

Curso de Angular

Frontend hoy en día

Frontend roadmap

Frontend roadmap

JS en 2018-2019

State of Javascript

Creación de un proyecto

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

```
ng new social-network --prefix=sn --style=scss
```

```
cd social-network
```

```
ng serve
```

Añadir scripts y hojas de estilos de forma global

```
// Instalar simple-reset-css
npm install simple-reset-css --save

// Añadir la hoja de estilos en `angular.json`
// en arquitect/build/styles y configurar stylePreprocessorOptions scss
...
"styles": [
  "src/scss/styles.scss",
  "./node_modules/simple-css-reset/reset.css"
],
"stylePreprocessorOptions": {
  "includePaths": ["src/scss"]
}
...
```

El proyecto Angular

Tema

Estructura de directorios

Docs

<https://angular.io/guide/file-structure>

Configuración workspace

<https://angular.io/guide/workspace-config>

Configuración npm

<https://angular.io/guide/npm-packages>

Soporte navegadores

<https://angular.io/guide/browser-support>

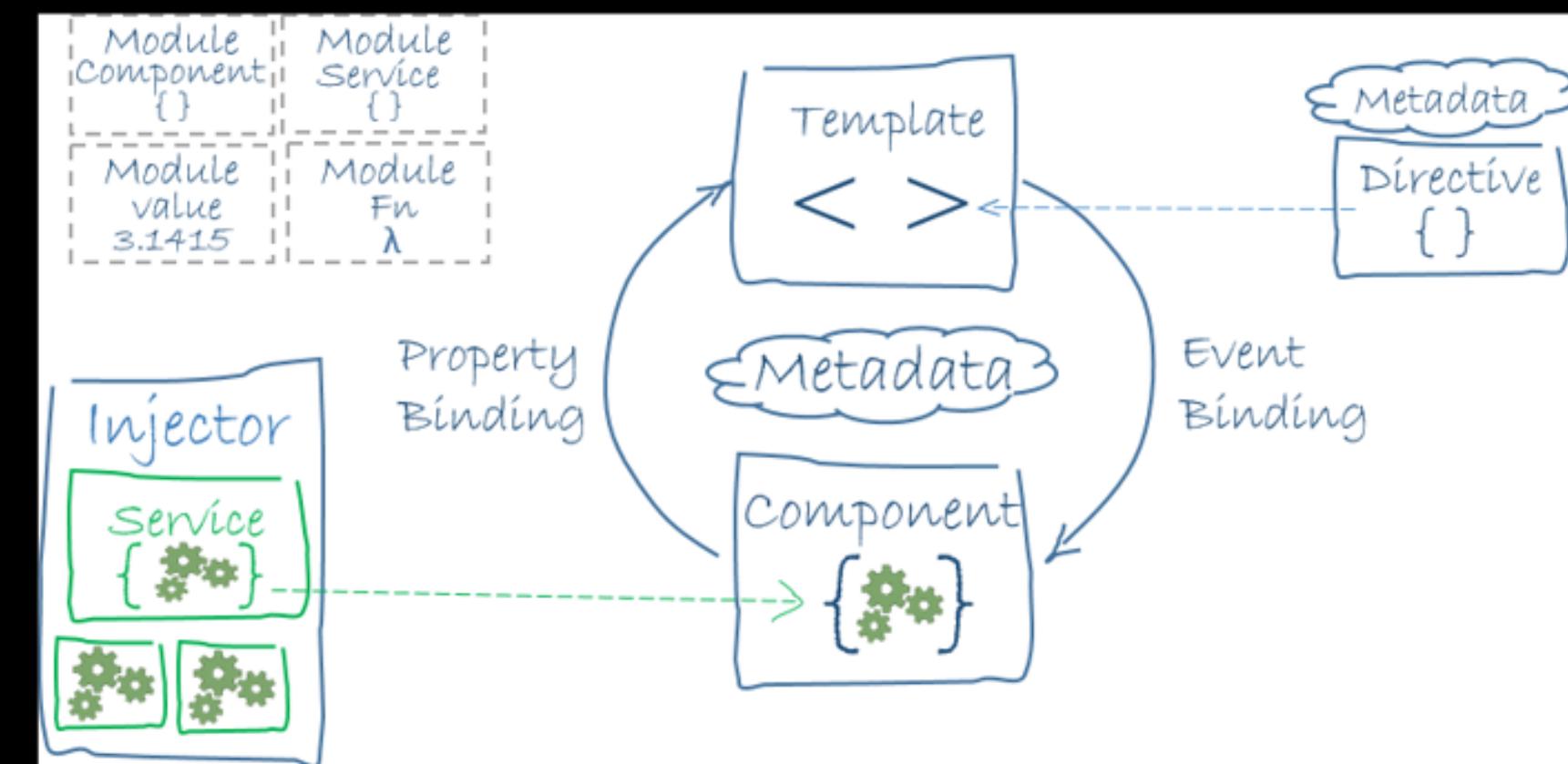
Configuración Typescript

<https://angular.io/guide/typescript-configuration>

Arquitectura Angular

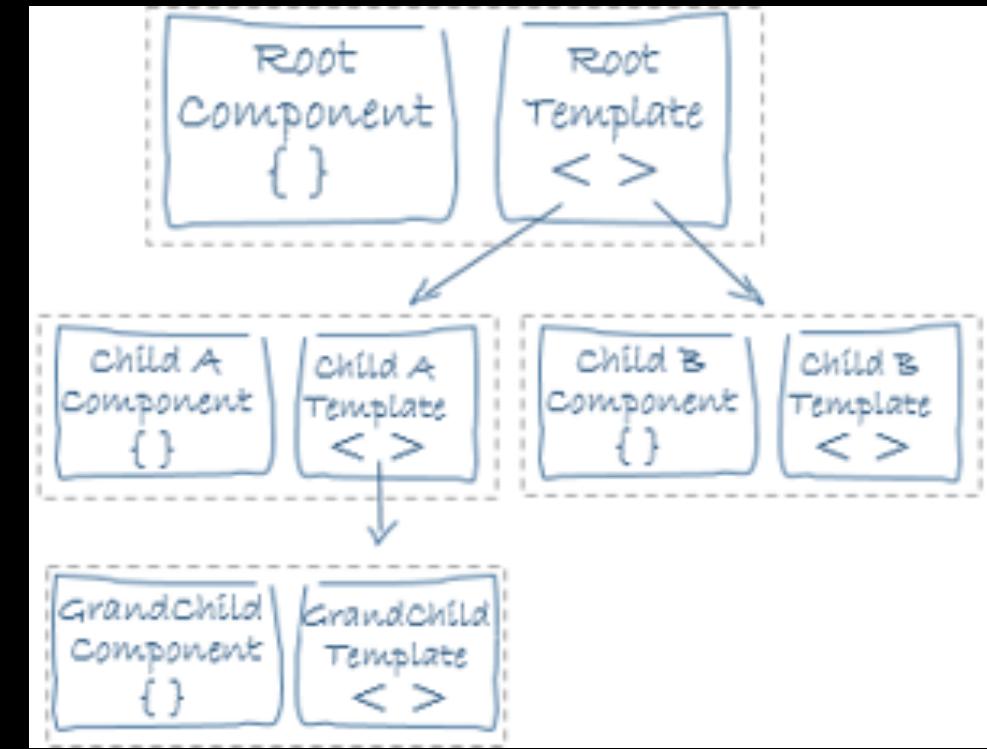
Que es Angular?

- Angular es una plataforma para la creación de aplicaciones en HTML y TypeScript.
- Angular es un ecosistema de librerías que tratan de resolver un problema complejo de una forma un tanto dogmática



Componentes y directivas

- Los componentes son las piezas fundamentales de nuestra aplicación encargadas de la gestión de diferentes partes de nuestra interfaz
- Un componente se encarga de controlar una vista
- Una aplicación es en general un arbol de componentes
- Se declaran mediante el decorator @Component o @Directive
- Una directiva no es mas que un componente sin plantilla
- Tienen un ciclo de vida que nos permite realizar acciones tanto en su creación y attach al DOM, como en su actualización, destrucción y eliminación



```
@Component({ ... })  
class MyComponent() {}
```

```
@Directive({ ... })  
class MyDirective() {}
```

- *selector*: string que permite identificar el componente a compilar y renderizar
- *providers*: Lista de providers para la vista y sus hijos
- *viewProviders*: Lista de providers solo para la vista
- *template* | *templateUrl*: Plantilla inline o externa de un componente. No aplica a directivas
- *styles* | *styleUrls*: Lista de estilos inline o hoja de estilos externa

Módulos

- Conjunto de componentes, servicios, directivas, pipes, etc.
- Los elementos del módulo deberían constituir una unidad funcional en nuestra aplicación
- Organizar una aplicación en módulos fomenta la reusabilidad
- Existirá al menos un módulo para realizar el *bootstrap* de la aplicación (Por convención *AppModule*)

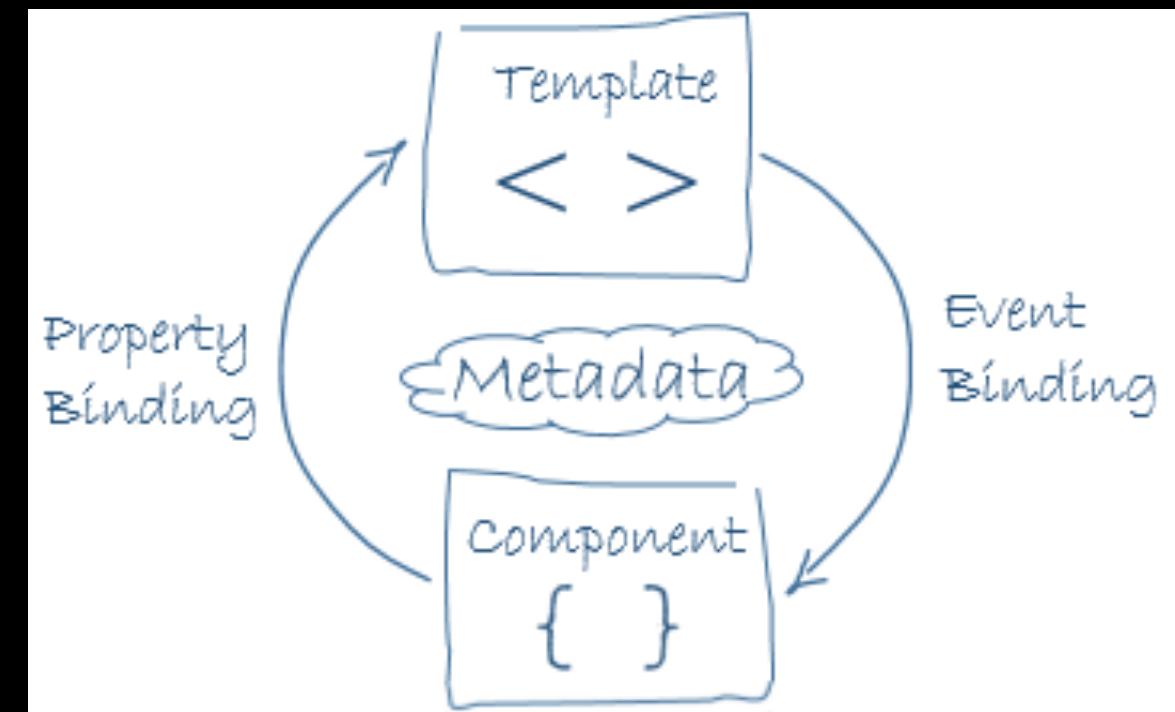
```
@NgModule({ declarations: ..., imports: ...,
exports: ..., providers: ..., bootstrap: ... })
class AppModule {}
```

- *declarations*: Componentes, directivas y pipes que forman parte del módulo
- *exports*: Subconjunto de *declarations* que se exporta para ser usados en otros módulos
- *imports*: Otros módulos utilizados en el *NgModule* que se está declarando. Se añaden a *declarations*
- *providers*: Declaración y definición de la manera de construir *providers* en nuestra aplicación. Se convierten en globales y accesibles desde cualquier parte de la aplicación aunque también se pueden crear a nivel de componente

- *entryComponents*: Lista de componentes no referenciados en ninguna template. Normalmente creados dinámicamente
- *bootstrap*: Componente raiz de nuestra aplicación principal. Sólo el modulo principal debe declarar la propiedad *bootstrap*

Templates

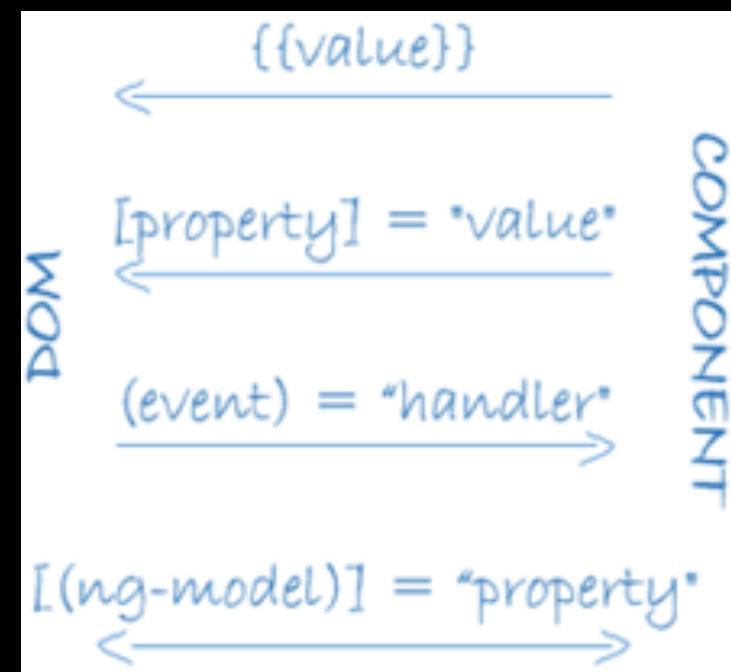
- Combinan HTML tradicional con una sintaxis especial que define Angular. Esta sintaxis permite enlazar los modelos de datos de nuestros componentes con las plantillas
- En las plantillas podemos usar directivas con el objetivo de realizar flujo de control, como si un lenguaje de programación se tratase (if, else, for, switch ...)
- Puedes mostrar datos de la aplicación enlazando controles en la plantilla HTML con propiedades en la clase @Component



Data binding

Data binding es el conjunto de utilidades o técnicas que crean un mecanismo para conectar la aplicación con el HTML

- *Event binding* permite a la aplicación responder a acciones del usuario (click, hover, mousedown ...)
- *Property binding* permite enviar valores calculados durante la ejecución de la aplicación al HTML de las plantillas



Services

Los Services son clases que ofrecen funcionalidad y gestión del estado de la aplicación. Normalmente los usaremos con datos que no pertenecen a ninguna vista concreta sino a la aplicación en global

Inyección de dependencias

- Es el mecanismo que permite injectar servicios en los componentes
- Este mecanismo nos permite injectar por ejemplo la misma instancia de un servicio en diferentes partes de la aplicación (Singleton)

TypeScript

TypeScript

- TypeScript es un superset de Javascript (ES5 y ES6), por tanto podemos utilizar prácticamente todas las posibilidades que nos brinden las nuevas versiones de Javascript y el compilador de Typescript se encargará de convertirlos ficheros al formato que hayamos configurado y que los navegadores actuales comprendan
- Continuamente se actualiza con nuevas funcionalidades que provienen del estándar
- La característica principal que añade TypeScript es la del tipado de datos
- Otras funcionalidades que aporta son los *Decorators*, ampliamente utilizados en Angular

Sintaxis general

```
let variable: type;
```

```
// Realmente aqui no es necesario el tipo ya que lo infiere
const counter: number = 0;
```

```
const name: string = 'Yago';
```

```
const post: Post = new Post();
```

Generics

```
// TypeScript soporta lo que en muchos lenguajes se denomina _Generics_  
  
const posts: Array<Post> = [new Post()];  
  
// Ahora el array sólo puede contener Post  
// En caso contrario el compilador nos lo indicaría con un error
```

any

// any representa cualquier valor

```
let changing: any = 2;
```

```
changing = true; // sin problema
```

Union types

// Nos permite usar varios tipos

```
let changing: number | boolean = 2;
```

```
changing = true; // sin problema
```

Enums

```
enum PostStatus {published, draft, deleted}
```

```
const post = new Post();
```

```
post.status = PostStatus.draft;
```

Return types

```
// Podemos especificar el tipo de retorno de una función  
function getPosts(): Post[] { }
```

```
// Si la función no devuelve nada puedo usar `void`  
function log(): void { }
```

Interfaces

```
// Javascript es dinámico y es una de sus grandes cualidades
function addPointsToScore(player, points) {
    player.score += points;
}
```

// Esta función se puede aplicar a cualquier objeto con una propiedad `score`. Cómo lo traduzco a TypeScript

```
function addPointsToScore(player: { score: number; }, points: number): void {
    player.score += points;
}
```

// Fuerza el parámetro a ser score y el compilador lo detectará

Interfaces

```
interface HasScore {  
    score: number;  
}  
function addPointsToScore(player: HasScore, points: number): void {  
    player.score += points;  
}
```

Interfaces

```
// Puedo definir funciones en la interfaz
interface Runner {
  id: number;
  run(meters: number): void;
}

function startRunning(runner: Runner): void {
  runner.run(20);
}
```

Parámetrosopcionales

```
// En Javascript si un parámetro no existe se convertirá en undefined al invocar a la función  
// Si la función esta tipada entonces el compilador nos avisará del error ...  
// ... pero que pasa si realmente el parámetro es opcional ?  
// Para estos casos tengo el operador '?'  
function addPointsToScore(player: HasScore, points?: number): void {  
    points = points || 0;  
    player.score += points;  
}
```

Classes. Implementación de interfaces

```
// Puedo usar clases ES6. Las clases pueden implementar interfaces  
// como en otros lenguajes de programación
```

```
interface CanRun {  
    run(meters: number): void;  
}  
interface CanEat {  
    eat(): void;  
}  
class HungryRunner implements CanRun, CanEat {  
    run(meters) { }  
    eat() { }  
}
```

Classes. Class properties

```
// Puedo utilizar propiedades de clase. Esto no es una característica  
// de ES6 y sólo lo puedo hacer a través de Typescript  
class Post {  
    maxLines = 25;  
  
    renderLine(numberOfLine: number) {  
        if (numberOfLine < this.maxLines) {  
            ...  
        }  
    }  
}
```

private y public shorcute

```
// En typescript
class Post {
  constructor(public id: number, private content: string) { }
}
```

```
// equivale a
class Post {
  public id: number;
  private content: string;

  constructor(id: number, content: string) {
    this.id = id;
    this.content = content;
  }
}
```

Decorators

- Posiblemente parte del estándar en el futuro
- Ampliamente utilizados por Angular
- Es una forma de añadir metadatos a un target como podría ser un método, una clase ...

Decorators

```
class PostService {
  @Log()
  getPosts() { }

  @Log()
 getPost(id: number) { }
}

const Log = function () {
  return (target: any, name: string, descriptor: any) => {
    logger.log(`call to ${name}`);
    return descriptor;
  };
};
```

Decorators

- *target*: El método objetivo de nuestro decorator
- *name*: El nombre del método objetivo
- *descriptor*: El descriptor del método objetivo (es enumerable, writable, ... ?)

Decorators

```
postService.getPosts();  
// logs: call to getPosts  
postService.getPost(1);  
// logs: call to getPost
```

Decorators

Explorando los diferentes decorators en Angular
Creación de decorators

Instalando types

Por ejemplo para usar Typescript con angular 1.x

```
npm install --save-dev @types/angular
```

DefinitelyTyped

Al instalar un tipo el compilador lo detectará automáticamente si está en node_modules

SCSS

SCSS

- Es un precompilador de CSS
- Nos da una serie de beneficios sobre escribir solamente CSS como el nesting o el uso de variables y mixins
- Parsea el fichero .scss y lo convierte a .css

Variables

SCSS

```
$font-stack: Helvetica, sans-serif;  
$primary-color: #333;
```

```
body {  
  font: 100% $font-stack;  
  color: $primary-color;  
}
```

Variables

CSS

```
body {  
  font: 100% Helvetica, sans-serif;  
  color: #333;  
}
```

Nesting

SCSS

```
nav {  
  ul {  
    margin: 0;  
    padding: 0;  
    list-style: none;  
  }  
  
  li { display: inline-block; }  
  
  a {  
    display: block;  
    padding: 6px 12px;  
    text-decoration: none;  
  }  
}
```

Nesting

CSS

```
nav ul {  
    margin: 0;  
    padding: 0;  
    list-style: none;  
}  
nav li {  
    display: inline-block;  
}  
nav a {  
    display: block;  
    padding: 6px 12px;  
    text-decoration: none;  
}
```

Partials

SCSS

```
// _reset.scss
```

```
html,  
body,  
ul,  
ol {  
    margin: 0;  
    padding: 0;  
}
```

Partials

CSS

// base.scss

```
@import 'reset';
```

```
body {  
  font: 100% Helvetica, sans-serif;  
  background-color: #eefefef;  
}
```

Mixins

```
@mixin transform($property) {  
  transform: $property;  
}  
  
.box { @include transform(rotate(30deg)); }
```

Mixins

```
.box {  
  -webkit-transform: rotate(30deg); // Angular ❤  
  -ms-transform: rotate(30deg); // Angular ❤  
  transform: rotate(30deg);  
}
```

Template Syntax

{{ Interpolation }}

Angular convierte las expresiones en strings y los inserta en el lugar en el que se están declarando

Interpolation es una sintaxis especial que internamente Angular convierte en *property bindings*

```
<p>My name is {{firstName}}</p>
```

```
<h3>
  {{title}}
  changed</span>  
  
<div *ngFor="let hero of heroes"> {{hero.name}} </div>  
  
<input #heroInput> {{heroInput.value}}
```

El scope de las variables esta limitado a la propia plantilla y el componente asociado. No se puede referenciar global scope (window)

Statements

Una declaración o statement responde a un evento lanzado por un binding target (element, component, directive)

La diferencia con las expresiones es que producen efectos colaterales. Una expresión evalúa pero un statement realiza una acción. Así es como modificamos el estado de nuestra aplicación. Respondiendo eventos

```
<button (click)="deleteHero()">Delete hero</button>
```

Statement context

Normalmente es el componente asociado

```
<button (click)="onSave($event)">Save</button>
<button *ngFor="let hero of heroes" (click)="deleteHero(hero)">{{hero.name}}</button>
<form #heroForm (ngSubmit)="onSubmit(heroForm)"> . . . </form>
```

REGLAS DE ORO

- Tanto las expresiones como las declaraciones han de ser simples
- Evitar cálculos complejos o expresiones complejas
- La complejidad se ha de derivar al componente o servicios

Data binding

One way

Data source - Target view

```
{ {expression} }  
[target]="expression"  
bind-target="expression" // No lo usaremos
```

Target view - Data source

```
(target)="statement"  
on-target="statement" // No lo usaremos
```

Two way

```
[target]="expression"  
bindon-target="expression"
```

HTML Attributes vs DOM Properties

- Los atributos inicializan propiedades o características del elemento al que pertenecen. El valor real realmente reside en la propiedad DOM del elemento.
- Algunos nombres de atributos se corresponden con las propiedades (Por ejemplo id) pero no tienen porque.

Un ejemplo claro es el uso de value en un input

```
<input value='My value' />
```

value como atributo inicializa el valor del input pero cuando escribo no se actualiza. La que se actualiza es la propiedad subyacente.

El data binding de Angular funciona a nivel de propiedades y Eventos

[Property binding]

Conocido también como one-way data binding porque solo se propaga desde el componente a la vista

```
<!-- [Propiedad del elemento] = "Propiedad del componente" -->
<img [src] = "heroImageUrl">
<button [disabled] = "isUnchanged">Cancel is disabled</button>
```

[Property binding]

```
<!-- [Directiva]="Propiedad del componente" -->  
<div [ngClass]="classes">binding to the classes property</div>
```

[Property binding]

```
<!-- Comunicación padre – hijo -->  
<app-hero-detail [hero]="currentHero"></app-hero-detail>
```

Si omito los brackets entonces la propiedad no se actualizará. En ocasiones es lo buscado

[Property binding] vs {{ interpolation }}

Normalmente son intercambiables y es una cuestión de preferencia.

```
<p> is the <i>interpolated</i> image.</p>
<p><img [src]="heroImageUrl"> is the <i>property bound</i> image.</p>
```

```
<p><span>"{{title}}</span> is the <i>interpolated</i> title.</p>
<p>"<span [innerHTML]="title"></span>" is the <i>property bound</i> title.</p>
```

Attribute binding (La excepción que confirma la regla)

Deciamos que **El data binding de Angular funciona a nivel de DOM Properties y Eventos**

Hay una excepción y es cuando utilizamos *attribute binding*. Hay ciertos atributos HTML que podemos necesitar enlazar utilizando nuestras clases modelo por ejemplo, atributos de accesibilidad (ARIA), SVGs, o *table spans (colspan)* que no tienen correspondencia con propiedades del elemento en el DOM

Cuando escribimos esto

```
<tr><td colspan="{{1 + 1}}>Three-Four</td></tr>
```

Tenemos como resultado el siguiente error

Template parse errors:

Can't bind to 'colspan' since it isn't a known native property

Lo cual efectivamente cumple nuestra regla de que Angular funciona a nivel de properties. Como lo solucionamos ?

```
<tr><td [attr.colspan]="1 + 1">One-Two</td></tr>
```

Utilizando *attribute binding*!!

También se suele usar para atributos de accesibilidad

```
<button [attr.aria-label]="actionName">{{actionName}} with Aria</button>
```

Class & Style binding

Podemos añadir clases o estilos inline usando [class] y [style] junto a la clase o propiedad que queremos enlazar

[class]

```
<!-- Añade o elimina todas las clases a la vez. 0 todas o ninguna -->
<div class="bad curly special" [class]="badCurly">Bad curly</div>

<!-- Añade o elimina la clase isSpecial en función de una propiedad del modelo -->
<div [class.special]="isSpecial">The class binding is special</div>

<!-- Sobreescritimos la clase con lo que contenga la propiedad del modelo -->
<div class="special" [class.special]="!isSpecial">This one is not so special</div>
```

[style]

```
<button [style.color]="isSpecial ? 'red': 'green'">Red</button>
<button [style.background-color]="canSave ? 'cyan': 'grey'">Save</button>
<button [style.fontSize.em]="isSpecial ? 3 : 1">Big</button>
<button [style.fontSize.%]!="!isSpecial ? 150 : 50">Small</button>
```

(Event binding)

- Los usuarios no se quedan simplemente mirando a la pantalla. El flujo de datos hasta ahora es siempre del componente a la vista
- El binding de eventos nos permite hacer el flujo contrario, de la vista al componente

Eventos nativos

```
<button (click)="onSave($event)">Save</button>
<button on-click="onSave($event)">On Save</button>
```

- El nombre entre () representa el evento (eventos estándar de los elementos DOM) sobre el que queremos añadir el listener.
- onSave() representa el método del componente asociado que queremos ejecutar
- \$event es un objeto normal de un evento DOM
- \$event existirá en el contexto de la plantilla y se puede pasar al método ejecutado

Eventos custom. EventEmitter

- Mediante el uso de EventEmitter puedo lanzar eventos propios de la aplicación
- Es de gran utilidad cuando el componente no necesita conocer la implementación concreta => Reusabilidad
- Es un patrón que se usa de forma extendida en las aplicaciones Angular

Ejemplo: El siguiente componente no sabe como efectuar un borrado, simplemente emite un evento:

```
template: `<div>
  
    {{prefix}} {{hero?.name}}
  </span>
  <button (click)="delete()">Delete</button>
</div>`
```

```
deleteRequest = new EventEmitter<Hero>();
```

```
delete() {
  this.deleteRequest.emit(this.hero);
}
```

Desde el padre, que conoce como efectuar el borrado, nos suscribimos al evento

```
<app-hero-detail  
  (deleteRequest)="deleteHero($event)"  
  [hero]="currentHero"></app-hero-detail>
```

[(Two-way binding)]

- A veces necesitaremos que el binding vaya en ambos sentidos
- Esta técnica es la que se usa en los formularios de Angular

```
@Component({
  selector: 'app-sizer',
  template: `
<div>
  <button (click)="dec()" title="smaller">-</button>
  <button (click)="inc()" title="bigger">+</button>
  <label [style.fontSize.px]="size">FontSize: {{size}}px</label>
</div>`
})
export class SizerComponent {
  @Input() size: number | string;
  @Output() sizeChange = new EventEmitter<number>();

  dec() { this.resize(-1); }
  inc() { this.resize(+1); }

  resize(delta: number) {
    this.size = Math.min(40, Math.max(8, +this.size + delta));
    this.sizeChange.emit(this.size);
  }
}
```

```
<app-sizer  
  [size]="fontSizePx"  
  (sizeChange)="fontSizePx=$event"></app-sizer>
```

Este patrón tiene su sintaxis propia en Angular

```
<app-sizer [(size)]="fontSizePx"></app-sizer>
```

```
<div [style.fontSize.px]="fontSizePx">Resizable Text</div>
```

Attribute & Structural Directives

- **Attribute**: Modifican el comportamiento de los elementos HTML y se aplican como atributos
- **Structural**: Se responsabilizan de controlar el DOM añadiendo o eliminando nodos

Built-in attribute directives

Las directivas de atributo van ligadas y modifican el comportamiento de los elementos HTML. Hay varias predefinidas

NgClass

Directiva avanzada para el control del atributo class. Podemos añadir o eliminar varias clases a la vez, cosa que no podríamos hacer a través de property binding con class

```
<div [ngClass]="currentClasses"></div>
```

```
currentClasses: {};
setCurrentClasses() {
  // CSS classes: added/removed per current state of component properties
  this.currentClasses = {
    'saveable': this.canSave,
    'modified': !this.isUnchanged,
    'special': this.isSpecial
  };
}
```

NgStyle

Similar a NgClass pero para los estilos inline

```
<div [ngStyle]="currentStyles"></div>
```

```
currentStyles: {};  
setCurrentStyles() {  
    // CSS styles: set per current state of component properties  
    this.currentStyles = {  
        'font-style': this.canSave ? 'italic' : 'normal',  
        'font-weight': !this.isUnchanged ? 'bold' : 'normal',  
        'font-size': this.isSpecial ? '24px' : '12px'  
    };  
}
```

NgModel (Two-way data binding)

Directiva especializada para controlar elementos de un formulario

```
<input [(ngModel)]="currentHero.name">
```

En realidad ngModel es simplemente un shorthand de un patrón que sería bastante habitual en Angular:

```
<input  
  [ngModel]="currentHero.name"  
  (ngModelChange)="currentHero.name=$event">
```

que a su vez oculta el funcionamiento real del elemento input que seria:

```
<input  
  [value]="currentHero.name"  
  (input)="currentHero.name=$event.target.value" >
```

El shorthand `[(ngModel)]` solo puede realizar el binding con una propiedad. Si se necesita algo más avanzado se puede recurrir a la forma extendida

```
<input  
  [ngModel]="currentHero.name"  
  (ngModelChange)="setUppercaseName($event)">
```

Built-in structural directives

Este tipo de directivas permiten modificar el DOM añadiendo, eliminando o modificando elementos

NgIf

Permite añadir o eliminar un elemento del DOM a través de la evaluación boolena de una propiedad o expresión

```
<app-hero-detail *ngIf="isActive"></app-hero-detail>
```

NgForOf

Esta directiva permite la repetición de un elemento DOM a través de una sintaxis propia que Angular puede interpretar y que se conoce como *microsyntax*

```
<div *ngFor="let hero of heroes; let i=index; trackBy: hero.id">
  {{i + 1}} - {{hero.name}}
</div>
```

- let crea una variable local en el contexto de la directiva
- index es un indice que comienza en cero y representa el elemento del array que se esta renderizando
- trackBy permite optimizar el rendimiento de la directiva al enlazar una nueva colección (Por ejemplo después de una llamada AJAX)

NgSwitch

Es un conjunto de directivas que emulan el comportamiento de un bloque switch

```
<div [ngSwitch]="currentHero.emotion">
  <app-happy-hero *ngSwitchCase="'happy'" [hero]="currentHero"></app-happy-hero>
  <app-sad-hero *ngSwitchCase="'sad'" [hero]="currentHero"></app-sad-hero>
  <app-confused-hero *ngSwitchCase="'confused'" [hero]="currentHero"></app-confused-hero>
  <app-unknown-hero *ngSwitchDefault [hero]="currentHero"></app-unknown-hero>
</div>
```

Sólo se mostrará el bloque que cumpla la condición. Nótese que ngSwitch es una directiva de atributo (Sin *)

Variables en plantillas

Se pueden definir variables locales en plantillas mediante #. La variable se puede referenciar en cualquier parte de la plantilla

```
<input #phone placeholder="phone number">
```

...

```
<button (click)="callPhone(phone.value)">Call</button>
```

Resumen

```
@Component({
  selector: 'app-little-tour',
  template: `
    <input #newHero
      (keyup.enter)="addHero(newHero.value)"
      (blur)="addHero(newHero.value); newHero.value=' '">

    <button (click)="addHero(newHero.value)">Add</button>

    <ul><li *ngFor="let hero of heroes">{{hero}}</li></ul>
  `
})
export class LittleTourComponent {
  heroes = ['Windstorm', 'Bombasto', 'Magneta', 'Tornado'];
  addHero(newHero: string) {
    if (newHero) {
      this.heroes.push(newHero);
    }
  }
}
```

Ciclo de vida de un componente

Los componentes y directivas en Angular tienen un ciclo de vida en el cual se crean, se modifican, se destruyen.

A través de una serie de métodos podremos engancharnos en dichos momentos para realizar las acciones que necesitemos.

```
export class MyComponent implements OnInit {  
  constructor() { }  
  
  ngOnInit() { }  
}
```

Métodos del ciclo de vida

- ***ngOnChanges()*** Se llama cada vez que las propiedades de entrada del componente (`@Input`) se modifican. Se llama antes de `ngOnInit()`.
- ***ngOnInit()*** Se llama sólo una vez después del primer `ngOnChanges()` una vez que el componente establece por primera vez sus propiedades de entrada y se ha inicializado.

Debería utilizarse para inicializaciones complejas o para realizar un primer setup del componente. Debe evitarse este tipo de complejidad en el constructor del componente ya que estamos ralentizando la instanciación innecesariamente. Las llamadas HTTP deberían ir en este método

Métodos del ciclo de vida

- ***ngDoCheck()*** Se lanza después de *ngOnChanges()* y *ngOnInit()*. Se llama cada vez que se actualiza alguna parte de la aplicación. Se llama con mucha frecuencia. **⚠ DEBE UTILIZARSE CON PRECAUCIÓN!!**
- ***ngAfterContentInit()*** Se ejecuta una vez que Angular proyecta contenido en la vista del componente. Se llama sólo una vez después de *ngDoCheck()*
- ***ngAfterContentChecked()*** Se ejecuta cada vez que el contenido del componente ha sido verificado por el mecanismo de detección de cambios de Angular, se llama después del método *ngAfterContentInit()* y en cada invocación a *ngDoCheck()*. **⚠ DEBE UTILIZARSE CON PRECAUCIÓN!!**

Métodos del ciclo de vida

- ***ngAfterViewInit()*** Se ejecuta una vez que la plantilla se ha instanciado e inicializado por completo. Se invoca después de *ngAfterContentChecked()*. Sólo aplica a componentes y no a directivas (no tienen vista). Se ejecuta sólo una vez. Es un buen sitio para inicializaciones que dependan de la vista (P.Ej un plugin sobre un elemento DOM)
- ***ngAfterViewChecked()*** Se ejecuta después de *ngAfterViewInit()* y cada vez que el componente verifique cambios (Incluso de otros componentes de los que es padre) **⚠ DEBE UTILIZARSE CON PRECAUCIÓN!!**
- ***ngOnDestroy()*** Se ejecuta cada vez que Angular destruye el componente. Útil para darse de baja de observables, eventos, liberar recursos, etc.

Demo

[https://stackblitz.com/angular/roqnqkkpdrl?
file=src%2Fapp%2Fspy.component.ts](https://stackblitz.com/angular/roqnqkkpdrl?file=src%2Fapp%2Fspy.component.ts)

Interacción entre componentes

@Input

- Nos permite enviar datos de un componente padre a su hijo
- Los datos del padre se envían al hijo mediante *property binding*

Uso de @Input

Componente hijo

```
@Component({  
  selector: 'app-hero-child',  
  template: `  
    <h3>{{hero.name}} says:</h3>  
    <p>I, {{hero.name}}, am at your service, {{masterName}}.</p>  
  `,  
})  
export class HeroChildComponent {  
  @Input() hero: Hero;  
  @Input('master') masterName: string;  
}
```

Uso de @Input

Componente padre

```
@Component({
  selector: 'app-hero-parent',
  template: `
    <h2>{{master}} controls {{heroes.length}} heroes</h2>
    <app-hero-child *ngFor="let hero of heroes"
      [hero]="hero"
      [master]="master">
    </app-hero-child>
  `
})
export class HeroParentComponent {
  heroes = HEROES;
  master = 'Master';
}
```

Interceptar @Input con set

Para observar cambios que afectan sólo a una propiedad

```
@Component({
  selector: 'app-name-child',
  template: '<h3>{{name}}</h3>'
})
export class NameChildComponent {
  private _name = '';

  @Input()
  set name(name: string) {
    this._name = (name && name.trim()) || '<no name set>';
  }

  get name(): string { return this._name; }
}

<app-name-child *ngFor="let name of names" [name]="name"></app-name-child>
```

Interceptar @Input con ngOnChanges

Para observar cambios que afectan a múltiples propiedades

```
@Component({
  selector: 'app-version-child',
  template: `
    <h3>Version {{major}}.{{minor}}</h3>
    <h4>Change log:</h4>
    <ul>
      <li *ngFor="let change of changeLog">{{change}}</li>
    </ul>
  `
})
export class VersionChildComponent implements OnChanges {
  @Input() major: number;
  @Input() minor: number;
  changeLog: string[] = [];

  ngOnChanges(changes: {[propKey: string]: SimpleChange}) {
    let log: string[] = [];
    for (let propName in changes) {
      let changedProp = changes[propName];
      let to = JSON.stringify(changedProp.currentValue);
      if (changedProp.isFirstChange()) {
        log.push(`Initial value of ${propName} set to ${to}`);
      } else {
        let from = JSON.stringify(changedProp.previousValue);
        log.push(`${propName} changed from ${from} to ${to}`);
      }
    }
    this.changeLog.push(log.join(', '));
  }
}

<app-version-child [major]="major" [minor]="minor"></app-version-child>
```

@Output

- Nos permite enviar datos de un component a su padre mediante EventEmitter
- Será el API pública del componente

El padre escucha por eventos que ocurran en su hijo

```
@Component( {  
  selector: 'app-voter',  
  template: `  
    <h4>{{name}}</h4>  
    <button (click)="vote(true)" [disabled]="didVote">Agree</button>  
    <button (click)="vote(false)" [disabled]="didVote">Disagree</button>  
  `  
})  
export class VoterComponent {  
  @Input() name: string;  
  @Output() voted = new EventEmitter<boolean>();  
  didVote = false;  
  
  vote(agreed: boolean) {  
    this.voted.emit(agreed);  
    this.didVote = true;  
  }  
}
```

El padre escucha por eventos que ocurrán en su hijo

```
@Component({  
  selector: 'app-vote-taker',  
  template: `  
    <h2>Should mankind colonize the Universe?</h2>  
    <h3>Agree: {{agreed}}, Disagree: {{disagreed}}</h3>  
    <app-voter *ngFor="let voter of voters"  
      [name]="voter"  
      (voted)="onVoted($event)">  
    </app-voter>  
  `,  
})  
export class VoteTakerComponent {  
  agreed = 0;  
  disagreed = 0;  
  voters = ['Mr. IQ', 'Ms. Universe', 'Bombasto'];  
  
  onVoted(agreed: boolean) {  
    agreed ? this.agreed++ : this.disagreed++;  
  }  
}
```

El padre interactúa con el hijo a través de una variable

```
@Component({
  selector: 'app-countdown-timer',
  template: '<p>{{message}}</p>'
})
export class CountdownTimerComponent implements OnInit, OnDestroy {

  intervalId = 0;
  message = '';
  seconds = 11;

  clearTimer() { clearInterval(this.intervalId); }

  ngOnInit() { this.start(); }
  ngOnDestroy() { this.clearTimer(); }

  start() { this.countDown(); }
  stop() {
    this.clearTimer();
    this.message = `Holding at T-${this.seconds} seconds`;
  }

  private countDown() {
    this.clearTimer();
    this.intervalId = window.setInterval(() => {
      this.seconds -= 1;
      if (this.seconds === 0) {
        this.message = 'Blast off!';
      } else {
        if (this.seconds < 0) { this.seconds = 10; } // reset
        this.message = `T-${this.seconds} seconds and counting`;
      }
    }, 1000);
  }
}
```

El padre interactúa con el hijo a través de una variable

```
@Component( {  
    selector: 'app-countdown-parent-lv',  
    template: `<h3>Countdown to Liftoff (via local variable)</h3>  
    <button (click)="timer.start()">Start</button>  
    <button (click)="timer.stop()">Stop</button>  
    <div class="seconds">{{timer.seconds}}</div>  
    <app-countdown-timer #timer></app-countdown-timer>  
    `,  
    styleUrls: [ './assets/demo.css' ]  
})  
export class CountdownLocalVarParentComponent { }
```

Interacción a través de @ViewChild

Componente padre

```
@Component({
  selector: 'app-countdown-parent-vc',
  template: `
    <h3>Countdown to Liftoff (via ViewChild)</h3>
    <button (click)="start()">Start</button>
    <button (click)="stop()">Stop</button>
    <div class="seconds">{{ seconds() }}</div>
    <app-countdown-timer></app-countdown-timer>
  `,
  styleUrls: ['./assets/demo.css']
})
export class CountdownViewChildParentComponent implements AfterViewInit {

  @ViewChild(CountdownTimerComponent)
  private timerComponent: CountdownTimerComponent;

  seconds() { return 0; }

  ngAfterViewInit() {
    // Redefine `seconds()` to get from the `CountdownTimerComponent.seconds` ...
    // but wait a tick first to avoid one-time devMode
    // unidirectional-data-flow-violation error
    setTimeout(() => this.seconds = () => this.timerComponent.seconds, 0);
  }

  start() { this.timerComponent.start(); }
  stop() { this.timerComponent.stop(); }
}
```

Comunicación a través de un Service

The service

```
@Injectable()
export class MissionService {

    // Observable string sources
    private missionAnnouncedSource = new Subject<string>();
    private missionConfirmedSource = new Subject<string>();

    // Observable string streams
    missionAnnounced$ = this.missionAnnouncedSource.asObservable();
    missionConfirmed$ = this.missionConfirmedSource.asObservable();

    // Service message commands
    announceMission(mission: string) {
        this.missionAnnouncedSource.next(mission);
    }

    confirmMission(astronaut: string) {
        this.missionConfirmedSource.next(astronaut);
    }
}
```

Comunicación a través de un Service

Component 1

```
@Component({
  selector: 'app-mission-control',
  template: `
    <h2>Mission Control</h2>
    <button (click)="announce()">Announce mission</button>
    <app-astronaut *ngFor="let astronaut of astronauts"
      [astronaut]="astronaut">
    </app-astronaut>
    <h3>History</h3>
    <ul>
      <li *ngFor="let event of history">{{event}}</li>
    </ul>
  `,
  providers: [MissionService]
})
export class MissionControlComponent {
  astronauts = ['Lovell', 'Swigert', 'Haise'];
  history: string[] = [];
  missions = ['Fly to the moon!',
    'Fly to mars!',
    'Fly to Vegas!'];
  nextMission = 0;

  constructor(private missionService: MissionService) {
    missionService.missionConfirmed$.subscribe(
      astronaut => {
        this.history.push(` ${astronaut} confirmed the mission`);
      });
  }

  announce() {
    let mission = this.missions[this.nextMission++];
    this.missionService.announceMission(mission);
    this.history.push(`Mission "${mission}" announced`);
    if (this.nextMission >= this.missions.length) { this.nextMission = 0; }
  }
}
```

Comunicación a través de un Service

Component 2

```
@Component({
  selector: 'app-astronaut',
  template: `
    <p>
      {{astronaut}}: <strong>{{mission}}</strong>
      <button
        (click)="confirm()"
        [disabled]="!announced || confirmed"
        Confirm
      </button>
    </p>
  `
})
export class AstronautComponent implements OnDestroy {
  @Input() astronaut: string;
  mission = '<no mission announced>';
  confirmed = false;
  announced = false;
  subscription: Subscription;

  constructor(private missionService: MissionService) {
    this.subscription = missionService.missionAnnounced$.subscribe(
      mission => {
        this.mission = mission;
        this.announced = true;
        this.confirmed = false;
      });
  }

  confirm() {
    this.confirmed = true;
    this.missionService.confirmMission(this.astronaut);
  }

  ngOnDestroy() {
    // prevent memory leak when component destroyed
    this.subscription.unsubscribe();
  }
}
```

Attribute Directives

Attribute Directives

Se encargan de modificar la apariencia o comportamiento de un elemento, componente o otra directiva

@Directive

Se definen usando el decorator @Directive

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

@Directive( {
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor() { }
}
```

ElementRef

Podemos injectar ElementRef para permitir acceso al elemento *host* a través de su propiedad *nativeElement*

```
import { Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor(el: ElementRef) {
    el.nativeElement.style.backgroundColor = 'yellow';
  }
}
```

HostListener

Con HostListener podemos suscribirnos a eventos DOM

```
@HostListener('mouseenter') onMouseEnter() {  
  this.highlight('yellow');  
}  
  
```

```
@HostListener('mouseleave') onMouseLeave() {  
  this.highlight(null);  
}  
  
```

```
private highlight(color: string) {  
  this.el.nativeElement.style.backgroundColor = color;  
}
```

@Input

Podemos usar @Input para enviar valores a la Directiva como venimos haciendo habitualmente

```
<p [appHighlight]="color">Highlight me! </p>

@Input('appHighlight') highlightColor: string;
@HostListener('mouseenter') onMouseEnter() {
  this.highlight(this.highlightColor || 'red');
}
```

@HostBinding

Puedo utilizar HostBinding para enlazar con atributos del elemento contenedor

```
@HostBinding('class.active') private ishovering: boolean;
```

Structural Directives

Structural Directives

- Son responsables de añadir, eliminar o modificar elementos del DOM
- Se pueden aplicar a cualquier *host element*

Eliminar vs ocultar

Otra forma de NO mostrar un elemento DOM sería

```
<p [style.display]="'none'">  
  Expression sets display to "none".  
  This paragraph is hidden but still in the DOM.  
</p>
```

Pero aunque el resultado visual sea el mismo las diferencias son mayores

- Cuando utilizo una directiva como `ngIf` se eliminan listeners y todos los elementos y clases asociados. Al hacer el attach de nuevo, comienza el ciclo de vida del componente
- Cuando utilizo `style` se mantiene el estado con todas las consecuencias. Puede que sea lo buscado en algún caso



- Las directivas estructurales tienen una sintaxis particular utilizando un asterisco *
- En compilación esto se traduce a un tag *ng-template* alrededor del *host*

Nglf

```
<div *ngIf="hero" class="name">{{hero.name}}</div>
```

```
<ng-template [ngIf]="hero">
  <div class="name">{{hero.name}}</div>
</ng-template>
```

NgFor

```
<div *ngFor="let hero of heroes; let i=index; let odd=odd; trackBy: trackById" [class.odd]="odd">
  ({{i}}) {{hero.name}}
</div>

<ng-template ngFor let-hero [ngForOf]="heroes" let-i="index" let-odd="odd" [ngForTrackBy]="trackById">
  <div [class.odd]="odd">({{i}}) {{hero.name}}</div>
</ng-template>
```

NgSwitch

```
<div [ngSwitch]="hero?.emotion">
  <app-happy-hero *ngSwitchCase="'happy'" [hero]="hero"></app-happy-hero>
  <app-sad-hero *ngSwitchCase="'sad'" [hero]="hero"></app-sad-hero>
  <app-confused-hero *ngSwitchCase="'app-confused'" [hero]="hero"></app-confused-hero>
  <app-unknown-hero *ngSwitchDefault [hero]="hero"></app-unknown-hero>
</div>
```

```
<div [ngSwitch]="hero?.emotion">
  <ng-template [ngSwitchCase]="'happy'">
    <app-happy-hero [hero]="hero"></app-happy-hero>
  </ng-template>
  <ng-template [ngSwitchCase]="'sad'">
    <app-sad-hero [hero]="hero"></app-sad-hero>
  </ng-template>
  <ng-template [ngSwitchCase]="'confused'">
    <app-confused-hero [hero]="hero"></app-confused-hero>
  </ng-template>
  <ng-template ngSwitchDefault>
    <app-unknown-hero [hero]="hero"></app-unknown-hero>
  </ng-template>
</div>
```

<ng-template />

- Es un elemento del framework para mostrar HTML
- Nunca se visualiza en el DOM, es un elemento Angular que desaparece tras la compilación
- Se sustituye por un comentario en la fase de render

<ng-container />

- Se puede usar ng-container en caso de no disponer de un elemento host
- Reducimos riesgo de recibir estilos globales
- Sólo se puede introducir una directiva estructural por elemento por lo que una buena forma de esquivar esta restricción es mediante el anidado de varios ng-container

Ejemplo

```
@Directive({
  selector: '[appUnless]'
})
export class UnlessDirective {
  private hasView = false;

  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef) { }

  @Input() set appUnless(condition: boolean) {
    if (!condition && !this.hasView) {
      this.viewContainer.createEmbeddedView(this.templateRef);
      this.hasView = true;
    } else if (condition && this.hasView) {
      this.viewContainer.clear();
      this.hasView = false;
    }
  }
}
```

Estilos en componentes

Dando estilos a nuestros componentes

- Las aplicaciones Angular pueden recibir estilos usando CSS. También podremos utilizar precompiladores como SASS de forma sencilla
- Los estilos se aplican de forma modular, es decir, el scope es el componente y las clases y estilos definidos no afectan al resto de la aplicación. Asimismo los estilos definidos en otros componentes tampoco afectarán al que estemos implementando

Selectores especiales

:host

El pseudoselector `:host` permite hacer referencia y dar estilos al elemento que se encarga de definir el componente para su renderizado. El elemento definido en la propiedad `selector`

```
:host {  
  display: block;  
  border: 1px solid black;  
}
```

```
:host(.active) {  
  border-width: 3px;  
}
```

Selectores especiales

:host-context

Permite hacer algo que con CSS estándar no podemos.
Seleccionar un elemento padre.

```
:host-context(.theme-light) h2 {  
    background-color: #eef;  
}
```

Carga de estilos en un componente

- A través de *styles* o *styleUrls*
- Inline en la *template* del componente
- CSS imports

Propiedad *styles*

```
@Component( {  
    selector: 'app-root' ,  
    template: `  
        <h1>Tour of Heroes</h1>  
        <app-hero-main [hero]="hero"></app-hero-main>  
    `,  
    styles: [ 'h1 { font-weight: normal; }' ]  
})  
export class HeroAppComponent {  
/* . . . */  
}
```

Propiedad *styleUrls*

```
@Component( {  
    selector: 'app-root' ,  
    template: `  
        <h1>Tour of Heroes</h1>  
        <app-hero-main [hero]="hero"></app-hero-main>  
    `,  
    styleUrls: [ './hero-app.component.css' ]  
})  
export class HeroAppComponent {  
/* . . . */  
}
```

Inline en la template

```
@Component({  
  selector: 'app-hero-controls',  
  template: `<style>  
    button {  
      background-color: white;  
      border: 1px solid #777;  
    }  
  </style>  
  <link rel="stylesheet" href="../../assets/hero-team.component.css">  
  <h3>Controls</h3>  
  <button (click)="activate()">Activate</button>  
`  
})
```

CSS @imports

```
@import './hero-details-box.css';
```

Definiendo el tipo de modularidad - *viewEncapsulation*

- **ShadowDom** - Implementación nativa de ShadowDom. Soporte limitado
- **Native** - Implementación nativa también pero de un estándar anterior
- **Emulated** - Por defecto si no se especifica otra. Simula ShadowDom y por tanto soporta todos los navegadores
- **None** - Los estilos se aplicarán de forma global

CSS generado

```
<hero-details _nghost-pmm-5>
  <h2 _ngcontent-pmm-5>Mister Fantastic</h2>
  <hero-team _ngcontent-pmm-5 _nghost-pmm-6>
    <h3 _ngcontent-pmm-6>Team</h3>
  </hero-team>
</hero-detail>
```

Pipes

Qué son?

- Una Pipe es simplemente una función que recibe datos como input y a través de una transformación produce una salida
- En Angular tenemos una serie de Pipes disponibles a través del framework pero también podemos crearlas nosotros

Angular Pipes

- DatePipe
- UpperCasePipe
- LowerCasePipe
- CurrencyPipe
- PercentPipe
- JsonPipe
- AsyncPipe

Parámetros

Un Pipe puede recibir cualquier número de parámetros optionales

```
<p>The hero's birthday is {{ birthday | date:'MM/dd/yy' }} </p>
```

El parámetro puede ser cualquier expresión válida de las que usamos en nuestras templates

template: `

```
<p>The hero's birthday is {{ birthday | date:format }}</p>
<button (click)="toggleFormat()">Toggle Format</button>
```

Encadenando Pipes

Podemos encadenar Pipes. El valor de salida de cada una será el de entrada de la siguiente

```
{ { birthday | date: 'fullDate' | uppercase} }
```

Creación de Pipes

```
@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements PipeTransform {
  transform(value: number, exponent: string): number {
    let exp = parseFloat(exponent);
    return Math.pow(value, isNaN(exp) ? 1 : exp);
  }
}
```

A tener en cuenta

- Puedes usar las Pipes creadas de la misma forma que usarías una propia del framework
- Las Pipes creadas se deben declarar en el módulo al que pertenezcan en el array de la propiedad *declarations*

Pure vs Impure

El mecanismo de detección de cambios en las Pipes es un tanto especial

- **Pure**: Sólo se actualiza la vista si se actualizan los valores de una primitiva o se modifica la referencia a un objeto
- **Impure**: Se actualiza con más frecuencia cada vez que se lanza un ciclo de detección en el framework, es decir, después de una pulsación de una tecla, movimiento del ratón, evento asíncrono, timeouts

Impure

```
@Pipe({  
  name: 'flyingHeroesImpure',  
  pure: false  
})  
export class FlyingHeroesImpurePipe extends FlyingHeroesPipe {}
```

Pure (Por defecto)

```
@Pipe({ name: 'flyingHeroes' })
export class FlyingHeroesPipe implements PipeTransform {
  transform(allHeroes: Flyer[]) {
    return allHeroes.filter(hero => hero.canFly);
  }
}
```

Las función transform de las Pipes puras ha de ser también pura, es decir, debe retornar un nuevo valor o referencia

AsyncPipe

- Es una Pipe importante dentro del framework. Se trata de una Pipe que es capaz de mantener su propio estado
- Permite suscribirnos a Promises o Observables y cada vez que se produce una actualización en los mismos se reflejará en la vista

Forms

Métodos de creación de formularios

- **Reactive**: Robustos, escalables, reusables y testables. Si la aplicación esta basada en formularios es una buena idea usarlos
- **Template-driven**: Fáciles de usar y comunes en aplicaciones que no contienen un gran número de formularios
- Con ambos tipos de formularios podemos hacer lo mismo

Comparación

	REACTIVE	TEMPLATE-DRIVEN
Configuración	Explícita en clases	Directivas en plantillas
Modelo de datos	Estructurado	Desestructurado
Previsibilidad	Síncrono	Asíncrono
Validaciones	Funciones	Directivas
Mutabilidad	Inmutable	Mutable
Escalabilidad	API Low-Level	Abstracciones High-Level

Reactive Forms

Que son?

- Se basan en la creación de modelos de datos en clases
- Inmutables. Cada cambio en el estado de un form devuelve un nuevo estado => Eficiencia
- La implementación esta basada en Observables (Pueden accederse de forma síncrona)
- Facilitan el testing al estar basados en modelos de datos
- Lo que programo es lo que va a pasar y por tanto puedo depurarlo. Tengo el control

ReactiveFormsModule

El módulo que nos permite el uso de los formularios reactivos. Se registra de la siguiente forma:

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

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

FormControl. Registro en componente

FormControl permite el registro de controles en nuestros componentes

```
@Component({  
  selector: 'app-name-editor',  
  templateUrl: './name-editor.component.html',  
  styleUrls: ['./name-editor.component.css']  
})  
export class NameEditorComponent {  
  name = new FormControl('');  
}
```

FormControl. Registro en template

Luego los asociaremos con las plantillas

```
<label>  
  Name:  
    <input type="text" [formControl]="name">  
</label>
```

FormControl. Acceso a valores

Podemos acceder a los valores subyacentes

- A través del Observable *valueChanges*
- A través de la propiedad *value*

```
<p>  
  Value: {{ name.value }}  
</p>
```

FormControl. Modificando de valores

- A través del método `setValue()`. Modifica el valor y lo valida contra la estructura definida en el FormControl

```
updateName() {  
  this.name.setValue('Nancy');  
}  
  
<p>  
  <button (click)="updateName()">Update Name</button>  
</p>
```

FormGroup

- FormControl permite realizar tracking sobre un elemento input
- FormGroup permite controlar el estado de un grupo de varios FormControl

FormGroup. Creación del modelo

```
@Component({  
  selector: 'app-profile-editor',  
  templateUrl: './profile-editor.component.html',  
  styleUrls: ['./profile-editor.component.css']  
})  
export class ProfileEditorComponent {  
  profileForm = new FormGroup({  
    firstName: new FormControl(''),  
    lastName: new FormControl(''),  
  });  
}
```

FormGroup. Asociación con template

```
<form [formGroup]="profileForm">
  <label>
    First Name:
    <input type="text" formControlName="firstName">
  </label>

  <label>
    Last Name:
    <input type="text" formControlName="lastName">
  </label>
</form>
```

FormGroup. Submit

```
<form [formGroup]="profileForm" (ngSubmit)="onSubmit()">  
  onSubmit() {  
    console.warn(this.profileForm.value);  
  }  
  
<button type="submit">Submit</button>
```

FormGroup. Nesting en el componente

```
@Component({  
  selector: 'app-profile-editor',  
  templateUrl: './profile-editor.component.html',  
  styleUrls: ['./profile-editor.component.css']  
})  
export class ProfileEditorComponent {  
  profileForm = new FormGroup({  
    firstName: new FormControl(''),  
    lastName: new FormControl(''),  
    address: new FormGroup({  
      street: new FormControl(''),  
      city: new FormControl(''),  
      state: new FormControl(''),  
      zip: new FormControl('')  
    })  
  });  
}
```

FormGroup. Nesting en template

```
<form [formGroup]="profileForm">
  ...
  <div formGroupName="address">
    <h3>Address</h3>

    <label>
      Street:
      <input type="text" formControlName="street">
    </label>

    <label>
      City:
      <input type="text" formControlName="city">
    </label>

    <label>
      State:
      <input type="text" formControlName="state">
    </label>

    <label>
      Zip Code:
      <input type="text" formControlName="zip">
    </label>
  </div>
</form>
```

Actualizaciones parciales con patchValue

- *setValue()* permite reemplazar el valor de un FormControl completamente
- *patchValue()* permite actualizar varios FormControl a la vez sin tener que especificar todos

Ejemplo con patchValue

```
updateUserProfile() {  
  this.profileForm.patchValue({  
    firstName: 'Nancy',  
    address: {  
      street: '123 Drew Street'  
    }  
  });  
}
```

FormBuilder

El proceso de creación de un formulario usando FormGroup
puede resultar repetitivo y monótono
Angular proporciona un método más declarativo y fácil de usar
que básicamente hace lo mismo

FormBuilder. Declaración

```
@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = this.fb.group({
    firstName: ['', ],
    lastName: ['', ],
    address: this.fb.group({
      street: ['', ],
      city: ['', ],
      state: ['', ],
      zip: ['', ]
    }),
  });
}

constructor(private fb: FormBuilder) { }
```

FormArray

FormArray permite añadir *FormControl* de forma dinámica a nuestros formularios

FormArray. Ejemplo

```
profileForm = this.fb.group({  
    firstName: ['', Validators.required],  
    lastName: ['',  
    address: this.fb.group({  
        street: ['',  
        city: ['',  
        state: ['',  
        zip: ['',  
    }),  
    aliases: this.fb.array([  
        this.fb.control('')  
    ])  
});
```

FormArray. Uso en clase

Accediendo a los alias

```
get aliases() {  
    return this.profileForm.get('aliases') as FormArray  
}
```

FormArray. Añadiendo nuevos alias

```
addAlias() {  
    this.aliases.push(this.fb.control(''));  
}
```

FormArray. Uso en template

```
<div formArrayName="aliases">
  <h3>Aliases</h3> <button (click)="addAlias()">Add Alias</button>

  <div *ngFor="let address of aliases.controls; let i=index">
    <!-- The repeated alias template -->
    <label>
      Alias:
      <input type="text" [formControlName]="i">
    </label>
  </div>
</div>
```

Template-driven Forms

Qué son?

- Se puede crear prácticamente cualquier formulario con una plantilla
- El flujo del formulario y control de errores lo controlamos directamente en la plantilla mediante el uso de directivas, data-binding y variables
- Menos flexibles y desde luego no muy reusables pero fáciles de crear

FormsModule. Registro

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

@NgModule({
  imports: [
    // other imports ...
    FormsModule
  ],
})
export class AppModule { }
```

El registro de FormsModule hace que automáticamente se evalúe cada formulario como un template-drive form añadiendo una directiva ngForm a cada uno

[(ngModel)]

```
<input type="text" id="name" required [(ngModel)]="model.name" name="name">
```

- [(ngModel)] two-way data binding permite comunicar el modelo con la vista y viceversa de una forma sencilla
- ngModel permite comprobar si el elemento ha sido visitado (touched), se ha introducido información (dirty) o si es válido (valid)
- Actualiza el control con una serie de clases en función de su estado (ng-touched, ng-dirty, ng-valid, ng-untouched, ng-pristine, ng-invalid)
- ngModel nos permite acceder a un FormControl subyacente

NgForm

```
<form #myForm="ngForm">
```

- NgForm se añade de forma automática cuando incluimos FormsModule a todos los formularios
- Complementa al elemento form con funcionalidad extra
- Contiene los controles que se han creado mediante la directiva ngModel y el atributo name (obligatorio) monitorizando sus propiedades incluida su validez.
- Podemos checkear la validez del formulario mediante la propiedad valid.
- Internamente Angular creará instancias de FormControl

Template variables

Necesitaremos variables para efectuar la gestión del formulario en la plantilla

```
<label for="name">Name</label>
<input type="text" class="form-control" id="name"
required
[(ngModel)]="model.name" name="name"
#name="ngModel">
<div [hidden]="name.valid || name.pristine"
class="alert alert-danger">
  Name is required
</div>
```

Submit

En este caso no difiere de lo visto en ReactiveForms

```
<form (ngSubmit)="onSubmit()" #heroForm="ngForm">
```

Form validation

Form validation

- Una parte crítica de la gestión de formularios es la validación de los mismos
- Angular proporciona una serie de utilidades para facilitar la validación tanto en formularios reactivos como template-driven

Reactive Forms

Validación en ReactiveForms

Se añaden validadores que proporciona el framework (built-in) o bien validadores custom

```
profileForm = this.fb.group({  
    firstName: ['', Validators.required],  
    lastName: ['', MailValidator],  
    address: this.fb.group({  
        street: ['',  
        city: ['',  
        state: ['',  
        zip: [''  
    } ),  
});
```

```
<input type="text" formControlName="firstName">
```

Built-in validators

```
class Validators {  
    static min(min: number): ValidatorFn  
    static max(max: number): ValidatorFn  
    static required(control: AbstractControl): ValidationErrors | null  
    static requiredTrue(control: AbstractControl): ValidationErrors | null  
    static email(control: AbstractControl): ValidationErrors | null  
    static minLength(minLength: number): ValidatorFn  
    static maxLength(maxLength: number): ValidatorFn  
    static pattern(pattern: string | RegExp): ValidatorFn  
    static nullValidator(control: AbstractControl): ValidationErrors | null  
    static compose(validators: ValidatorFn[]): ValidatorFn | null  
    static composeAsync(validators: AsyncValidatorFn[]): AsyncValidatorFn | null  
}
```

Custom Validation

Lo único que se necesita es pasar el FormControl y devolver un objeto con el resultado de la validación o null

```
export function MailValidator(  
  control: AbstractControl  
): { [key: string]: boolean } {  
  const EMAIL_REGEXP =  
    /^[a-zA-Z0-9!#\$%&'*+\r\n/.=^_`{|}~.-]+@[a-zA-Z0-9]( [a-zA-Z0-9-]*[a-zA-Z0-9] )?( \. [a-zA-Z0-9]( [a-zA-Z0-9-]*[a-zA-Z0-9] )? )*$/i;  
  
  if (  
    control.value &&  
    (control.value.length <= 5 || !EMAIL_REGEXP.test(control.value))  
  ) {  
    return { malformedMail: true };  
  }  
  
  return null;  
}
```

Validación cruzada

```
const heroForm = new FormGroup({  
  'name': new FormControl(),  
  'alterEgo': new FormControl(),  
  'power': new FormControl()  
}, { validators: identityRevealedValidator });
```

...

```
export const identityRevealedValidator: ValidatorFn =  
(control: FormGroup): ValidationErrors | null => {  
  const name = control.get('name');  
  const alterEgo = control.get('alterEgo');  
  
  return name && alterEgo && name.value === alterEgo.value ? { 'identityRevealed': true } : null;  
};
```

Template-driven

Validación en template-driven forms

- Se realiza a partir de atributos al igual que la validación HTML nativa
- Angular usa directivas para emparejar dichos atributos con funciones encargadas de realizar la validación
- Cada vez que un valor cambia se ejecuta la validación
- Podemos inspeccionar el resultado exportando el control a través de una variable
- Podemos utilizar validaciones personalizadas a través de creación de directivas

Validación

```
<input id="name" name="name" class="form-control"
       required minlength="4" appForbiddenName="bob"
       [(ngModel)]= "hero.name" #name="ngModel" >

<div *ngIf="name.invalid && (name.dirty || name.touched)"
      class="alert alert-danger">

  <div *ngIf="name.errors.required">
    Name is required.
  </div>
  <div *ngIf="name.errors.minLength">
    Name must be at least 4 characters long.
  </div>
  <div *ngIf="name.errors.forbiddenName">
    Name cannot be Bob.
  </div>

</div>
```

Creando validadores

En template-driven forms no tenemos acceso directo al FormControl por lo que tenemos que añadir una directiva a la plantilla que actúe como contenedor

Para que Angular la reconozca como tal necesitamos registrarla en el módulo correspondiente

```
// multi porque se pueden crear desde varios sitios. Reune las validaciones en un único provider como array
// useExisting para reutilizar la instancias. useClass crearia una cada vez que utilizamos la directiva
providers: [
  { provide: NG_VALIDATORS, useExisting: MailValidatorDirective, multi: true }
]
```

Creando validadores. La directiva

```
@Directive({
  selector: '[snMail]',
  providers: [
    { provide: NG_VALIDATORS, useExisting: MailValidatorDirective, multi: true }
  ]
})
export class MailValidatorDirective implements Validator {
  @Input('snMail') mail: string;

  validate(control: AbstractControl): { [key: string]: any } | null {
    return MailValidator(control);
  }
}
```

Validación cruzada

```
@Directive({
  selector: '[appIdentityRevealed]',
  providers: [{ provide: NG_VALIDATORS, useExisting: IdentityRevealedValidatorDirective, multi: true }]
})
export class IdentityRevealedValidatorDirective implements Validator {
  validate(control: AbstractControl): ValidationErrors {
    return identityRevealedValidator(control)
  }
}
```

```
<form #heroForm="ngForm" appIdentityRevealed>
```

Clases CSS

Generación automática de clases

- Angular genera de forma automática clases por nosotros
- Nos facilita la maquetación de la interfaz

Clases

.ng-valid

.ng-invalid

.ng-pending

.ng-pristine

.ng-dirty

.ng-untouched

.ng-touched

CSS

```
.ng-valid[required], .ng-valid.required {  
    border-left: 5px solid #42A948;  
}  
}
```

```
.ng-invalid:not(form) {  
    border-left: 5px solid #a94442;  
}  
}
```

Async validation

Async validation

- ValidatorFn -> AsyncValidatorFn
- En lugar de devolver una función devolvemos una Promise o Observable
- El Observable debe retornarse en un estado completado (first, last, take, or takeUntil)
- Cuando comienza la validación podemos visualizar el estado pending en nuestros controles asociados

Pending Async validation

```
<input [(ngModel)]="name" #model="ngModel" appSomeAsyncValidator>
<app-spinner *ngIf="model.pending"></app-spinner>
```

Ejemplo

```
@Injectable({ providedIn: 'root' })
export class UniqueAlterEgoValidator implements AsyncValidator {
  constructor(private heroesService: HeroesService) {}

  validate(
    ctrl: AbstractControl
  ): Promise<ValidationErrors | null> | Observable<ValidationErrors | null> {
    return this.heroesService.isAlterEgoTaken(ctrl.value).pipe(
      map(isTaken => (isTaken ? { uniqueAlterEgo: true } : null)),
      catchError(() => null)
    );
  }
}
```

UpdateOn

- Es interesante en algunas situaciones validar solamente los formularios una vez que hacemos foco fuera del campo
- Un caso claro serian las validaciones asincronas. Seria muy negativo hacerlo cada vez que tecleamos sobre el campo

UpdateOn

Template-driven

```
<input [(ngModel)]="name" [ngModelOptions]="{updateOn: 'blur'}">
```

FormControl

```
new FormControl(' ', {updateOn: 'blur'});
```

FormGroup

```
{ updateOn: 'blur' }
```

NgModule

Módulos en Angular

- Los módulos en Angular permiten agrupar componentes, directivas, servicios y otra serie de artefactos que constituyen una unidad funcional
- Son una forma de organizar la aplicación
- El core de Angular esta formado por una serie de NgModules como FormsModule, HttpClientModule, o RouterModule
- No tienen nada que ver con ES6 Modules o CommonJS

Sintaxis

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

declarations

- Lista de clases que pertenecen al módulo. Pipes, Services, Components, Directives
- Las templates de los componentes se compilan en el contexto del módulo. Necesito por tanto tener todos los artefactos disponibles o bien declarados o importados
- Las clases declaradas en declarations no se pueden declarar en otros módulos

providers

- Lista de proveedores para registrar en el injector del NgModule y que nos permitirán su uso a través del mecanismo de DI. Si es el módulo encargado del bootstrap entonces es el root injector
- Los servicios pasan a estar disponibles para cada componente, directiva, pipe o service hijo del injector

imports

- Otros módulos que voy a usar en el que estoy definiendo
- Los módulos importados es como si sus exports se hubiesen declarado en el NgModule que los importa y por tanto todos sus artefactos pasan a estar disponibles

exports

- Lista de declarations que quiero exportar para su uso por otros módulos mediante import
- Es el API pública del NgModule.
- declarations son privadas y exports públicas
- Los módulos importados se pueden a su vez exportar

bootstrap

- Lista de componentes raiz para arrancar la aplicación
- Normalmente sólo es uno => Root component

entryComponents

- Lista de componentes que se pueden cargar de forma dinámica en la aplicación. No están definidos en ninguna plantilla
- El componente encargado de hacer el bootstrap y los componentes cargados por el router se añaden de forma automática a los entryComponents
- Otros componentes que se pretendan cargar de forma dinámica se tienen que añadir

NgModules comunes

NgModule	Import	Uso
BrowserModule	@angular/platform-browser	Para usar la aplicación en un Browser
CommonModule	@angular/common	Importa ciertas directivas como NgIf o NgFor
FormsModule	@angular/forms	Template-driven forms
ReactiveFormsModule	@angular/forms	Reactive Forms
RouterModule	@angular/router	Cuando quiero usar el router
HttpClientModule	@angular/common/http	Para comunicarme con un servidor

NgModules comunes

- BrowserModule importa y exporta CommonModule
- Por tanto, se debe usar BrowserModule en AppModule y CommonModule en el resto de módulos funcionales

Feature Modules

- Agrupaciones funcionales para una aplicación Angular
- No tienen nada de especial, simplemente se usan a nivel organizativo
- Normalmente tendremos un módulo raiz y varios funcionales
- El módulo raiz importara los módulos funcionales para formar una aplicación

Lazy Loading

Lazy Loading

- Técnica para importar de forma asíncrona otros módulos de la aplicación
- Permite que el tamaño del bundle sea inferior en la primera carga
- Es la forma que tiene Angular de activar Code Splitting en nuestra aplicación

Definiendo rutas

```
const routes: Routes = [
  {
    path: 'customers',
    loadChildren: './customers/customers.module#CustomersModule'
  },
  {
    path: 'orders',
    loadChildren: './orders/orders.module#OrdersModule'
  },
  {
    path: '',
    redirectTo: '',
    pathMatch: 'full'
  }
];
```

Routing module

```
const routes: Routes = [
  {
    path: '',
    component: CustomerListComponent
  }
];
@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class CustomersRoutingModule { }
```

Importante !!



- Cuando estoy haciendo lazy loading debo eliminar el import del módulo de AppModule o la carga se hará de forma normal y no funcionará correctamente

Angular Router

Single Page Applications (SPA)

- Tradicionalmente la navegación se realizaba a través de un servidor que generaba el código HTML de la página y lo enviaba al cliente conectado
- Desde hace unos años nuevos modelos se han establecido como Single Page Applications o Server Side Rendering
- En una SPA entregamos una plantilla HTML básica al cliente con una serie de bundles javascript y a partir de ahí es el cliente el encargado de construir la página

SPA. Ventajas

- Una vez cargada la página la velocidad y respuesta de la página es mayor y la experiencia de usuario mejora
- El cacheo de datos en local es muy efectivo
- Capacidad de depuración aumenta con los frameworks JS y herramientas disponibles en los navegadores
- Desacoplamiento claro de un API REST

SPA. Inconvenientes

- El renderizado inicial es mas lento
- SEO. Google no renderiza perfectamente JS
- Browser History, moverse entre estados. Necesitamos ayuda, por ejemplo el **Router de Angular**

El navegador y su modelo de navegación

- A través de direcciones en la barra de direcciones
- A través de eventos click en links
- Utilizando los botones de navegación

Angular Router

El router de Angular, como en el resto de Frameworks JS intenta replicar el modelo de navegación de los navegadores y facilitar el uso de la SPA

Configuración del router

Necesito configurar la URL base o decirle a Angular a partir de dónde encontrar las rutas

```
<base href="/">
```

Tengo que importar el módulo

```
import { RouterModule, Routes } from '@angular/router';
```

Configuración del router

Creación de las rutas. El orden importa y gana la primera que coincide

```
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'hero/:id',
    component: HeroDetailComponent },
  {
    path: 'heroes',
    component: HeroListComponent,
    data: { title: 'Heroes List' }
  },
  { path: '',
    redirectTo: '/heroes',
    pathMatch: 'full'
  },
  { path: '**', component: PageNotFoundComponent }
];
```

Configuración del router en NgModule

```
@NgModule( {  
    imports: [  
        RouterModule.forRoot(  
            appRoutes,  
            { enableTracing: true } // Debug  
        )  
        ...  
    ],  
    ...  
})  
export class AppModule { }
```

Router

Es el servicio que nos permite realizar la navegación entre rutas así como el acceso a todas las propiedades principales

Routes

El tipo que define el array de rutas de la aplicación y sus mapeos con el correspondiente componente

Route

- La interfaz de cada uno de los objetos que forman el array de Routes
- Define cómo el Router debe navegar a un componente basado en un patrón de URL

RouterOutlet

Directiva que se encarga de marcar la posición en la que empezar a renderizar en la plantilla

```
<router-outlet></router-outlet>
```

RouterLink

Directiva para navegación entre rutas. Admite strings o array dinámico

```
<h1>Angular Router</h1>
<nav>
  <a routerLink="/crisis-center">Crisis Center</a>
  <a routerLink="/heroes">Heroes</a>
</nav>
<router-outlet></router-outlet>
```

RouterLink dinámico

```
// /team/11/user/bob;details=true
<a [routerLink]=["/team", teamId, 'user', userName, {details: true}]>
</a>
```

```
// /user/bob?debug=true
<a [routerLink]=["/user/bob"] [queryParams]={debug: true}>
</a>
```

RouterLinkActive

Directiva para añadir o eliminar clases en función de si una ruta esta activa o no

```
<a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
```

RouterState

- El estado actual del Router
- Después de cada navegación se construye el árbol de rutas activadas (ActivatedRoute)
- Se puede acceder al RouterState a través del servicio Router

RouterState

```
@Component({templateUrl:'template.html'})  
class MyComponent {  
  constructor(router: Router) {  
    const state: RouterState = router.routerState;  
    const root: ActivatedRoute = state.root;  
    const child = root.firstChild;  
    const id: Observable<string> = child.params.map(p => p.id);  
    //...  
  }  
}
```

ActivatedRoute

Servicio que se utiliza para acceder a la información de la ruta actual

```
@Component({...})
class MyComponent {
  constructor(route: ActivatedRoute) {
    const id: Observable<string> = route.params.map(p => p.id);
    const url: Observable<string> = route.url.map(segments => segments.join(''));
    // route.data includes both `data` and `resolve`
    const user = route.data.map(d => d.user);
  }
}
```

ActivatedRouteSnapshot

- En muchos casos utilizaré el snapshot de la ruta.
- ActivatedRoute contiene muchas propiedades Observables por lo que resulta interesante para suscribirse a cambios
- ActivatedRouteSnapshot en cambio nos proporciona acceso al valor actual de dichas propiedades

Router Events

- El router emite eventos en cada navegación.
- Emite eventos a través de la propiedad Observable Router.events

Usando Los eventos del Router para realizar acciones

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

  navStart: Observable<NavigationStart>;

  constructor(private router: Router) {
    // Create a new Observable the publishes only the NavigationStart event
    this.navStart = router.events.pipe(
      filter(evt => evt instanceof NavigationStart)
    ) as Observable<NavigationStart>;
  }

  ngOnInit() {
    this.navStart.subscribe(evt => console.log('Navigation Started!'));
  }
}
```

Routing component

Se dice que un componente es un routing component cuando tiene un RouterOutlet en su plantilla

Routing Module

Es habitual crear uno o más módulos para definir las rutas Este módulo se importará en el módulo principal o funcional asociado

```
const appRoutes: Routes = [
  { path: '', redirectTo: '/heroes', pathMatch: 'full' },
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'heroes', component: HeroListComponent },
  { path: '**', component: PageNotFoundComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(appRoutes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

Child Routes. Definiendo las rutas

```
const crisisCenterRoutes: Routes = [
  {
    path: 'crisis-center',
    component: CrisisCenterComponent,
    children: [
      {
        path: '',
        component: CrisisListComponent,
        children: [
          {
            path: ':id',
            component: CrisisDetailComponent
          },
          {
            path: '',
            component: CrisisCenter HomeComponent
          }
        ]
      }
    ]
  }
];
```

Child Routes. Definiendo el Routing Module como módulo funcional

```
@NgModule({  
  imports: [  
    RouterModule.forChild(crisisCenterRoutes)  
,  
  exports: [  
    RouterModule  
  ]  
})  
export class CrisisCenterRoutingModule { }
```

Guards

```
@Injectable({  
  providedIn: 'root',  
})  
export class AuthGuard implements CanActivate {  
  canActivate(  
    next: ActivatedRouteSnapshot,  
    state: RouterStateSnapshot): boolean {  
    console.log('AuthGuard#canActivate called');  
    return true;  
  }  
}
```

Guards

```
const adminRoutes: Routes = [
  {
    path: 'admin',
    component: AdminComponent,
    canActivate: [AuthGuard],
    children: [
      {
        path: '',
        children: [
          { path: 'crises', component: ManageCrisesComponent },
          { path: 'heroes', component: ManageHeroesComponent },
          { path: '', component: AdminDashboardComponent }
        ],
      }
    ]
  }
];
```

Lazy Loading

Mediante lazy loading puedo cargar módulos de forma asíncrono al navegar a la ruta correspondiente

```
{  
  path: 'admin',  
  loadChildren: './admin/admin.module#AdminModule',  
},
```

Resolvers. Definición

```
@Injectable({
  providedIn: 'root',
})
export class CrisisDetailResolverService implements Resolve<Crisis> {
  constructor(private cs: CrisisService, private router: Router) {}

  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<Crisis> | Observable<never> {
    let id = route.paramMap.get('id');

    return this.cs.getCrisis(id).pipe(
      take(1),
      mergeMap(crisis => {
        if (crisis) {
          return of(crisis);
        } else { // id not found
          this.router.navigate(['/crisis-center']);
          return EMPTY;
        }
      })
    );
  }
}
```

Resolvers. Uso

```
{  
  path: ':id',  
  component: CrisisDetailComponent,  
  resolve: {  
    crisis: CrisisDetailResolverService  
  }  
},  
  
ngOnInit() {  
  this.route.data  
    .subscribe((data: { crisis: Crisis }) => {  
      this.editName = data.crisis.name;  
      this.crisis = data.crisis;  
    });  
}
```

Inyección de dependencias (DI)

En qué consiste el patrón DI

```
class Car {  
    constructor() {  
        this.engine = new Engine();  
        this.tires = Tires.getInstance();  
        this.doors = app.get('doors');  
        this.milesDriven = 0;  
    }  
  
    drive(miles) {  
        this.milesDriven = this.milesDriven + miles;  
    }  
}
```

```
class Car {  
    constructor(engine, tires, doors) {  
        this.engine = engine;  
        this.tires = tires;  
        this.doors = doors;  
        this.milesDriven = 0;  
    }  
  
    drive(miles) {  
        this.milesDriven = this.milesDriven + miles;  
    }  
}
```

```
var car = new Car(  
    new Engine(),  
    new Tires(),  
    new Doors()  
);
```

```
var mockCar = new Car(  
    new MockEngine(),  
    new MockTires(),  
    new MockDoors()  
);
```

DI en Angular

Cómo inyectar un servicio en Angular

```
@Component( {  
    selector: 'example-component',  
    template: ' <div>I am a component</div> '  
})  
class ExampleComponent {  
    constructor(private http: Http) {  
        this.http.get() ...  
    }  
}
```

Cómo inyectar un servicio en Angular

- A través de la definición del tipo Http con Typescript Angular asigna una instancia del servicio al parámetro http
- tsconfig.json contiene una propiedad *emitDecoratorMetadata* establecida a true. Esto emite metadata acerca del tipo del parámetro al compilar el TypeScript a JavaScript

Cómo inyectar un servicio en Angular

Versión compilada de nuestro Componente

```
var ExampleComponent = (function() {
  function ExampleComponent(http) {
    this.http = http;
  }
  return ExampleComponent;
})();
ExampleComponent = __decorate(
[
  Component({
    selector: 'example-component',
    template: '<div>I am a component</div>',
  }),
  __metadata('design:paramtypes', [Http]),
],
ExampleComponent
);
```

Cómo inyectar un servicio en Angular

Al compilar a JavaScript estamos informando a Angular que el primer parámetro de nuestro componente necesita una instancia del servicio Http

```
__metadata( 'design:paramtypes' , [Http] );
```

Angular se encargará a través de una serie de reglas de buscar la instancia de ese servicio para proporcionársela al constructor

@Inject

@Inject es una forma manual de buscar un determinado **token** (Http) para su inyección

```
@Component({  
  selector: 'example-component',  
  template: '<div>I am a component</div>'  
})  
class ExampleComponent {  
  constructor(@Inject(Http) private http) { }  
}
```

Si hay un montón de dependencias el constructor puede convertirse en ilegible. Dado que Angular soporta la inyección a través de la emisión de metadatos la mayor parte del tiempo (o nunca probablemente) no es necesario usar @Inject

@Injectable

Usamos `@Injectable` para declarar un servicio como apto para ser usado como dependencia. Realmente sólo es obligatorio si el servicio cuenta a su vez con alguna dependencia (Si no lo añadimos no se emitiría metadata) pero por convención se añade SIEMPRE

```
@Injectable()
export class UserService {
  constructor(private http: Http) {}
  isAuthenticated(): Observable<boolean> {
    return this.http.get('/api/user').map((res) => res.json());
  }
}
```

Registrando un provider

```
@NgModule({  
  providers: [AuthService],  
})  
class ExampleModule {}
```

es equivalente a:

```
@NgModule({  
  providers: [  
    {  
      provide: AuthService,  
      useClass: AuthService,  
    },  
  ],  
})  
class ExampleModule {}
```

Registrando un provider

- La propiedad provide es el token para el provider que estamos registrando
- Angular puede buscar qué esta almacenado bajo el token AuthService usando el valor de useClass
- Una de las ventajas que nos proporciona es que podemos tener 2 proveedores diferentes usando la misma clase o sobreescibir el proveedor con otro diferente manteniendo el nombre del token

Sobreescribiendo providers

```
@NgModule({
  declarations: [LoginComponent, UserInfoComponent],
  providers: [
    {
      provide: AuthService,
      useClass: MyAppAuthService,
    },
  ],
})
export class AuthModule {}
```

```
@NgModule({
  declarations: [LoginComponent, UserInfoComponent],
  providers: [
    {
      provide: AuthService,
      useClass: FacebookAuthService,
    },
  ],
})
export class AuthModule {}
```

@Optional

Podemos hacer que una dependencia sea opcional

```
constructor(@Optional() private logger: Logger) {  
    if (this.logger) {  
        this.logger.log(some_message);  
    }  
}
```

Value Providers

Podemos proporcionar string, numbers, o objetos en lugar de pedir a Angular que cree una instancia de una clase

```
export function SilentLoggerFn() {}
```

```
const silentLogger = {  
  logs: ['Silent logger says "Shhhhh!". Provided via "useValue"',  
  log: SilentLoggerFn  
};  
  
[{ provide: Logger, useValue: silentLogger }]
```

Factory Providers

Podemos utilizar una factoría para crear instancias de forma dinámica

```
constructor(  
  private logger: Logger,  
  private isAuthorized: boolean) { }  
  
getHeroes() {  
  let auth = this.isAuthorized ? 'authorized' : 'unauthorized';  
  this.logger.log(`Getting heroes for ${auth} user. `);  
  return HEROES.filter(hero => this.isAuthorized || !hero.isSecret);  
}
```

Qué pasa si no quiero injectar un servicio completo AuthService y solamente quiero saber si el usuario está autorizado.

Podemos injectar Logger pero no isAuthorized como tal. Necesitamos una factoría

Factory Providers

```
let heroServiceFactory = (logger: Logger, userService: UserService) => {
  return new HeroService(logger, userService.user.isAuthorized);
};

export let heroServiceProvider =
{ provide: HeroService,
  useFactory: heroServiceFactory,
  deps: [Logger, UserService]
};
```

Factory Providers

```
@Component( {  
    selector: 'app-heroes' ,  
    providers: [ heroServiceProvider ] ,  
    template: `  
        <h2>Heroes</h2>  
        <app-hero-list></app-hero-list>  
    `  
})  
export class HeroesComponent { }
```

Tokens y Providers de Angular

- Angular proporciona una serie de tokens y providers que podemos usar para modificar el comportamiento del framework
- *PLATFORMINITIALIZER, APPBOOTSTRAPLISTENER, APPINITIALIZER, NG_VALIDATORS, RouterModule.forRoot, RouterModule.forChild* son algunos ejemplos

Tree-Shaking de providers

Cuando especifico el array de providers en el NgModule, el servicio acabará en el bundle final aunque no se esté usando en la aplicación ya que no hay forma de saberlo de antemano

Hay una forma de evitarlo (Angular 6+)

```
@Injectable({
  providedIn: 'root', // Ahora el bundler puede verificar si se usa o no
})
export class Service {  
  
@Injectable({
  providedIn: 'root',
  useFactory: () => new Service('dependency'),
})
export class Service {
  constructor(private dep: string) {
  }
}
```

Injectors

Injectors

- El sistema de inyección de dependencias de Angular es jerárquico
- Hay un arbol de inyectores paralelo al arbol de componentes de la aplicación
- Estos inyectores son reconfigurables

Platform Injector

- Se usa internamente durante el bootstrap de la app y es el injector que encontramos más arriba en la jerarquía
- Configura servicios específicos de la plataforma
- Se pueden configurar más proveedores a través de `extraProviders` en la función `platformBrowser()`

```
const platform = platformBrowserDynamic([ {  
  provide: SharedService,  
  deps: []  
}]);  
platform.bootstrapModule(AppModule);  
platform.bootstrapModule(AppModule2);
```

Root injector

- Se crea y asocia con nuestro AppModule

@Injectable

- Podemos configurar en donde queremos injectar el servicio a través de este decorador y junto con providedIn
- Es típico usar 'root' que representa el injector de AppModule y válido para la mayoría de casos de uso. Esto generará un servicio Singleton para toda la aplicación

@Injectable. Como se usa

En este caos indicamos a Angular que el injector de AppModule será el responsable de generar las instancias

```
@Injectable({  
  providedIn: 'root',  
})  
export class HeroService {  
  constructor() {}  
}
```

@Injectable. Como se usa

Aquí declaramos que HeroService debe ser creado por cualquier inyector de HeroModule

```
@Injectable({  
  providedIn: HeroModule,  
})  
export class HeroService {  
  getHeroes() { return HEROES; }  
}
```

Es casi lo mismo que usar NgModule para configurar el injector con la salvedad de que hacerlo así permite hacer uso del tree-shaking y por tanto es más óptimo

@NgModule

Se puede configurar en el inyector del módulo. La diferencia es la comentada anteriormente respecto a usar @Injectable

```
providers: [
  { provide: LocationStrategy, useClass: HashLocationStrategy }
]
```

@Component

Los componentes de un módulo tienen sus propios inyectores. Puedo configurar los providers a este nivel y limitar el uso del servicio a este componente y sus hijos

```
@Component({  
  selector: 'app-heroes',  
  providers: [ HeroService ],  
  template: `  
    <h2>Heroes</h2>  
    <app-hero-list></app-hero-list>  
  `,  
})  
export class HeroesComponent { }
```

@Directive

- Un @Component es una directiva al fin y al cabo. por tanto puedo configurarlo también a nivel de directiva
- El injector en un componente o directiva se crea a nivel de DOM Element. Si tengo un componente con varias directivas sobre el mismo elemento, compartirán el injector

Injector bubbling

- El mecanismo de búsqueda de un proveedor para mi dependencia es el típico que se sigue por ejemplo en el bubbling de eventos del DOM
- Comienzo la búsqueda por el injector del componente que solicita la dependencia
- Luego seguiré por el injector del componente padre
- Luego me voy al de módulo si no hay más padres
- Luego al root
- Finalmente me iría al platform injector si no lo he encontrado en ningún otro sitio. En este último caso se lanzaría una excepción

@Host

Limita la búsqueda del proveedor al componente en el que lo usamos

```
@Component({
  selector: 'app-hero-contact',
  template: `
    <div>Phone #: {{phoneNumber}}
    <span *ngIf="hasLogger">!!!</span></div>
  `})
export class HeroContactComponent {
  hasLogger = false;
  constructor(
    @Host() private heroCache: HeroCacheService,
    @Host() @Optional() private loggerService: LoggerService
  ) {
    if (loggerService) {
      this.hasLogger = true;
      loggerService.logInfo('HeroContactComponent can log!');
    }
  }
  get phoneNumber() { return this.heroCache.hero.phone; }
}
```

Animaciones

Animaciones

- Angular tiene su propio lenguaje de definición de animaciones
- El sistema está construido sobre la nueva api de animaciones web (WAAPI)
- Para empezar tenemos que añadir el módulo BrowserAnimationsModule

BrowserAnimationsModule

// @angular/animations en package.json

```
import {BrowserAnimationsModule}
  from '@angular/platform-browser/animations';

@NgModule({
  imports: [BrowserAnimationsModule]
})
class AppModule {}
```

@triggers

- Las animaciones en Angular se inician a través de un tipo especial de property binding **[@{trigger_name}]**
- Cuando una propiedad cambia de estado o un elemento se inserta o desaparece del DOM se lanzará una animación
- El @trigger está enlazado con el componente, que es donde realmente se declara y define la animación

@triggers

```
<!-- Anima cuando `someStateValue` cambia. Usamos state() -->
<div [@myAnimationTrigger]="someStateValue">
  ...
</div>

<!-- Anima cuando se añade o elimina del DOM (enter/leave) -->
<div @myAnimationTrigger *ngIf="exp">
  ...
</div>
```

Definición de animaciones

La animación se define en el @Component

```
@Component({
  template: `
    <div [@myAnimationTrigger]="myValue">...</div>
  `,
  animations: [
    trigger('myAnimationTrigger', [
      transition('* => someState', [
        // animaciones
      ])
    ])
  ]
})
class MyComponentWithAnimations {
  myValue = 'someState';
}
```

Creación de triggers

Un trigger lanza la transición que consiste en varios pasos o estados

```
trigger('myAnimationTrigger', [
  transition('* => visible', [
    style({ opacity: 0 }), // Estado inicial
    animate('500ms', style({ opacity: 1 })) // Qué tiene que hacer
  ),
  transition('* => hidden', [
    animate('500ms', style({ opacity: 0 }))
  ])
])
```

Creación de estados

Cuando la animación se completa se irá al estado correspondiente y se aplicarán los estilos definidos

Esta animación y la anterior son equivalentes

```
trigger('myAnimationTrigger', [
  state('visible', style({ opacity: 1 })), // Estilos iniciales
  state('hidden', style({ opacity: 0 })), // Estilos iniciales
  transition('* => visible', [
    animate('500ms')
  ]),
  transition('* => hidden', [
    animate('500ms')
  ])
])
```

style()

Aplica el estilo al elemento de forma inmediata

```
trigger('myAnimationTrigger', [
  transition('* => visible', [
    // fade out del elemento de forma inmediata
    style({ opacity: 0 }),

    //...
  ]),
  transition('* => hidden', [
    // Usa la opacidad actual del elemento
    style({ opacity: '*' }),

    //...
  ])
])
```

animate() + style()

animate() + style() aplicarán el estilo al elemento a lo largo de una duración indicada

```
trigger('myAnimationTrigger', [
  transition('* => visible', [
    // fade out del elemento de forma inmediata
    style({ opacity: 0 }),

    // Animar la opacidad a lo largo de 1 sec
    animate(1000, style({ opacity: 1 }))
  ]),

  transition('* => hidden', [
    // Usa la opacidad actual del elemento
    style({ opacity: '*' }),

    // Anima la opacidad a lo largo de 500ms
    animate('500ms', style({ opacity: 0 }))
  ])
])
```

animate() + keyframes()

Aplicará una serie de keyframes a lo largo de una linea temporal

```
trigger('myAnimationTrigger', [
  transition('* => visible', [
    animate('1s', keyframes([
      style({ opacity: 0 }), // 0%
      style({ opacity: 0.8 }), // 33%
      style({ opacity: 0.2 }), // 66%
      style({ opacity: 1 }), // 100%
    ]))
  ]),
  transition('* => hidden', [
    animate('1s', keyframes([
      style({ opacity: 1, offset: 0 }), // 0%
      style({ opacity: 0.2, offset: 0.8 }), // 80%
      style({ opacity: 0.4, offset: 0.9 }), // 90%
      style({ opacity: 0, offset: 1 }), // 100%
    ]))
  ])
])
```

:enter & :leave

Cuando un item se inserta o elimina del DOM se usan :enter y :leave

```
// Template => <div *ngIf="exp" @myInsertRemoveTrigger>...</div>

trigger('myInsertRemoveTrigger', [
  transition(':enter', [
    style({ opacity: 0 }),
    animate('1s', style({ opacity: 1 }))
  ]),
  transition(':leave', [
    animate('1s', style({ opacity: 0 }))
  ])
])
```

:enter se lanza cuando algún *ngIf / *ngFor introduce un nuevo elemento en el DOM
:leave se lanza cuando se elimina un elemento del DOM

(@animation.callback)

Cuando una animación comienza o finaliza emite callbacks

```
@Component({  
  animations: [  
    trigger('myInsertRemoveTrigger', [  
      //...  
    ])  
  ],  
  template: `  
    <div *ngIf="exp"  
      @myInsertRemoveTrigger  
      (@myInsertRemoveTrigger.start)="onAnimationEvent($event)"  
      (@myInsertRemoveTrigger.done)="onAnimationEvent($event)">...</div>  
  `  
})  
class MyInsertRemoveComponent {  
  onAnimationEvent(event: AnimationEvent) { /* ... */ }  
}
```

(@animation.callback)

- (@trigger.start) Se llama al empezar la animación
- (@trigger.done) Cuando se completa
- Un AnimationEvent se envía via \$event

(@animation.callback)

```
onAnimationEvent(event: AnimationEvent) {  
    console.log(event.triggerName); // myInsertRemoveTrigger  
    console.log(event.phaseName); // start o done  
    console.log(event.totalTime); // 1000  
    console.log(event.fromState); // void o *  
    console.log(event.toState); // * o void  
    console.log(event.element); // el element  
}
```

query()

El gran poder de las animaciones con Angular esta en query()
Se utiliza para seleccionar elementos DOM desde un padre

```
trigger('pageAnimations', [
  transition(':enter', [
    query('.hero, form', [
      style({opacity: 0, transform: 'translateY(-100px)'}),
      stagger(-30, [
        animate('500ms cubic-bezier(0.35, 0, 0.25, 1)', 
          style({ opacity: 1, transform: 'none' }))])])])])])
```

stagger()

Se usa junto con query para producir un retardo entre los elementos a animar

Muy útil para la animación de listas

```
transition(':decrement', [
  query(':leave', [
    stagger(50, [
      animate('300ms ease-out',
        style({ opacity: 0, width: '0px' }))),
    ]),
  ])
])
```

group() & sequence()

Sirve para lanzar animaciones en paralelo

```
transition('void => *', [
  style({ width: 10, transform: 'translateX(50px)', opacity: 0 }),
  // or sequence()
  group([
    animate('0.3s 0.1s ease', style({
      transform: 'translateX(0)',
      width: 120
    })),
    animate('0.3s ease', style({
      opacity: 1
    }))
  ])
])
```

:increment & :decrement

Se lanzan cuando un valor numérico se incrementa o decremente

```
trigger('filterAnimation', [
  transition(':enter, * => 0, * => -1', []),
  transition(':increment', [
    query(':enter', [
      style({ opacity: 0, width: '0px' }),
      stagger(50, [
        animate('300ms ease-out', style({ opacity: 1, width: '*' })),
      ]),
    ], { optional: true })
  ]),
  transition(':decrement', [
    query(':leave', [
      stagger(50, [
        animate('300ms ease-out', style({ opacity: 0, width: '0px' })),
      ]),
    ])
  ]),
])
```

animateChild()

Si voy a animar elementos hijos es necesario que esperen a que el padre haya acabado sus propias animaciones ya que tienen prioridad.

En caso contrario no se verían las animaciones del hijo. Esta diseñado para funcionar con query()

```
transition(':enter', [
  query('@listItems:not(sn-post-comment)', 
    stagger(300, animateChild()), {
      optional: true // Puede no estar presente en el DOM
    })
])
```

HttpClient

HttpClient

- Navegadores => XMLHttpRequest y fetch
- HttpClient es una implementación de Angular para la comunicación con APIs

Configuración

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

Uso

Normalmente en servicios (mejor práctica) en lugar de directamente en componentes

```
@Injectable()  
export class MyService {  
  constructor(private http: HttpClient) { }  
}
```

Métodos ofrecidos

- get
- post
- put
- delete
- patch
- head
- jsonp

Uso de HttpClient

Se devuelve un observable que por tanto puedo transformar mediante operadores (o incluso convertir a Promise conm toPromise())

```
http.get(`$baseUrl}/api/races`)  
  .subscribe((response: Array<RaceModel>) => {  
    console.log(response);  
});
```

Leyendo la respuesta

Por defecto se lee el body.

A veces podemos necesitar leer cabeceras o cualquier otra cosa en la respuesta

```
uploadAvatar(image: File) {  
  const formData = new FormData();  
  formData.append('avatar', image);  
  return this.http.post(` ${environment.apiUrl}/user/avatar`, formData, {  
    observe: 'response'  
  });  
}
```

Añadiendo Headers

```
const httpOptions = {  
  headers: new HttpHeaders({  
    'Content-Type': 'application/json',  
    'Authorization': 'my-auth-token'  
  })  
};  
  
this.http.post<Config>(this.configUrl, config, httpOptions)  
.pipe(  
  catchError(this.handleError)  
);
```

Parámetros en la URL

```
search(text: string) {  
  return this.http.get<Profile[]>(  
    `${environment.apiUrl}/user/search` , {  
      params: { q: text }  
    } );  
}
```

Gestión de errores

catchError se encarga de capturar el error y no provoca el final en la ejecución del Observable

```
getConfig() {  
    return this.http.get<Config>(this.configUrl)  
        .pipe(  
            catchError(this.handleError)  
        );  
}
```

También podría utilizar la función de error del Observer pasado a la suscripción pero catchError es mejor práctica

Type-checking de las respuestas

El siguiente código provoca un error en el compilador de TypeScript

```
.subscribe((data: Config) => this.config = {
  configUrl: data.configUrl, // data.configUrl typescript error
  textfile: data.textfile
});
```

Pero si tipamos la respuesta mediante generics

```
getConfig() {
  return this.http.get<Config>(this.configUrl);
}
```

Ahora ya no tendríamos dicho error por lo que es la convención

Interceptors

```
@Injectable()
export class MyInterceptor implements HttpInterceptor {

  intercept(req: HttpRequest<any>,
            next: HttpHandler): Observable<HttpEvent<any>> {
    return next.handle(req);
  }
}
```

Interceptors

- intercept transforma una request en un Observable
- next permite llamar al siguiente interceptor en la cadena de interceptors

Interceptors

```
{ provide: HTTP_INTERCEPTORS, useClass: MyInterceptor, multi: true }
```

multi indica a Angular que este token es una array de valores, es decir, puedo proporcionar múltiples interceptors en diferentes puntos y Angular lo convertirá en un Array

Observables

Que es un Observable

- Un Observable representa un stream por el que fluyen datos que podemos manipular mediante operadores y finalmente recuperarlos mediante una suscripción
- Proporciona soporte para el envío de mensajes entre publicadores y subscriptores. Patrón pub/sub
- Son declarativos, no se ejecutan hasta que consumidor se subscribe
- El consumidor recibe notificaciones hasta que la función se completa, se da de baja o se produce un error no controlado

Creando un Observable

```
const observable = Observable.create(function (observer) {  
    observer.next(1);  
    observer.next(2);  
    observer.next(3);  
    setTimeout(() => {  
        observer.next(4);  
        observer.complete();  
    }, 1000);  
});  
  
// A subscribe le paso el observer => { next, error, complete }  
observable.subscribe(next, error, complete); // Lo ejecutamos
```

Qué nos proporcionan los Observables que no tengamos ya

Single

Multiple

Pull

Function

Generator

Push

Promise

Observable

Pull vs Push

- En un sistema Pull el consumidor determina cuando recibe los datos del productor. El productor no sabe cuando se enviarán estos datos. Las funciones de Javascript y los generators son un claro ejemplo
- En un sistema Push el productor determina cuando enviar los datos al consumidor. El consumidor no sabe cuando va a recibir estos datos. Las Promises y los Observables encajan en este sistema
- Las Funciones y los Observables son sistemas Pasivos, es decir, hasta que no se invocan no comienzan a producir valores. Las Promises y los generators son sistemas Activos que producen valores
- Suscribirse a un observable es equivalente a llamar a una función (), call() o apply() vs subscribe()

Creación de un Observable de forma declarativa

El siguiente Observable emite un string hi cada segundo durante un tiempo indefinido

Observable.create es un alias para el constructor de Observable

```
const observable = Observable.create(function subscribe(observer) {  
  const id = setInterval(() => {  
    observer.next('hi')  
  }, 1000);  
});
```

Suscripción a un Observable

Cuando me suscribo a un Observable le paso un Observer, que basicamente es un objeto con tres funciones { next, error, complete }

```
observable.subscribe(x => console.log(x));
```

Cada suscripción a un Observable es independiente y es como una nueva "llamada a la función"

Tipos de notifications que un Observable puede emitir

- Next: Envía un valor como un número, string o objeto
- Error: Envía un error JavaScript o una excepción. Corta la ejecución del Observable y nada más se enviará
- Complete: No envía nada. Corta la ejecución del Observable y nada más se enviará

Error

Cuando se produce un error también se dejan de enviar valores por lo que es una buena idea controlarlos mediante try/catch

```
const observable = Observable.create(function subscribe(observer) {  
  try {  
    observer.next(1);  
    observer.next(2);  
    observer.next(3);  
    observer.complete();  
  } catch (err) {  
    observer.error(err); // delivers an error if it caught one  
  }  
});
```

Complete

```
const observable = Observable.create(function subscribe(observer) {  
  observer.next(1);  
  observer.next(2);  
  observer.next(3);  
  observer.complete();  
  observer.next(4); // no se envía  
});
```

Eliminando suscripciones y liberando recursos

Una suscripción representa un recurso que se puede liberar mediante la función unsubscribe

```
const subscription = observable.subscribe(x => console.log(x));  
  
subscription.unsubscribe();
```

El pipe async de Angular cancela las suscripciones por si mismo por lo que nos ahorraremos hacerlo en el OnDestroy

Retornando la función para eliminar la suscripción

A veces necesito más control para eliminar la suscripción

```
var observable = Observable.create(function subscribe(observer) {  
    var intervalID = setInterval(() => {  
        observer.next('hi');  
    }, 1000);  
  
    return function unsubscribe() {  
        clearInterval(intervalID);  
    };  
});
```

Subjects

- Cada subscribe sobre un Observable representaba un contexto de ejecución diferente en el que no se compartía nada
- Subject es un tipo especial de Observable que permite compartir suscripciones a varios Observers y además puede emitir valores a todos ellos (multicast)
- **Una Subject es un Observable** y por tanto podemos subscribirnos a ella proporcionando un Observer. El funcionamiento es el mismo pero internamente no se crea un nuevo contexto de ejecución y simplemente registra al nuevo Observer en la lista de Observers de la Subject
- **Una Subject es un Observer** y por tanto tiene acceso a los métodos next, error y complete. Puede por tanto emitir valores a todos los Observers conectados
- Además, como una Subject es un Observer puedo proporcionarla como parámetro a la función subscribe de cualquier Observable

Subject con varios subscriptores

```
const subject = new Subject<number>();

subject.subscribe({
  next: (v) => console.log(`observerA: ${v}`)
});

subject.subscribe({
  next: (v) => console.log(`observerB: ${v}`)
});

subject.next(1);
subject.next(2);

// Logs:
// observerA: 1 | observerB: 1 | observerA: 2 | observerB: 2
```

Subject como Observer

```
const subject = new Subject<number>();

subject.subscribe({
  next: (v) => console.log(`observerA: ${v}`)
});

subject.subscribe({
  next: (v) => console.log(`observerB: ${v}`)
});

const observable = from([1, 2, 3]);
observable.subscribe(subject);

// Logs:
// observerA: 1 | observerB: 1 | observerA: 2 |
// observerB: 2 | observerA: 3 | observerB: 3
```

BehaviorSubject

- Es lo mismo que una Subject pero cuya peculiaridad es que tiene conocimiento de cual es el valor actual y se lo entrega a cada nuevo subscriptor en cuanto ejecuta la función subscribe
- Es muy útil para diversas situaciones cuando uso RxJS

BehaviorSubject. Cómo se usa?

```
const subject = new BehaviorSubject(0); // 0 es el valor inicial

// Nada más suscibirse recibe el valor inicial
subject.subscribe({
  next: (v) => console.log(`observerA: ${v}`)
});
subject.next(1);
subject.next(2);
subject.subscribe({
  next: (v) => console.log(`observerB: ${v}`)
});
subject.next(3);

// Logs
// observerA: 0 | observerA: 1 | observerA: 2 |
// observerB: 2 | observerA: 3 | observerB: 3
```

Operators

- Son funciones que permiten manipular los resultados emitidos por los Observables
- Tenemos funciones de creación de Observables, condicionales, combinación, filtrado, transformación, ...

Operators

Esta función genera valores para los cuadrados de los números impares

```
const nums$ = of(1, 2, 3, 4, 5).pipe(  
  filter((n: number) => n % 2 !== 0),  
  map(n => n * n)  
);
```

```
nums$.subscribe(x => console.log(x));
```

El operador pipe es una función de un Observable RxJS que habilita realizar composición de operadores

Operators

- Creation: from, fromEvent, of, ...
- Combination: combineLatest, concat, merge, startWith , withLatestFrom, zip, ...
- Filtering: debounceTime, distinctUntilChanged, filter, take, takeUntil, ...
- Transformation: bufferTime, concatMap, map, mergeMap, scan, switchMap, ...
- Utility: tap
- Multicasting: share

Observables en Angular

- EventEmitter extiende la clase Observable
- El módulo HTTP usa observables para la gestión de peticiones y respuestas
- El módulo del Router y de Forms usan observables para escuchar y responder a eventos del usuario

EventEmitter

```
@Component({
  selector: 'zippy',
  template: `
<div class="zippy">
  <div (click)="toggle()">Toggle</div>
  <div [hidden]="!visible"><ng-content></ng-content></div>
</div>`)
export class ZippyComponent {
  visible = true;
  @Output() open = new EventEmitter<any>();
  @Output() close = new EventEmitter<any>();

  toggle() {
    this.visible = !this.visible;
    if (this.visible) { this.open.emit(null); }
    else { this.close.emit(null); }
  }
}
```

Peticiones HTTP

Las ventajas que proporciona sobre el uso de Promises son:

- Observables no modifican la respuesta del servidor por lo que se pueden aplicar operadores para transformarla a nuestro gusto
- Las peticiones HTTP se pueden cancelar a través de unsubscribe()
- Se puede configurar para comprobar el progreso de la petición y obtener feedback
- Las peticiones que fallan se pueden reintentar fácilmente

AsyncPipe

El pipe async permite la subscripcion a un Observable

```
@Component({  
  selector: 'async-observable-pipe',  
  template: `<div><code>observable|async</code>:  
    Time: {{ time | async }}</div>`  
})  
export class AsyncObservablePipeComponent {  
  time = new Observable(observer =>  
    setInterval(() => observer.next(new Date().toString()), 1000)  
  );  
}
```

Router

Router.events proporciona eventos de navegación como observables. Se pueden filtrar para buscar el que necesito

```
@Component({
  selector: 'app-routable',
  templateUrl: './routable.component.html',
  styleUrls: ['./routable.component.css']
})
export class Routable1Component implements OnInit {
  navStart: Observable<NavigationStart>;
  constructor(private router: Router) {
    this.navStart = router.events.pipe(
      filter(evt => evt instanceof NavigationStart)
    ) as Observable<NavigationStart>;
  }

  ngOnInit() {
    this.navStart.subscribe(evt => console.log('Navigation Started!'));
  }
}
```

Router

ActivatedRoute es un servicio de Angular que se basa en propiedades observables para obtener información acerca del path, parametros, querystring, ... de cada ruta navegada

```
@Component({
  selector: 'app-routable',
  templateUrl: './routable.component.html',
  styleUrls: ['./routable.component.css']
})
export class Routable2Component implements OnInit {
  constructor(private activatedRoute: ActivatedRoute) {}

  ngOnInit() {
    this.activatedRoute.url
      .subscribe(url => console.log('The URL changed to: ' + url));
  }
}
```

Reactive Forms

Los formularios contienen una serie de propiedades como observables para monitorizar los valores y las modificaciones en los mismos

FormControl proporciona propiedades como valueChanges o statusChanges que nos dan eventos de cambio en los valores de los inputs

```
@Component({
  selector: 'my-component',
  template: 'MyComponent Template'
})
export class MyComponent implements OnInit {
  nameChangeLog: string[] = [];
  heroForm: FormGroup;
  ngOnInit() {
    this.logNameChange();
  }
  logNameChange() {
    const nameControl = this.heroForm.get('name');
    nameControl.valueChanges.forEach(
      (value: string) => this.nameChangeLog.push(value)
    );
  }
}
```

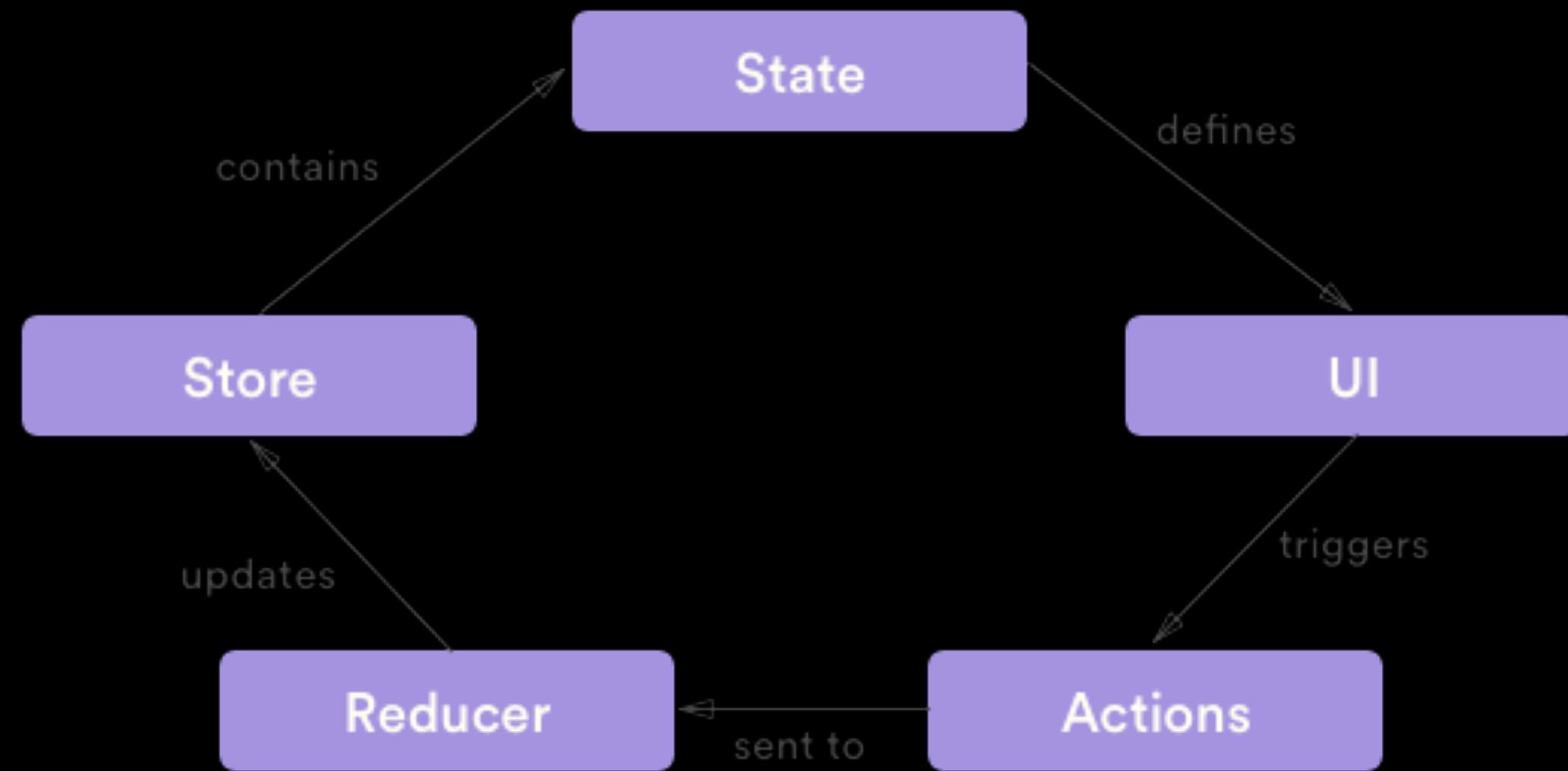
Recursos

RxJs API

Rx Visualizer

Rx Marbles

Redux



Store

- Un Store es el contenedor del estado de nuestra aplicación
- Sólo se puede modificar mediante dispatch de acciones

State

- El State es el estado actual de los datos de nuestra aplicación en un momento del tiempo

Actions

- Contienen información acerca de las modificaciones que queremos realizar en un Store
- Objetos formados por dos propiedades (... o no 😊)

```
{  
  type: ACTION_TYPE,  
  payload: {}  
}
```

Action creators

- Funciones que se encargan de crear un Action

```
function GetPost(postId) {  
  return {  
    type: GET_POSTS,  
    postId  
  }  
}
```

Reducers

- Funciones puras que reciben el estado actual de la aplicación más un Action para devolver un nuevo estado
- El nuevo estado ha de ser **IMMUTABLE**

Reducers

```
function myReducer(state, action) {
  switch (action.type) {
    case GET_POSTS:
      return {
        loading: true
      }
    case GET_POSTS_SUCCESS:
      return {
        ...state,
        loading: false,
        posts: action.posts
      });
    case GET_POSTS_ERROR:
      return {
        loading: false
      }
    default:
      return state
  }
}
```

Selectors

- Un Selector es una función de utilidad que nos permite seleccionar fragmentos del estado de nuestra aplicación
- Lo usaremos con nuestras vistas
- Proporcionan memoizing => Mismos parámetros misma salida sin ejecución

Selectors

```
export const totalSelector = createSelector(  
  subtotalSelector,  
  taxSelector,  
  (subtotal, tax) => ({ total: subtotal + tax })  
)
```

ngxs-store

Porqué ngxs?

- Es simple. ngrx/store y ngrx/effects generan demasiado boilerplate
- Puedo utilizar RxJS pero no es totalmente necesario un conocimiento profundo del framework
- Encaja perfectamente con Angular
- Mejor experiencia de usuario

Instalación

```
$ npm install  
  @ngxs/store  
  @ngxs/devtools-plugin  
  @ngxs/logger-plugin  
  @ngxs/router-plugin --save
```

Bootstrap en AppModule

```
@NgModule({  
  ...  
  imports: [  
    ...  
    NgxsReduxDevtoolsPluginModule.forRoot({  
      disabled: environment.production  
    }),  
    NgxsLoggerPluginModule.forRoot({ logger: console, collapsed: false }),  
    NgxsRouterPluginModule.forRoot(),  
    NgxsModule.forRoot([], { developmentMode: !environment.production }),  
    ...  
  ]  
  ...  
})  
export class AppModule {}
```

Bootstrap en feature modules

```
NgxsModule.forRoot([PostState])
```

Como funciona ngxs/store

Diagrama de flujo ngxs

Actions

```
export class Login {  
    static readonly type = '[Auth] Login';  
    constructor(public loginRequest: LoginRequestModel) {}  
}
```

```
export class LoginSuccess {  
    static readonly type = '[Auth] LoginSuccess';  
    constructor(public loginResponse: LoginResponseModel) {}  
}
```

```
export class LoginFailed {  
    static type = '[Auth] LoginFailed';  
    constructor(public errors: ErrorModel[] ) {}  
}
```

Dispatching Actions

El dispatch devuelve un Observable por lo que me puedo subscribir

```
constructor(private store: Store) {}  
...  
this.store.dispatch(new Login(form.value))  
.subscribe(() => this.form.reset());
```

State

```
@State<ErrorModel[]>({  
  name: 'errors',  
  defaults: []  
})  
export class ErrorState {  
  constructor() {}  
  
  @Action(SetErrors)  
  setErrors({ setState }: StateContext<ErrorModel[]>, { errors }: SetErrors) {  
    setState(errors);  
  }  
  
  @Action(ResetErrors)  
  resetErrors({ setState }: StateContext<ErrorModel[]>) {  
    setState([]);  
  }  
}
```

Async State

```
export interface AuthStateModel {  
    refreshToken: string;  
    accessToken: string;  
    uuid: string;  
    email: string;  
}  
  
@State<AuthStateModel>({  
    name: 'auth',  
    defaults: {  
        currentUser: null  
    }  
})  
export class AuthState {  
    constructor(private store: Store, private authService: AuthService) {}  
  
    // ngxs will subscribe to the post observable for you if you return it from the action  
    @Action(Login)  
    login({ dispatch }: StateContext<AuthStateModel>, action: Login) {  
        return this.authService.login(action.login).pipe(  
            tap(data => dispatch(new LoginSuccess(data))),  
            catchError(error => dispatch(new LoginFailed(error.error)))  
        );  
    }  
}
```

Select

```
export class HomeComponent implements OnInit {  
  @Select(PostState) posts$: Observable<PostViewModel[]>;  
  @Select(state => state.posts) otherWayToGetPosts$: Observable<PostViewModel[]>;  
  
  constructor(private store: Store) {}  
  
  ngOnInit() {  
    this.store.dispatch(new GetPosts());  
  }  
}
```

Memoized Selectors

- El término memoizing indica un tipo de función que cachea y devuelve el mismo valor siempre que los parámetros de entrada no se modifiquen. Es un concepto ampliamente utilizado con las librerías de State Management
- ngxs nos proporciona un decorator `@Selector` para realizar memoizing de partes de nuestro estado

Selectors

```
export class PostState {  
  
    @Selector()  
    static postFromPaul(state: PostViewModel[]){  
        return state.filter(p => p.name === 'Paul');  
    }  
}  
  
...  
  
@Select(PostState.postFromPaul) posts$: Observable<PostViewModel[]>;
```

Dynamic Selectors

```
@State<PostViewModel[]>({  
    name: 'posts',  
    defaults: []  
})  
export class PostState {  
    static postByType(type: string) {  
        return createSelector([PostState], (state: PostViewModel[]) => {  
            return state.filter(p => p.type(type) > -1);  
        });  
    }  
}  
...  
@Select(PostState.postByType('images')) postsByType$: Observable<PostViewModel[]>;
```

Seleccionando de múltiples estados

```
@State<PostViewModel[]>({ ... })  
export class PostState {  
  @Selector([AuthState])  
  static postsFromCurrentUser(state: PostViewModel[], authState: Auth) {  
    return state.filter(p => p.user.id === authState.currentUser.id);  
  }  
}
```