

Back-end

CFP

Programador full-stack

Rutas y APIs

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!

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 errores del protocolo HTTP.
 - 200 ok, 404 not found, etc.

API REST - EJEMPLO

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

API Rest

Las URL exponen **recursos** a través de diferentes direcciones o *endpoints*.

SUSTANTIVOS

Ejemplos de *endpoints* en una aplicación:

- Usuarios → ex.com/usuario/
- Tareas → ex.com/tarea/5
- Imágenes de usuario → ex.com/usuario/2/imagenes

Pretty URLs

- No puedo tener un archivo HTML para cada factura.
- 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 factura 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>

Ejemplo

En lugar de usar el mock con un archivo, ahora vamos a generarlo en nuestro servidor programado en TS y Nest

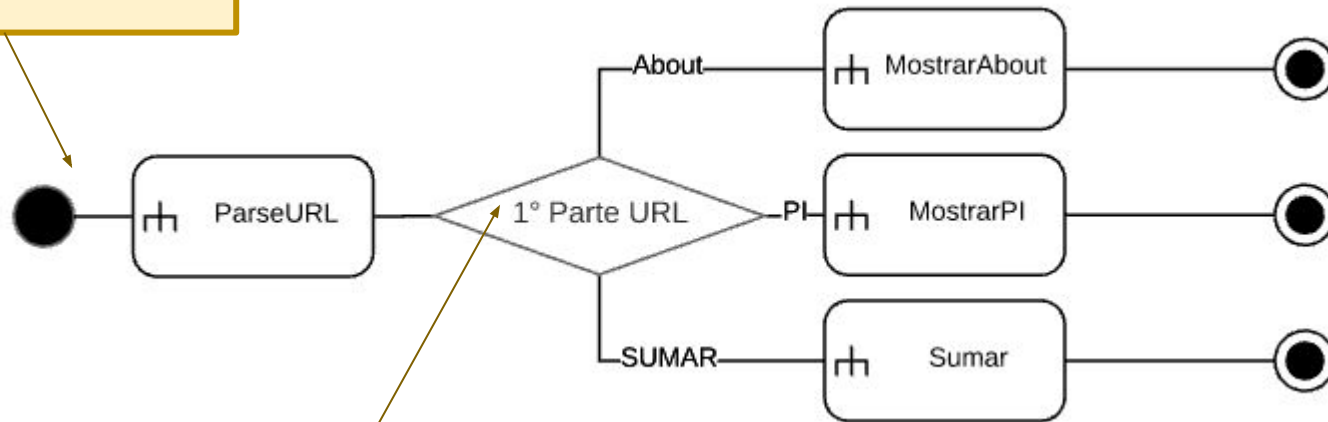
Routing

- 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
- Entonces podemos usarlos como queramos
- Implica romper la lógica de “cada URL es un archivo”

Diagramas de flujo

Estado inicial
(inicio del flujo)

Estado final (fin de flujo)



Condición (IF)
(Equipos -
hardware)

Actividades
(código
ejecutado)

Que vamos a crear?

Vamos a agregar un servicio a nuestro proyecto nest que nos provea un listado de productos.

Endpoint: <http://localhost:3000/productos>

Para esto, dentro de la carpeta src vamos a crear una nueva (producto) que contenga el controller y el servicio que se van a encargar de manejar el request.

Crear un servicio

Dentro de nuestro proyecto nest, en la carpeta src/producto creamos un archivo llamado producto.service.ts que contenga lo siguiente:

```
import { Injectable } from '@nestjs/common';
@Injectable()
export class ProductoService {
  private static readonly CANTIDAD_PRODUCTOS = 10;
  public getProducto(): any {
    let productos = [];
    for (let i = 0; i < ProductoService.CANTIDAD_PRODUCTOS; i++) {
      let producto = {
        'producto_nombre': 'producto_' + i,
        'precio': Math.floor(Math.random() * 100)
      };
      productos.push(producto);
    }
    return productos;
  }
}
```

Crear un Controller

Dentro de nuestro proyecto nest, en la carpeta src/producto creamos un archivo llamado producto.controller.ts que contenga lo siguiente:

```
import { Controller, Get } from '@nestjs/common';
import { ProductoService } from '../producto.service';
@Controller('productos')
export class ProductoController {
  constructor(private productoService: ProductoService) {}
  @Get()
  public getProducto(): string {
    return this.productoService.getProducto()
  }
}
```

Modificar app.module.js

Ahora vamos a agregar el controller y el servicio a nuestro módulo, donde se servían archivos estáticos

Se importan los archivos creados:

```
import { ProductoController } from './producto/producto.controller';  
import { ProductoService } from './producto/producto.service';
```

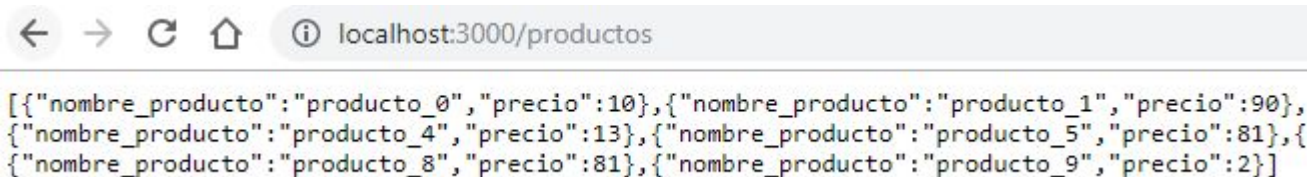
Se agregan en @Module como controller y provider:

```
controllers: [AppController, ProductoController],  
providers: [AppService, ProductoService],
```

Probando el servicio

Si corremos nuestro proyecto (npm run start:dev) y abrimos la url <http://localhost:3000/productos>

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



A screenshot of a web browser window. The address bar shows the URL 'localhost:3000/productos'. The main content area displays a JSON array of product objects. The array contains five objects, each with 'nombre_producto' and 'precio' fields. The products are: producto_0 (precio: 10), producto_1 (precio: 90), producto_4 (precio: 13), producto_5 (precio: 81), and producto_8 (precio: 81). The last object in the array is partially cut off.

```
[{"nombre_producto": "producto_0", "precio": 10}, {"nombre_producto": "producto_1", "precio": 90}, {"nombre_producto": "producto_4", "precio": 13}, {"nombre_producto": "producto_5", "precio": 81}, {"nombre_producto": "producto_8", "precio": 81}, {"nombre_producto": "producto_9", "precio": 2}]
```


Llamando al servicio desde el Front

Hasta ahora usábamos solo un mock para simular la respuesta del servidor, ahora vamos a cambiar la url en el fetch para llamar al servicio que creamos

```
async function load(){  
  let r = await fetch('/productos');  
  compras = await r.json();  
  mostrarTablaCompras();  
}
```

Ejercicios

- 1 - Modificar el servicio existente para que además de devolver el nombre del producto y el precio también devuelva una descripción.
- 2 - El servicio debe pasar de generar la información de manera aleatoria a devolver datos que se encuentren guardados en memoria (Arreglos).

Repaso - Manejo de Archivos de Texto

- Instalar paquete → `npm install @types/node`
- Crear archivo 'abc.txt'
 - Escribir adentro 'hola como andas todo bien'

```
import * as fs from 'fs';

let texto: string = fs.readFileSync('abc.txt', 'utf8');

let palabras: string[] = texto.split(' ');

console.log(palabras);
```

Nombre del archivo que vamos a leer

```
PS C:\Users\Francisco\Documents\CFP\3. POO\Ejercicios> tsc ejemplo-txt.ts
PS C:\Users\Francisco\Documents\CFP\3. POO\Ejercicios> node ejemplo-txt.js
[ 'hola', 'como', 'andas', 'todo', 'bien' ]
```