03-2021

code URJC

Desarrollo Web SPA con Angular

3. APIs REST y Servicios

Universidad Rey Juan Carlos

Micael Gallego
Correo: micael.gallego@urjc.es
Twitter: @micael_gallego

code URJC

Cliente REST

Cliente REST

- Angular dispone de su propio **cliente de API REST**
- Es un objeto de la clase **HttpClient**

```
HttpClient httpClient = ...

httpClient.get(url).subscribe(
  response => console.log(response),
  error => console.error(error)
);
```

https://angular.io/docs/ts/latest/guide/server-communication.html

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

Cliente REST

- Angular dispone de su propio **cliente de API REST**
- Es un objeto de la clase **HttpClient**

El método **subscribe** recibe dos parámetros:
1) La función que se ejecutará cuando la petición sea **correcta**
2) La función que se ejecutará cuando la petición sea **errónea**

```
HttpClient httpClient = ...

httpClient.get(url).subscribe(
  response => console.log(response),
  error => console.error(error)
);
```

https://angular.io/docs/ts/latest/guide/server-communication.html

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

Cliente REST

- Angular dispone de su propio **cliente de API REST**
- Es un objeto de la clase **HttpClient**

Si la respuesta de la API REST es un JSON, la variable **response** es un objeto JavaScript con la respuesta

```
HttpClient httpClient = .

httpClient.get(url).subscribe(
  response => console.log(response),
  error => console.error(error)
);
```

https://angular.io/docs/ts/latest/guide/server-communication.html

https://angular.io/docs/ts/latest/api/http/index/Http-class.html

Cliente REST

- **¿Cómo podemos acceder al objeto HttpClient?**
  - ¿Creamos el objeto con new?
    - Podríamos crear un objeto nuevo cada vez que nos haga falta
    - Esta opción **dificulta hacer tests automáticos** unitarios porque necesitaríamos que el servidor REST estuviese disponible y sería más difícil probar diferentes situaciones

Cliente REST

- **¿Cómo podemos acceder al objeto HttpClient?**
  - Pidiendo al framework que proporcione el objeto
    - Cuando la **aplicación** se ejecute, el objeto HttpClient sería un **objeto real** que hace peticiones al servidor REST
    - Cuando ejecutemos los **tests**, el objeto http puede ser un **sustituto (*mock*)** que se comporte como queramos en el test pero no haga peticiones reales

Cliente REST

code URJC

- **Inyección de dependencias**
  - Las **dependencias** que un módulo necesita son **inyectadas** por el sistema
  - Esta técnica de solicitar dependencias que sean inyectadas por el **framework** se denomina **inyección de dependencias**
  - Se ha hecho muy popular en el desarrollo de *back-end* en frameworks como **Spring, Java EE o Nest.js**

https://angular.io/docs/ts/latest/guide/dependency-injection.html

Cliente REST

- Para usar un objeto **HttpClient** tenemos que usar la **inyección de dependencias**

```
import { Component } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {

  constructor(private httpClient: HttpClient) { }

  search(title: string) {
    ....
  }
}
```

Cliente REST

- Para usar un objeto **HttpClient** tenemos que usar la **inyección de dependencias**

```
import { Component } from '@angular/cor
import { HttpClient } from '@angular/co

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {

  constructor(private httpClient: HttpClient) { }

  search(title: string) {
    this.httpClient.get(...)
  }
}
```

Definimos un **parámetro Http** en el **constructor** de un componente

Cuando Angular construya el componente, **inyectará** el objeto http solicitado

Cliente REST

- Para usar un objeto **HttpClient** tenemos que usar la **inyección de dependencias**

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

import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, FormsModule,
    HttpClientModule],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Cliente REST

- Para usar un objeto **HttpClient** tenemos que usar la **inyección de dependencias**

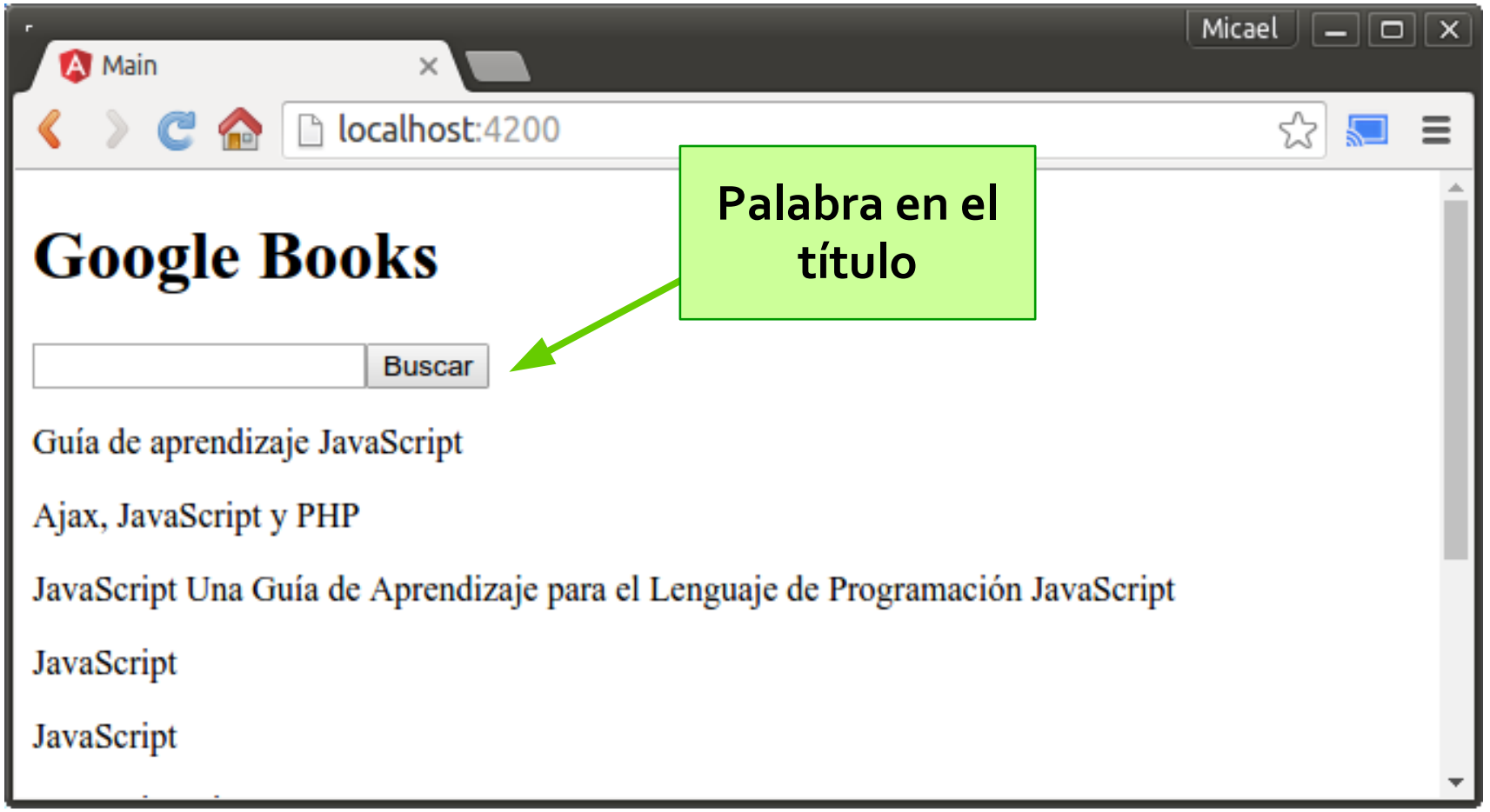
```
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/common/http';

import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, FormsModule,
    HttpClientModule],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Ponemos una referencia al módulo necesario para peticiones REST: **HttpClientModule**

Cliente REST

ejem10

- Ejemplo de **buscador libros** en Google Books



Cliente REST

code URJC

ejem10

- Ejemplo de **buscador libros** en Google Books

app.component.html

```
<h1>Google Books</h1>

<input #title type="text">

<button
(click)="search(title.value); title.value=''">
Buscar</button>

<p *ngFor="let book of books">{{book}}</p>
```

app.component.ts

ejem10

```
import { Component } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {

  private titles: string[] = [];

  constructor(private httpClient: HttpClient) { }

  search(title: string) {

    this.books = [];
    let url = "https://www.googleapis.com/books/v1/volumes?q=intitle:"+title;
    this.httpClient.get(url).subscribe(
      response => {
        let data: any = response;
        for (var i = 0; i < data.items.length; i++) {
          let bookTitle = data.items[i].volumeInfo.title;
          this.titles.push(bookTitle);
        }
      },
      error => console.error(error)
    );
  }
}
```

Cliente REST

- Peticiones **POST**

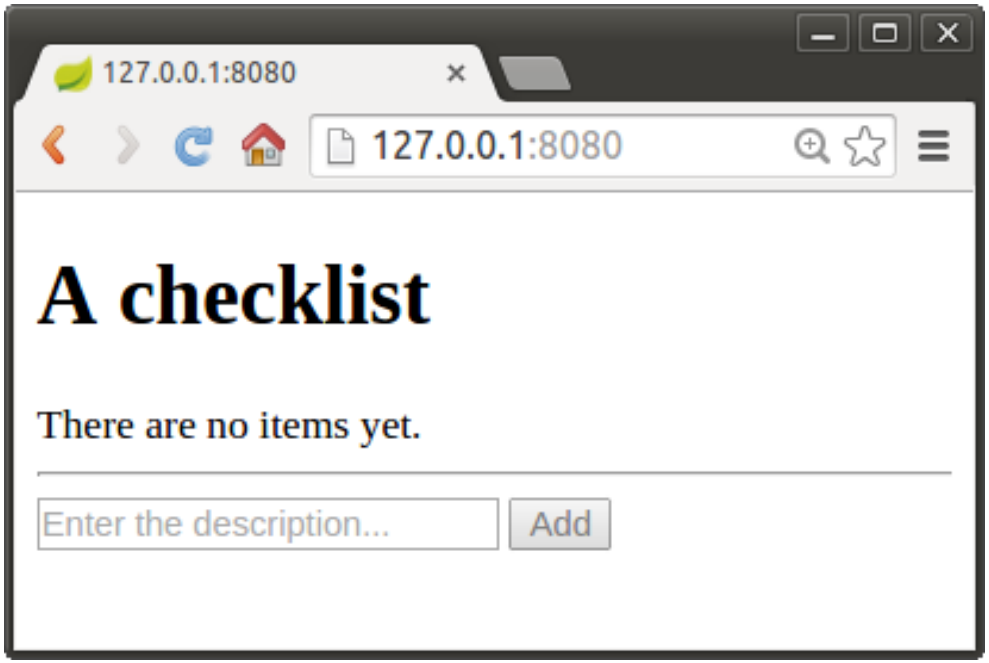
```
let data = { ... }
this.httpClient.post(url, data).subscribe(
  response => console.log(response),
  error => console.error(error)
);
```

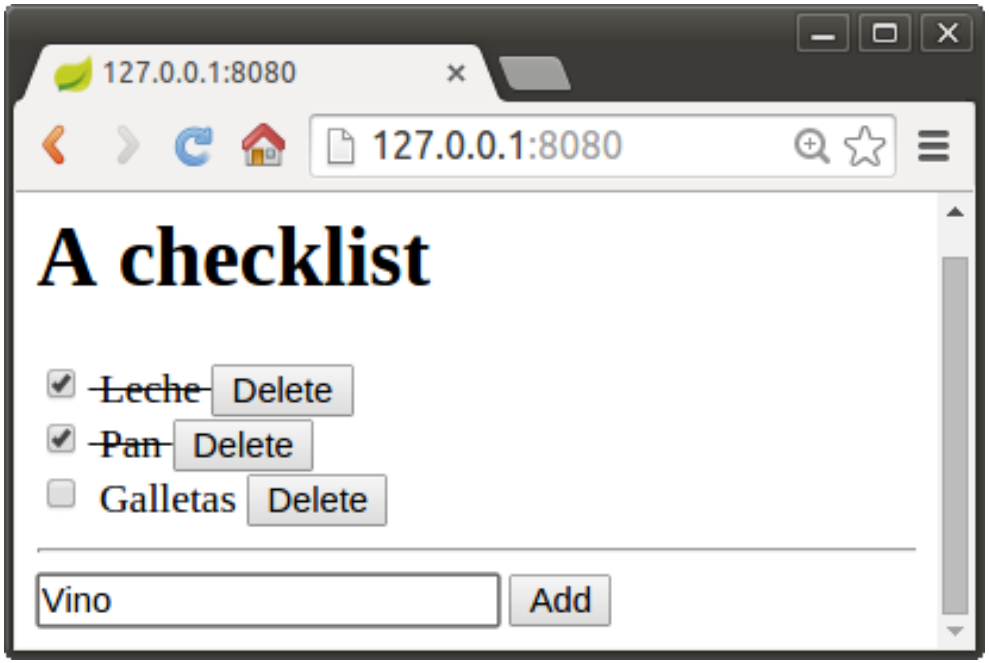
- Peticiones **PUT**

```
let data = { ... }
this.httpClient.put(url, data).subscribe(
  response => console.log(response),
  error => console.error(error)
);
```

Ejercicio 4

- Amplía el **servicio de gestión de items** para que utilice una **API REST** para gestionar los items





Ejercicio 4

code URJC

- Se proporciona un servicio web que exporta una API REST

Java

```
$ cd ejer4_backend/java

$ java -jar items-service.jar
```

Node

```
$ cd ejer4_backend/node

$ node src/app.js
```

POSTMAN

```
Items.postman_collection.json
```

Ejercicio 4

code URJC

- **API REST Items**
  - Creación de items
    - Method: POST
    - URL: http://127.0.0.1:8080/items/
    - Headers: Content-Type: application/json
    - Body:

```
{ "description" : "Leche", "checked": false }
```

    - Result:

```
{ "id": 1, "description" : "Leche", "checked": false }
```

    - Status code: 201 (Created)

Ejercicio 4

code URJC

- **API REST Items**
  - Consulta de items
    - Method: GET
    - URL: http://127.0.0.1:8080/items/
    - Result:

```
[
  { "id": 1, "description": "Leche", "checked": false },
  { "id": 2, "description": "Pan", "checked": true }
]
```

    - Status code: 200 (OK)

Ejercicio 4

code URJC

- **API REST Items**
  - Modificación de items
    - Method: PUT
    - URL: http://127.0.0.1:8080/items/1
    - Headers: Content-Type: application/json
    - Body:

```
{ "id": 1, "description" : "Leche", "checked": true }
```

    - Result:

```
{ "id": 1, "description" : "Leche", "checked": true }
```

    - Status code: 200 (OK) / 404 (Not Found)

Ejercicio 4

code URJC

- **API REST Items**
  - Modificación de items
    - Method: DELETE
    - URL: http://127.0.0.1:8080/items/1
    - Result:

```
{ "id": 1, "description" : "Leche", "checked": true }
```

    - Status code: 200 (OK) / 404 (Not Found)

Cliente REST

- **CORS (*Cross-origin Resource Sharing*)**
  - Por defecto una web no puede ejecutar peticiones REST de un servidor y puerto diferente al que fue cargado.
    - Una aplicación Angular cargada desde http://localhost:4200 por defecto no puede acceder a una API REST en http://localhost:8080
  - La API REST puede "configurar el CORS" para dar permiso a recibir peticiones de páginas web de otros dominios

El backend del ejemplo 4 tiene configurado CORS

Cliente REST

code URJC

- **CORS (*Cross-origin Resource Sharing*)**
  - Cuando la aplicación se va a desplegar en el mismo dominio que la API REST:
    - No hace falta configurar CORS en la API REST
    - La ruta de acceso a la API REST puede ser relativa (no se especifica el host y puerto)
    - Se configura un proxy en el servidor de desarrollo de Angular (ng serve)

Cliente REST

code URJC

ejem10b

- Proxy en ng serve

```
{
    "/books/*": {
        "target": "https://www.googleapis.com/",
        "secure": false,
        "logLevel": "debug",
        "changeOrigin": true
    }
}
```

```
$ ng serve --proxy-config proxy.conf.json
```

Ruta relativa en el código

```
let url = "/books/v1/volumes?q=intitle:" + title;
```

Servicios

Servicios

code URJC

- **Acoplar** en el componente la lógica de las peticiones http **no es una buena práctica**
- El componente podría llegar a ser muy **complejo** y **difícil de ampliar / modificar**
- Es mucho más **difícil implementar tests unitarios** si el componente tiene muchas responsabilidades
- Es mucho mejor **modularizar** la aplicación en **elementos** que tengan una única **responsabilidad**
  - Componente: Interfaz de usuario
  - Otro elemento: Peticiones http

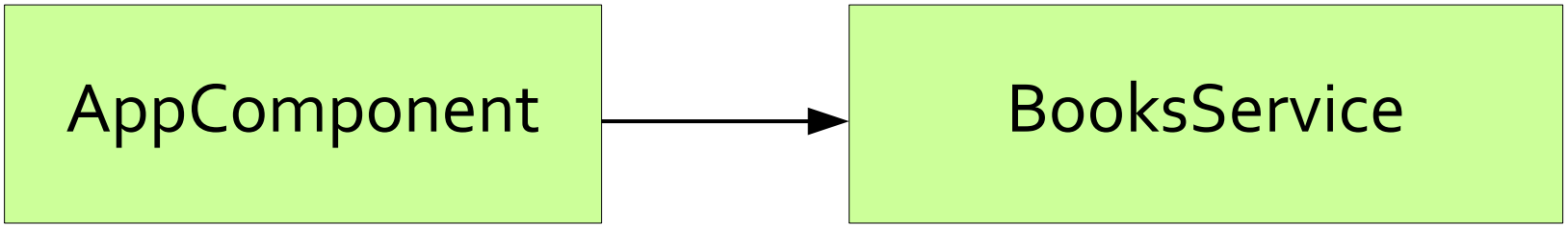
Servicios

code URJC

- A los **elementos** de la aplicación que no se encargan del interfaz de usuario se les conoce como **servicios**
- Angular ofrece muchos **servicios predefinidos**
- El objeto **HttpClient** se considera un servicio de acceso a APIs REST, pero existen más
- El desarrollador puede **implementar** sus propios **servicios** en la aplicación
- Para que sea más sencillo implementar tests, los servicios se **inyectan** en los **componentes**

Servicios

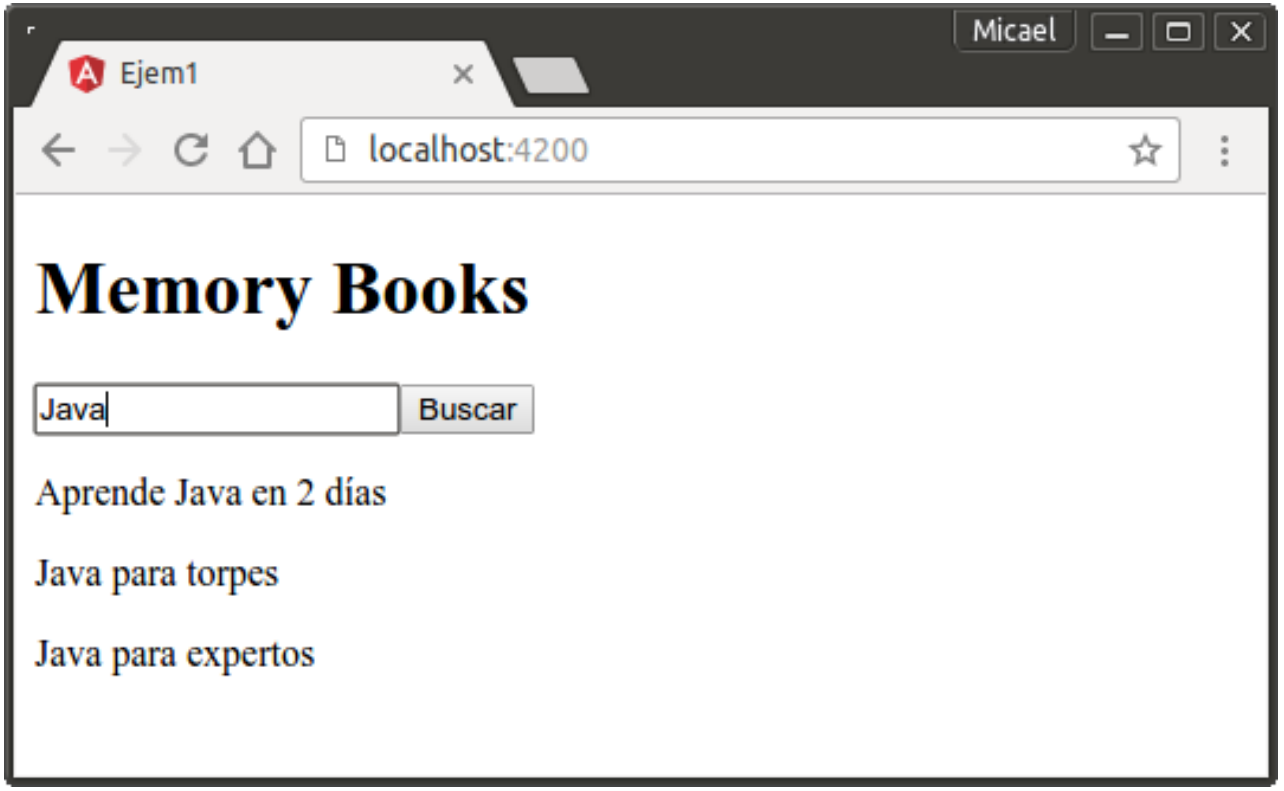
- **¿Cómo se implementa un servicio?**
  - Se crea una nueva **clase** para el servicio
  - Se anota esa clase con **`@Inyectable({providedIn:'root'})`**
  - Se pone como **parámetro en el constructor** del componente que usará ese servicio



ejem11

¿Cómo se implementa un servicio?

Ejemplo de buscador de libros con información en memoria



code URJC

ejem11

¿Cómo se implementa un servicio?

books.service.ts

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

@Injectable({ providedIn: 'root' })
export class BooksService {

  getBooks(title: string){
    return [
      'Aprende Java en 2 días',
      'Java para torpes',
      'Java para expertos'
    ];
  }
}
```

code URJC

ejem11

¿Cómo se implementa un servicio?

app.component.ts

```
import { Component } from '@angular/core';
import { BooksService } from './books.service';

@Component({
    selector: 'app-root',
    templateUrl: './app.component.html'
})
export class AppComponent {

  titles: string[] = [];

  constructor(private booksService: BooksService){}

  search(title: string){
    this.titles = this.booksService.getTitles(title);
  }
}
```

Servicios

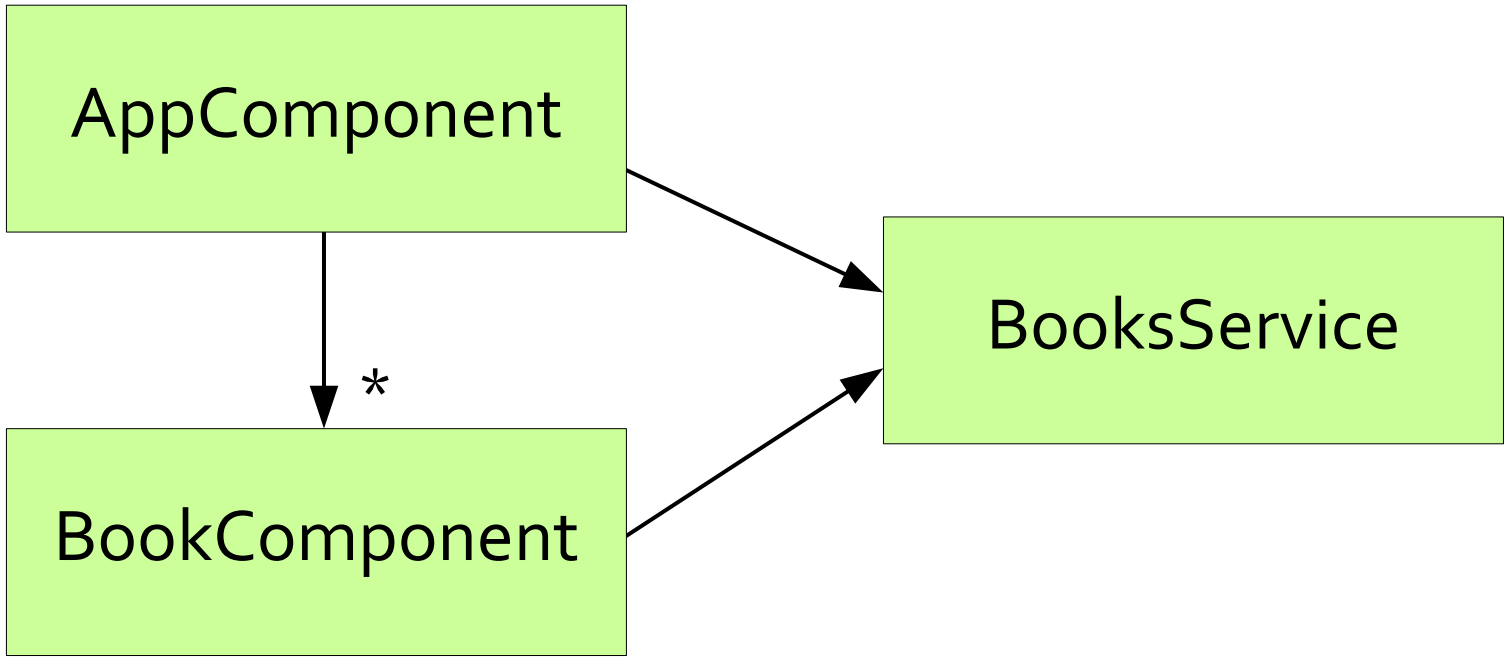
code URJC

Compartir servicios entre componentes

- Es habitual que haya un **único objeto** de cada **servicio** en la aplicación (*singleton*)
- Es decir, todos los componentes **comparten "el mismo" servicio**
- De esa forma los **servicios** mantienen el **estado** de la aplicación y los **componentes** ofrecen el **interfaz** de usuario

code URJC

Compartir servicios entre componentes



code URJC

Servicio exclusivo para componente

- Se puede hacer que servicio **no sea compartido** entre todos los componentes de la aplicación (**no sea *singleton***)
- Se puede crear un **servicio exclusivo** para un **componente y sus hijos**
- Se declara en el atributo **providers** del **@Component** y se quita del @Inyectable el providedIn
- Puede ser compartido por el componente (**padre**) y por sus componentes **hijos** (incluidos en él)

code URJC

Servicio exclusivo para componente

app.component.ts

```
import { Component } from '@angular/core';
import { BooksService } from './books.service';

@Component({
   selector: 'app-root',
   templateUrl: './app.component.html',
   providers: 'BooksService'
})
export class AppComponent {

  constructor(private booksService: BooksService){}

  ...
}
```

code URJC

Peticiones http en un servicio

- **No** es buena práctica hacer peticiones http desde un componente
- Es mejor **encapsular** el acceso al backend con API REST en un **servicio**
- **Ventajas**
  - Varios componentes pueden acceder al mismo backend **compartiendo** el mismo servicio (*singleton*)
  - Es más fácil de **testear**
  - Es una **buena práctica** (más fácil de **entender** por otros desarrolladores)

code URJC

Peticiones http en un servicio

- ¿Cómo se **implementan** los métodos de ese servicio?
  - **No** pueden devolver información de forma **inmediata**
  - Sólo pueden devolver información cuando llega la **respuesta del servidor**
  - En JavaScript los métodos **no se pueden bloquear** esperando la respuesta
  - Son **asíncronos** / **reactivos**

Peticiones http en un servicio

- ¿Cómo se **implementan** los métodos de ese servicio?

```
private service: BooksService = …

let books = this.booksService.getBooks(title);

console.log(books);
```

Un servicio que hace peticiones a una API REST **NO PUEDE** implementarse de forma **síncrona** (**bloqueante**) en JavaScript

code URJC

Peticiones http en un servicio

- Existen principalmente 3 formas de implementar un servicio con **operaciones asíncronas** en JavaScript
  - Callbacks
  - Promesas
  - Observables

code URJC

Peticiones http en un servicio

- **Callbacks:** Se pasa como parámetro una función (de *callback*) que será ejecutada cuando llegue el resultado. Esta función recibe como primer parámetro el **error** (si ha habido)

```
service.getBooks(title, (error, books) => {
  if(error){
    return console.error(error);
  }
  console.log(books);
});
```

code URJC

Peticiones http en un servicio

- **Promesas:** El método devuelve un objeto **Promise**. Con el método **then** de ese objeto se define la función que se ejecutará cuando llegue el resultado. Con el método **catch** de ese objeto se define la función que se ejecutará si hay algún error

```
service.getBooks(title)
  .then(books => console.log(books))
  .catch(error => console.error(error));
```

code URJC

Peticiones http en un servicio

- **Promesas con async / await:** Las promesas también se pueden usar con await de forma que el modelo de programación es similar a un modelo síncrono de programación

```
try {
  let books = await service.getBooks(title);
  console.log(books);
} catch(error){
  console.error(error);
}
```

code URJC

Peticiones http en un servicio

- **Observables:** Similares a las promesas pero con más funcionalidad. Con el método **subscribe** se definen las funciones que serán ejecutadas cuando llegue el resultado o si se produce un error

```
service.getBooks(title).subscribe(
  books => console.log(books),
  error => console.error(error)
);
```

code URJC

Peticiones http en un servicio

- Implementación de métodos asíncronos
  - **Callbacks:** Hay muchas librerías implementadas así. Ya no se recomienda este enfoque porque es más limitado
  - **Promesas:** La forma estándar en ES6. La forma recomendada si la funcionalidad es suficiente
  - **Observables:** Implementados en la librería **RxJS**. Es la forma recomendada por **Angular** por ser la más completa (aunque más compleja)

code URJC

Servicio con Observables de RxJS

- **RxJS:** Extensiones **reactivas** para JavaScript
- La librería **RxJS** está incluida en Angular
- Es mucho más **potente** que las **promesas** (estándar de ES6)
- Nos vamos a centrar únicamente en los aspectos que nos permitan implementar **servicios** con llamadas a una **API REST**

https://github.com/ReactiveX/RxJS

code URJC

Servicio con Observables de RxJS

ejem12

- Tenemos que ofrecer **objetos de alto nivel** a los clientes del servicio (p.e. **array de títulos de libros**)
- Al hacer una petición REST con HttpClient obtenemos un objeto **Response**
- El servicio **transforma** el objeto Response en un array de títulos

```
service.getBooks(title).subscribe(
  books => console.log(titles),
  error => console.error(error)
);
```

Objeto de alto nivel

code URJC

Servicio con Observables de RxJS

ejem12

```
...
import { map } from 'rxjs/operators';
import { Observable } from 'rxjs';

@Injectable(...)
export class BooksService {
  ...
  getTitles(title: string): Observable<string[]> {
    let url = ...
    return this.httpClient.get(url).pipe(
      map(response => this.extractTitles(response as any))
    )
  }
  private extractTitles(response) { ... }
}
```

Con el método **map** se indica la **transformación** que hacemos a la respuesta para obtener el objeto de **alto nivel**

```
service.getTitles(title).subscribe(
  titles => console.log(titles),
  error => console.error(error)
);
```

El cliente del servicio accede al **array de títulos** en vez de a la response

Servicios

app.component.ts

ejem12

```
import { Component } from '@angular/core';
import { BooksService } from './books.service';

@Component({
    selector: 'app-root',
    templateUrl: './app.component.html'
})
export class AppComponent {

    private titles: string[] = [];

    constructor(private booksService: BooksService) {}

    search(title: string) {

        this.titles = [];
        this.booksService.getTitles(title).subscribe(
          titles => this.titles = titles,
          error => console.error(error)
        );
    }
}
```

Cuando llega la respuesta se actualiza el array de books

ejem13

Servicio con Observables de RxJS

- Al igual que transformamos el resultado cuando la petición es correcta, también podemos **transformar el error** para que sea de más alto nivel
- Usamos el método **catch** para gestionar el error. Podemos **devolver un nuevo error** o simular una respuesta correcta (con un valor por defecto)

```
getBooks(title: string) {
  let url = ...
  return this.httpClient.get(url).pipe(
    map(response => this.extractTitles(response as any)),
    catchError(error => throwError('Server error')));
}
```

Lanzamos un nuevo error

code URJC

Estado en los servicios http

- **Servicios stateless (sin estado)**
  - No guardan información
  - Sus métodos devuelven valores, pero no cambian el estado del servicio
  - Ejemplo: **BooksService con llamadas a Google**
- **Servicios statefull (con estado)**
  - Mantienen estado, guardan información
  - Al ejecutar sus métodos cambian su estado interno, y también pueden devolver valores
  - Ejemplo: **BooksService con información en memoria**

code URJC

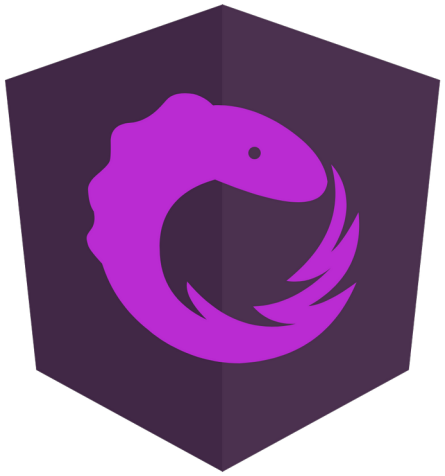
Estado en los servicios http

- **¿Stateless vs statefull?**
  - Los servicios **stateless** son más **fáciles** de implementar porque básicamente encapsulan las peticiones REST al backend
  - Pero la aplicación es **menos eficiente** porque cada vez que se visualiza un componente se tiene que pedir de nuevo la información
  - Los servicios **statefull** son más **complejos** de implementar porque hay que definir una política de sincronización entre frontend y backend
  - Pero la aplicación podría ser **más eficiente** porque no se consulta al backend constantemente

code URJC

Gestores de estado para Angular

Para implementar servicios statefull



NgRx

https://ngrx.io/

NGXS

https://www.ngxs.io/

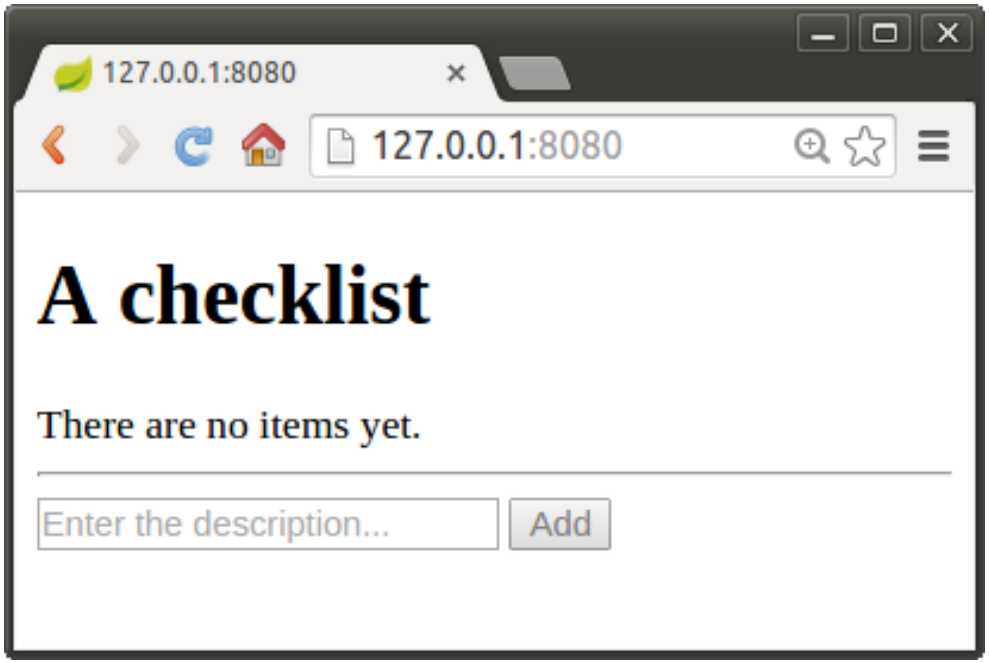
Akita

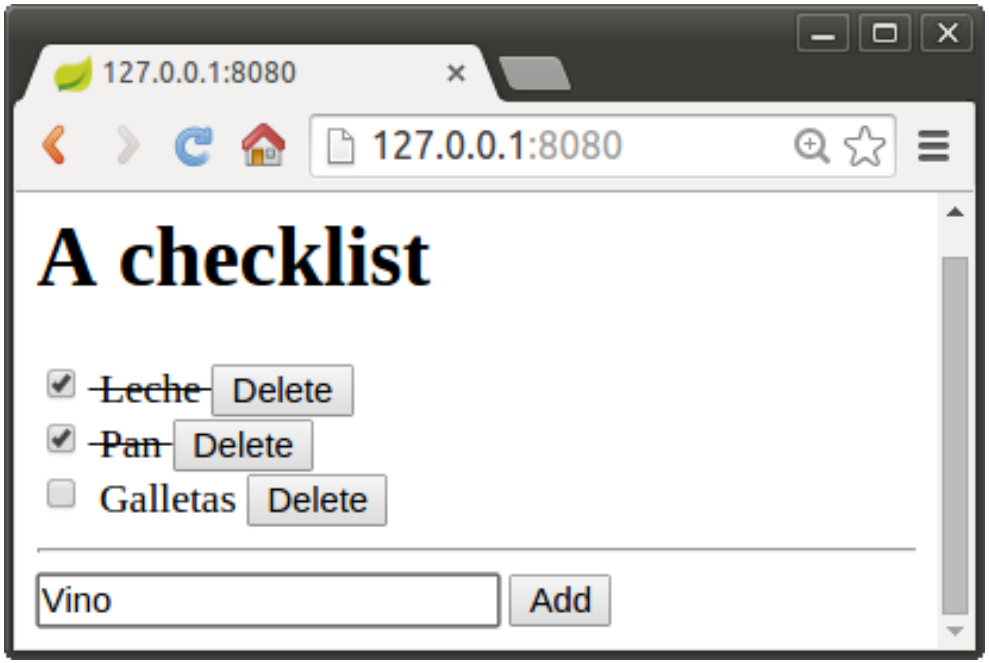
https://datorama.github.io/akita/

https://ordina-jworks.github.io/angular/2018/10/08/angular-state-management-comparison.html

Ejercicio 5

- Refactoriza el **Ejercicio 4** para que las llamadas a la API REST estén en un servicio stateless **ItemsService**





code URJC

Async Pipe

ejem14

- El **pipe async** permite trabajar directamente con **observables** y promesas en las plantillas

```
export class AppComponent {

  $titles: Observable<string>;

  constructor(private service: BooksService) { }

  search(title: string) {
    this.$titles = this.service.getTitles(title);
  }
}
```

```
<p *ngFor="let title of ($titles | async)">{{title}}</p>
```

https://angular.io/docs/ts/latest/api/common/index/AsyncPipe-pipe.html

code URJC

Async Pipe

ejem15

- Se puede asignar el valor de la promesa/observable a una nueva variable con la sintaxis "as"

```
<div *ngIf="$titles | async as titles; else loading">
  <p *ngFor="let title of titles">{{title}}</p>
  <p>{{titles.length}} total books</p>
</div>

<ng-template #loading>
  <p *ngIf="$titles !== undefined">Loading...</p>
</ng-template>
```

Ejercicio 5b

code URJC

- Refactoriza el ejercicio anterior para usar **async pipe**