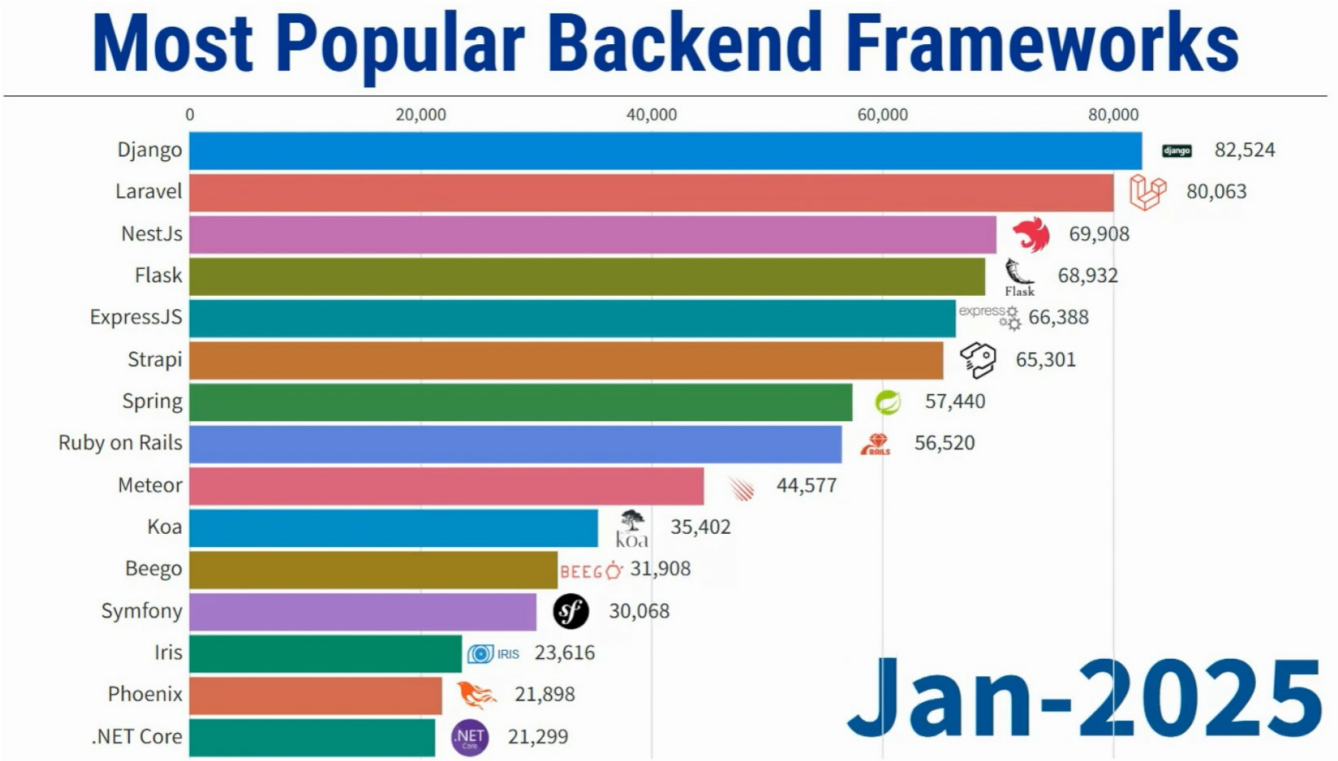
IMPLANTACIÓN DE APLICACIONES WEB

5- Programación una API usando NEST

Domingo López Oller

ÍNDICE

BACKEND

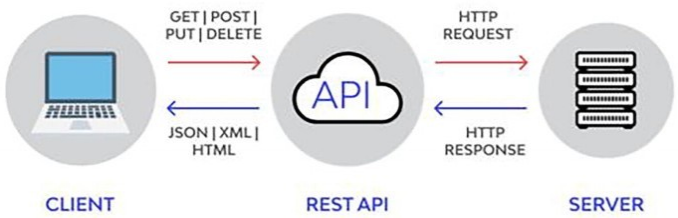


Top 10 de framework más utilizados

Tened en mente que el backend va a estar ligado a las características deseables de la aplicación

Vídeo: Evolución del uso de frameworks para Backend

API REST



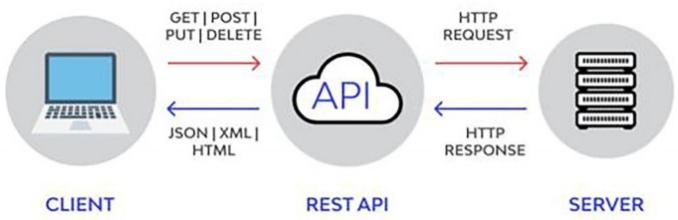
- API REST (Representational State Transfer) es un conjunto de reglas y convenciones para crear y consumir servicios web. El acrónimo REST fue creado por Roy Fielding mientras trabajaba en su tesis doctoral en la Universidad de California en Irvine en el año 2000.
- La idea central detrás de REST es que los recursos (html, archivos, imágenes, videos, etc.) deben ser identificados por un URI (Uniform Resource Identifier), y los clientes pueden acceder y manipular estos recursos mediante un conjunto predefinido de operaciones HTTP. El formato que se utiliza en las APIRest es JSON.
- Los métodos HTTP más comunes utilizados en las API son:
  - **GET**: utilizado para recuperar información del servidor. Esta solicitud no cambia nada en el servidor y se considera una operación "segura" ya que no modifica ningún dato en el servidor.
  - **POST**: utilizado para enviar información al servidor para que la procese. Esta solicitud puede crear o modificar datos en el servidor y no es considerada una operación "segura".
  - **PUT**: utilizado para actualizar información existente en el servidor. Esta solicitud puede crear nuevos datos si no existen y también puede modificar datos existentes. Es considerada una operación "no segura".
  - **DELETE**: utilizado para eliminar información del servidor. Esta solicitud es considerada una operación "no segura" ya que elimina información del servidor.

- Hasta ahora, hemos usado sólo la operación GET y utilizando la función propia de REACT con fetch

Para ver lo que sale de API puedes usar Thunder Client en VSCode o la de Postman

Práctica: Vamos a ver ejemplos con la api: https://jsonplaceholder.typicode.com/ y PokeAPI

API REST



- Las aplicaciones web tienen que permitir no sólo consultar sino también hacer otras operaciones como insertar, modificar, consultas individuales,… y esto puede hacerse sobre SGBD diferentes (MySQL, SQLite, Oracle, MongoDB,...)
- Si quiero usar las funciones GET, POST, PUT y DELETE en NEXTJS podría utilizar directamente fetch en código pero es buena práctica crear el fichero route.js dentro de una carpeta api en app.
- **Práctica:** crea una carpeta api/products dentro de app y en el fichero route.js incluye lo siguiente: export async function GET() { return new NextResponse("Listado de productos de la tienda");}
- Ahora incluye en la carpeta products la carpeta [id] y dentro de otro route.js incluye: export async function GET() { return new NextResponse("Detalle del producto");}
- **Práctica 2:** ¿Y si es de una API externa? Haz un route.js en api/pokemon con el caso de PokeAPI:

```
export async function GET() { // GET(request: Request, { params }: { params: { id: string } }
})
    try {
        //const { id } = params;
        const response = await fetch(`https://pokeapi.co/api/v2/pokemon?limit=10`);

        if (!response.ok) {
            return NextResponse.json( { error: 'Pokémon no encontrado' }, { status: 404 } );
        }

        const data = await response.json();
        return NextResponse.json(data, { status: 200 });
        } catch (error:any) {
        return NextResponse.json({ error: error.message }, { status: 500 });
    }
}
```

- Ten en cuenta que estos NextResponse son la respuesta que saldrá de la API para que el navegador la interprete. Para ver su resultado ahora, mira en ThunderClient al poner la ruta .../api/x. Ahora es tu aplicación la que gestiona esa operación GET y crear esa abstracción entre app y API
- Fijaros que es útil dar la salida junto con el status para que el navegador lo interprete

La operación fetch ya es una operación GET

API REST

AXIOS vs. {fetch API}

- **¿Sólo se puede usar fetch? NO, también está AXIOS para casos REACT**
  - **Axios** es una biblioteca diseñada para simplificar la tarea de solicitudes a API. Para su instalación hay que hacer: npm install axios. Hasta la versión 14 convivían ambos, siendo mejor axios, pero desde ahí para proyectos NEXTJS es mejor usar fetch.
  - Dos formas de hacer lo mismo y gestionando el error:

```
try{
  const response=fetch('https://api.example.com/data', {
  method: 'GET',
  headers: {
    'Content-Type': 'application/json',
  },
})

    if (!response.ok) {
      throw new Error('Error en la solicitud');
    }
    const data = await response.json();
    return NextResponse.json(response.data, { status: 200 });

} catch (error:any) { return NextResponse.json({ error:
error.message }, { status: 500 }); }
```

Esta es la forma más general de fetch y que usaremos

```
import axios from 'axios';
try {
      const response = await axios.get(`https://pokeapi.co/api/v2/pokemon/${id}`);
      return NextResponse.json(response.data, { status: 200 });
    } catch (error) {
      return NextResponse.json({ error: error.message }, { status: 500 });
    }
```

En ambos casos la ruta es conocida como **endpoint** que en el caso de axios se puede fijar con create:

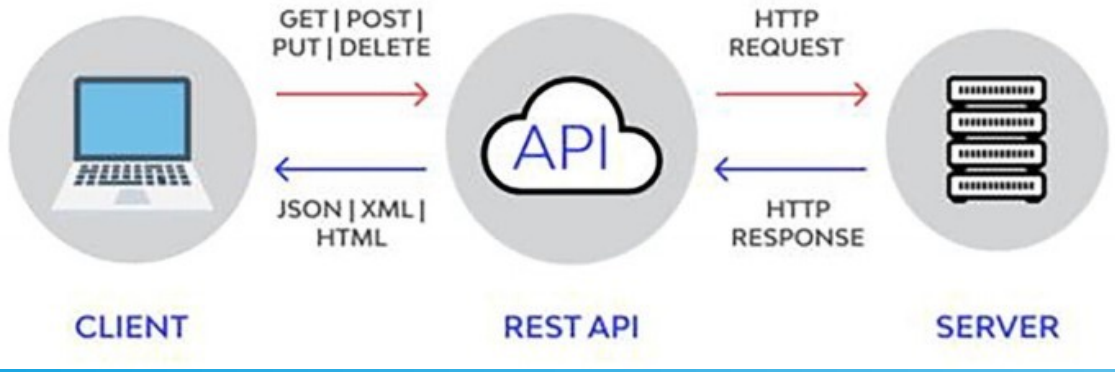
```
const client = axios.create({
  baseURL: "https://pokeapi.co/api/v2/pokemon"
});
Ahora en lugar de hacer axios.get, podremos hacer client.get("/1")
```

Ejemplo:

https://www.freecodecamp.org/espanol/news/como-usar-axios-con-react/

API REST

- **Práctica 3:**
  - Vamos a montar un pequeño ejemplo completo de API REST local para la gestión de tareas, donde podamos consultar, crear, borrar y actualizar un vector de cadenas de texto que serán las tareas.
  - Para la simulación vamos a usar tanto el navegador como Thunder Client
  - Después haremos el frontend con un formulario que realizará llamadas asíncronas y mostrará el estado del vector



API REST

- **Práctica 3:**
  - Fichero route.ts:
    - Consultar las tareas: GET
    - Crear nuevas tareas: POST
    - Crear nuevas tareas: PUT/PATCH
    - Crear nuevas tareas: DELETE

Como no tenemos una base de datos lo haremos con un ejemplo en RAM

```
import { NextResponse } from 'next/server';
export let tareas = [   { id: 1, texto: 'Aprender Next.js' },  { id: 2, texto: 'Crear un CRUD' }, ];

export async function GET() {
  return NextResponse.json(tareas,{ status: 200 });
}
```

```
export async function POST(request: Request) {
  const { texto } = await request.json();
  const nuevaTarea = { id: tareas.length+1, texto };
  tareas.push(nuevaTarea);
  return NextResponse.json(nuevaTarea, { status: 201 });
}
```

```
export async function PUT(request: Request) {
  const { id, texto } = await request.json();
  const tarea = tareas.find(t => t.id === id);
  if (tarea) {
    tarea.texto = texto;
    return NextResponse.json(tarea,{ status: 201 });
  }
  return NextResponse.json({ error: 'Tarea no encontrada' }, { status: 404 });
}
```

Esta request es crucial para recibir el json con el que poder crear la nueva tarea o poder encontrar la tarea a modificar/borrar

```
export async function DELETE(request: Request) {
  const { id } = await request.json();
  const index = tareas.findIndex(t => t.id === id);
  if (index !== -1) {
    tareas.splice(index, 1);
    return NextResponse.json({ message: 'Tarea eliminada' },{ status: 201 });
  }
  return NextResponse.json({ error: 'Tarea no encontrada' }, { status: 404 });
}
```

Debate: ¿Hace falta definir todas las funciones para nuestras aplicaciones?

API REST

- **Práctica 3:**
  - Fichero page.tsx: Vamos a incluir el formulario siguiente
  - Crea una página nueva y dentro del return escribe esto… (Tailwindcss)

```
<div className="container mt-5" style={{ maxWidth: '500px' }}>
    <h1 className="mb-4 text-center">Lista de Tareas</h1>
    <form onSubmit={handleSubmit} className="mb-3 d-flex">
     <input   type="text"  value={nuevaTarea}
      onChange={(e) => setNuevaTarea(e.target.value)}
      className="form-control me-2"
      placeholder="Nueva tarea"
     />
     <button type="submit" className="btn btn-success">  Agregar  </button>
    </form>

    <ul className="list-group">
     {tareas.map(tarea => (
      <li key={tarea.id}  className="list-group-item d-flex justify-content-between align-items-center" >
       <div className="flex-grow-1 me-2">
        {tareaEditando?.id === tarea.id ? (
         <input type="text" value={tareaEditando.texto}  onChange={(e) =>
           SetTareaEditando({...tareaEditando, texto: e.target.value })
          }
          className="form-control"
         />
        ) : ( tarea.texto  )}
       </div>
...
```

Añade esto al comienzo:

```
'use client'
import { useEffect, useState } from 'react'
type Tarea = { id: number texto: string}
```

```
...
<div className="btn-group">
          {tareaEditando?.id === tarea.id ? (
           <button  onClick={() => handleEditar(tarea)}
            className="btn btn-success btn-sm">    Guardar     </button>
          ) : (
           <button  onClick={() => setTareaEditando(tarea)}
            className="btn btn-warning btn-sm"    >   Editar      </button>
          )}

          <button  onClick={() => handleEliminar(tarea.id)}
           className="btn btn-danger btn-sm" >        Eliminar     </button>
        </div>
       </li>
      ))}
     </ul>
    </div>
```

Acciones de los botones

- **Agregar:** Hará la inserción de la tarea como Submit
- **Editar:** Habilitará que se cambie el texto y aparecerá el botón de Guardar
- **Eliminar:** Borrará la tarea

API REST

- **Práctica 3:**
  - Fichero page.tsx: Vamos a leer las tareas al inicio
  - Queremos que al cargar la página ya muestre las tareas que haya hasta ese momento de manera asíncrona. Pon antes del return lo siguiente:

```
const [tareas, setTareas] = useState<Tarea[]>([])
const [nuevaTarea, setNuevaTarea] = useState<string>('')
const [tareaEditando, setTareaEditando] = useState<Tarea
| null>(null)


useEffect(() => {
  fetchTareas()
}, [])


const fetchTareas = async (): Promise<void> => {
  const res = await fetch('/api/tareas')
  const data: Tarea[] = await res.json()
  setTareas(data)
}
```

Fíjate que en el fetch estás indicando TÚ ruta a la API para que realice la acción de lectura. Tal y como lo harías en Thunder Client. Luego en la ruta /tareas en app es donde utilizas ese resultado en page.tsx

El resultado se tiene que pasar a formato JSON para colocarlo en la variable de estado correspondiente y mostrar las tareas en el formulario.

Importante: Estamos usando llamadas asíncronas desde cliente para que las acciones se vayan haciendo a la par que se gestionan desde el Backend, si quisiéramos que se hiciera sólo desde el servidor, se utilizan Server actions, pero habría que refrescar la página por cada click

API REST

- **Práctica 3:**
  - Fichero page.tsx: Incluir las funciones de los botones
  - Vamos a incluir las funciones que se llaman con los botones. Pon antes del return lo siguiente:

Normalmente cuando se obtienen datos no hace falta incluir esta parte con el método GET, pero para el resto de operaciones sí que hay que indicarle tanto el método, como el body con el contenido con el que hay que trabajar en la request

```
const handleSubmit = async (e:
React.FormEvent<HTMLFormElement>) => {
    e.preventDefault()
    if (!nuevaTarea.trim()) return

    const res = await fetch('/api/tareas', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ texto: nuevaTarea })
    })

    const nuevaTareaObj: Tarea = await res.json()
    setTareas([...tareas, nuevaTareaObj])
    setNuevaTarea('')
  }
```

```
const handleEditar = async (tarea: Tarea): Promise<void> => {
  if (!tareaEditando) return
  const res = await fetch('/api/tareas', {
    method: 'PUT',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({
      id: tarea.id,
      texto: tareaEditando.texto
    })
  })
  const tareaActualizada: Tarea = await res.json()
  setTareas(
    tareas.map(t => t.id === tarea.id ? tareaActualizada : t)
  )
  setTareaEditando(null)
}
```

```
const handleEliminar = async (id: number):
Promise<void> => {
    await fetch('/api/tareas', {
      method: 'DELETE',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ id })
    })

    setTareas(tareas.filter(t => t.id !== id))
  }
```

Una vez hecho esto sólo queda probar la aplicación a ver qué hace.
¿Y podemos hacer esto con rutas dinámicas? ¿Es decir, que al colocar tareas/1 veamos sólo esa y según la operación la podamos borrar?

API REST

- **Práctica 4: Tratar API con rutas anidadas**
  - Hay que crear la ruta [id] con su page.tsx y la correspondiente en api/tareas/[id]/route.js al igual que se hace en la carpeta tareas de NEXT
  - Primero haremos la ruta dinámica en next y el fichero page.tsx

```
'use client'
import { useState, useEffect } from 'react';
import Link from 'next/link';
type Tarea = {id: number; texto: string; }
type PageProps = {  params: { id: string };}

export default function Page({ params }: PageProps) {
  const [tarea, setTarea] = useState<Tarea | null>(null);
  const [error, setError] = useState<string | null>(null);
  const [eliminada, setEliminada] = useState(false);
  useEffect(() => {
    const fetchTarea = async () => {
      try {
        const { id } = await params;
        const res = await fetch(`/api/tareas/${id}`);
        if (!res.ok) {
          throw new Error('Tarea no encontrada');
        }
        const data: Tarea = await res.json();
        setTarea(data);
      } catch (err: any) {
        setError(err.message || 'Error desconocido');
      }
    };    fetchTarea();  }, [params]);
...
```

```
  const borrarTarea = async () => {
    try {
      const { id } = await params;
      const res = await fetch(`/api/tareas/${id}`, {
        method: 'DELETE',
      });
      if (!res.ok) {
        throw new Error('No se ha podido borrar la tarea');
      }
      setEliminada(true);
    } catch (err: any) {
      setError(err.message || 'Error desconocido');
    }
  }
  if (error) {
    return (
      <div className="container mt-5">
        <h1 className="text-danger">{error}</h1>
        <Link href='/tareas' className="btn btn-primary mt-3">
Volver atrás</Link>
      </div>
    )
  }
  ...
```

```
...
  if (eliminada) {
    return (
      <div className="container mt-5">
        <h1 className="text-success">Se ha eliminado
correctamente</h1>
        <Link href='/tareas' className="btn btn-primary mt-
3">Volver atrás</Link>
      </div>
    )
  }

  if (!tarea) return <div className="container mt-
5">Cargando...</div>;

  return (
    <div className="container mt-5">
      <h1>{tarea.id} - {tarea.texto}</h1>
      <button
        onClick={borrarTarea}
        className="btn btn-danger mt-3"
      >
        Borrar Tarea
      </button>
    </div>
  )
}
```

API REST

- **Práctica 4: Tratar API con rutas anidadas**
  - Ahora crearemos el correspondiente fichero route.jsx en api/tareas/[id]/route.js

```
import { NextResponse } from 'next/server';
import { tareas } from '../route';
type PageProps = {
  params: { id: string };
}

export async function GET(request:Request,{params}:PageProps) {
    const {id} = await params;
    console.log(tareas);
    const tarea = tareas.find(tarea => tarea.id === parseInt(id));
    if (tarea) {
      return NextResponse.json(tarea);
    }
    return NextResponse.json({ error: 'Tarea no encontrada' }, { status: 404 });
}
export async function DELETE(request:Request,{params}:PageProps) {
    const { id } = await params;
    const index = tareas.findIndex(t => t.id === parseInt(id));
    if (index !== -1) {
      tareas.splice(index, 1);
      return NextResponse.json({ message: 'Tarea eliminada' }, { status: 201 });
    }
    return NextResponse.json({ error: 'Tarea no encontrada' }, { status: 404 });
  }
```

Vamos a probar la aplicación completa. Fíjate que ahora hay que indicar el parámetro request y params en todas las funciones. En este caso sólo es necesario params para conocer id, con request podemos tener datos para las operaciones como ya ocurre en el caso anterior en los métodos PUT y POST.

Acabamos de “crear” una API para la gestión de tareas utilizando un JSON en NEXT, sin embargo, como todo se hace en el navegador y no hay persistencia, hay cosas que no funcionarán bien. Ten en cuenta que lo normal es acceder a API que están conectadas a bases de datos que pueden estar en MySQL, Oracle, MongoDB,…

Ahora bien, el objetivo que se persigue es que puedas acceder a los datos con la API sin saber qué base de datos hay debajo. Piensa que antes había que hacer el acceso a la base de datos en cuestión y procesarlo en tu código, Por ejemplo: ttradicionalmente en PHP se ha hecho así: (**enlace**)

Para el desarrollo de este backend utilizaremos **NESTJS**

API REST

- ¿Qué queremos plantear con NESTJS?

La idea es plantear un modelo MVC similar a como hace ANGULAR de manera que damos un servicio de una API y abstraemos al usuario de la base de datos que tiene debajo.

SGBD	Backend (NEST)
Relacional (**Oracle**) • entidad → **Tabla** • instancia → **registro** No Relacional (**MongoDB**) • entidad → **Colección** • instancia → **documento**	Directorio (cars) • modulo → cars.module.ts (centralizar recursos) • controlador → cars.controller.ts (recibir las peticiones url) • servicio → cars.service.ts (enviar la petición aL SGBD) • interfaces/dto/entity → cars.interface.ts

http.//localhost/cars —> **get** → lista todos

http.//localhost/cars/1 —> **get** → detalle de un registro

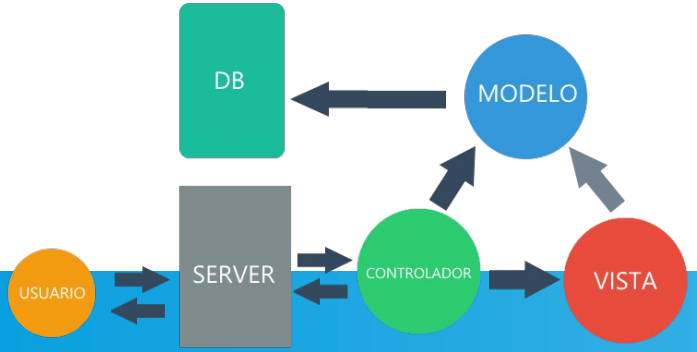
http.//localhost/cars —> **post** → crear un registro http.//localhost/cars/new

http.//localhost/cars/1 —> **patch/put** → actualizar un registro http.//localhost/cars/update/1

http.//localhost/cars/1 —> **delete** → borrar de un registro http.//localhost/cars/delete/1

API REST

- ¿Qué es el Modelo-Vista-Controlador?
  - El MVC es un diseño de software para implementar interfaces de usuario en ordenadores que separan la entrada, el procesamiento y la salida de una aplicación a partir de definir sus componentes:
    - **Modelo:** Se encarga de los datos, generalmente consultando la base de datos con actualizaciones, consultas, búsquedas, etc.
    - **Controlador:** Recibe las órdenes del usuario y se encarga de solicitar los datos al modelo y de comunicárselos a la vista para que los muestre.
    - **Vistas:** Son la representación visual de los datos, todo lo que tenga que ver con la interfaz gráfica va aquí. Ni el modelo ni el controlador se preocupan de cómo se verán los datos.

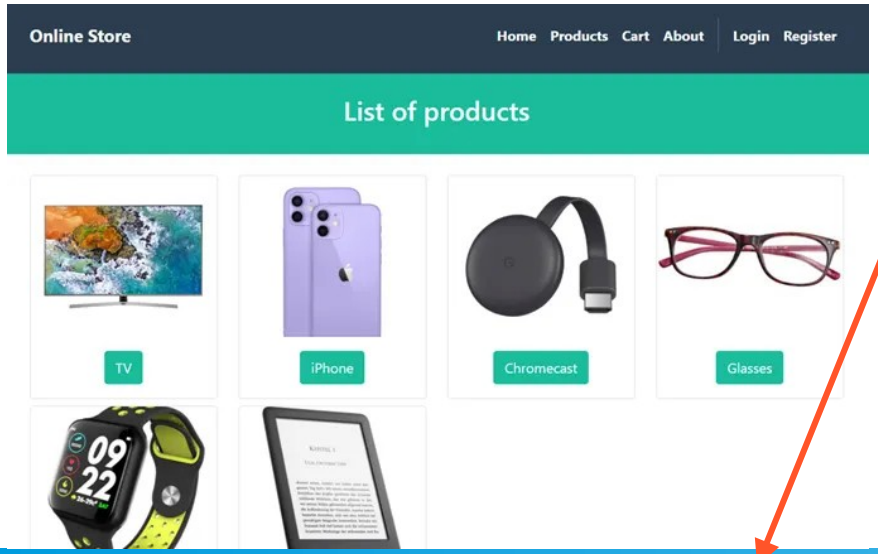


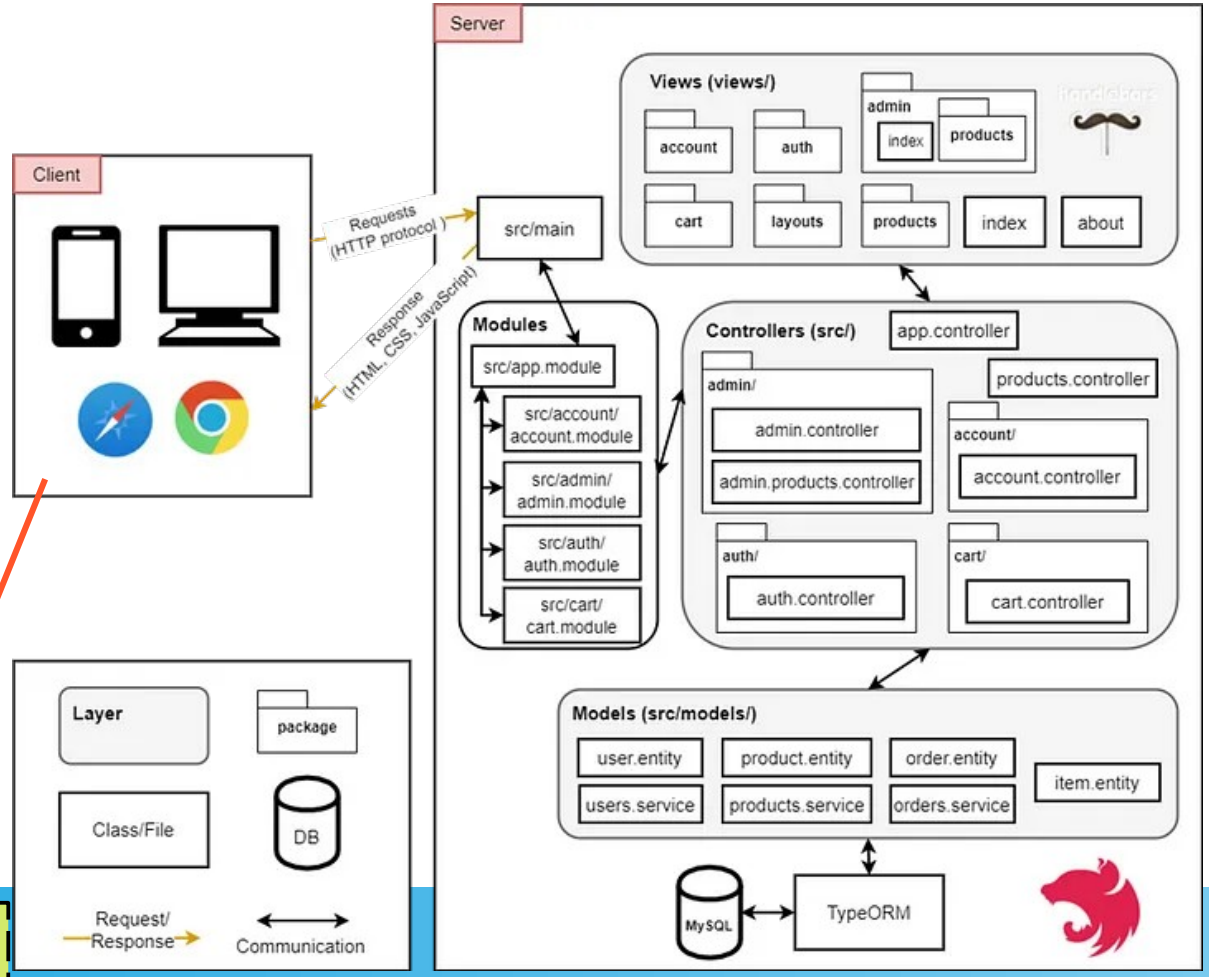
FUNCIONAMIENTO MVC

- El usuario solicita una acción al servidor
- El servidor atiende la petición y manda a llamar al controlador
- El controlador llama al modelo necesario
- El modelo atiende la petición y realiza las operaciones de datos correspondientes
- El modelo regresa el resultado
- El controlador llama a la vista, enviándole los datos procesados del modelo
- La vista presenta los datos
- El controlador devuelve la vista al servidor
- El servidor presenta el resultado al cliente

API REST

- Ejemplo de MVC para una tienda online en NESTJS (enlace)





Como NESTJS es backend, la parte de la vista la realizará NEXTJS con las respuestas JSON

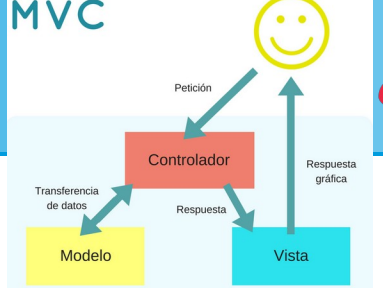
NESTJS

Vídeo tutoriales NESTJS: (vídeo 1, vídeo 2, vídeo 3)

- NESTJS Es un framework de backend para el desarrollo de API sencillo desarrollado por Kamil Mysliwiec en 2017 y que está fuertemente inspirado en Angular y que emplea TypeScript. Kamil es experto en desarrollo de Google y amante de los gatos (blog).
- Nest es un Framework Backend que está basado en Node Express (o Fastify) que facilita la creación de los Building blocks (componentes, servicios, Exception Filters, Módulos, Decoradores,…) a través de su consola cliente.
- NestJS tiene como objetivo liberar al desarrollador de tareas comunes y repetitivas, ofrecer una arquitectura sólida para aplicaciones mantenibles y permitir preocuparse más del desarrollo y menos del setup de los proyectos.
- Además, Nest.js mantiene una documentación detallada, y su comunidad de desarrolladores y colaboradores es muy activa y está preparada para responder a los problemas sobre la marcha. (enlace)

¿Quién usa NESTJS?
(enlace, enlace)

NESTJS



- **Elementos de la arquitectura NESTJS más destacados:**
  - **Módulos:** En Nest.js, cada aplicación está dividida en módulos. Estos son conjuntos de componentes, controladores y servicios relacionados que trabajan juntos para realizar una tarea específica.
  - **Controladores:** Los controladores son clases que manejan las solicitudes HTTP y devuelven una respuesta. Los controladores se asignan a rutas específicas. Esto significa que sólo manejarán las solicitudes que coincidan con esa ruta.
  - **Servicios:** Los servicios son clases que contienen lógica de negocio y pueden ser inyectadas en controladores o en otras clases de servicio. Los servicios se utilizan a menudo para interactuar con bases de datos o realizar otras tareas que no son específicas de la lógica de presentación.
  - **Pipes:** Los pipes son clases que se pueden utilizar para transformar o validar los datos que entran o salen de una aplicación. Los pipes se pueden aplicar a los argumentos de los métodos de controlador o a los valores de los campos de entrada y salida de una solicitud o respuesta HTTP. (Vídeo)
  - **Guards:** Son componentes que se utilizan para proteger las rutas de acceso no autorizado. Los Guards se pueden utilizar para verificar si un usuario tiene permisos para acceder a una ruta determinada y para tomar medidas adecuadas si no es así.
  - **Middleware:** El middleware es código que se ejecuta entre la llegada de una solicitud y la ejecución del controlador correspondiente. Se puede utilizar para realizar tareas como la autenticación o la validación de datos antes de que lleguen al controlador.(Vídeo)
  - **Decoradores:** Los decoradores son funciones que se aplican a clases, métodos o propiedades y modifican su comportamiento. Nest.js incluye varios decoradores predefinidos que se pueden utilizar para realizar tareas como la asignación de rutas a controladores o la inyección de dependencias.

NESTJS

- **Primer proyecto NESTJS:**
  - **Instalación del cliente:** npm i -g @nestjs/cli (comprueba la versión nest -v y la ayuda con nest --help)
  - **Creación del primer proyecto:** nest new [CARPETA]

Va a crear un proyecto con su propio .git

- A diferencia de NEXTJS, iniciamos el proyecto o con **npm run start:dev** (usaremos esta por tener el --watch) y accedemos con http://localhost:3000 para ver que sale Hello world!
- Antes de explicar más sobre el desarrollo de una API en NESTJS, vamos a intentar conectar NEXTSJS (frontend) y NESTJS (backend) y ver algunos errores que irán saliendo
- **Práctica:** Crea la siguiente page.tsx en el proyecto NEXTJS. Antes de nada, en el proyecto NESTJS en src/main.ts cambia el puerto a 3001 y reinicia. Si no lo haces te dará fallo o abrirá un puerto diferente al estar ocupado.

```
'use client'
import { useState, useEffect } from 'react'
export default function Home() {
  const [message, setMessage] = useState<string>('Cargando…')
  useEffect(() => {
    const fetchMessage = async () => {
      try {
        const res = await fetch('http://localhost:3001')
        const data = await res.text()
        setMessage(data)
      } catch { setMessage('Error al cargar el mensaje') }
    }
    fetchMessage()
  }, [])
  return ( <p>Mensaje del servidor: {message}</p> )
}
```

Ahora ten arrancados tanto el servidor NEST como NEXT y en la ruta mira el resultado.

¿No funciona verdad? ¿Por qué?

Fichero src/main.ts

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

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


  await app.listen(process.env.PORT ?? 3001);
}
bootstrap();
```

NESTJS

- **¿Por qué se produce este fallo?** No es un fallo, ya que los navegadores implementan una política de seguridad llamada "**Same-Origin Policy**" (Política del Mismo Origen). Esta política restringe cómo un documento o script cargado desde un origen puede interactuar con recursos de otro origen. Es decir, dado que son servidores diferentes y actúan en puertos diferentes, se consideran recursos de diferente origen. Esto por ejemplo no pasaría si fuera el mismo puerto, como es el caso cuando se usa el servidor Apache para aplicaciones con PHP.

- Para arreglar esto, hay que habilitar en NEST los diferentes orígenes que puede atender habilitando **CORS (Cross-Origin Resource Sharing)**. En nuestro caso, en desarrollo, bastará con incluir la línea app.enableCors(); en el fichero src/main.ts

- En el caso de producción, habrá que indicar los diferentes orígenes de consulta e incluso el tipo de operación que está permitido realizar. No es conveniente usar en origen '*' por criterios de seguridad. Este no es el caso de PokeAPI, que seguro que de tenerla en NEST usará origin: '*' y de methods:'GET'

Fichero src/main.ts

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

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

  app.enableCors({
    origin: ['http://localhost:3000'],
    methods: ['GET', 'POST', 'PUT', 'DELETE', 'PATCH', 'OPTIONS'],
    allowedHeaders: ['Content-Type', 'Authorization'],
    credentials: true,
  });

  await app.listen(process.env.PORT ?? 3001);
}
bootstrap();
```

Ahora sí deberías ver el mensaje Hola mundo desde NEXTJS con http://localhost:3000

NESTJS

- **Ficheros más importantes en NESTJS:**
  - **nest-cli.json:** Contiene la configuración de la consola cli de nest.
  - **tsconfig.json:** Contiene la configuración de TS para este proyecto de NEST.
  - **.env:** El archivo .env se utiliza para almacenar variables de entorno en tu aplicación. Es una práctica común para manejar configuraciones sensibles o específicas del entorno sin exponerlas en el código fuente. Por ejemplo cuando accedamos a una base de datos, no queremos que en GitHub se suba el acceso con el usuario y clave en texto plano. Este fichero no es exclusivo de NESTJS, se puede usar también en NEXTJS
  - **Carpeta src:** contiene nuestro código fuente que contiene inicialmente un ejemplo en app y main.ts
    - **main.ts:** Contiene la importación del core de NestJS y el módulo principal de la aplicación (app.module). Luego realiza el propio arranque de la aplicación con la función bootstrap() y desde el puerto 3000
    - **app.module.ts:** Está precedida de un decorador @module, que es el que hace que esta clase se comporte como un módulo de aplicación
    - **app.controller.ts:** los controladores son las piezas de software que se encargan de gestionar las solicitudes, realizando todo el trabajo necesario para gestionar el request y componer la respuesta.
    - **app.service.ts:** está decorado con @injectable(). Básicamente este decorador permite que este servicio se pueda enviar al constructor de los controladores, mediante la inyección de dependencias que nos ofrece NestJS.y proporciona las funciones que son llamadas en el controlador
    - **app.controller.spec.ts:** Se crea para la realización de las pruebas del MVC. Fíjate que hay un directorio test y una forma de llamar a las pruebas. Cambia la salida en app.service.ts y haz **npm run test** para ver la salida.

NESTJS

- **Ficheros más importantes en NESTJS:**
  - El fichero **.env** es el único que no está creado en un principio. Hay que añadirlo al fichero gitignore (ya suele estarlo) o llamarlo también como .env.local para que no se suba al repositorio. Un ejemplo del contenido sería:

```
DATABASE_URL=mysql://usuario:contraseña@localhost:3306/mi_base_de_datos
API_KEY=mi_clave_secreta
PORT=4000
```

  - Tanto NEXT como NEST hacen uso de estas variables de entorno a lo largo del código.
    - Se puede tener acceso a estas variables según sea servidor o cliente como:
      - **Acceso desde servidor:** Hay que configurar el acceso para acceder como: process.env.DATABASE_URL
      - **Acceso desde cliente:** Basta con añadir el prefijo NEXT_PUBLIC_ a cada variable en .env para acceder como: process.env.NEXT_PUBLIC_DATABASE_URL
    - ¿Cómo lo usamos en NEST? Hay que seguir unos pasos:
      - npm install @nestjs/config
      - Añadir ConfigModule en app.module.ts
      - Acceder a la variable

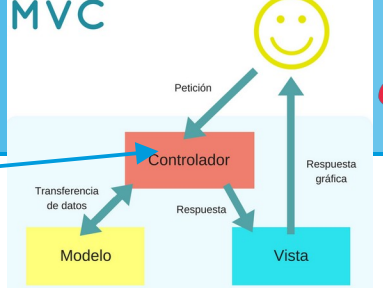
main.ts

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

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

```
import { Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';
import { AppController } from './app.controller';
import { AppService } from './app.service';

@Module({
  imports: [
    ConfigModule.forRoot({isGlobal: true,}),
  ],
  controllers: [AppController],
  providers: [AppService],
})

export class AppModule {}
```

Si reinicias el servidor NEST verás que ahora hay que poner http://localhost:4000

NESTJS



- **El controlador:**
  - Son la primera pieza que vamos a usar en el desarrollo de aplicaciones en Nest. Implementan las rutas de las aplicaciones web y permiten trabajar con la solicitud y preparar la respuesta que será enviada al navegador o al cliente que consuma las API REST
  - **Formas para crear un controlador:** **nest generate controller** (ruta en src) / nest g co (ruta en src) Ej: nest g co products
  - Es posible crear el controlador sin la opción de test (fichero .spec): nest g co products --no-spec
  - Automáticamente se ha enlazado el nuevo controlador en el fichero app.module.ts que gestiona la aplicación.
  - Vamos a ver la diferencia entre el controlador vacío que hemos creado y el que viene por defecto en app:

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

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

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

Muchos lenguajes utilizan decoradores @x

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

@Controller('products')
export class ProductsController {}
```

¿¿Esto es importante??

El controlador va a atender peticiones por la ruta que le indiquemos como parámetro. Es decir, el controlador products sólo atenderá peticiones a partir de la ruta /products. Esto se conoce como **endpoint** products

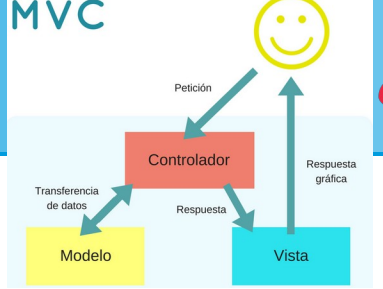
Añade esto dentro de la clase para probar la ruta / y /products:

```
@Get()
getHelloInProducts(): string {
  return "Estamos en productos!!!";
}
```

El nombre de la función da igual, en cuanto pongas /products se ejecuta el decorador GET.

Práctica: ¿Probamos a poner un controlador dentro de otro a ver si así hay rutas como NEXT?
Debate: ¿Por qué no sale el resultado detalle?

NESTJS



- **El controlador:**
  - **¿Cómo incluyo otras rutas dentro de products o los detalles del producto/1?** Indicando esta ruta en @GET o el decorador correspondiente. Eso sí, cuidado con el orden en el que los ponemos!

```
@Get('hot')
   GetSpecialProducts():string {
      return "Te vamos a mostrar los productos más calientes!!";
   }
@Get(':id')
   find( @Param() params):string {
      return `Estás consultando el producto ${params.id}`;
   }
```

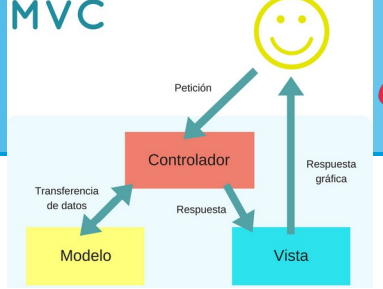
Práctica: Incluye este código tal cual en el controlador de products y mira el resultado. Después cambia el orden de estos @GET y vuelve a probar las rutas en navegador y ThunderClient

En la función find() o como la llamemos, le indicamos los parámetros de la ruta y así se puede acceder a params.id

  - **¿Y si hubiera recibido más de un parámetro en la ruta?** Ej: products/1/grande

```
interface datos {id:string,size:string} //tras import
// Opción 2: type datos= {id:string,size:string}
@Get(':id/:size')
findWithSize( @Param() params:datos):string {
return `En esta ruta obtenemos el producto ${params.id}, pero
en su tamaño ${params.size}`;
}
//Si no conocemos el tipo se pone :any o :unknown
```

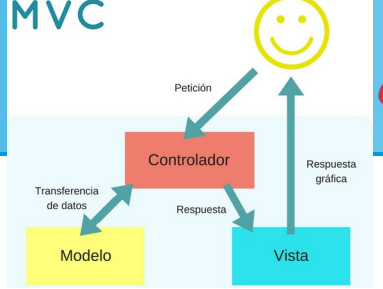
NESTJS



- **El controlador:**
  - En el caso anterior recibía todos los parámetros y ya iba seleccionando en el cuerpo de la función. También puedo **desestructurar el decorador @Param** para acceder sólo a los parámetros que necesito. Por ejemplo, del decorador @Param extrae sólo los parámetros que necesito.
  - Cualquiera de los dos métodos es correcta

```
@Get('hot')
   getSpecialProducts():string {
      return "Te vamos a mostrar los productos más calientes!!";
   }
@Get(':id')
   find(@Param('id') id:string):string {
      return `Estás consultando el producto ${id}`;
   }
   @Get(':id/:size')
   molaNest(@Param('id') id:number, @Param('size') size:string):string {
      return `En esta ruta obtenemos el producto ${id}, pero en su tamaño ${size}`;
   }
```

NESTJS



- **El controlador:**
  - A través del ejemplo anterior hemos visto el método GET, vamos a ir viendo el resto de métodos que conforma el CRUD de una API (POST, PUT, DELETE)
  - **Método básico de @Post**

```
@Post()
CreateProduct():string {
return 'Estamos atendiendo una solicitud de tipo Post';
}
```

  - POST es un método muy común para operaciones de formularios, pero ¿Cómo se reciben y procesan esos datos? Antes teníamos el @Param de la ruta y ahora **@Body para el request**

```
@Post()
createProduct(@Body() body:any):string{
return `Creo un producto ${body.articulo} con precio ${body.precio}`;
}
```

Cuando hagas pruebas de desarrollo, puede ser útil hacer return body; para ver qué estas recibiendo

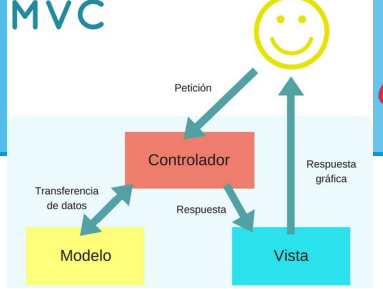
Práctica: Prueba el nuevo método incluyendo en el body de ThunderClient lo siguiente →

```
{
  "articulo":"ordenador",
  "precio":"475,99"
}
```

Después cambia los nombres de los campos a ver qué pasa.

Práctica: Cambia la función @Post() para que reciba body con el tipo interfaz y el return correspondiente. Ya veremos que esto lo haremos con un DTO

NESTJS



- **El controlador:**
  - Cuando ejecutes el caso anterior en ThunderClient verás algo como esto:

POST localhost:3001/products Send

Query Headers 2 Auth Body 1 Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1  {
2    "articulo":"ordenador",
3    "precio":"475,99"
4  }
```

Status: 201 Created Size: 44 Bytes Time: 7 ms

Response Headers 7 Cookies Results Docs {}

```
1  Creo un producto ordenador con precio 475,99
```

Copy

¿Qué pasaría si la estructura de Body no está completa? Lo veremos más adelante con los **DTO** que se encargará de validar la entradas y generar el 404 en caso de error

Práctica: ¿Cambia la ejecución de ThunderClient tras cambiar lo siguiente? Prueba otros

HttpCode establece un estado base que puede cambiar en el resultado de la función con status

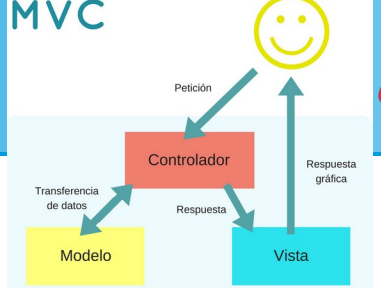
```
@Post()
@HttpCode(HttpStatus.NO_CONTENT)
createProduct(@Body() body) {
    return body;
}
```

Práctica: ¿Puede utilizarse param y body a la vez? SÍ

Práctica: También puedes hacer todo en return como: return {statusCode: HttpStatus.OK,body}

Si te fijas, hay un status de operación y ya hicimos uso de esto cuando creamos las funciones CRUD en NEXTJS. En este caso con NEST es más sencillo, basta con agregar tras el decorador @Post el decorador @HttpCode(código) o usar @HttpCode(HttpStatus.estado) siendo estado alguno de estos (enlace)
Nosotros podemos indicar el código devuelto aunque no es obligatorio, ya NEST genera el que corresponda

NESTJS



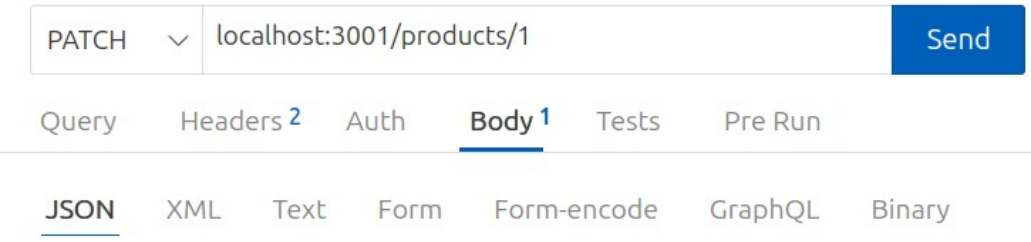
- **El controlador:**
  - ¿Cómo puedo actualizar un producto indicado en la ruta? Ej: producto/1
  - **Método básico @PUT:**

Fíjate que estamos accediendo al parámetro id de la URL y leyendo lo que hay en el Body

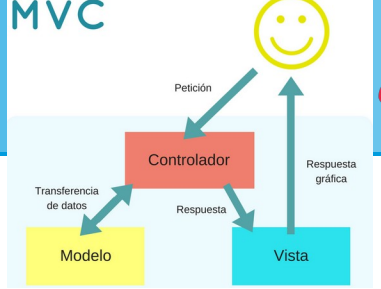
```
@Put(':id')
    update(@Param('id') id: number, @Body() body:any):string {
        return `Estás haciendo una operación de actualización del recurso ${id}
con artículo: ${body.articulo} y precio: ${body.precio}`;
    }
```

  - Hay otro método para actualizar que es el conocido como **@PATCH**, pero este se usa para actualizaciones parciales: (ver vídeo que explica cuándo usar cada uno)

```
@Patch(':id')
partialUpdate(@Param('id') id: number, @Body() body:any):string {
    return `Actualización parcial del ítem ${id}`;
}
```

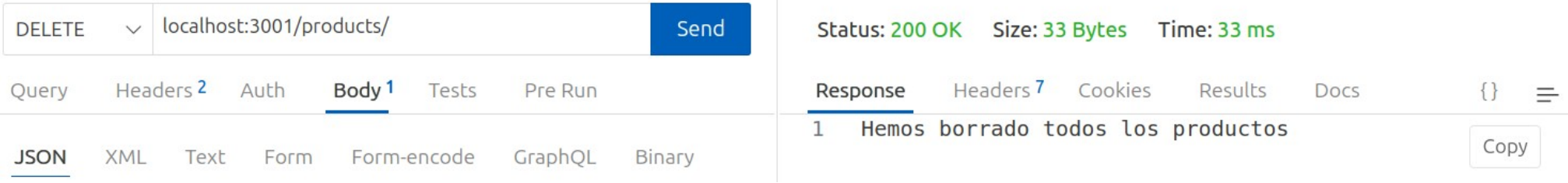


NESTJS

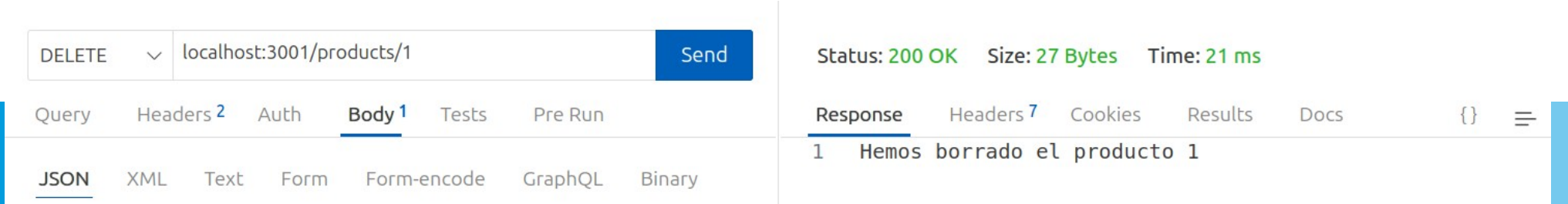


- **El controlador:**
  - ¿Cómo puedo borrar todos los productos o uno en particular? Ej: Borrar products/1
  - **Método básico @DELETE:**

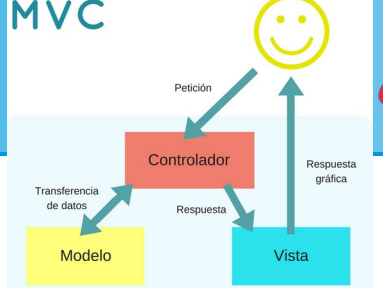
```
@Delete()
delete():string {
    return `Hemos borrado todos los productos";
}
```



  - **Práctica:** ¿Qué habría que cambiar para que ahora saliera que se está eliminando sólo el producto indicado?



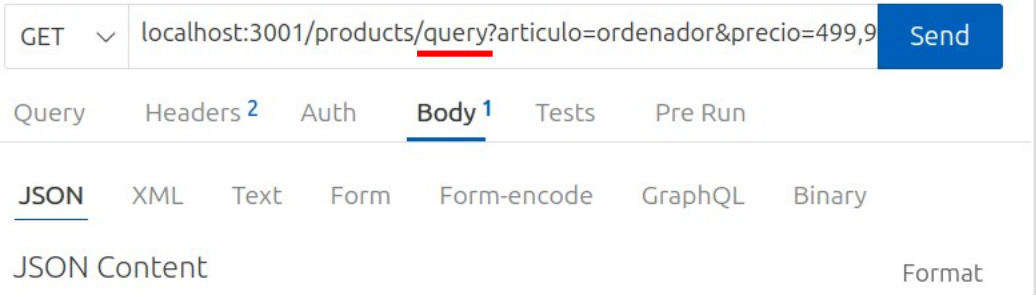
NESTJS

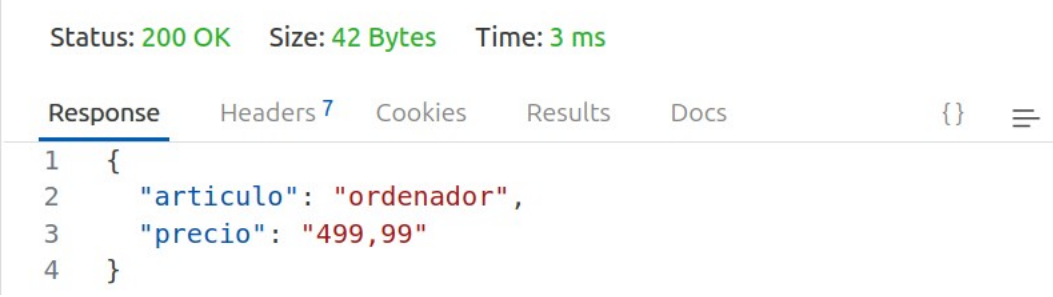


- **El controlador:**
  - **¿Puedo obtener los productos filtrados por contenido en lugar de sólo por su id?** Sí, haciendo algo similar a como hace Google con una consulta y que pone en la URL algo como https://www.google.es/search?q=que+es+nest o en formulario empleando GET que tiene un formato como http://example.com?dato1=valor1&dato2=valor2. Fíjate que ambas tienen una ruta, luego '?' y finalmente los datos a buscar
  - Para esto vamos a procesar el decorador @Query() y ver este ejemplo:

Recuerda de ponerlo antes del @Get(':id')

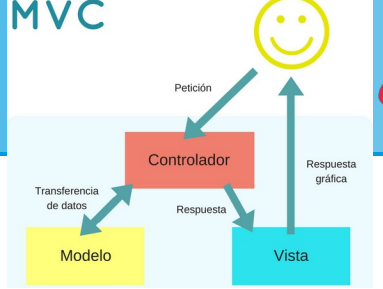
```
@Get('query')
rutaQuery(@Query() consulta){
    return consulta;
}
```





Al igual que hicimos con @Param, podemos acceder a los correspondientes campos de @Body ya sea por separado o como JSON como: return `El producto ${consulta.articulo} cuesta ${consulta.precio} €`;

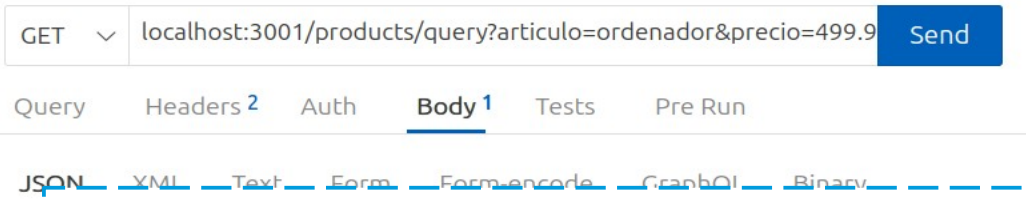
NESTJS



- **El controlador:**
  - **¿Puedo obtener los productos filtrados por contenido en lugar de sólo por su id? (cont.)**
  - Es conveniente que los datos que leemos por la consulta deben ser correctos, es decir, tenemos que validar lo que recibimos antes de lanzar una consulta a una base de datos con el tiempo que esto conlleva.
  - Por ejemplo: El valor en precio debe ser un número real y no una cadena de texto. **Aquí usas PIPE**

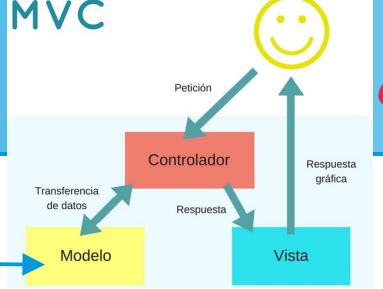
```
@Get('query')
    rutaQuery(@Query('articulo') articulo:string, @Query('precio',ParseFloatPipe) precio:number) :string{
        return `El producto ${articulo} cuesta ${precio} €`;
    }
```

  - Este es un ejemplo de transformación de tipo realizado por Pipes pero también se usan como mecanismo de validación y no es el único (enlace). Prueba este ejemplo con un precio como 499,99



Actividad: Crea un controlador y sus operaciones CRUD para una tienda de pinturas con los atributos: producto, marca, precio. Incluye la operación query para que diga algo como… “Buscando las pinturas de color azul”

NESTJS



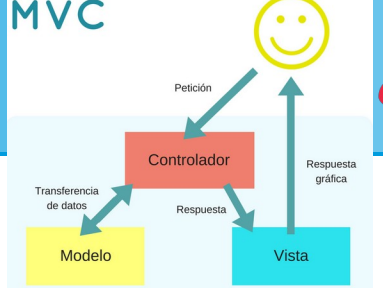
- **El servicio:**
  - Los servicios o providers son los encargados de realizar las “acciones” que se solicitan en el controlador sobre la base de datos. Se encargan de la lógica de negocio.
  - Fíjate en el ejemplo base que viene con app.service.ts, cómo hace uso de él en app.controller.ts y cómo aparece en app.module.ts
  - ¿Cómo se crea un servicio en NEST? **nest generate service products** o nest g s products
  - Fíjate que el servicio se añade a app.module.ts como ya pasó con el controlador
  - El código base de un servicio es el siguiente:

```
import { Injectable } from '@nestjs/common';
@Injectable()
export class ProductsService {}
```

  - Está vacío de funcionalidad, pero gracias al decorador @Injectable permitirá que se pueda usar en el controlador. Para ello hay que añadir el servicio al constructor del controlador como:

```
import { ProductsService } from './products.service';

@Controller('products')
export class ProductsController {
constructor(private readonly productsService: ProductsService) { }
// ...
```

NESTJS



- **El servicio:**
  - De momento vamos a trabajar con un array local que llamaremos **products** al que le iremos haciendo las consultas del controlador. Por ejemplo, vamos a crear las funciones básicas para ver todos los productos e insertar uno nuevo.

```
import { Injectable } from '@nestjs/common';
@Injectable()
export class ProductsService {
    private products = [];
    getAll() {
        return this.products;
    }
    insert(product) {
        this.products = [
            ...this.products,
            product
        ];
        return { status:HttpStatus.Ok,  msg:
"Nuevo producto insertado"
        }
    }
}
```

Es importante que avises de que se ha hecho correctamente la operación.

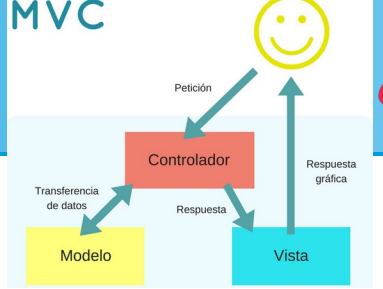
En el controlador vamos a cambiar las siguientes funciones así:

```
@Get()
    getAllProducts() {
        return this.productsService.getAll();
    }
@Post()
    createProduct(@Body('articulo') articulo:string,
        @Body('precio',ParseFloatPipe) precio:number) {
        return this.productsService.insert({
            id: this.productsService.getAll().length+1,
            articulo,
            precio
            });
    }
```

Práctica: Haz una copia de lo que llevas hasta ahora o crea un controlador y servicio limpio para montar este ejemplo concreto

Ojo, products se está almacenando en memoria y verás que se reinicia si haces algún cambio en NEST por su proceso watch

NESTJS



- **El servicio:**
  - Recuerda que hay que usar TypeScript también en el servicio y hay que definir los tipos, ahora bien, ¿Qué tipo tiene product? ¿Qué tipo se devuelve en getAll()? Necesitamos definir su **INTERFAZ**
  - **¿Qué es una interfaz?** Es una forma de especificar los campos que trataremos en el servicio y así definir su tipo. Igual que con @Param o @Body no es obligatorio, ahora sí. Para definir una interfaz de un módulo tenemos que hacer: **nest generate interface products** o nest g itf products
  - Esto habrá creado un nuevo fichero products.interface.ts donde hay que colocar:

```
export interface Product {
id: number;
articulo: string;
precio: number;
}
```

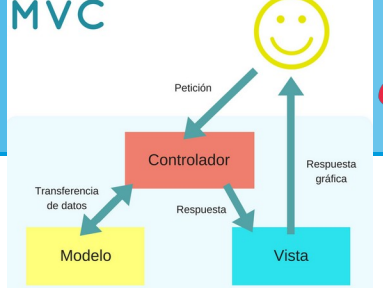
```
import { Injectable } from '@nestjs/common';
import { Product } from './products.interface';
@Injectable()
export class ProductsService {
    private products: Product[]=[];
    getAll():Product[] {
        return this.products;
    }
    insert(product:Product):
{ status: HttpStatus; msg: string }{
        this.products = [
            ...this.products,
            product
        ];
        return { status:HttpStauts.Ok,  msg: "Nuevo producto insertado"}
    }}
```

Intenta crear dos productos en el array products en código y mira que pasa si te dejas alguno de sus atributos o si no lo pones en el controlador. **La interfaz nos va a ayudar a validar la estructura JSON**

Y lo mismo con la salida, no siempre va a ser un string y podría definirse como vimos como type, interface o directamente

Práctica: Fíjate los cambios que hay que añadir al fichero de servicios y lo mismo hay que hacer en el controlador.

NESTJS



- **El servicio:**
  - Vamos a añadir algunas funciones típicas de CRUD en el fichero de servicio como:

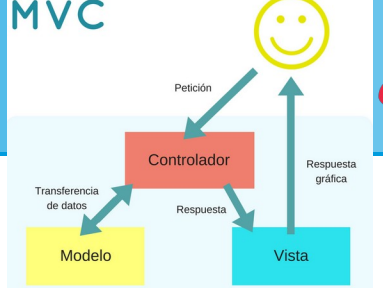
No es lo mismo find y filter!! (enlace)

```
//Obtener el producto indicado (products/1)
getId(id: number): Product {
return this.products.find( (item: Product) => item.id == id);
}

//Modificar el producto indicado con los parámetros del body
update(id: number, body: any) {
let product: Product = {
id,
articulo: body.articulo,
precio: body.precio,
}
this.products = this.products.map( (item: Product) => {
console.log(item, id, item.id == id);
return item.id == id ? product : item;
});
}
```

```
//Borrar el producto indicado
delete(id: number) {
this.products = this.products.filter( (item: Product) => item.id != id );
}

//Total de prouctos
total(): number {
return this.products.length - 1;
}
```

Práctica: Crea las operaciones en el controlador para hacer uso de estas funciones. En el caso de total() haz la ruta products/total
Incorpora un return con status y la opción de query por artículo.
Actividad: Realiza estas mismas funciones en el servicio correspondiente para el controlador que hiciste en la tienda de pinturas

NESTJS



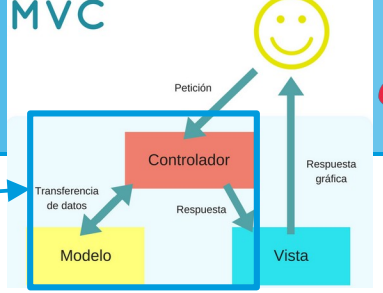
- **El servicio:**
  - Estamos dando por supuesto que todo va a ir bien pero…. ¿Y si hay algún fallo, ya sea por el usuario o por el acceso a la base de datos?
  - NESTJS tiene una serie de Excepciones que se pueden atender durante la ejecución de un programa

```
//En el controlador:
@Get(':id')
   getId(@Param('id',ParseIntPipe) id: number): Products {
      return this.serviceProducts.findOne(id);
      }

//En el servicio
findOne(id:number):Products{
      const product = this.products.find( (item: Products) => item.id == id);
      if(product) {
      return product;
      } else {
      throw new NotFoundException(`No encontramos el producto ${id}`);
      }
   }
```

Recuerda que los pipes también hacen validación, por ejemplo, si introduces product/a nos dará un error por no poder convertir a entero. No obstante podemos poner el error más específico si queremos en ese proceso: new ParseIntPipe({ errorHttpStatusCode: HttpStatus.NOT_ACCEPTABLE, exceptionFactory: (error) => { throw new BadRequestException('El ID debe ser un número entero válido'); })

En este caso, tratamos de buscar el producto con ese id en la ruta, pero si no existiera se lanza la excepción NotFoundException
Podemos usar también una excepción genérica como :
throw new HttpException(`No existe el producto ${id}`, HttpStatus.NOT_FOUND);

NESTJS

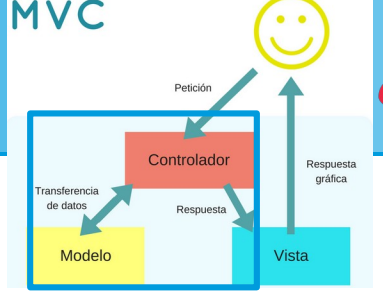


- **El módulo:**
  - Los módulos son clases que funcionan como contenedores de otras clases o artefactos, como son los controladores, servicios y otros componentes desarrollados con Nest. Los módulos sirven para agrupar elementos, de modo que una aplicación podrá tener varios módulos con clases altamente relacionadas entre sí.
  - Todas las aplicaciones tienen al menos un módulo (app), que es el módulo raíz o módulo principal, y generalmente de este módulo principal dependerán otros módulos secundarios. ¿Recuerdas que una base de datos NO es una única tabla?
  - Ahora mismo nuestro app.module.ts estará más o menos así:

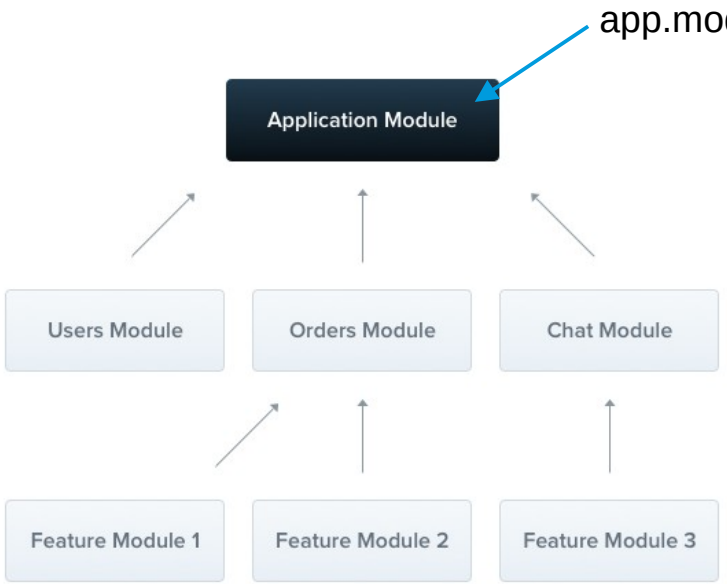
```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ProductsController } from './products/products.controller';
import { ProductsService } from './products/products.service';

@Module({
  imports: [],
  controllers: [AppController, ProductsController],
  providers: [AppService, ProductsService],
})
export class AppModule {}
```

Dentro del decorador **@Module** hay que definir los controladores y servicios a utilizar pero también podemos importar otros módulos, entendiendo estos como una unidad que tiene una funcionalidad en mi aplicación.
Por ejemplo, si estamos definiendo un controlador y servicio de productos, lo lógico es diseñar esto dentro de un módulo. Fíjate que ya de por sí están dentro de una carpeta products.

NESTJS



- **El módulo:**
  - ¿Cómo se crea un módulo? **nest generate module products o nest g mo products**
  - Fíjate que además de incorporarlo a la carpeta products, también aparece en app.module.ts



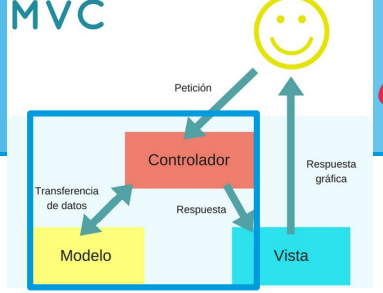
app.module.ts

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ProductsController } from './products/products.controller';
import { ProductsService } from './products/products.service';
import { ProductsModule } from './products/products.module';

@Module({
  imports: [ProductsModule],
  controllers: [AppController, ProductsController],
  providers: [AppService, ProductsService],
})
export class AppModule {}
```

Este es el contenido del módulo creado para Products

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

@Module({})

export class ProductsModule {}
```

NESTJS

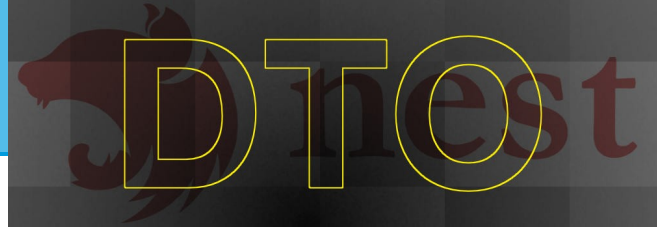


- **El módulo:**
  - Ahora vamos a quitar los elementos que están en app y que deberían estar en el módulo de Products y que quedaría así:

```
import { Module } from '@nestjs/common';
import { ProductsController } from './products.controller';
import { ProductsService } from './products.service';
@Module({
        imports: [], // Recuerda que este a su vez puede importar otros módulos.
        controllers: [ProductsController],
        providers: [ProductsService]
})
export class ProductsModule {}
```

En este caso hemos empezado la “casa por el tejado”, si hubiéramos hecho primero el módulo y luego creado el controlador y servicio, ya estaría esto creado automáticamente su contenido y enlazado en app.module.ts

NESTJS



- **Los DTO (Data Transfer Object)**
  - Supongamos que hacemos una operación GET, ¿cómo es la estructura que se recibe del servidor? O ¿Cómo son los datos de entrada para la operación POST o PUT?
  - Un DTO es un objeto que se transfiere por la red entre dos sistemas, típicamente usados en aplicaciones cliente/servidor y en las aplicaciones web modernas. En el caso web, se suele utilizar JSON para las operaciones CRUD.
  - ¿Cómo se crea un DTO? **nest generate class products/dto/product.dto** o nest g cl products/dto/product.dto
  - ¿Y qué ponemos dentro de este fichero?

```
export class ProductDto {
        articulo: string;
        precio: number;
        stock?: number;
}
```

Pero… si se puede hacer esto también usando interface ¿no? ¿Qué aporta el DTO?

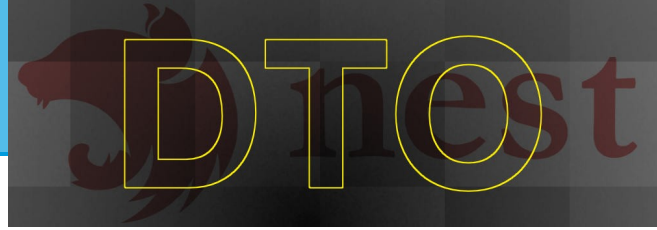
  - ¿Y esto para qué sirve? Para concretar el formato del decorador @Body y para **validar** los datos que se introducen, más allá de simples tipos.

```
import { ProductDto } from './dto/create-product.dto';
@Post()
createProduct(@Body() productDto: ProductDto){
        this.productsService.insert(productDto);
}
```

¿Qué significa '?' En stock? ¿Hay más?

- Usa **?** para propiedades opcionales.
- Usa **readonly** para propiedades inmutables.
- Usa **|** para permitir múltiples tipos en una propiedad.

```
Ej: class OrderDTO {
  id: number;
  status: 'pending' | 'completed' | 'cancelled';
  description?: string | null;
}
```



- **Los DTO (Data Transfer Object)**
  - ¿Se puede validar un DTO? Dado que puede haber diferentes rutas en las que se haga uso de un determinado DTO, es conveniente crear un pipe general que valide toda nuestra aplicación.
  - Antes de escribir el DTO hay que hacer un pipe global en el fichero main.ts:

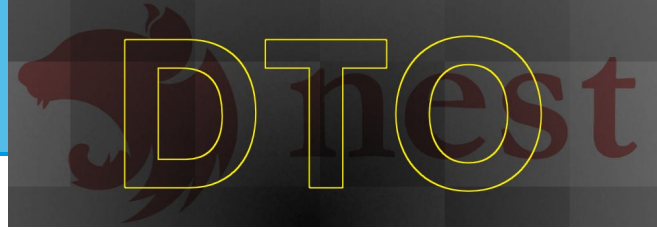
```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { ValidationPipe } from '@nestjs/common';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.enableCors();
  app.useGlobalPipes(new ValidationPipe())
  await app.listen(process.env.PORT ?? 3001);
}
bootstrap();
```

No obstante, si sólo quieres comprobar una función del controlador con el uso de un DTO puedes usar: @UsePipes(new ValidationPipe()) en la función a comprobar. Ej:

```
@Post()
@UsePipes(new ValidationPipe())
async post(@Body() body: TagDto):
Promise<Tag> {
  return this.tagsService.insert(body);
}
```

  - Además, para hacer validaciones, también hay que ejecutar lo siguiente para incluir reglas en el DTO: **npm i class-validator class-transformer**

NESTJS



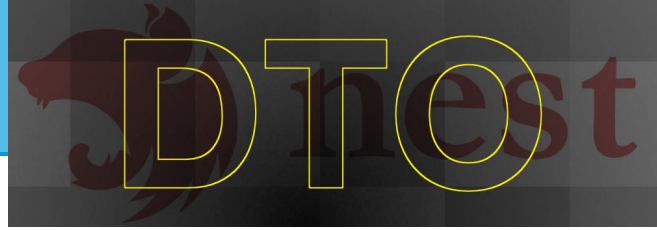
- **Los DTO (Data Transfer Object)**
  - Ten en cuenta que queremos comprobar que lo que llega por @Body es correcto y que el proceso de validación de un DTO se hace mediante anotaciones. Por ejemplo, quiero que el valor introducido de artículo sea un texto y en el precio un número. De igual forma podré ver si es un booleano, un carácter, un array, un correo, el valor mínimo, una longitud, es una fecha, es opcional, mínima longitud, … (enlace)
  - Incluso dentro de esa validación se puede indicar el fallo que hay que mostrar. Veamos un ejemplo:

```
export class ProductDto {
    @IsString({ message: 'El artículo debe de ser un string' })
    @MinLength(5,{message: 'La longitud mínima del artículo es 5'})
    articulo:string;
    @IsInt({message: 'Precio debe ser un número'})
    @Min(0,{ message: 'El precio no puede ser negativo' })
    @Max(99999,{message:'El precio no puede superar los 99.999 €'})
    precio:number;
    @IsInt({ message: 'El stock debe de ser un número entero' })
    @Min(0,{ message: 'El stock no puede ser negativo' })
    stock?: number;
}
```

IsOptional	IsPositive	IsMongoId
IsArray	IsString	IsUUID
IsDecimal	IsDate	IsDateString
IsBoolean	IsEmail	IsUrl

Estas comprobaciones se hacen ANTES de que se solicite el acceso a la base de datos. Hasta el punto de modificar campos con @BeforeInsert o @BeforeUpdate
¿Recordáis de GBD los check y los triggers?

Práctica: Hagamos pruebas en la función POST a ver las salidas que va dando



- **Los DTO (Data Transfer Object)**
  - De hecho, podríamos crear nuestra propia función de validación y crear así nuestra anotación.

```
//FICHERO: accountNumberValidator.ts
import { registerDecorator, ValidationOptions, ValidationArguments } from 'class-validator';
export function IsAccountNumber(validationOptions?: ValidationOptions) {
  return function (object: Object, propertyName: string) {
    registerDecorator({
      name: 'isAccountNumber',
      target: object.constructor,
      propertyName: propertyName,
      options: validationOptions,
      validator: {
        validate(value: any, args: ValidationArguments) {
          let account = value.replace(/\s/g, '');
          // Define la lógica de validación aquí, por ejemplo, un formato de cuenta
          return typeof account === 'string' && /^ES\d{22}$/.test(account); // Tiene ESxx y luego 20 dígitos
        },
        defaultMessage(args: ValidationArguments) {
          return 'El número de cuenta no es válido';
        }
      }
    });
  };
}
//USO
class AccountDTO {
  @IsAccountNumber({ message: 'Número de cuenta inválido: solo debe contener entre ES y 22 dígitos numéricos' })
  numeroCuenta: string;
}
```

- **Resumiendo...**
  - ¿Cómo hay que empezar a trabajar con NESTJS?
    - Cada vez que necesitemos desarrollar una API para acceder a una base de datos hay que crear un módulo. Si no nos interesa hacer pruebas quita spec
    - Hay que concretar las consultas que se vayan a realizar y que gestionará el controlador. Al menos un simple CRUD, después añadirás más.
    - Hay que desarrollar el servicio que atenderá estas peticiones
    - Hay que definir la interfaz del objeto resultado para ciertas operaciones como GET
    - Hay que definir el DTO que se usa para el @Body para ciertas operaciones
    - Esto puedes ir creando cada componente por separado o…. crear un recurso que ya te dará toda la estructura de ficheros. Hay que ejecutar: **nest generate resource X** (enlace) ¿Cómo elijo REST, GraphQL o WebSocket? (enlace)
  - Ahora que tenemos clara la estructura de ficheros… nos falta por conocer cómo conectamos el servicio a la base de datos destino y hacer uso de ese fichero que ha generado **entity**. Para esto usaremos **TypeORM**

ORM

- Ya tenemos claro cómo manejar un módulo en NEST siguiendo el modelo MVC y así crear una API sencilla. El siguiente paso es enlazar nuestra API con una base de datos y que esto sea “trasparente” para la aplicación que solicita los datos
- **¿Qué es un ORM?** Un ORM (Object-Relational Mapping, o Mapeo Objeto-Relacional) es una herramienta que facilita la interacción entre una aplicación y una base de datos relacional (como MySQL, PostgreSQL, SQLite, etc.) al permitir que trabajes con datos usando objetos y clases en lugar de consultas SQL directas. Ojo, esto no es un ODBC.
  - **Ejemplos de ORMs Populares (enlace)**
    - **TypeORM:** Popular en proyectos Node.js grandes, especialmente con NestJS.
    - **Prisma:** Facilita el desarrollo en Node.js y TypeScript, con generación de código y tipos muy avanzados.
    - **Sequelize:** Un ORM de alto nivel para Node.js que soporta múltiples bases de datos.
    - **Mongoose**: Un ORM para manejar MongoDB
    - **Entity Framework (para .NET):** Utilizado en el ecosistema de Microsoft y aplicaciones de C#.

Prisma

Sequelize

TYPEORM

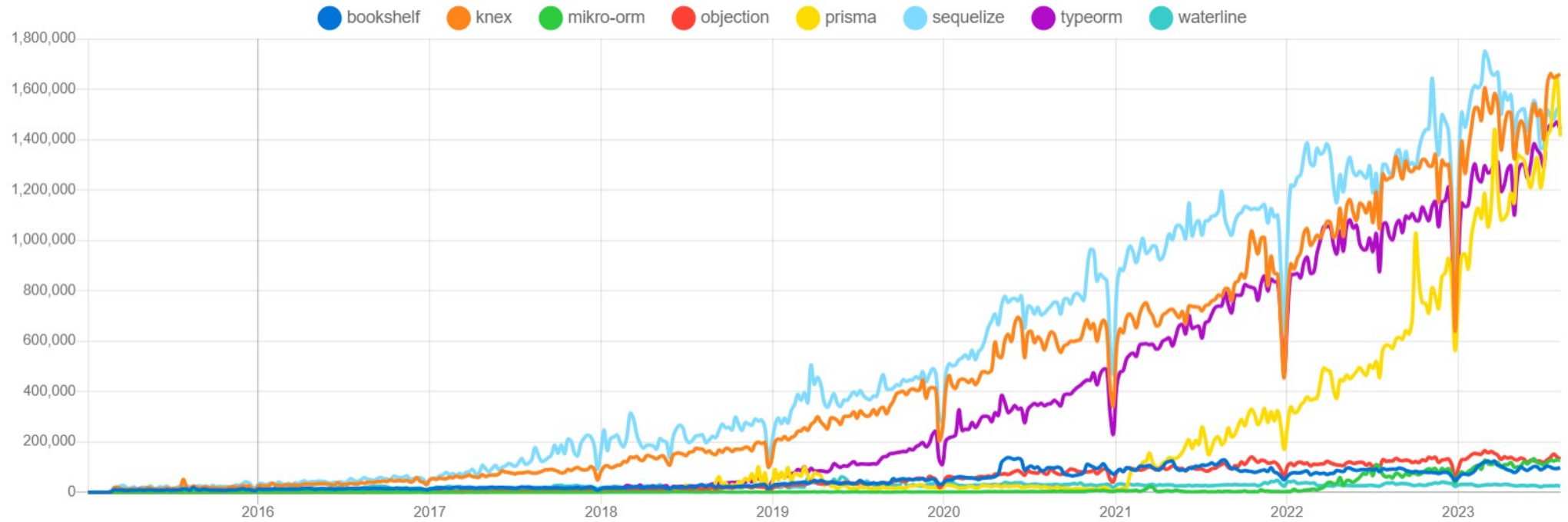
Drizzle ORM

Entity Framework

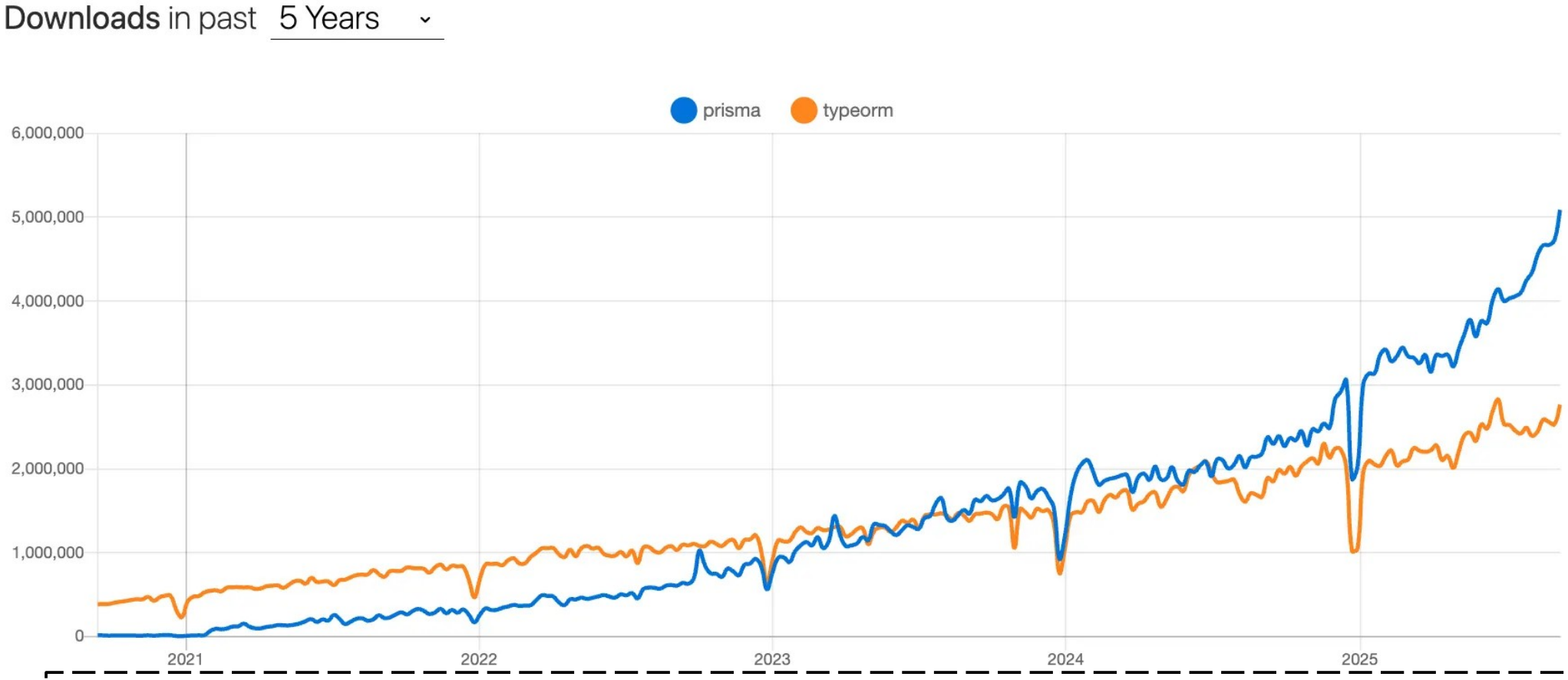
Mongoose

ORM

Downloads in past All time



ORM



Importante, ¡no te guíes por las tendencias! Analiza tu proyecto y usa el que mejor se adapte

ORM

- ¿Difiere mucho el diseño TypeORM de Prysma?

🔵 Estructura con TypeORM

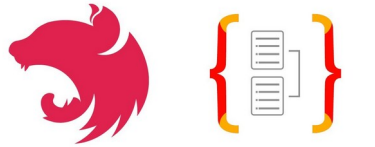
```bash
src/
│
├── app.module.ts
├── main.ts
│
├── product/
│   ├── product.module.ts
│   ├── product.controller.ts
│   ├── product.service.ts
│   └── product.entity.ts
│
└── config/
    └── typeorm.config.ts
```

🟢 Estructura con Prisma

```bash
src/
│
├── app.module.ts
├── main.ts
│
├── prisma/
│   ├── prisma.module.ts
│   └── prisma.service.ts
│
├── product/
│   ├── product.module.ts
│   ├── product.controller.ts
│   └── product.service.ts
│
prisma/
│
├── schema.prisma
└── migrations/
```

Veamos el proyecto por sus ficheros (enlace)

TypeORM



Tutorial TypeORM-MySQL: https://www.youtube.com/watch?v=RWUmlsdZ1e4

- TypeORM es un ORM para NodeJS, capaz de funcionar en muchos ambientes, como el propio Node, Cordova, Ionic, Electron y por supuesto NestJS. Es una herramienta que permite trabajar con diversas bases de datos, ayudando no sólo gracias a su capa de abstracción, sino también a la hora de montar las consultas y realizar operaciones con los datos.
- **Instalación de TypeORM:** npm i @nestjs/typeorm typeorm {mysql2, pg, sqlite3, oracledb, ...)
- Para hacer uso de typeORM en nuestra aplicación, tenemos que incluir en el fichero app.module.ts lo siguiente: import { TypeOrmModule } from '@nestjs/typeorm';
- Además, ahora hay que indicar en import, las características de la conexión a la base de datos objetivo:

```
@Module({
Imports: [..., TypeOrmModule.forRoot({
      type: 'mysql',
      host: 'localhost', //URL del servidor
      port: 3306, // Puerto de MySQL
      username: 'tu_usuario',
      password: 'tu_contraseña',
      database: 'tu_base_de_datos',
      entities: [__dirname + '/**/*.entity{.ts,.js}']
})], ...
```

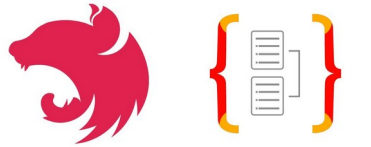
Una vez configures la conexión MySQL, cuando arranques NEST verás si ha funcionado bien o hay fallo de conexión.

Estas variables hay que especificarlas en el fichero .env y usarlas como: process.env.USER_DATABASE(enlace)

Le podríamos indicar uno a una las entidades de la aplicación, así las buscará automáticamente

Si agregamos **syncronize: true** veremos que nos va a crear la tabla en la base de datos indicada. ¡Ojo! Esto es útil sólo en desarrollo para ver cómo se crea

TypeORM



- ¿Cómo indicamos los campos de una tabla al ORM? Tenemos que definir la **entidad**
- Para ello, en la carpeta del módulo hay que crear el fichero xxxxxx.entity.ts y dentro hay que crear la entidad de forma “parecida” a como se creaba la estructura SQL de una tabla, con algunas diferencias como veremos en el siguiente ejemplo de una tabla usuario.

```
import { Entity, Column, PrimaryGeneratedColumn, CreateDateColumn, UpdateDateColumn } from 'typeorm';
@Entity()
export class Usuario {
  @PrimaryGeneratedColumn() // Genera un id autoincremental, si sólo fuera clave sería @PrimaryColumn()
  id: number;
  @Column({ type: 'varchar', length: 50 })
        nombre: string;
  @Column({ unique: true })
        email: string;
  @Column()
        password: string;
  @Column({ default: true })
        activo: boolean;
  @CreateDateColumn() // Fecha de creación automática
        fechaCreacion: Date;
  @UpdateDateColumn() // Fecha de actualización automática
        fechaActualizacion: Date;
}
```

Como se aprecia, puedes indicar el nombre de la tabla, cuál es su clave primaria, restricciones de columna como si es único y tipo en la base de datos,..

Estas entidades hay que referenciarlas durante la conexión a la base de datos en el fichero X.module.ts como entities y en los módulos donde se utilice en su import como:
imports: [TypeOrmModule.forFeature([Usuario])],

Importante: Si hubiera más de una clave primaria, hay que indicar cada campo como @PrimayColumn()
Vamos a hacer un recurso usuario para ver un ejemplo de CRUD con MySQL

Práctica: Teniendo activado syncronize, prueba la carga de esta entidad y mira en la base de datos.

TypeORM

- ¿Cómo se accede a la base de datos entonces desde el **servicio**? Con los **repositorios**
- Los repositorios son una de las piezas de software que están implementadas en el lado de TypeORM y que sirven para proveer de las funcionalidades típicas de acceso a los datos y, por supuesto, las operaciones de escritura y consulta en las tablas: create,findOne, save, insert, delete

```
import { Injectable, NotFoundException } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { Usuario } from './usuario.entity';
import { UsuarioDto } from './dto/usuario.dto';

@Injectable()
export class UsuariosService {
  constructor(
    @InjectRepository(Usuario)
    private usuarioRepository: Repository<Usuario>,
  ) {}
  async create(usuarioDto: UsuarioDto): Promise<Usuario> {
    const usuario = this.usuarioRepository.create(usuarioDto);
    return this.usuarioRepository.save(usuario);
  }
  async findAll(): Promise<Usuario[]> {
    return this.usuarioRepository.find();
  }
  async findOne(id: number): Promise<Usuario> {
    const usuario = await this.usuarioRepository.findOne({ where: { id } });
    if (!usuario) {
      throw new NotFoundException(`Usuario con ID ${id} no encontrado`);
    }
    return usuario;
  }
...
  async update(id: number, usuarioDto: UsuarioDto): Promise<Usuario> {
    const usuario = await this.findOne(id);
    this.usuarioRepository.merge(usuario, usuarioDto);
    return this.usuarioRepository.save(usuario);
  }
  async remove(id: number): Promise<void> {
    const usuario = await this.findOne(id);
    await this.usuarioRepository.remove(usuario);
  }
  async findByEmail(email: string): Promise<Usuario | undefined> {
    return this.usuarioRepository.findOne({ where: { email } });
  }
  async activateUser(id: number): Promise<Usuario> {
    const usuario = await this.findOne(id);
    usuario.activo = true;
    return this.usuarioRepository.save(usuario);
  }
  async deactivateUser(id: number): Promise<Usuario> {
    const usuario = await this.findOne(id);
    usuario.activo = false;
    return this.usuarioRepository.save(usuario);
  }
}
```

Para conectar este servicio con la tabla de MySQL hay que crear el repositorio

Al tener una función asíncrona, lo que se devuelve es una promesa.

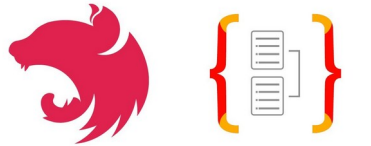
Si fuera un patch sería Partial<Usuario>

```
import { IsString, IsEmail, IsNotEmpty,
MinLength } from 'class-validator';
export class UsuarioDto {
    @IsString() @IsNotEmpty()
    nombre: string;
    @IsEmail() @IsNotEmpty()
    email: string;
    @IsString() @IsNotEmpty()
@MinLength(6)
    password: string;
}
```

Si cabe la posibilidad de que no devuelva nada hay que o bien controlar el error o devolver undefined.

Como ahora vamos a acceder a una BD, tenemos que hacer llamadas asíncronas.

TypeORM



- El **controlador** para gestionar usuarios sería el siguiente...

```
@Controller("usuario")
export class UsuariosController {
  constructor(private readonly usuariosService: UsuariosService) {}

  @Post()
  async create(@Body() usuarioDto: UsuarioDto): Promise<Usuario> {
    return this.usuariosService.create(usuarioDto);
  }

  @Get()
  async findAll(): Promise<Usuario[]> {
    return this.usuariosService.findAll()
  }

  @Get(':id')
  async findOne(@Param('id') id: string): Promise<Usuario> {
      return await this.usuariosService.findOne(+id);
  }

  @Put(":id")
  async update(@Param('id') id: string, @Body() usuarioDto: UsuarioDto):
Promise<Usuario> {
      return await this.usuariosService.update(+id, usuarioDto)
  }
...
   @Delete(':id')
  @HttpCode(HttpStatus.NO_CONTENT)
  async remove(@Param('id') id: string): Promise<void> {
      await this.usuariosService.remove(+id);
  }

  @Get('email/:email')
  async findByEmail(@Param('email') email: string): Promise<Usuario> {
    const usuario = await this.usuariosService.findByEmail(email);
    if (!usuario) {
      throw new NotFoundException(`Usuario con email ${email} no encontrado`);
    }
    return usuario;
  }

  @Put(':id/activate')
  async activateUser(@Param('id') id: string): Promise<Usuario> {
      return await this.usuariosService.activateUser(+id);
  }

  @Put(':id/deactivate')
  async deactivateUser(@Param('id') id: string): Promise<Usuario> {
      return await this.usuariosService.deactivateUser(+id);
  }
}
```

Ten en cuenta que id vendrá como un string y ha que convertirlo a number. Tengo que hacer esto en la función para que se pase a número

Práctica: Vamos a montar el ejemplo para entender lo que hace este código. Después haz pruebas en Thunder Client y mira el resultado en MySQL.

TypeORM

- Te habrás dado cuenta que con los ejemplos de prueba más este último, todas las tablas se están creando en la misma base de datos que has creado. ¿Eso quiere decir que todos mis proyectos API van a tener todas las tablas ahí y que se van a mezclar?
- NO, lo que hemos hecho en app.module es crear un punto raiz, pero puede haber varios a los que demos nombre y que éste se indique en la conexión de ese módulo en particular.

¡¡Importante!! No va a ser lo mismo, autoLoadEntities cogerá sólo las que llamemos en los módulos y entities cogerá todos los que encuentre.
Sólo dará lo mismo si sólo tienes una única conexión.

```
@Module({
  imports: [
    TypeOrmModule.forRoot({
      name: 'default',
      type: 'mysql',
      host: 'localhost',
      port: 3306,
      username: 'root',
      password: 'password',
      database: 'default_db',
      entities: [__dirname + '/**/*.entity{.ts,.js}'],
      synchronize: true,
    }),
    TypeOrmModule.forRoot({
      name: 'users',
      type: 'mysql',
      host: 'localhost',
      port: 3306,
      username: 'root',
      password: 'password',
      database: 'users_db',
      autoLoadEntities: true,
      synchronize: true,
    }),
    UsersModule,
    ProductsModule,
  ],
})
```

//Cambio en product.module.ts

```
TypeOrmModule.forFeature([Product], "default"),
```

//Cambio en product.service.ts

```
constructor( @InjectRepository(Product, 'default')
private productsRepository: Repository<Product>,
) {}
```

//Si suponemos otro módulo usuarios sería:

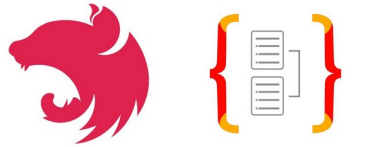
//Cambio en user.module.ts

```
TypeOrmModule.forFeature([Product], "users"),
```

//Cambio en user.service.ts

```
constructor( @InjectRepository(Product, 'users')
private productsRepository: Repository<Product>,
) {}
```

TypeORM



- **Práctica:** ¿Cómo sería el recurso para la base de datos que almacena libros?

Table: LIBRARY

No.	Title	Author	Subject	Publisher	Quantity	Price
1	Data Structure	Lipschute	DS	McGraw	4	217.00
2	DOS Guide	NORTRON	OS	PHI	3	175.00
3	Turbo C++	Robort Lafore	Prog	Galgotia	5	270.00
4	Dbase Dummies	Palmer	DBMS	PustakM	7	130.00
5	Mastering Windows	Cowart	OS	BPB	1	225.00
6	Computer Studies	French	FND	Galgotia	2	75.00
7	COBOL	Stern	Prog	John W	4	1000.00
8	Guide Network	Freed	NET	Zpress	3	200.00
9	Basic for Beginners	Norton	Prog	BPB	3	40.00
10	Advanced Pascal	Schildt	Prog	McGraw	4	350.00

Recuerda que para usar TypeORM tienes que enlazar la entidad con el módulo principal app y en su propio módulo.
No olvides la configuración para usar la validación DTO

Vamos a añadir una función para filtrar por editorial y otra por stock. Podemos incluir opciones en el find (enlace)

```
@Column("decimal", { precision: 6, scale: 2 })
price: number
```

TypeORM

- **Tarea:** ¿Cómo sería el recurso para la base de datos que almacena películas? ¿Y Pokemon? Realiza un recurso en NEST con la funcionalidad base que se ha hecho en la práctica

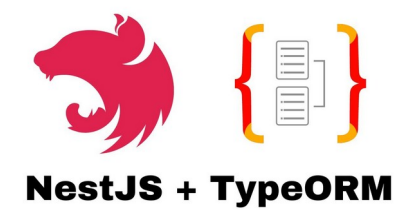
Movies table

id	title	director	year	length_minutes
1	La La Land	Steve McQueen	2010	81
2	Zootopia	Steve McQueen	2014	95
3	Deadpool	Steve McQueen	2016	93
4	Monsters, Inc.	Pete Docter	2015	92
5	Finding Nemo	Andrew Stanton	2013	107
6	The Nice Guys	Richard Linklater	1996	116
7	Bee Movie	Steve McQueen	2016	117
8	Begin Again	Richard Linklater	2017	115
9	WALL-E	Andrew Stanton	1999	104
10	Up	Joel Coen	2009	101
11	Boss Baby	Lee Unkrich	2010	103
12	X-Men Apocalypse	Steve McQueen	1998	120
13	Moana	Brenda Chapman	2012	102
14	Frozen	Dan Scanlon	2013	110

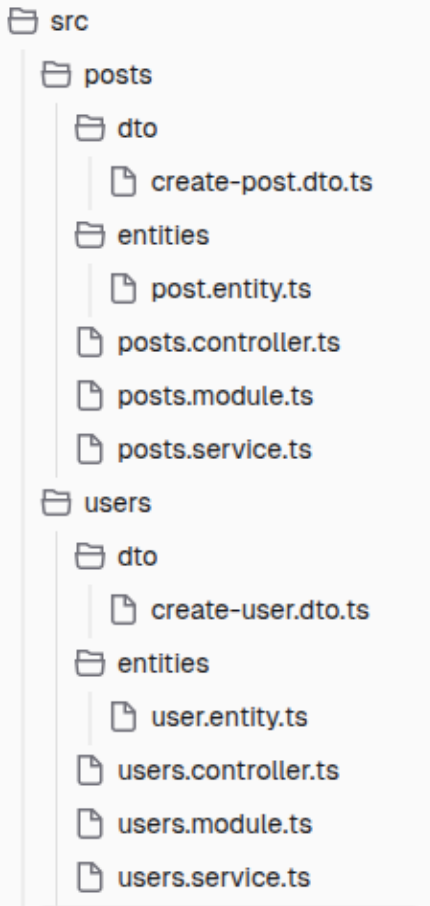
Agrega funciones para buscar por: título y between año o nombre, tipo y más HP

Introduce 15 Pokemon que quieras. Te puedes inventar los datos o cogerlos de la PokeAPI

TypeORM

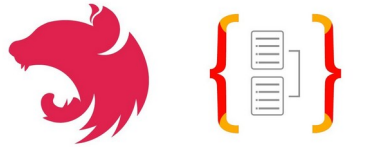


- Pero… **Una base de datos relacional no es una sola tabla!!** Supongamos que tenemos dos tablas en nuestro diseño y estamos desarrollando una aplicación que gestiona usuarios y posts en NEST. Un ejemplo de la estructura de ficheros que nos quedaría sería así:
- Otra opción sería incluir los ficheros de posts en la correspondiente carpeta de users si queremos tener sólo un único recurso. Esta será la versión recomendable para las bases de datos con varias tablas y que éstas no se usen en otros proyectos.
- Con esto queda claro que hay que tratar por un lado, operaciones que se van a realizar sobre la “tabla” posts y la “tabla” users, ahora bien, hay que ver cómo se comunican entre sí para gestionar las referencias entre entidades. ¿Referencias entre entidades?
- Recuerda que en GBD había una fase de diseño del modelo E-R donde indicabas los campos que eran referencia externa a otras, las famosas **Foreign Keys.** ¿Cómo se gestiona esto desde NESTJS con las entidades?



Práctica: Vamos a crear esta estructura que vaya en el directorio raiz. Este será un módulo sólo pero que conoce los módulos que lleva dentro, ya cada uno de ellos hará la conexión a su tabla. En las rutas del controlador evita poner /api/users y pon users

TypeORM



- **¿Cómo gestionamos estas referencias externas que vienen de relaciones N:N, 1:1, 1:N?**
- ¿Recuerdas las participaciones? Supongamos que tienes dos entidades: User y Post. Cada usuario puede tener múltiples publicaciones (One-to-Many), y cada publicación pertenece a un solo usuario (Many-to-One). (enlace) En este caso estamos tratando **una relación 1:N**

```
// src/users/entities/user.entity.ts
import { Entity, PrimaryGeneratedColumn, Column, OneToMany }
from 'typeorm';
import { Post } from '../../posts/entities/post.entity';


@Entity()
export class User {
  @PrimaryGeneratedColumn()
  id: number;


  @Column()
  nombre: string;


  @OneToMany(() => Post, (post) => post.user)
  posts: Post[];
}
```

Un usuario escribe muchos posts

La referencia externa en Post para conocer los posts que ha creado ese user

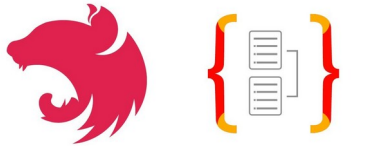
```
// src/users/entities/post.entity.ts
import { Entity, PrimaryGeneratedColumn, Column, ManyToOne } from 'typeorm';
import { User } from '../../users/entities/user.entity';


@Entity()
export class Post {
  @PrimaryGeneratedColumn()
  id: number;


  @Column()
  titulo: string;


  @Column()
  contenido: string;


  @ManyToOne(() => User, (user) => user.posts, { onDelete: 'CASCADE' })
  user: User;
}
```

Importante: No te olvides que las relaciones pueden tener atributos. Supongamos que a la relación 1:N hay un atributo de valoración. En este caso el Post (parte N) agregaría esa columna como:

```
@Column({ type: 'float', default: 0 })
  valoracion: number;
```

Hay que establecer la participación de la cardinalidad entre cada entidad. **Práctica:** Vamos a ver cómo se refleja esto en la BD

TypeORM



- **Ejemplo de NESTJS para gestionar usuarios y posts**

```
export class CreatePostDto {
  @IsString()
  titulo: string;
  @IsString()
  contenido: string;
  @IsBoolean()
  publicado: boolean;
  @IsNumber()
  autorId: number;
}
```

```
export class CreateUserDto {
  @IsString()
  usuario: string;

  @IsEmail()
  email: string;
}
```

DTO

```
@Module({
  imports:
[TypeOrmModule.forFeature(
[Post])],
  controllers:
[PostsController],
  providers: [PostsService],
})
```

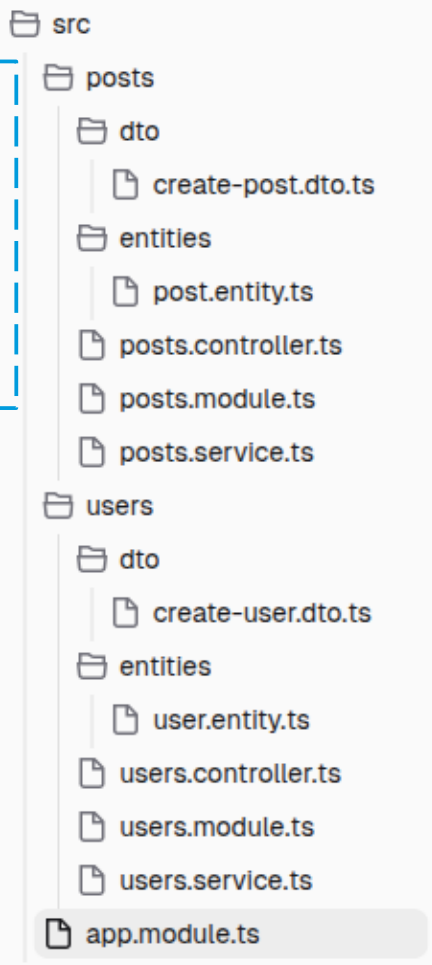
```
@Module({
  Imports:
[ TypeOrmModule.forFeature
([User])],
  controllers:
[UsersController],
  providers: [UsersService]
})
```

MODULE

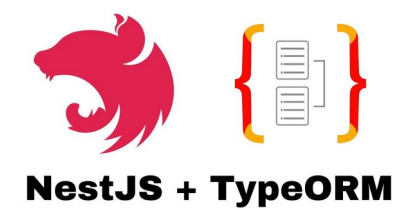
```
@Entity()
export class Posts {
  @PrimaryGeneratedColumn()
  id: number;
  @Column()
  titulo: string;
  @Column()
  contenido: string;
  @Column({ default: false })
  publicado: boolean;

  @ManyToOne(() => User, user => user.posts)
  user: User;
}
```

```
@Entity()
export class User {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  usuario: string;

  @Column()
  email: string;

  @OneToMany(() => Post, post => post.user)
  posts: Post[];
}
```

ENTITIES



TypeORM

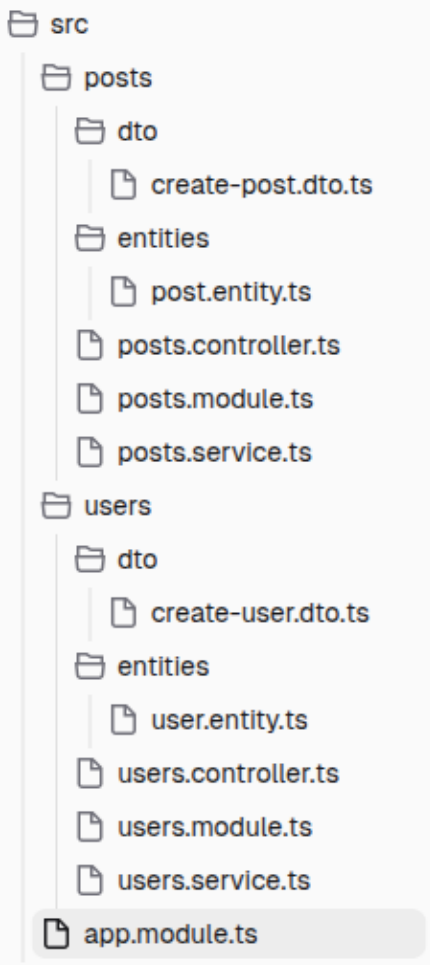


- **Ejemplo de NESTJS para gestionar usuarios y posts**

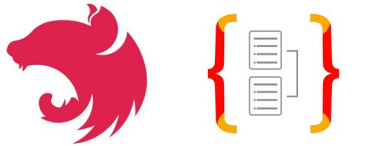
```
@Controller('posts')
export class PostsController {
  constructor(private readonly postService: PostsService) {}
  @Post()
  create(@Body() createPostDto: CreatePostDto) {
    return this.postService.create(createPostDto);
  }
  @Get()
  findAll() {   return this.postService.findAll();   }
  @Get(':id')
  findOne(@Param('id') id: number) {  return this.postService.findOne(id);   }
  @Put(':id')
  update(@Param('id') id: number, @Body() updatePostDto: UpdatePostDto) {
    return this.postService.update(id, updatePostDto);
  }
  @Delete(':id')
  remove(@Param('id') id: number) {
    return this.postService.remove(id);
  }
  @Get('user/:userId')
  findByUser(@Param('userId') userId: number) {
    return this.postService.findByUser(userId);
  }
}
```

CONTROLLER

Para users es igual sin la última función que es la que usaremos para ver qué posts ha escrito cada usuario



TypeORM



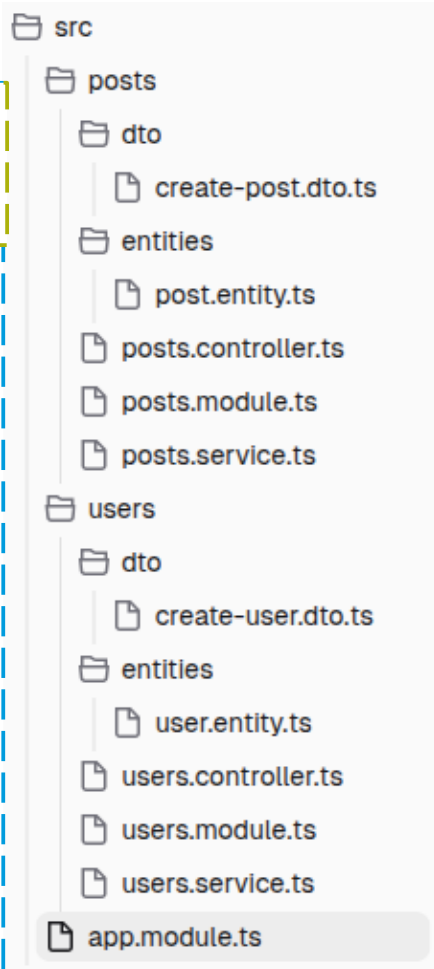
- **Ejemplo de NESTJS para gestionar usuarios y posts**

```
export class PostsService {
  constructor(
    @InjectRepository(Post,'base2')
    private postsRepository: Repository<Post>,
  ) {}
  async create(createPostDto: CreatePostDto): Promise<Posts> {
    const post = this.postsRepository.create({  ...createPostDto,  user: { id: createPostDto.autorId }, });
    return this.postsRepository.save(post);
  }
  async findAll(): Promise<Post[]> {  return this.postsRepository.find({ relations: ['user'] });  }
  async findOne(id: number): Promise<Post> {
    const post = await this.postsRepository.findOne({ where: { id }, relations: ['user'] });
    if (!post) {  throw new NotFoundException(`Post with ID "${id}" not found`); }
    return post;
  }
  async update(id: number, updatePostDto: UpdatePostDto): Promise<Post> {
    const post = await this.findOne(id);
    this.postsRepository.merge(post, updatePostDto);
    return this.postsRepository.save(post);
  }
  async remove(id: number): Promise<void> {
    const post = await this.findOne(id);
    await this.postsRepository.remove(post);
  }
  async findByUser(userId: number): Promise<Post[]> {
    return this.postsRepository.find({  where: { user: { id: userId } },  relations: ['user'], });
  }
}
```

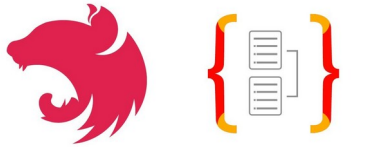
Cuando utilices un repositorio tienes que indicar sobre qué entidad actúa y en qué base de datos. Si no tienes varias, puedes obviar el término “base2”.

Necesito “enlazar” con la tabla users a través de su campo “user” para conocer quién ha creado el post. Si lo quito sólo estaré consultando la información del post con ese ID.
Este es el potencial que aporta el hecho de incluir la relación entre tablas.
Por ejemplo un usuario pregunta por un autor y debajo le muestras algunas de sus obras más destacadas.

SERVICE



TypeORM



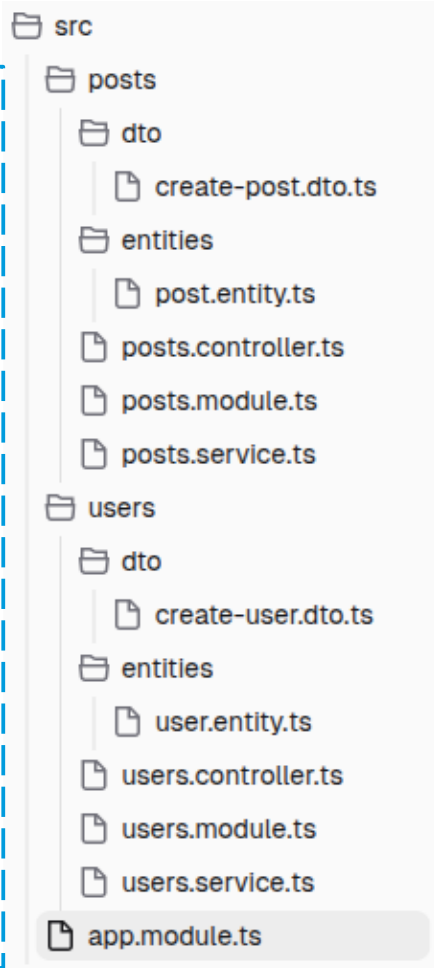
- **Ejemplo de NESTJS para gestionar usuarios y posts**

```
@Injectable()
export class UsersService {
  constructor(
    @InjectRepository(User,'base2')
    private userRepository: Repository<User>,
  ) {}
  async create(createUserDto: CreateUserDto): Promise<User> {
    const user = this.userRepository.create(createUserDto);
    return this.userRepository.save(user);
  }
  async findAll(): Promise<User[]> {
    return this.userRepository.find();
  }
  async findOne(id: number): Promise<User> {
    const user = await this.userRepository.findOne({ where: { id }, relations: ['posts'] });
    if (!user) {
      throw new NotFoundException(`User with ID "${id}" not found`);
    }
    return user;
  }
  async update(id: number, updateUserDto: UpdateUserDto): Promise<User> {
    const user = await this.findOne(id);
    this.userRepository.merge(user, updateUserDto);
    return this.userRepository.save(user);
  }
  async remove(id: number): Promise<void> {
    const user = await this.findOne(id);
    await this.userRepository.remove(user);
  }
}
```

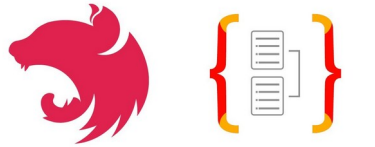
SERVICE

Aquí está la opción complementaria al otro, con el usuario conoceré todos sus posts. Aquí esto sería opcional

Práctica: Veamos la ejecución en ThunderClient



TypeORM



- **¿Y si tenemos una relación 1:1?** Lo intuitivo sería pensar que basta con indicar el decorador @OneToOne y ya está no? NO, recuerda que la clave de una de ellas se irá a la otra entidad y en la otra no se altera para nada, además en la entidad que añade la relación habrá que añadir también el decorador @JoinColumn() (enlace)

- Por ejemplo, supongamos el caso de que cada usuario tiene un único perfil personal

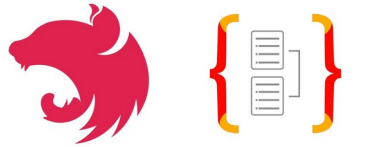
```
@Entity('profiles')
export class Profile {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  firstName: string;

  @Column()
  lastName: string;

  @Column()
  age: number;

  @OneToOne(() => User, (user) => user.profile)
  user: User;
}
```

```
@Entity('users')
export class User {
  @PrimaryGeneratedColumn()
  id: number;

  @Column({ unique: true })
  email: string;

  @Column()
  password: string;

  @OneToOne(() => Profile, (profile) => profile.user, { cascade: true })
  @JoinColumn() // La clave foránea estará en esta tabla
  profile: Profile;
}
```

Normalmente no hace falta pero hay dos tablas user y hay que diferenciarlas

La idea ahora es que al crear el perfil este se asocia al usuario

TypeORM



- **¿Podemos crear de golpe ambas filas en las entidades User y Profile? Sí, todo depende de cómo definas el DTO que aportas y cómo lo quieras programar en el servicio correspondiente**
- Es decir, NO hay que definir siempre todo el CRUD de TODO!! Sólo lo necesario. Esto estará en el mismo fichero DTO.

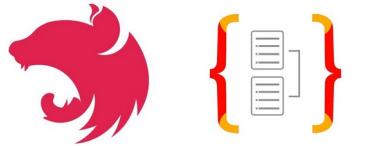
```
@Module({
  imports:
[TypeOrmModule.forFeature([User,Profile])],
  controllers: [UsersController],
  providers: [UserService],
})
```

```
class profileDTO {
  @IsString()
  firstName: string;

  @IsString()
  lastName: string;

  @IsInt()
  age: number;
};
export class CreateUserDto {
 @IsEmail()
 email: string;
 @IsString()
 password: string;
 @IsNotEmpty()
@ValidateNested()
 @Type(() => profileDTO)
 profile: profileDTO;
}
```

Esto se usará para crear el Profile correspondiente. De esta forma se indica como un tipo más que se tiene que introducir.

Fíjate la diferencia a como se ha hecho antes, se estaban haciendo CRUD completos en cada recurso, pero no tiene por qué hacerse obligatoriamente. No es mejor una que otra, **todo es cuestión de cómo queramos diseñar nuestra API.**

TypeORM



- **¿Y si tenemos una relación 1:1?** Vamos a ver cómo sería el servicio y controlador de User

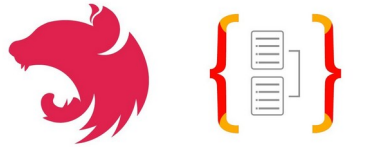
Esto hará que haya que importar Profiles en el import de User

```
@Injectable()
export class UsersService {
  constructor(
    @InjectRepository(User)
    private readonly userRepository: Repository<User>,
    @InjectRepository(Profile)
    private readonly profileRepository: Repository<Profile>,
  ) {}
  async create(createUserDto: CreateUserDto): Promise<User> {
    const { profile, ...userData } = createUserDto;
    const newProfile = this.profileRepository.create(profile);
    await this.profileRepository.save(newProfile);
    const user = this.userRepository.create({ ...userData, profile: newProfile });
    return this.userRepository.save(user);
  }
  async findAll(): Promise<User[]> {
    return this.userRepository.find({ relations: ['profile'] });
  }
  async findOne(id: number): Promise<User> {
    const user = await this.userRepository.findOne({ where: { id }, relations: ['profile'] });
    if (!user) throw new NotFoundException(`User with id ${id} not found`);
    return user;
  }
  async update(id: number, updateUserDto: UpdateUserDto): Promise<User> {
    const user = await this.findOne(id);
    Object.assign(user, updateUserDto);
    return this.userRepository.save(user);
  }
  async remove(id: number): Promise<void> {
    const result = await this.userRepository.delete(id);
    if (result.affected === 0) throw new NotFoundException(`User with id ${id} not found`);
  }
}
```

```
@Controller('users')
export class UsersController {
  constructor(private readonly usersService: UsersService) {}
  @Post()
  create(@Body() createUserDto: CreateUserDto) {
    return this.usersService.create(createUserDto);
  }
  @Get()
  findAll() {
    return this.usersService.findAll();
  }
  @Get(':id')
  findOne(@Param('id') id: string) {
    return this.usersService.findOne(+id);
  }
  @Put(':id')
  update(@Param('id') id: string, @Body() updateUserDto:
UpdateUserDto) {
    return this.usersService.update(+id, updateUserDto);
  }
  @Delete(':id')
  remove(@Param('id') id: string) {
    return this.usersService.remove(+id);}
```

Es importante que entiendas que **NO** hay una única forma de plantear una API. TÚ eliges el diseño!!
Por ejemplo, si quiero no le defino controlador o servicio, eso sí, la entidad sí es necesaria

TypeORM



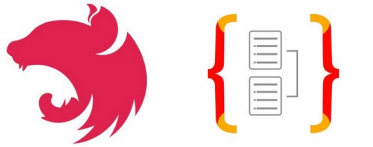
- **¿Y si tenemos una relación N:N?** Recuerda que una relación N:N genera una nueva tabla o entidad que recoge las claves primarias y atributos. Para este caso hay que considerar esta nueva tabla como la parte N de la relación 1:N con las otras dos (enlace) **Este sería el enfoque tradicional**
- Por ejemplo, supongamos que tenemos productos y tallas y que la relación es N:N incluye el atributo **precio**. Las entidades quedarían como sigue:

```
@Entity()
export class Product {
  @PrimaryGeneratedColumn()
  id: number

  @Column()
  name: string

  @Column()
  description: string

  @OneToMany(
    () => ProductSize,
    (productSize) => productSize.product,
  )
  productSizes: ProductSize[]
}
```

```
@Entity()
export class Size {
  @PrimaryGeneratedColumn()
  id: number

  @Column()
  name: string

  @OneToMany(
    () => ProductSize,
    (productSize) => productSize.size,
  )
  productSizes: ProductSize[]
}
```

```
@Entity()
export class ProductSize {
  @PrimaryGeneratedColumn()
  id: number

  @Column("decimal", { precision: 10, scale: 2 })
  price: number

  @ManyToOne(
    () => Product,
    (product) => product.productSizes,
  )
  product: Product

  @ManyToOne(
    () => Size,
    (size) => size.productSizes,
  )
  size: Size
}
```

TypeORM



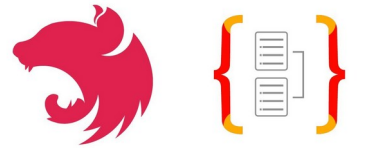
- **Opción 2: ¿Y si tenemos una relación N:N?** En el caso de que la tabla N:N sólo tenga las referencias a las claves primarias de las tablas, es posible realizar un montaje de las entidades con el decorador @ManyToMany y colocando en la entidad más relevante el decorador @JoinTable() (enlace)
- Para el ejemplo, tenemos productos y tallas y que la relación es N:N. Lo lógico es que sea la entidad más relevante el producto no? Por lo que esta parte de relación N:N quedará así en cada una:

```
@Entity()
export class Product {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  name: string;

  @Column()
  description: string;

  @ManyToMany(() => Size)
  @JoinTable()
  sizes: Size[];
}
```

```
@Entity()
export class Size {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  name: string;

  @ManyToMany(() => Product)
  products: Product[];
}
```

En el caso de que no haya atributo en la relación podría hacerse así también. **NO** es mejor una que otra pero la primera es más didáctica

TypeORM



- **Práctica: Ahora veamos el caso general de relación N:N**

```
@Entity()
export class Producto {
  @PrimaryGeneratedColumn()
  id: number

  @Column()
  nombre: string

  @Column()
  descripcion: string

  @OneToMany(
    () => ProductoTalla,
    (productoTalla) => productoTalla.producto,
  )
  productoTallas: ProductoTalla[]
}
```

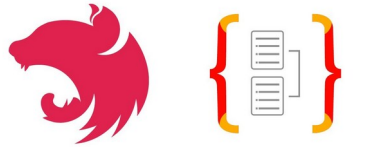
```
@Entity()
export class Talla {
  @PrimaryGeneratedColumn()
  id: number

  @Column()
  nombre: string

  @OneToMany(
    () => ProductoTalla,
    (productoTalla) => productoTalla.talla,
  )
  productoTallas: ProductoTalla[]
}
```

```
@Entity()
export class ProductoTalla {
  @PrimaryGeneratedColumn()
  id: number

  @Column("decimal", { precision: 10, scale: 2 })
  precio: number

  @ManyToOne(
    () => Producto,
    (producto) => producto.productoTallas,
  )
  producto: Producto

  @ManyToOne(
    () => Talla,
    (talla) => talla.productoTallas,
  )
  talla: Talla
}
```

Estas son las FK

Vamos a crear 3 recursos dentro de una carpeta **NN**

TypeORM



- **Práctica: Ahora veamos el caso general de relación N:N**

```
export class CreateProductoDto {
  @IsNotEmpty()
  @IsString()
  nombre: string

  @IsNotEmpty()
  @IsString()
  descripcion: string

  @IsArray()
  @ValidateNested({ each: true })
  @Type(() => ProductoTallaDto)
  tallas: ProductoTallaDto[]
}
```

```
export class ProductoTallaDto {
  @IsNotEmpty()
  productoId: number
  @IsNotEmpty()
  tallaId: number

  @IsNotEmpty()
  precio: number
}
```

```
export class CreateTallaDto {
  @IsNotEmpty()
  @IsString()
  nombre: string
}
```

DTO

```
@Module({
  imports: [TypeOrmModule.forFeature([Producto])],
  controllers: [ProductosController],
  providers: [ProductosService],
  exports: [ProductosService],
})
@Module({
  imports: [TypeOrmModule.forFeature([Talla])],
  controllers: [TallasController],
  providers: [TallasService],
  exports: [TallasService],
})
@Module({
  imports: [TypeOrmModule.forFeature([ProductoTalla,
Producto, Talla])],
  controllers: [ProductoTallaController],
  providers: [ProductoTallaService],
  exports: [ProductoTallaService],
})
```

MÓDULOS

TypeORM


NestJS + TypeORM

- **Práctica: Ahora veamos el caso general de relación N:N**

Igual con TALLA

Recuerda que el parámetro body se puede partir

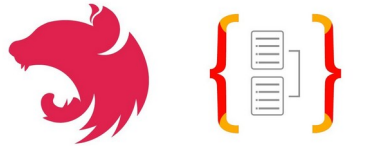
```
@Controller('productos')
export class ProductosController {
  constructor(private readonly productosService: ProductosService) {}

  @Post()
  async crearProducto(@Body() body: { nombre: string; descripcion: string }) {
    return await this.productosService.crearProducto(body.nombre,
body.descripcion);
  }

  @Get()
  async obtenerTodos() {
    return await this.productosService.obtenerTodos();
  }

  @Get(':id')
  async obtenerPorId(@Param('id') id: number) {
    return await this.productosService.obtenerPorId(id);
  }

  @Delete(':id')
  async eliminarProducto(@Param('id') id: number) {
    return await this.productosService.eliminarProducto(id);
  }
}
```

CONTROLADOR

```
@Injectable()
export class ProductosService {
  constructor(
    @InjectRepository(Producto)
    private readonly productoRepository: Repository<Producto>,
  ) {}
  async crearProducto(nombre: string, descripcion: string): Promise<Producto> {
    const producto = this.productoRepository.create({ nombre, descripcion });
    return await this.productoRepository.save(producto);
  }
  async obtenerTodos(): Promise<Producto[]> {
    return await this.productoRepository.find();
  }
  async obtenerPorId(id: number): Promise<Producto|null> {
    return await this.productoRepository.findOne({ where: { id } });
  }
  async eliminarProducto(id: number): Promise<string> {
    const producto = await this.productoRepository.findOne({ where: { id } });

    if (!producto) {
      throw new Error(`Producto con ID ${id} no encontrado`);
    }
    await this.productoRepository.remove(producto);
    return `Producto con ID ${id} eliminado correctamente`;
  }
}
```

SERVICIO

TypeORM



- **Práctica: Ahora veamos el caso general de relación N:N**

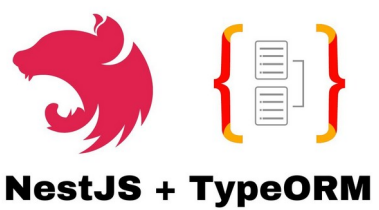
```
@Injectable()
export class ProductoTallaService {
  constructor(
    @InjectRepository(ProductoTalla)
    private readonly productoTallaRepository: Repository<ProductoTalla>,
    @InjectRepository(Producto)
    private readonly productoRepository: Repository<Producto>,
    @InjectRepository(Talla)
    private readonly tallaRepository: Repository<Talla>,
  ) {}

  async asignarPrecio(productoId: number, tallaId: number, precio: number): Promise<ProductoTalla> {
    const producto = await this.productoRepository.findOne({ where: { id: productoId } });
    const talla = await this.tallaRepository.findOne({ where: { id: tallaId } });

    if (!producto || !talla) {
      throw new Error('Producto o talla no encontrados');
    }

    const productoTalla = this.productoTallaRepository.create({ producto, talla, precio });
    return await this.productoTallaRepository.save(productoTalla);
  }

  async obtenerProductoTalla(productoId: number, tallaId: number): Promise<ProductoTalla | null> {
    return await this.productoTallaRepository.findOne({
      where: { producto: { id: productoId }, talla: { id: tallaId } },
      relations: ['producto', 'talla'],
    });
  }
}
```

SERVICIO PRODUCTO-TALLA

```
@Controller('producto-talla')
export class ProductoTallaController {
  constructor(private readonly productoTallaService: ProductoTallaService) {}

  @Post()
  async asignarPrecio(
    @Body() body: { productoId: number; tallaId: number; precio: number }
  ) {
    return await this.productoTallaService.asignarPrecio(body.productoId, body.tallaId, body.precio);
  }

  @Get(':productoId/:tallaId')
  async obtenerProductoTalla(@Param('productoId') productoId: number, @Param('tallaId') tallaId: number) {
    return await this.productoTallaService.obtenerProductoTalla(productoId, tallaId);
  }
}
```

CONTROLADOR

TypeORM



- **Práctica: Vamos a realizar el segundo caso:**

```
@Entity()
export class Producto {
  @PrimaryGeneratedColumn()
  id: number

  @Column()
  nombre: string

  @Column()
  descripcion: string

  @ManyToMany(() => Talla)
  @JoinTable()
  tallas: Talla[]
}
```

```
@Entity()
export class Talla {
  @PrimaryGeneratedColumn()
  id: number

  @Column()
  nombre: string

  @ManyToMany(() => Producto)
  productos: Producto[]
}
```

ENTIDADES

```
export class CreateProductoDto {
  @IsNotEmpty()
  @IsString()
  nombre: string

  @IsNotEmpty()
  @IsString()
  descripcion: string

  @IsArray()
  @IsNumber({}, { each: true })
  tallaIds: number[]
}
```

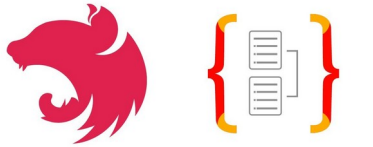
Fíjate que cada producto tendrá asociado un array de tallas

```
export class CreateTallaDto {
  @IsNotEmpty()
  @IsString()
  nombre: string
}
```

DTO

En el desarrollo, vamos a seguir el primer enfoque donde crearemos un módulo **NN2** y luego iremos agregando recursos para cada tabla que necesitemos.

TypeORM



- **Práctica: Vamos a realizar el segundo caso que parece más simple:**

```
@Module({
  imports: [TypeOrmModule.forFeature([Productos, Talla])],
  providers: [ProductosService],
  controllers: [ProductosController],
})
```

```
@Module({
  imports: [TypeOrmModule.forFeature([Talla])],
  providers: [TallasService],
  controllers: [TallasController],
})
```

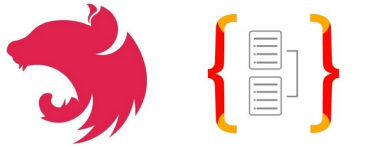
MODULOS

```
@Controller('productos')
export class ProductosController {
  constructor(private readonly productosService: ProductosService) {}
  @Post()
  async crearProducto(
    @Body() body: { nombre: string; descripcion: string; tallaIds: number[] }
  ) {
    return await this.productosService.crearProducto(body.nombre, body.descripcion, body.tallaIds);
  }
  @Get()
  async obtenerTodos() {
    return await this.productosService.obtenerTodos();
  }
  @Get(':id')
  async obtenerPorId(@Param('id') id: number) {
    return await this.productosService.obtenerPorId(id);
  }
  @Put(':id/tallas')
  async asignarTallas(
    @Param('id') id: number,
    @Body() body: { tallaIds: number[] }
  ) {
    return await this.productosService.asignarTallas(id, body.tallaIds);
  }
  @Delete(':id')
  async eliminarProducto(@Param('id') id: number) {
    return await this.productosService.eliminarProducto(id);
  }
}
```

El CRUD de Talla es el básico
Aquí pondremos lo que afecta a Producto

CONTROLADORES

TypeORM



- **Práctica: Vamos a realizar el segundo caso que parece más simple:**

```
@Injectable()
export class ProductosService {
  constructor(
    @InjectRepository(Producto)
    private readonly productoRepository:
Repository<Producto>,
    @InjectRepository(Talla)
    private readonly tallaRepository: Repository<Talla>,
  ) {}

  async crearProducto(nombre: string, descripcion: string,
tallaIds: number[]): Promise<Producto> {
    const producto = this.productoRepository.create({ nombre,
descripcion });

    if (tallaIds.length > 0) {
      const tallas = await this.tallaRepository.find({ where: { id:
In(tallaIds)}} );
      producto.tallas = tallas;
    }

    return await this.productoRepository.save(producto);
  }
...
```

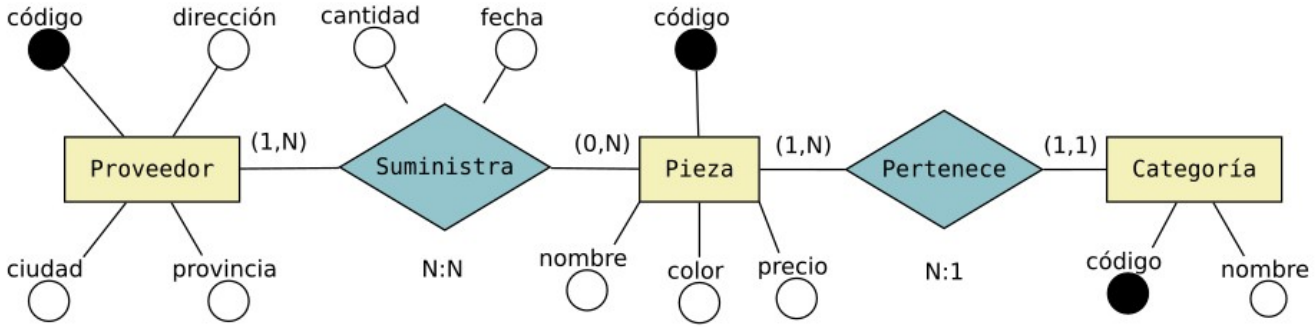
Cuando se crea el producto ya se colocan las tallas

SERVICIO PRODUCTO

```
  async obtenerTodos(): Promise<Producto[]> {
    return await this.productoRepository.find({ relations: ['tallas'] });
  }

  async obtenerPorId(id: number): Promise<Producto> {
    return await this.productoRepository.findOne({ where: { id }, relations: ['tallas'] });
  }
  async asignarTallas(productoId: number, tallaIds: number[]): Promise<Producto> {
    const producto = await this.productoRepository.findOne({ where: { id: productoId }, relations: ['tallas'] });
    if (!producto) {
      throw new Error(`Producto con ID ${productoId} no encontrado`);
    }

    const tallas = await this.tallaRepository.findByIds(tallaIds);
    producto.tallas = tallas;

    return await this.productoRepository.save(producto);
  }

  async eliminarProducto(id: number): Promise<string> {
    const producto = await this.productoRepository.findOne({ where: { id } });

    if (!producto) {
      throw new Error(`Producto con ID ${id} no encontrado`);
    }

    await this.productoRepository.remove(producto);
    return `Producto con ID ${id} eliminado correctamente`;
  }
}
```

TypeORM

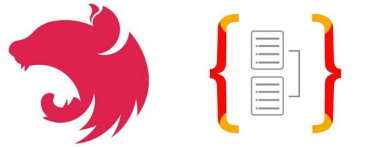
Práctica: Vamos a montar el ejemplo considerando una base de datos de almacén



- PROVEEDOR(**código**, dirección, ciudad, provincia)
- CATEGORÍA(**código**, nombre)
- PIEZA(**código**, nombre, color, precio, código_categoria)
  - *código_categoria*: FOREIGN KEY de CATEGORÍA(código)
- PROVEEDOR_SUMINISTRA_PIEZA(**código_proveedor**, **código_pieza**, **fecha_hora**, cantidad)
  - *código_proveedor*: FOREIGN KEY de PROVEEDOR(código)
  - *código_pieza*: FOREIGN KEY de PIEZA(código)

Vamos a desarrollar una API llamada **recambios** donde vayamos desarrollando un CRUD básico.

TypeORM



Práctica: Vamos a montar el ejemplo considerando una base de datos de almacén

```
@Entity()
export class Proveedor {
  @PrimaryGeneratedColumn()
  codigo: number;

  @Column()
  direccion: string;

  @Column()
  ciudad: string;

  @Column()
  provincia: string;

  @OneToMany(() => ProveedorSuministraPieza, psp =>
psp.proveedor)
  proveedorSuministraPiezas: ProveedorSuministraPieza[];
}
```

```
@Entity()
export class Pieza {
  @PrimaryGeneratedColumn()
  codigo: number;

  @Column()
  nombre: string;

  @Column()
  color: string;

  @Column('decimal', { precision: 7, scale: 2 })
  precio: number;

  @ManyToOne(() => Categoria, categoria => categoria.piezas)
  categoria: Categoria;

  @OneToMany(() => ProveedorSuministraPieza, psp => psp.pieza)
  proveedorSuministraPiezas: ProveedorSuministraPieza[];
}
```

TypeORM


NestJS + TypeORM

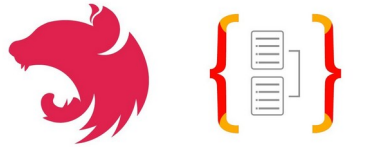
Práctica: Vamos a montar el ejemplo considerando una base de datos de almacén

```
@Entity()
export class Categoria {
  @PrimaryGeneratedColumn()
  codigo: number;

  @Column({ unique: true })
  nombre: string;

  @OneToMany(() => Pieza, pieza => pieza.categoria)
  piezas: Pieza[];
}
```

ENTIDADES

```
@Entity()
export class ProveedorSuministraPieza {
  @PrimaryColumn()
  codigoProveedor: number;

  @PrimaryColumn()
  codigoPieza: number;

  @PrimaryColumn()
  fecha: Date;

  @Column()
  cantidad: number;

  @ManyToOne(() => Proveedor, proveedor => proveedor.proveedorSuministraPiezas)
  @JoinColumn({ name: 'codigoProveedor' })
  proveedor: Proveedor;

  @ManyToOne(() => Pieza, pieza => pieza.proveedorSuministraPiezas)
  @JoinColumn({ name: 'codigoPieza' })
  pieza: Pieza;
}
```

TypeORM



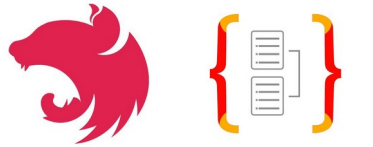
Práctica: Vamos a montar el ejemplo considerando una base de datos de almacén

```
export class CreateProveedorDto {
  @IsNotEmpty()
  @IsString()
  direccion: string;

  @IsNotEmpty()
  @IsString()
  ciudad: string;

  @IsNotEmpty()
  @IsString()
  provincia: string;
}

export class CreateCategoriaDto {
  @IsNotEmpty()
  @IsString()
  nombre: string;
}
```

```
export class CreatePiezaDto {
  @IsNotEmpty()
  @IsString()
  nombre: string;

  @IsNotEmpty()
  @IsString()
  color: string;

  @IsNotEmpty()
  @IsNumber()
  @IsPositive()
  precio: number;

  @IsNotEmpty()
  @IsNumber()
  codigoCategoria: number;
}
```

```
export class CreateProveedorSuministraPiezaDto
{
  @IsNotEmpty()
  @IsNumber()
  codigoProveedor: number;

  @IsNotEmpty()
  @IsNumber()
  codigoPieza: number;

  @IsNotEmpty()
  @IsDateString()
  fecha: string;

  @IsNotEmpty()
  @IsNumber()
  @IsPositive()
  cantidad: number;
}
```

TypeORM

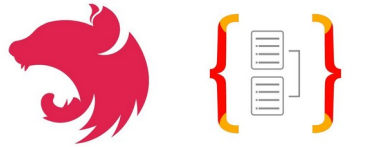


Práctica: Vamos a montar el ejemplo considerando una base de datos de almacén

SERVICE CATEGORÍA

```
@Injectable()
export class CategoriaService {
  constructor(
    @InjectRepository(Categoria)
    private categoriaRepository: Repository<Categoria>,
  ) {}

  create(createCategoriaDto: CreateCategoriaDto) {
    return this.categoriaRepository.save(createCategoriaDto);
  }
  findAll() {
    return this.categoriaRepository.find();
  }
  findOne(id: number) {
    return this.categoriaRepository.findOne({ where: { codigo: id } });
  }
  update(id: number, updateCategoriaDto: UpdateCategoriaDto) {
    return this.categoriaRepository.update(id, updateCategoriaDto);
  }
  remove(id: number) {
    return this.categoriaRepository.delete(id);
  }
}
```

TypeORM

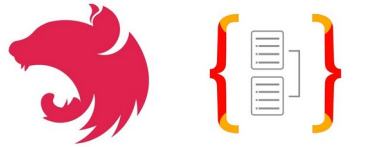


Práctica: Vamos a montar el ejemplo considerando una base de datos de almacén

SERVICE PIEZA

```
@Injectable()
export class PiezaService {
  constructor(
    @InjectRepository(Pieza)
    private piezaRepository: Repository<Pieza>,
  ) {}
  create(createPiezaDto: CreatePiezaDto) {
    return this.piezaRepository.save(createPiezaDto);
  }
  findAll() {
    return this.piezaRepository.find({ relations: ['categoria'] });
  }
  findOne(id: number) {
    return this.piezaRepository.findOne({ where: { codigo: id }, relations:
['categoria'] });
  }
  update(id: number, updatePiezaDto: UpdatePiezaDto) {
    return this.piezaRepository.update(id, updatePiezaDto);
  }
  remove(id: number) {
    return this.piezaRepository.delete(id);
  }
}
```

TypeORM

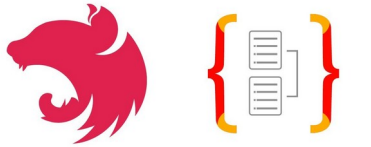


Práctica: Vamos a montar el ejemplo considerando una base de datos de almacén

SERVICE PROVEEDOR

```
@Injectable()
export class ProveedorService {
  constructor(
    @InjectRepository(Proveedor)
    private proveedorRepository: Repository<Proveedor>,
  ) {}

  create(createProveedorDto: CreateProveedorDto) {
    return this.proveedorRepository.save(createProveedorDto);
  }
  findAll() {
    return this.proveedorRepository.find();
  }
  findOne(id: number) {
    return this.proveedorRepository.findOne({ where: { codigo: id } });
  }
  update(id: number, updateProveedorDto: UpdateProveedorDto) {
    return this.proveedorRepository.update(id, updateProveedorDto);
  }
  remove(id: number) {
    return this.proveedorRepository.delete(id);
  }
}
```

TypeORM



Práctica: Vamos a montar el ejemplo considerando una base de datos de almacén

SERVICE PROVEEDOR_SUMINISTRA_PIEZA

```
@Injectable()
export class ProveedorSuministraPiezaService {
  constructor(
    @InjectRepository(ProveedorSuministraPieza)
    private proveedorSuministraPiezaRepository: Repository<ProveedorSuministraPieza>,
  ) {}
  create(createProveedorSuministraPiezaDto: CreateProveedorSuministraPiezaDto) {
    return this.proveedorSuministraPiezaRepository.save(createProveedorSuministraPiezaDto);}
  findAll() {
    return this.proveedorSuministraPiezaRepository.find({ relations: ['proveedor', 'pieza'] });
  }
  findOne(codigoProveedor: number, codigoPieza: number, fecha: Date) {
    return this.proveedorSuministraPiezaRepository.findOne({where: { codigoProveedor, codigoPieza, fecha },
      relations: ['proveedor', 'pieza'], });
  }
  update(codigoProveedor: number, codigoPieza: number, fecha: Date, updateProveedorSuministraPiezaDto:
UpdateProveedorSuministraPiezaDto) {
    return this.proveedorSuministraPiezaRepository.update( { codigoProveedor, codigoPieza, fecha },
      updateProveedorSuministraPiezaDto);
  }
  remove(codigoProveedor: number, codigoPieza: number, fecha: Date) {
    return this.proveedorSuministraPiezaRepository.delete({ codigoProveedor, codigoPieza, fecha });
  }
}
```

TypeORM

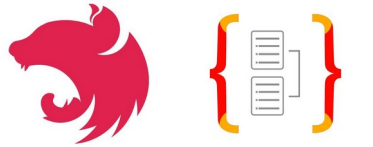

NestJS + TypeORM

Práctica: Vamos a montar el ejemplo considerando una base de datos de almacén

CONTROLADOR CATEGORÍA

```
@Controller('categoria')
export class CategoriaController {
  constructor(private readonly categoriaService: CategoriaService) {}

  @Post()
  create(@Body() createCategoriaDto: CreateCategoriaDto) {
    return this.categoriaService.create(createCategoriaDto);
  }
  @Get()
  FindAll() {     return this.categoriaService.findAll();   }
  @Get(':id')
  findOne(@Param('id') id: string) {
    return this.categoriaService.findOne(+id);
  }
  @Put(':id')
  update(@Param('id') id: string, @Body() updateCategoriaDto: UpdateCategoriaDto) {
    return this.categoriaService.update(+id, updateCategoriaDto);
  }
  @Delete(':id')
  remove(@Param('id') id: string) {
    return this.categoriaService.remove(+id);
  }
}
```

TypeORM

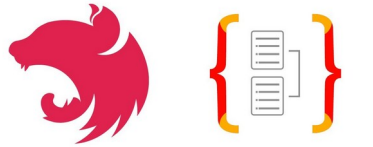


Práctica: Vamos a montar el ejemplo considerando una base de datos de almacén

CONTROLADOR PIEZA

```
@Controller('pieza')
export class PiezaController {
  constructor(private readonly piezaService: PiezaService) {}

  @Post()
  create(@Body() createPiezaDto: CreatePiezaDto) {
    return this.piezaService.create(createPiezaDto);
  }
  @Get()
  findAll() {    return this.piezaService.findAll();  }
  @Get(':id')
  findOne(@Param('id') id: string) {
    return this.piezaService.findOne(+id);
  }
  @Put(':id')
  update(@Param('id') id: string, @Body() updatePiezaDto:
UpdatePiezaDto) {
    return this.piezaService.update(+id, updatePiezaDto);
  }
  @Delete(':id')
  remove(@Param('id') id: string) {
    return this.piezaService.remove(+id);
  }
}
```

TypeORM

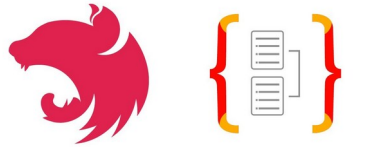


Práctica: Vamos a montar el ejemplo considerando una base de datos de almacén

CONTROLADOR PROVEEDOR

```
@Controller('proveedor')
export class ProveedorController {
  constructor(private readonly proveedorService: ProveedorService) {}

  @Post()
  create(@Body() createProveedorDto: CreateProveedorDto) {
    return this.proveedorService.create(createProveedorDto);
  }
  @Get()
  findAll() {    return this.proveedorService.findAll();  }
  @Get(':id')
  findOne(@Param('id') id: string) {
    return this.proveedorService.findOne(+id);
  }
  @Put(':id')
  update(@Param('id') id: string, @Body() updateProveedorDto:
UpdateProveedorDto) {
    return this.proveedorService.update(+id, updateProveedorDto);
  }
  @Delete(':id')
  remove(@Param('id') id: string) {
    return this.proveedorService.remove(+id);
  }
}
```

TypeORM



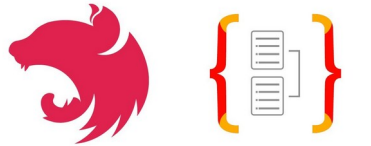
Práctica: Vamos a montar el ejemplo considerando una base de datos de almacén

```
@Controller('proveedor-suministra-pieza')
export class ProveedorSuministraPiezaController {
  constructor(private readonly proveedorSuministraPiezaService: ProveedorSuministraPiezaService) {}
  @Post()
  create(@Body() createProveedorSuministraPiezaDto: CreateProveedorSuministraPiezaDto) {
    return this.proveedorSuministraPiezaService.create(createProveedorSuministraPiezaDto);
  }
  @Get()
  findAll() { return this.proveedorSuministraPiezaService.findAll();}
  @Get(':codigoProveedor/:codigoPieza/:fecha')
  findOne(@Param('codigoProveedor') codigoProveedor: string,@Param('codigoPieza') codigoPieza: string,
    @Param('fecha') fecha: string) {
    return this.proveedorSuministraPiezaService.findOne(+codigoProveedor, +codigoPieza, new Date(fecha));
  }
  @Put(':codigoProveedor/:codigoPieza/:fecha')
  update(@Param('codigoProveedor') codigoProveedor: string,@Param('codigoPieza') codigoPieza: string,
    @Param('fecha') fecha: string,    @Body() updateProveedorSuministraPiezaDto: UpdateProveedorSuministraPiezaDto) {
    return this.proveedorSuministraPiezaService.update(+codigoProveedor, +codigoPieza, new Date(fecha),
updateProveedorSuministraPiezaDto);
  }
  @Delete(':codigoProveedor/:codigoPieza/:fecha')
  remove(@Param('codigoProveedor') codigoProveedor: string,@Param('codigoPieza') codigoPieza: string,
    @Param('fecha') fecha: string) {
    return this.proveedorSuministraPiezaService.remove(+codigoProveedor, +codigoPieza, new Date(fecha));
  }
}
```

CONTROLADOR PROVEEDOR_SUMINISTRA_PIEZA

No olvidéis que el DTO se puede fragmentar

TypeORM



- Hasta ahora se ha estado buscando el componente entero por id o todos y relacionando con las tablas en el caso N, ¿no podemos hacer consultas SQL sobre la Base de Datos?
- Imagina que tenemos una entidad que está relacionada con tamaños y que guardará cosas como: ¿Cómo puedo yo sacar los productos camisa de tamaño L?
- Sí, como vimos definiendo la query dentro del find. Ejemplos:

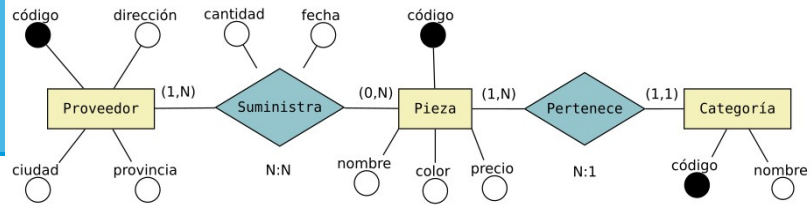
```
return this.usersRepository.find(); //Select * from Users;

return this.usersRepository.findOne({ where: { id } });
//Select * from Users where id='id'

return this.productsRepository.find({select: ['name', 'stock']});
//Select name, stock, from Products

return this.productsRepository.find({relations: ['sizes']});
//Select con INNER JOIN

return this.productsRepository.find({where: {name: 'Silla', stock: 5 }
//Select con where usando AND

return this.productsRepository.find({where: [ {name: 'Silla', stock: 5 }]
//Select con where usando OR

return this.productsRepository.find({ order: { stock: 'DESC',
name: 'ASC' } }); //ordena por stock(desc) y luego name(asc)
```

```
{
   "name": "Camisa Giga",
   "description": "Una camisa gigante",
   "stock": 100,
   "sizes": ["L", "XL", "XXL"]
}
```

Hay que tener presente que find se utiliza para consultas muy simples y en la mayoría de los casos serán sencillas.

¿Pero qué pasa si quiero una consulta avanzada, de esas que visteis en GBD? Una forma que nos recuerda más a SQL sería el uso de **QueryBuilder** (enlace)

TypeORM



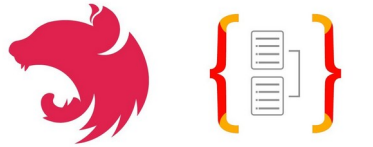
- Pongamos un ejemplo de ambas consultas: **Supongamos que queremos obtener el listado de proveedores y categorías de las piezas que son rojas** **¿Recuerdas cómo sería la SQL?**

```
async getPiezasRojasProveedorCategoria(): Promise<{ proveedor: string;
categoria: string }[]> {
    const piezasRojas = await this.piezaRepository.find({
      where: { color: "rojo" },
      relations: ["proveedorSuministraPiezas",
"proveedorSuministraPiezas.proveedor", "categoria"],
      select: ["proveedorSuministraPiezas", "categoria"],
    })

    return piezasRojas.flatMap((pieza) =>
      pieza.proveedorSuministraPiezas.map((psp) => ({
        proveedor: `${psp.proveedor.codigo} - ${psp.proveedor.direccion}`,
        categoria: `${pieza.categoria.codigo} - ${pieza.categoria.nombre}`,
      })),
    )
  }
```

```
async getPiezasRojasProveedorCategoria(): Promise<{ proveedor: string;
categoria: string }[]> {
    const result = await this.piezaRepository
      .createQueryBuilder("pieza")
      .innerJoin("pieza.proveedorSuministraPiezas", "psp")
      .innerJoin("psp.proveedor", "proveedor")
      .innerJoin("pieza.categoria", "categoria")
      .where("pieza.color = :color", { color: "rojo" })
      .select([
        "proveedor.codigo AS proveedorCodigo",
        "proveedor.direccion AS proveedorDireccion",
        "categoria.codigo AS categoriaCodigo",
        "categoria.nombre AS categoriaNombre",
      ])
      .distinct(true)
      .getRawMany()

    return result.map((item) => ({
      proveedor: `${item.proveedorCodigo} - ${item.proveedorDireccion}`,
      categoria: `${item.categoriaCodigo} - ${item.categoriaNombre}`,
    }))
  }
```

Se está haciendo una unión natural

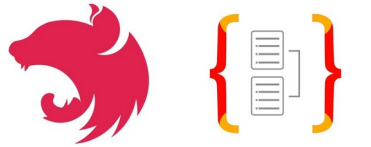
TypeORM



- Fíjate que cada vez que quieres manejar relaciones tienes que controlarlas en el servicio. ¿Esto no se puede automatizar? En TypeORM hay propiedades así, '**eager**' y '**cascade**', que son opcionales.
- **'cascade: true'**: (enlace)
  - Cuando está habilitado, las operaciones realizadas en la entidad principal (User) se propagarán automáticamente a la entidad relacionada (Profile). Es decir, si creas un nuevo User con un Profile asociado, TypeORM guardará automáticamente tanto el User como el Profile.
- **'eager: true'**: (enlace)
  - Cuando está habilitado, la relación se cargará automáticamente cada vez que recuperes la entidad principal de la base de datos. Es decir, cada vez que consultes un User, su Profile asociado se cargará automáticamente.

Se podría pensar…. Pues siempre pongo cascade y eager a true no? NO, aunque su uso simplifica significativamente el código del servicio, también provoca que haya un menor control sobre la operación con la relación entre las entidades. La elección de usarlo o no dependerá de los requisitos específicos de tu aplicación y de cuánto control necesites sobre el proceso de gestión de las relaciones.
Por ejemplo, en un caso de una librería, puede ser interesante usar eager para que dado un autor nos traigamos los libros que tiene escritos a la vez. Este es el caso cuando nos muestran sugerencias de un mismo autor.

TypeORM



- **¿Cuándo se suele usar EAGER y CASCADE?**
  - **Cascade:** Se emplea en creación, update y delete para hacer cambios en cascada. Se puede indicar incluso para la operación a la que se aplica (insert, update o general con true)
  - **Eager:** Permite la carga automática de la parte de la relación, así no hay que definir los relations en el find
- Vamos a ver el caso de create con cascade. Ej: Una pregunta puede tener varias categorías y de cada categoría puede haber varias preguntas

```
async create(text: string, categories: Partial<Category>[]): Promise<Question> {
    const question = new Question();
    question.text = text;

    const savedCategories = await Promise.all(
      categories.map(async (cat) => {
        let category = await this.categoriesRepository.findOne({ where: { name: cat.name } });
        if (!category) {
          category = this.categoriesRepository.create(cat);
          await this.categoriesRepository.save(category);
        }
        return category;
      })
    );
    question.categories = savedCategories;
    return this.questionsRepository.save(question);
  }
```

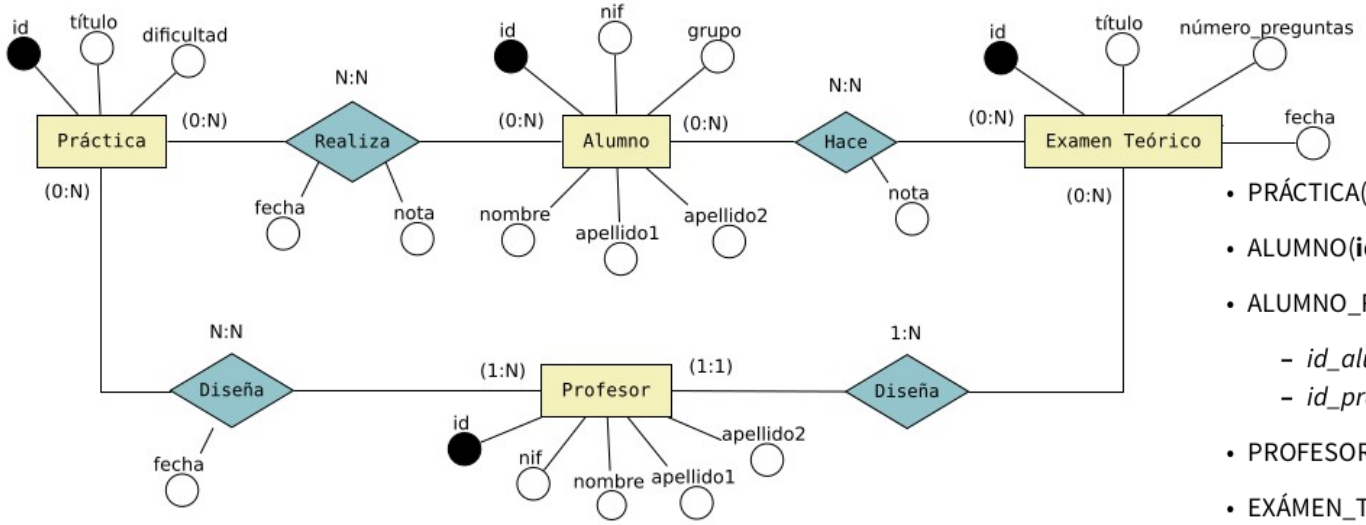
```
async create(text: string, categories: Partial<Category>[]): Promise<Question> {
    const question = new Question();
    question.text = text;
    question.categories = categories.map(cat => {
      const category = new Category();
      category.name = cat.name;
      return category;
    });
    return this.questionsRepository.save(question);
  }
```

Fíjate que como lo hemos hecho hasta ahora, en cuanto creamos una pregunta, le asociamos una categoría que hay que guardar. Con la opción cascade esto se automatiza si las categorías existen, pero... ¿y si no lo están? En ese caso hay que hacer el caso general (enlace)

TypeORM

NestJS + TypeORM

Evaluación: Realiza el siguiente proyecto API para la base de datos dada



Vamos a desarrollar un recurso llamado **evaluación** donde vayamos indicando en diferentes recursos con las entidades, dto, controladores, servicios, módulos,…

Grabarás un vídeo explicando cómo lo has hecho
Mostrarás ejemplos de su funcionamiento

- PRÁCTICA(**id**, título, dificultad)
- ALUMNO(**id**, nif, grupo, nombre, apellido1, apellido2)
- ALUMNO_REALIZA_PRÁCTICA(**id_alumno**, **id_práctica**, fecha, nota)
  - *id_alumno*: FOREIGN KEY de ALUMNO(id)
  - *id_práctica*: FOREIGN KEY de PRÁCTICA(id)
- PROFESOR(**id**, nif, nombre, apellido1, apellido2)
- EXÁMEN_TEÓRICO(**id**, título, número_preguntas, fecha, *id_profesor*)
  - *id_profesor*: FOREIGN KEY de PROFESOR(id)
- ALUMNO_HACE_EXAMEN_TEÓRICO(**id_alumno**, **id_examen_teórico**, nota)
  - *id_alumno*: FOREIGN KEY de ALUMNO(id)
  - *id_examen_teórico*: FOREIGN KEY de EXAMEN_TEÓRICO(id)
- PROFESOR_DISEÑA_PRÁCTICA(**id_profesor**, **id_práctica**, fecha)
  - *id_profesor*: FOREIGN KEY de PROFESOR(id)
  - *id_práctica*: FOREIGN KEY de PRÁCTICA(id)