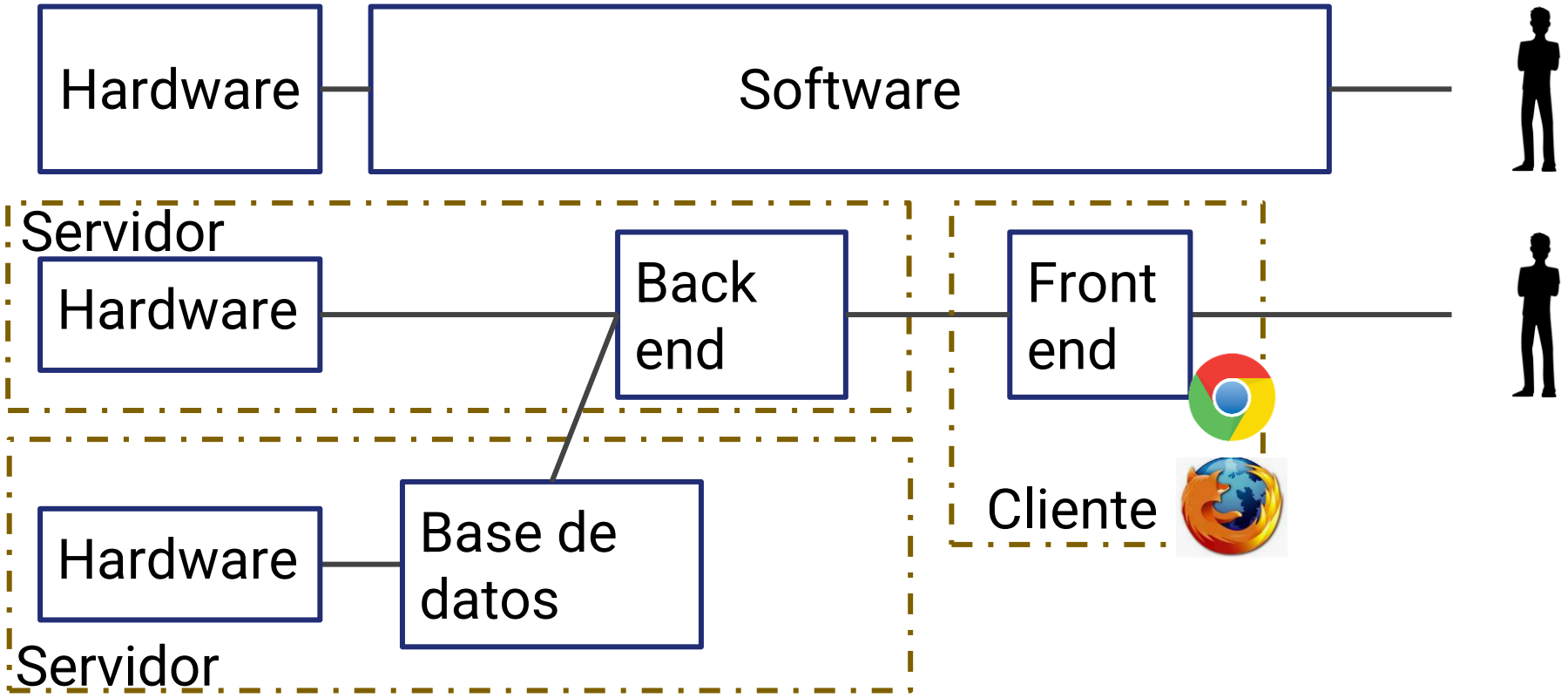
Back End

Carrera Programador full-stack

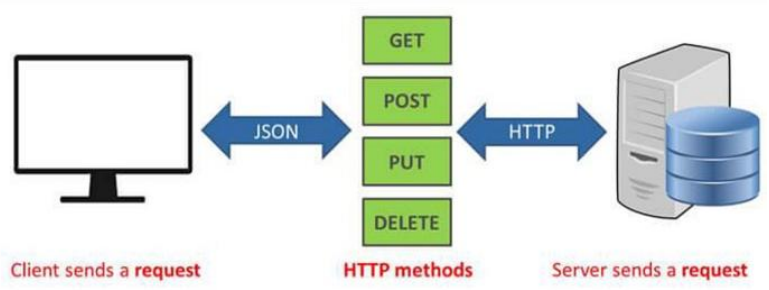
MONTAMOS UNA DB DE PRUEBA
Y LA CONECTAMOS CON NUESTRA API

Arquitectura de Sistemas Web



Cliente

Se encarga de manejar la interfaz gráfica del usuario



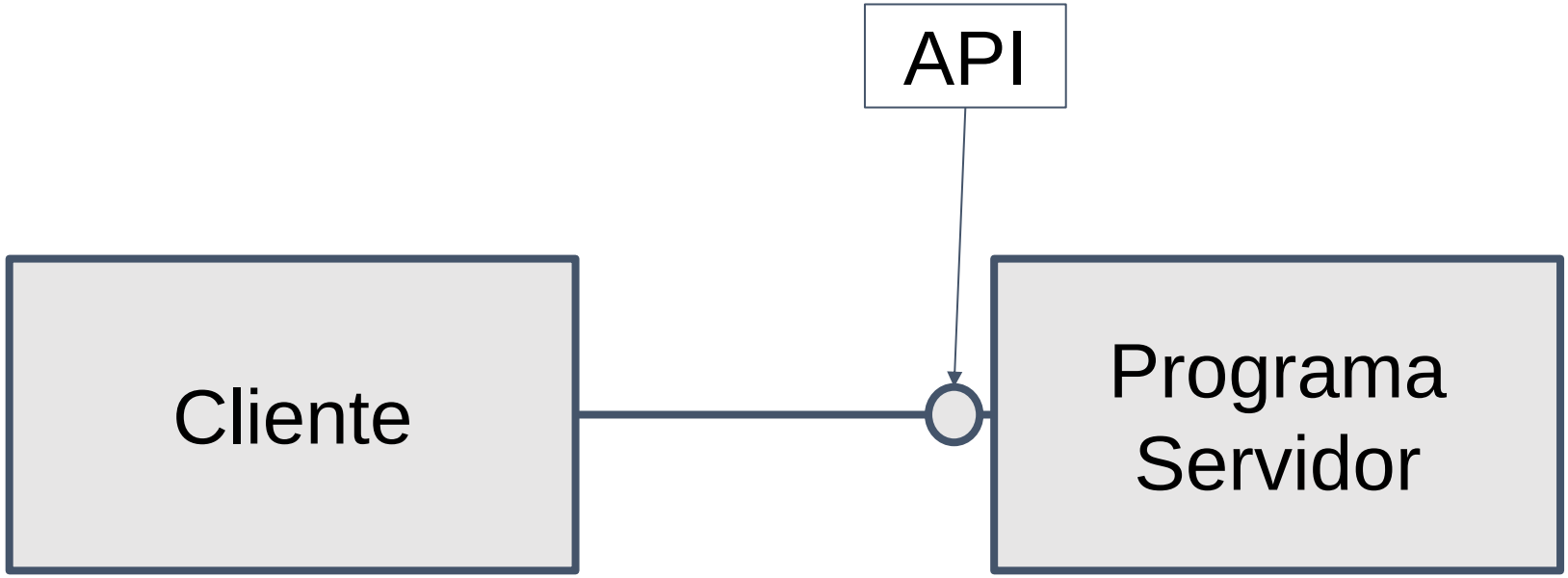
Servidor

Se encarga de:

- Almacenar los datos
- Dar seguridad (que cada uno vea solo sus datos, etc)

API

- Una API es una interfaz que nos da una aplicación para comunicarnos con ella



Pasos:

Estos son los pasos que vamos a seguir a través de las clases (el orden puede variar levemente):

1. Crear datos.
2. Traer datos estáticos que están en el servidor con dos endpoints (traer **todos los tracks** y traer **un track** según su id).
3. Modificar datos.
4. Borrar datos.

REST

- REST: Representational State Transfer, es un tipo de arquitectura de desarrollo web que se apoya totalmente en el estándar HTTP.
- Es el tipo de arquitectura más natural y estándar para crear APIs para servicios orientados a Internet.
- La mayoría de las APIs REST usan JSON para comunicarse.

REST

- Se asocian URLs a recursos.
- Al que se puede acceder o modificar mediante los métodos del protocolo HTTP.
- Se basa en acciones (llamadas verbos) que manipulan los datos.
  - POST: Crear un recurso
  - GET: Obtener uno o muchos recursos
  - PUT: Actualizar uno o muchos recursos
  - DELETE: Borrar un recurso
- Se utilizan los códigos de respuesta del protocolo HTTP.
  - 200 ok, 404 not found, etc.

API REST - EJEMPLO

- GET /pistas (en genérico /pistas)
  - Acceder al listado de pistas
- POST /pistas (en genérico /pistas)
  - Crear una pista nueva
- GET /pistas/123 (en genérico /pistas/:identificador)
  - Acceder al detalle de **una** pista
- PUT /pistas/123 (en genérico /pistas/:identificador)
  - Editar la pista, sustituyendo la **totalidad** de la información anterior por la nueva.
- DELETE /pistas/123 (en genérico /pistas/:identificador)
  - Eliminar la pista

Manejo de errores en REST

Se pueden utilizar los errores del protocolo HTTP:

- 200 OK Standard response for successful HTTP requests
- 201 Created
- 202 Accepted
- 301 Moved Permanently
- 400 Bad Request
- 401 Unauthorized
- 402 Payment Required
- 403 Forbidden
- 404 Not Found
- 405 Method Not Allowed
- 500 Internal Server Error
- 501 Not Implemented

Servidor estático vs dinámico

- Mostrar un archivo estático (sea HTML, CSS, JS, o JSON) podemos hacerlo con sitios estáticos como hemos hecho
- Pero… en algún momento necesitamos ejecutar código en el servidor

Rutas

Dijimos que las APIs responden a diferentes endpoint que eran diferentes URLs

Repasemos!

Pretty URLs

- No puedo tener un archivo HTML para cada pista.
- Necesitamos que nuestro servidor sepa entender (a.k.a. parsear) la URL
- En base a la ruta elige qué código ejecutar
- Algunas partes van a ser “parámetros” (el código de pista 123 por ejemplo)

Pretty URLs

URLs semánticas (amigables o *pretty urls*)

- Fáciles de **entender** para los usuarios
- Mejoran el **posicionamiento** web
- Proporcionan información sobre la **estructura** del sitio
- Fáciles de **comunicar**, ej: whatsapp, llamada, divulgación
- Más **estéticas**, ej: imprimirlas en folletos, facebook, etc.

Ej: usuario en Twitter: https://twitter.com/starwars

Routing

- Seleccionar el PATH a donde redireccionaremos
- Consiste en poder usar los datos de la URL como si fueran parámetros que se le pasan al servidor (Recuerdan el enrutamiento en React?)
- Entonces podemos usarlos como queramos
- Implica romper la lógica de “cada URL es un archivo”

Que vamos a crear?

Vamos a agregar un servicio a nuestro proyecto Nest que nos provea un listado de audios.

Endpoint: http://localhost:3000/tracks

Creación de componentes

1. **controladores** (dónde está el recurso y quién lo maneja).
2. **servicios** manejan la lógica de negocios.

Vamos a usar la generación de código automática de NestJS.

En la raíz del proyecto ejecutamos:

nest generate controller track (o nest g co track)*

nest generate service track (o nest g s track))*

Esto nos crea la carpeta **track** y archivos en ella:

src/track
src/track/track.controller.ts
src/track/track.service.ts

"generateOptions": { "spec": false }

En nest-cli.json:
Evita que nest nos cree los archivos .spec

*el modificador --dry-run simula y muestra los pasos pero no crea ningún archivo…

Gracias a que hemos creado el controlador y el servicio por medio del CLI ambos se han registrado automáticamente dentro del módulo principal de la aplicación. Además, el controlador se ha añadido al array de "**controllers**" y el servicio se agregó a los “**providers**”, todo ello en el decorador @**Module**.

src > TS app.module.ts > ...

```
1  import { ServeStaticModule } from '@nestjs/serve-static';
2  import { join } from 'path';
3  import { Module } from '@nestjs/common';
4  import { AppController } from './app.controller';
5  import { AppService } from './app.service';
6  import { TrackController } from './track/track.controller';
7  import { TrackService } from './track/track.service';
8
9  @Module({
10   imports: [
11     ServeStaticModule.forRoot({ rootPath: join(__dirname, '..', 'client') }),
12   ],
13   controllers: [AppController, TrackController],
14   providers: [AppService, TrackService],
15 })
16 export class AppModule {}
```

@decoradores

Ya que hemos introducido el concepto de decorador, vamos a explicarlo brevemente:

- En general, un **decorador** es un patrón de software que se utiliza para alterar el funcionamiento de una determinada pieza de código; ya sea una función o una clase, sin la necesidad de emplear otros mecanismos como la herencia

- En **NestJS**, los **decoradores** son funciones especiales que se usan para **agregar metadatos** a clases, métodos, propiedades o parámetros. Estos metadatos permiten que Nest entienda cómo debe comportarse esa parte del código.

NestJS los usa para:

- Definir **controladores**, **servicios**, **módulos**, etc.
- Manejar **rutas**, **inyección de dependencias**, **validaciones**, etc.

Backend...

I need more coffee

Aunque anteriormente definimos una interfaz **iTrack** (eso está muy bien) tenemos el código dentro del servicio (eso no está tan bien).

¿Qué demonios era una interfaz?

En la mayoría de los lenguajes de POO es una especie de clase abstracta que contiene declaraciones de métodos sin su implementación. Podemos crear clases que implementan esas interfaces, de modo que el compilador indique errores en tiempo de desarrollo.

En la práctica podemos usar una interfaz para crear un nuevo tipo de datos, que nos sirva para tipar más adelante las entidades que necesitemos y ser ayudados en el tiempo de desarrollo gracias a esos tipos.

Cómo definir una interfaz en Nest

Como otras veces, nos podemos ayudar del CLI para crear las interfaces en Nest. Las interfaces las podemos generar con el siguiente comando:

```
nest g interface track
```

A continuación…

En lugar de usar el arreglo en memoria que contiene los objetos de tipo track, vamos a generar un archivo en nuestro servidor programado en TS y Nest.

Ese archivo simulará una base de datos que va a contener todos nuestros recursos.

Para esto vamos a utilizar la dependencia **JSON Server**.

Mockear una API

Primero vamos a mockear nuestra API

Mockear significa “reemplazar una implementación real por una falsa y simple”

Como las API devuelven JSON vamos a hacer nuestro archivo JSON estático

Para qué *mockeamos*?

Nos permite trabajar con el *front end* aunque nuestro *back end* no tenga implementada la conexión con la base de datos.

1. Creamos el archivo **`/data/tracks.json.`**

Como imaginará, en este módulo copiaremos el arreglo de tracks. (**Ojo con el formato**, debe ser un **JSON**, no un objeto JavaScript)

2. Instalamos json-server: **`npm i json-server –D`**

3. Creamos un script en el archivo package.json para montar el servicio de json-server:

`"db": "json-server data/tracks.json --watch --port 3030“`

4. Montamos la mock database con: **`npm run db`**

5. Navegamos al puerto local donde montamos la DB

One last push!!

Todo está listo para el último paso de este proceso. A través del **controlador** y el **servicio** de **tracks haremos lo siguiente:**

1. Nuestro *backend* recibirá una **request**
2. Se conectará a la **base de datos** de prueba
3. Devolverá una **response** con los datos pertinentes

TS track.controller.ts M

src > track > TS track.controller.ts > ...

```ts
import { Get, Controller } from '@nestjs/common';
import { TrackService } from './track.service';
import { Track } from './track.interface';

@Controller('tracks')
export class TrackController {
  constructor(private readonly trackService: TrackService) {}
  @Get()
  getTracks(): Promise<Track[]> {
    return this.trackService.getTracks();
  }
}
```

TS track.service.ts M

src > track > TS track.service.ts > ...

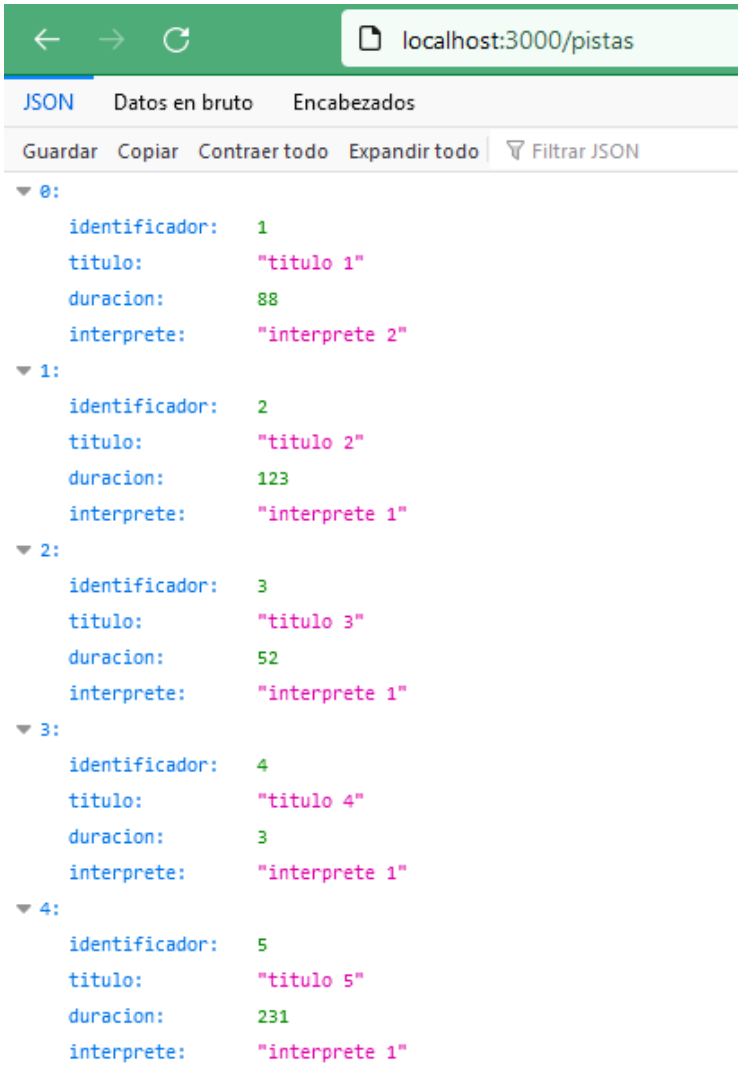
```ts
import { Injectable } from '@nestjs/common';
import { Track } from './track.interface';
const BASE_URL = 'http://localhost:3030/tracks';
@Injectable()
export class TrackService {
  async getTracks(): Promise<Track[]> {
    const res = await fetch(BASE_URL);
    const parsed = await res.json();
    return parsed;
  }
}

```

Probando el servicio

Si corremos nuestro proyecto (**npm run start:dev**) y abrimos la url http://localhost:3000/tracks

Vamos a ver desde el navegador la respuesta del servidor generada en track.service.ts

(Siempre correr la db mockeada y el proyecto a la vez en distinta terminal)



Seeeee…

Abrite una chelaaa!!!