Angular (Incompleto)

Introducción

Introduccción

Angular es un **web framework** creado por Google

Web framework

- **Conjunto de herramientas** cohesionado que provee una **forma estándar** de crear aplicaciones web

Introduccción

La forma habitual de trabajar con Angular es usando **Typescript**

Angular utiliza herramientas que hemos visto

- Webpack
- Babel
- Polyfills

Instalación de entorno

Entorno

Angular dispone de una aplicación **CLI** para ayudarnos a crear un nuevo proyecto

```
npm install -g @angular/cli
```

Este comando también instala todas las librerías Javascript necesarias para utilizar Angular

Entorno

Para crear una nueva **aplicación** se utiliza

```
ng new my-app
```

Este comando

- Instala las dependencias necesarias
- Crea una nueva **Welcome app**

Entorno

Para lanzar la nueva App

```
cd my-app
ng serve --open
```

- ng serve **compila**, **monta el servidor** y entra en **modo watch**
- **--open** abre nuestro navegador cargando la url de la aplicación

Ya podemos modificar el proyecto y automáticamente se aplicarán los cambios

Entorno

Las Apps de angular se construyen a partir de piezas que llamamos **componentes**

El componente raíz se llama **AppComponent** y contiene toda la página

Entorno

Cada componente tiene su propia lógica

- HTML
- CSS
- Javascript (Typescript)

Entorno

Puedes ver el código del **AppComponent** en src/app

`app.component.html`
`app.component.sass`
`app.component.ts`
`app.component.spec.ts`
`app.module.ts`

Primeros pasos:

- Eliminar el contenido de `app.component.html`
- Cambiar el título de la página en `app.component.ts`

Demo

Templating {{}}

Ejercicio entorno

Crea un nuevo proyecto con Angular CLI llamado **mini-imdb**

- Selecciona SASS como preprocesador CSS
- Limpia el html de AppComponent
- Inserta el título de la página en **app.component.ts**
- Inserta un h1 con el título de la página en **app.components.html**

Mini IMDB

Componentes

Componentes

Podemos crear nuevos componentes

```
ng generate component films
```

- El nuevo componente se crea en **src/app/films**
- Tiene la misma estructura que el **AppComponent**

Componentes

Puedes ver el código del **nuevo componente** en src/app/films

`films.component.html`
`films.component.sass`
`films.component.ts`
`films.component.spec.ts`

Componentes

Puedes ver la lógica del **nuevo componente** en src/app/films/films.component.ts

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


@Component({
  selector: 'app-films',
  templateUrl: './films.component.html',
  styleUrls: ['./films.component.sass']
})

export class FilmsComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

}
```

Componentes

Podemos declarar v**ariables de clase** en **FilmsComponent**

```
export class FilmsComponent implements OnInit {

film = 'Blade Runner'

  constructor() { }

  ngOnInit() {
  }

}
```

Componentes

y acceder a ella en films.component.html

```
<h2>{{film}}</h2>
```

Meter componentes en app.html

Demo onInit

Componentes

Si quisiéramos mostrar más datos, podemos crear un objeto

```
export class FilmsComponent implements OnInit {

   film = {
         id: 1,
         name: 'Blade Runner'
    }


  constructor() { }

  ngOnInit() {
  }

}
```

Componentes

y acceder a sus propiedades en films.component.html

```
<h2>{{film.name}} Details</h2>
<div><span>id: </span>{{film.id}}</div>
<div><span>name: </span>{{film.name}}</div>
```

Componentes

Estamos en Typescript, qué nos falta para ser más correctos?

```
export class FilmsComponent implements OnInit {

   film = {
         id: 1,
         name: 'Blade Runner'
    }


  constructor() { }

  ngOnInit() {
  }

}
```

Componentes

Podemos crear una **interfaz Film** en **src/app/film.ts**

```
export interface Film {
    id: number;
    name: string;
}
```

Componentes

y importarla en **src/app/films/films.component.ts**

```
import { Component, OnInit } from '@angular/core';
import { Film } from '../film'

// …

export class FilmsComponent implements OnInit {
    film: Film = {
        id: 1,
        name: 'Blade Runner'
    }

    constructor() { }
    ngOnInit() {
    }
}
```

Ejercicio Componentes

Crea un nuevo componente **films**

- Su HTML debe mostrar **5** películas con estos datos:
  - Id, nombre, fecha

- El objeto que guarda los datos de una película debe tener una interfaz Typescript asignada

Directivas

*ngFor

Directivas

y si tuviésemos que mostrar muchas películas?

```
export class FilmsComponent implements OnInit {
   films: string[] = [
         "Saw I",
         "Saw II",
         "Saw III",
         "Saw IV",
         "Saw V",
         "Saw VI",
         "Saw VII"
     ]
   // …
}
```

Directivas

La **directiva** *ngFor nos permite crear multiples nodos HTML imitando el comportamiento de un **bucle for of**

```
export class FilmsComponent implements OnInit {

   films: string[] = [
         "Saw I",
         "Saw II",
         "Saw III",
         "Saw IV",
         "Saw V",
         "Saw VI",
         "Saw VII"
     ]
   // …
}
```

```
<ul class="heroes">
    <li *ngFor="let film of films">
        Name: {{film}}
    </li>
 </ul>
```

Directivas

Las **directivas** como ***ngFor** se pueden añadir a cualquier **tag**, sean HTML o componentes de Angular

```
<div *ngFor="let film of films">
    Name: {{film}}
</div>
```

```
// Crearía un div por cada film
```

```
<span *ngFor="let film of films">
    Name: {{film}}
</span>
```

```
// Crearía un span por cada film
```

```
<app-film *ngFor="let film of films"></app-film>
```

```
// Crearía un componente por cada film
```

Directivas

Es posible capturar el **índice** de cada elemento

```
<div *ngFor="let film of films; index as i">
    {{i}}: {{film}}
</div>
```

```
// Crearía un div por cada film
```

0 : Saw I
1 : Saw II
2 : Saw III
3 : Saw IV
4 : Saw V
5 : Saw VI
6 : Saw VII

Ejercicio *ngFor

Partiendo del ejercicio anterior

- Mete las películas del ejercicio anterior en un array que tenga una interfaz typescript asignada

- Pinta la información de cada película siguiendo esta estructura:

```
<span class="badge">{{hero.id}}</span> {{hero.name}}
```

- Utiliza ***ngFor**

*ngIf

Directivas

La **directiva *ngIf** nos permite renderizar tags únicamente si se cumple una condición

```
//films.component.ts
export class FilmsComponent implements OnInit {

   show = true
   // …
}
```

```
//films.component.html
<div> Resultado: </div>
<div *ngIf="show"> Esto solo es visible si show == true </div>
```

Resultado:
Esto solo es visible si show == true

Directivas

La **directiva *ngIf** nos permite renderizar tags únicamente si se cumple una condición

```
//films.component.ts
export class FilmsComponent implements OnInit {

   show = false
   // …
}
```

```
//films.component.html
<div> Resultado: </div>
<div *ngIf="show"> Esto solo es visible si show == true </div>
```

Resultado:

Directivas

La **directiva *ngIf** acepta **condiciones**

```
//films.component.ts
export class FilmsComponent implements OnInit {

    num = 5
    // …
}
```

```
//films.component.html
<div> Resultado: </div>
<div *ngIf="num > 10"> Esto solo es visible si num > 10 </div>
```

Resultado:

Directivas

La **directiva *ngIf** acepta **condiciones**

```
//films.component.ts
export class FilmsComponent implements OnInit {

    num = 20
    // …
}
```

```
//films.component.html
<div> Resultado: </div>
<div *ngIf="num > 10"> Esto solo es visible si num > 10 </div>
```

Resultado:
Esto solo es visible si num > 10

Directivas

La **directiva *ngIf** acepta **logical operators**

```
//films.component.ts
export class FilmsComponent implements OnInit {

    num = 20
    show = false
    // …
}
```

```
//films.component.html
<div> Resultado: </div>
<div *ngIf="num > 10 && show == true"> Si num > 10 y show == true</div>
```

Resultado:

Directivas

La **directiva *ngIf** acepta **logical operators**

```
//films.component.ts
export class FilmsComponent implements OnInit {

    num = 20
    show = false
    // …
}
```

```
//films.component.html
<div> Resultado: </div>
<div *ngIf="num > 10 && show || true"> Si num > 10 o show == true</div>
```

Resultado:

```
Si num > 10 o show == true
```

Directivas

La **directiva *ngIf** acepta **logical operators**

```
//films.component.ts
export class FilmsComponent implements OnInit {

    num = 20
    show = false
    // …
}
```

```
//films.component.html
<div> Resultado: </div>
<div *ngIf="num > 10 && show || true"> Si num > 10 o show == true</div>
```

Resultado:
Si num > 10 o show == true

Directivas

La **directiva *ngIf** puede usar variables creadas en otras directivas

```
//films.component.html
div *ngFor="let film of films; index as i" >
    <div *ngIf="film=='Saw V'">{{i}} : {{film}}</div>
</div>
```

4: Saw V

Ejercicio *ngFor

Partiendo del ejercicio anterior

- Utiliza *ngIf para mostrar solo las películas de un año específico

Ejercicio *ngFor

Partiendo del ejercicio anterior

- Utiliza *ngIf para mostrar solo las tres primeras películas

Ejercicio estilos

Partiendo del ejercicio anterior

- Comenta los *ngIf que hemos usado hasta ahora, queremos ver toda la lista completa
- Aplica estilos a la lista hasta conseguir este diseño

Event binding

Event binding

Angular nos permite asignar funciones a los **eventos de usuario** de un nodo HTML

```
//films.component.ts
export class FilmsComponent implements OnInit {

  clickHandler(){
     console.log("div was clicked")
  }
  //...
}


//films.component.html
<div (click)="clickHandler()"> Click me </div>
```

Event binding

```
//films.component.ts
export class FilmsComponent implements OnInit {

  onMouseover(){
      console.log("Mouse in")
  }
  onMouseout(){
      console.log("Mouse out")
  }
  //...
}
```

```
//films.component.html
<div (mouseover)="onMouseover()"
     (mouseout)="onMouseout()"
> Hover me </div>
```

Event binding

Podemos pasar **parámetros** a través del event binding

```
//films.component.ts
export class FilmsComponent implements OnInit {

  clickHandler(index){
      console.log("div number " + index + " was clicked")
  }
  //...
}
```

```
//films.component.html
<div *ngFor="let film of films; index as i" (click)="clickHandler(i)"> Click me </div>l
```

Ejercicio event binding

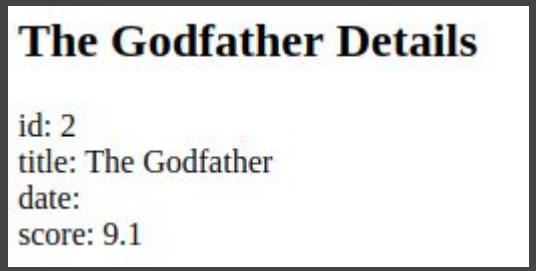
Partiendo del ejercicio anterior

- Al pulsar una película de la lista, haz event binding para que se imprima el objeto con la información de esa película por la consola

Ejercicio event binding II

Partiendo del ejercicio anterior

- Al pulsar una película de la lista, haz event binding para que se muestre la información de la película **seleccionada** en el HTML debajo de la lista de películas



Class binding

Class binding

Angular permite asignar clases a un tag **de forma condicional**

```
//films.component.ts
export class FilmsComponent implements OnInit {
  hidden = true
  //…
}
```

```
//films.component.html
<div [class.invisible]="hidden == true"> Invisible </div>
```

Class binding

También utilizando variables de **directivas**

```
//films.component.html
div *ngFor="let film of films; index as i" >
    <div [class.highlighted]=" film == 'Saw V'" >{{i}} : {{film}}</div>
</div>
```

Ejercicio class binding

Partiendo del ejercicio anterior

- Al pulsar una película de la lista, esta debe cambiar su estilo para aparecer resaltada
  - Eg: Cambia el color de fondo

Angular services

Services

Hasta ahora hemos utilizado datos falsos para nuestros componentes, vamos a usar datos verdaderos

Url: https://api.myjson.com/bins/v1p7u

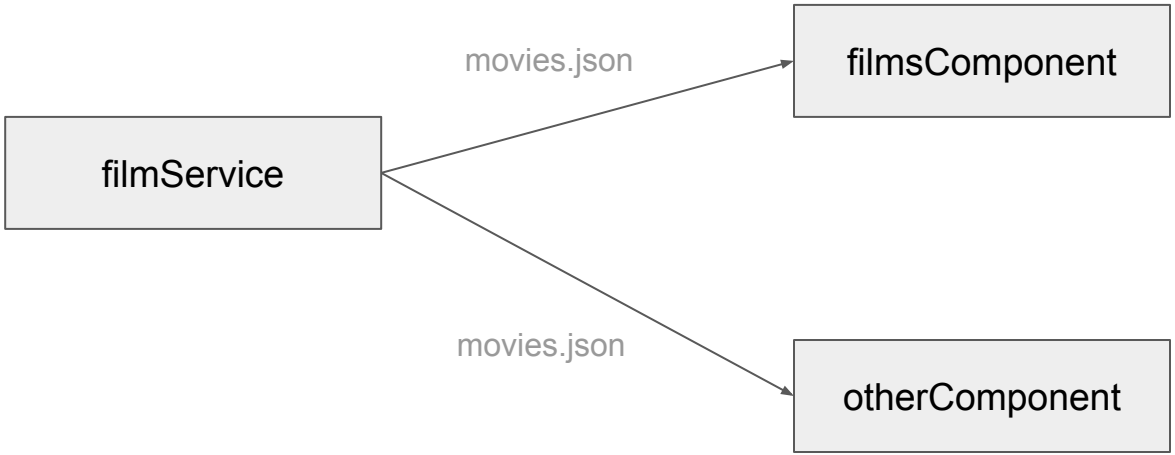
Cómo podríamos hacerlo en Angular?

Services

```
//films.component.ts
export class FilmsComponent implements OnInit {
   films: Film[]
   //...
   ngOnInit() {
        fetch('https://api.myjson.com/bins/v1p7u')
            .then(response => response.json())
            .then(data => films = data)
    }
}
```

Services - Separation of concerns

No es la mejor manera. Los componentes que se encargan de renderizar información no deberían preocuparse de la lógica de cargar datos.



Services

Crear un servicio

```
ng generate service film
```

```
// film.service.ts
import { Injectable } from '@angular/core';

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

  constructor() { }

}
```

Services

Primero vamos a devolver los datos de prueba que hemos creado antes

```
// film.service.ts
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})

export class FilmService {
  constructor() { }

  getFilms(): Film[]{
      let films = datos de prueba
      return films
  }

}
```

Services

Para conectar servicios y componentes, Angular utiliza un sistema de **inyección de dependencias**

```
// film.service.ts
import { Injectable } from '@angular/core';

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

Services

Inyectamos el servicio en filmsComponent

```
import { FilmService } from '../film.service'

export class FilmsComponent implements OnInit {
    films: Film[] = []

    constructor(private filmService: FilmService) { }

    //...

    ngOnInit() {
    }

}
```

Services

Accedemos a los datos a través del servicio

```
import { FilmService } from '../film.service'

export class FilmsComponent implements OnInit {
    films: Film[] = []

    constructor(private filmService: FilmService) { }

    //...

    ngOnInit() {
        this.films = this.filmService.getFilms()
    }

}
```

Demo

Asincronía en Angular

Asincronía

Hasta ahora el servicio ha devuelto datos de forma síncrona

```
// film.service.ts
import { Injectable } from '@angular/core';

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

  constructor() { }

  getFilms(): Film[]{
     let films = datos de prueba
     return films
  }

}
```

Hay que hacer un fetch

Asincronía

Angular gestiona la asincronía de una forma específica: el patrón **observable**

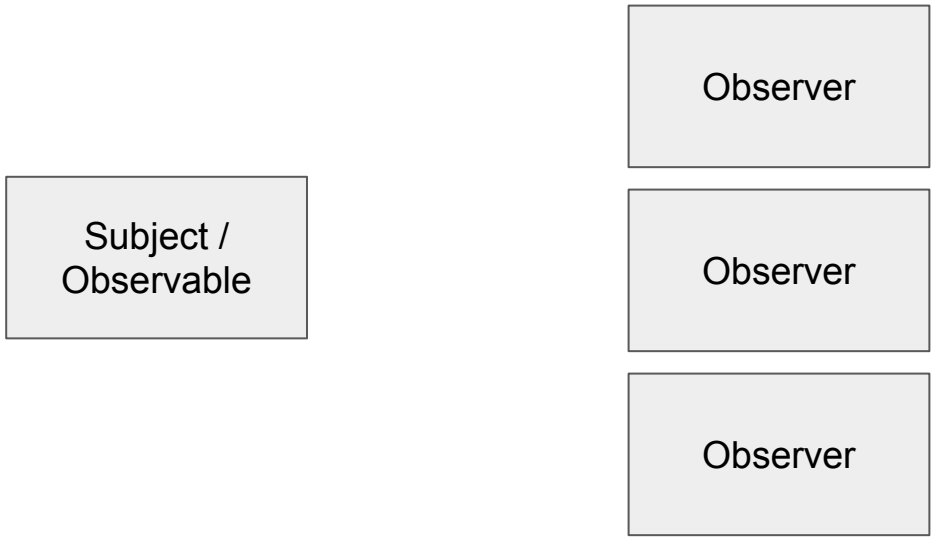
Por tanto no utilizaremos **fetch,** ya que no devuelve **observables** sino **promesas**

Angular tiene su propio módulo para hacer peticiones: **http** que sí devuelve observables

Observables

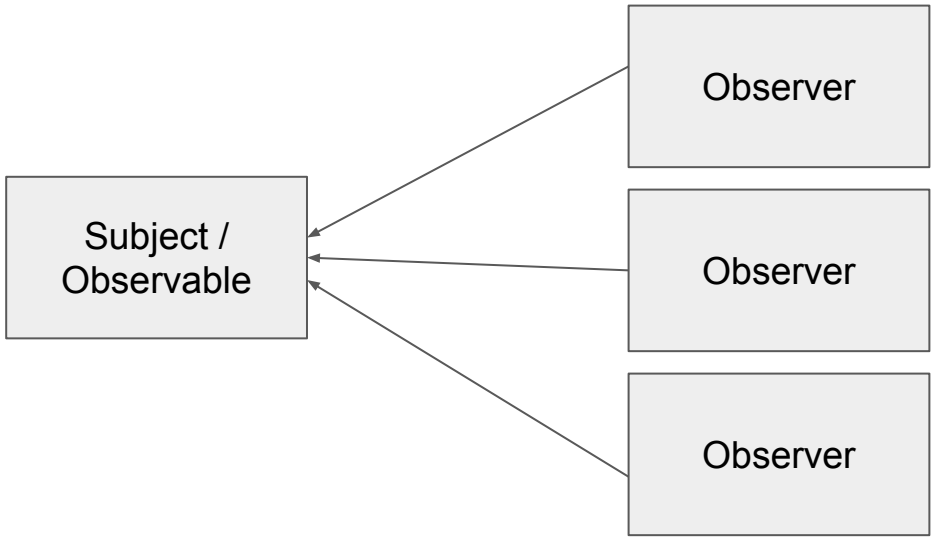
Asincronía

El patrón observable se utiliza principalmente para la **gestión de eventos**



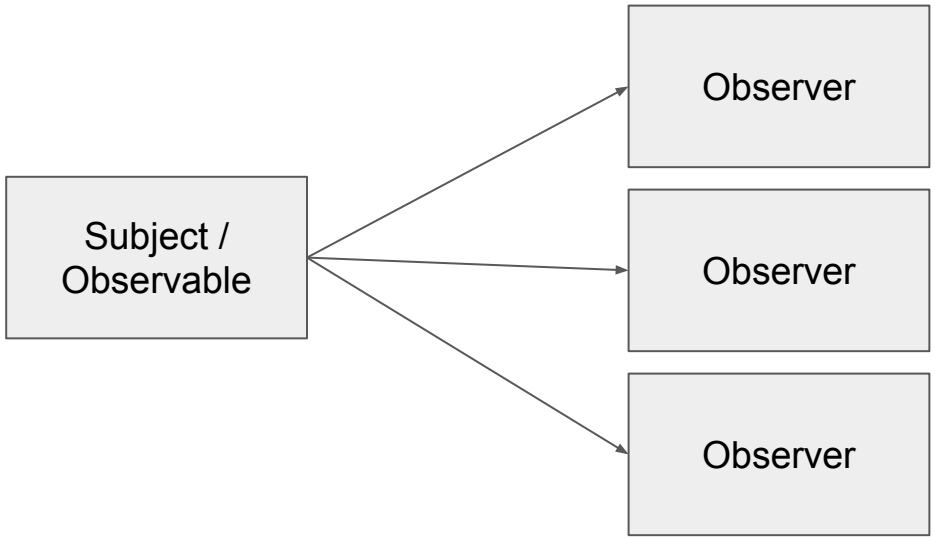
Asincronía

Los observers se **suscriben** al observable para ser notificados de cualquier cambio



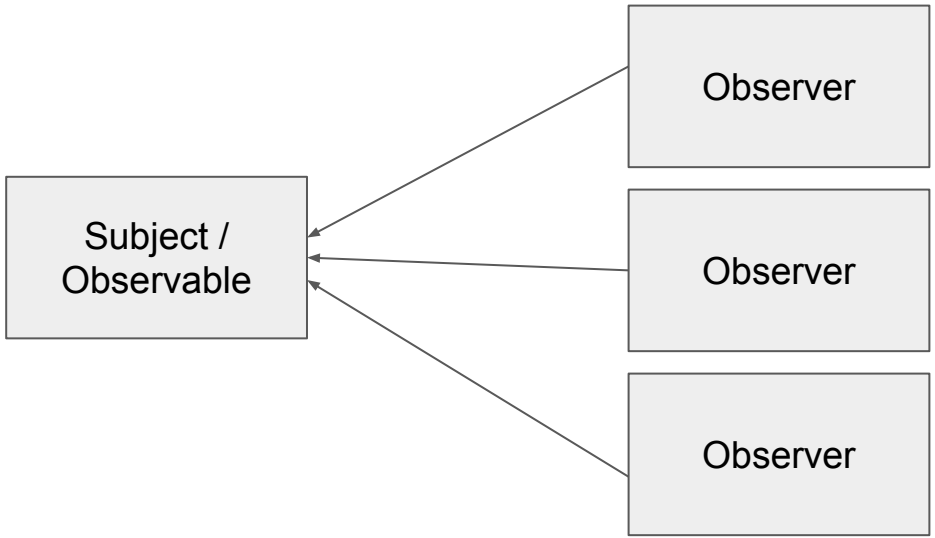
Asincronía

El observable **notifica** a los observers cuando se realiza un cambio



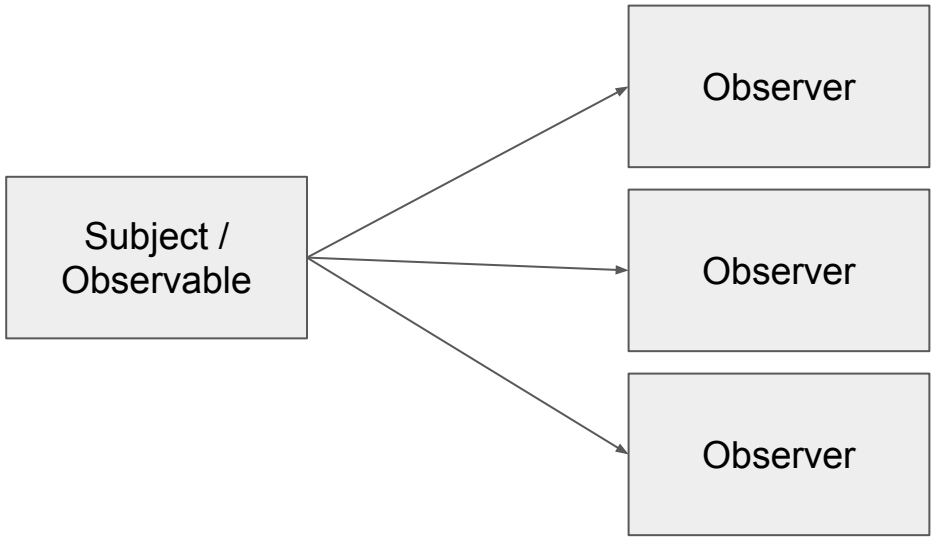
Asincronía

Los observers se **suscriben** pasando **funciones**



Asincronía

El observable **ejecuta las funciones** que ha recibido de cada observer cuando hay algún cambio



RxJS

RxJS

Es una librería de **programación reactiva** basada en **observables,** iteradores y programación funcional

Angular utiliza la **RxJS** para gestionar su asincronía

HttpClient

HttpClient

HttpClient es un **módulo** de Angular que permite realizar peticiones http, al estilo de fetch, devolviendo **observables**

No tenemos acceso a HttpClient por defecto. Es necesario **añadirlo** en el proyecto de Angular

Añadir un módulo de Angular

Añadimos módulos Angular en el fichero **app/app.module.ts**

```
// app.module.ts
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
    BrowserModule,
    // import HttpClientModule after BrowserModule.
    HttpClientModule,
  ],
 //...
```

Ahora ya podemos **inyectarlo** en cualquier parte de nuestro proyecto

Añadir un módulo de Angular

Inyectamos el módulo HttpClient

```
// film.service.ts
import { HttpClient } from '@angular/common/http'
//...
export class FilmService {

    constructor(private http: HttpClient) { }

    getFilms(): Film[] {
        //...
    }
}
```

Ahora ya tenemos acceso a **http** en **FilmService**

HttpClient

http.get nos permite hacer peticiones y se encarga de **parsear el JSON**

```
// film.service.ts
import { HttpClient } from '@angular/common/http'
//...
export class FilmService {

    constructor(private http: HttpClient) { }

    getFilms() {
        let observable = this.http.get(url)
        observable.subscribe(data => console.log(data))
    }
}
```

Cuando el observable reciba los datos, ejecutará la función que le hemos pasado mediante **subscribe**

HttpClient

http.get nos permite hacer peticiones y se encarga de **parsear el JSON**

```
// film.service.ts
import { HttpClient } from '@angular/common/http'
//...
export class FilmService {

    constructor(private http: HttpClient) { }

    getFilms() {
        this.http.get(url)
            .subscribe(data => console.log(data))
    }
}
```

Cuando el observable reciba los datos, ejecutará la función que le hemos pasado mediante **subscribe**

HttpClient

Devolvemos el observable para que **FilmsComponent** se pueda suscribir

```
// film.service.ts
import { HttpClient } from '@angular/common/http'
//...
export class FilmService {

    constructor(private http: HttpClient) { }

    getFilms():{
        return this.http.get(url)
    }
}
```

Estamos usando **Typescript**. Qué falta?

HttpClient

Podemos declarar el tipo **Observable**

```
// film.service.ts
import { HttpClient } from '@angular/common/http'
import { Observable} from 'rxjs';
//...
export class FilmService {

    constructor(private http: HttpClient) { }

    getFilms(): Observable<Film[]>{
        return this.http.get(url)
    }
}
```

HttpClient

El observable ejecutará la función que pasamos mediante **subscribe** y se actualizará la lista de películas

```
// films.component.ts
   //…
    films: string[]
   //…
   ngOnInit() {
        this.filmService.getFilms()
            .subscribe((films: Film[]) => this.films = films)
    }

}
```

Ejercicio httpClient

Partiendo del ejercicio anterior

- Crea un servicio que cargue la lista de películas de esta ‘API’
  - https://api.myjson.com/bins/v1p7u

- Conectate desde FilmsComponent para cargar la nueva lista y renderizarla
- Utililza **httpClient** y **Observables** para la carga de datos

Múltiples componentes

Múltiples componentes

Hasta ahora hemos mostrado la lista de las películas y los detalles de las películas en un mismo componente

Englobar toda la lógica de la aplicación en un único componente **no es mantenible**

Es recomendable dividir componentes grandes en pequeños componentes que tengan **una función específica**

Múltiples componentes

Vamos a mostrar la lista de películas con el componente existente y los detalles de la película mediante otro componente

Creamos un nuevo componente con la CLI

```
ng generate component film-detail
```

Ejercicio múltiples componentes

Partiendo del ejercicio anterior

- Traslada el código que pinta los detalles de la película seleccionada a un nuevo componente **film-detail**
- Descubre el obstáculo que impide resolver este ejercicio

Input properties

Input properties

Los componentes de Angular pueden pasar variables a sus descendientes

A estas variables se les llama **input properties**

```
//films.component.ts
export class FilmsComponent implements OnInit {
  chosenText = "lorem ipsum"
  //…
}
```

```
//films.component.html
<app-film-detail [text]="chosenText"></app-film-detail>
```

Input properties

Para poder recibirlas es necesario utilizar el decorador **@Input**

```
//film-detail.component.ts
import { Component, OnInit, Input } from '@angular/core'

export class FilmDetailComponent implements OnInit {
    @Input() text: string
    //film-detail.component.html

}
```

```
//film-detail.component.html
<div>Message: {{text}} </div>
```

Demo Input properties

Ejercicio input properties

Partiendo del ejercicio anterior

- Acaba de solucionar el ejercicio anterior utilizando una input property para recibir la película escogida en **film-detail**

Ejercicio múltiples componentes II

Partiendo del ejercicio anterior

- Traslada el código que pinta la lista de películas a un nuevo componente **film-list**
- Descubre el obstáculo que impide resolver este ejercicio

Output properties

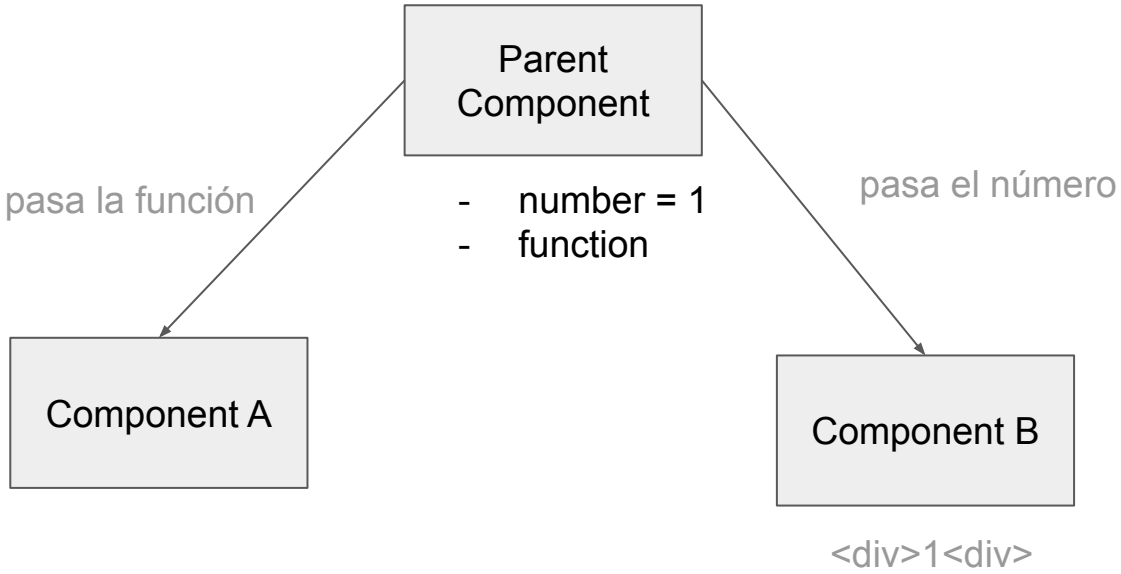
Output properties

Cuando más de un componente interviene en una parte del estado, la variable se traslada al padre para que ambos componentes tengan acceso

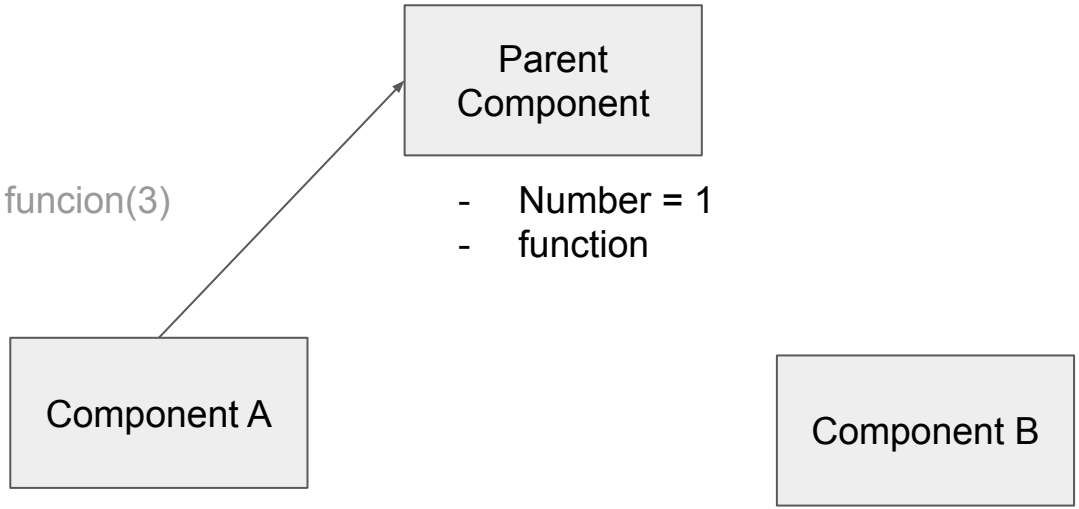
El padre pasa el estado al componente que vaya a **consultarlo** mediante **@Input**

El padre pasa una **función** al componente que vaya a **modificarlo** mediante **@Output**. Esta función modificadora es del padre, pero **la llama el hijo**

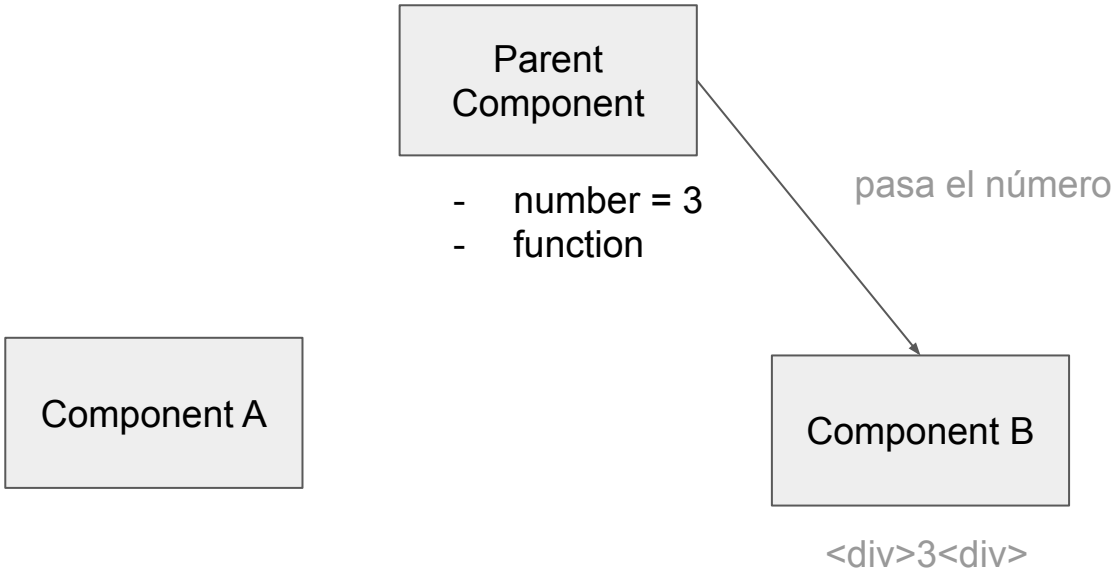
Output properties



Output properties



Output properties



Output properties

El padre contiene una función que **modifica** el valor de la variable

```
//parent.component.ts
    num = 1
    updateNum(newNumber: number): void {
        this.num = newNumber
    }
```

Le pasamos la función al hijo incluyendo un parámetro especial **$event**

```
//parent.component.html
<app-child-component (changeNumEvent)="updateNum($event)"></app-child-component>
```

Output properties

Para ejecutar la función desde el hijo tenemos que crear un **EventEmitter**

```
//child.component.ts
export class FilmListComponent implements OnInit {
    @Output() changeNumEvent = new EventEmitter<number>()
  //…
```

El EventEmitter nos permite **emitir** un valor que recibirá la función del padre

```
//child.component.ts
    changeNum(i: number) {
        this.changeNumEvent.emit(i)
    }
```

Demo output properties

Ejercicio input properties

Partiendo del ejercicio anterior

- Acaba de solucionar el ejercicio anterior utilizando una output property para conectar los componentes film-list y film-detail

SPA

Single Page Application

Una SPA consiste en incluir todos los contenidos y toda la lógica de una aplicación web en una sola página

Esta práctica permite mejorar la experiencia de usuario considerablemente al evitar la recarga de la página cada vez que cargamos nuevo contenido

Las SPAs también facilitan la subida a **producción**

Routing

Routing

A pesar de las ventajas de las SPAs, estamos acostumbrados a navegar de una forma determinada:

- Que una URL específica nos lleve a una sección concreta de una página
- Botones de back y forward del navegador

Angular nos permite implementar este modelo en nuestras SPAs mediante **routing**

Routing

El **Angular Router** se encarga de

- Interpretar una URL y decidir qué componentes se muestran
- Extraer parámetros de esa URL para que los componentes sepan qué contenido deben mostrar
- Almacenar la actividad del usuario en la historia del navegador para que los botones de back y forward funcionen correctamente

Routes

Para utilizar el router es necesario añadir el módulo **RouterModule**

```
//app.module.ts
import { RouterModule, Routes } from '@angular/router';
```

Al incluirlo en el **imports**, es necesario especificar **qué rutas debe considerar**

```
//app.module.ts
@NgModule({
  imports: [
    BrowserModule,
    //...
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true }          // Muestra información de las rutas en consola, uso para debug
    )
  ],
```

Routes

RouterModule requiere un objeto de tipo **Routes**

```
//app.module.ts
const appRoutes: Routes = [
  { path: 'list', component: FilmsComponent },
  { path: 'list/:id', component: FilmsComponent },
  { path: 'add', component: AddFormComponent },
  { path: '',   redirectTo: '/list', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
]
```

appRoutes enlaza cada URL con un **componente**

Routes

RouterModule requiere un objeto de tipo **Routes**

```
//app.module.ts
const appRoutes: Routes = [
  { path: 'list', component: FilmsComponent },
  { path: 'list/:id', component: FilmsComponent },
  { path: 'add', component: AddFormComponent },
  { path: '',   redirectTo: '/list', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
]
```

localhost:4200/list mostrará el componente FilmsComponent

Routes

RouterModule requiere un objeto de tipo **Routes**

```
//app.module.ts
const appRoutes: Routes = [
  { path: 'list', component: FilmsComponent },
  { path: 'list/:id', component: FilmsComponent },
  { path: 'add', component: AddFormComponent },
  { path: '',   redirectTo: '/list', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
]
```

Las rutas pueden aceptar parámetros a los que los componentes tienen acceso para determinar qué mostrar

Routes

RouterModule requiere un objeto de tipo **Routes**

```
//app.module.ts
const appRoutes: Routes = [
  { path: 'list', component: FilmsComponent },
  { path: 'list/:id', component: FilmsComponent },
  { path: 'add', component: AddFormComponent },
  { path: '',   redirectTo: '/list', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
]
```

localhost:4200/add mostrará el componente AddFormComponent

Routes

RouterModule requiere un objeto de tipo **Routes**

```
//app.module.ts
const appRoutes: Routes = [
  { path: 'list', component: FilmsComponent },
  { path: 'list/:id', component: FilmsComponent },
  { path: 'add', component: AddFormComponent },
  { path: '',   redirectTo: '/list', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
]
```

localhost:4200 redirigirá a **localhost:4200/list**

Routes

RouterModule requiere un objeto de tipo **Routes**

```
//app.module.ts
const appRoutes: Routes = [
  { path: 'list', component: FilmsComponent },
  { path: 'list/:id', component: FilmsComponent },
  { path: 'add', component: AddFormComponent },
  { path: '',   redirectTo: '/list', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
]
```

** (**wildcard**) encaja **cualquier** URL

La URL entrante se comprueba **secuencialmente** (por eso ** va última)

Routes

Para que el router sepa dónde debe pintar estos componentes:

```
//app.component.html
<router-outlet></router-outlet>
```

El componente asignado se pintará como descendiente de router-outlet

Demo Routes

Ejercicio Routing

Partiendo del ejercicio anterior

- Genera un nuevo componente **AddForm** (de momento deja el HTML por defecto)
- Añade routing a la aplicación
  - **list** debe mostrar el componente FilmsComponent
  - **add** debe mostrar el componente AddForm

Capturando parámetros

Parámetros

Para que un componente pueda acceder a los parámetros inyectamos la dependencia **ActivatedRoute**

```
//component.ts
import { ActivatedRoute, ParamMap } from '@angular/router'

    constructor(
        private route: ActivatedRoute,
        //...
    ) { }
```

ActivatedRoute guarda información sobre la URL actual

Parámetros

route.ParamMap nos devuelve un **Observable** al que nos podemos suscribir para recibir un objeto que contiene los parámetros

```
//component.ts
this.route.paramMap
    .subscribe((params: ParamMap) => {
         console.log(params.get('id'))]
     })
```

Tomando en consideración este parámetro podemos decidir qué información va a mostrar el componente

Ejercicio Routing II

Partiendo del ejercicio anterior

- Añade una nueva ruta para que list pueda capturar el id de la película seleccionada mediante un parámetro
- Consigue que se aplique la lógica de selección de película introduciendo rutas en el navegador

Ejercicio Routing III

Partiendo del ejercicio anterior

- Muestra la lista de películas únicamente si no hay ninguna película seleccionada
- Si hay una película seleccionada solo se debe mostrar el componente detail de esa película

Router links

Parámetros

routerLink nos permite provocar cambios en la URL desde el código

```
//component.html
<a routerLink="/add"> Add </a>
```

Demo router links

Ejercicio Routing IV

Partiendo del ejercicio anterior

- Sustituye la lógica (click) de la lista de películas por router links
- Comprueba que los botones **back** y **forward** del navegador funcionan correctamente

AppRoutingModule (pendiente)

Angular Forms

Angular forms

Angular tiene mecanismos propios para gestionar la gestión y validación de formularios

Renderizado del formulario: <form> e <input> en el html del componente

Lógica del formulario: uso de módulos Angular en el código del componente

Ejercicio Forms

Partiendo del ejercicio anterior

- Crea un formulario HTML para añadir una nueva película a la lista
  - Title
  - Year
  - Score
  - Añadir a la lista (checkbox)
  - Submit button

ReactiveForms

ReactiveForms

El módulo **ReactiveFormsModule** nos permite la creación de formularios **reactivos** en Angular

Para utilizarlo es necesario añadir el módulo a **app.module.ts**

```
// app.module.ts
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  imports: [
    // other imports ...
    ReactiveFormsModule
  ],
})
```

FormControl

Una vez añadido podemos crear objetos **FormControl** que nos permiten

- Responder a cambios en el valor de un **input concreto**
- Actualizar por código el valor
- Validar el contenido

FormControl

Para utilizar **formControl** es necesario crearlo

```
// component.ts
import { FormControl } from '@angular/forms';

export class FormComponent implements OnInit {
    name = new FormControl('')
    //...
}
```

Y conectarlo en **app.component.html**

```
<input type="text" [formControl]="name">
```

FormControl

Una vez conectado el FormControl y el input:

```
// component.ts
    this.name.setValue("newName")      // Cambia el valor actual del input

    console.log(this.name.value)       // Imprime el valor actual del input
```

El valor también es accesible en **app.component.html**

```
// component.html
Name: {{name.value}}
```

Demo FormControl

FormGroup

FormControl nos permite tomar el control de un input específico

No obstante, un formulario tiene que controlar v**arios elementos** además de la **lógica de submit** y validación a nivel de todo el formulario

Para gestionar todo eso necesitamos el objeto **FormGroup**

FormGroup

Para utilizar **formGroup** es necesario crearlo y asignar un **formControl** para cada input

```
// component.ts
import { FormControl, FormGroup } from '@angular/forms';

export class FormComponent implements OnInit {
    nameForm = new FormGroup({
        firstName: new FormControl(''),
        lastName: new FormControl('')
    })

    //...
}
```

FormGroup

En el fichero HTML conectamos con el FormGroup

```
// component.html
<form [formGroup]="nameForm" (ngSubmit)="onSubmit()">
     <input type="text" formControlName="firstName">
     <input type="text" formControlName="lastName">
     <button type="submit" [disabled]="!signUp.valid">Submit</button>
</form>
```

disabled es necesario porque todavía no estamos validando el formulario

FormGroup

Podemos acceder a los valores del FormGroup en forma de **objeto**

```
// component.ts
onSubmit() {
  console.log(this.nameForm.value);          // {firstName: "Lorem", lastName: "Ipsum"}
}
```

Los input tipo **checkbox** y **radius** se representan con true o false

Demo FormGroup

Ejercicio Forms II

Partiendo del ejercicio anterior

- Implementa ReactiveForms para controlar el formulario que has creado
- Imprime el contenido del formulario al pulsar el botón submit

Form validation

Validation

El módulo ReactiveForms también facilita la validación de campos

El primer paso es importar el objeto **Validators**

```
// component.ts
import { Validators } from '@angular/forms';
```

Validators

Los Validators se asignan en la creación de los objetos FormControl

```
// component.ts
import { FormControl, FormGroup, Validators } from '@angular/forms';

export class FormComponent implements OnInit {
    nameForm = new FormGroup({
        firstName: new FormControl('', Validators.required),
        lastName: new FormControl('', Validators.required, Validators.minLength(4))
    })

    //...
}
```

Lista completa de built-in validators

Validators

Alternativamente, se pueden asignar como atributos en los elementos input

```
// component.ts
<input type="text" required formControlName="firstName">
<input type="text" required minlength="4" formControlName="lastName">
```

Demo validators

Ejercicio Forms III

Partiendo del ejercicio anterior

- Implementa la validación del formulario
  - Confirma que todos los campos se han rellenado
  - El año debe tener exactamente 4 caracteres
  - Score debe ser un número (Pista: Regexp)
  - El checkbox debe estar activo
  - Solo entonces el usuario debe poder hacer submit