PROYECTO FINAL: OPCIÓN 1

Proyecto de desarrollo de Angular con API Rest NodeJS Express

Por Álvaro Ley

Lo primero que haremos será crear la carpeta que nos pide el ejercicio, la cual se denomina según mi número de matrícula, mi nombre y mis apellidos. En mi caso, por lo tanto, se la carpeta raíz de mi proyecto se titula "9513418_AlvaroLey_ProyectoFinal".

1. Creación y desarrollo de API Rest con NodeJS

1.1. Creación de proyecto NodeJS y configuración de "package.json"

El primer paso será crear nuestro proyecto **NodeJS** con el comando `npm init`, personalizando la configuración de nuestro archivo "package.json" según nuestro criterio.

A continuación, para reiniciar automáticamente nuestro servidor con cada cambio en el código, instalamos la herramienta **Nodemon** con el comando `npm install nodemon --save-dev`, y posteriormente, en el "package.json", lo incluimos en la sección de "scripts".

Instalamos también el framework **Express**, que lo necesitaremos para crear la **API Rest**. Para ello introducimos en la terminal el comando `npm install express`.

El archivo "package.json" nos queda entonces de la siguiente manera:

```
{
  "name": "9513418_alvaroley_proyectofinal",
  "version": "1.0.0",
  "description": "Proyecto Final ApiRest Angular NodeJS Express",
  "main": "app.js",
  ▷ Depurar
  "scripts": {
    "start": "nodemon app",
    "dev": "nodemon app.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "ApiRest",
    "Angular",
    "NodeJS",
    "Express"
  ],
  "author": "ALG",
  "license": "ISC",
  "devDependencies": {
    "nodemon": "^3.1.7"
  },
  "dependencies": {
    "@angular/common": "^18.2.12",
    "cors": "^2.8.5",
    "express": "^4.21.1"
  }
}
```

1.2. Creación de servidor Express y punto de entrada de la API

En el siguiente paso, vamos a crear un archivo “app.js” en la raíz del proyecto, que será nuestro será crear nuestro punto de entrada para la **API Rest**. En este, configuramos un servidor básico de **Express**. Incluimos el *middleware* `use()` para parsear el **JSON**.

```
const express = require('express');
const app = express();
const port = 3000;

app.use(express.json());

app.listen(port, () => {
    console.log(`Servidor escuchando en http://localhost:${port}`);
});
```

2. Creación de enrutamiento y desarrollo de peticiones CRUD

2.1. Creación de archivo de enrutamiento para una entidad

Ahora creamos en la raíz del proyecto una carpeta que denominaremos “routes” donde ubicaremos el archivo de enrutamiento para la entidad “proveedor” que nos pide el ejercicio, y que denominamos “providers.js”. Este es el archivo donde desarrollaremos todas las peticiones **CRUD** (*Create*, *Read*, *Update* y *Delete*) de nuestra aplicación.

En él, inicialmente, importamos **Express**, y para trabajar de una forma modular y organizada, asignamos el *middleware* `express.Router()` a una constante desde la cual manejar las rutas, que será la que exportaremos. Esto nos permite encapsular las rutas en un solo archivo, manteniendo el archivo principal “app.js” limpio y fácil de entender.

```
const express = require('express');
const router = express.Router();
```

2.2. Creación de array de proveedores

Para almacenar los proveedores, creamos el array “providers”, y le agregamos un objeto con las propiedades que nos indica el ejercicio, con unos valores a modo de ejemplo.

```
const providers = [
    {
        "cif": "12345",
        "name": "Proveedor A",
        "activity": "Suministros",
        "address": "Calle 123",
        "city": "Ciudad X",
        "cp": "28001",
        "phone": "123456789"
    }
];
```

2.3. Petición “get” para mostrar todos los proveedores del array

Empezamos con una petición “get” que muestre todos los elementos del array, y le añadimos una validación que detecte cuando no exista proveedor alguno en la lista.

```
router.get('/', (req, res) => {
    if (providers.length === 0) {
        return res.status(404).json({
            message: "No hay proveedores"
        });
    }

    return res.status(200).json({
        message: "Lista de proveedores",
        providers
    });
});
```

2.4. Petición “get” para mostrar un proveedor según su cif

Creamos otra petición “get” que devuelva el proveedor que ingrese el usuario en la **URL** según su “cif”. Para facilitarnos las cosas, creamos una constante `urlCif` para almacenar el parámetro de la **URL**, que lo obtenemos con `req.params.cif`; y otra constante `existingProvider` a la que asignamos el valor del proveedor que contenga un “cif” igual que nuestra constante `urlCif`, filtrando el array con el método `find()`, y que en caso de no encontrar ninguna coincidencia, no se asignaría objeto alguno a esa constante, y quedaría vacía.

Ahora implementamos una validación para cuando el valor introducido no corresponda al “cif” de ningún proveedor ya existente en la lista. Para validar esto, simplemente podríamos usar el operador de negación `!` delante de la constante `existingProvider` para comprobar si su valor booleano es `false`, lo que significaría que está vacía. Si es así, se lanzará el estado **404**.

Si existe ese cif en algún proveedor de la lista, devolverá ese proveedor con el estado estándar de éxito **200** y su correspondiente mensaje.

```
router.get('/:cif', (req, res) => {
    const urlCif = req.params.cif;
    const existingProvider = providers.find(p => p.cif === urlCif);

    if (!existingProvider) {
        return res.status(404).json({
            message: "Proveedor no encontrado"
        });
    }

    return res.status(200).json({
        message: "Proveedor encontrado",
        existingProvider
    });
});
```

2.5. Añadir un proveedor con una petición “post”

Continuamos nuestro archivo de rutas configurando ahora una petición “post” para añadir un proveedor a la lista. Para crear las validaciones, nos vendrá bien en primer lugar usar la desestructuración de objetos para extraer la propiedad “cif” del `req.body` en una constante `cifBody`, y en segundo lugar usar el método `find()` para buscar en el array un “cif” cuyo valor sea igual que el introducido, para almacenarlo en otra constante `cifInUse`, y que en el caso de no existir coincidencias, quedará vacía.

La primera validación comprobará que el objeto recibido contenga almenos la propiedad “cif”, que es indispensable para realizar algunas peticiones. Si no la contuviera, la constante estaría vacía, y por lo tanto podemos verificar esto de nuevo con el operador de negación `!cifBody`. En ese caso, devolvería el estado **400** con un mensaje de error.

La segunda validación comprobará que el “cif” ingresado no esté siendo usado ya por otro proveedor, lo que provocaría un conflicto. Para esto simplemente volvemos a usar el operador de negación para comprobar si la constante `cifInUse` está vacía. En ese caso, el estado que se lanzaría sería el **409**, que corresponde al de “conflicto”

Si superase las dos validaciones anteriores, almacenamos el cuerpo de la petición en una constante, utilizamos el método `push()` para agregar el objeto a la lista, y retornamos el estado **201** para confirmar que se ha creado un nuevo proveedor.

```
router.post('/', (req, res) => {
    const { cif: cifBody } = req.body
    const cifInUse = providers.find(p => p.cif === cifBody);

    if (!cifBody) {
        return res.status(400).json({
            message: "El proveedor añadido debe incluir un CIF"
        });
    } else if (cifInUse) {
        return res.status(409).json({
            message: "Este CIF está siendo usado por otro proveedor"
        });
    }

    const newProvider = req.body;
    providers.push(newProvider);

    return res.status(201).json({
        message: "Proveedor creado",
        newProvider
    });
});
```

2.6. Modificar un proveedor con una petición “put”

Ahora vamos a crear una petición “put”, por la cual modificamos un proveedor, indicándole en la **URL** el “cif” para identificarlo. Para simplificar el código, al igual que en nuestro segundo método “get”, almacenamos en una constante `urlCif` el parámetro de la **URL** con `req.params`, y en otra `existingProvider` el valor del proveedor que contenga un “cif” igual que nuestra constante `urlCif` (con el método `find()`), y que como hemos dicho antes, en caso de no existir ese “cif” en ningún proveedor de la lista, quedaría vacía.

Ahora creamos una validación (idéntica también a la que hemos desarrollado en el segundo método “get”) que verifica si el valor introducido en la **URL** no corresponde a ningún “cif” de la lista de proveedores, usando `!existingProvider` para comprobarlo.

Después, para comprobar que el nuevo objeto no contiene un “cif” igual al de otro proveedor, lo que provocaría conflicto (a no ser que sea ese mismo, lo cual no sería conflictivo), creamos una constante para almacenar el valor de dicho “cif” en caso de que ya exista, y en caso de que no sea el mismo que se está actualizando. Dicha constante la usaríamos para, en caso de que fuese `true`, lanzar un error **409**.

Si se superan las dos validaciones, entonces, para mayor versatilidad, atrapamos el valor del `req.body` en una constante `updatedProvider` con los valores a actualizar del proveedor; y en otra constante, el número del índice del proveedor que queremos actualizar, usando `indexOf()`. Después, combinamos las propiedades que no se han modificado con las que sí se han modificado, para almacenar el proveedor actualizado. Para ello le indicamos el proveedor a actualizar con el índice que hemos extraído antes, y usamos el operador de propagación `{ … }` para combinar dichas propiedades. Después de este paso, se lanzará el estado de éxito y su correspondiente mensaje.

```
router.put('/:cif', (req, res) => {
    const urlCif = req.params.cif;
    const existingProvider = providers.find(p => p.cif === urlCif);

    if (!existingProvider) {
        return res.status(404).json({
            message: "Proveedor no encontrado"
        });
    }

    const cifInUse = providers.find(p => p.cif === req.body.cif && p.cif !== urlCif);
    if (cifInUse) {
        return res.status(409).json({
            message: "Este CIF está siendo usado por otro proveedor"
        });
    }

    const updatedProvider = req.body;
    const providerIndex = providers.indexOf(existingProvider);
    providers[providerIndex] = { ...existingProvider, ...updatedProvider };

    return res.status(200).json({
        message: "Proveedor actualizado",
        provider: providers[providerIndex]
    });
});
```

2.7. Eliminar un proveedor con una petición “delete”

La última petición de nuestra serie **CRUD** es “delete”, por la cual eliminaremos un proveedor a través del “cif” ingresado en la **URL**.

La validación que vamos a desarrollar aquí es la misma que hemos hecho en la anterior peticion, la cual verifica si el “cif” ingresado corresponde al de algún proveedor de la lista, el código que realizaremos será igual. Almacenamos el `urlCif` en una constante y buscamos con `find()` un proveedor cuyo “cif” sea igual que dicha variable, y lo almacenamos en otra constante, que usaremos para verificar si existe correspondencia.

Después, si no cae en dicha validación, para eliminar el proveedor usamos el método `splice()` compuesto por el índice del proveedor (obtenido a través del método `indexOf()`) y el número de elementos que queremos eliminar (1). Tras esto, lanzamos el mensaje y el estado de éxito.

```
router.delete('/:cif', (req, res) => {
    const urlCif = req.params.cif;
    const existingProvider = providers.find(p => p.cif === urlCif);

    if (!existingProvider) {
        return res.status(404).json({
            message: "Proveedor no encontrado"
        });
    }

    providers.splice(providers.indexOf(existingProvider), 1);
    return res.status(200).json({
        message: "Proveedor eliminado",
        existingProvider
    });
});
```

2.8. Exportación de rutas e importación en el archivo principal

Como último paso en nuestro archivo de enrutamiento, exportamos la constante router desde la cual hemos creado todas las rutas de nuestra aplicación **Express**.

```
module.exports = router;
```

A continuación la importamos en nuestro archivo principal “app.js”, e implementamos el middleware `use()` para que todas las rutas definidas en él estén accesibles a través del prefijo `/providers` en la URL.

```
const providersRouter = require('./routes/providers');
app.use('/providers', providersRouter);
```

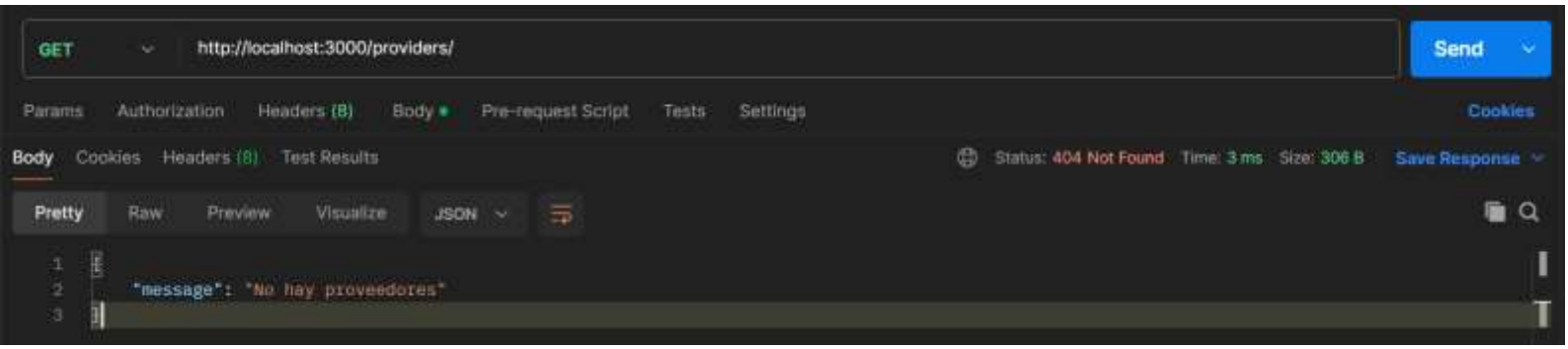
2.9. Comprobación de peticiones en Postman

Para comprobar las peticiones **CRUD** que hemos desarrollado, utilizaremos la herramienta **Postman**. Para llevar esto a cabo, primero debemos introducir el comando `npm start` en la teminal de nuestro proyecto raíz, lo que iniciaría nuestra aplicación.

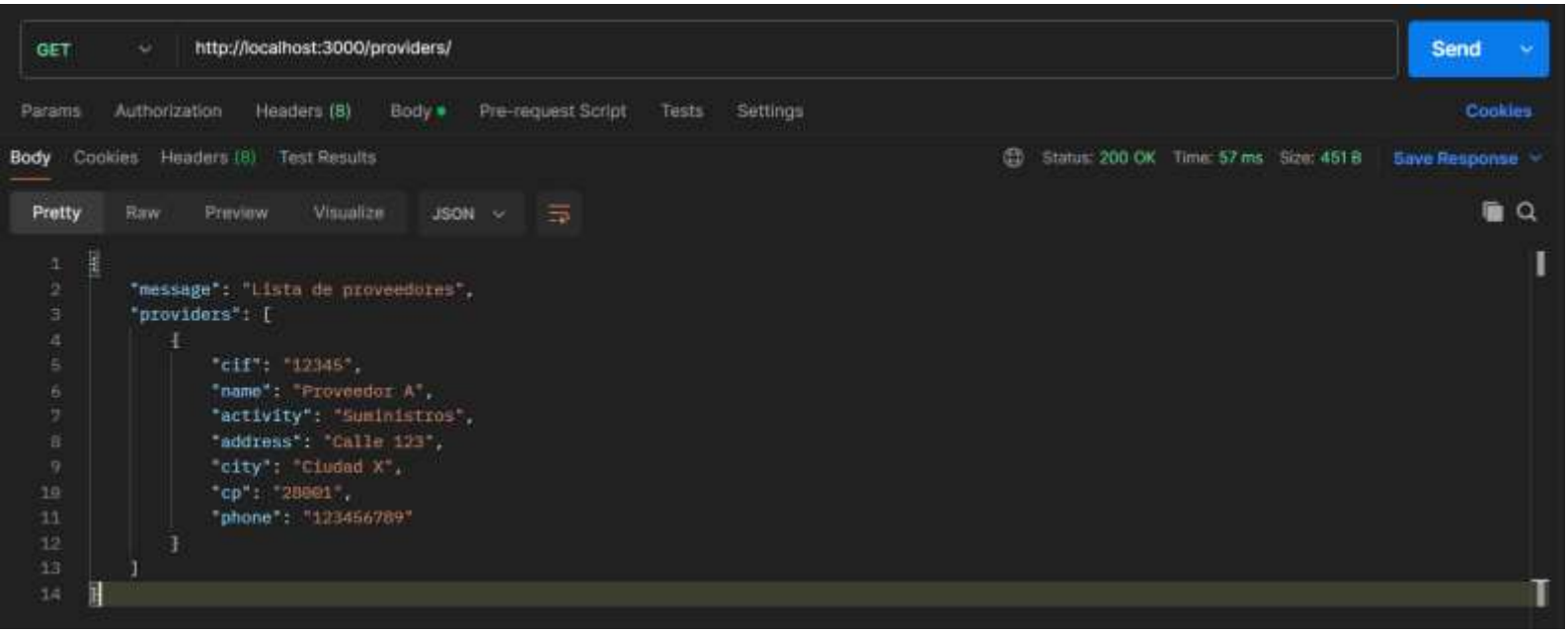
Iremos comprobando las peticiones según el orden en el que las hemos desarrollado, con sus correspondientes validaciones.

2.9.1. Petición “get”

Para poder comprobar la validación de “get”, tenemos que eliminar momentáneamente el proveedor que hemos incluido a modo de ejemplo en el archivo “providers.js”.

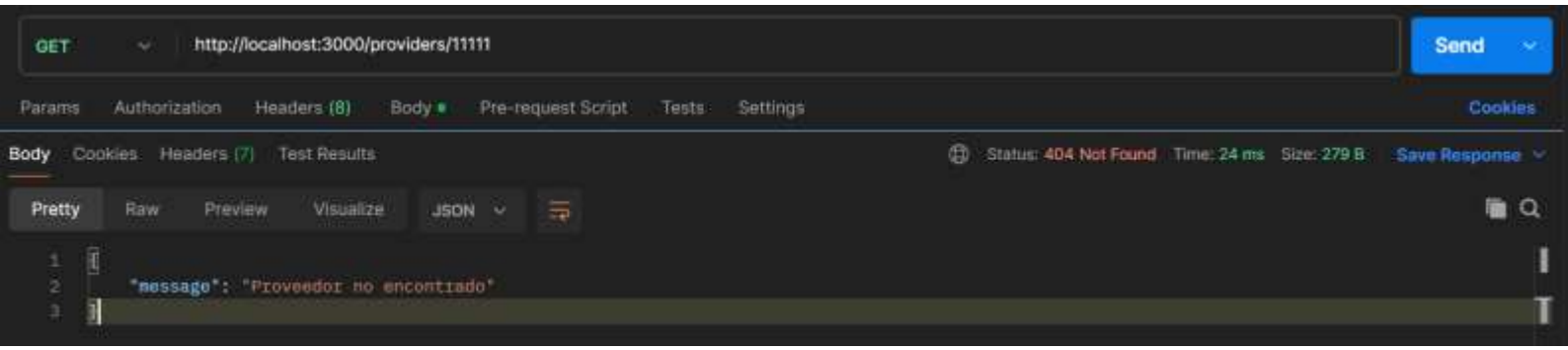


Tras esto, volvemos a incluir nuestro proveedor ejemplo en el código, para poder obtenerlo con nuestra primera petición.

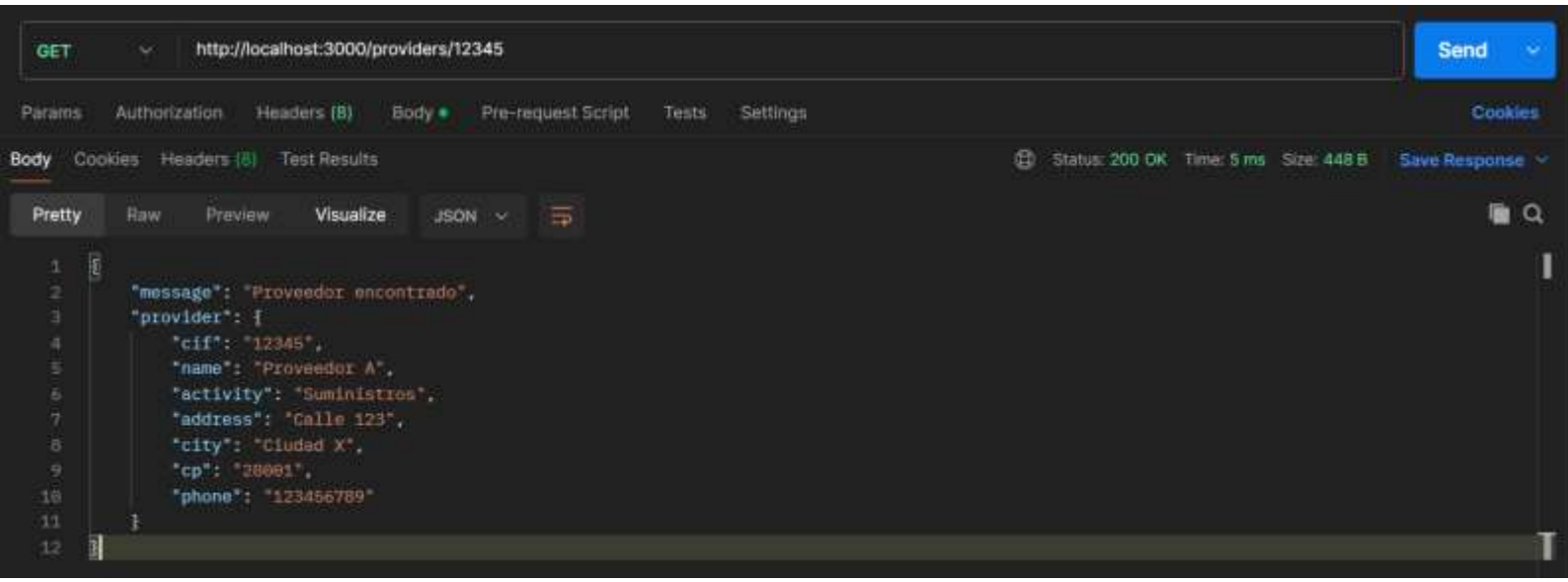


2.9.2. Petición “get” a través de la propiedad “cif”

Comprobamos esta petición, empezando por introducir un “cif” inexistente para obtener el error de la validación.



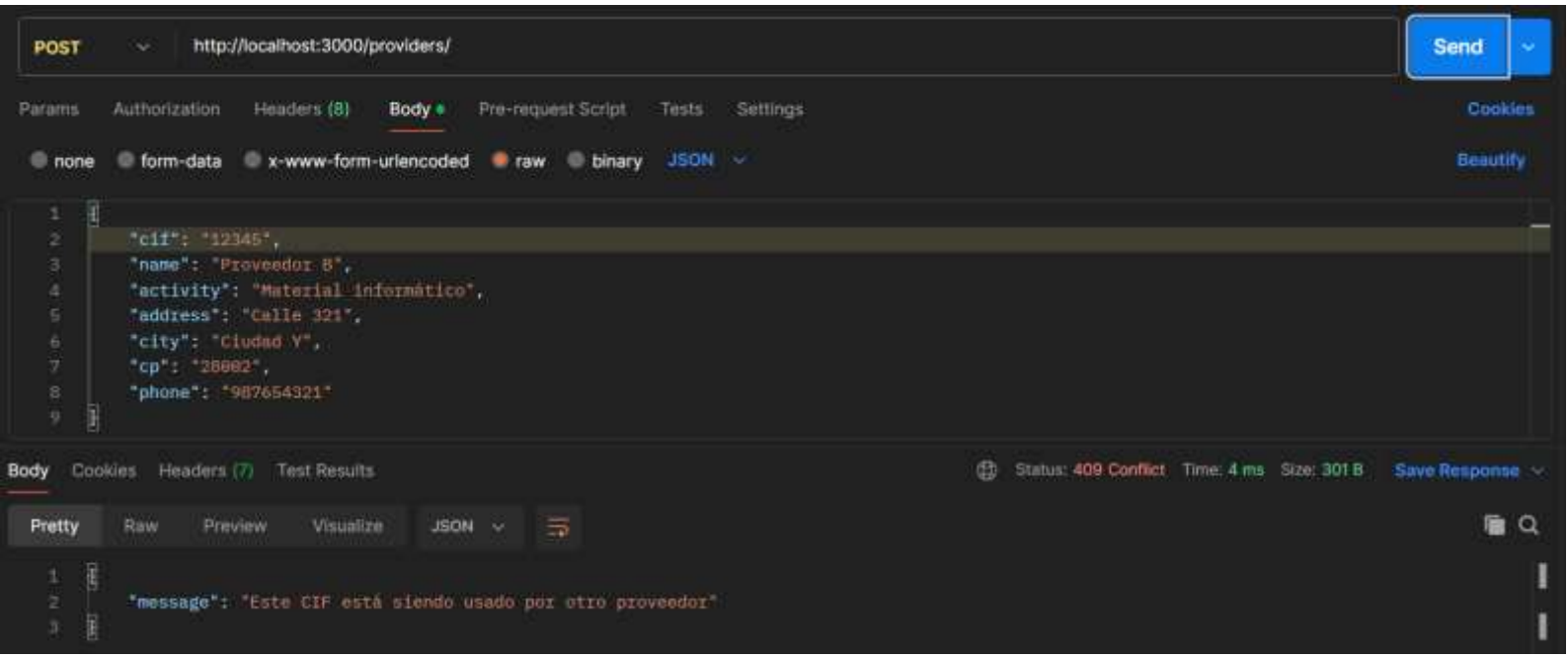
Y después ingresamos el existente, para comprobar que funciona correctamente.



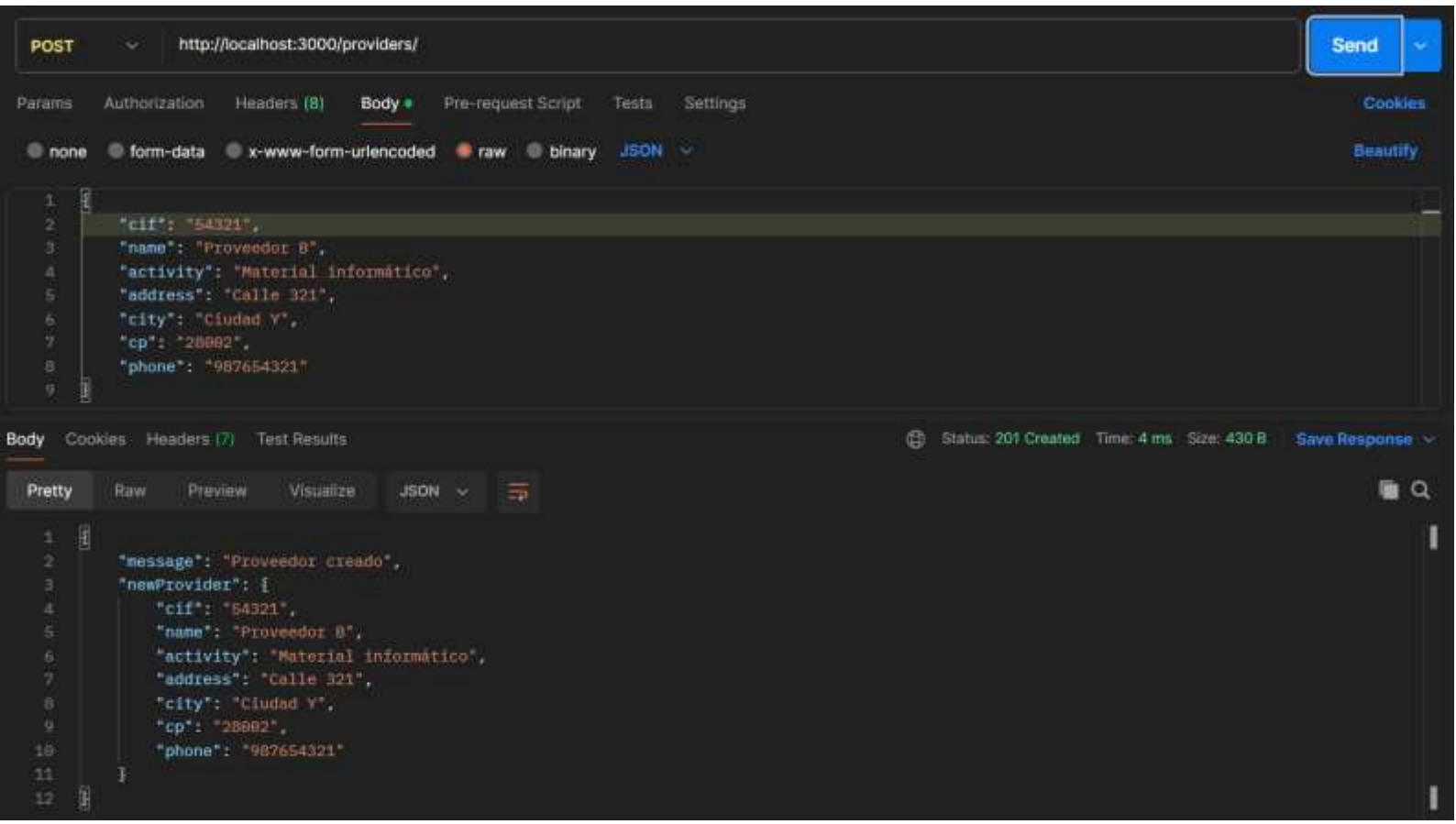
2.9.3. Petición “post”

Vamos ahora con la comprobación de de la petición “post”. Primero ingresamos un proveedor sin la propiedad “cif”.

Ahora introducimos un “cif” que ya existe en otro proveedor, para comprobar la segunda validación.

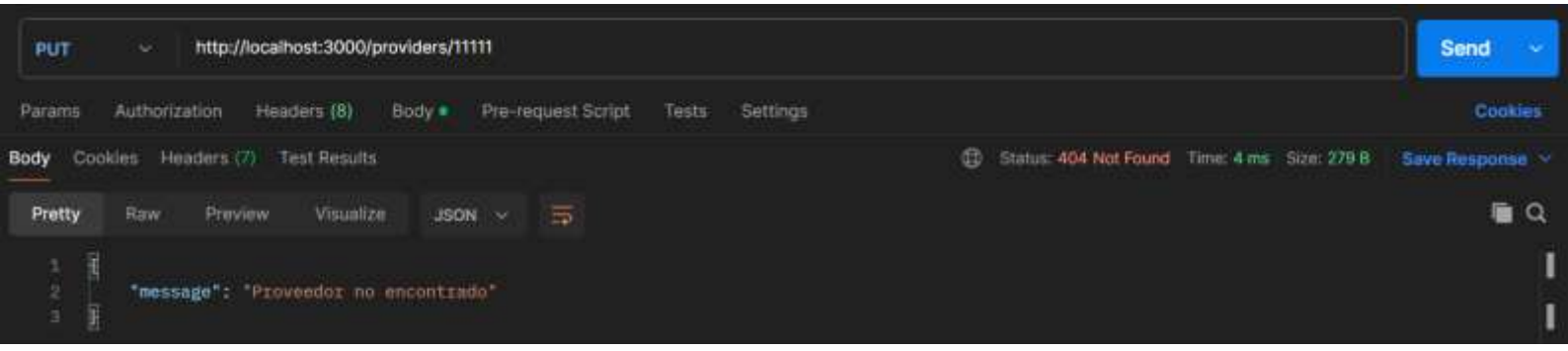


Y finalmente ingresamos un “cif” que no está siendo usado por otro proveedor, lo que nos añade el nuevo proveedor exitosamente.

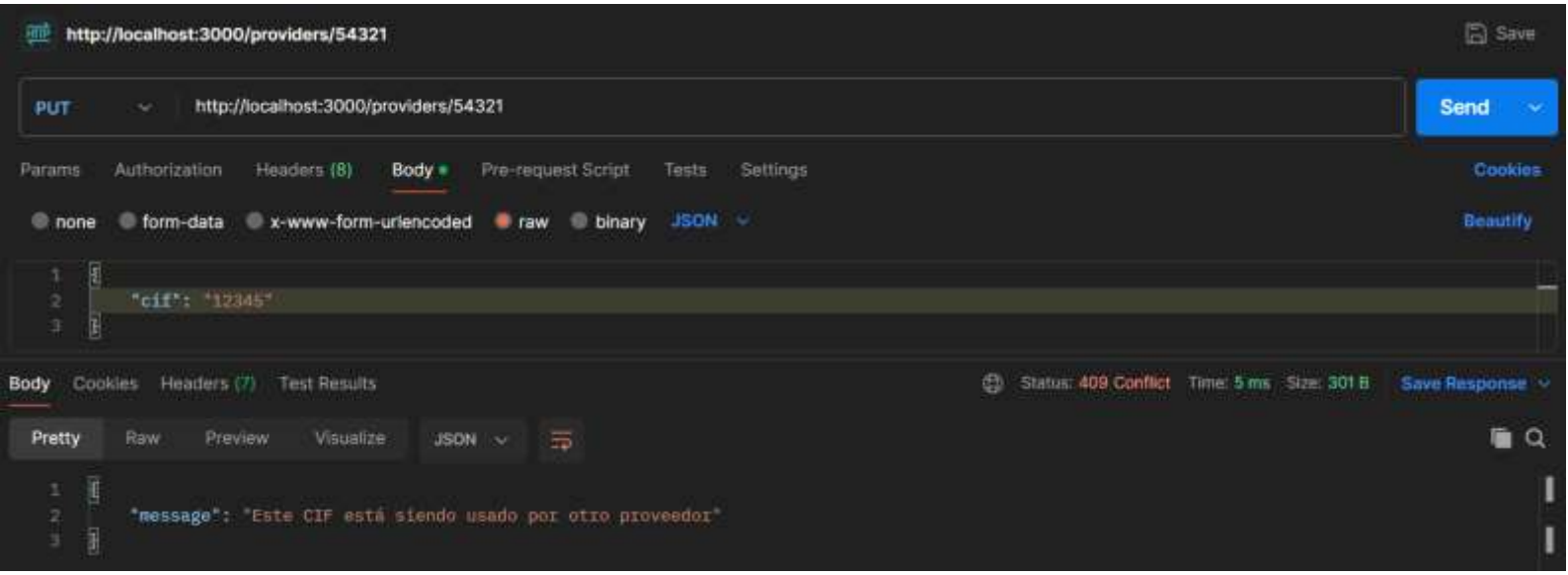


2.9.4. Petición “put”

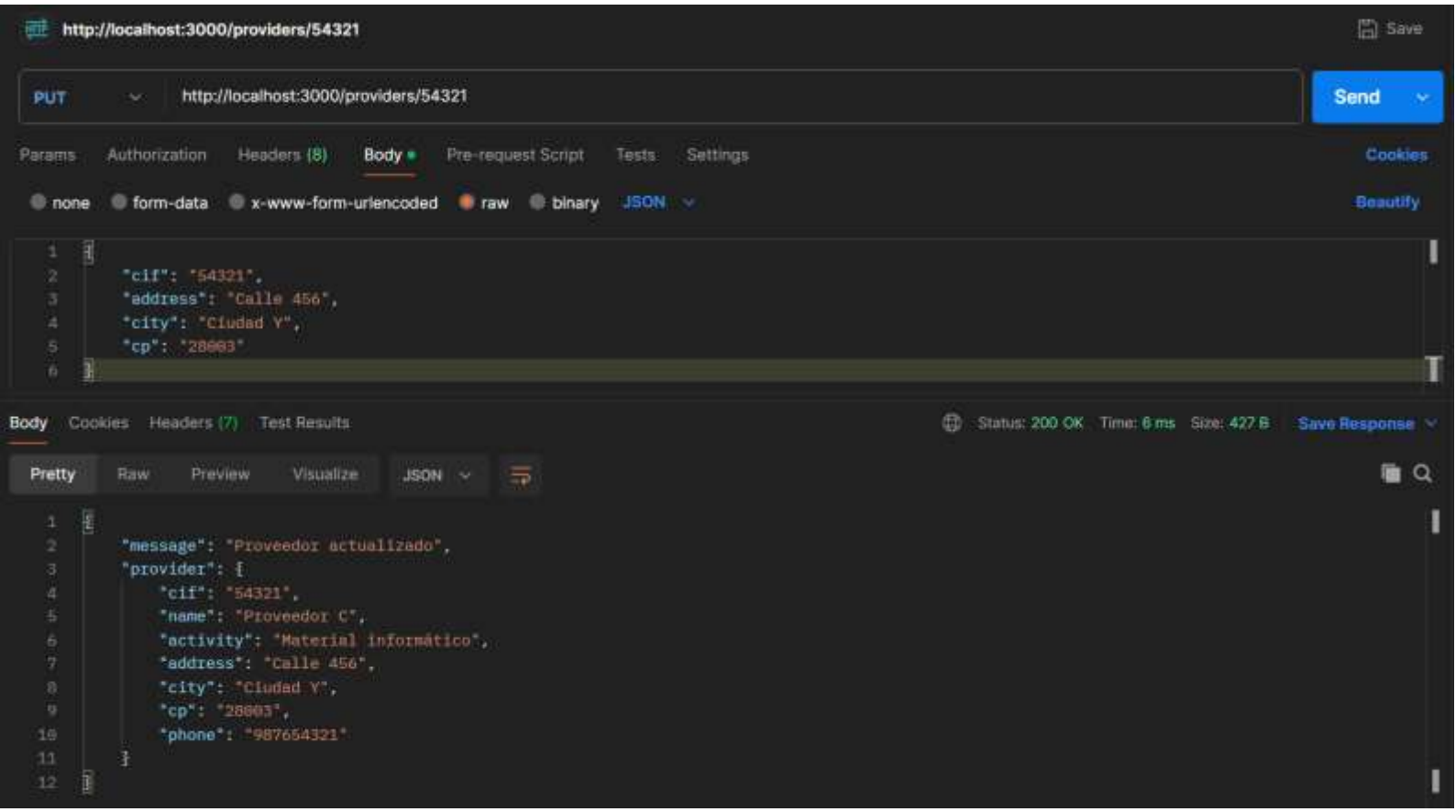
Es el momento de comprobar la petición “put”, por la que queremos modificar un proveedor. Primero, comprobemos las validaciones. Ingresamos una **URL** que no corresponde con el “cif” de ningún proveedor, para comprobar la primera.



Para comprobar la siguiente validación, ingresamos el “cif” de un proveedor existente en la **URL**, fingiendo que queremos modificar precisamente su “cif”, pero introducimos en el cuerpo **JSON** de la petición uno que ya posee otro proveedor.

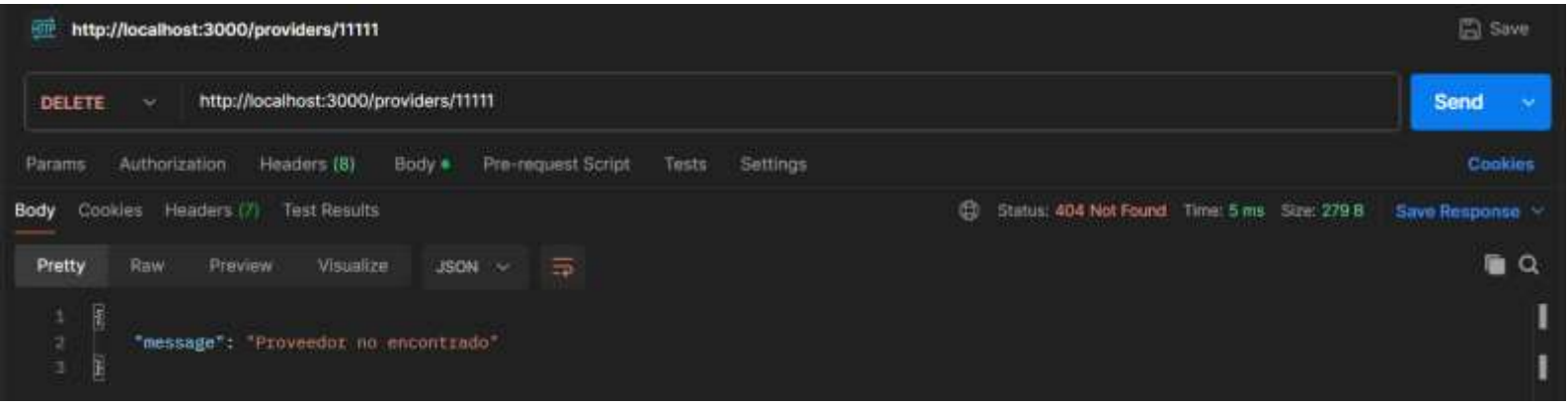


A continuación, comprobamos el correcto funcionamiento de la petición “put”, modificando algunas propiedades de un proveedor. Aprovechamos para introducir el mismo “cif” que tiene el proveedor, cosa que no tendría por qué ser conflictiva, para ver cómo actualiza igualmente el proveedor, de forma correcta.

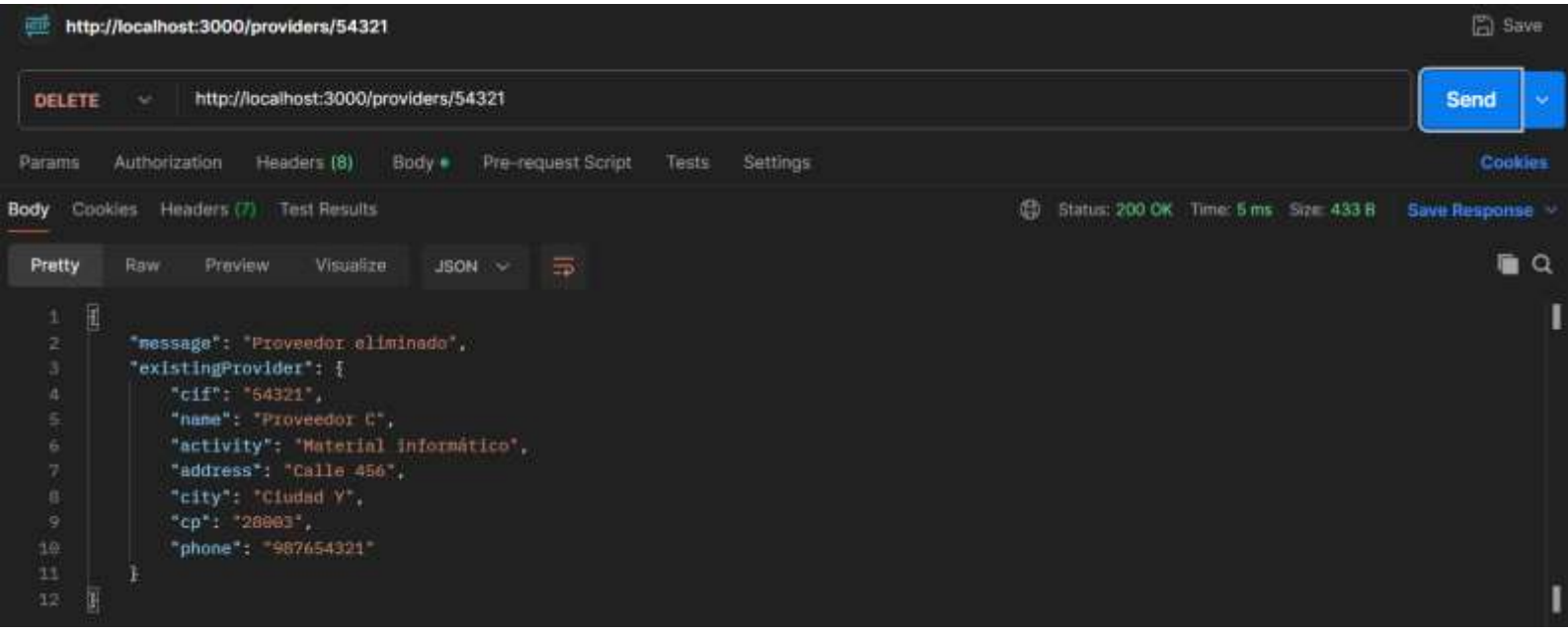


2.9.5. Petición “delete”

Por último, comprobamos cómo funciona nuestra petición “delete”, empezando por la validación que muestra el mensaje de error cuando el “cif” introducido en la **URL** no corresponde al de ningún proveedor de la lista.



Y después, el correcto funcionamiento de la petición, que elimina el proveedor cuyo “cif” introducimos en la **URL**.



3. Integración de proyecto Angular a la API Rest

Es el momento, tal y como indica el enunciado, de desarrollar el segundo elemento de la arquitectura *frontend*, el proyecto **Angular**, con el objetivo de poder realizar las operaciones **CRUD** con la **API Rest NodeJS**.

3.1. Implementación de CORS en archivo principal

Antes de ponernos con ello, debemos tener en cuenta que Angular ejecutará el proyecto *frontend* en un dominio diferente a la **API Rest** del *backend*, y para que se integren correctamente, es necesario implementar el mecanismo de seguridad **CORS** en nuestro archivo principal. Sin esto, nuestro navegador no permitirá la comunicación entre ellos.

```
const express = require('express');
const app = express();
const port = 3000;
const cors = require('cors');

app.use(express.json());
app.use(cors());

const providersRouter = require('./routes/providers');
app.use('/providers', providersRouter);

app.listen(port, () => {
    console.log(`Servidor escuchando en http://localhost:${port}`);
});
```

3.2. Creación de proyecto Angular

A continuación, podemos comenzar con la creación del proyecto **Angular**, introduciendo en la terminal de la raíz de nuestro proyecto el comando `ng new` seguido del nombre que le queremos dar a nuestro proyecto **Angular**, que en este caso será “providers-app”. Durante la instalación, cuando nos pregunte, le indicamos que agregue el enrutador y que use los estilos en **CSS**.

3.3. Creación y desarrollo de servicio para las peticiones CRUD

Creamos entonces un servicio con el que gestionaremos los métodos **CRUD** a la **API**, escribiendo en la terminal el comando `ng generate service services/providers`. Esto nos crea el servicio en la dirección `src/app/services`. Si abrimos el archivo, vemos que por defecto contiene ya una importación del inyectable, para poder reutilizar este servicio en otras partes, y su decorador. Le añadimos las siguientes líneas:

1. Creamos una interfaz `Provider` para estructurar los datos del proveedor, para garantizar así que cualquier objeto tenga disponible esas propiedades con sus respectivos tipos. Exportamos esta interfaz para que pueda ser accedida desde distintos puntos de nuestra aplicación.
2. Vamos con la exportación de nuestra clase – servicio `ProvidersService`:
    a. Primero, debemos indicarle el `endpoint`, es decir, la **URL** base de la **API** donde se realizarán las solicitudes **HTTP**. Lo declararemos de forma privada, ya que sólo lo usaremos dentro del servicio.
    b. Para realizar las solicitudes **HTTP** a la **API**, inyectamos el servicio proporcionado por **Angular** `HttpClient` en el constructor, que sirve expresamente para gestionar las operaciones **HTTP** de una manera más eficaz. Lo tendremos que importar en nuestro archivo.

Es el momento de crear los métodos de las peticiones. Para proporcionar flexibilidad y eficiencia a nuestra aplicación, todos los métodos devolverán un `Observable`, generalmente de tipo `Provider` (menos en el caso de “delete”), el cual tendremos que importar desde la librería **rxjs**. Esto, nos permite gestionar los datos de forma asíncrona, para que el programa pueda seguir ejecutándose mientras espera la respuesta de la **API**. En todos los casos le indicaremos entre los símbolos `< >` el tipo de dato que esperamos.

    c. Para la petición “get” que devuelve el array completo, creamos un método indicándole con los corchetes `[ ]`, que el `Observable` que devuelve será un array. Dentro, realizamos una solicitud “get” usando una instancia de `HttpClient`, la cual recibe el `endpoint` como parámetro. A continuación, aplicamos el método `pipe()` para encadenar un operador `map()` de **rxjs** (que debemos importar en el archivo), el cual itera sobre la respuesta recibida y transforma los datos, extrayendo y devolviendo únicamente el array de proveedores.
    d. Para la solicitud que devuelve un solo proveedor por su “cif”, creamos un método que recibe como parámetro un `cif`. Usamos el servicio `HttpClient` para realizar una solicitud "get" a la **API**. En este caso, utilizamos los símbolos `` ` `` y `$` para concatenar el `cif` al `endpoint`, y así formar la **URL** de la solicitud.
    e. Para agregar un nuevo proveedor a la lista, creamos un método que recibe como parámetro un objeto `provider` de tipo `Provider`. Usamos `HttpClient` para realizar una solicitud "post", pasando el `endpoint` y el objeto `provider` como cuerpo de la solicitud, lo que envía el objeto al servidor.
    f. Creamos un nuevo método para actualizar un proveedor existente, el cual recibe dos parámetros: el `cif` del proveedor a actualizar, y el objeto `provider` con los nuevos datos. Usamos de nuevo `HttpClient` para realizar una solicitud "put" a la **API**, pasando el cif del proveedor en la URL y el objeto provider como cuerpo de la solicitud. Para ello usamos de nuevo la concatenación que hemos usado en el segundo método “get”.
    g. Para eliminar un proveedor, creamos otro método, esta vez de tipo `void` ya que no devuelve nada, y que recibe como parámetro el `cif` del proveedor a eliminar. Realizamos una solicitud "delete" con `HttpClient`, con el cif del proveedor incluido en la URL. La URL se forma igual que en el caso anterior, concatenando el `endpoint` con el signo `/` y el `cif`.

```typescript
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';

export interface  Provider {
  cif: string;
  name: string;
  activity: string;
  address: string;
  city: string;
  cp: string;
  phone: string;
}
```

1

```typescript
@Injectable({
  providedIn: 'root'
})
```

2

```typescript
export class ProvidersService {

  private endpoint: string = 'http://localhost:3000/providers';

  constructor(private http: HttpClient) { }

  getProviders(): Observable<Provider[]> {
    return this.http.get(this.endpoint)
      .pipe(map((data: any) => data.providers));
  }

  getProviderByCif(cif: string): Observable<Provider> {
    return this.http.get<Provider>(`${this.endpoint}/${cif}`);
  }

  addProvider(provider: Provider): Observable<Provider> {
    return this.http.post<Provider>(this.endpoint, provider);
  }

  updateProvider(cif: string, provider: Provider): Observable<Provider> {
    return this.http.put<Provider>(`${this.endpoint}/${cif}`, provider);
  }

  deleteProvider(cif: string): Observable<Provider> {
    return this.http.delete<Provider>(`${this.endpoint}/${cif}`);
  }
}
```

a

b

c

d

e

f

g

3.4. Creación del primer componente: tabla de proveedores del array

Es el momento de ponernos con los componentes de nuestra app. Empecemos por el de la tabla que muestra los proveedores existentes en el array. Comenzamos tecleando en nuestra terminal el comando para crearlo `ng generate component`, seguido de la carpeta que queremos que cree para almacenarlos `components`, y del nombre del componente `/providers-table`.

Para la lógica en el archivo **TypeScript**, solamente vamos a necesitar que indique qué elementos mostrar en cada iteración de la directiva *ngFor, y la que indica qué elemento borrar con el botón "delete" de cada proveedor, puesto que los otros dos botones simplemente conducirán a la ruta de otros componentes, que llevarán su propia lógica.

Por defecto ya viene escrita la importación de la dependencia `Component`, con su decorador, que declara el selector a usar para integrar el código **HTML** en el archivo padre, con la ruta del archivo **HTML** y su correspondiente archivo de estilos **CSS**; y la exportación de la clase donde desarrollaremos la lógica del componente. Añadimos los siguientes elementos:

1. Implementamos `OnInit` en la declaración de la clase, para que los datos de los proveedores estén cargados desde el inicio del ciclo de vida del componente, y asegurar que la tabla se muestre correctamente. Para poder usar esta interfaz debemos importarla en nuestro archivo.
2. Declaramos un array `providers` vacío de tipo de la interfaz `Provider` (tenemos que importarla desde el servicio) donde iremos almacenando cada proveedor que vaya cargando desde la **API**.
3. Inyectamos el servicio `ProvidersService` en el constructor como una propiedad privada, lo que hace que el servicio esté disponible en toda la clase para realizar las operaciones necesarias con la **API**.
4. Declaramos el método `ngOnInit`, vinculado a la interfaz `OnInit`, que será el método que se ejecute automáticamente cuando se inicializa el componente. Usamos el método `getProviders()` del servicio para obtener la lista de proveedores desde la **API**. Como este método devuelve un `Observable`, nos suscribimos a él utilizando `.subscribe()`. Dentro del bloque de suscripción, recibimos los datos como un array de `Provider`. Asignamos estos datos a la propiedad providers, lo que permite renderizar la tabla de proveedores en el archivo **HTML** asociado al componente.
5. Para darle funcionalidad al botón de "delete" de cada fila de proveedor que se muestre en la tabla, creamos un método que recibe como argumento el "cif" del proveedor que queremos eliminar. Desde ahí, llamamos al método `deleteProvider()` del servicio para realizar la eliminación en la **API**. Nos suscribimos a la respuesta del método, que indica cuándo la operación de eliminación ha sido exitosa. Para reflejar el cambio en la tabla, actualizamos la lista local `providers`, filtrando con el método `filter()` para excluir del array cualquier proveedor cuyo "cif" coincida con el que acabamos de eliminar.

```typescript
import { Component, OnInit } from '@angular/core';
import { Provider, ProvidersService } from 'src/app/services/providers.service';

@Component({
  selector: 'app-providers-table',
  templateUrl: './providers-table.component.html',
  styleUrls: ['./providers-table.component.css']
})
export class ProvidersTableComponent implements OnInit {
  providers: Provider[] = [];

  constructor(private providersService: ProvidersService) { }

  ngOnInit(): void {
    this.providersService.getProviders().subscribe((data: Provider[]) => {
      this.providers = data;
    });
  }

  deleteProvider(cif: string): void {
    this.providersService.deleteProvider(cif).subscribe(() => {
      this.providers = this.providers.filter(provider => provider.cif !== cif);
    });
  }
}
```

Ahora desarrollemos el código del archivo HTML, que contendrá las siguientes líneas:

- Un elemento `<h2>` donde mostraremos el título de la página.
- Un elemento `<table>` para mostrar la tabla.
  - Un elemento `<thead>` que contenga la parte de arriba que describe lo que es cada columna.
    - Un elemento `<tr>` para crear una fila
      - 7 elementos `<th>` para mostrar una columna para cada una de las propiedades de nuestros proveedores,
      - Una columna `<th>` adicional para los botones de “editar” y de “eliminar” cada proveedor.
  - Un elemento `<tbody>` que contendrá el cuerpo de la tabla, donde se mostrarán las filas dinámicas con los datos de los proveedores.
    - Un elemento `<tr>` con la directiva estructural `*ngFor`, que iterará sobre el array de proveedores y generará una fila por cada elemento en la lista.
      - 7 columnas `<td>` para mostrar los valores de cada propiedad del proveedor. Usamos interpolación `{{ }}` para renderizar los valores.
      - Un último elemento `<td>` para las acciones, con un atributo `id` para manejarlo en el archivo de estilos, y donde incluimos dos botones:
        - Para poder editar cada proveedor, creamos un botón con un `id` para manejarlo en el **CSS**, y que utiliza la directiva `routerLink` (aplicando **binding** con los corchetes `[ ]` para asignar un valor

dinámico ya que depende de la fila en la que estemos) para redirigir al formulario de edición, pasando como parámetros la **URL** a la que acudir con el cif del proveedor actual.

- Para eliminar un proveedor de la lista, creamos un botón con un nuevo `id` para manejarlo en el **CSS**, y que al hacer "clic" en él, invoca directamente el método `deleteProvider`, enviando como parámetro el "cif" del proveedor de esa misma fila.

- Fuera de la tabla, un botón adicional para agregar un nuevo proveedor al array, con un atributo `id` propio para mejor manejo en los estilos, y la directiva `routerLink` que redirige a la ruta que pertenecerá al formulario de creación de un nuevo proveedor, componente que desarrollaremos a continuación.

Con esta estructura, la tabla es completamente dinámica, mostrando automáticamente los datos recibidos del componente y proporcionando interactividad mediante los botones de edición y eliminación.

```
<h2>Lista de Proveedores</h2>
<table>
    <thead>
        <tr>
            <th>CIF</th>
            <th>Nombre</th>
            <th>Actividad</th>
            <th>Dirección</th>
            <th>Localidad</th>
            <th>Código Postal</th>
            <th>Teléfono</th>
            <th>Acciones</th>
        </tr>
    </thead>
    <tbody>
        <tr *ngFor="let provider of providers">
            <td>{{ provider.cif }}</td>
            <td>{{ provider.name }}</td>
            <td>{{ provider.activity }}</td>
            <td>{{ provider.address }}</td>
            <td>{{ provider.city }}</td>
            <td>{{ provider.cp }}</td>
            <td>{{ provider.phone }}</td>
            <td id="actionButtons">
                <button id="editButton" [routerLink]="['/edit-provider', provider.cif]">Editar</button>
                <button id="deleteButton" (click)="deleteProvider(provider.cif)">Eliminar</button>
            </td>
        </tr>
    </tbody>
</table>
<div>
    <button id="addButton" routerLink="/add-provider">Agregar nuevo proveedor</button>
</div>
```

3.5. Segundo componente: formulario para añadir un proveedor

Vamos con la creación del segundo componente, el que tiene la función de mostrar un formulario por el cual agregar un nuevo proveedor al array, vinculado a la petición “post”.

En primer lugar, introducimos en la terminal integrada de nuestra carpeta `components` el comando `ng generate component add-provider`. Esto nos crea automáticamente la estructura de archivos para un componente **Angular**.

Usaremos el módulo de formularios reactivo de **Angular**, `ReactiveFormsModule`, el cual lo importaremos más adelante en el archivo de módulos de nuestra aplicación Angular, junto con los componentes, cuando terminemos de desarrollarlos.

Vamos con los códigos del componente. Empecemos por desarrollar la lógica en el archivo **TypeScript**. Al código que por defecto traen los componentes en **Angular**, le añadimos lo siguiente dentro de la declaración de la clase:

- Para usar de forma estructurada el conjunto de campos del formulario, es conveniente usar la clase `FormGroup`, así que declaramos una propiedad `providerForm` de este tipo. Para usar esta clase tenemos que importarla.
- Para poder mostrar el mensaje de error cuando se envíe el formulario y el campo “cif” esté vacío, necesitamos crear una bandera booleana, que llamaremos `formSubmitted`, inicializada en `false`.
- Para poder usar los métodos **CRUD** que hemos desarrollado en el servicio, lo inyectamos como parámetro en nuestro método constructor.
- También inyectamos el servicio `Router` para poder redirigir al usuario a otra página cuando lo necesitemos.
- Dentro del método constructor, inicializamos un `FormGroup` para nuestro formulario. Le añadimos a cada una de las propiedades un `FormControl` para enlazarlas a los campos del archivo **HTML**, y las inicializamos vacías. Para el caso de la propiedad “cif”, además, tendremos que cerciorarnos de que no esté vacía, por lo que usamos la validación de **Angular** `Validators.required`. Para usarlos, tendremos que importar a nuestro archivo tanto `FormControl` como `Validators`.
- Para configurar el envío del formulario, creamos un metodo `addProvider()`, que irá enlazado al botón de “enviar”. Dentro de éste, en primer lugar, cambiamos la bandera booleana de `formSubmitted` a true. A continuación, usando la propiedad de control de formularios **Angular** `invalid`, le indicamos que, si se cumple ésta, termine la tarea sin almacenar el proveedor. Si no cae en esa validación, almacenamos los datos usando el método `addProvider()` del servicio, suscribiéndonos a él, y pasándole como parámetro el `value` del formulario en conjunto. Además, cuando pulsemos el botón, redirigimos al componente tabla usando el servicio `router` y su método `navigate()`.

```typescript
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';
import { ProvidersService } from 'src/app/services/providers.service';
import { Router } from '@angular/router';

@Component({
  selector: 'app-add-providers',
  templateUrl: './add-provider.component.html',
  styleUrls: ['./add-provider.component.css']
})
export class AddProviderComponent {
  providerForm: FormGroup;
  formSubmitted = false;

  constructor(
    private providersService: ProvidersService,
    private router: Router
  ) {
    this.providerForm = new FormGroup({
      cif: new FormControl('', Validators.required),
      name: new FormControl(''),
      activity: new FormControl(''),
      address: new FormControl(''),
      city: new FormControl(''),
      cp: new FormControl(''),
      phone: new FormControl('')
    });
  }

  addProvider(): void {
    this.formSubmitted = true;

    if (this.providerForm.invalid) {
      return;
    }

    this.providersService.addProvider(this.providerForm.value).subscribe(() => {
      this.router.navigate(['/providers']);
    });
  }
}
```

Ahora es el momento de trabajar en el archivo **HTML**, así que escribimos los siguientes elementos:

- Titulamos nuestra página con un elemento `<h2>`.
- Para el formulario, usamos un elemento `<form>`, el cual usa el atributo `[formGroup]` para vincularse al objeto `providerForm` del archivo **TypeScript**, y el evento `(ngSubmit)` para asociar el envío del formulario al método `addProvider()`, encargado de manejar la lógica de creación del proveedor.
    - Para cada propiedad de nuestros proveedores, tendremos que crear una etiqueta `<label>`, donde describir el título de cada una.

- Y debajo, sus correspondientes campos `<input>`, cada uno vinculado a su propiedad del formulario mediante la directiva `formControlName`.
- Además, para el caso de la propiedad "cif", incluiremos un elemento `<small>` para crear un mensaje de error con la interpolación dinámica, que se muestre cuando el formulario se intenta enviar, y que impida hacerlo. Esto lo conseguimos con la directiva de **Angular** `*ngIf`, la cual evalúa si el formulario ha sido enviado con la propiedad `formSubmitted`, y si el campo "cif" es válido, usando el método `get()` de la directiva `FormGroup` para indicarle el campo, el operador `?` para comprobar si es `null` o `undefined`, y la propiedad `invalid` que hemos desarrollado en el archivo **TypeScript**. Por último, le damos un atributo `class` para poder darle unos estilos personalizados.
- Un elemento `<div>` que contiene dos botones para las acciones principales del formulario:
  - Un botón de tipo `submit` que envía los datos del formulario y que está vinculado al método `addProvider()` con la directiva `(ngSubmit)`.
  - Un botón adicional con la directiva `routerLink` (sin binding, ya que la ruta es estática), que redirige al usuario a la tabla de proveedores para volver sin realizar ninguna acción.

De esta forma, el formulario es funcional y dinámico, y se valida en tiempo real gracias a **Angular Forms**.

```
<h2>Agregar Nuevo Proveedor</h2>

<form [formGroup]="providerForm" (ngSubmit)="addProvider()">
    <label for="cif">CIF:</label>
    <input id="cif" formControlName="cif" type="text"/>
    <small *ngIf="formSubmitted && providerForm.get('cif')?.invalid" class="error">El CIF es obligatorio.</small>

    <label for="name">Nombre:</label>
    <input id="name" formControlName="name" type="text"/>

    <label for="activity">Actividad:</label>
    <input id="activity" formControlName="activity" type="text"/>

    <label for="address">Dirección:</label>
    <input id="address" formControlName="address" type="text"/>

    <label for="city">Localidad:</label>
    <input id="city" formControlName="city" type="text"/>

    <label for="cp">Código Postal:</label>
    <input id="cp" formControlName="cp" type="text"/>

    <label for="phone">Teléfono:</label>
    <input id="phone" formControlName="phone" type="text"/>

    <div class="formButtons">
        <button type="submit">Agregar</button>
        <button routerLink="/providers">Volver</button>
    </div>
</form>
```

3.6. Tercer componente: formulario para modificar un proveedor existente

El último componente que nos pide el ejercicio es un nuevo formulario a través del cual modificar uno de los proveedores existentes en el array. En términos generales, el código se parecerá bastante al del formulario para añadir un nuevo proveedor, aunque con unas diferencias clave. En primer lugar, como es natural, este componente usará métodos distintos del servicio. En segundo lugar, desde su inicialización, deberá cargar datos desde la **API**, obteniendo desde la **URL** el "cif" sobre el que debe actuar, y los valores actuales del proveedor, para mostrarlos en las etiquetas y facilitar así la tarea de edición al usuario.

El código **TypeScript**, por lo tanto, tendrá lo siguiente:

- Como habrá datos que necesitaremos cargar desde la inicialización del compenente, implementamos OnInit en la clase principal del componente.
- Al igual que en el otro formulario, creamos una instancia de `FormGroup`, una variable vacía donde almacenar el "cif" de cada proveedor, y una bandera booleana inicializada en false para determinar si se ha enviado el formulario.
- También como en el otro componente, declaramos un constructor donde inyectamos tanto nuestro servicio para obtener los métodos CRUD, como el servicio router de Angular para poder navegar a otras rutas.
- Para poder saber qué proveedor se está pretendiendo modificar, le diremos que obtenga el "cif" que lo identifica desde la **URL**. Para esto, importaremos la clase `ActivatedRoute` de **Angular** que proporciona datos de la ruta activa, y también dentro del método constructor, inyectamos una instancia privada de ésta, llamada `currentRoute`.
- Igual que en el otro componente de formulario, dentro del método constructor, inicializamos un `FormGroup` con cada una de las propiedades asociadas a un `FormControl`, y con la validación requerida para la propiedad "cif".
- Declaramos el método `ngOnInit`, donde, como ya sabemos, irá la lógica que nuestro componente deberá adoptar cada vez que sea inicializado, que será así:
    - Para obtener el "cif" de la **URL**, usamos `currentRoute` para acceder a `ActivatedRoute`, desde la cual nos servimos de su objeto `snapshot`, el cual captura una instantánea de la **URL** actual, y de su propiedad `paramMap` que permite acceder a los valores del parámetro de la ruta.
    - Para obtener los datos del proveedor que estamos modificando y mostrarlos dentro de cada campo, utilizamos el método `getProviderByCif` de nuestro servicio para suscribirnos a la respuesta de la solicitud **HTTP**, y almacenarla así en un parámetro data. Si el "cif" existe, podemos acceder a sus datos utilizando la propiedad `existingProvider`, que hemos definido en el archivo de la **API** original. Luego, los mostramos en el formulario utilizando el método `patchValue()` perteneciente a `FormGroup`, accediedo a él a través de su instancia `providerForm`.
- Creamos un método para manejar el envío del formulario, el cual cambia la bandera booleana a `true`, se frena si el formulario es inválido, y si no, llama al método `updateProvider` de nuestro servicio pasándole el "cif" y los valores del formulario, y por último, navega hasta el componente de la tabla para mostrar los resultados.

```typescript
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';
import { ProvidersService } from 'src/app/services/providers.service';
import { Router, ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-edit-provider',
  templateUrl: './edit-provider.component.html',
  styleUrls: ['./edit-provider.component.css']
})
export class EditProviderComponent implements OnInit {
  providerForm: FormGroup;
  cif: string = '';
  formSubmitted = false;

  constructor(
    private providersService: ProvidersService,
    private router: Router,
    private currentRoute: ActivatedRoute
  ) {
    this.providerForm = new FormGroup({
      cif: new FormControl('', Validators.required),
      name: new FormControl(''),
      activity: new FormControl(''),
      address: new FormControl(''),
      city: new FormControl(''),
      cp: new FormControl(''),
      phone: new FormControl('')
    });
  }

  ngOnInit(): void {
    this.cif = this.currentRoute.snapshot.paramMap.get('cif') || '';
    this.providersService.getProviderByCif(this.cif).subscribe((data: any) => {
      this.providerForm.patchValue(data.existingProvider);
    });
  }

  sendForm(): void {
    this.formSubmitted = true;
    if (this.providerForm.invalid) { return }
    this.providersService.updateProvider(this.cif, this.providerForm.value).subscribe(() => {
      this.router.navigate(['/providers']);
    });
  }
}
```

Vamos ahora con el código del archivo **HTML**. Será idéntico al del componente para agregar un nuevo proveedor, cambiando solamente el título, el nombre del método al que llama (aunque también este haga lo mismo), y el nombre del botón para enviar el formulario.

De hecho, creo que en este caso lo ideal sería haber usado un solo componente para la función de actualizar y la de editar, pero como el ejercicio textualmente pedía un componente para cada función, he preferido ceñirme al enunciado.

```html
<h2>Agregar Nuevo Proveedor</h2>

<form [formGroup]="providerForm" (ngSubmit)="addProvider()">
    <label for="cif">CIF:</label>
    <input id="cif" formControlName="cif" type="text"/>
    <small *ngIf="formSubmitted && providerForm.get('cif')?.invalid" class="error">El CIF es obligatorio.</small>

    <label for="name">Nombre:</label>
    <input id="name" formControlName="name" type="text"/>

    <label for="activity">Actividad:</label>
    <input id="activity" formControlName="activity" type="text"/>

    <label for="address">Dirección:</label>
    <input id="address" formControlName="address" type="text"/>

    <label for="city">Localidad:</label>
    <input id="city" formControlName="city" type="text"/>

    <label for="cp">Código Postal:</label>
    <input id="cp" formControlName="cp" type="text"/>

    <label for="phone">Teléfono:</label>
    <input id="phone" formControlName="phone" type="text"/>

    <div class="formButtons">
        <button type="submit">Agregar</button>
        <button routerLink="/providers">Volver</button>
    </div>
</form>
```

3.7. Importación de componentes y librerías en archivo de módulos

Desarrollados los tres componentes, tenemos que importarlos en el archivo de módulos, junto a las librerías que hemos utilizado (`HttpClientModule` y `ReactiveFormsModule`). Añadimos también los componentes en la propiedad `declarations`, y las librerías en la de `imports`.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

import { HttpClientModule } from '@angular/common/http';
import { ReactiveFormsModule } from '@angular/forms';

import { ProvidersTableComponent } from './components/providers-table/providers-table.component';
import { AddProviderComponent } from './components/add-provider/add-provider.component';
import { EditProviderComponent } from './components/edit-provider/edit-provider.component';

@NgModule({
  declarations: [
    AppComponent,
    ProvidersTableComponent,
    AddProviderComponent,
    EditProviderComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule,
    ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

3.8. Desarrollo de estilos CSS

Para finalizar nuestra aplicación, y con el objetivo de mejorar la experiencia visual, otorgaremos unos estilos **CSS** en cada uno de los componentes, así como en el archivo global.

Empezaremos por el archivo de estilos principal, que afecta de forma global a toda la aplicación de **Angular**. En él, pondremos los estilos comunes de los elementos que afectan al programa completo.

```css
@import url('https://fonts.googleapis.com/css?family=Lato&display=swap');

body {
    font-family: 'Lato', sans-serif;
    background-color: #1b1b1b;
    color: #e0e0e0;
    margin: 20px;
}

h2 {
    margin-top: 15px;
    font-size: 33px;
    color: #e0e0e0;
    text-align: center;
}

hr {
    margin: 70px;
}

button {
    margin-top: 15px;
    padding: 14px 20px;
    background-color: #250dad;
    color: #fff;
    font-size: large;
    border: none;
    border-radius: 7px;
    cursor: pointer;
    transition: background-color 400ms ease-in-out;
}

button:hover {
    background-color: #624ae8;
}
```

A continación, vamos con el archivo de estilos del componente de la tabla, con sus respectivos elementos.

```css
table {
    width: 100%;
    margin-top: 20px;
    margin-bottom: 24px;
}

th {
    padding: 10px 20px;
    background-color: #250dad;
    color: #e0e0e0;
    text-align: center;
    text-decoration: solid;
}

td {
    padding: 0px 20px;
    border: 1px solid #444;
    text-align: left;
}
```

```css
tr:nth-child(even) {
    background-color: #2b2b2b;
}

#actionButtons {
    display: flex;
    justify-content: space-around;
}

#editButton,
#deleteButton {
    margin: 3px 3px;
    padding: 4px 8px;
    font-size: medium;
}
```

Por último, vamos con los estilos de los componentes para agregar y editar un proveedor, los cuales al ser exactamente iguales los pondré una sola vez.

```css
form {
    display: flex;
    flex-direction: column;
    width: 100%;
    max-width: 500px;
    margin: 0 auto;
}

form label {
    margin-top: 15px;
    margin-bottom: 5px;
    font-weight: bold;
    font-size: 14px;
}

form input {
    padding: 8px;
    border: 1px solid #444;
    background-color: #333;
    color: #e0e0e0;
    width: 100%;
}
```

```css
.formButtons {
    display: flex;
    justify-content: flex-end;
    gap: 14px;
}

.error {
    margin-top: 3px;
    color: red;
    font-size: 12px;
}
```