Angular

Una plataforma open source para desarrollo de aplicaciones web

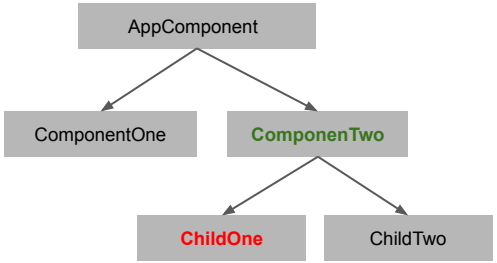
+Info: https://angular.io/docs

Uso de routing
Continuación

Routing - Rutas hijas y pasaje de parámetros

Alguna rutas pueden ser accesibles dentro del contexto de otras rutas, para este caso es posible usar rutas hijas. Convenientemente también se pueden pasar parámetros de la siguiente manera:

```
export const routes: Routes = [
  { path: '', redirectTo: 'component-one', pathMatch: 'full' },
  { path: 'component-one', component: ComponentOne },
  { path: 'component-two/:id', component: ComponentTwo,
    children: [
      { path: '', redirectTo: 'child-one', pathMatch: 'full' },
      { path: 'child-one', component: ChildOne },
      { path: 'child-two', component: ChildTwo }
    ]
  }
];
```



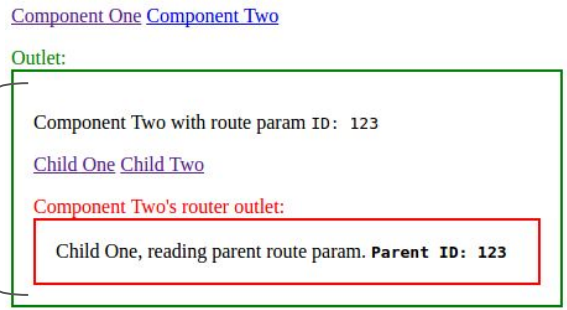
Ejemplo de URL: component-two/123 ó component-two/123/child-one

Podría aplicarse a URLs como: productos/31/ ó productos/31/caracteristicas ó productos/31/especificacion

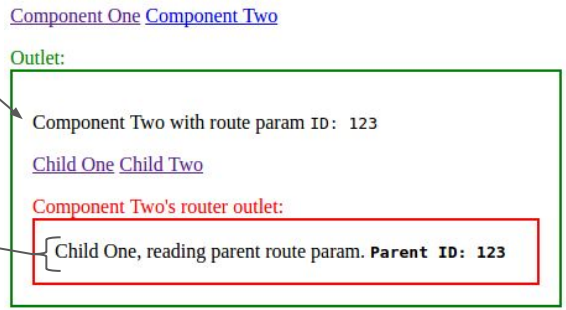
Routing - Rutas hijas y pasaje de parámetros

```
import {Component} from '@angular/core';
import { Router } from '@angular/router';

@Component({
  standalone: true,
  selector: 'app',
  template: `
    <nav>
      <a [routerLink]="['/component-one']">Component One</a>
      <a [routerLink]="['/component-two', 123]">Component Two (id: 123)</a>
    </nav>

    <div style="color: green; margin-top: 1rem;">Outlet:</div>
    <div style="border: 2px solid green; padding: 1rem;">
      <router-outlet></router-outlet>
    </div>
  `
})
export class AppComponent {

  constructor (private router: Router) {}

}
```

Pasaje de parámetro en una ruta
Equivalente a tipear /component-two/123 en el navegador



```
import { Component } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

@Component({
  standalone: true,
  selector: 'component-two',
  template: `
    <p>Component Two with route param <code>ID: {{ id }}</code></p>
    <nav>
      <a [routerLink]="['child-one']">Child One</a>
      <a [routerLink]="['child-two']">Child Two</a>
    </nav>
    <div style="color: red; margin-top: 1rem;">
      Component Two's router outlet:
    </div>
    <div style="border: 2px solid red; padding: 1rem;">
      <router-outlet></router-outlet>
    </div>
  `
})
export default class ComponentTwo {
  private id: number;

  constructor(private route: ActivatedRoute) {}

  private ngOnInit() {
    this.sub = this.route.params.subscribe(params => {
       this.id = +params['id']; // (+) converts string 'id' to a number
    });
  }

  private ngOnDestroy() {
    this.sub.unsubscribe();
  }
}
```

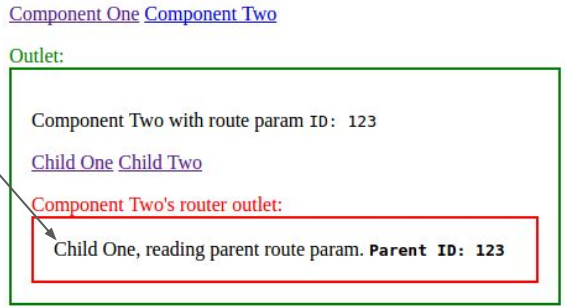


Obtiene los parámetros de la ruta activa

```ts
import { Component } from '@angular/core';
import { Router, ActivatedRoute } from '@angular/router';

@Component({
  standalone: true,
  selector: 'child-one',
  template: `
  Child One, reading parent route param.
    <b><code>Parent ID: {{ parentRouteId }}</code></b>
  `
})
export default class ChildOne {
  private sub: any;
  private parentRouteId: number;

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

  ngOnInit() { //otra forma -> this.route.parent.params.subscribe
    this.sub = this.router.routerState.parent(this.route)
      .params.subscribe(params => {
        this.parentRouteId = +params["id"];
      });
  }

  ngOnDestroy() {
    this.sub.unsubscribe();
  }
}
```

Obtiene los parámetros del padre

```ts
import { Component } from '@angular/core';

@Component({
  standalone: true,
  selector: 'child-two',
  template: 'Child Two'
})
export default class ChildTwo {
}
```



Ejemplo online de ruteo avanzado:
https://angular.io/generated/live-examples/router/stackblitz.html

Mas información de ruteo: https://angular.io/guide/router

Inyección de dependencias y servicios

Inyección de dependencias

- La inyección de dependencias **es un patrón de diseño POO**, en el que se suministran objetos a una clase en lugar de ser la propia clase la que crea los objetos.
- Angular tiene **su propio framework de DI**
- Utiliza un ***injector*** para llevar a cabo esta tarea.
- Un ***provider*** provee el valor de una dependencia. El ***injector*** depende del ***provider*** para inyectar **servicios** en componentes u otros servicios.
- Un **servicio** es una clase que se registra generalmente mediante el decorator **@Injectable()**

DI - Creando un servicio

Angular implícitamente crea un ***injector*** para toda la aplicación

Con angular-cli:

```
$ ng generate service logger
```

..crea logger.service.ts

//src/app/services/logger.service.ts

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class LoggerService {
  logs: string[] = [];

  log(mensaje: string): void {
    this.logs.push(mensaje);
    console.log(mensaje);
  }
}
```

significa que el servicio se proporcionará en el inyector raíz de la aplicación

DI - Usando el servicio en un componente

```
import { Component } from '@angular/core';
import { LoggerService } from './services/logger.service';


@Component({
  selector: 'app-root',
  standalone: true,
      //providers: [LoggerService],
  template: `<p>Consulta la consola</p>`
})
export class AppComponent {
  constructor(logger: LoggerService) {
    logger.log('AppComponent inicializado');
  }
}
```

..registrando un provider acá hace que Angular cree una instancia de LoggerService por cada instancia del componente

DI - Creando un servicio que usa otro servicio

```
$ ng generate service services/hero    ..crea hero.service.ts

//src/app/services/hero.service.ts
import { Injectable } from '@angular/core';
import { LoggerService } from './logger.service';

@Injectable({
  providedIn: 'root'
})
export class HeroService {
  private heroes = ['Iron Man', 'Captain Marvel', 'Spider-Man'];

  constructor(private logger: LoggerService) {}

  obtenerHeroes(): string[] {
        //const logger = inject(LoggerService); ..a partir de 14+ también se puede utilizar inject()
      this.logger.log('Se solicitaron los héroes');
      return this.heroes;
  }
}
```

DI - Usando HeroService en un componente

```
import { Component } from '@angular/core';
import { HeroService } from './services/hero.service';

@Component({
  selector: 'app-heroes',
  standalone: true,
  imports: [CommonModule], // Necesario para *ngFor y *ngIf
  template: `
    <h2>Héroes</h2>
    <ul>
      <li *ngFor="let heroe of heroes">{{ heroe }}</li>
    </ul>
  `
})
export class HeroesComponent {
  heroes: string[];

  constructor(heroService: HeroService) {
    this.heroes = heroService.obtenerHeroes();
  }
}
```

Comunicaciones con servidor remoto utilizando HTTP

La clase HttpClient

- La clase **HttpClient** implementa un cliente HTTP.
- **HttpClient** es una clase inyectable, con métodos que nos permiten realizar peticiones HTTP.
- La invocación de los métodos de esta clase nos retorna un objeto **Observable**.

,

HttpClient

Si estamos utilizando módulos, para usar **HttpClient** es necesario importar **HttpClientModule**.
Si estamos utilizando componentes standalone, se puede importar **HttpClientModule** en el componente.

/src/app/app.module.ts

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

@NgModule({
  imports: [
    BrowserModule,
    // importar HttpClientModule después de BrowserModule.
    HttpClientModule,
  ],
  declarations: [
    AppComponent,
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule {}
```

Los componentes standalone son aquellos que no requieren estar declarados dentro de un módulo para ser utilizados

```
@Component({
  selector: 'app-standalone-example',
  standalone: true,
  imports: [CommonModule, HttpClientModule],
  template: `
    <h1>Posts (Standalone Component)</h1>
    <ul>
      <li *ngFor="let post of posts">{{ post.title }}</li>
    </ul>
    <button (click)="loadPosts()">Cargar Posts</button>
  `,
  styles: ``
})
export class StandaloneComponent  {...
```

+Info: https://angular.io/guide/http

HttpClient

Si todos los componentes son standalone, se puede utilizar **`provideHttpClient()`** en el providers de la aplicación

/src/main.ts

```
import { bootstrapApplication } from '@angular/platform-browser';
import { provideHttpClient } from '@angular/common/http';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent, {
  providers: [
    provideHttpClient() // Configura HttpClient para toda la aplicación
  ]
});
```

`provideHttpClient()` es la forma moderna (standalone-friendly) de configurar HttpClient

Reemplaza completamente la necesidad de importar `HttpClientModule`

La clase HttpClient

Principales métodos de la clase **HttpClient**, para realizar peticiones HTTP:

get(url: string, options: {...}): Observable<any> *15 sobrecargas*

post(url: string, body: any | null, options: {...}): Observable<any> *15 sobrecargas*

put(url: string, body: any | null, options: {...}): Observable<any> *15 sobrecargas*

delete(url: string, options: {...}): Observable<any> *15 sobrecargas*

request(first: string | **HttpRequest**<any>, url?: string, options: {...}): Observable<any> *17 sobrecargas*

Tener en cuenta lo siguiente:

- **No se ejecuta ninguna petición HTTP hasta que no se llama al método subscribe()** sobre el objeto Observable.
- Los objetos HttpRequest, HttpHeaders y HttpParams son inmutables.

Arrow functions

ECMAScript 6 permite definir funciones anónimas mediante la sintaxis arrow function

función anónima

```
setTimeout(function() {
    console.log("setTimeout called!");
}, 1000);
```

función anónima con sintaxis arrow function

```
setTimeout(() => {
    console.log("setTimeout called!")
}, 1000);
```

Si el cuerpo es solo una expresión se puede escribir sin las llaves

```
setTimeout(() => console.log("setTimeout called!"), 1000);
```

HttpClient utiliza objetos observables

Los observables son una colección invocable de valores futuros. La implementación se encuentra en la librería RxJS que más adelante detallaremos.

```
// Create simple observable that emits three values
const myObservable = of(1, 2, 3);        //of() devuelve un Observable

// Create observer object
const myObserver = {
   next: x => console.log('Observer got a next value: ' + x),
   error: err => console.error('Observer got an error: ' + err),
   complete: () => console.log('Observer got a complete notification'),
};

// Execute with the observer object
myObservable.subscribe(myObserver);

// Logs:
// Observer got a next value: 1
// Observer got a next value: 2
// Observer got a next value: 3
// Observer got a complete notification
```

Un objeto observable empieza a publicar valores sólo cuando alguien se suscribe

El método subscribe está sobrecargado, también permite recibir tres funciones

```
myObservable.subscribe(
  x => console.log('Observer got a next value: ' + x),
  err => console.error('Observer got an error: ' + err),
  () => console.log('Observer got a complete notification')
);
```

La clase HttpClient

Ejemplo de uso de método get

```
export class ItemsService {


  constructor(private http: HttpClient) { }


  getItems(){
     ..
     http.get('/api/items')  // devuelve un objeto obserbable
     .subscribe(data => {    // espera los datos en formato JSON
         console.log(data['someProperty']);
      });
}
```

- El método get de HttpClient retorna un objeto Observable.
- El método subscribe() de Observable dispara la petición HTTP en forma asincrónica.
- Al llegar la respuesta se ejecuta la función pasada como parámetro.

HttpClient: JSON como respuesta por defecto

HttpClient devuelve directamente el body de la respuesta en formato JSON. Pero es posible indicar mediante la propiedad responseType otro formato.

```
//old Angular Http service
http.get('/api/items')
.map(res => res.json())
.subscribe(data => {
    console.log(data['someProperty']);
});
```

```
Desde Angular 4.3+ JSON es el formato default
//new Angular HttpClient service
http.get('/api/items')
.subscribe(data => {//data is already a JSON object
    console.log(data['someProperty']);
});
```

```
Es posible especificar otro formato de esta manera

//new Angular HttpClient service
http.get('/api/items', {responseType: 'text'})
.subscribe(data => {   // data is a string
    console.log(data);
});

Posibles responseType: arraybuffer, blob, json (es la opción por defecto), text
```

```
Es posible acceder a los headers de esta manera

//new Angular HttpClient service
http.get('/api/items', {observe: 'response'})
.subscribe(response => {   //get() retorna un Observable de HttpResponse
    console.log(response.headers.get('X-Custom-Header');
    console.log(response.body['someProperty']); //response.body is a JSON
});
```

HttpClient: Ejemplo de sobrecarga

Una de las sobrecargas de get que construye una petición GET e interpreta el body como un JSON y retorna un objeto HttpResponse:

```
get(url: string, options: {
    headers?: HttpHeaders | {
        [header: string]: string | string[];
    };
    observe: 'response';
    params?: HttpParams | {
        [param: string]: string | string[];
    };
    reportProgress?: boolean;
    responseType?: 'json';
    withCredentials?: boolean;
}): Observable<HttpResponse<Object>>
```

Recibe una url y un objeto options.

En este caso el objeto options contiene básicamente un objeto: **{observe: 'response'}**, *las otras propiedades de options son opcionales.*

+Info API de HttpClient: https://angular.io/api/common/http/HttpClient

HttpClient: Manejo de errores

Opción 1) Se puede manejar el error utilizando el segundo parámetro del método **subscribe()**

```
this.http.get<UserResponse>('https://api.github.com/users/seeschweiler').subscribe(
      data => {
        console.log("User Login: " + data.login);
        console.log("Bio: " + data.bio);
        console.log("Company: " + data.company);
      },
      err => {
        console.log("Error occured.")
      }
    );
```

https://angular.io/api/common/http/HttpErrorResponse

```
class HttpErrorResponse extends HttpResponseBase
implements Error {
    constructor(init: {...})
    get name: 'HttpErrorResponse'
    get message: string
    get error: any | null
    get ok: false
    // inherited from common/http/HttpResponseBase
    get headers: HttpHeaders
    get status: number
    get statusText: string
    get url: string | null
    get ok: boolean
    get type: HttpEventType.Response |
                    HttpEventType.ResponseHeader
}
```

//Para obtener más información del error

```
this.http.get<UserResponse>('https://api.github.com/users/seeschweiler')
.subscribe(
   data => {
        console.log("User Login: " + data.login);
        console.log("Bio: " + data.bio);
        console.log("Company: " + data.company);
      },
   (err: HttpErrorResponse) => {
      console.log(err.error);
      console.log(err.name);
      console.log(err.message);
      console.log(err.status);
    }
 );
```

HttpClient: Manejo de errores

Opción 2) Se puede manejar el error utilizando **catchError**

```
@Injectable()
export class HeroesService {
...
  /** GET heroes from the server */
  getHeroes (): Observable<Hero[]> {
    return this.http.get<Hero[]>(this.heroesUrl)
      .pipe(       //pipe se utiliza para composición de operadores separados por coma
        retry(3),  //retry a failed request up to 3 times
        catchError((err, caught) => {
              return Observable.empty();
         })
      );
  }
...
```

```
export class HeroesComponent implements OnInit {
...
  getHeroes(): void {
    this.heroesService.getHeroes()
      .subscribe(heroes => this.heroes = heroes);    //si utilizaramos el 2do parámetro de subscribe nunca sería invocado
  }
...
```

Reactive Extensions Library for JavaScript (RxJS)

RxJS es una librería de terceros, avalada por Angular, para programación reactiva “reactive programming” usando Observables, que hace más fácil componer código asincrónico basado en Callbacks.

RxJS es una reescritura de **Reactive-Extensions/RxJS,** más performante, con mejor modularidad, manteniendo la compatibilidad hacia atrás.

RxJS combina el **patrón Observer** con el **patrón Iterator** y la programación funcional con colecciones.

Sitio → http://reactivex.io API → https://rxjs-dev.firebaseapp.com/api

Provee una clase principal: **Observable**

RxJS y la programación reactiva +Info:
http://www.arquitecturajava.com/rxjs-la-programacion-reactiva/

Reactive Extensions Library for JavaScript (RxJS)

Conceptos esenciales

- **Observable:** representa la idea de una colección invocable de futuros valores y eventos.
- **Observer:** es una colección de callbacks que saben cómo escuchar los valores entregados por el Observable. Un observer se subscribe a un Observable, para luego reaccionar a cualquier ítem o secuencia de ítems emitida por el Observable.
- **Subscription:** representa la ejecución de un Observable. Es útil para cancelar la ejecución
- **Operators:** son funciones **puras** que permiten un estilo de programación funcional para trabajar con colecciones.

RxJS - Operadores

Los *operadores* permiten transformar, combinar, manipular y trabajar con la secuencia de ítems emitida por Observables

Los *operadores* son métodos de la clase Observable. Cuando son invocados, no cambian la instancia existente de Observable, sino que devuelven un nuevo Observable cuya lógica de suscripción está basada en el primer Observable.

Se los consideran funciones **puras**, porque el primer Observable permanece sin modificaciones.

Toman un Observable como entrada y generan otro Observable como salida

Subscribirse al Observable de salida implica una subscripción al Observable de entrada.

La mayoría de los operadores trabajan sobre un Observable y retornan un Observable, lo que permite “encadenar” los operadores.

RxJS - Operadores

Algunos operadores

- public **map**(project: function(value: T, index: number): R, thisArg: any): Observable<R>

  Aplica una función a cada valor emitido por el observable fuente, y emite los valores resultantes como un Observable.

- public **catchError**(selector: function): Observable Captura errores en un observable para ser manejado retornando un nuevo observable o arrojando un error.

- public static **of**(values: ...T, scheduler: Scheduler): Observable<T> Crea un Observable que emite valores especificados como argumentos, inmediatamente uno después de otro, y luego emite una notificación de completitud.

RxJS - Importación

Para importar el conjunto principal de funcionalidades:

```
import Rx from 'rxjs/Rx';
```

Ej de uso: **Rx.Observable.of(1,2,3)**

La librería RxJS es grande, así que es aconsejable **incluir solo las características necesarias**.

Para importar sólo lo que se necesita:

```
import { Observable } from 'rxjs/Observable';

import 'rxjs/add/observable/of';

import 'rxjs/add/operator/map';
```

Ej de uso: **Observable.of(1,2,3).map(x => x + '!!!');** **1!!! 2!!! 3!!!**

RxJS - Operadores

pipe: es un método usado para componer operadores, se introdujo en la versión 5.5, para transformar código

```
of(1,2,3).map(x => x + 1).filter(x => x > 2);
```

en

```
of(1,2,3).pipe(
  map(x => x + 1),
  filter(x => x > 2)
);
```

filter: emite los ítems provenientes del Observable fuente pero que satisfagan una condición dada.

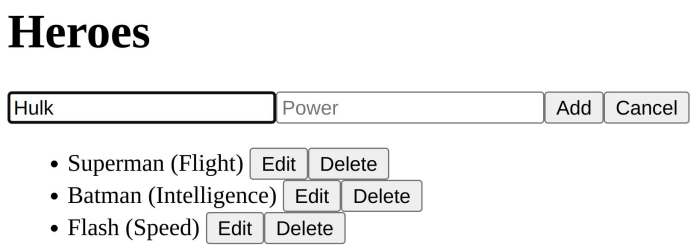
tap: antes de la versión 5.5 se llamaba *do()*, es similar al map(), pero nunca modifica el propio stream de datos que recibe. Es muy utilizado para inspeccionar o auditar el flujo de otros operadores.

En https://rxjs-dev.firebaseapp.com/api se encuentra documentada lista completa de operadores

Ejemplo de CRUD

/src/app/heroes.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { HeroesService, Hero } from './heroes.service';

@Component({
  selector: 'app-heros',
  standalone: true,
  imports: [CommonModule, FormsModule],
  template: `<h1>Heroes</h1>
    <form (ngSubmit)="save()">
      <input [(ngModel)]="form.name" name="name" placeholder="Name" required />
      <input [(ngModel)]="form.power" name="power" placeholder="Power" required />
      <button type="submit">{{ form.id ? 'Update' : 'Add' }}</button>
      <button type="button" *ngIf="form.name || form.power" (click)="cancel()">Cancel</button>
    </form>
    <div *ngIf="errorMessage" style="color:red;">{{ errorMessage }}</div>
    <ul>
      <li *ngFor="let hero of heroes">
        {{ hero.name }} ({{ hero.power }})
        <button (click)="edit(hero)">Edit</button>
        <button (click)="remove(hero.id)">Delete</button>
      </li>
    </ul>`
})
export class HerosComponent {
  heroes: Hero[] = [];
  form: Partial<Hero> = { name: '', power: '' };
  errorMessage = '';
```



Ejemplo de CRUD

/src/app/heroes.component.ts

```
constructor(private service: HeroesService) {
  this.loadHeroes();
}
loadHeroes() {
  this.service.getHeroes().subscribe({
    next: data => {
      this.heroes = data;
      this.errorMessage = '';
    },
    error: () => {//se ejecuta solo si se propaga el error
      this.errorMessage = 'Error al cargar los héroes';
    }
  });
}

save() {
  const request = this.form.id
    ? this.service.updateHero(this.form as Hero)
    : this.service.addHero(this.form);

  request.subscribe({
    next: () => {
      this.loadHeroes();
      this.resetForm();
      this.errorMessage = '';
    },
    error: () => {
      this.errorMessage = 'Error al guardar el héroe';
    }
  });
}

edit(hero: Hero) {
  this.form = { ...hero };
}

remove(id: number) {
  this.service.deleteHero(id).subscribe({
    next: () => {
      this.loadHeroes();
      this.errorMessage = '';
    },
    error: () => {
      this.errorMessage = 'Error al eliminar el héroe';
    }
  });
}

cancel() {
  this.resetForm();
  this.errorMessage = '';
}

private resetForm() {
  this.form = { name: '', power: '' };
}
}
```

El componente no utiliza HttpClient. El acceso a datos está encapsulado en un servicio.

Ejemplo de CRUD

/src/app/heroes.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable, of, throwError } from 'rxjs';
import { catchError } from 'rxjs/operators';

export interface Hero {id: number; name: string; power: string;}

@Injectable({ providedIn: 'root' })
export class HeroesService {
  private apiUrl = 'api/heroes';
  constructor(private http: HttpClient) {}

  getHeroes(): Observable<Hero[]> {
    return this.http.get<Hero[]>(this.apiUrl).pipe(
      catchError(error => {
        console.error('Error fetching heroes:', error);
        return of([]); // no propaga el error
        //return throwError(() => error); // propaga el error
      })
    );
  }

  getHero(id: number): Observable<Hero> {
    return this.http.get<Hero>(`${this.apiUrl}/${id}`).pipe(
      catchError(error => {
        console.error('Error fetching hero:', error);
        return throwError(() => error);
      })
    );
  }

  addHero(hero: Partial<Hero>): Observable<Hero> {
    return this.http.post<Hero>(this.apiUrl, hero).pipe(
      catchError(error => {
        console.error('Error adding hero:', error);
        return throwError(() => error);
      })
    );
  }

  updateHero(hero: Hero): Observable<Hero> {
    return this.http.put<Hero>(`${this.apiUrl}/${hero.id}`, hero).pipe(
      catchError(error => {
        console.error('Error updating hero:', error);
        return throwError(() => error);
      })
    );
  }

  deleteHero(id: number): Observable<void> {
    return this.http.delete<void>(`${this.apiUrl}/${id}`).pipe(
      catchError(error => {
        console.error('Error deleting hero:', error);
        return throwError(() => error);
      })
    );
  }
}
```

El componente no utiliza HttpClient. El acceso a datos está encapsulado en un servicio.