Schema({
  ...,
  author: {
    type: SchemaTypes.ObjectId,
    ref: 'User',
    required: true,
  },
  ...,
  comments: [{
    user: {
      type: SchemaTypes.ObjectId,
      ref: 'User',
      required: true,
    },
    content: String,
    votes: Number
  }];
});
...
```

En esta sección, se definen el **author** y **comments.user** como **SchemaTypes.ObjectId**, incorporando además una referencia al modelo de usuario mediante el atributo **ref**. Esta elección posibilita realizar **joins** en los datos en una etapa posterior. Es fundamental recordar realizar la desestructuración de los **SchemaTypes** de Mongoose al principio del archivo.

Finalmente, se procede a actualizar el archivo **index.js**. Para ello, será necesario importar el nuevo modelo de usuario, generar un usuario nuevo y crear un artículo adicional utilizando el **_id** del usuario recién creado.

```
...
import User from './model/User.js';

...

const user = await User.create({
  name: 'Jesse Hall',
  email: 'jesse@email.com',
});

const article = await Blog.create({
  title: 'Awesome Post!',
  slug: 'Awesome-Post',
  author: user._id,
  content: 'This is the best post ever',
  tags: ['featured', 'announcement'],
});

console.log(article);
```

Es notable que actualmente existe una colección de usuarios en conjunto con la colección de blogs en la base de datos de MongoDB. En la visualización actual, únicamente se mostrará el **_id** del usuario en el campo de autor. Entonces, surge la pregunta de cómo obtener toda la información del autor junto con el artículo. La solución radica en emplear el método **populate()** de Mongoose.

```
const article = await Blog.findOne({ title: "Awesome Post!" }).populate("author");
console.log(article);
```

En este momento, la información del autor ha sido poblada o "unida" a los datos del artículo. Mongoose, en realidad, utiliza el método **$lookup** de MongoDB en segundo plano para llevar a cabo esta operación.

*17) Middleware*

Dentro de Mongoose, los middlewares son funciones que se ejecutan antes y/o durante la ejecución de funciones asíncronas a nivel del esquema. Un caso ilustrativo implica la actualización de la fecha de modificación cada vez que se guarda o actualiza un artículo, y esto se añadirá al modelo **Blog.js** como ejemplo.

```
blogSchema.pre('save', function(next) {
  this.updated = Date.now(); // update the date every time a blog post is saved
  next();
});
```

Posteriormente, en el archivo **index.js**, se localiza un artículo, se modifica el título y, finalmente, se procede a su almacenamiento.

```
const article = await Blog.findById("6247589060c9b6abfa1ef530").exec();
article.title = "Updated Title";
await article.save();
console.log(article);
```

Es perceptible que en la actualidad el documento cuenta con una fecha de actualización. Además del método **pre()**, en Mongoose también dispone de una función de middleware **post()**.

*18) Próximos pasos*

Se sugiere que este ejemplo podría beneficiarse de la implementación de otro esquema para gestionar los comentarios. Se recomienda crear dicho esquema y probarlo mediante la adición de algunos usuarios y comentarios.

Es importante destacar que existen numerosos métodos adicionales de Mongoose que son altamente útiles y no se abordan en esta sección. Para obtener información detallada y más ejemplos, se aconseja revisar la documentación oficial de Mongoose.

## VII. INTEGRACIÓN DE NESTJS CON MONGODB

En esta sección se hará la implementación de una aplicación sencilla para la gestión de tareas, utilizando como marco de trabajo para el back-end a NestJS y como base de datos para el almacenamiento a MongoDB.

### *A. Prerrequisitos*

Para poder implementar la aplicación de tareas, es necesario contar con las siguientes herramientas instaladas en el entorno de desarrollo:

1. Nodejs: Es el entorno de ejecución de JavaScript fuera del navegador, que permite la creación de aplicaciones web. En este entorno es donde se ejecuta el marco de trabajo NestJS
2. MongoDB: Es una base de datos NoSQL basada en documentos, que permite la gestión y el almacenamiento de datos. Para utilizar MongoDB de manera local, se recomienda utilizar MongoDB Community Server. O bien, se puede optar por MongoDB Atlas, que es una base de datos como servicio, que cuenta con una capa gratuita para implementaciones de práctica.
3. Visual Studio Code: Es un editor de código bastante potente y utilizando en el mundo del desarrollo de software.

### *B. Estructura de carpetas*

Para organizar el proyecto que se va a desarrollar en esta guía, se recomienda tener una carpeta o directorio que se encargue de contener los proyectos relacionados al desarrollo web, una opción válida puede ser:

```
C:\Dev-Web
```

AVISO
La elección del nombre de la carpeta es a gusto del desarrollador.

### *C. Crear un nuevo proyecto*

Para crear un nuevo proyecto siga los siguientes pasos:

1. Abrir una nueva terminal del sistema operativo y ubicarse en la carpeta encargada de contener los proyectos de desarrollo web, para nuestro caso particular:

```
C:\Dev-Web
```

2. Ejecutar el siguiente comando para instalar el CLI (Command Line Interface) de NestJS:

```
npm install -g @nestjs/cli
```

3. Para crear un nuevo proyecto utilice el siguiente comando

```
nest new nombre-del-proyecto
```

4. Para este ejemplo:

```
nest new taskapi
```

AVISO
El CLI de Nestjs va a preguntar que manejador de paquetes queremos utilizar, para este ejemplo se hace uso **npm**

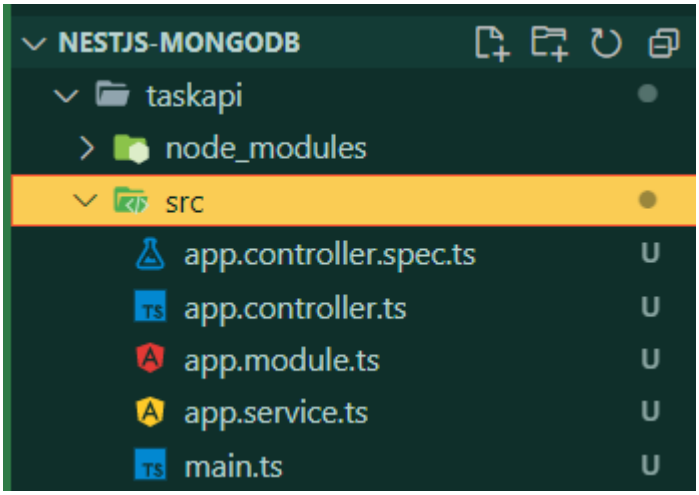
5. Abrir el proyecto en VSCode con el comando:

```
cd  taskapi
code .
```

*D. Eliminar archivos innecesarios para el proyecto*

Al crear un nuevo proyecto se van a generar los siguientes archivos dentro de la carpeta **taskapi\src:**

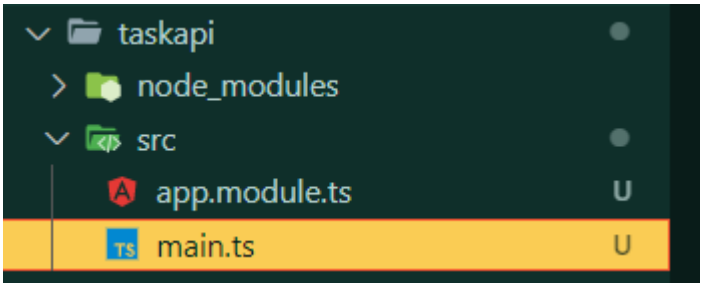
*Imagen 13 Estructura de carpetas y archivos de taskapi*




*Fuente: Autor*

Eliminamos los archivos: **app.controller.spec.ts, app.controller.ts, app.service.ts,** ya que no son necesarios para desarrollar el ejemplo.

*Imagen 14 Estructura de archivos de taskapi/src*




*Fuente: Autor*

Una vez estén eliminados los archivos, limpiamos el archivo **app.module.ts**, dejándolo de la siguiente forma:

app.module.ts

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

@Module({
  imports: [],
})
export class AppModule {}
```

*E. Modificar el archivo main.ts*

Al archivo **main.ts** se le agregan unas cuantas configuraciones adicionales para mejorar la experiencia de desarrollo, el archivo debe quedar de la siguiente manera:

main.ts

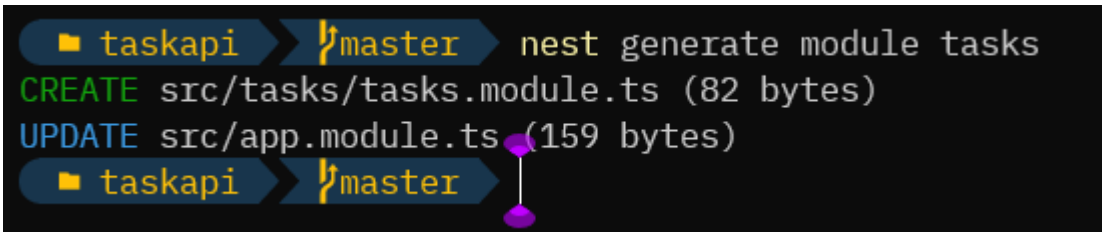
```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { ValidationPipe } from '@nestjs/common';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.enableCors();
  app.setGlobalPrefix('api');
  app.useGlobalPipes(new ValidationPipe());
  await app.listen(3000);
}
bootstrap();
```

*F. Generar el módulo tasks*

Para generar el módulo tasks, se debe abrir una nueva terminal en la ubicación **C:\Web-Dev\taskapi** y luego ejecutar el siguiente comando:

```
nest generate module tasks
```

*Imagen 15 Generación del módulo tasks*



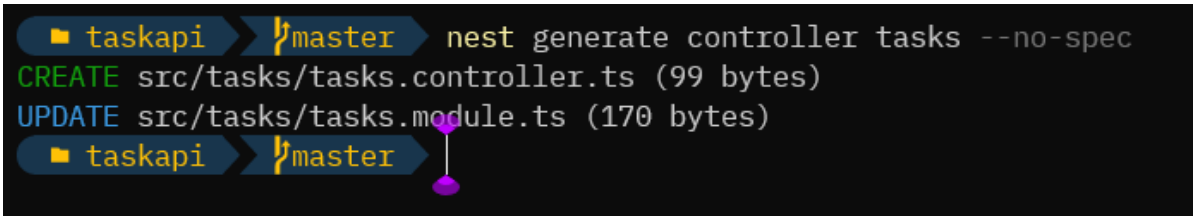

*Fuente: Autor*

*G. Generar el controlador tasks sin el archivo de testing*

Para generar el controlador tasks, sin el archivo de testeo, se debe abrir una nueva terminal en la ubicación **C:\Web-Dev\taskapi** y luego ejecutar el siguiente comando:

```
nest generate controller tasks --no-spec
```

*Imagen 16 Generación del controlador de task, sin el archivo de testing*



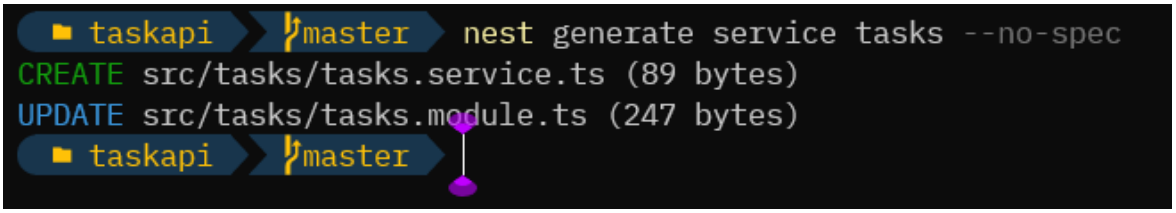

*Fuente: Autor*

*H. Generar el servicio tasks sin el archivo de testing*

Para generar el servicio tasks, sin el archivo de testeo, se debe abrir una nueva terminal en la ubicación **C:\Web-Dev\taskapi** y luego ejecutar el siguiente comando:

```
nest generate service tasks --no-spec
```

*Imagen 17 Generación del servicio, sin el archivo de testing*



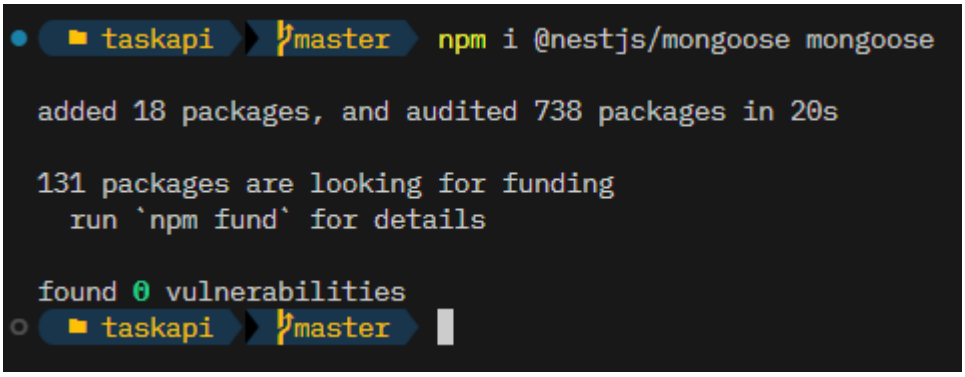

*Fuente: Autor*

*I. Conectar la aplicación con MongoDB*

Para instalar Mongoose, el ODM para manejar MongoDB, se debe abrir una nueva terminal en la ubicación **C:\Dev-Web\taskapi** y luego ejecutar el siguiente comando:

```
npm i @nestjs/mongoose mongoose
```

*Imagen 18 Instalar los paquetes necesarios para trabajar con Mongoose*




*Fuente: Autor*

Una vez instalado, importamos el **MongooseModule** dentro de **AppModule:**

app.module.ts

```
import { Module } from '@nestjs/common';
import { TasksModule } from './tasks/tasks.module';
import { MongooseModule } from '@nestjs/mongoose';

@Module({
  imports: [MongooseModule.forRoot('mongodb://127.0.0.1:27017/taskdb'), TasksModule],
})
export class AppModule { }
```

En este caso se utiliza la conexión **mongodb://127.0.0.1:27017/taskdb** debido a que se tiene una implementación en local de **MongoDB Community Server**.

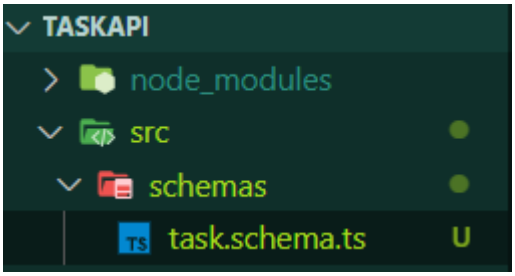
CONSEJO
Para obtener más información acerca de las conexiones con un servidor local de MongoDB o MongoDB Atlas diríjase a este enlace.

CONSEJO
Para obtener más información acerca de las conexiones con Mongoose diríjase a este enlace.

*J. Construir el esquema de task*

Dentro de la ruta **taskapi\src** vamos a crear una nueva carpeta llamada **schemas** y dentro de la carpeta **schemas** un nuevo archivo llamado **task.schema.ts**

*Imagen 19 Estructura de carpetas del archivo para el esquema*




*Fuente: Autor*

El archivo **task.schema.ts** queda definido de la siguiente manera:

task.schema.ts

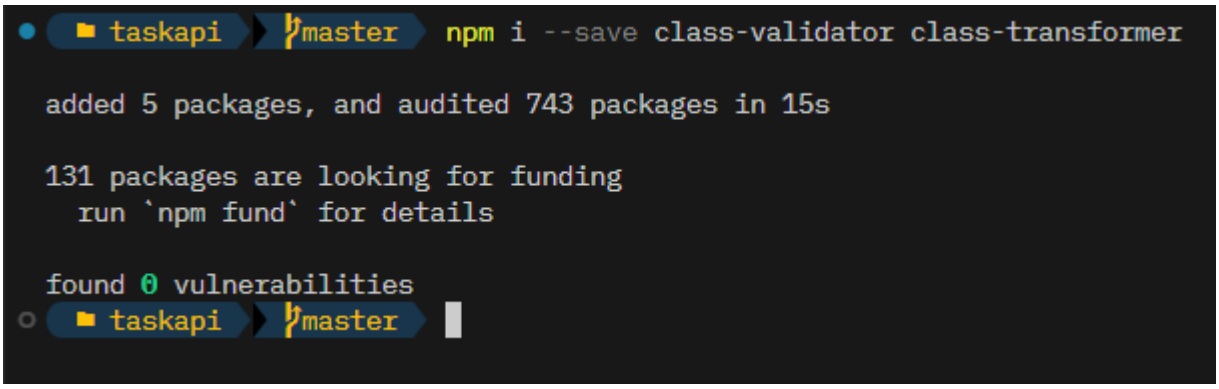
```
import { Schema, Prop, SchemaFactory } from '@nestjs/mongoose';

@Schema({ timestamps: true })
export class Task {
  @Prop({ unique: true, required: true, trim: true })
  title: string;
  @Prop({ trim: true })
  description: string;
  @Prop({ default: false })
  done: boolean;
}

export const TaskSchema = SchemaFactory.createForClass(Task);
```

*K. Construir los DTO (Data Transfer Object)*

Para poder utilizar DTOs dentro de la aplicación, se deben instalar los paquetes de validación, para ello abrimos una nueva terminal en la ruta **C:\Dev-Web\taskapi** y ejecutamos el comando:

```
npm i --save class-validator class-transformer
```

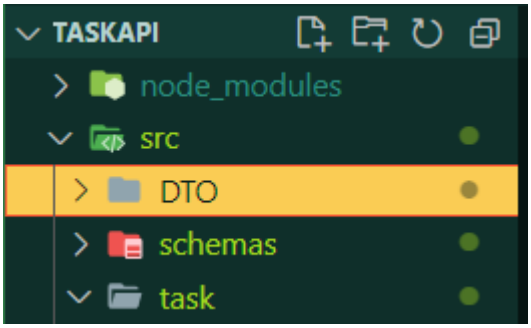
*Imagen 20 Instalar paquetes necesarios para trabajar con los DTOs*




*Fuente: Autor*

Dentro de la ruta **taskapi\src** creamos una nueva carpeta llamada **DTO**

*Imagen 21 Carpeta DTO*




*Fuente: Autor*

1) *Create Task DTO*

Dentro de la carpeta **DTO** creamos un nuevo archivo llamado **create-task.dto.ts,** que contiene lo siguiente:

create-task.dto.ts

```
import { IsString, IsBoolean, IsOptional, IsNotEmpty } from 'class-validator';

export class CreateTaskDTO {
  @IsString()
  @IsNotEmpty()
  title: string;
  @IsString()
  @IsOptional()
  description?: string;
  @IsBoolean()
  @IsOptional()
  done?: boolean;
}
```

2) *Update Task DTO*

Dentro de la carpeta **DTO** creamos un nuevo archivo llamado **update-task.dto.ts,** que contiene lo siguiente:

update-task.dto.ts

```
import { IsString, IsBoolean, IsOptional } from 'class-validator';

export class UpdateTaskDTO {
  @IsString()
  @IsOptional()
  title?: string;
  @IsString()
  @IsOptional()
  description?: string;
  @IsBoolean()
  @IsOptional()
  done?: boolean;
}
```

*L. Construir el servicio*

El servicio se encarga de definir que operaciones va a realizar la aplicación de tareas, el archivo **tasks.service.ts** queda definido de la siguiente manera:

tasks.service.ts

```
import { Injectable } from '@nestjs/common';
import { InjectModel } from '@nestjs/mongoose';
import { Task } from '../schemas/task.schema';
import { Model } from 'mongoose';
import { CreateTaskDTO } from '../dto/create-task.dto';
import { UpdateTaskDTO } from '../dto/update-task.dto';

@Injectable()
export class TasksService {
  constructor(@InjectModel(Task.name) private taskModel: Model<Task>) {}

  async findAll() {
    return this.taskModel.find();
  }

  async findOne(id: string) {
    return this.taskModel.findById(id);
  }

  async createOne(createTask: CreateTaskDTO) {
    const newTask = new this.taskModel(createTask);
    return newTask.save();
  }

  async updateOne(id: string, task: UpdateTaskDTO) {
    return this.taskModel.findByIdAndUpdate(id, task, { new: true });
  }

  async deleteOne(id: string) {
    return this.taskModel.findByIdAndDelete(id);
  }
}
```

*M. Construir el controlador*

El controlador se encarga de gestionar el ciclo solicitud-respuesta, el archivo **tasks.controller.ts** queda definido de la siguiente manera:

tasks.controller.ts

```
import { CreateTaskDTO } from 'src/dto/create-task.dto';
import { TasksService } from './tasks.service';
import {
  Controller,
  Get,
  Post,
  Put,
  Delete,
  Param,
  Body,
  ConflictException,
  NotFoundException,
  HttpCode,
} from '@nestjs/common';
import { UpdateTaskDTO } from 'src/dto/update-task.dto';

@Controller('tasks')
export class TasksController {
  constructor(private tasksService: TasksService) {}

  @Get()
  findAll() {
    return this.tasksService.findAll();
  }

  @Get(':id')
  async findOne(@Param('id') id: string) {
    const task = await this.tasksService.findOne(id);
    if (!task) {
      throw new NotFoundException('Task not found');
    }
    return task;
  }

  @Post()
  async create(@Body() createTask: CreateTaskDTO) {
    try {
      return await this.tasksService.createOne(createTask);
    } catch (error) {
      if (error.code === 11000) {
        throw new ConflictException('Task already exists');
      }
      throw error;
    }
  }

  @Put(':id')
  async updateOne(@Param('id') id: string, @Body() updateTask: UpdateTaskDTO) {
    const task = await this.tasksService.updateOne(id, updateTask);
    if (!task) {
      throw new NotFoundException('Task not found');
    }
    return task;
  }
```

```
  @Delete(':id')
  @HttpCode(204)
  async deleteOne(@Param('id') id: string) {
    const task = await this.tasksService.deleteOne(id);
    if (!task) {
      throw new NotFoundException('Task not found');
    }
    return task;
  }
}
```

*N. Construir el módulo*

El módulo se encarga de definir la estructura de la aplicación, el archivo **task.module.ts** queda definido de la siguiente manera:

task.module.ts

```
import { Module } from '@nestjs/common';
import { TasksController } from './tasks.controller';
import { TasksService } from './tasks.service';
import { MongooseModule } from '@nestjs/mongoose';
import { Task, TaskSchema } from '../schemas/task.schema';

@Module({
  imports: [
    MongooseModule.forFeature([{ name: Task.name, schema: TaskSchema }]),
  ],
  controllers: [TasksController],
  providers: [TasksService],
})
export class TasksModule {}
```

*O. Integrar el back-end a un front-end*

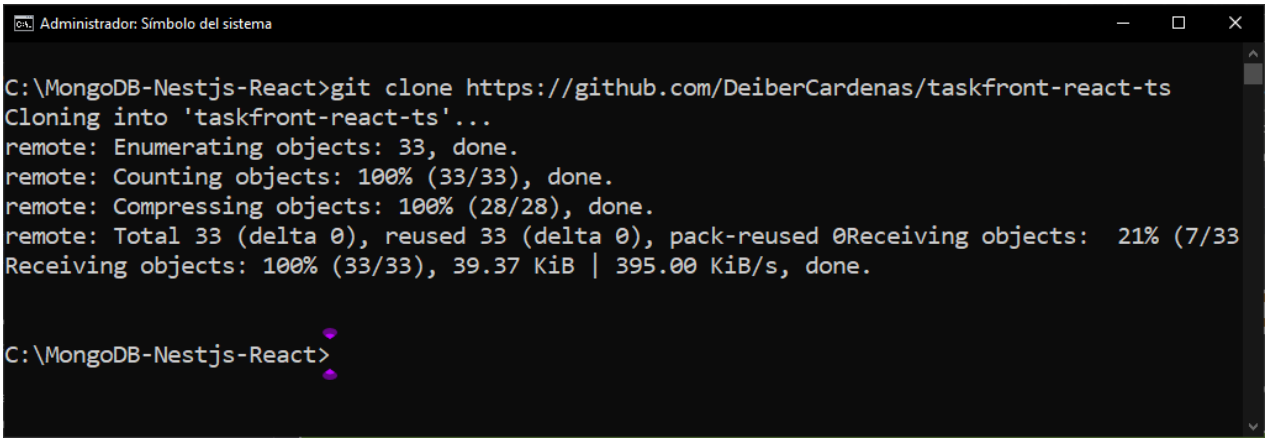
Para realizar las pruebas de funcionamiento de la aplicación de tareas, vamos a integrar el back-end con un front-end básico ya construido.

*1) Clonar el repositorio:*

Para obtener el código fuente del front-end la opción recomendada es clonar el repositorio, para ello, abra una nueva terminal desde la ruta madre del proyecto (**C:\Dev-Web**) y utilice el siguiente comando:

```
git clone https://github.com/DeiberCardenas/taskfront-react-ts
```

*Imagen 22 Clonar el proyecto del front-end*

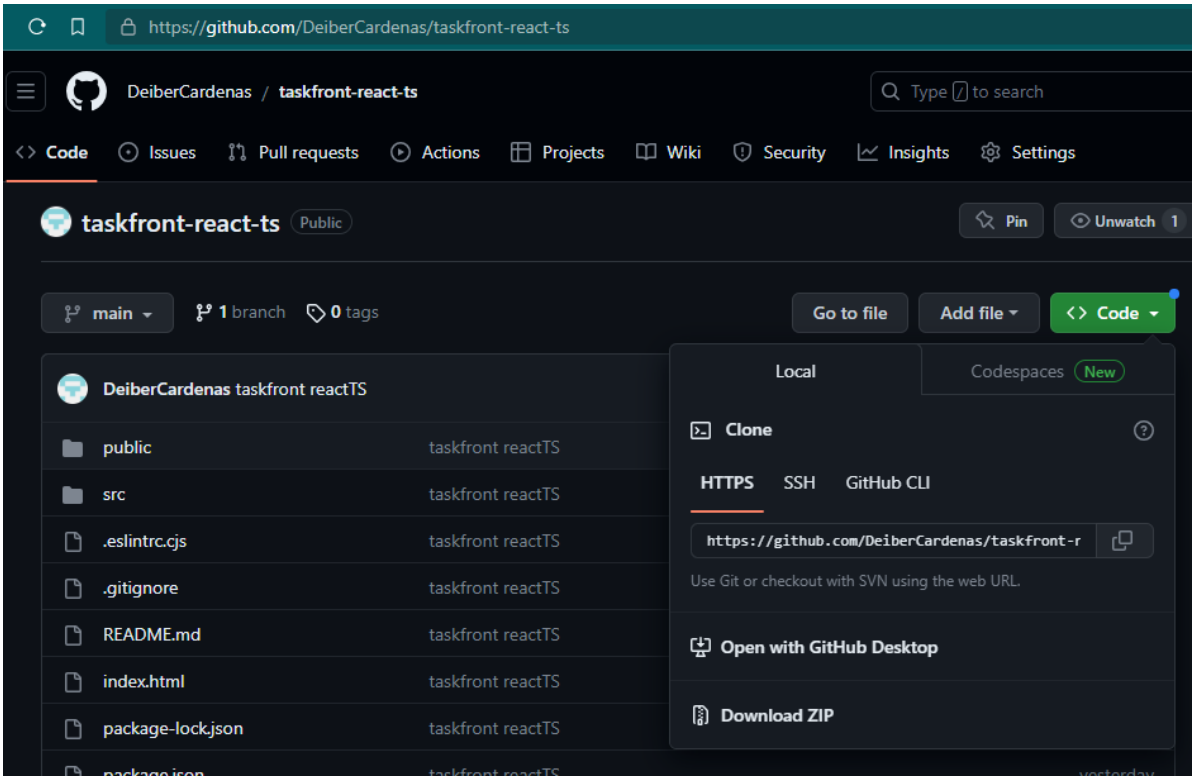



*Fuente: Autor*

2) *Descargar el repositorio:*

Si bien la opción de clonar el repositorio es la más recomendada para implementar el código del front-end, puede optar por descargar el repositorio del front-end ingresando a https://github.com/DeiberCardenas/taskfront-react-ts , luego dar clic en el botón **Code** y después dar clic en **Download ZIP** para descargar el código.

*Imagen 23 Descargar el repositorio del front-end*




*Fuente: Autor*

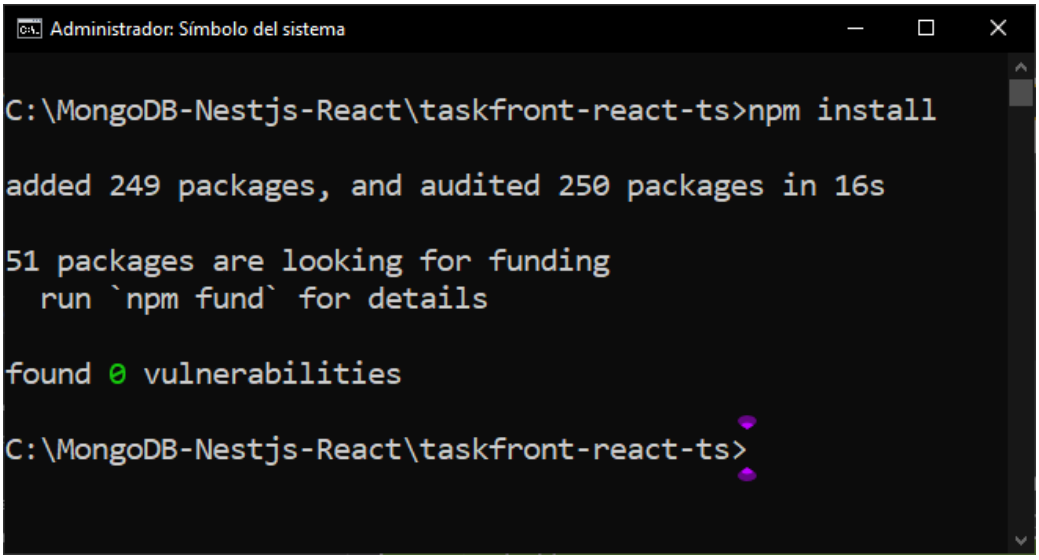
CONSEJO
Descargue y descomprima el contenido en la carpeta madre, se sugiere la carpeta **C:\Dev-Web**

3) *Instalar los paquetes del front-end*

Para instalar los paquetes necesarios para el funcionamiento del front-end, abra una nueva terminal desde la ubicación del proyecto, para nuestro caso en particular **C:\Dev-Web\taskfront-react-ts** y luego ejecute el siguiente comando:

```
npm install
```

*Imagen 24 Instalar los paquetes del front-end*



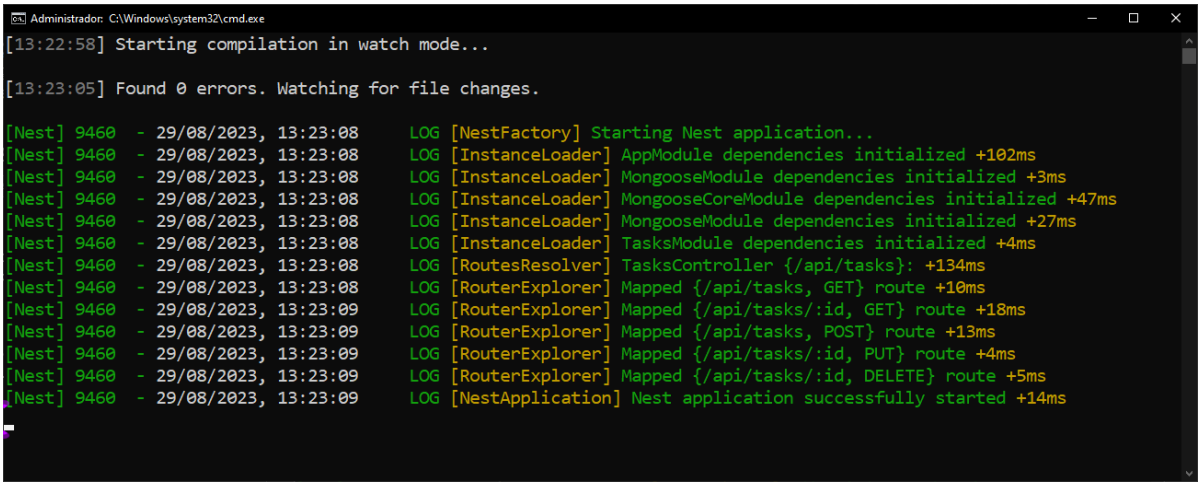

*Fuente: Autor*

*4) Ejecutar el back-end*

Para ejecutar el back-end de la aplicación, abrimos una nueva terminal en la ruta **C:\Dev-Web\taskapi,** luego ejecutamos el siguiente comando:

```
npm run start:dev
```

*Imagen 25 Ejecución del proyecto back-end*



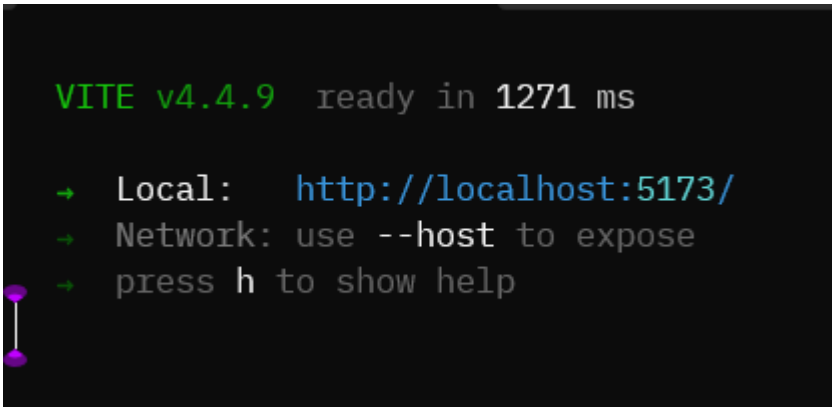

*Fuente: Autor*

*5) Ejecutar el front-end*

Para ejecutar el front-end de la aplicación, abrimos una nueva terminal en la ruta **C:\Dev-Web\taskfront-react-ts,** luego ejecutamos el siguiente comando:

```
npm run dev
```

*Imagen 26 Ejecución del proyecto front-end*

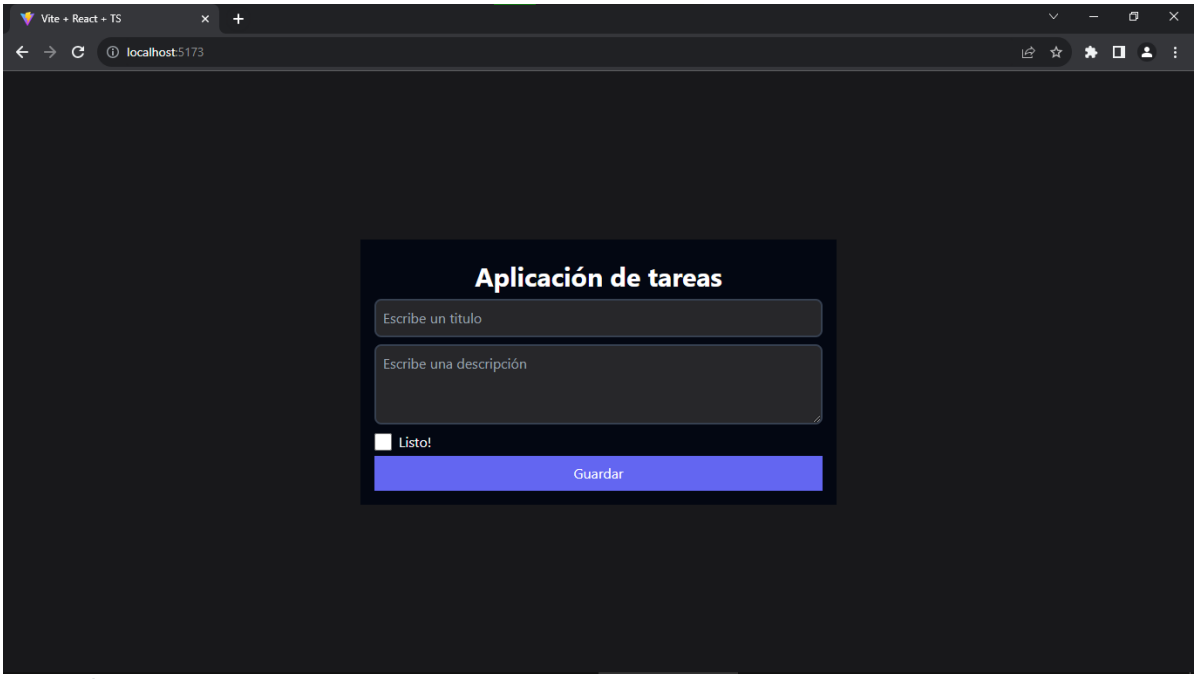



*Fuente: Autor*

6) *Ver resultados*

Abra un navegador web (En este caso se utilizó Google Chrome), y diríjase a http://localhost:5173/

*Imagen 27 Resultados de la ejecución del proyecto de gestión de tareas*




*Fuente: Autor*

## VIII. RESULTADOS DE LA REVISIÓN DEL SEMILLERO VERITATE

Para asegurar la idoneidad y aplicabilidad de la guía desarrollada, se realizó un proceso de revisión y verificación en colaboración con el semillero Veritate Software. Este proceso se desarrolló en las siguientes etapas:

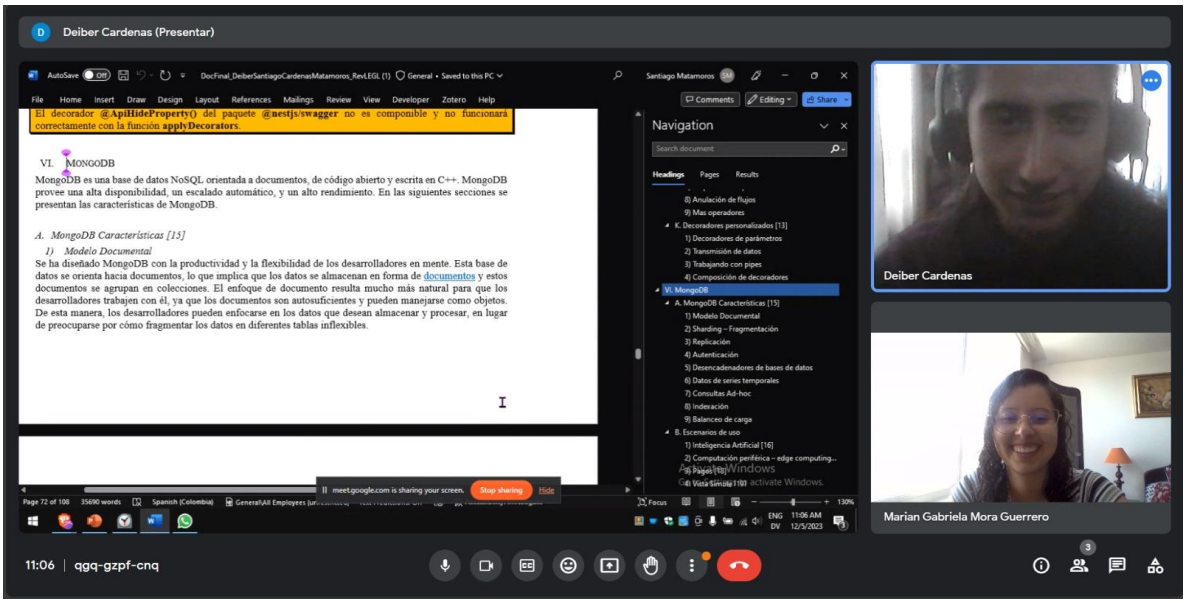
### *A. Convocatoria y presentación*

Se convocó a una sesión especial con los miembros del semillero Veritate Software para presentarles la guía de conocimiento desarrollada. En esta presentación se dio a conocer:

- Los objetivos generales y específicos de la guía.
- La estructura del protocolo diseñada.
- La metodología empleada en la construcción de la guía.
- Los contenidos enfocados en MongoDB y NestJS, resaltando su relevancia en el desarrollo de aplicaciones backend.

### *B. Sesión interactiva*

*Imagen 28 Evidencia sesión interactiva*




*Fuente: Autor*

Se llevó a cabo una sesión interactiva donde se fomentó la participación de los miembros del semillero. Durante esta sesión:

- Se permitió a los participantes explorar la guía y su contenido.
- Se abrió un espacio para preguntas, aclaraciones y sugerencias.
- Se solicitó la realimentación detallada sobre la claridad, profundidad y utilidad de la guía en relación con el objetivo de capacitar en MongoDB y NestJS.

Las conclusiones de esta sesión interactiva fueron las siguientes:

1. El documento presenta demasiados tecnicismos, lo cual dificulta la compresión y asimilación de los temas presentados, si bien la guía se enfoca en herramientas especificas es recomendable utilizar un lenguaje más natural que mejore la accesibilidad y claridad del documento.
2. Es necesario incluir un glosario que permita a los lectores revisar la terminología del documento, sin la necesidad de diccionarios adicionales.

### *C. Evaluación y realimentación*

Con las conclusiones de la sesión interactiva se realizó un análisis en profundidad acerca de las falencias de documento y como resultado de este análisis se planteó realizar los siguientes cambios en el documento:

1. Utilizar un lenguaje más natural en las secciones de NestJS y MongoDB.
2. Añadir un glosario que incluya los términos técnicos utilizados en la guía.

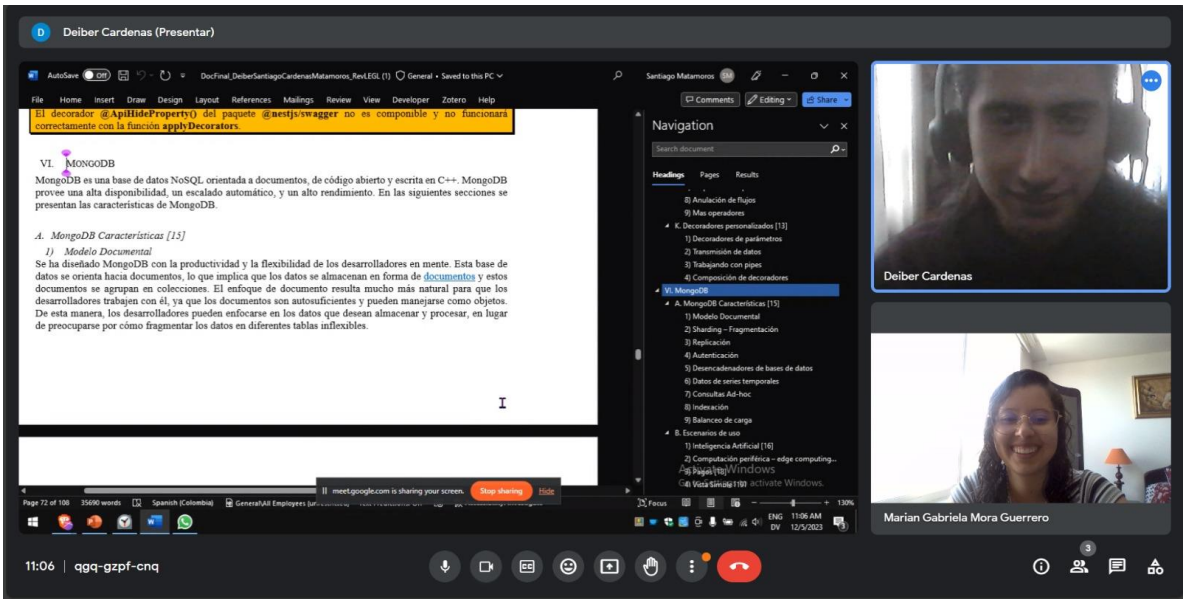
### *D. Implementación de mejoras*

Con base a la realimentación recibida, se realizaron los cambios pertinentes en la guía para mejorar su claridad y utilidad. Estos cambios se resumen de la siguiente manera:

1. Cambiar los términos técnicos en ingles por términos en español.
2. Dar explicaciones técnicas con un lenguaje más natural para la comodidad del lector.
3. Añadir un glosario con la terminología técnica utilizada a lo largo de la guía como una sección inicial del documento.

### *E. Validación final*

*Imagen 29 Evidencia sesión interactiva*




*Fuente: Autor*

Se convocó a una sesión final para validar las mejoras implementadas con la colaboración del semillero. Se verificó que las modificaciones realizadas hayan abordado adecuadamente las preocupaciones o sugerencias previamente planteadas.

## IX. CONCLUSIONES

1. Contribución al Desarrollo Profesional: Este trabajo de grado representa una contribución significativa al desarrollo profesional en el campo del desarrollo de software. La guía de conocimiento tecnológico desarrollada proporciona un recurso valioso para la formación de profesionales en el uso eficiente y efectivo de MongoDB y NestJS. Al dotar a los desarrolladores con habilidades específicas en estas tecnologías de vanguardia, se fomenta la creación de aplicaciones backend robustas y escalables, lo que impacta directamente en la calidad y competitividad de los productos y servicios tecnológicos.

2. Innovación Educativa en Tecnologías Emergentes: La implementación de la guía tecnológica con ejemplos funcionales y la validación a través del semillero Veritate Software no solo aseguran la aplicabilidad práctica de los conocimientos impartidos, sino que también establecen un modelo innovador para la enseñanza de tecnologías emergentes en entornos académicos. Este enfoque práctico no solo brinda a los estudiantes una comprensión teórica, sino que también les permite enfrentarse a situaciones del mundo real, preparándolos mejor para los desafíos que enfrentarán en la industria del desarrollo de software.

3. Fomento de la Colaboración Académica e Industrial: La guía creada, al involucrar al semillero Veritate Software, ha establecido un puente efectivo entre la academia y la industria: proporciona a los estudiantes no solo conocimientos actualizados en MongoDB y NestJS, sino también una experiencia práctica al aplicar estos conocimientos en proyectos reales. La retroalimentación continua con el semillero ha mejorado la guía, garantizando su relevancia y adaptación a las demandas cambiantes del entorno empresarial, mientras que la conexión directa con problemas reales ha enriquecido la formación, preparando a los estudiantes para resolver desafíos auténticos al ingresar al mundo laboral. Esta colaboración ha fomentado redes profesionales y una comprensión más profunda de las necesidades de la industria, brindando a los estudiantes una ventaja significativa al graduarse.

## X. RECOMENDACIONES

1. Explorar Recursos Adicionales y Comunidades en Línea: Los desarrolladores deben aprovechar al máximo los recursos en línea y las comunidades especializadas en MongoDB y NestJS. Plataformas como documentación oficial, tutoriales en línea, y foros de desarrollo pueden proporcionar información adicional, casos de uso específicos y soluciones a desafíos comunes. La participación en comunidades en línea también permite establecer conexiones con otros profesionales que comparten experiencias y conocimientos valiosos.

2. Participar en Proyectos Prácticos y Contribuir a Repositorios de Código Abierto: La práctica continua es esencial para el dominio de cualquier tecnología. Los desarrolladores deben involucrarse en proyectos prácticos que implementen las tecnologías MongoDB y NestJS de manera significativa. Contribuir a repositorios de código abierto relacionados con estas tecnologías no solo proporciona experiencia práctica, sino que también ofrece la oportunidad de recibir retroalimentación de la comunidad, mejorar habilidades de codificación y entender en profundidad las mejores prácticas de desarrollo.

3. Mantenerse Actualizado con las Novedades y Mejores Prácticas: Dada la naturaleza dinámica del desarrollo de software, es crucial mantenerse al día con las últimas actualizaciones, características y mejores prácticas en MongoDB y NestJS. Sus respectivas comunidades y sitios web oficiales suelen publicar noticias, actualizaciones y guías que son fundamentales para estar al tanto de los cambios tecnológicos. La adopción temprana de nuevas características y el seguimiento de las mejores prácticas aseguran que los desarrolladores continúen evolucionando y aprovechando al máximo estas tecnologías en constante cambio.

## XI. REFERENCIAS


[1] M. Inc, 'Introduction to MongoDB'. [Online]. Available: https://www.mongodb.com/docs/manual/introduction/
[2] K. Myśliwiec, 'Nestjs'. 2023. [Online]. Available: https://docs.nestjs.com/
[3] K. Myśliwiec, 'Firts Steps'. 2023. [Online]. Available: https://docs.nestjs.com/first-steps
[4] K. Myśliwiec, 'Controllers'. 2023. [Online]. Available: https://docs.nestjs.com/controllers
[5] K. Myśliwiec, 'Providers'. 2023. [Online]. Available: https://docs.nestjs.com/providers
[6] K. Myśliwiec, 'Modules'. 2023. [Online]. Available: https://docs.nestjs.com/modules
[7] K. Myśliwiec, 'Middleware'. 2023. [Online]. Available: https://docs.nestjs.com/middleware
[8] K. Myśliwiec, 'Exception filters'. 2023. [Online]. Available: https://docs.nestjs.com/exception-filters
[9] K. Myśliwiec, 'Pipes'. 2023. [Online]. Available: https://docs.nestjs.com/pipes
[10] K. Myśliwiec, 'Built-in pipes'. [Online]. Available: Built-in pipes
[11] K. Myśliwiec, 'Guards'. 2023. [Online]. Available: https://docs.nestjs.com/guards
[12] K. Myśliwiec, 'Interceptors'. 2023. [Online]. Available: https://docs.nestjs.com/interceptors
[13] K. Myśliwiec, 'Custom decorators'. 2023. [Online]. Available: https://docs.nestjs.com/custom-decorators
[14] K. Myśliwiec, 'Param decorators'. [Online]. Available: https://docs.nestjs.com/custom-decorators#param-decorators
[15] M. Inc, 'MongoDB Features'. Accessed: Nov. 07, 2023. [Online]. Available: https://www.mongodb.com/features
[16] M. Inc, 'Embedding Generative AI and Advanced Search into your Apps with MongoDB'. Jul. 2023. [Online]. Available: https://www.mongodb.com/library/ai/embedding-generative-ai?lb-mode=overlay
[17] A. Lee, 'Introducing Atlas for the Edge'. [Online]. Available: https://www.mongodb.com/library/atlas-edge/introducing-atlas-for-the-edge?lb-mode=overlay
[18] M. Inc, 'Modernize Your Payments Architecture'. [Online]. Available: https://www.mongodb.com/library/payments/modernize-your-payments-systems?lb-mode=overlay
[19] M. Inc, 'Single View'. [Online]. Available: https://www.mongodb.com/use-cases/single-view
[20] M. Inc, 'Personalization'. [Online]. Available: https://www.mongodb.com/use-cases/personalization
[21] M. Inc, 'Catalog'. [Online]. Available: https://www.mongodb.com/use-cases/catalog
[22] M. Inc, 'Content Management'. [Online]. Available: https://www.mongodb.com/use-cases/content-management
[23] M. Inc, 'Reference Architecture: Mainframe Modernization'. Nov. 2022. [Online]. Available: https://www.mongodb.com/collateral/mainframe-modernization-reference-architecture
[24] M. Inc, 'MongoDB for Gaming'. [Online]. Available: https://www.mongodb.com/use-cases/gaming
[25] M. Inc, 'MongoDB CRUD Operations'. [Online]. Available: https://www.mongodb.com/basics/crud
[26] J. Hall, 'Getting Started with MongoDB & Mongoose'. [Online]. Available: https://www.mongodb.com/developer/languages/javascript/getting-started-with-mongodb-and-mongoose/