# 11
# Trabajando con servicios

# 11. Trabajando con servicios

## Evolución de la programación asíncrona con JavaScript

# 11. Trabajando con servicios

## Patrón Observable

Las comunicaciones entre navegadores y servidores son varios órdenes de magnitud más lentas que las operaciones en memoria. Por tanto deben realizarse de manera asíncrona para garantizar una buena experiencia al usuario

Esta experiencia no siempre fue tan buena para el programador. Sobre todo con las primeras comunicaciones *AJAX* basadas en el paso de funciones *callback*. La aparición de las *promises* mejoró la claridad del código, y ahora con los *Observables* tenemos además una gran potencia para manipular la **información asíncrona**.

**“El patrón Observable fue implementado por Microsoft en la librería Reactive Extensions más conocida como RxJs. El equipo de Angular decidió utilizarla para el desarrollo de las comunicaciones asíncronas. Esta extensa librería puede resultar intimidante en un primer vistazo. Pero con muy poco conocimiento puedes programar casi todas las funcionalidades que se te ocurran.”**

# 11. Trabajando con servicios

## Patrón Observable

El patrón observable no es más que un modo de implementación de la programación reactiva, que básicamente pone en funcionamiento diversos actores para producir los efectos deseados, que es reaccionar ante el flujo de los distintos eventos producidos. Mejor dicho, producir dichos eventos y consumirlos de diversos modos.
Los componentes principales de este patrón son:

•**Observable**: Es aquello que queremos observar, que será implementado mediante una colección de eventos o valores futuros. Un observable puede ser creado a partir de eventos de usuario derivados del uso de un formulario, una llamada HTTP, un almacén de datos, etc. Mediante el observable nos podemos suscribir a eventos que nos permiten hacer cosas cuando cambia lo que se esté observando.

•**Observer**: Es el actor que se dedica a observar. Básicamente se implementa mediante una colección de funciones callback que nos permiten escuchar los eventos o valores emitidos por un observable. Las callbacks permitirán especificar código a ejecutar frente a un dato en el flujo, un error o el final del flujo.

•**Subject**: es el emisor de eventos, que es capaz de crear el flujo de eventos cuando el observable sufre cambios. Esos eventos serán los que se consuman en los observers.
Estas son las bases del patrón. Sabemos que hemos puesto varios conceptos que sólo quedarán más claros cuando los veamos en código. Será dentro de poco.

# 11. Trabajando con servicios

## HTTP

Las comunicaciones *http* son una pieza fundamental del desarrollo web, y en **Angular** siempre han sido fáciles y potentes. En La versión actual **consumir un servicio REST** es una cosa muy sensilla

Claro que para ello tendremos que jugar con los **observables** y los servicios de la librería *@angular/common/http* con los que realizar comunicaciones asíncronas en Angular

Con HttpClient, @angular/common/http proporciona una API simplificada para funcionalidad HTTP para usar con aplicaciones Angular, construyendo encima de interfaz XMLHttpRequest expuesta por navegadores.

Los beneficios adicionales de HttpClient incluyen

- soporte para pruebas
- tipado fuerte de objetos de petición y respuesta
- mejor manejo de errores vía apis basadas en Observables.

# 11. Trabajando con servicios

## Inyección de dependencias

La **Inyección de Dependencias (DI)** es un **mecanismo para proporcionar nuevas instancias de una clase** con todas aquellas dependencias que requiere, **completamente inicializadas**.

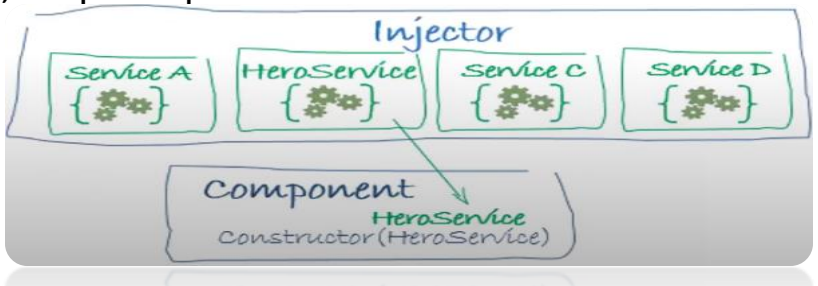
La mayoría de dependencias son servicios y **Angular** usa la **DI** para proporcionar nuevos componentes a los servicios que los necesitan.

Como hemos adelantado al hablar sobre servicios, gracias a **TypeScript**, **Angular** es capaz de conocer de qué servicios depende un componente solamente con mirar los parámetros de su constructor.

El **Inyector/Injector** es el principal mecanismo tras la **DI**.

A nivel interno un **inyector** dispone de un contenedor con las instancias de servicios que crea él mismo. Si una instancia no está en el contenedor, el **inyector** crea una nueva a través de su **Provider** y la añade al contenedor antes de devolver el servicio a **Angular**. Los servicios son **singletons**.

Cuando todos los servicios de los que depende el contenedor se han resuelto, **Angular** puede llamar al constructor del componente, al que le pasa las instancias de esos servicios como argumento.



# 11. Trabajando con servicios

## Provider

El **provider** es cualquier cosa que pueda crear o devolver un servicio. A diferencia de **AngularJS**, donde existía una sintaxis específica, en **Angular** el **provider** suele ser la propia clase.

En **Angular** **es la propia clase la que define el servicio**.

Los **providers** pueden registrarse en cualquier nivel del árbol de componentes de la aplicación a través de los metadatos de componentes, o a nivel raíz, en el **NgModule** de la aplicación.

Al registrar un **provider** en el **NgModule** éste estará disponible para toda la aplicación.

Si el servicio que quieres declarar sólo afecta a una pequeña parte de tu app, como puede ser un componente o un componente y sus hijos, tiene más sentido que se declare a nivel de componente.

Cuando registramos un **provider** **a nivel de componente**, **obtendremos una nueva instancia del servicio por cada nueva instancia del componente**.

# 11. Trabajando con servicios

## Servicios

Los **servicios** son una pieza fundamental en **Angular**, se definen a partir de simples clases.

Todo valor, función o característica que nuestra aplicación necesita, desde constantes a lógica de negocio, se encapsula dentro de un **servicio**.

Los **Componentes** **son grandes consumidores de servicios**. No recuperan datos del servidor, ni validan inputs de usuario, ni loguean nada directamente. **Delegan todo este tipo de tareas a los servicios**.

Los **servicios** deben contener/realizar algo muy específico. Por ejemplo serían susceptibles de encapsular un **servicios**:

- Servicio de logging.
- Servicio de datos.
- Bus de mensajes.
- Cálculo de impuestos.
- Configuración de una app.

```
//app/shared/logger.service.ts
export class Logger {
  log(msg: any)   { console.log(msg); }
  error(msg: any) { console.error(msg); }
  warn(msg: any)  { console.warn(msg); }
}
```

Ejemplo de servicio simple de Loggin

# 11. Trabajando con servicios

## Estructura de un servicio en Angular

Decorador Injectable

```
//app/todos/shared/todo.service.ts
import {Injectable} from '@angular/core';
import { Todo } from './todo.model';
import { Logger } from '../../shared/logger.service';

@Injectable()
export class TodoService{

    todos:Todo[] = [];

    constructor(public logger: Logger){}

    addTodo(todo:Todo){
        this.todos = [...this.todos, todo];
        this.logger.log(this.todos);
    }

    getTodos(){
        this.logger.log(this.todos);
        return this.todos;
    }

}
```

# 11. Trabajando con servicios

## Injectable

•**@Injectable:** Este decorador **avisa a Angular de que el servicio espera utilizar otros servicios** y genera los metadatos que necesita el servicio para detectar la Inyección de Dependencias (*DI*) en el constructor. No es necesario ponerlo si nuestro servicio no tiene *DI* de otros servicios, pero es recomendable para evitar errores si en el futuro queremos añadirle alguna dependencia.

•***Dependency Injection* (*DI*) en el Constructor:** Aquí aprovechamos las bondades de *TypeScript* para pasar explícitamente un objeto tipo *Logger* en el constructor de TodoService. De este modo, Angular es capaz de inferir la Inyección de Dependencias.

•Los *Guide Lines* de Angular recomiendan **utilizar siempre** el decorador **@Injectable** al definir nuestros servicios.

# 11. Trabajando con servicios

## Registrar un servicio en NgModule

Puede pasarse un **array** con los **providers** en los metadatos del módulo que contiene el **servicio** (normalmente el principal).

```
//src/app/app.module.ts

//...some imports ...
import { Logger } from '../shared/logger.service';


@NgModule({
    //..some stuff...
  providers: [ Logger ]
})

export class AppModule { }
```

A partir de ese momento cualquier otro **servicio** o **componente** de este módulo que lo reclame será provisto con una misma instancia de este **servicio**. Se crea un **singleton** por cada **módulo** en el que se provea un **servicio**. Normalmente si el **servicio** es para un sólo módulo funcional se provee en este únicamente. **Algunos servicios de uso común se proveen en el módulo raíz**, garantizando así su disponibilidad en toda la aplicación.

# 11. Trabajando con servicios

## Registrar el Providers en los componentes

También podemos usar el metadato **providers** del componente para indicar los **providers** que necesita el componente o cualquiera de sus subcomponentes.

```
@Component({
  //...some metadata params...
  providers:   [TodoService]
})
export class TodoListComponent {
  //TodoList component stuff
}
```

# PRACTICA SERVICIOS

## Creamos nuestra aplicación

En el siguiente ejercicio vamos a construir un sistema de alta de clientes y un listado de clientes que irá incrementando ítems, a medida que los demos de alta.

El primer paso es, usando el CLI, crear nuestra aplicación nueva. Lo haces entrando en la carpeta de tus proyectos y ejecutando el comando.

```
ng new clientes-app
```

Luego puedes entrar en la carpeta de la aplicación y lanzar el servidor web de desarrollo, para ver lo que se ha construido hasta el momento.

```
cd clientes app
ng serve -o
```

# PRACTICA SERVICIOS

Creamos nuestro módulo de clientes

La aplicación recién generada ya contiene un módulo principal, sin embargo, yo prefiero dejar ese módulo con pocas o ninguna cosa más de las que nos entregan por defecto al generar la aplicación básica. Por ello crearemos como primer paso un módulo nuevo, llamado "ClientesModule".

Encargamos a Angular CLI la creación del esqueleto de nuestro módulo con el siguiente comando:

```
ng generate module clientes
```

# PRACTICA SERVICIOS

## Definir el modelo de datos

Vamos a comenzar nuestro código por definir los tipos de datos que vamos a usar en esta aplicación. Vamos a trabajar con clientes y grupos.

Crearemos el modelo de datos dentro de la carpeta del módulo "clientes". No existe en Angular un generador de este tipo de modelos, por lo que crearé el archivo a mano con el editor. Lo voy a nombrar "cliente.model.ts".

En este archivo coloco las interfaces de TypeScript que definen los datos de mi aplicación.

# PRACTICA SERVICIOS

Definir el modelo de datos

```
export interface Cliente {
  id: number;
  nombre: string;
  cif: string;
  direccion: string;
  grupo: number;
}

export interface Grupo {
  id: number;
  nombre: string;
}
```

Como ves, he creado el tipo de datos Cliente y, por complicarlo un poquito más, el tipo de datos Grupo. Así, cada cliente generado pertenecerá a un grupo.

**Nota:** Esta parte de la creación de interfaces es perfectamente opcional. Solo la hacemos para usar esos tipos en la declaración de variables. El compilador de TypeScript nos avisará si en algún momento no respetamos estos tipos de datos, ayudando en tiempo de desarrollo y ahorrando algún que otro error derivado por despistes.

# PRACTICA SERVICIOS

Crear un servicio para los clientes

Lo ideal es crear un servicio (service de Angular) donde concentremos las tareas de trabajo con los datos de los clientes, descargando de código a los componentes de la aplicación y centralizando en un solo archivo la lógica de la aplicación.

El servicio lo vamos a crear dentro de la carpeta del módulo clientes, por lo que especificamos la ruta completa.

ng generate service clientes/clientes

En el servicio tengo que hacer el import del modelo de datos, interfaces de Cliente y Grupo (creadas en el paso anterior).

```
import { Cliente, Grupo } from './cliente.model';
```

# PRACTICA SERVICIOS

## Crear un servicio para los clientes

Nuestro servicio no tiene nada del otro mundo. Vamos a ver su código y luego explicaremos algún que otro punto destacable.

```
import { Injectable } from '@angular/core';
import { Cliente, Grupo } from './cliente.model';

@Injectable()
export class ClientesService {

  private clientes: Cliente[];
  private grupos: Grupo[];

  constructor() {
    this.grupos = [
      {
        id: 0,
        nombre: 'Sin definir'
      },
      {
        id: 1,
        nombre: 'Activos'
      },
      {
```

# PRACTICA SERVICIOS

Crear un servicio para los clientes

1.Las dos propiedades del servicio contienen los datos que va a mantener. Sin embargo, las hemos definido como privadas, de modo que no se puedan tocar directamente y tengamos que usar los métodos del servicio creados para su acceso.

2.Los grupos los construyes con un literal en el constructor. Generalmente los traerías de algún servicio REST o algo parecido, pero de momento está bien para empezar.

3.Agregar un cliente es un simple "push" al array de clientes, de un cliente recibido por parámetro.

4.Crear un nuevo cliente es simplemente devolver un nuevo objeto, que tiene que respetar la interfaz, ya que en la función nuevoCliente() se está especificando que el valor de devolución será un objeto del tipo Cliente.

5.Fíjate que en general está todo tipado, tarea opcional pero siempre útil.

# PRACTICA SERVICIOS

Declarar el servicio para poder usarlo en los componentes

Una tarea fundamental para poder usar los servicios es declararlos en el "module" donde se vayan a usar.

Para añadir el servicio en el module "clientes.module.ts", el primer paso es importarlo.

```
import { ClientesService } from './clientes.service';
```

Luego hay que declararlo en el array "providers".

```
providers: [
  ClientesService
]
```

# PRACTICA SERVICIOS

Crear componente que da de alta clientes

Vamos a continuar nuestra práctica creando un primer componente. Es el que se encargará de dar de alta los clientes.

Generamos el esqueleto usando el Angular CLI.

***ng generate component clientes/altaCliente***

Comenzaremos editando el archivo del componente y luego iremos a trabajar con el template. Por tanto, vamos a abrir el fichero "alta-cliente.component.ts".

# PRACTICA SERVICIOS

## Agregar el servicio al componente

Muy importante. Para poder usar el servicio anterior, tengo que agregarlo al componente recién creado, realizando el correspondiente import.

```
import { ClientesService } from './../clientes.service';
```

Y posteriormente ya podré inyectar el servicio en el constructor del componente.

```
constructor(private clientesService: ClientesService) { }
```

# PRACTICA SERVICIOS

Agregar el modelo de datos

Para poder seguir usando los tipos de datos de mi modelo, vamos a agregar el archivo donde se generaron las interfaces.

```
import { Cliente, Grupo } from './../cliente.model';
```

# PRACTICA SERVICIOS

## Código TypeScript completo del componente

El código completo de "alta-cliente.component.ts", para la definición de mi componente, quedaría más o menos así

```
import { Cliente, Grupo } from './../cliente.model';
import { ClientesService } from './../clientes.service';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-alta-cliente',
  templateUrl: './alta-cliente.component.html',
  styleUrls: ['./alta-cliente.component.css']
})
export class AltaClienteComponent implements OnInit {

  cliente: Cliente;
  grupos: Grupo[];

  constructor(private clientesService: ClientesService) { }

  ngOnInit() {
    this.cliente = this.clientesService.nuevoCliente();
    this.grupos = this.clientesService.getGrupos();
  }
```

# PRACTICA SERVICIOS

Código TypeScript completo del componente

Es importante mencionar estos puntos.

1.El componente declara un par de propiedades, el cliente y el array de grupos.

2.En el constructor, que se ejecuta lo primero, conseguimos una instancia del servicio de clientes, mediante la inyección de dependencias.

3.Posteriormente se ejecuta ngOnInit(). En este punto ya se ha recibido el servicio de clientes, por lo que lo puedo usar para generar los valores que necesito en las propiedades del componente.

4.El método nuevoCliente() es el que se ejecutará cuando, desde el formulario de alta, se produzca el envío de datos. En este código usamos el servicio clientesService, para agregar el cliente y generar un cliente nuevo, para que el usuario pueda seguir dando de alta clientes sin machacar los clientes anteriormente creados.

# PRACTICA SERVICIOS

Template del componente, con el formulario de alta de cliente

Vamos a ver ahora cuál es el HTML del componente de alta de clientes, que básicamente contiene un formulario.

Pero antes de ponernos con el HTML, vamos a hacer una importante tarea. Consiste en declarar en el módulo de clientes que se va a usar la directiva "ngModel". Para ello tenemos que hacer dos pasos:

En el archivo "clientes.module.ts" comenzamos por importar "FormsModule".

```
import { FormsModule } from '@angular/forms';
```

En el decorador, indicamos el imports del FormsModule.

```
imports: [
  CommonModule,
  FormsModule
],
```

# PRACTICA SERVICIOS

Template del componente, con el formulario de alta de cliente

Ahora veamos el código del template, en el que reconocerás el uso de la propiedad "cliente" declarada en el constructor, así como el array de grupos.

```
<h2>Alta cliente</h2>
<p>
  <span>Nombre:</span>
  <input type="text" [(ngModel)]="cliente.nombre">
</p>
<p>
  <span>CIF:</span>
  <input type="text" [(ngModel)]="cliente.cif">
</p>
<p>
  <span>Dirección:</span>
  <input type="text" [(ngModel)]="cliente.direccion">
</p>
<p>
  <span>Grupo:</span>
  <select [(ngModel)]="cliente.grupo">
    <option *ngFor="let grupo of grupos" value="{{grupo.id}}">{{grupo.nombre}}</option>
  </select>
</p>
<p>
```

# PRACTICA SERVICIOS

## Usar el componente Alta cliente

Este componente, de alta de clientes, lo quiero usar desde el componente raíz de mi aplicación. Como el componente raíz está declarado en otro módulo, necesito hacer que conozca al AltaClienteComponent. Esto lo consigo en dos pasos:

1.- Agregar al exports AltaClienteComponent

En el módulo de clientes "clientes.module.ts" agrego al exports el componente que quiero usar desde otros módulos.

```
exports: [
    AltaClienteComponent
  ]
```

# PRACTICA SERVICIOS

Usar el componente Alta cliente

2.- Importar en el módulo raíz

Ahora, en el módulo raíz, "app.module.ts", debes declarar que vas a usar componentes que vienen de clientes.module.ts. Para ello tienes que hacer el correspondiente import:

```
import { ClientesModule } from './clientes/clientes.module';
```

Y luego declaras el módulo en el array de imports:

```
imports: [
  BrowserModule,
  ClientesModule
],
```

# PRACTICA SERVICIOS

## Usar el componente Alta cliente

Hechos los dos pasos anteriores, ya puedes usar el componente en el template. Para ello simplemente tenemos que escribir su tag, en el archivo "app.component.html".

<app-alta-cliente></app-alta-cliente>

Llegado a este punto, si todo ha ido bien, deberías ver ya el componente de alta de clientes funcionando en tu página.

# PRACTICA SERVICIOS

## Crear el componente listado-cliente

Para acabar nuestra práctica vamos a crear un segundo componente de aplicación. Será el componente que nos muestre un listado de los clientes que se van generando.

Como siempre, comenzamos con un comando del CLI.

ng generate component clientes/listadoClientes

# PRACTICA SERVICIOS

Crear el componente listado-cliente

Ahora el flujo de trabajo es similar al realizado para el componente anterior. Vamos detallando por pasos.

Creas los import del servicio y de los tipos de datos del modelo.

***import { Cliente, Grupo } from './../cliente.model';***
***import { ClientesService } from './../clientes.service';***

Inyectas el servicio en el constructor.

***constructor(private clientesService: ClientesService) { }***

En este componente tendremos como propiedad el array de clientes que el servicio vaya creando. Así pues, declaras dicho array de clientes:

clientes: Cliente[];

# PRACTICA SERVICIOS

Crear el componente listado-cliente

Cuando se inicialice el componente tienes que solicitar los clientes al servicio. Esto lo hacemos en el método ngOnInit().

```
ngOnInit() {
  this.clientes = this.clientesService.getClientes();
}
```

# PRACTICA SERVICIOS

## Código completo del componente ListadoClientesComponent

```
import { Cliente, Grupo } from './../cliente.model';
import { ClientesService } from './../clientes.service';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-listado-clientes',
  templateUrl: './listado-clientes.component.html',
  styleUrls: ['./listado-clientes.component.css']
})
export class ListadoClientesComponent implements OnInit {

  clientes: Cliente[];
  constructor(private clientesService: ClientesService) { }

  ngOnInit() {
    this.clientes = this.clientesService.getClientes();
  }
}
```

# PRACTICA SERVICIOS

## Código de la vista

Ahora podemos ver cómo sería la vista, código HTML, del listado de componentes.

```
<h2>
  Listado clientes
</h2>
<div *ngIf="! clientes.length">No hay clientes por el momento</div>
<div>
  <article *ngFor="let cliente of clientes">
    <span>{{cliente.nombre}}</span>
    <span>{{cliente.cif}}</span>
    <span>{{cliente.direccion}}</span>
    <span>{{cliente.grupo}}</span>
  </article>
</div>
```

# PRACTICA SERVICIOS

Código de la vista

No lo hemos comentado anteriormente, pero puedes darle un poco de estilo a los componentes editando el archivo de CSS. Por ejemplo, este sería un poco de CSS que podrías colocar en el fichero "listado-clientes.component.css".

```
article {
  display: flex;
  border-bottom: 1px solid #ddd;
  padding: 10px;
  font-size: 0.9em;
}
span {
  display: inline-block;
  width: 22%;
  margin-right: 2%;
}
```

# PRACTICA SERVICIOS

Usar el componente del listado

Para usar este componente de listado de clientes, ya que lo queremos invocar desde el módulo raíz, tienes que ampliar el exports del module "clientes.module.ts".

```
exports: [
  AltaClienteComponent,
  ListadoClientesComponent
]
```

Como para el anterior componente, de alta de clientes, ya habíamos importado el módulo de clientes, no necesitas hacer nada más. Ahora ya puedes usar el usar el componente directamente en el template del componente raíz "app.component.html".

```
<app-listado-clientes></app-listado-clientes>
```

# PRACTICA SERVICIOS

Usar el componente del listado

Es hora de ver de nuevo la aplicación construida y disfrutar del buen trabajo realizado. El aspecto de la aplicación que hemos realizado debería ser más o menos el siguiente:

# 11. Trabajando con servicios

## Creación de servicios

Podremos crear nuestros servicios con:

**ng g s ruta/nombre_servicio**

Un **warning** de este tipo nos dirá que dicho servicio no está siendo usado aún:

```
WARNING Service is generated but not provided, it must be provided to be used
```

- Ahora vamos a incorporar un servicio a nuestro proyecto:

Como primer paso escribiremos en consola: **ng g s servicio/productos**

```
PS C:\Dev\workspace\jhernand\proyecto1> ng g s servicio/productos
CREATE src/app/servicio/productos.service.spec.ts (372 bytes)
CREATE src/app/servicio/productos.service.ts (138 bytes)
```

# 11. Trabajando con servicios

## Creación de servicios

Una vez creado el servicio pasaremos a darlo de alta en nuestra aplicación en el **app.module.ts**:

```
import { ProductosService } from "./servicio/productos.service";
```

Y en la parte del @**NgModule:**

```
providers: [ProductosService],
```

En nuestro fichero '**productos.service.ts**' vamos a crear un objeto **productos** y lo inicializamos con el json de: https://github.com/dotnet-presentations/ContosoCrafts/blob/master/src/wwwroot/data/products.json

```
export class ProductosService {

  productos: any = [
    {
      "Id": "jenlooper-cactus",
      "Maker": "@jenlooper",
      "img": "https://user-images.githubusercontent.com/41929050/61567048-13938600-aa33-11e9-9cfd-712191013192.jpeg",
      "Url": "https://www.hackster.io/agent-hawking-1/the-quantified-cactus-an-easy-plant-soil-moisture-sensor-e65393",
      "Title": "The Quantified Cactus: An Easy Plant Soil Moisture Sensor",
      "Description": "This project is a good learning project to get comfortable with soldering and programming an Arduino.",
      "Ratings": [
        5,
        5
      ]
    },
```

# 11. Trabajando con servicios

## Creación de servicios

Vamos a utilizarlo en nuestro componente **c-seccion3**, por ello deberemos darlo de alta a través de la **inyección de dependencias**:

app > c-seccion3 > c-seccion3.component.ts > ...

```
import { ProductosService } from '../servicio/productos.service';
```

```
constructor(private route: ActivatedRoute, private router: Router, private productosService: ProductosService) {  }
```

Una vez dado de alta, ahora vamos a crearnos nuestro array de productos en el componente y lo inicializamos en el constructor:

```
export class CSeccion3Component implements OnInit {

  nombrePersona: String;
  productos: any;

  constructor(private route: ActivatedRoute, private router: Router, private productosService: ProductosService) {
    this.productos = productosService.productos;
  }
```

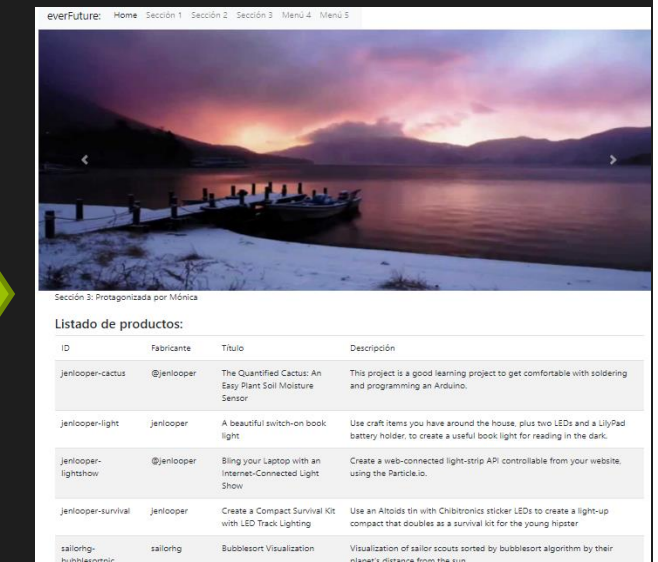
# 11. Trabajando con servicios

## Creación de servicios

A continuación personalizamos la vista del componente **c-seccion3** para mostrar los productos:

src > app > c-seccion3 > c-seccion3.component.html > ...

```
1   <p>Sección 3: <label *ngIf="nombrePersona">Protagonizada por {{ nombrePersona }}</label></p>
2
3   <h4>Listado de productos:</h4>
4
5   <table class="table table-striped">
6       <thead>
7           <tr>
8               <td>ID</td> <td>Fabricante</td> <td>Título</td> <td>Descripción</td>
9           </tr>
10      </thead>
11      <tbody>
12          <tr *ngFor="let prod of productos">
13              <td> {{prod.Id}} </td>
14              <td> {{prod.Maker }} </td>
15              <td> {{prod.Title}} </td>
16              <td> {{prod.Description}} </td>
17          </tr>
18      </tbody>
19  </table>
20  <br />
```



ever FUTURE

# 12
# Consumiendo REST

# 12. Consumiendo REST

## HTTP

Las comunicaciones **HTTP** son una pieza imprescindible en el desarrollo web. Con Angular siempre han sido fáciles y potentes, así vamos a ver lo sencillo que es consumir un servicio **REST**:

```
class HttpClient {
  constructor(handler: HttpHandler)
  request(first: string | HttpRequest<any>, url?: string, options: {...}): Observable<any>
  delete(url: string, options: {...}): Observable<any>
  get(url: string, options: {...}): Observable<any>
  head(url: string, options: {...}): Observable<any>
  jsonp<T>(url: string, callbackParam: string): Observable<T>
  options(url: string, options: {...}): Observable<any>
  patch(url: string, body: any | null, options: {...}): Observable<any>
  post(url: string, body: any | null, options: {...}): Observable<any>
  put(url: string, body: any | null, options: {...}): Observable<any>
}
```

# 12. Consumiendo REST

A continuación creamos nuestro nuevo componente: '**ng g c usuario**':

```
PS C:\Dev\workspace\jhernand\proyecto1>  ng g c usuario
CREATE src/app/usuario/usuario.component.html (22 bytes)
CREATE src/app/usuario/usuario.component.spec.ts (635 bytes)
CREATE src/app/usuario/usuario.component.ts (279 bytes)
CREATE src/app/usuario/usuario.component.css (0 bytes)
UPDATE src/app/app.module.ts (1766 bytes)
```

Tras esto crearemos un nuevo servicio: '**ng g s servicio/usuarios**':

```
PS C:\Dev\workspace\jhernand\proyecto1> ng g s servicio/usuarios
CREATE src/app/servicio/usuarios.service.spec.ts (367 bytes)
CREATE src/app/servicio/usuarios.service.ts (137 bytes)
```

Y añadimos la configuración necesaria al '**app.module.ts**':

app > app.module.ts > ...

```
import { UsuariosService } from "./servicio/usuarios.service";
```

```
@NgModule({
...
  providers: [ProductosService, UsuariosService],
```

Lo siguiente que vamos a incorporar es la importación del **HttpClientModule**:

app > app.module.ts > AppModule

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

```
@NgModule({
  imports: [
    HttpClientModule,
```

# 12. Consumiendo REST

A continuación empezaremos a crear la vista de nuestro componente **usuario**:

src > app > usuario > usuario.component.html > ...

```
1   <h4>Probando HTTP desde Angular</h4>
2
3   <div class="container">
4       <div class="row">
5           <div class="col-8">Listado de usuarios</div>
6           <div class="col-4">Formulario de carga de usuario</div>
7       </div>
8   </div>
```

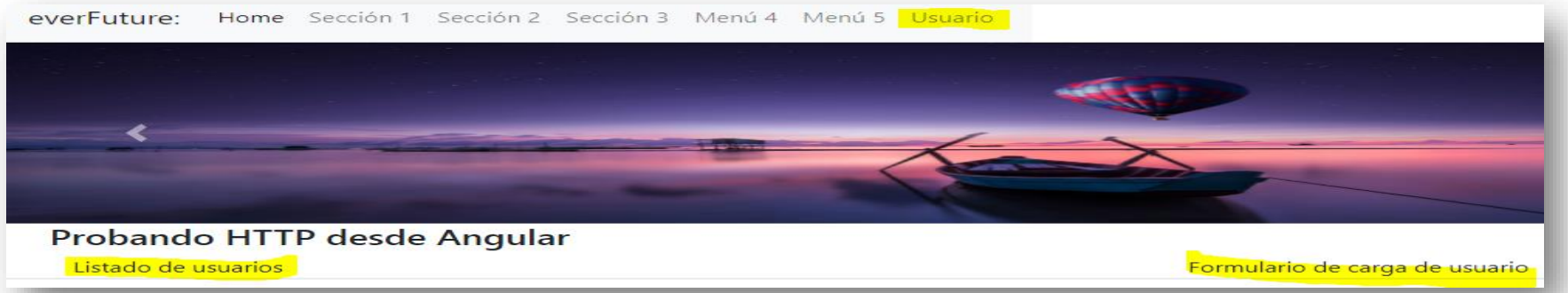
Vamos a crear una nueva opción de menú para mostrarlo:

app > header > header.component.html > div.row

```
<li class="nav-item"> <a class="nav-link" [routerLink]="['/usuario']">Usuario</a> </li>
```

app > app.module.ts > ...

```
{path:'usuario', component: UsuarioComponent}
```



# 12. Consumiendo REST

Añadimos un objeto '**usuarios**' a **usuario.component.ts**:

app > usuario > usuario.component.ts > ...

```
export class UsuarioComponent implements OnInit {
  usuarios: any;
  constructor() { }
```

Definimos la vista del '**Listado de Usuarios**':

```
<h4>Probando HTTP desde Angular</h4>

<div class="container">
    <div class="row">
        <div class="col-8">
            <table class="table table-striped">
                <thead>
                        <th>ID</th> <th>Nombre</th> <th>Apellidos</th> <th>Email</th> <th>Teléfono</th> <th>*</th>
                </thead>
                <body>
                    <tr *ngFor="let usu of usuarios">
                        <td>{{usu.id}}</td> <td>{{usu.name}}</td> <td>{{usu.username}}</td>
                        <td>{{usu.email}}</td> <td>{{usu.phone}}</td>
                        <td>
                            <!--<button (click)="" eliminarUsuario (usu.id) class="btn btn-primary">Eliminar</button>-->
                        </td>
                    </tr>
                </body>
            </table>
        </div>
        <div class="col-4">Formulario de carga de usuario</div>
```

```
<h4>Probando HTTP desde Angular</h4>

<div class="container">
  <div class="row">
    <div class="col-8">
      <table class="table table-striped">
        <thead>
            <th>ID</th> <th>Nombre</th> <th>Apellidos</th> <th>Email</th> <th>Teléfono</th> <th>*</th>
        </thead>
        <body>
          <tr *ngFor="let usu of usuarios">
            <td>{{usu.id}}</td> <td>{{usu.name}}</td> <td>{{usu.username}}</td>
            <td>{{usu.email}}</td> <td>{{usu.phone}}</td>

            <td>
              <!--
<button (click)="eliminarUsuario (usu.id)" class="btn btn-primary">Eliminar</button>-->
            </td>
          </tr>
        </body>
      </table>
    </div>
    <div class="col-4">Formulario de carga de usuario</div>
  </div>
</div>
```

# 12. Consumiendo REST

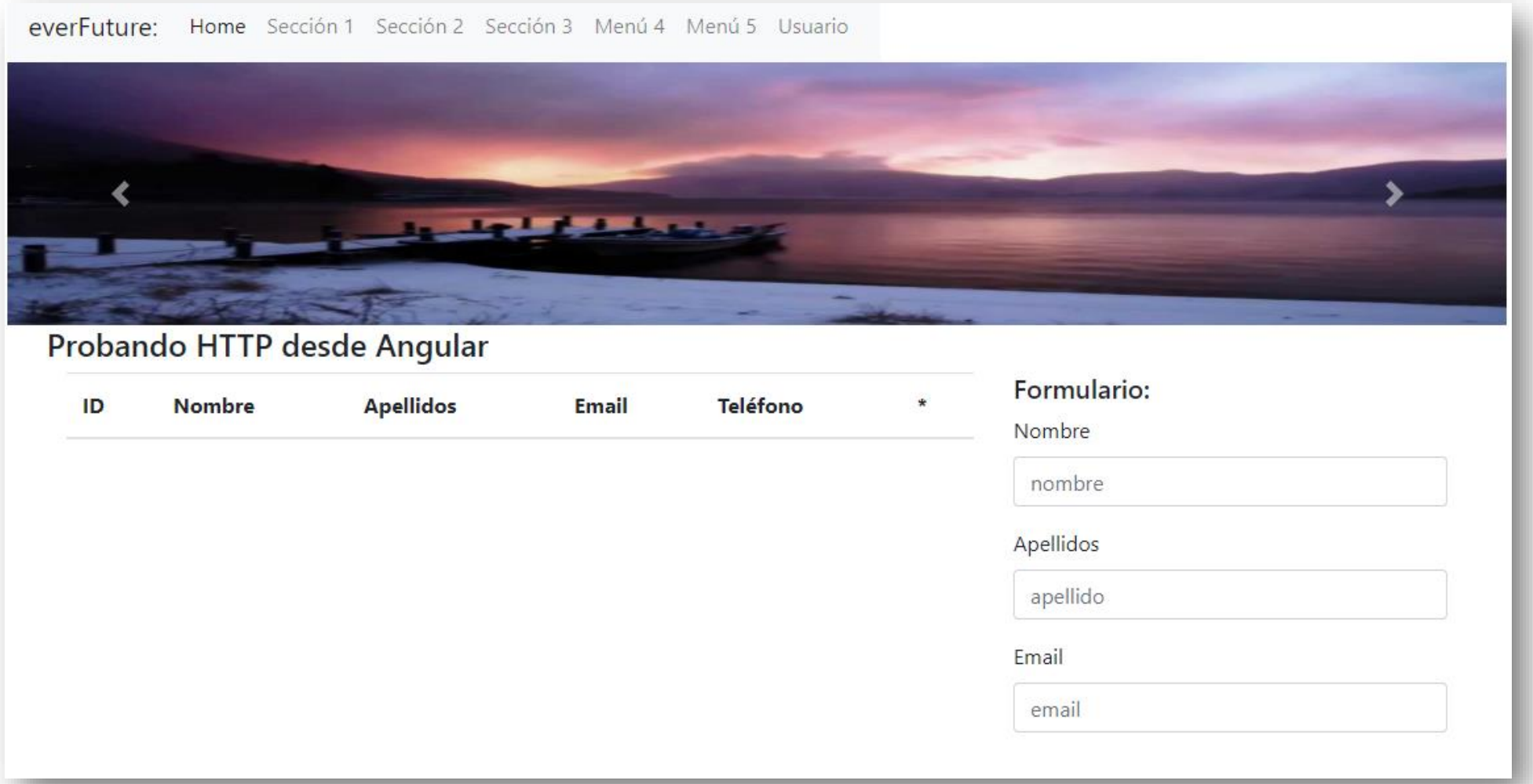
A continuación definimos el '**Formulario**':

```
<div class="col-4">
    <h5>Formulario:</h5>
    <div class="form-group">
        <label for="controlNombre">Nombre</label>
        <input type="text" class="form-control" placeholder="nombre" />
    </div>
    <div class="form-group">
        <label for="controlApellido">Apellidos</label>
        <input type="text" class="form-control" placeholder="apellido" />
    </div>
    <div class="form-group">
        <label for="controlEmail">Email</label>
        <input type="text" class="form-control" placeholder="email" />
    </div>
    <!--<button (click)="nuevoUsuario ()" class="btn btn-primary">Añadir</button>-->
</div>
```

```
<h4>Probando HTTP desde Angular</h4>
...
      <div class="col-4">
        <h5>Formulario:</h5>
        <div class="form-group">
          <label for="controlNombre">Nombre</label>
          <input type="text" class="form-
control" placeholder="nombre" />
        </div>
        <div class="form-group">
          <label for="controlApellido">Apellidos</label
>
          <input type="text" class="form-
control" placeholder="apellido" />
        </div>
        <div class="form-group">
          <label for="controlEmail">Email</label>
          <input type="text" class="form-
control" placeholder="email" />
        </div>
        <!--
<button (click)="nuevoUsuario ()" class="btn btn-
primary">Añadir</button>-->
      </div>
```

# 12. Consumiendo REST

Ya tenemos lista nuestra visual:



# 12. Consumiendo REST

Y ahora **importaremos** e **inyectaremos** nuestro **servicio Usuario**:

src > app > usuario > usuario.component.ts > ...

```
1   import { Component, OnInit } from '@angular/core';
2   import {UsuariosService} from "../servicio/usuarios.service";
3
4   @Component({
5     selector: 'app-usuario',
6     templateUrl: './usuario.component.html',
7     styleUrls: ['./usuario.component.css']
8   })
9   export class UsuarioComponent implements OnInit {
10    usuarios: any = [{id:'', name:'', surname:'', email:'', address:{}, phone:'', website:'', company:{} }];
11    usuarioAagregar: any = {name:'', surname:'', email:'' };
12    constructor(private usuariosService:UsuariosService) { }
13
14    ngOnInit(): void {
15    }
16
17    eliminarUsuario (id_usuario) {
18      console.log("Eliminar usuario:" +id_usuario)
19    }
20
21    nuevoUsuario () {
22      console.log("NuevoUsuario.");
23    }
24  }
```

Aprovecharemos para definir de forma más completa nuestro objeto 'usuarios' y crearnos 'usuarioAagregar'.

# 12. Consumiendo REST

Ahora vamos a vincular los controles **input** a nuestros campos:

```
<div class="col-4">
    <h5>Formulario:</h5>
    <div class="form-group">
        <label for="controlNombre">Nombre</label>
        <input type="text" [(ngModel)]="usuarioAagregar.name" class="form-control" placeholder="nombre" />
    </div>
    <div class="form-group">
        <label for="controlApellido">Apellidos</label>
        <input type="text" [(ngModel)]="usuarioAagregar.surname" class="form-control" placeholder="apellido" />
    </div>
    <div class="form-group">
        <label for="controlEmail">Email</label>
        <input type="text" [(ngModel)]="usuarioAagregar.email" class="form-control" placeholder="email" />
    </div>
    <button (click)="nuevoUsuario ()" class="btn btn-primary">Añadir</button>
</div>
```

# 12. Consumiendo REST

Ahora vamos a definir la lógica de nuestro **servicio** '**usuarios**':

Primero identificamos los métodos a utilizar. Y luego añadiremos los **imports** y **DI** necesarios:

src > app > servicio > usuarios.service.ts > ...

```
1  import { Injectable } from '@angular/core';
2  import { HttpClient, HttpHeaders} from '@angular/common/http';
3  import { Observable } from 'rxjs';
4  @Injectable({
5    providedIn: 'root'
6  })
7  export class UsuariosService {
8
9    constructor(private httpClient:HttpClient) { }
10
11   listarUsuarios(){
12
13   }
14
15
16   nuevoUsuario ( nuevoUsuario:any ) {
17   }
18
19   eliminarUsuario(id_usuario) {
20
21   }
22 }
```

# 12. Consumiendo REST

La implementación de los métodos de nuestro **servicio** '**usuarios**' sería la siguiente:

```
export class UsuariosService {

  constructor(private httpClient:HttpClient) { }

  listarUsuarios():Observable<any> {
    return this.httpClient.get("https://jsonplaceholder.typicode.com/users");
  }

  eliminarUsuario(id_usuario):Observable<any> {
    return this.httpClient.delete("https://jsonplaceholder.typicode.com/users/"+id_usuario);
  }

  nuevoUsuario ( nuevoUsuario:any ) {
    let json = JSON.stringify (nuevoUsuario);

    let header = new HttpHeaders().set('Content-Type', 'application/json');

    return this.httpClient.post("https://jsonplaceholder.typicode.com/users/", json, {headers: header});
  }
}
```

(basándonos en el simulador REST: https://jsonplaceholder.typicode.com/ )

# 12. Consumiendo REST

Ahora engancharemos la lógica del listado en nuestro '**usuario.component.ts**':

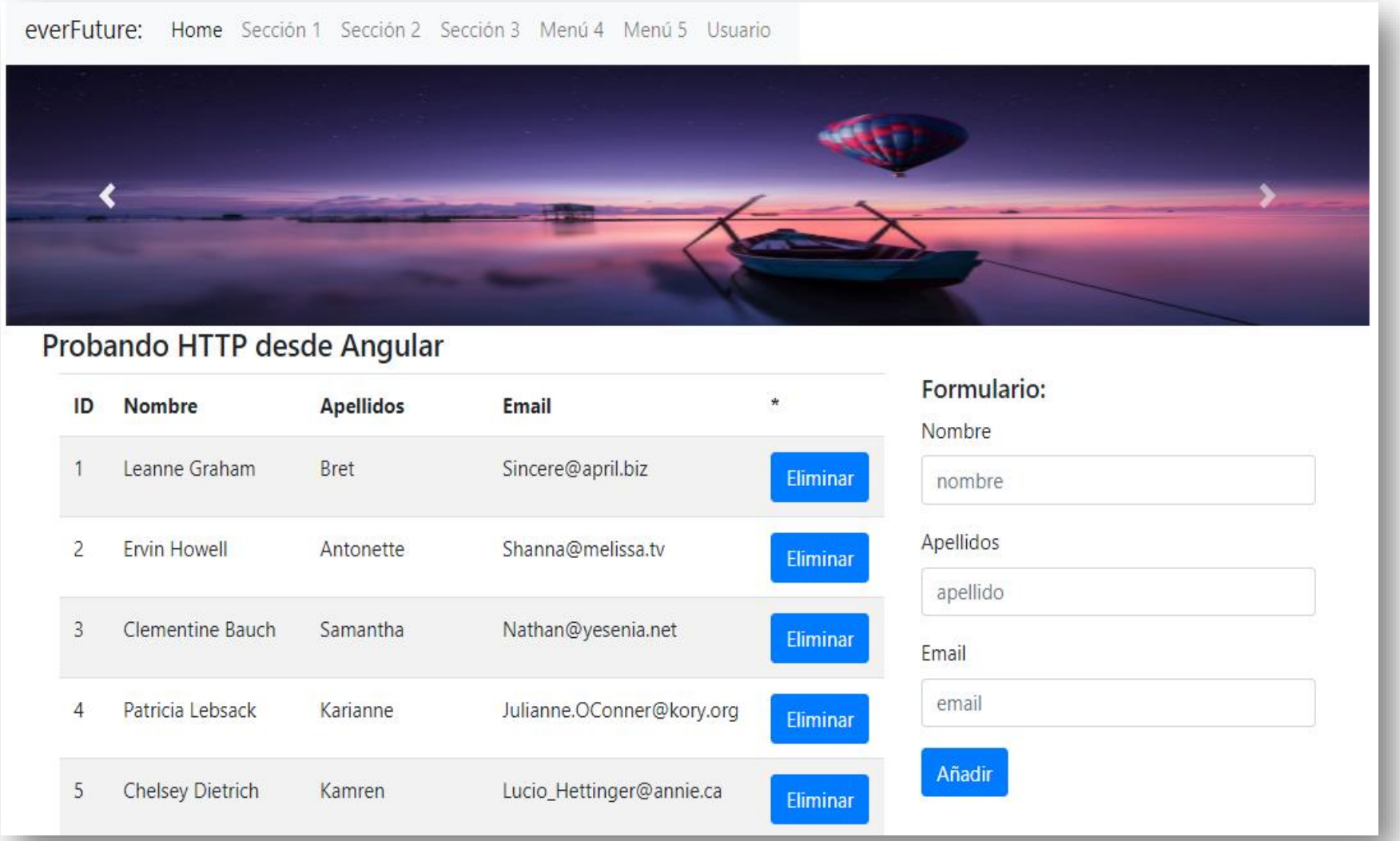
```
export class UsuarioComponent implements OnInit {
  usuarios: any; // = [{id:'', name:'', surname:'', email:'', address:{}, phone:'', website:'', company:{} }];
  usuarioAagregar: any = {name:'', surname:'', email:'' };

  constructor(private usuariosService:UsuariosService) {
    this.listarUsuarios();
  }

  ngOnInit(): void {
  }

  listarUsuarios () {
    this.usuariosService.listarUsuarios().subscribe(
                                          resultado => {this.usuarios = resultado;},
                                          error => {console.log(JSON.stringify(error));});
  }
```

Y comentamos el campo teléfono para que nos quepan todos los campos en pantalla:

usuario.component.html > div.container > div.row > div.col-8

```
    <thead>
        <th>ID</th> <th>Nombre</th> <th>Username</th> <th>Email</th> <!--<th>Teléfono</th>--> <th>*</th>
    </thead>
    <tbody>
        <tr *ngFor="let usu of usuarios">
            <td>{{usu.id}}</td> <td>{{usu.name}}</td> <td>{{usu.username}}</td>
            <td>{{usu.email}}</td> <!--<td>{{usu.phone}}</td> -->
            <td>
                <button (click)="eliminarUsuario (usu.id)" class="btn btn-primary">Eliminar</button>
            </td>
        </tr>
    </tbody>
</table>
```

# 12. Consumiendo REST



# 12. Consumiendo REST

Ahora vamos a implementar el evento '**eliminar**':

```
eliminarUsuario (id_usuario) {
  console.log("Eliminar usuario:" +id_usuario)
  this.usuariosService.eliminarUsuario(id_usuario).subscribe (
                                        _resultado => { this.listarUsuarios(); },
                                        error => {console.log(JSON.stringify(error));}
                                      );
}
```

Y a continuación el de **añadir nuevo usuario**:

```
nuevoUsuario () {
  console.log("NuevoUsuario.");
  this.usuariosService.nuevoUsuario (this.usuarioAagregar).subscribe (
                                      _resultado => { this.listarUsuarios(); },
                                      error => {console.log(JSON.stringify(error));}
                                    );
}
```

Visualmente no nos van a funcionar, pero es porque estamos trabajando con un falso REST API.