Angular Avanzado

DIRECTIVAS

Introducción

- Las directivas son marcas en los elementos del árbol DOM, en los nodos del HTML, que indican al Angular que debe asignar cierto comportamiento a dichos elementos o transformarlos según corresponda.
- La recomendación es que:

  El único sitio donde se debe manipular el árbol DOM es en las directivas, para que entre dentro del ciclo de vida de compilación, binding y renderización del HTML que sigue el Angular.
- Las directivas son clases decoradas con @Directive.
- Podríamos decir que las directivas nos permiten añadir comportamiento dinámico al árbol DOM, haciendo uso de las directivas propias del Angular o extender la funcionalidad hasta donde necesitemos creando las nuestras propias directivas, en las plantillas utilizando la sintaxis de los atributos o etiquetas del HTML.

Registrar

- Módulos:

```
@NgModule({
  …
  declarations: [ … , MyDirective ],
  …
  exports: [ MyDirective ] // En los módulos de características
})
export class AppModule { }
```

- Componentes independientes:

```
@Component({
  …
  standalone: true,
  imports: [ … , MyDirective ],
  …
})
export class AppComponent { … }
```

Características de las directivas

- Las directivas se clasifican en tres tipos:
  - directivas atributos: que alteran la apariencia o comportamiento de un elemento existente.
  - directivas estructurales: que alteran el diseño mediante la adición, eliminación y sustitución de elementos en DOM.
  - componentes: son técnicamente directivas-con-plantilla, aunque conceptualmente son completamente diferentes.
- Prefijos en Directivas
  - Se recomienda que las directivas tengan siempre un prefijo de forma que no choquen con otras directivas creadas por otros desarrolladores o con actuales o futuras etiquetas de HTML. Por eso las directivas de Angular empiezan siempre por “ng”.
- Nombre
  - El nombre de las directivas utiliza la notación camelCase que sigue el formato de los identificadores en JavaScript.

@Directive

- **selector**: Selector CSS que indica a Angular que debe ejecutar la directiva cuando se encuentra un elemento con ese selector en el HTML generado.
- **providers**: Lista de los proveedores disponibles para esta directiva.
- **inputs**: Lista de nombres de las propiedades que se enlazan a datos con las entradas del componente.
- **outputs**: Lista de los nombres de las propiedades de la clase que exponen a los eventos de salida que otros pueden suscribirse.
- **exportAs**: nombre bajo el cual se exporta la instancia de la directiva para asignarlas en las variables referencia en las plantillas.
- **host**: enlace de eventos, propiedades y atributos del elemento host con propiedades y métodos de la clase.
- **queries**: configura las consultas que se pueden inyectar en el componente

Directiva personalizada

```
import { Directive, Input, Output, HostListener, EventEmitter, HostBinding } from '@angular/core';

@Directive({  selector: `[winConfirm]` })
export class WindowConfirmDirective {
  @Input() winConfirmMessage = '¿Seguro?';
  @Output() winConfirm: EventEmitter<any> = new EventEmitter();
  @HostBinding('class.pressed') isPressed: boolean = false;

  @HostListener('click', ['$event'])
  confirmFirst() {
    if (window.confirm(this.winConfirmMessage)) {
      this.winConfirm.emit(null);
    }
  }
  @HostListener('mousedown') hasPressed() { this.isPressed = true; }
  @HostListener('mouseup') hasReleased() { this.isPressed = false; }
}
@Directive({ selector: '[show]' })
export class ShowDirective {
  @HostBinding('hidden') hidden: boolean = false;
  @Input('myShow') set show(value: boolean) { this.hidden = !value; }
}
<button (winConfirm)="vm.delete(p.id)" winConfirmMessage="¿Estás seguro?" [show]="isValid">Borrar</button>
```


Selector

- El selector CSS desencadena la creación de instancias de una directiva.
- Angular sólo permite directivas que se desencadenen sobre los selectores CSS que no crucen los límites del elemento sobre el que están definidos.
- El selector se puede declarar como:
  - element-name: nombre de etiqueta.
  - .class: nombre de la clase CSS que debe tener definida la etiqueta.
  - [attribute]: nombre del atributo que debe tener definida la etiqueta.
  - [attribute=value]: nombre y valor del atributo definido en la etiqueta.
  - [attribute][ngModel]: nombre de los atributos que debe tener definida la etiqueta, condicionado a que aparezca el segundo atributo.
  - :not(sub_selector): Seleccionar sólo si el elemento no coincide con el sub_selector.
  - selector1, selector2: Varios selectores separados por comas.

Atributos del selector

- Los atributos del selector de una directiva se comportan como los atributos de las etiquetas HTML, permitiendo personalizar y enlazar a la directiva desde las plantillas.
- Propiedades de entrada: con {required: true} son obligatorias, con {transform: booleanAttribute | numberAttribute} evita interpretar todos los atributos como cadenas.

```
@Input() init: string;
<my-comp myDirective [init]="1234"></my-comp>
```

- Eventos de salida

```
@Output() updated: EventEmitter<any> = new EventEmitter();
this.updated.emit(value);
<myDirective (updated)="onUpdated($event)"></myDirective>
```

- Propiedades bidireccionales: Es la combinación de una propiedad de entrada y un evento de salida con el mismo nombre (el evento obligatoriamente con el sufijo Change):

```
@Input() size: number | string;
@Output() sizeChange = new EventEmitter<number>();
```


Directiva atributo con valor

- La presencia de la directiva como etiqueta o como atributo dispara la ejecución de la misma.
- En algunos casos, con la notación atributo, es necesario asignar un valor que la personalice.
- Como propiedad de entrada:

```
@Input('myDirective') valor: string;
```

- Inyectado en el constructor:

```
constructor(@Attribute('myDirective') public valor: string) {}
```

- Se enlaza como cualquier otra propiedad:

```
<div [myDirective]="myValue" (myDirective)="click()" >
```

Eventos del DOM

- Se podría llegar al DOM con JavaScript estándar y adjuntar manualmente los controladores de eventos pero hay al menos tres problemas con este enfoque:
  - Hay que asociar correctamente el controlador.
  - El código debe desasociar manualmente el controlador cuando la directiva se destruye para evitar fugas de memoria.
  - Interactuar directamente con API DOM no es una buena práctica.
- Mediante el decorador @HostListener de @angular/core se puede asociar un evento a un método de la clase:

```
@HostListener('mouseenter', ['$event'])
onMouseEnter(e: any) { … }
```

- Se pueden asociar eventos de window, document y body:

```
@HostListener('document:click', ['$event'])
handleClick(event: Event) { … }
```


Vinculación de propiedades

- Podemos usar directivas de atributos que afecten el valor de las propiedades en el nodo del host usando el decorador @HostBinding.
- El decorador @HostBinding permite programar un valor de propiedad en el elemento host de la directiva.
- Funciona de forma similar a una vinculación de propiedades definida en una plantilla, excepto que se dirige específicamente al elemento host.
- La vinculación se comprueba para cada ciclo de detección de cambios, por lo que puede cambiar dinámicamente si se desea.

```
@HostBinding('class.pressed') isPressed: boolean = false;
@HostListener('mousedown') hasPressed() { this.isPressed = true; }
@HostListener('mouseup') hasReleased() { this.isPressed = false; }
```

Acceso directo al DOM

- El servicio ElementRef permite el acceso directo al elemento DOM a través de su propiedad nativeElement.
- Solo se debe usar como el último recurso cuando se necesita acceso directo al DOM:
  - puede hacer la aplicación más vulnerables a los ataques XSS.
  - crea acoplamiento entre la aplicación y las capas de renderizado imposibilitando su separación para el uso Web Workers o SSR.
- El servicio Renderer2 proporciona un API que se puede utilizar con seguridad incluso cuando el acceso directo a elementos nativos no es compatible.

```
@Directive({ selector: '[myShadow]' })
export class ShadowDirective {
  constructor(el: ElementRef, renderer: Renderer2) {
    //el.nativeElement.style.boxShadow = '10px 10px 5px #888888';
    renderer.setStyle(el.nativeElement, 'box-shadow', '10px 10px 5px #888888');
  }
}
```


Directivas estructurales

- El elemento <ng-template> define una plantilla que no se representa de forma predeterminada.
- Con <ng-template>, se puede definir el contenido de una plantilla que Angular solo representa cuando, ya sea directa o indirectamente, se le indica específicamente que lo haga, lo que permite definir fragmentos en la plantilla del componente y tener un control total sobre cómo y cuándo se muestra el contenido. Si se envuelve el contenido dentro de un <ng-template> sin indicarle a Angular que lo renderice, dicho contenido no aparecerá nunca en una página.
- Uno de los principales usos de <ng-template> es mantener el contenido de la plantilla que será utilizado por las directivas estructurales. Las directivas estructurales son directivas aplicadas a un elemento <ng-template> que reproducen de forma condicional o reiterada el contenido de dicho elemento. Estas directivas pueden agregar y eliminar copias del contenido de la plantilla según su propia lógica.
- Angular admite una sintaxis abreviada para directivas estructurales que evita la necesidad de crear explícitamente un elemento <ng-template>. Las directivas estructurales se pueden aplicar directamente a un elemento anteponiendo un asterisco (*) al selector de atributos de directiva, como *ngFor o *nglf.

```
<li *ngFor="let provincia of provincias; trackBy: trackByFn">{{provincia.nombre}}</li>
```

- Angular transforma el asterisco que se encuentra delante de una directiva estructural en un <ng-template> que alberga la directiva y rodea al elemento y sus descendientes.

```
<ng-template ngFor let-provincia [ngForOf]="provincias" [ngForTrackBy]="trackByProvincias">
  <li>{{provincia.nombre}}</li>
</ng-template>
```

Microsintaxis de las directivas estructurales

- La microsintaxis de Angular permite configurar una directiva en una cadena compacta y fácil de usar:
  :prefix="(:let | :expression) (';' | ',')? (:let | :as | :keyExp)"
  let = "let" :local "=" :export ";"?
  as = :export "as" :local ";"?
  keyExp = :key ":"? :expression ("as" :local)? ";"?
- El analizador de microsintaxis traduce esa cadena en atributos en <ng-template>:
  *ngIf="exp" → <ng-template [ngIf]="exp">
  *ngFor="let item of source as items; trackBy: myTrack; index as i" →
  <ng-template ngFor let-item [ngForOf]="source" let-items="ngForOf" [ngForTrackBy]="myTrack" let-i="index">
- El let declara una variable de plantilla item, crea la propiedad let- y, si se realiza asignación, se vincula al export o, si no, se vincula a la propiedad $implicit de la directiva. La sintaxis as es una alternativa a let.
- Una expression es un par clave-expresión "from source", donde "from" es una clave de enlace y source es una expresión regular de plantilla. Las claves de enlace se asignan a propiedades de la directiva prefijadas por el selector de directiva estructural, transformándolas a camelCase y se vincula a la expresión source.

Directivas estructurales

- Las directivas estructurales son responsables del diseño HTML. Le dan forma o le cambian la estructura al arbol DOM, generalmente agregando, eliminando o manipulando elementos. Al igual que con otras directivas, se aplica a un elemento host. Despueas, la directiva hace lo que se supone que debe hacer con ese elemento host y sus descendientes.
- La gran diferencia con las directivas atributo es que, debido a la naturaleza de las directivas estructurales vinculadas a una plantilla, tenemos acceso a TemplateRef, un objeto que representa al elemento <ng-template> a la que se adjunta la directiva (elementRef), y a ViewContainerRef, que gestiona las vistas del contenido.

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

- Una directiva estructural sencilla crea una vista incrustada para el <ng-template> generado por Angular e inserta la vista en el contenedor de vistas adyacente al elemento host que contiene la directiva.
- Solo se puede aplicar una directiva estructural por elemento debido a que solo hay un <ng-template> en el que se desenvuelve esa directiva.

Directivas estructurales

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';
@Directive({ selector: '[myUnless]'})
export class UnlessDirective {
  private hasView = false;

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

  @Input() set myUnless(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;
    }
  }
}
```

```
<div *myUnless="isValid" >
```


Contenedores de vistas

- Los ViewContainer son contenedores en los que se pueden adjuntar una o más vistas.
- Las vistas representan algún tipo de diseño que se va a representar y el contexto en el que se debe procesar.
- Los ViewContainer están anclados a los componentes y son responsables de generar su salida, de modo que esto significa que el cambio de las vistas que se adjuntan al ViewContainer afecta a la salida final del componente.
- Se pueden adjuntar dos tipos de vistas a un contenedor de vistas: vistas de host que están vinculadas a un componente y vistas incrustadas vinculadas a una plantilla.
- Las directivas estructurales interactúan con las plantillas por lo que utilizan vistas incrustadas.

Contenedores de vistas

createEmbeddedView(templateRef: TemplateRef<C>, context?: C, index?: number): EmbeddedViewRef<C>

- Instancia una vista incrustada basada en `templateRef`, asociada a un contexto y la inserta en el contenedor en la posición especificada por el índice.
- Si no se especifica el índice, la nueva vista se insertará como la última vista del contenedor.

```
this.viewContainer.createEmbeddedView(this.templateRef);
```

- Se puede adjuntar un objeto de contexto que debe ser un objeto de claves/valor donde las claves estarán disponibles para la vinculación por las declaraciones de plantilla local (la clave $implicit en el objeto de contexto establecerá el valor predeterminado para las claves no existentes).

clear(): Destruye todas las vistas del contenedor.

```
this.viewContainer.clear();
```


API de composición de directivas (v15)

- La API de composición de directivas permite aplicar directivas al elemento host de un componente desde dentro del propio componente o directiva, siendo una poderosa estrategia de reutilización de código. La API de composición de directivas solo funciona con directivas autónomas.
- La composición de directivas se aplica agregando una propiedad hostDirectives al decorador @Directive (o @Component). Acepta una serie de directivas autónomas y las aplicará en el componente host.

```
@Directive({...})
export class Menu { }

@Directive({...})
export class Tooltip { }

// MenuWithTooltip puede componer comportamientos a partir de otras directivas
@Directive({
  hostDirectives: [Tooltip, Menu],
})
export class MenuWithTooltip { }
```

API de composición de directivas (v15)

- Las entradas y salidas de las directivas de host no se incluyen en la API de su componente de forma predeterminada. Se pueden incluir explícitamente las entradas y salidas en la API de su componente expandiendo la entrada (se pueden crear alias de entradas y salidas) en la propiedad hostDirectives:

```
@Component({
  selector: 'admin-menu',
  template: 'admin-menu.html',
  hostDirectives: [{
    directive: MenuBehavior,
    inputs: ['menuId'],
    outputs: ['menuClosed: closed'],
  }],
})
export class AdminMenu { }
<admin-menu menuId="top-menu" (closed)="logMenuClosed()">
```


Validación personalizada

- A través de las directivas se pueden implementar validaciones personalizadas para los formularios dirigidos por plantillas.
- Deben implementar el método validate del interface Validator.

  validate(control: AbstractControl): { [key: string]: any } { … }
- El método recibe el AbstractControl al que se aplica la validación, usando la propiedad control.value se accede al valor a validar.
- El método devuelve null si el valor valido (el valor vacío se considera siempre valido salvo en el required) o un array de claves/valor. El resultado se mezcla con los resultados del resto de validaciones en la colección AbstractControl.errors, donde las claves sirven para identificar los errores producidos.
- La validación debería implementarse mediante la invocación de una función externa con la misma firma que el método para favorecer su reutilización en los formularios reactivos.

Validación personalizada

- El selector debería incluir los atributos [formControlName], [formControl] y [ngModel] para limitar el ámbito de selección.
- Angular, como mecanismo interno para ejecutar los validadores en un control de formulario, mantiene un proveedor múltiple de dependencias (array de dependencias) denominado NG_VALIDATORS. Utilizando la propiedad multi: true se indica que acumule la dependencia en un array (proveedor múltiple) en vez de sustituir el anterior.
- Todos los validadores predefinidos ya se encuentran agregados a NG_VALIDATORS. Así que cada vez que Angular instancia un control de formulario, para realizar la validación, inyecta las dependencias de NG_VALIDATORS, que es la lista de todos los validadores, y los ejecuta uno por uno con el control instanciado.
- Es necesario registrar los nuevos validadores en NG_VALIDATORS como providers en @Directive.

Validación personalizada

```
import { Directive } from '@angular/core';
import { Validator, AbstractControl, NG_VALIDATORS, ValidatorFn } from '@angular/forms';

export function naturalNumberValidator(): ValidatorFn {
  return (control: AbstractControl): {[key: string]: any} => {
    return /^[1-9]\d*$/.test(control.value) ? null : { naturalNumber: { valid: false } };
  };
}
@Directive({
    selector: '[naturalNumber][formControlName],[naturalNumber][formControl],[naturalNumber][ngModel]',
    providers: [{ provide: NG_VALIDATORS, useExisting: NaturalNumberValidatorDirective, multi: true }]
})
export class NaturalNumberValidatorDirective implements Validator {
    validate(control: AbstractControl): { [key: string]: any } {
        if (control.value) {
          return naturalNumberValidator()(control);
        }
        return null;
    }
}
```

```
<input type="text" name="edad" id="edad" [(ngModel)]="model.edad" #edad="ngModel" naturalNumber>
<output class="errorMsg" [hidden]="!edad?.errors?.naturalNumber">No es un número entero positivo.</output>
```

Validaciones personalizadas

```
export function NIFValidator(): ValidatorFn {
 return (control: AbstractControl): { [key: string]: any } | null => {
    if (!control.value) { return null; }
    const err = { nif: { invalidFormat: true, invalidChar: true } };
    if (/^\d{1,8}\w$/.test(control.value)) {
       const letterValue = control.value.substr(control.value.length - 1);
       const numberValue = control.value.substr(0, control.value.length - 1);
       err.nif.invalidFormat = false;
       return letterValue.toUpperCase() === 'TRWAGMYFPDXBNJZSQVHLCKE'.charAt(numberValue % 23) ? null : err;
    } else { return err; }
 };
}
@Directive({
 selector: '[nif][formControlName],[nif][formControl],[nif][ngModel]',
 providers: [{ provide: NG_VALIDATORS, useExisting: NIFValidatorDirective, multi: true }]
})
export class NIFValidatorDirective implements Validator {
 validate(control: AbstractControl): ValidationErrors | null {
    return NIFValidator()(control);
 }
}
<input type="text" name="dni" [(ngModel)]="model.dni" #dni="ngModel" nif>
<div *ngIf="dni?.errors?.nif">No es un NIF valido.</div>
```


Validaciones personalizadas

```
@Directive({
 selector: '[type][formControlName],[type][formControl],[type][ngModel]',
 providers: [
    { provide: NG_VALIDATORS, useExisting: forwardRef(() => TypeValidatorDirective), multi: true }
 ]
})
export class TypeValidatorDirective implements Validator {
 constructor(private elem: ElementRef) { }
 validate(control: AbstractControl): ValidationErrors | null {
    const valor = control.value;
    if (valor) {
      const dom = this.elem.nativeElement;
      if (dom.validity) { // dom.checkValidity();
        return dom.validity.typeMismatch ? { 'type': dom.validationMessage } : null;
      }
    }
    return null;
 }
}

<input type="url" id="foto" name="foto" [(ngModel)]="elemento.foto" #foto="ngModel">
<output class="error" [hidden]="!foto.errors?.type">{{foto.errors.type}}</output>
```

Validaciones personalizadas

```
import { Directive, forwardRef, Attribute } from '@angular/core';
import { Validator, ValidationErrors, AbstractControl, NG_VALIDATORS } from '@angular/forms';

@Directive({
  selector: '[equalsTo][formControlName],[equalsTo][formControl],[equalsTo][ngModel]',
  providers: [ { provide: NG_VALIDATORS, useExisting: forwardRef(() => EqualsValidator), multi: true } ]
})
export class EqualsValidator implements Validator {
  constructor( @Attribute('validateEqual') public validateEqual: string) {}
  validate(control: AbstractControl): ValidationErrors | null {
     let valor = control.value;
     let cntrlBind = control.root.get(this.validateEqual);

     if (valor) {
       return (!cntrlBind || valor !== cntrlBind.value) ? { 'equalsTo': `${valor} <> ${cntrlBind?.value}` } : null;
     }
     return null;
  }
}
```

```
<input type="password" ... #pwd="ngModel">
<input type="password" name="valPwd" id="valPwd" [(ngModel)]="valPwd" #valPwd="ngModel" equalsTo="pwd">
<div *ngIf="valPwd?.errors?.equalsTo">Las dos claves deben ser iguales.</div>
```


validator.js

https://github.com/validatorjs/validator.js

- Una de las bibliotecas de validadores y desinfectantes de cadenas mas populares. Los valores deben convertirse en cadena si es necesario.
  - npm install validator
- Para importar todo el módulo:
  import validator from 'validator';
- Para importar solo un subconjunto de la biblioteca:
  import isEmail from 'validator/lib/isEmail';
- Para realizar la validación:
  if(validator.isEmail('demo@example.com'))

Shared Component

- Aunque un componente es técnicamente una directiva-con-plantilla, conceptualmente son dos elementos completamente diferentes.
- Los componentes se pueden clasificar como:
  - Componentes de aplicación: Son componentes íntimamente ligados a la aplicación cuya existencia viene determinada por la existencia de la aplicación. Estos componentes deben favorecer el desacoplamiento de entre la plantilla y la clase del componente.
  - Componentes compartidos: Son componentes de bajo nivel, que existen para dar soporte a los componentes de aplicación pero son independientes de aplicaciones concretas. Estos componentes requieren un mayor control sobre la plantilla por los que el acoplamiento será mayor, así como la dependencia entre ellos.

Dynamic Component

- Las plantillas de los componentes no siempre son fijas. Es posible que una aplicación necesite cargar nuevos componentes en tiempo de ejecución.
- Angular dispone del servicio ComponentFactoryResolver para cargar componentes dinámicamente.

```
@Component({
  selector: 'dynamyc-template',
  template: `<ng-template my-host></ng-template>`
})
export class DynamicComponent implements AfterViewInit {
  @ViewChild(MyHostDirective) myHost: MyHostDirective;
  constructor(private componentFactoryResolver: ComponentFactoryResolver) { }
  ngAfterViewInit() { this.loadComponent(); }
  loadComponent() {
    let componentFactory = this.componentFactoryResolver.resolveComponentFactory(DemoTemplateComponent);
    let viewContainerRef = this.myHost.viewContainerRef;
    viewContainerRef.clear();
    let componentRef = viewContainerRef.createComponent(componentFactory);
  }
}
```

ng-container

- La directiva <ng-container> es un elemento de agrupación que proporciona un punto donde aplicar las directivas sin interferir con los estilos o el diseño porque Angular no la refleja en el DOM.
- Una parte del párrafo es condicional:

```
<p>I turned the corner
  <ng-container *ngIf="hero"> and saw {{hero.name}}. I waved</ng-container>
  and continued on my way.</p>
```

- No se desea mostrar todos los elemento como opciones:

```
<select [(ngModel)]="hero">
  <ng-container *ngFor="let h of heroes">
    <ng-container *ngIf="showSad || h.emotion !== 'sad'">
      <option [ngValue]="h">{{h.name}} ({{h.emotion}})</option>
    </ng-container>
  </ng-container>
</select>
```


*ngComponentOutlet

- Permite instanciar e insertar un componente el la vista actual, proporcionando un enfoque declarativo a la creación de componentes dinámicos.

```
<ng-container *ngComponentOutlet="componentTypeExpression;
 injector: injectorExpression; content: contentNodesExpression; ngModuleFactory:
moduleFactory;"></ng-container>
```

- Se puede controlar el proceso de creación de componentes mediante los atributos opcionales:
  - injector: inyector personalizado que sustituye al del contenedor componente actual.
  - content: Contenido para transcluir dentro de componente dinámico.
  - moduleFactory: permite la carga dinámica de otro módulo y, a continuación, cargar un componente de ese módulo.
- El componentTypeExpression es la clase del componente que debe ser expuesta desde el contexto:

```
<ng-container *ngComponentOutlet="componente"></ng-container>
public componente: Type<any> = MyComponent;
```

*ngTemplateOutlet

- Inserta una vista incrustada desde un TemplateRef preparado

```
@Component({
  selector: 'ng-template-outlet-example',
  template: `
    <ng-container *ngTemplateOutlet="greet"></ng-container>
    <hr>
    <ng-container *ngTemplateOutlet="eng; context: myContext"></ng-container>
    <hr>
    <ng-container *ngTemplateOutlet="svk; context: myContext"></ng-container>
    <hr>

    <ng-template #greet><span>Hello</span></ng-template>
    <ng-template #eng let-name><span>Hello {{name}}!</span></ng-template>
    <ng-template #svk let-person="localSk"><span>Ahoj {{person}}!</span></ng-template>
`
})
export class NgTemplateOutletExample {
  myContext = {$implicit: 'World', localSk: 'Svet'};
}
```


ngTemplateOutletContext

- Enlaza una vista incrustada ngTemplateOutlet con un contexto de datos

```
@Component({
  selector: 'demo-template',
  template: `
    <ng-template #templateRef let-label="label" let-url="url">
      <div><a href="{{url}}">{{label}} *</a></div>
    </ng-template>
    <h1>Menu</h1>
    <div [ngTemplateOutlet]="templateRef" [ngTemplateOutletContext]="menu[0]"></div>
    <div [ngTemplateOutlet]="templateRef" [ngTemplateOutletContext]="menu[1]"></div>
  `
})
export class DemoTemplateComponent {
  menu:any = [
    { "id": 1, "label": "AngularJS", "url": "http:\/\/angularjs.org" },
    { "id": 2, "label": "Angular", "url": "http:\/\/angular.io" }
    ];
}
```

ng-content

- La directiva <ng-content> es un elemento de transclusión (trasladar e incluir) del contenido de la etiqueta del componente a la plantilla.

```
<card [header]="'Hola mundo'">Este es el contenido a transcluir que puede contener otras etiquetas y
    componentes</card>

@Component({
 selector: 'card',
 template: `
<div class="card">
   <H1>{{ header }}</H1>
   <ng-content></ng-content>
</div>`,
})
export class CardComponent {
   @Input() header: string = 'this is header';
```


ng-content

- <ng-content> acepta un atributo select que permite definir un selector CSS que indica el contenido que se muestra en la directiva.
- Con el uso del atributo select se puedan definir varios ng-content, uno por cada tipo de contenido.
- Selector asociado a un atributo de HTML:

```
<ng-content select="[titulo]"></ng-content>
```

- Selector asociado a un class de CSS:

```
<ng-content select=".cuerpo"></ng-content>
```

- Contenido:

```
<card [header]="'Hola mundo'">
   <div titulo>Ejemplo</div>
   <div class="cuerpo">Este es el contenido a transcluir que puede contener otras etiquetas y
      componentes</div>
</card>
```

Controles de formulario personalizados

- Para que un componente pueda interactuar con ngModel como un control de formulario debe implementar el interfaz ControlValueAccessor.
- La interfaz ControlValueAccessor se encarga de:
  - Escribir un valor desde el modelo de formulario en la vista / DOM
  - Informar a otras directivas y controles de formulario cuando cambia la vista / DOM
- Los métodos a implementar son:
  - writeValue(obj: any): void
  - registerOnChange(fn: any): void
  - registerOnTouched(fn: any): void
  - setDisabledState(isDisabled: boolean)?: void
- Hay que registrar el nuevo control:
  { provide: NG_VALUE_ACCESSOR, useExisting: MyControlComponent, multi: true }

Consultas a la plantilla

- Desde la clase del componente se pueden realizar consultas a las plantillas con atributos decorados con @ViewChild, pero crea acoplamiento plantilla - clase del componente y no se pueden acceder antes de que se hayan ejecutado ngAfterViewInit o ngAfterViewChecked (en le constructor, primer ngOnChanges o ngOnInit).
- Se puede utilizar @ViewChild para obtener la primera etiqueta o directiva que coincida con el selector en la vista DOM. Si el DOM de la vista cambia y un nuevo hijo coincide con el selector, la propiedad se actualizará.
- Se admiten los siguientes selectores.
  - Cualquier clase con el decorador @Component o @Directive
  - Una variable de referencia de plantilla como una cadena (por ejemplo, consulta <my-component #cmp></my-component> con @ViewChild('cmp'))
  - Cualquier proveedor definido en el árbol de componentes secundarios del componente actual (por ejemplo, @ViewChild(SomeService) someService: SomeService) o definido a través de un token de cadena (por ejemplo , @ViewChild('someToken') someTokenVal: any)
  - A TemplateRef(por ejemplo, consulta <ng-template></ng-template>con @ViewChild(TemplateRef) template;)

```
miForm: NgForm;
@ViewChild('miForm') currentForm: NgForm;
ngAfterViewChecked() {
   if (this.currentForm !== this.miForm)  this.miForm = this.currentForm;
```

- Para consultar el contenido proyectado se utiliza @ContentChild , no antes de ngAfterContentInit o ngAfterContentChecked.

XSS

- Los scripts de sitios cruzados (XSS) permiten a los atacantes inyectar código malicioso en las páginas web. Tal código puede, por ejemplo, robar datos de usuario (en particular, datos de inicio de sesión) o realizar acciones para suplantar al usuario. Este es uno de los ataques más comunes en la web.
- Para bloquear ataques XSS, se debe evitar que el código malicioso entre al DOM (Modelo de Objetos de Documento). Por ejemplo, si los atacantes pueden engañarte para que insertes una etiqueta <script> en el DOM, pueden ejecutar código arbitrario en tu sitio web.
- El ataque no está limitado a las etiquetas <script>, muchos elementos y propiedades en el DOM permiten la ejecución del código, por ejemplo, <img onerror="...">. Si los datos controlados por el atacante ingresan al DOM, se generan vulnerabilidades de seguridad. <a href="javascript:...">
- Para bloquear sistemáticamente los errores XSS, por defecto Angular trata todos los valores como no confiables. Cuando se inserta un valor en el DOM desde una plantilla, mediante propiedad, atributo, estilo, enlace de clase o interpolación, Angular desinfecta y escapa de los valores no confiables.
- Las plantillas Angular son las mismas que las del código ejecutable: se confía en que el HTML, los atributos y las expresiones vinculantes (pero no los valores vinculados) en las plantillas sean seguros. Esto significa que las aplicaciones deben evitar que los valores que un atacante puedan controlar puedan convertirse en el código fuente de una plantilla. Nunca se debe generar código fuente de plantilla concatenando la entrada del usuario y de las plantillas.
- Para evitar estas vulnerabilidades, debe usarse el compilador de plantilla sin conexión, también conocido como inyección de plantilla

Contextos de saneamiento y seguridad

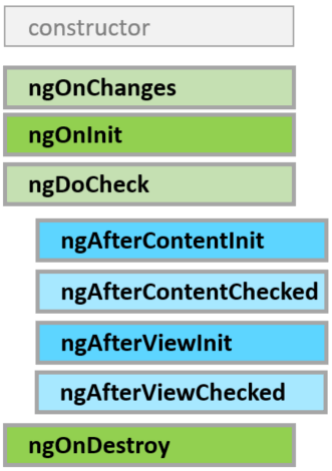
- La desinfección es la inspección de un valor que no es de confianza, convirtiéndolo en un valor que si es seguro insertar en el DOM. En muchos casos, la desinfección no cambia un valor en absoluto. La desinfección depende del contexto: un valor inofensivo en CSS es potencialmente peligroso en una URL.
- Angular define los siguientes contextos de seguridad:
  - HTML se usa cuando se interpreta un valor como HTML, por ejemplo, cuando se vincula a innerHtml.
  - El estilo se usa cuando se vincula CSS a la propiedad style.
  - La URL se usa para propiedades de URL como <a href>.
  - La URL es una URL de recurso que se cargará y ejecutará como código, por ejemplo, en <img src>.
- Angular desinfecta los valores que no son de confianza para HTML, estilos y URL. La desinfección de URL de recursos no es posible porque contienen código arbitrario.
- En modo de desarrollo, Angular imprime una advertencia en consola cuando tiene que cambiar un valor durante la desinfección.

DomSanitizer

- A veces, las aplicaciones realmente necesitan incluir código ejecutable, mostrar un <iframe> desde alguna URL o construir una URL potencialmente peligrosas. Para evitar la desinfección automática en cualquiera de estas situaciones, se le puede decir a Angular que ya se inspeccionó un valor, verificó cómo se generó y se aseguró de que siempre sea seguro. El peligro está en confiar en un valor que podría ser malicioso, se está introduciendo una vulnerabilidad de seguridad en la aplicación.
- Para marcar un valor como confiable, hay que inyectar el servicio DomSanitizer y llamar a uno de los siguientes métodos para convertir la entrada del usuario en un valor confiable:
  - bypassSecurityTrustHtml
  - bypassSecurityTrustScript
  - bypassSecurityTrustStyle
  - bypassSecurityTrustUrl
  - bypassSecurityTrustResourceUrl
- Un valor es seguro dependiendo del contexto, es necesario elegir el contexto correcto para su uso previsto del valor.

CONCEPTOS AVANZADOS

Ciclo de vida



- Cada componente tiene un ciclo de vida gestionado por el Angular.
- Angular lo crea y pinta, proyecta su contenido, crea y pinta sus hijos, comprueba cuando sus propiedades enlazadas a datos cambian, y lo destruye antes de quitarlo del DOM.
- Angular ofrece ganchos al ciclo de vida que proporcionan visibilidad a dichos momentos clave y la capacidad de actuar cuando se producen.

Ciclo de vida

Gancho	Propósito y temporización
ngOnChanges	Responder cuando Angular (re) establece las propiedades de entrada enlazadas a datos.
ngOnInit	Inicializar el componente después de que Angular muestre las primeras propiedades enlazadas a datos y establece las propiedades de entrada del componente.
ngDoCheck	Llamado cada vez que las propiedades de entrada de un componente o una directiva se comprueban. Lo utilizan para extender la detección de cambios mediante la realización de una comprobación personalizada.
ngAfterContentInit	Responder después de que Angular proyecta el contenido externo en la vista del componente.
ngAfterContentChecked	Responder después de que Angular chequee el contenido proyectado en el componente.
ngAfterViewInit	Responder después de que inicialice las vistas del componente y sus hijos (@ViewChild).
ngAfterViewChecked	Responder después de que Angular chequee las vistas del componente y sus hijos.
ngOnDestroy	Limpiar justo antes de que Angular destruya el componente.

Ciclo de vida

- Las instancias de componentes y directivas tienen un ciclo de vida ya que Angular las crea, las actualiza y las destruye. Los desarrolladores pueden aprovechar los momentos clave en ese ciclo de vida mediante la implementación de una o más de las interfaces de enlace de ciclo de vida en la biblioteca core de Angular denominados ganchos.
- Las interfaces son opcionales para desarrolladores de JavaScript y de Typescript desde una perspectiva puramente técnica. El lenguaje JavaScript no tiene interfaces. Angular no puede ver las interfaces de TypeScript en tiempo de ejecución porque desaparecen del JavaScript transpilado.
- Afortunadamente, no son imprescindibles. No es necesario agregar las interfaces para beneficiarse de los propios ganchos.
- En cambio, Angular inspecciona las clases de directivas y componentes buscando los métodos del interfaz y los invoca en caso de estar definidos.
- No obstante, es una buena práctica agregar interfaces a las clases de directivas y componentes de TypeScript para beneficiarse de las herramientas de los entornos de desarrollo.

OnChanges

- Angular llama al método ngOnChanges() cada vez que detecta cambios en las propiedades de entrada del componente (o directiva).
- El método ngOnChanges(changes: SimpleChanges) recibe un objeto que asigna cada nombre de propiedad cambiado a un objeto SimpleChange que contiene el valor actual y anterior de propiedad que ha cambiado.

```
let cur = changes[propName].currentValue;
let prev = changes[propName].previousValue;
```

- Angular solo llama a ngOnChanges() cuando cambia el valor de la propiedad de entrada. En los objetos el valor es la referencia por lo que los cambios interiores del objetos en el mismo objeto no se detectan.
- Otra posibilidad de interceptar cambios en una propiedad de entrada es con un setter:

```
@Input() set name(value: string) { … }
```

OnInit

- Se debe usar ngOnInit() para:
  - realizar inicializaciones complejas poco después de la construcción.
  - configurar el componente después de que Angular establezca las propiedades de entrada.
- Los constructores deben ser ligeros y seguros de construir. No deben solicitarse datos en un constructor de componentes. No debe de preocuparnos que un componente nuevo intente contactar a un servidor remoto cuando se haya creado para probar o antes de que se decida mostrarlo. Los constructores no deberían hacer más que establecer las variables locales iniciales en valores simples.
- Un ngOnInit() es un buen lugar para que un componente obtenga sus datos iniciales. Es donde pertenece la lógica pesada de inicialización.
- Hay que destacar que las propiedades de entrada de datos de una directiva no se establecen hasta después de la construcción. Eso es un problema si se quiere inicializar el componente en el constructor basándose en dichas propiedades. En cambio, ya se habrán establecido cuando se ejecute ngOnInit().
- Aunque para acceder a esas propiedades suele se preferible el método ngOnChanges() dado que se ejecuta antes que ngOnInit() y tantas veces como es necesario. Solo se llama a ngOnInit() una vez.

DoCheck

- Después de cada ciclo de detección de cambios se invoca ngDoCheck() para detectar y actuar sobre los cambios que Angular no atrapa por sí mismo.
- Debe inspeccionar ciertos valores de interés, capturando y comparando su estado actual con los valores anteriores cacheados.
- Si bien ngDoCheck() puede detectar cambios que Angular pasó por alto, tiene un costo aterrador dado que se llama con una frecuencia enorme, después de cada ciclo de detección de cambio, sin importar dónde ocurrió el cambio
- La mayoría de estas comprobaciones iniciales se desencadenan por la primera representación de Angular de datos no relacionados en otro lugar de la página. El simple hecho de pasar a otro <input> dispara una llamada y la mayoría de las llamadas no revelan cambios pertinentes.
- La implementación debe ser muy ligera o la experiencia de usuario se resiente.

AfterContent

- ngAfterContentInit() y ngAfterContentChecked() se invocan después de que Angular haya proyectado el contenido externo en el componente, el textContent del componente.
- La proyección de contenido es una forma de importar contenido HTML desde fuera del componente e insertar ese contenido en la plantilla del componente en el lugar designado (anteriormente transclusión).

```
<app-demo>Mi contenido</app-demo>
```

- Nunca debe colocarse contenido entre las tag de un componente a menos que se tenga la intención de proyectar ese contenido en el componente.
- Actúan basándose en valores cambiantes en un elemento secundario de contenido, que solo se pueden obtener consultando a través de la propiedad decorada con @ContentChild.

AfterView

- ngAfterViewInit() y ngAfterViewChecked() se invocan después de que Angular cree las vistas secundarias de un componente.
- Actúan basándose en valores cambiantes dentro de la vista secundaria, que solo se pueden obtener consultando la vista secundaria a través de la propiedad decorada con @ViewChild.
- La regla de flujo de datos unidireccionales de Angular prohíbe las actualizaciones de la vista una vez que se haya compuesto. Ambos ganchos se disparan después de que se haya compuesto la vista del componente.
- Angular lanza un error si el gancho actualiza inmediatamente una propiedad enlazada a datos del componente, hay que posponerla a un ciclo la modificación.

```
window.setTimeout(() => vm.model = value, 0);
```

- Angular llama con frecuencia AfterViewChecked(), a menudo cuando no hay cambios de interés. La implementación debe ser muy ligera o la experiencia de usuario se resiente.

AfterContent y AfterView

- Aunque AfterContent y AfterView son similares, la diferencia clave está en el componente hijo:
  - Los ganchos AfterContent se refieren con @ContentChildren a los componentes secundarios que Angular **proyectó** en el componente.
  - Los ganchos de AfterView se refieren con @ViewChildren a los componentes secundarios cuyas etiquetas de elemento aparecen **dentro de la plantilla** del componente.
- Otra diferencia radica en que AfterContent puede modificar inmediatamente las propiedades de enlace de datos del componente sin necesidad de esperar: Angular completa la composición del contenido proyectado antes de finalizar la composición de la vista del componente.

OnDestroy

- Debe implementarse la lógica de limpieza en ngOnDestroy(), la lógica que debe ejecutarse antes de que Angular destruya la directiva.
- Este es el momento de notificar a otra parte de la aplicación que el componente va a desaparecer.
- Es el lugar para:
  - Liberar recursos que no serán eliminados automáticamente por el recolector de basura.
  - Darse de baja de Observables y eventos DOM.
  - Detener temporizadores.
  - Eliminar el registro de todas las devoluciones de llamada que se registraron en servicios globales o de aplicaciones.
- Nos arriesgamos a fugas de memoria y de proceso si no se hace.

Representación (v.17)

- Las funciones afterRender y afterNextRender permiten registrar una devolución de llamada de renderizado para invocarla después de que Angular haya terminado de renderizar todos los componentes de la página en el DOM. afterRender se ejecuta cada vez que **todos** los componentes se han procesado en el DOM y afterNextRender se ejecutara la próxima vez que **todos** los componentes se hayan procesado en el DOM.
- Se pueden utilizar para realizar operaciones DOM manuales o diseños que requieren que todos los objetos ya estén creados.
- Estas funciones son diferentes del resto de ganchos del ciclo de vida. En lugar de un método de clase, son funciones independientes que reciben una devolución de llamada y deben llamarse en un contexto de inyección, normalmente **en el constructor de un componente**. Se les puede especificar, como segundo parámetro, la fase de renderizado (EarlyRead, MixedReadWrite, Write y Read), que brinda control sobre la secuenciación de las operaciones DOM, permitiendo secuenciar las operaciones de escritura antes de las operaciones de lectura para minimizar la alteración del diseño.
- La ejecución de devoluciones de llamada de renderizado no está vinculada a una instancia de componente específica, sino a un enlace para toda la aplicación.
- Estas funciones no se ejecutan durante el renderizado del lado del servidor ni durante el renderizado previo en tiempo de compilación.

Detección de cambios

- El mayor reto de cualquier framework Javascript es mantener sincronizado el estado del componente con su representación en la vista mediante nodos DOM dado que el proceso de renderizado es lo más costoso a nivel de proceso.
- Angular tiene una mecanismo denominado ChangeDetector para detectar inconsistencias (cambios), entre el estado del componente y la vista. El desarrollador es responsable de actualizar el modelo de la aplicación y Angular, mediante la detección de cambios, es responsable de reflejar el estado del modelo en la vista.
- El ChangeDetector es un algoritmo ultra eficiente, optimizado para la maquina virtual de Javascript, dado que se genera (en tiempo de transpilación) un gestor de cambios especifico para cada componente.
- Es un mecanismo que se comienza a ejecutar en el nodo padre (root-element) y que se va propagando a cada nodo hijo.
- Si se detecta un cambio no aplicado en la vista, éste generará una modificación en el árbol DOM.

ChangeDetector

- En tiempo de ejecución, Angular creará clases especiales que se denominan detectores de cambios, uno por cada componente que hemos definido. Por ejemplo, Angular creará dos clases: AppComponent y AppComponent_ChangeDetector.
- El objetivo de los detectores de cambio es saber qué propiedades del modelo se utilizaron en la plantilla de un componente y que han cambiado desde la última vez que se ejecutó el proceso de detección de cambios.
- Para saberlo, Angular crea una instancia de la clase de detector de cambio apropiada con un enlace al componente, al que se supone debe verificar, y propiedades para almacenar los valores anteriores de las propiedades.
- Cuando el proceso de detección de cambios quiere saber si la instancia del componente ha cambiado, ejecutará el método detectChanges pasando los valores actuales del modelo para compararlos con los anteriores. Si se detecta un cambio, cachea los nuevos valores y el componente se actualiza.

Como detectar cambios

- Conseguir un buen rendimiento en la aplicación, va a estar muy relacionado con cuántas veces (y con qué frecuencia), se ejecuta el proceso de detección de cambios.
- En el 99% de los casos, el cambio en el estado del componente está provocado por:
  - Eventos en la interfaz (clicks, mouseover, resizes, etc)
  - Peticiones Ajax
  - Ejecuciones dentro de temporizadores (setTimeout o setInterval)
- Se puede concluir con cierta seguridad que se deben actualizar las vistas (o comprobarlo al menos) después de producirse uno de estos tres supuestos.

zone.js

- Zone.js es una librería externa, inspirada en el zones de dart:async, que crea un contexto de ejecución para operaciones asíncronas. Una zona se encarga de ejecutar una porción de código asíncrono, siendo capaz de conocer cuando terminan todas las operaciones asíncronas. Zone dispone del método run que recibe como argumento la función a ejecutar con las acciones asíncronas y termina cuando concluyen todas las acciones pendientes.
- Una zona tiene varios hooks disponibles:
  - onZoneCreated (al crear un nuevo fork)
  - beforeTask (antes de ejecutar una nueva tarea con zone.run)
  - afterTask (después de ejecutar la tarea con zone.run)
  - onError (Se ejecuta con cualquier error lanzado desde zone.run)
- Al hacer un fork en una zona se obtiene una nueva zona con todos los hooks definidos en su padre.

zone.js

```
function miMetodoAsync() {
  // Peticiones AJAX
  // Operaciones setTimeout
}

const myZoneConf = {
  beforeTask: () => { console.log('Antes del run'); },
  afterTask: () => { console.log('Después del run'); }
};

const myZone = zone.fork(myZoneConf);
myZone.run(miMetodoAsync);
```

NgZone

- NgZone es la implementación utilizada en Angular para la ejecución de tareas asíncronas. Es un fork (bifurcación) de zone.js con ciertas funcionalidades extra orientadas a gestionar la ejecución de componentes y servicios dentro (o fuera) de la zona de Angular.
- Será también el encargado de notificar al ChangeDetector que debe ejecutarse para buscar posibles cambios en el estado de los componentes y actualizar el DOM si fuera necesario.
- NgZone se utiliza como un servicio inyectable en componentes o servicios que expone los siguientes métodos la gestión de las zonas:
  - run: ejecuta cierto código dentro de la zona
  - runGuard: ejecuta cierto código dentro de la zona pero los errores se inyectan en onError en lugar de ser re-ejecutados
  - runOutsideAngular: ejecuta el código fuera de la zona de angular por lo que al concluir no se ejecuta ChangeDetector

NgZone

```
progress = 0;
outZone() {
  this.progress = 0
  this.ngZone.runOutsideAngular(() => {
    const interval = setInterval(() => {
      if (this.progress % 100)
        this.progress++
      else
        this.ngZone.run(() => this.progress++)
      if (this.progress >= 1000) {
        clearInterval(interval)
        console.log('end interval');
        this.ngZone.run(() => this.progress = 1000)
      }
    }, 10)
  })
}
<div>
  <progress max="1000" [value]="progress" ></progress><input type="button" value="Out Zone" (click)="outZone()" >
</div>
```

NgZone

- NgZone expone un conjunto de Observables que nos permiten determinar el estado actual o la estabilidad de la zona de Angular:
  - onUnstable: notifica cuando el código ha entrado y se está ejecutando dentro de la zona Angular.
  - onMicrotaskEmpty: notifica cuando no hay más microtasks en cola para su ejecución. Angular se suscribe a esto internamente para indicar que debe ejecutar la detección de cambios.
  - onStable: notifica cuándo el último onMicroTaskEmpty se ha ejecutado, lo que implica que todas las tareas se han completado y se ha producido la detección de cambios.
  - onError: notifica cuando se ha producido un error. Angular se suscribe a esto internamente para enviar errores no detectados a su propio controlador de errores, es decir, los errores que ve en su consola con el prefijo 'EXCEPCIÓN:'.

```
constructor(private ngZone: NgZone) {
  this.ngZone.onStable.subscribe(() => console.log('Zone are stable'));
  this.ngZone.onUnstable.subscribe(() => console.log('Zone are unstable'));
  this.ngZone.onError.subscribe((error) => console.error('Error', error instanceof Error ? error.message :
      error.toString()));
}
```


NgZone

- Cualquier código ejecutado dentro de la zona y que sea susceptible a cambios (actualización Ajax, eventos de ratón, …) va a forzar la ejecución del detector de cambios en todo el árbol de la aplicación.
- Hay que tener en cuenta que esa detección de cambios, aunque optimizada, puede tener un coste de proceso alto, sobre todo si se ejecuta varias veces por segundo.
- En determinadas ocasiones existe un elevado número de ejecuciones asíncronas, que nos llevarían a una ejecución del detector de cambios con más frecuencia de lo deseado.
- Por ejemplo, capturar el evento mousemove en un determinado componente, invocará el ChangeDetector en todo el árbol de la aplicación con cada pixel por el que pase el puntero. Es una buena práctica capturar ese evento fuera de la zona de Angular (runOutsideAngular), aunque con cautela porque puede dar lugar a inconsistencias entre el estado de los componentes y el DOM.

Estrategias de detección de cambios

- Con cada actualización en NgZone, Angular deberá ejecutar el detector de cambios en cada uno de los componentes del árbol. Para que un nodo DOM se actualice, es imprescindible que se ejecute el detector de cambios en dicho nodo, para lo cual también se debe ejecutar en todos su antecesores hasta el document raíz.
- En principio no hay manera de asociar determinadas zonas a determinados componentes: una ejecución de cualquier método asíncrono en la zona, desencadenará la ejecución de ChangeDetector en todo el árbol de la aplicación.
- Angular ha implementado dos estrategias de detección de cambios en el estado del componente: Default y OnPush, que determinan cómo y, sobre todo, cuándo se ejecuta el ChangeDetector y en qué componentes del árbol de componentes que es la aplicación.

```
@Component({
  changeDetection: ChangeDetectionStrategy.OnPush
```


ChangeDetectionStrategy.Default

- ChangeDetectionStrategy.Default es la estrategia por defecto, que hace que todo funcione correctamente sin necesidad de preocuparnos de la detección de cambios; eso si, se realiza una especie de estrategia de fuerza bruta.
- Por cada cambio ejecutado y detectado en la zona, Angular realiza comparaciones en todas las variables referenciadas en la plantilla, tanto por referencia como por valor (incluso en objetos muy profundos), y en todos los componentes de la aplicación.
- Hay que tener en cuenta que esas comparaciones por valor, son muy costosas en cuanto a consumo de CPU, ya que deberá ir comparando cualquier objeto que esté asociado a la vista. Y es una operación que se ejecutará con cualquier cambio detectado, tenga o no que ver con el componente.
- No obstante, por norma general, en el 95% de las aplicaciones web, es una estrategia válida y que consigue un rendimiento más que aceptable.

ChangeDetectionStrategy.OnPush

- En ChangeDetectionStrategy.OnPush, con cada cambio registrado en la zona, la única comprobación que se realizará por componente será la de los parámetros @Input de dicho componente.
- Únicamente se invocará el ChangeDetector de ese componente, si se detecta cambios en la referencia de los parámetros @Input del componente. La comprobación por referencia es mucho más óptima y rápida, pero puede dar lugar a ciertos problema si no se controla bien (inmutabilidad).
- Solo cuando una referencia se actualiza será cuando se volverá a renderizar la vista. El resto de los casos que necesiten renderización requerirán un “empujón”.
- Es una estrategia mucho más barata en términos de consumo de CPU. La estrategia se hereda de contenedores a contenidos, por lo que podrá estar definida en todo el árbol de la aplicación o bien limitada a ciertas ramas.

ChangeDetectorRef

- Angular ofrece el servicio denominado ChangeDetectorRef que es una referencia inyectable al ChangeDetector.
- Este servicio facilita poder gestionar el detector de cambios a voluntad, lo cual resulta muy útil cuando se utiliza la estrategia de actualización OnPush o cuando se ejecuta código fuera de la zona.
- Con ChangeDetectorRef.markForCheck() nos aseguramos que el detector de cambios del componente, y de todos sus contenedores, se ejecutará en la próxima ejecución de la zona. Una vez realizada la siguiente detección de cambios en el componente, se volverá a la estrategia OnPush.

```
myObservable.subscribe(data => {
  // …
  this.changeDetectorRef.markForCheck();
});
```

ChangeDetectorRef

- ChangeDetectorRef.detach(): Para sacar el componente y todo su contenido de la futura detección de cambios de la aplicación. Las futuras sincronizaciones entre los estados del componente y las plantillas, deberá realizarse manualmente.
- ChangeDetectorRef.reattach(): Para volver a incluir el componente y todo su contenido en las futuras detecciones de cambios de la aplicación.
- ChangeDetectorRef.detectChanges(): Para ejecutar manualmente el detector de cambios en el componente y todo su contenido. Utilizado en componentes en los que se ha invocado “detach“, consiguiendo así una actualización de la vista.

Optimización y rendimiento

- Una aplicación de rendimiento exigente es aquella que ejecuta un elevado número de operaciones de cambio por segundo (eventos/XHR/websocket) y con un elevado número de enlaces en la vista que deben actualizarse en base a dichas operaciones.
- Con la estrategia ChangeDetectionStrategy.Default, sencilla y funcional, es recomendable:
  - Ejecutar fuera de la zona cualquier evento de alta frecuencia, de manera que sus ejecuciones queden desacopladas de la ejecución de los detectores de cambios en el árbol de componentes.
  - Evitar en la medida de lo posible “getters” costosos directamente enlazados a la vista (usar memoizes o caches).
  - Desacoplar (detach) cuando sea necesario los detectores de cambios y ejecutarlos únicamente bajo demanda.

Optimización y rendimiento

- La estrategia ChangeDetectionStrategy.OnPush supone un mayor rendimiento dado que el algoritmo de detección de cambios se va a ejecutar muchas menos veces. Pero esta estrategia va a obligar a disponer de un diseño mucho más cuidado y específico en los componentes.
- Las recomendaciones son:
  - crear (y anidar) componentes con la mayor profundidad posible. Esto implica que nuestro componentes serán más simples y reutilizables.
  - inyectar propiedades en estos componentes mediante @Input, utilizando estrategias de inmutabilidad:
    - Utilizar immutable.js o Mori
    - Utilizar ngrx, que trae un estado inmutable (redux) a Angular
    - O directamente con Object.assing({}, objecto)
  - (ab)usar de los Observables que notifican cuando llega un nuevo dato: permitiendo invocar markForCheck en el suscriptor o delegar la suscripción directamente en la plantilla mediante AsyncPipe.

Optimizar con Pipes puros

- Una función en una expresión de plantilla se invoca cada vez que se ejecuta la detección de cambios. Debido a esto, se considera mejor sustituir la llamada a la función en la expresión con un pipe puro que encapsula su lógica.
- La mejora en el rendimiento proviene del hecho de que Angular optimiza los pipes puros llamando a transform solo cuando cambian el valor al que se aplica o los parámetros adicionales que de la función de transformación. Si alguno de estos valores cambia, se vuelve a ejecutar el método transform del pipe.
- Se puede crear un pipe genérico que invoque a las funciones:

```
@Pipe({ name: 'exec' })
export class ExecPipe implements PipeTransform {
  transform(fn: Function, ...args: any[]): any { return fn(...args); }
}
```

- Para sustituir:

```
{{ calcula(a, b) }}
```

- Por:

```
{{ calcula | exec:a:b }}
```

Nota:
Dado que los métodos de las clases en JavaScript pierden el contexto si se usan con nombre (this por defecto es el objeto Global o Window), se debe hacer un bind explicito en el constructor:

```
calcula = calcula.bind(this)
```

Internacionalización I18n

- La internacionalización es el proceso de diseño y preparación de una aplicación para que pueda utilizarse en diferentes idiomas. La localización es el proceso de traducir la aplicación internacionalizada a idiomas específicos para lugares específicos.
- Angular simplifica los siguientes aspectos de la internacionalización:
  - Visualización de fechas, números, porcentajes y monedas en un formato local.
  - Preparación del texto de las plantillas de los componentes para la traducción.
  - Manejo de formas plurales de palabras.
  - Manejo de textos alternativos.
- Para la localización, se puede utilizar Angular CLI para generar la mayor parte de la plantilla necesaria para crear archivos para traductores y publicar la aplicación en varios idiomas. Una vez que haya configurado la aplicación para usar i18n, el CLI simplifica los siguientes pasos:
  - Extraer texto localizable en un archivo que puede enviar para ser traducido.
  - Construir y servir la aplicación con una configuración regional determinada, usando el texto traducido.
  - Creación de versiones en múltiples idiomas de la aplicación.
- De forma predeterminada, Angular usa la configuración regional en-US, que es el inglés tal como se habla en los Estados Unidos de América.

Pipes I18n

- Los pipes DatePipe, CurrencyPipe, DecimalPipe y PercentPipe muestran los datos localizados basándose en el LOCALE_ID.
- Por defecto, Angular solo contiene datos de configuración regional para en-US. Si se establece el valor de LOCALE_ID en otra configuración regional, se debe importar los datos de configuración regional para la nueva configuración regional. El CLI importa los datos de la configuración regional cuando con ng serve y ng build se utiliza el parámetro --configuration.

```
ng serve --configuration=es
```

- Si se desea importar los datos de configuración regional de otros idiomas, se puede hacer manualmente:

```
import { registerLocaleData } from '@angular/common';
import localeEs from '@angular/common/locales/es';
import localeEsExtra from '@angular/common/locales/extra/es';
registerLocaleData(localeEs, 'es', localeEsExtra);
```

Localización

- El equipo de Angular ha fijado como mejor practica generar tantas aplicaciones diferentes como idiomas se deseen soportar, frente a una única aplicación con múltiples idiomas.
- El proceso de traducción de plantillas i18n tiene una serie de fases:
  - Para aprovechar las funciones de localización de Angular hay que agregar:
    - ng add @angular/localize
  - Marcar todos los mensajes de texto estático en las plantillas de componentes para la traducción.
  - Crear un archivo de traducción: el comando ng xi18n permite extraer los textos marcados en las plantillas a un archivo fuente de traducción estándar de la industria.
    ng extract-i18n --output-path src/locale
  - Duplicar el archivo tantas veces como idiomas se deseen utilizando como sub extensión el identificador de idioma Unicode y, opcionalmente, la configuración regional.
  - Editar el archivo de traducción duplicado correspondiente y traducir el texto extraído al idioma de destino.
  - Crear en angular.json las configuraciones específicas de los diferentes idiomas.
  - Fusionar el archivo de traducción completado con la aplicación:
    - ng build --prod --configuration=es

Localización: Marcado

- Los atributo i18n y i18n-atributo de Angular marcan un contenido como traducible y hay que utilizarlos en cada etiqueta o atributo cuyo contenido literal deba ser traducido.
  <h1 i18n>Hello!</h1> <ng-container i18n>sin tag</ng-container>
- i18n no es una directiva angular, es un atributo personalizado reconocido por las herramientas y compiladores de Angular que después de la traducción es eliminado en la compilación.
- Para traducir un mensaje de texto con precisión, el traductor puede necesitar información adicional o contexto. El traductor también puede necesitar conocer el significado o la intención del mensaje de texto dentro de este contexto de aplicación en particular.
- Como el valor del atributo i18n se puede agregar una descripción del mensaje de texto y, opcionalmente, precederla del sentido <meaning>|<description>@@<id>:
  <h1 i18n="site header|An introduction header for this sample">Hello i18n!</h1>
- Todas las apariciones de un mensaje de texto que tengan el mismo significado tendrán la misma traducción. El mismo mensaje de texto que está asociado con significados diferentes puede tener diferentes traducciones.
- La herramienta de extracción genera una hash como identificador de cada literal, se puede especificar uno propio usando el prefijo @@: i18n="@@myID"

Localización: Traducción

- Para realizar la traducción se añade un elemento target a continuación del elemento source, aunque el target es el único imprescindible por cada id en el fichero traducido:

```
<source>Hello!</source>
<target>¡Hola!</target>
```

- Por defecto se genera un archivo de traducción denominado messages.xlf en el Formato de archivo de intercambio de localización XML (XLIFF, versión 1.2), pero también acepta los formatos XLIFF 2 y Paquete de mensajes XML (XMB).
  - ng extract-i18n --i18n-format=xlf2
  - ng extract-i18n --i18n-format=xmb
- Los archivos XLIFF tienen la extensión .xlf. El formato XMB genera archivos de origen .xmb pero utiliza archivos de traducción .xtb
- La mayoría de las aplicaciones se traducen a más de un idioma, es una práctica estándar dedicar una carpeta a la localización y almacenar los activos relacionados, como los archivos de internacionalización, allí.

Localización: Código

- No solo el texto de las plantillas requiere traducción, también todos los literales utilizados en el código.
- Angular no provee de una utilidad similar a la de las plantillas para el código pero se puede implementar el mismo mecanismo utilizado para el environment.
- Crear en la carpeta de localización un fichero denominado messages.ts (por ejemplo):

```
export const messages = {
  AppComponent: {
    title: 'Hola Mundo',
  }
};
```

- Crea un duplicado por idioma siguiendo el mismo convenio messages.en.ts.

```
export const messages = {
  AppComponent: {
    title: 'Hello Word',
  }
};
```

- Sustituir los literales del código por:

```
title = messages.AppComponent.title;
```

Localización: Configuración

```
"projects": {
  "curso": {
    "i18n": {
      "sourceLocale": "es",
      "locales": {
        "en": {
          "translation": "src/locale/messages.en.xlf",
          "baseHref": ""
        }
      }
    },
    "architect": {
      "build": {
        "builder": "@angular-devkit/build-angular:browser",
        "options": {
          "localize": true,
          "i18nMissingTranslation": "error"
        },
        "configurations": {
          "production": {
            "localize": true,
          },
          "en": {
            "localize": ["en"],
            "fileReplacements": [{
              "replace": "src/locale/messages.ts",
              "with": "src/locale/messages.en.ts"
            }]
          }
        },
      },
      "serve": {
        "en": {
          "browserTarget": "curso:build:development,en"
        }
      },
    },
```


Apache2 configuration

```
<VirtualHost *:80>
  ServerName www.myapp.com
  DocumentRoot /var/www
  <Directory "/var/www">
    RewriteEngine on
    RewriteBase /
    RewriteRule ^../index\.html$ - [L]
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteRule (..) $1/index.html [L]
    RewriteCond %{HTTP:Accept-Language} ^fr [NC]
    RewriteRule ^$ /fr/ [R]
    RewriteCond %{HTTP:Accept-Language} ^es [NC]
    RewriteRule ^$ /es/ [R]
    RewriteCond %{HTTP:Accept-Language} !^es [NC]
    RewriteCond %{HTTP:Accept-Language} !^fr [NC]
    RewriteRule ^$ /en/ [R]
  </Directory>
</VirtualHost>
```

Módulos como bibliotecas

- Si encuentra que necesita resolver el mismo problema en más de una aplicación (o desea compartir su solución con otros desarrolladores), tiene un candidato para una biblioteca.
- Para que su solución sea reutilizable, se debe ajustar para que no dependa del código específico de la aplicación. Algunas cosas a considerar al migrar la funcionalidad de la aplicación a una biblioteca son:
  - Las declaraciones, como componentes y pipes, deben diseñarse sin estado, lo que significa que no dependen de variables externas ni las alteran. Los componentes deben exponer sus interacciones a través de entradas para proporcionar contexto y salidas para comunicar eventos a otros componentes.
  - Todas las clases o interfaces personalizadas utilizadas en componentes o servicios deben estar disponibles en la biblioteca. Cualquier elemento observable al que los componentes se suscriban internamente debe limpiarse y desecharse durante el ciclo de vida de esos componentes.
  - Si los componentes dependen de servicios, estos deben estar disponibles en la biblioteca. Los servicios deben declarar sus propios proveedores, en lugar de declarar proveedores en el NgModule o componente, permitiendo que el árbol de servicios se pueda sacudir y el compilador deje fuera del paquete el servicio si nunca se inyecta en la aplicación que importa la biblioteca. Si se registran proveedores de servicios globales o se comparten proveedores en múltiples NgModules, se deben usar los patrones de diseño forRoot() y forChild() utilizados por el RouterModule.
  - Si el código de la biblioteca o sus plantillas dependen de otras bibliotecas (como Angular Material), se debe configurar la biblioteca con esas dependencias.

Bibliotecas

- Angular CLI v6 viene con soporte de biblioteca a través de ng-packagr conectado al sistema de compilación que utilizamos en Angular CLI, junto con esquemas para generar una biblioteca.
- Se puede crear una biblioteca en un espacio de trabajo existente ejecutando los siguientes comandos:

  ng generate library my-lib
- Ahora deberías tener una biblioteca adentro projects/my-lib, que contiene un componente y un servicio dentro de un NgModule dentro de la carpeta lib/src, donde se crearan el resto de los miembros de la biblioteca.
- Puede probar y compilar la biblioteca a través de ng test my-lib y ng build my-lib.
- La nueva librería se puede utilizar directamente dentro de su mismo un espacio de trabajo dado que la generación de la biblioteca agrega automáticamente su ruta al archivo tsconfig.

Publicación

- Aunque existen numerosas estrategias para publicar y distribuir la biblioteca de componentes, es altamente recomendado publicar la biblioteca en NPM. NPM es un registro de software en línea para compartir bibliotecas, herramientas, utilidades, paquetes, etc.
- Una vez que la biblioteca se publica en NPM, otros proyectos pueden agregar la biblioteca como una dependencia y usar los componentes dentro de sus propios proyectos.
- El archivo package.json es una especie de manifiesto del proyecto. Puede hacer muchas cosas: es un repositorio central de configuración de herramientas, es donde npm y yarn almacenan los nombres y versiones de todos los paquetes instalados, permite definir tareas de ejecución, …
- El archivo puede contener información adicional para su publicación en el repositorio de NPM: author, homepage, license, keywords, repository, engines, browserslist, bugs …

Publicación de módulos con NPM

- Para poder utilizar la biblioteca en cualquier espacio de trabajo se puede publicar en el repositorio npmjs.com para que se pueda instalar vía npm install.
- Es necesario disponer de una cuenta de usuario y seguir tres paso:
  1. Documentar en el Readme.md la biblioteca y completar en el package.json local la información complementaria como author, homepage, license, keywords, repository, …
  2. ng build my-lib --prod
  3. cd dist/my-lib
  4. npm publish
- Como paso opcional quizás sea necesario renombrar la librería dado que el nombre debe ser único en el repositorio.
- La opción --prod se debe usar cuando se compile para publicar, ya que de antemano limpiará por completo el directorio de compilación para la biblioteca, eliminando el código anterior que queda de versiones anteriores.
- Para instalar la librería en un nuevo proyecto:

  npm install my-lib --save

package.json

```
{
  "name": "my-lib",
  "description": "Biblioteca demo del curso",
  "keywords": ["curso", "demos", "angular"],
  "homepage": "https://github.com/jmagit",
  "repository": { "type": "git", "url": "https://github.com/jmagit" },
  "version": "0.0.1",
  "peerDependencies": {
    "@angular/common": "^16.0.0",
    "@angular/core": "^16.0.0"
  },
  "dependencies": {
    "tslib": "^2.3.0"
  },
  "sideEffects": false
}
```


Migración desde AngularJS a Angular

- La biblioteca ngUpgrade en Angular es una herramienta muy útil para actualizar cualquier cosa menos las aplicaciones muy pequeñas.
- Con él se puede mezclar y combinar AngularJS y componentes Angular en la misma aplicación y hacer que funcionen sin problemas.
- Eso significa que no se tiene que hacer el trabajo de actualización todo de una vez, ya que hay una coexistencia natural entre los dos marcos durante el período de transición.
- https://angular.io/guide/upgrade
- https://vsavkin.com/migrating-angular-1-applications-to-angular-2-in-5-simple-steps-40621800a25b

Meta tags sociales

- Esto es lo que tienes que hacer para integrar en las noticias, paginas de detalle de vídeo y foto galerías de tu sitio web las etiquetas de Facebook y Twitter.
- La idea es ponérselo fácil a los sistemas de compartir en redes sociales, como Twitter, Google+, Facebook o Pinterest, para mejorar la viralidad del contenido de tu sitio web.
- Las etiquetas meta sociales son simples. Contiene los datos mínimos para poder compartir contenido en Twitter, Facebook, Google+ y Pinterest.
  - Open Graph (Facebook) https://developers.facebook.com/docs/sharing/webmasters
  - Twitter cards https://developer.twitter.com/en/docs/tweets/optimize-with-cards/guides/getting-started
- Si debemos elegir un tipo de metadatos para incluir en una página web, estos serían los datos Open Graph (Facebook). Esto se debe a que todas las plataformas pueden utilizarlo como reserva, incluyendo Twitter. Las twitter cards también mejoran la iteración de nuestros usuarios con Twitter.

SEO, Google+

```
<!-- COMMON TAGS -->
<meta charset="utf-8">
<title>Titulo de la página: 60-70 characters max</title>
<!-- Search Engine -->
<meta name="description" content="Descripción larga: 150 characters for SEO, 200 characters for
    Twitter & Facebook">
<meta name="image" content="https://example.com/site_image.jpg">
<!-- Google Authorship and Publisher Markup -->
<link rel="author" href=" <a href="blank">https://plus.google.com/[Google+_Profile]/posts"/</a>>
<link rel="publisher" href=" <a href="blank">https://plus.google.com/[Google+_Page_Profile]"/</a>>
<!-- Schema.org markup for Google+ -->
<meta itemprop="name" content="Titulo">
<meta itemprop="description" content="Descripcion">
<meta itemprop="image" content="http://www.example.com/image.jpg">
```

Twitter Card

```
<!-- Twitter Card data -->
<meta name="twitter:card" content="summary_large_image">
<meta name="twitter:site" content="@publisher_handle">
<meta name="twitter:title" content="Titulo">
<meta name="twitter:description" content="Descripcion que no supere los
    200 caracteres">
<meta name="twitter:creator" content="@author_handle">
<!-- Twitter summary card with large image. Al menos estas medidas
    280x150px -->
<meta name="twitter:image:src"
    content="http://www.example.com/image.html">
```


Facebook: Open Graph

```
<!-- Open Graph data -->
<meta property="og:title" content="Titulo" />
<meta property="og:type" content="article" />
<meta property="og:url" content=" http://www.example.com/" />
<meta property="og:image" content=" http://example.com/image.jpg" />
<meta property="og:description" content="Descripcion" />
<meta property="og:site_name" content="Nombre de la web, i.e. Moz" />
<meta property="article:published_time" content="2013-09-17T05:59:00+01:00" />
<meta property="article:modified_time" content="2013-09-16T19:08:47+01:00" />
<meta property="article:section" content="Seción de la web" />
<meta property="article:tag" content="Article Tag" />
<meta property="fb:admins" content="ID de Facebook " />
```

Optimización de imágenes

- La imagen que vincules en tus datos sociales, no tiene porque estar en la página, pero debería representar el contenido correctamente. Es importante utilizar imágenes de alta calidad.
- Toda plataforma social tiene distintos estándares para el tamaño de sus imágenes. Obviamente, lo más sencillo es elegir una imagen que sirva para todos los servicios:
  - Imagen de miniatura en Twitter: 120x120px
  - Imagen grande en Twitter: 280x150px
  - Facebook: los estándares varían, pero una imagen de, al menos, 200x200px, funciona mejor. Facebook recomienda imágenes grandes de hasta 1200px de ancho. En resumen, cuanto más grandes son las imágenes, más flexibilidad vas a tener.

Servicios Title y Meta

- En este caso, nos gustaría establecer la etiqueta del título de la página y completar también las etiquetas meta como la descripción.
- Dado que una aplicación Angular no se puede iniciar en todo el documento HTML (etiqueta <html>), no es posible enlazar a la propiedad textContent de la etiqueta <title> ni generar con *ngFor las etiquetas <meta>, pero podemos hacer que el uso de los servicios Title y Meta.

```
constructor(private title: Title, private meta: Meta) {}
ngOnInit() {
  // …
  // SEO metadata
  this.title.setTitle(this.model.name);
  this.meta.addTag({name: 'description', content: this.model.description});
   // Twitter metadata
   this.meta.addTag({name: 'twitter:card', content: 'summary'});
   this.meta.addTag({name: 'twitter:title', content: this.model.description});
}
```

RXJS (REACTIVE PROGRAMMING)

Reactive Programming

- La Programación Reactiva (o Reactive Programming) es un paradigma de programación asíncrono enfocado en el trabajo con flujos de datos finitos o infinitos y la propagación de los cambios. Estos flujos se pueden observar y reaccionar en consecuencia.
- RxJS es una biblioteca para la programación reactiva que usa Observables, para componer programas asincrónicos y basados en eventos mediante el uso de secuencias observables. Proporciona una clase principal, el Observable, clases satélite y operadores inspirados en Array#extras (map, filter, reduce, every, etc) para permitir el manejo de eventos asíncronos como colecciones.
- Los conceptos esenciales en RxJS que resuelven la administración de eventos asíncronos son:
  - Observable: representa una colección invocable de valores o eventos futuros.
  - Observer: conjunto de controladores que procesan los valores entregados por el Observable.
  - Subscription: representa la ejecución de un Observable, necesario para cancelar la ejecución.
  - Operators: son funciones puras que permiten un estilo de programación funcional de tratar las colecciones con operaciones como map, filter, concat, flatMap, etc.
  - Subject: es el equivalente a un EventEmitter y la única forma de difundir un valor o evento a múltiples observadores.
  - Schedulers: son los programadores centralizados de control de concurrencia, que permiten coordinar cuanto cómputo ocurre en, por ejemplo, setTimeout, requestAnimationFrame u otros.

Observables

- Los Observables proporciona soporte para pasar mensajes entre el publicador o editor y los suscriptores en la aplicación. Los observables ofrecen ventajas significativas sobre otras técnicas para el manejo de eventos, programación asíncrona y manejo de valores múltiples.
- Los observables son declarativos; es decir, definen una función para publicar valores, pero no se ejecuta hasta que un suscriptor se suscribe. El suscriptor suscrito recibe notificaciones hasta que la función se complete o hasta que cancele la suscripción. Un observable puede entregar múltiples valores de cualquier tipo: literales, mensajes o eventos, según el contexto.
- La API para recibir valores es la misma independientemente de que los valores se entreguen de forma síncrona o asíncrona. Debido a que la lógica de configuración y extracción es manejada por el observable, el código de la aplicación solo necesita preocuparse de suscribirse para consumir valores y, al terminar, cancelar la suscripción. Ya sea un flujo de pulsaciones de teclas, una respuesta HTTP o un temporizador de intervalos, la interfaz para escuchar los valores y detener la escucha es la misma.
- Debido a estas ventajas, Angular usa ampliamente los observables y los recomiendan para el desarrollo de aplicaciones.

Anatomía de un Observable

- Los Observables se crean usando Observable.create() o un operador de creación, se suscriben con un Observer, ejecutan next/error/complete para entregar notificaciones al Observer y su ejecución puede ser eliminada.
- Estos cuatro aspectos están codificados en una instancia Observable, pero algunos de estos aspectos están relacionados con otros tipos, como Observer y Subscription.
- Las principales preocupaciones son:
  - Crear Observables
  - Suscribirse a Observables
  - Ejecutar Observables
  - Desechar Observables

Anatomía de un Observable

- Los Observables se pueden crear con create, pero por lo general se utilizan los llamados operadores de creación como of, from, interval, etc.
- Suscribirse a un Observable es como llamar a una función, proporcionando devoluciones de llamadas donde se entregarán los datos.
- En una ejecución observable, se pueden entregar notificaciones de cero a infinito.
- Si se entrega una notificación de error o de completado, entonces no se podrá entregar nada más después.
- Al suscribirse, se obtiene un objeto Subscription, que representa la ejecución en curso. Basta con llamar al unsubscribe() de dicho objeto para cancelar la suscripción.

Publicador - Suscriptor

- Para crear un publicador, se crea una instancia de la clase Observable en cuyo constructor se le suministra la función publicadora que define cómo obtener o generar los valores o mensajes para publicar. Es la función que se ejecuta cuando un suscriptor llama al método subscribe().

```
let publisher = new Observable(observer => {
    // ...
    observer.next("next value");
    // ...
    return {unsubscribe() { ... }};
  });
```

- Para crear un suscriptor, sobre la instancia Observable creada (publicador), se llama a su método subscribe() pasando como parámetros un objeto Observer con lo que se desea ejecutar cada vez que el publicador entregue un valor.

```
let subscriber = publisher.subscribe( {
    next: notify => console.log("Observer got a next value: " + notify),
    error: err => console.error("Observer got an error: " + err)
  });
```

- La llamada al método subscribe() devuelve un objeto Subscription que tiene el método unsubscribe() para dejar de recibir notificaciones.

```
subscriber.unsubscribe();
```

Publicador Síncrono

- En algunos casos la secuencia de valores ya está disponible pero se requiere un Observable para unificar el tratamiento de las notificaciones mediante suscriptores.
- RxJS dispone de una serie de funciones que crean una instancia observable:
  - of(...items): instancia que entrega sincrónamente los valores proporcionados como argumentos.
  - from(iterable): convierte el argumento en una instancia observable. Este método se usa comúnmente para convertir una matriz en un observable.
  - empty(): instancia un observable completado.

```
const publisher = of(1, 2, 3);
let subscriber = publisher.subscribe( {
    next: notify => console.log("Observer got a next value: " + notify),
    error: err => console.error("Observer got an error: " + err)
  });
```


Observer

- La función de publicadora recibe un parámetro que implementa la interfaz Observer y define los métodos de devolución de llamada para manejar los tres tipos de notificaciones que un observable puede enviar:
  - **next:** obligatorio, la función invocada para cada valor a devolver. Se llama cero o más veces después de que comience la ejecución.
  - **error**: opcional, un controlador para una notificación de error. Un error detiene la ejecución de la instancia observable.
  - **complete**: opcional, controlador para notificar se ha completado la ejecución.
- Se corresponden con los controladores suministrados al crear la suscripción.

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

- Alternativamente, el método subscribe() puede aceptar como argumentos los controladores next, error y complete.

Suscripción

- Una Suscripción es un objeto que representa un recurso desechable, generalmente la ejecución de un Observable. La Suscripción tiene el método unsubscribe, sin argumentos, para eliminar el recurso Subscription de la suscripción. En versiones anteriores de RxJS, la Subscription se llamaba "Disposable".
- Una Suscripción esencialmente solo tiene el metodo unsubscribe() para liberar recursos o cancelar ejecuciones de Observable.

```
let observable = interval(1000);
let subscription = observable.subscribe({next: x => console.log(x)});
// …
subscription.unsubscribe();
```

- Las suscripciones también se pueden mezclar, de modo que una llamada a unsubscribe() de una suscripción puede dar de baja de varias suscripciones.

```
let subscription = observable1.subscribe({next: x => console.log(x)});
subscription.add(observable2.subscribe({next: x => console.log('second: ' + x)}));
subscription.unsubscribe();
```


Geolocalización

```
publisher$ = new Observable(observer => {
  if(!window.navigator.geolocation) {
    observer.error('Geolocalización no disponible en el navegador')
  }
  let watchId = window.navigator.geolocation.watchPosition(pos => {
    observer.next({ latitud: pos.coords.latitude, longitud: pos.coords.longitude })
  }, error => {
    switch (error.code) {
      case error.PERMISSION_DENIED: observer.error('Permiso denegado por el usuario'); break;
      case error.POSITION_UNAVAILABLE: /*observer.error('Posición no disponible');*/ console.warn('Posición no disponible'); break;
      case error.TIMEOUT: observer.error('Tiempo de espera agotado'); break;
      default: observer.error(`Error desconocido en la geolocalización: ${error.code}`); break;
    }
  });
  return function unsubscribe() {
    window.navigator.geolocation.clearWatch(watchId)
  };
});
```

Subject

- Un Subjet RxJS es un tipo especial de Observable que permite la multidifusión de valores a varios observadores. Mientras que los Observables simples son unidifusión (cada Observador suscrito posee una ejecución independiente del Observable), los Subjet son multidifusión.
- Los Subjet son como EventEmitters: mantienen un registro de muchos oyentes.
- Cada Subjet es un observable.
  - Un Subjet permite subscripciones proporcionando un Observador, que comenzará a recibir valores normalmente.
  - Desde la perspectiva del Observer, no se puede distinguir si la ejecución proviene de un Observable unidifusión o multidifusión.
  - Internamente para el Subjet, subscribe() no invoca una nueva ejecución que entrega valores, simplemente registra el Observer dado en una lista de Observadores, de forma similar al addListener de los eventos.
- Cada Subjet es un observador.
  - Es un objeto con los métodos next(v), error(e) y complete(), la invocación ellos será difundida a todos los suscriptores del Subjet.

Subject

- Como observable:

```
let subject = new Subject();
subject.subscribe({ next: (v) => console.log('observerA: ' + v) });
subject.subscribe({ next: (v) => console.log('observerB: ' + v) });
subject.next(1);
subject.next(2);
subject.complete();
```

- Como observador:

```
let observable = from([1, 2, 3]);
let subject = new Subject();
observable.subscribe(subject); //de unidifusión a multidifusión
```

BehaviorSubject

- Una de las variantes de Subject es el BehaviorSubject, que tiene la noción de "valor actual". Almacena el último valor emitido a sus consumidores y, cada vez que un nuevo observador se suscriba, recibirá de inmediato el "valor actual" de BehaviorSubject.
- Los BehaviorSubjects son útiles para representar "valores a lo largo del tiempo".
- BehaviorSubject recibe en su constructor el valor inicial que el primer Observer recibe cuando se suscribe.

```
let subject = new BehaviorSubject(-1);
interval(2000).subscribe(subject);
subject.subscribe({ next: (v) => console.log('observerA: ' + v) });
setTimeout(() => subject.subscribe({
    next: (v) => console.log('observerB: ' + v)
}), 2500);
```


ReplaySubject

- El ReplaySubject es similar a BehaviorSubject en que puede enviar valores antiguos a nuevos suscriptores, pero también puede registrar una parte de la ejecución Observable. Un ReplaySubject recuerda múltiples valores de la ejecución Observable y se los notifica a los nuevos suscriptores.
- Al crear un ReplaySubject se puede especificar el tamaño del búfer: cuantos de los últimos valores notificados debe recordar. Además también se puede especificar la ventana temporal en milisegundos para determinar la antigüedad de los valores grabados.
- A los nuevos suscriptores les suministrará directamente tantos valores como indique el tamaño del búfer siempre que se encuentren dentro de la ventana temporal:

```
let subject = new ReplaySubject(5, 3000);
subject.subscribe({ next: (v) => console.log('observerA: ' + v) });
let observable = interval(1000).pipe(take(10));
observable.subscribe(subject);
setTimeout(() => subject.subscribe({ next: (v) => console.log('observerB: ' + v) }), 4500);
```

AsyncSubject

- AsyncSubject es una variante donde solo se envía a los observadores el último valor de la ejecución del observable y solo cuando se completa la ejecución.

```
let subject = new AsyncSubject();
subject.subscribe({ next: (v) => console.log('observerA: ' + v) });
setTimeout(() => subject.subscribe({ next: (v) =>
    console.log('observerB: ' + v) }), 1500);
let observable = interval(1000).pipe(take(3));
observable.subscribe(subject);
```


ConnectableObservable

- Un Observable arranca cuando se suscribe el primer observador y empieza a notificar los valores.
- Un "Observable unidifusión simple" solo envía notificaciones a un solo Observador que recibe la serie completa de valores.
- Un "Observable multidifusión" pasa las notificaciones a través de un Subject que puede tener muchos suscriptores, que reciben los valores a partir del momento en que se suscriben.
- En determinados escenarios es necesario controlar cuando empieza a generar las notificaciones.
- Con el operador multicast, los observadores se suscriben a un Subject subyacente y el Subject se suscribe a la fuente Observable.
- El operador multicast devuelve un ConnectableObservable que es a su vez un Observable pero que dispone de los métodos connect() y refCount().

connect

- El método connect() permite determinar cuándo comenzará la ejecución compartida del Observable y devuelve una Subscription con el unsubscribe() que permite cancelar la ejecución del Observable compartido.

```
let multicasted = interval(2000).pipe(multicast(() => new BehaviorSubject(0)));
let subscriberA = multicasted.subscribe({ next: (v) => console.log('observerA: ' + v) });
setTimeout(() => subject.subscribe({ next: (v) => console.log('observerB: ' + v) }), 4500);
// …
let subscriptionConnect = multicasted.connect();
// …
subscriberA.unsubscribe(); // Cancela la suscripción
// …
subscriptionConnect.unsubscribe(); // Cancela la ejecución
```


refCount

- Llamar manualmente a connect() y manejar las suscripciones es a menudo engorroso. Por lo general, queremos conectarnos automáticamente cuando llega el primer Observer y cancelar automáticamente la ejecución compartida cuando el último Observer anula la suscripción.
- El método refCount() de ConnectableObservable devuelve un Observable que realiza un seguimiento de cuántos suscriptores tiene.
- Hace que el Observable comience a ejecutarse automáticamente cuando llega el primer suscriptor y deja de ejecutarse cuando el último suscriptor se va.

```
let multicasted = (interval(1000).pipe(multicast(() => new ReplaySubject(5, 3000))) as
    ConnectableObservable<number>).refCount());
// …
let subscriberA = multicasted.subscribe({ next: (v) => console.log('observerA: ' + v) });
// …
subscriberA.unsubscribe();
```

Operadores

- Los operadores son funciones que se basan en un observable para permitir la manipulación sofisticada de colecciones. Por ejemplo, RxJS define operadores como map(), filter(), concat(), y flatMap().
- Los operadores reciben argumentos de configuración y devuelven una función que toma una fuente observable. Al ejecutar esta función devuelta, el operador observa los valores emitidos de la fuente observable, los transforma y devuelve un nuevo observable con los valores transformados.
- Se usan tuberías o canalizaciones para vincular varios operadores juntos. Las canalizaciones permiten combinar múltiples operaciones en una sola operación. El método pipe() recibe como argumentos la lista de operadores que se desea combinar, y devuelve una nueva función que, al ser ejecutada, ejecuta las funciones compuestas en secuencia.
- Un conjunto de operadores aplicados a un observable es una receta, es decir, un conjunto de instrucciones para producir los valores finales. En sí misma, la receta no hace nada. Se debe llamar a subscribe() para producir un resultado a través de la receta.
- RxJS proporciona muchos operadores (más de 150 de ellos), pero solo un puñado se usa con frecuencia.

Operadores

AREA	OPERADORES
Creación	from, fromPromise, fromEvent, of, range, interval, timer, empty, throw, throwError
Combinación	combineLatest, concat, merge, startWith, withLatestFrom, zip
Filtrado	debounceTime, distinctUntilChanged, filter, take, takeUntil
Transformación	bufferTime, concatMap, map, mergeMap, scan, switchMap
Utilidad	tap, delay, toArray, toPromise
Multidifusión	share, cache, multicast, publish
Errores	catch, retry, retryWhen
Agregados	count, max, min, reduce

Operadores

```
const squareOdd = of(1, 2, 3, 4, 5)
  .pipe(
    filter(n => n % 2 !== 0),
    map(n => n * n)
  );
squareOdd.subscribe({ next: x => console.log(x)});
let stream$ = of(5,4,7,-1).pipe(max())
stream$ = of(5,4,7,-1).pipe(
   max((a, b) => a == b ? 0 : (a > b ? 1 : -1))
).subscribe({ next: x => console.log(x)});
const sample = val => of(val).pipe(delay(5000));
const example = sample('First Example')
  .pipe(toPromise())
  .then(result => {
    console.log('From Promise:', result);
  });
```


Scheduler

- Un scheduler controla cuándo se inicia una suscripción y cuándo se entregan las notificaciones. Son despachadores centralizados de control de concurrencia, lo que nos permite coordinar cuando sucede algún “computo”, por ejemplo setTimeout, requestAnimationFrame u otros.
- Consta de tres componentes:
  - Estructura de datos: Sabe cómo almacenar y poner en cola tareas basadas en la prioridad u otros criterios.
  - Contexto de ejecución: Denota dónde y cuándo se ejecuta la tarea. Un ejemplo sería que se ejecutara inmediatamente, también podría ejecutarse en otro mecanismo de devolución de llamada como setTimeout.
  - Reloj (virtual): Proporciona una noción de “tiempo”, para ello se usa el método now(). Las tareas que se programan en un programador en particular se adhieren sólo al tiempo indicado por ese reloj.

Tipos de Scheduler

Scheduler	Propósito
null	Al no pasar ningún planificador, las notificaciones se entregan de forma síncrona y recursiva. Usado para operaciones de tiempo constante o operaciones recursivas de cola.
Rx.Scheduler.queue	Programar en una cola en el marco de evento actual (planificador de trampolín). Usado para operaciones de iteración.
Rx.Scheduler.asap	Programar en la cola de micro tareas, que utiliza el mecanismo de transporte más rápido disponible, ya sea process.nextTick() de Node.js, Web Worker MessageChannel, setTimeout u otros. Usado para conversiones asincrónicas.
Rx.Scheduler.async	La programación funciona con setInterval. Usado para operaciones basadas en tiempo.

Migración v.6

- Re factorización de las rutas de importación.

```
import { Observable, Subject, asapScheduler, pipe, of, from, interval, merge, fromEvent } from 'rxjs';
import { map, filter, scan } from 'rxjs/operators';
```

- El estilo de codificación anterior de los operadores de encadenamiento se ha reemplazado por conectar el resultado de un operador a otro mediante tuberías.
- La creación de diferentes tipos de Observables mediante los correspondientes métodos de clase create() se han reemplazado por funciones y operadores.
- El cambio de estilo de codificación ha obligado a renombrar aquellos que son palabras reservadas en JavaScript:
  - do -> tap
  - catch -> catchError
  - switch -> switchAll
  - finally -> finalize

Migración v.6

Antes

```
source
 .do(rslt => console.log(rslt))
 .map(x => x + x)
 .mergeMap(n => of(n + 1, n + 2)
  .filter(x => x % 1 == 0)
  .scan((acc, x) => acc + x, 0)
 )
 .catch(err => of('error found'))
 .subscribe(printResult);

Observable.if(test, a$, b$);
Observable.throw(new Error());

import { merge } from 'rxjs/operators';
a$.pipe(merge(b$, c$));

obs$ = ArrayObservable.create(myArray);
```

Ahora

```
source.pipe(
 tap(rslt => console.log(rslt))
 map(x => x + x),
 mergeMap(n => of(n + 1, n + 2).pipe(
  filter(x => x % 1 == 0),
  scan((acc, x) => acc + x, 0),
 )),
 catchError(err => of('error found')),
).subscribe(printResult);

iif(test, a$, b$);
throwError(new Error());

import { merge } from 'rxjs';
merge(a$, b$, c$);

obs$ = from(myArray);
```


Observables en Angular

- Angular hace uso de los observables como una interfaz común para manejar gran variedad de operaciones asincrónicas.
- La clase EventEmitter se extiende de Observable.
- El módulo HTTP usa observables para manejar solicitudes y respuestas AJAX.
- Los módulos de Enrutador y Formularios Reactivos usan observables para escuchar y responder a eventos de entrada de usuario.
- El AsyncPipe se suscribe a un observable o promesa y devuelve el último valor que ha emitido. Cuando se emite un nuevo valor, la tubería marca el componente a verificar para ver si hay cambios.

AsyncPipe

- El AsyncPipe (impuro) acepta una Promise o un Observable como entrada que devuelve los valores asíncronamente, a los que se suscribe automáticamente y muestran el valor cuando son resueltas o se recibe el siguiente valor.
- El AsyncPipe es 'con estado', mantiene una suscripción a la entrada Observable y sigue entregando valores del Observable medida que llegan.

```
<li *ngFor="let item of dao.list() | async">
```


Convenciones en la nomenclatura para observables

- Debido a que las aplicaciones Angular están escritas principalmente en TypeScript, normalmente se sabrá cuándo una variable es observable.
- Aunque el Angular no hace cumplir una convención en la nomenclatura para observables, a menudo los observables utilizan como sufijo un signo "$".
- Esto puede ser útil al escanear a través del código en búsqueda de valores observables.
- Además, si se desea que una propiedad almacene el valor más reciente de un observable, puede ser conveniente simplemente usar el mismo nombre con o sin el "$".

Promise Pattern

- El Promise Pattern es un patrón de organización de código que permite encadenar llamadas a métodos que se ejecutaran a la conclusión del anterior (flujos).
- Simplifica y soluciona los problemas comunes con el patrón Callback:
  - Llamadas anidadas
  - Complejidad de código

  o.m(1, 2, f(m1(3, f1(4,5,ff(8))) → o.m(1, 2).f().m1(3).f1(4, 5).ff(8)
- Aunque se utiliza extensamente para las operaciones asíncronas, no es exclusivo de las mismas.
- Las promesas se han incorporado a los objetos estándar de JavaScript en la versión EcmaScript 2015.

Objeto Promise

- Una “promesa” es un objeto que actúa como proxy en los casos en los que no se puede utilizar el verdadero valor porque aún no se conoce (no se ha generado, llegado, …) pero se debe continuar sin esperar a que este disponible (no se puede bloquear la función esperando a su obtención).
- Una “promesa” puede tener los siguientes estados:
  - Pendiente: Aún no se sabe si se podrá o no obtener el resultado.
  - Resuelta: Se ha podido obtener el resultado (Promise.resolve())
  - Rechazada: Ha habido algún tipo de error y no se ha podido obtener el resultado (Promise.reject())
- Los métodos del objeto promesa devuelven al propio objeto para permitir apilar llamadas sucesivas.
- Como objeto, la promesa se puede almacenar en una variable, pasar como parámetro o devolver desde una función, lo que permite aplicar los métodos en distintos puntos del código.

Crear promesas (ES2015)

- El objeto Promise gestiona la creación de la promesa y los cambios de estados de la misma.

```
list() {
  return new Promise((resolve, reject) => {
    this.http.get(this.baseUrl).subscribe( data => resolve(data),  err => reject(err) )
  });
 }
```

- Para crear promesas ya concluidas:
  - Promise.reject: Crea una promesa nueva como rechazada cuyo resultado es igual que el argumento pasado.
  - Promise.resolve: Crea una promesa nueva como resuelta cuyo resultado es igual que su argumento.
- **Un Observable se puede convertir en una promesa**:

```
import 'rxjs/add/operator/toPromise';
list() {
  return this.http.get(this.baseUrl).toPromise();
 }
```


Invocar promesas

- El objeto Promise creado expone los métodos:
  - then(fnResuelta, fnRechazada): Recibe como parámetro la función a ejecutar cuando termine la anterior y, opcionalmente, la función a ejecutar en caso de que falle la anterior.
  - catch(fnError): Recibe como parámetro la función a ejecutar en caso de que falle.

```
list().then(calcular, ponError).then(guardar)
```

- Otras formas de crear e invocar promesas son:
  - Promise.all: Combina dos o más promesas y realiza la devolución solo cuando todas las promesas especificadas se completan o alguna se rechaza.
  - Promise.race: Crea una nueva promesa que resolverá o rechazará con el mismo valor de resultado que la primera promesa que se va resolver o rechazar entre los argumentos pasados.

Patrón Event Bus

- El patrón Event Bus básicamente permite a objetos suscribirse a ciertos eventos del Bus, de modo que cuando un evento es publicado en el Bus se propaga a cualquier suscriptor interesado.

```
@Injectable({ providedIn: 'root' })
export class EventBusService {
  private subject$ = new Subject<EventData>();
  emit(event: EventData): void;
  emit(name: string, value?: any): void;
  emit(eventOrName: EventData | string, value?: any): void {
    if (eventOrName instanceof EventData) {
      this.subject$.next(eventOrName);
    } else {
      this.subject$.next(new EventData(eventOrName ?? '', value))
    }
  }
  on(eventName: string, action: any): Subscription {
    return this.subject$.pipe(
      filter((e: EventData) => e.Name === eventName),
      map((e: EventData) => e.Value)
    ).subscribe(action);
  }
}
export class EventData {
  constructor(private name: string, private value?: any) {}
  public get Name() { return this.name }
  public get Value() { return this.value }
}
this.eventBus.emit('my.event');
this.subscription = this.eventBus.on('my.event', () => {
    :
  })
```


ANGULAR SIGNALS

Reactividad de grano fino

- La reactividad de grano fino es un paradigma de programación que permite la reevaluación automática del código en respuesta a cambios en los datos o el estado. En este paradigma, el código se divide en pequeñas unidades reactivas independientes que son sensibles a los cambios en sus datos de entrada. Cuando los datos de entrada cambian, estas unidades reactivas se vuelven a evaluar automáticamente, actualizando su salida en consecuencia.
- La reactividad de grano fino puede mejorar el rendimiento y la capacidad de mantenimiento de aplicaciones complejas al reducir la necesidad de actualizaciones manuales de la interfaz de usuario y permitir un procesamiento de datos más eficiente. También puede ayudar a garantizar que la interfaz de usuario de la aplicación se mantenga coherente con los datos subyacentes en todo momento, lo que mejora la experiencia del usuario.
- La reactividad de grano fino se usa comúnmente en frameworks de programación como Solid.js, Vue.js y Svelte.

Signals y ZoneJs

- Hasta ahora Angular depende de zone.js para su detección automática de cambios. ZoneJs es una librería que provee un mecanismo llamado zonas para encapsular e interceptar actividades de nuestra aplicación, es decir, cuando ocurre un cambio zonejs lo detecta y activa la detección de cambios de Angular para actualizar el estado de la aplicación, entonces, el framework (Angular) pasa por todos los componentes del árbol para verificar si su estado ha cambiado o no y si el nuevo estado afecta la vista. Si ese es el caso, se actualiza la parte DOM del componente que se ve afectada por el cambio. Esto significa que, a menudo, incluso si no tenemos la intención de actualizar el DOM, Angular deberá verificar el árbol de componentes completo y ver si alguno de los valores de nuestros enlaces de datos debe actualizarse.
- La principal diferencia entre Signals y ZoneJs es que Signals se centra en detectar cambios solo en los componentes de la interfaz de usuario que son necesarios, en lugar de realizar una detección de cambios exhaustiva en todo el árbol de componentes. Esto reduce el impacto en el rendimiento del sistema de detección de cambios y permite una actualización más rápida de la interfaz de usuario.
- Otra diferencia importante es que el uso de Signals también permite la detección de cambios asíncronos, lo que significa que los cambios realizados por eventos que ocurren fuera del ciclo de vida normal de Angular, como eventos del DOM o solicitudes HTTP, también se pueden detectar y actualizar en la interfaz de usuario de manera eficiente.

Angular Signals

- Angular Signals es un sistema que realiza un seguimiento granular de cómo y dónde se usa el estado en una aplicación, lo que permite que el marco optimice las actualizaciones de representación.
- Una señal es un envoltorio alrededor de un valor que puede notificar a los consumidores interesados cuando cambia su valor. Las señales pueden contener cualquier valor, desde primitivas simples hasta estructuras de datos complejas.
- El valor de una señal siempre se lee a través de una función getter, que permite a Angular rastrear dónde se usa la señal.
- Las señales pueden ser de lectura/escritura o de solo lectura.
- Angular suministra 3 primitivas para manejar las señales:
  - signal()
  - compute()
  - effect()

signal()

- La primitiva signal() crea e inicializa una señal de lectura/escritura del tipo genérico WritableSignal<>.

```
public count: WritableSignal<number> = signal(0);
```

- Se utiliza como función (getter) para consultar o vincular su valor:

```
console.log('The count is: ' + this.count());
<output>{{count()}}<output>
```

- Para establezcer directamente la señal en un nuevo valor y notificar a los dependientes:

```
this.count.set(3);
```

- Para sustituir el valor de la señal en función de su valor actual y notificar a los dependientes:

```
this.count.update(oldValue => oldValue + 1);
```

- Con el método asReadonly() se devuelve una versión de solo lectura de la señal. Se puede acceder a las señales de solo lectura para leer su valor, pero no se pueden cambiar mediante métodos set o update.

```
const readOnlyCount: Signal<number> = this.count.asReadonly();
```

compute()

- La primitiva compute() permite crear señales derivadas de otras señales. Son señales de solo lectura del tipo genérico Signal<number>.
- Para crear la señal se debe especificar la función de calculo en la que deben intervenir otras señales:

```
const count: WritableSignal<number> = signal(0);
const doubleCount: Signal<number> = computed(() => count() * 2);
```

- La primitiva captura las notificaciones de las señales que participan en la función de calculo, de tal forma que la función no se ejecuta para calcular su valor hasta que se lee por primera vez. Una vez calculado, el valor se almacena en caché para las futuras lecturas que devolverán el valor almacenado en caché sin volver a calcular. Cuando recibe una notificación de una de sus señales invalida la cache para que se vuelva a calcular en la próxima lectura.
- Se utiliza como función (getter) para consultar o vincular su valor.
- No se pueden asignar valores directamente a una señal calculada, se produce un error de compilación, Signal no tiene el método set(): ~~doubleCount.set(3);~~

Igualdad

- Al crear una señal, se puede proporcionar una función de igualdad, que se utilizará para verificar si el nuevo valor es realmente diferente al anterior.

```
prov = signal({ id: 1, nombre: 'Madrid' }, { equal: (a, b) => a.id === b.id })
prov.set({ id: 1, nombre: 'madrid' }) // no cambia
prov.set({ id: 2, nombre: 'Madrid' }) // cambia
```

- Si la función de igualdad determina que dos valores son iguales:
  - bloquea la actualización del valor de la señal
  - omite la propagación de los cambios
- Las funciones de igualdad se pueden proporcionar tanto a las señales de lectura/escritura como a las calculadas.

effect()

- Las señales son útiles porque pueden notificar a los consumidores interesados cuando cambian. Un efecto es una operación que se ejecuta cada vez que cambian uno o más valores de señal.
- Los efectos siempre se ejecutan al menos una vez. Cuando se ejecuta un efecto, rastrea cualquier lectura de valor de señal. Cada vez que cambia alguno de estos valores de señal, el efecto vuelve a ejecutarse. Al igual que las señales calculadas, los efectos realizan un seguimiento de sus dependencias de forma dinámica y solo rastrean las señales que se leyeron en la ejecución más reciente.
- Los efectos son raramente necesarios en la mayoría de los códigos de aplicación, pero pueden ser útiles en circunstancias específicas, podría ser una buena solución para:
  - Mostar datos de registro y cuándo cambian, ya sea para análisis o como herramienta de depuración
  - Mantener los datos sincronizados con window.localStorage
  - Agregar un comportamiento DOM personalizado que no se puede expresar con la sintaxis de la plantilla
  - Optimizar la representación personalizada en un <canvas>, una biblioteca de gráficos u otra biblioteca de interfaz de usuario de terceros
- No deben usarse los efectos para la propagación de cambios de estado porque pueden generar actualizaciones circulares infinitas o ciclos de detección de cambios innecesarios.
- Los efectos siempre se ejecutan de forma asíncrona, durante el proceso de detección de cambios.

effect()

- Para registrar un nuevo efecto se requiere un "contexto de inyección" (acceso a inject()) disponibles en componentes, directivas, servicios y funciones factoría de servicios.
- Se pueden crear en el constructor:

```
constructor() {
  effect(() => { console.log(`The count is: ${this.count()})`); });
}
```

- Se pueden crear como inicializador de un atributo (campo) de la clase, que también le da un nombre descriptivo:

```
private loggingEffect = effect(() => { console.log(`The count is: ${this.count()})`); });
```

- Para crear un efecto en un método fuera del constructor, se debe pasar un Injector al effect a través de sus opciones:

```
constructor(private injector: Injector) {}
init() {
  effect(() => { console.log(`The count is: ${this.count()})`); }, {injector: this.injector});
}
```

effect()

- Los efectos pueden iniciar operaciones de larga duración, que deben cancelarse si el efecto se destruye o se vuelve a ejecutar antes de que finalice la primera operación. Cuando se crea un efecto, en la función se puede definir un parámetro donde se le inyecta la función para registrar la devolución de llamada que el efecto debe invocar antes de que comience la siguiente ejecución del efecto, o cuando se destruye el efecto.

```
effect((onCleanup) => {
  const user = currentUser();
  const timer = setTimeout(() => {
    console.log(`1 second ago, the user became ${user}`);
  }, 1000);
  onCleanup(() => { clearTimeout(timer); });
});
```


Interoperabilidad RxJS

- toSignal(): permite crear una señal que rastree el valor de un Observable. Se comporta de manera similar al pipe async, pero es más flexible y se puede usar en cualquier lugar de una aplicación. toSignal se suscribe al Observable inmediatamente, lo que puede desencadenar efectos secundarios, y dicha suscripción se cancela automáticamente cuando el componente o servicio que llama a toSignal se destruye.

```
counterSignal = toSignal(this.counter$, {initialValue: 0});
```

- toObservable(): permite crear un Observable (ReplaySubject) que rastrea el valor de una señal con un effect y emite el valor del Observable cuando cambia. Al suscribirse, el primer valor (si está disponible) se emite de forma síncrona y los posteriores serán asíncronos. A diferencia de los Observables, las señales nunca proporcionan una notificación síncrona de cambios: si se actualiza el valor de la señal varias veces, solo emitirá el valor después de que la señal se estabilice.

```
counter$ = toObservable(counterSignal);
```

Developer Preview

- Signal inputs:
  - Las señales input son una alternativa reactiva a las basadas en decoradores @Input().
- Model inputs:
  - Las señales model definen tanto una entrada como una salida. son una alternativa reactiva al enlace bidireccional de propiedades simples.
- Signal queries
  - Un componente o directiva puede definir consultas que encuentren elementos secundarios y lean valores de sus inyectores para recuperar referencias a componentes, directivas, elementos DOM y más. Hay dos categorías de consultas: consultas de vista (elementos en la propia plantilla del componente) y consultas de contenido (elementos anidados en el contenido del componente).
    - viewChild: Para declarar una consulta de vista con un único resultado
    - viewChildren: Para declarar una consulta de vista con múltiples resultados
    - contentChild: Para declarar una consulta de contenido con un único resultado
    - contentChildren: Para declarar una consulta de contenido con múltiples resultados

FORMULARIOS

Formularios

- Un formulario crea una experiencia de entrada de datos coherente, eficaz y convincente.
- Un formulario Angular coordina un conjunto de controles de usuario enlazados a datos bidireccionalmente, hace el seguimiento de los cambios, valida la entrada y presenta errores.
- Angular ofrece dos tecnologías para trabajar con formularios: formularios basados en plantillas y formularios reactivos. Pero divergen marcadamente en filosofía, estilo de programación y técnica. Incluso tienen sus propios módulos: FormsModule y ReactiveFormsModule.
- Como el API de gestión de formularios cuenta con su propio módulo, es necesario importar y referenciar en el módulo principal:

```
import { FormsModule } from '@angular/forms';
@NgModule({
  imports: [ BrowserModule, FormsModule ],
  // …
})
export class AppModule { }
```


Formularios Reactivos

- Los Formularios Reactivos facilitan un estilo reactivo de programación que favorece la gestión explícita de los datos que fluyen entre un modelo de datos (normalmente recuperado de un servidor) y un modelo de formulario orientado al UI que conserva los estados y valores de los controles HTML en la pantalla. Los formularios reactivos ofrecen la facilidad de usar patrones reactivos, pruebas y validación.
- Los formularios reactivos permiten crear un árbol de objetos Angular de control de formulario (AbstractControl) en la clase del componente y vincularlos a elementos de control de formulario nativos (DOM) en la plantilla de componente.
- Así mismo, permite crear y manipular objetos de control de formulario directamente en la clase de componente. Como la clase del componente tiene acceso inmediato tanto al modelo de datos como a la estructura de control de formulario, puede trasladar los valores del modelo de datos a los controles de formulario y retirar los valores modificados por el usuario.
- El componente puede observar los cambios en el estado del control en el formulario y reaccionar ante esos cambios.

Formularios Reactivos

- Una ventaja de trabajar directamente con objetos de control de formulario es que las actualizaciones de valores y las validaciones siempre son sincrónicas y bajo tu control. No encontrarás los problemas de tiempo que a veces plaga los formularios basados en plantillas. Las pruebas unitaria pueden ser más fáciles en los formularios reactivos al estar dirigidos por código.
- De acuerdo con el paradigma reactivo, el componente preserva la inmutabilidad del modelo de datos, tratándolo como una fuente pura de valores originales. En lugar de actualizar el modelo de datos directamente, la clase del componente extrae los cambios del usuario y los envía a un componente o servicio externo, que hace algo con ellos (como guardarlos) y devuelve un nuevo modelo de datos al componente que refleja el estado del modelo actualizado en el formulario.
- El uso de las directivas de formularios reactivos no requiere que se sigan todos los principios reactivos, pero facilita el enfoque de programación reactiva si se decide utilizarlo.

Formularios Reactivos

- Como el API de gestión de formularios cuenta con su propio módulo, es necesario importar y referenciar en el módulo principal:

```
import { ReactiveFormsModule } from '@angular/forms';
@NgModule({
  imports: [ BrowserModule, ReactiveFormsModule ],
  // …
})
export class AppModule { }
```

- Se pueden usar los dos paradigmas de formularios en la misma aplicación.
- La directiva ngModel no forma parte del ReactiveFormsModule, para poder utilizarla hay importar FormsModule y, dentro de un formulario reactivo, hay que acompañarla con:

```
[ngModelOptions]="{standalone: true}"
```

Clases esenciales

- **AbstractControl<>**: es la clase base abstracta para las tres clases concretas de control de formulario: FormControl, FormGroup y FormArray. Proporciona sus comportamientos y propiedades comunes, algunos son observables.
- **FormControl<>**: rastrea el valor y el estado de validez de un control de formulario individual. Corresponde a un control de formulario HTML, como un <input>, <select> o <textarea>.
- **FormGroup<>**: rastrea el valor y el estado de validez de un grupo de instancias de AbstractControl. Las propiedades del grupo incluyen sus controles hijos. El formulario en si es un FormGroup.
- **FormArray<>**: rastrea el valor y el estado de validez de una matriz indexada numéricamente de instancias de AbstractControl.
- **FormBuilder**: es un servicio que ayuda a generar un FormGroup mediante una estructura que reduce la complejidad, la repetición y el desorden al crearlos.

Creación del formulario

- Hay que definir en la clase del componente una propiedad publica de tipo FormGroup a la que se enlazará la plantilla.
- Para el modelo:

```
model = { user: 'usuario', password: 'P@$$w0rd', roles: [{ role: 'Admin'
    }, { role: 'User' }] };
```

- Hay que definir los diferentes FormControl a los que se enlazaran los controles de la plantilla instanciándolos con el valor inicial y, opcionalmente, las validaciones:

```
let pwd = new FormControl('P@$$w0rd', Validators.minLength(2));
```

Creación del formulario

- Para instanciar el FormGroup pasando un array asociativo con el nombre del control con la instancia de FormControl y, opcionalmente, las validaciones conjuntas:

```
const form = new FormGroup({
  password: new FormControl('', Validators.minLength(2)),
  passwordConfirm: new FormControl('', Validators.minLength(2)),
}, passwordMatchValidator);
```

- El FormGroup es una colección de AbstractControl que se puede gestionar dinámicamente con los métodos addControl, setControl y removeControl.

Sub formularios

- Un FormGroup puede contener otros FormGroup (sub formularios), que se pueden gestionar, validar y asignar individualmente:

```
const form = new FormGroup({
  user: new FormControl(''),
  password: new FormGroup({
     passwordValue: new FormControl('', Validators.minLength(2)),
     passwordConfirm: new FormControl('', Validators.minLength(2)),
   }, passwordMatchValidator),
});
```

Agregaciones

- Un FormGroup puede contener uno o varios arrays (indexados numéricamente) de FormControl o FormGroup para gestionar las relaciones 1 a N.
- Hay que crear una instancia de FormArray y agregar un FormGroup por cada uno de los N elementos.

```
const fa = new FormArray(this.model.roles.map(
    item => new FormGroup({ role: new FormControl(item.role, Validators.required) })
));
this.miForm = new FormGroup({
  user: new FormControl(''),
  password: // ...
  roles: fa
});
```


Agregaciones

- Para añadir un nuevo elemento a la agregación:

```
addRole() {
  (this.miForm.get('roles') as FormArray).push(
      new FormGroup({ role: new FormControl('', Validators.required) })
  );
}
```

- Para eliminar un elemento de la agregación:

```
deleteRole(ind: number) {
  (this.miForm.get('roles') as FormArray).removeAt(ind);
}
```

Validaciones

- La clase Validators expone las validaciones predefinidas en Angular y se pueden crear funciones de validación propias.
- Las validaciones se establecen cuando se instancian los FormControl o FormGroup. En caso de múltiples validaciones se suministra una array de validaciones:

```
this.miForm = new FormGroup({
  user: new FormControl('', [Validators.required, Validators.minLength(2),
            Validators.maxLength(20)]),
  password: new FormGroup({…}, passwordMatchValidator);
  roles: fa
});
```


Validaciones personalizadas

- Para crear un validación personalizada se crea una factoría (función que devuelve una función) que recibe un numero arbitrario de parámetros y devuelve una función de validación configurada (función que se invocara en las validaciones). La firma de la función de validación devuelta es:
  - (control: AbstractControl): { [key: string]: any } | ValidationErrors | null
- La función recibe el AbstractControl al que se aplica la validación, usando la propiedad control.value se accede al valor a validar. No puede recibir mas argumentos pero puede utilizar los parámetros de la factoría vía clausura.
- Devuelve un valor nulo si el valor del control es válido o un objeto de error de validación. El objeto de error de validación normalmente tiene una propiedad cuyo nombre es la clave de validación y cuyo valor no debe ser falsify y puede ser escalar o un diccionario arbitrario de valores que se pueden insertar en un mensaje de error. El resultado se mezcla con los resultados del resto de validaciones en la colección AbstractControl.errors, donde las claves sirven para identificar los errores producidos. Por convenio el valor vacío del control se considera siempre valido para combinar con required cuando no lo sea.
- Los validadores asíncronos deben devolver una promesa u observable que luego emita el nulo u objeto de error de validación. En el caso de un observable, el observable debe completarse, momento en el que el formulario utiliza el último valor emitido para la validación.

Validaciones personalizadas

- Para un FormControl (por convenio: sin valor se considera valido salvo en required):

```
export function naturalNumberValidator(): ValidatorFn {
   return (control: AbstractControl): ValidationErrors | null => {
      return (!control.value || /^[1-9]\d*$/.test(control.value)) ? null : { naturalNumber: { valid: false
            } };
   };
}
```

- Para un FormGroup:

```
export const passwordMatchValidator: ValidatorFn = (control: AbstractControl): ValidationErrors |
     null => {
  const pwdOrg = control.get('passwordValue');
  const pwdCopy = control.get('passwordConfirm');
  return pwdOrg && pwdCopy && pwdOrg.value === pwdCopy.value ? null : {
     passwordMatchValidator: true };
};
```


Enlace de datos

- Según el paradigma reactivo, el componente preserva la inmutabilidad del modelo de datos, por lo que es necesario traspasar manualmente los datos del modelo a los controles y viceversa.
- Con los setValue y patchValue se pasan los datos hacia el formulario. El método setValue comprueba minuciosamente el objeto de datos antes de asignar cualquier valor de control de formulario. No aceptará un objeto de datos que no coincida con la estructura FormGroup o falte valores para cualquier control en el grupo. patchValue permite actualizaciones parciales y tiene más flexibilidad para hacer frente a datos divergentes y fallará en silencio.

```
this.miForm.setValue(this.modelo);
this.miForm.get('user').setValue(this.modelo.user);
```

- Para recuperar los datos del formulario se dispone de la propiedad value:

```
aux = this.miForm.value;
this.modelo.user = this.miForm.get('user').value;
```

- Para borrar los datos del formulario o control:

```
this.miForm.reset();
```

Enlace de datos

- Para modificar el estado de un FormGroup o FormControl:

```
const control = miForm.get('user')
control.markAsTouched();
control.markAsUntouched();
control.markAsDirty();
control.markAsPristine();
```

- Para cambiar los validadores:

```
const validators: ValidatorFn[] = […];
this.miForm.setValidators(validators);
this.miForm.get('user').setValidators(validators);
```

- Para borrar los validadores del formulario o control:

```
this.miForm.clearValidators();
this.miForm.get('user').clearValidators();
```


Detección de cambios

- Los herederos de AbstractControl exponen Observables, a través de valueChanges y statusChanges, que permiten suscripciones que detectan los cambios en el formulario y sus controles.
- Para un formulario o FormGroup:

```
this.miForm.valueChanges
  .subscribe(data => {
    // data: datos actuales del formulario
  });
```

- Para un FormControl:

```
this.miForm.get('user')?.valueChanges
  .subscribe(data => {
    // data: datos actuales del control
  });
```

Eventos (v.18)

- Las clases FormControl, FormGroup y FormArray exponen una propiedad observable llamada events, que permite suscribirse a un flujo de eventos para un control del formulario. Con ella, se puede realizar un seguimiento unificado de los cambios en las propiedades value, status, pristine o touched del AbstractControl.

```
const nameControl = new FormControl<string|null>('name', Validators.required);
nameControl.events.subscribe(event => {
  // process the individual events
});
```

- El tipo de evento se identifica como instancia de: TouchedChangeEvent, PristineChangeEvent, StatusChangeEvent, ValueChangeEvent, FormResetEvent and FormSubmitEvent.

FormBuilder

- El servicio FormBuilder facilita la creación de los formularios. Es necesario inyectarlo.

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

- Mediante una estructura se crea el formulario:

```
this.miForm = this.fb.group({
  user: [this.model.user, [Validators.required, Validators.minLength(2),
          Validators.maxLength(20)]],
  password: this.fb.group({
     passwordValue: [this.model.password, Validators.minLength(2)],
     passwordConfirm: '',
   }, { validator: passwordMatchValidator }),
  roles: this.fb.array(this.model.roles.map(item => this.fb.group({ role: [item.role,
          Validators.required] })))
});
```

Directivas

- formGroup: para vincular la propiedad formulario al formulario.
  <form [formGroup]="miForm" >
- formGroupName: establece una etiqueta contenedora para un subformulario (FormGroup dentro de otro FormGroup).

- formControlName: para vincular <input>, <select> o <textarea> a su correspondiente FormControl. Debe estar correctamente anidado dentro de su formGroup o formControlName.
  <input type="password" formControlName="passwordValue" >
- formArrayName: establece una etiqueta contenedora para un array de FormGroup. Se crea un formGroupName con el valor del índice del array.

```
<div formArrayName="roles">
    <div *ngFor="let row of elementoForm.get('roles').controls; let i=index" [formGroupName]="i" >
        <input type="text" formControlName="role" >
</div></div>
```


Mostrar errores

- Errores en los controles del formulario:

```
<!-- null || true -->
<output class="errorMsg" [hidden]="!miForm?.controls['user'].errors?.required">Es obligatorio.</output>
<output class="errorMsg" [hidden]="!miForm?.get('user')?.errors?.required">Es obligatorio.</output>
<!-- false || true -->
<output class="errorMsg" [hidden]="!miForm.hasError('required', 'user')">Es obligatorio.</output>
```

- Errores en los controles de sub formularios:

```
{{miForm?.get('password')?.get('passwordValue')?.errors | json}}
{{miForm?.get(['password', 'passwordValue'])?.errors | json}}
```

- Errores en las validaciones del FormGroup:

```
{{miForm?.get('password')?.errors | json}}
```

- Errores en los elementos de un FormArray:

```
<ul *ngFor="let row of miForm.get('roles').controls … >
  {{row?.get('role')?.errors | json}}
```

Plantilla (I)

```
<form [formGroup]="miForm">
  <label>User: <input type="text" formControlName="user" ></label>
  {{miForm?.get('user')?.errors | json}}
  <fieldset formGroupName="password" >
   <label>Password: <input type="password"
      formControlName="passwordValue" ></label>
   {{miForm?.get('password')?.get('passwordValue')?.errors | json}}
   <label>Confirm Password: <input type="password"
      formControlName="passwordConfirm" ></label>
  </fieldset>
  {{miForm?.get('password')?.errors | json}}
```


Plantilla (II)

```
  <div formArrayName="roles">
   <h4>Roles</h4><button (click)="addRole()">Add Role</button>
   <ul *ngFor="let row of $any(miForm.get('roles')).controls; let i=index"
        [formGroupName]="i">
    <li>{{i + 1}}: <input type="text" formControlName="role">
    {{row?.get('role')?.errors | json}}
    <button (click)="deleteRole(i)">Delete</button></li>
   </ul>
  </div>
  <button (click)="send()">Send</button>
</form>{{ miForm.value | json }}
```

Comprobación estricta de tipos (v14)

- En las versiones anteriores de Angular Reactive Forms el valor emitido por el formulario era de tipo any, el marco no tenía mucha información disponible sobre los tipos de cada control de formulario por lo que no disponía de seguridad de tipos (comprobación estricta de tipos).
- Pero a partir de Angular 14 y en adelante, Angular ha agregado la seguridad de tipos completa integrada a Reactive Forms, lo que significa que se pueden recibir mensajes de error útiles y permite el autocompletado cuando se trabaja con valores de formulario.
- Se han reemplazado las clases FormControl, FormGroup y FormArray por sus versiones genéricas que permiten establecer explícitamente o inferir el tipo del valor de inicialización.
- Para mantener la retro compatibilidad, han aparecido las versiones UntypedFormControl, UntypedFormGroup y UntypedFormArray sin tipo (tipo any).
- Todos los FormControl permiten la nulabilidad salvo que se utilice en el constructor la opción {nonNullable: true}, en cuyo caso .reset restablece a su valor inicial en lugar de null.
- En los FormArray, el parámetro de tipo corresponde al tipo de cada control interno. Si se desea tener varios tipos de elementos diferentes se debe usar un UntypedFormArray.

Comprobación estricta de tipos

- Algunos FormGroup (grupos dinámicos) tienen controles que pueden o no estar presentes, que se pueden agregar y eliminar en tiempo de ejecución. En estos casos se debe definir un interface o tipo usando campos opcionales con las claves. Las operaciones .addControl y .removeControl están restringidas a los opcionales.
- Cuando las claves no se conocen de antemano, la clase FormRecord permite añadirlos dinámicamente siempre que todos los controles sean del mismo tipo.

```
const addresses = new FormRecord<FormControl<string|null>>({});
addresses.addControl('Andrew', new FormControl('2340 Folsom St'));
```

- Si se necesita un FormGroup que sea tanto dinámico (abierto) como heterogéneo (los controles son de diferentes tipos) solo se puede usar UntypedFormGroup.
- La clase FormBuilder agrega los controles como nullables. Se ha añadido .nonNullable como una forma abreviada de especificar {nonNullable: true} en cada control y eliminar un texto repetitivo significativo de formularios grandes que no aceptan valores NULL
- También se puede inyectar como servicio usando el nombre NonNullableFormBuilder.

Sincronismo

- Los formularios reactivos son síncronos y los formularios basados en plantillas son asíncronos.
- En los formularios reactivos, se crea el árbol de control de formulario completo en el código. Se puede actualizar inmediatamente un valor o profundizar a través de los descendientes del formulario principal porque todos los controles están siempre disponibles.
- Los formularios basados en plantillas delegan la creación de sus controles de formulario en directivas. Para evitar errores "changed after checked", estas directivas tardan más de un ciclo en construir todo el árbol de control. Esto significa que debe esperar una señal antes de manipular cualquiera de los controles de la clase de componente.
- Por ejemplo, si se inyecta el control de formulario con una petición @ViewChild (NgForm) y se examina en el ngAfterViewInit, no tendrá hijos. Se tendrá que esperar, utilizando setTimeout, antes de extraer un valor de un control, validar o establecerle un nuevo valor.
- La asincronía de los formularios basados en plantillas también complica las pruebas unitarias. Se debe colocar el bloque de prueba en waitForAsync() o en fakeAsync() para evitar buscar valores aún no disponibles en el formulario. Con los formularios reactivo, todo está disponible tal y como se espera que sea.

API DE COMPONENTES INDEPENDIENTES

Componentes independientes (v15)

- Los componentes independientes proporcionan una forma simplificada de crear aplicaciones Angular a partir de la versión 14.2 (experimental) y la 15 (estable).
- Los componentes independientes, las directivas y las canalizaciones (pipes) tienen como objetivo optimizar la experiencia de creación al reducir la necesidad de NgModules.
- Las aplicaciones existentes pueden adoptar de forma incremental y opcional el nuevo estilo independiente sin ningún cambio importante.
- Los componentes, las directivas y las canalizaciones ahora se pueden marcar como standalone: true. Las clases Angular marcadas como autónomas no necesitan declararse en un NgModule (el compilador Angular informará un error si lo intenta).
- Los componentes independientes especifican sus dependencias directamente en lugar de pasarla a través NgModule.
- Los imports de los componente independientes también se puede usar para hacer referencia a directivas y canalizaciones autónomas. De esta forma, los componentes independientes se pueden escribir sin necesidad de crear un archivo NgModule para administrar las dependencias de la plantilla.

Arranque de la aplicación

- Creación de un proyecto standalone:
  - ng new --standalone
- Modular:

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

- Operación de arranque independiente:

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

bootstrapApplication(AppComponent);
```

Componentes independientes (v15)

- Los componentes se pueden marcar como standalone: true y no necesitan declararse en un NgModule (da error si se intenta), pero deben especificar explícitamente sus dependencias con la propiedad imports (NgModules y componentes, directivas y pipes independientes).

```
@Component({
  selector: 'photo-gallery',
  standalone: true,
  imports: [ImageGridComponent, MatButtonModule, ngIf, ngForOf],
  template: `
    <image-grid [images]="imageList"></image-grid>
    <button mat-button>Next Page</button>
  `,
})
export class PhotoGalleryComponent {
```

- Se han refactorizado las directivas y pipes integrados de los módulos estándar de Angular a sus versiones autónomas para proporcionar una forma directa de importación.

Inyección de Dependencias

- El módulo principal dispone de la propiedad providers para registrar los servicios e importa los providers de los módulos importado.
- La operación de arranque independiente se basa en la configuración explícita de una lista de proveedores para la inyección de dependencias. La configuración se realiza a través de la propiedad providers del segundo parámetro de bootstrapApplication().
- En Angular, las funciones de los módulos prefijadas con provide se usan para importar los providers de un módulos. Si un módulo no ofrece dichas funciones, se puede usar la utilidad importProvidersFrom() con el módulo para importar sus providers en contextos independientes.
- Para inicializar la aplicación (anteriormente en el constructor del módulo principal), se usa el nuevo token múltiple ENVIRONMENT_INITIALIZER.

```
bootstrapApplication(AppComponent,  {
  providers: [
    { provide: ERROR_LEVEL, useValue: environment.ERROR_LEVEL },
    { provide: ENVIRONMENT_INITIALIZER, multi: true, useValue: () => inject(AppService).init() }
    provideRouter(MAIN_ROUTES), importProvidersFrom(SecurityModule)
  ]
});
```

Interceptores

- En la operación de arranque independiente, los providers de HttpClientModule se deben registrar con provideHttpClient(). Para pasar los parámetros de configuración se usan las funciones withInterceptors, withInterceptorsFromDi, withXsrfConfiguration, withNoXsrfProtection, withJsonpSupport y withRequestsMadeViaParent.
- Desde Angular v15, también se pueden usar interceptores funcionales (para la inyección de dependencia se usa inject()) del tipo HttpInterceptorFn:

```
export function ajaxWaitInterceptor(req: HttpRequest<unknown>, next: HttpHandlerFn): Observable<HttpEvent<unknown>> {
  const srv: AjaxWaitService = inject(AjaxWaitService);
  srv.Mostrar();
  return next(req).pipe(finalize(() => srv.Ocultar()));
}
```

- Los interceptores de HTTP_INTERCEPTORS se configuran con withInterceptorsFromDi() y los interceptores funcionales con withInterceptors().

```
bootstrapApplication(AppComponent, {
  providers: [
    { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true, },
    provideHttpClient(withInterceptorsFromDi(), withInterceptors([ajaxWaitInterceptor])),
  ]
});
```


Configuración del enrutador

- En la operación de arranque independiente, el servicio Router y las rutas dadas se configuran con provideRouter([]):

```
bootstrapApplication(AppComponent, {
  providers: [ provideRouter([
      { path: '', component: HomeComponent, pathMatch: 'full', title: 'Home' },
      :
    ]),
    :
  ]});
```

- Cualquier ruta puede cargar un componente independiente con loadComponent:

```
{ path: 'grafico', loadComponent: () => import('./grafico-svg/grafico-svg.component'), },
{ path: 'panel', loadComponent: () => import('./admin/panel.component')
    .then(mod => mod.AdminPanelComponent)},
```

Lazy loading

- La operación loadChildren admite la carga de una tabla de rutas sin necesidad de escribir una carga diferida NgModule que importa RouterModule.forChild para declarar las rutas. Esto permite muchas rutas a la vez pero solo funciona cuando cada ruta cargada de esta manera usa un componente independiente.

```
{path: 'admin', loadChildren: () => import('./admin/routes').then(mod => mod.ADMIN_ROUTES)},
// ./admin/routes.ts
export const ADMIN_ROUTES: Route[] = [
  {path: '', component: AdminHomeComponent},
  {path: 'users', component: AdminUsersComponent},
  // ...
];
```

- Al usar loadChildren y loadComponent, el enrutador entiende y resuelve automáticamente las llamadas dinámicas import() con exportaciones default. Se puede aprovechar esto para omitir las operaciones .then() de carga diferida.
- El enrutador ahora admite especificar explícitamente providers adicionales en un Route, lo que permite brindar servicios solo a un subconjunto de rutas en la aplicación.

```
{ path: 'admin', providers: [ {provide: ADMIN_API_KEY, useValue: '12345'}, ], children: [ …
```


Inyección de dependencias con rutas

- La API de carga diferida con loadChildren crea un nuevo inyector de "módulo" cuando carga los hijos de una ruta cargados de forma diferida. Esta característica a menudo era útil para brindar servicios solo a un subconjunto de rutas en la aplicación.
- El enrutador ahora admite especificar explícitamente providers adicionales en un Route, lo que permite brindar servicios solo a un subconjunto de rutas en la aplicación.

```
{ path: 'admin', providers: [ {provide: ADMIN_API_KEY, useValue: '12345'}, ], children: [ …
```

- También es posible con loadChildren combinar con providers en una configuración de enrutamiento adicional.

```
{path: 'admin', loadChildren: () => import('./admin/routes').then(mod => mod.ROUTES)},
export const ROUTES: Route[] = [
  { path: '',  pathMatch: 'prefix',  providers: [AdminService, ],  children: […
```

Migrar un proyecto existente a independiente

- A partir de la versión 15.2.0, Angular ofrece un schema para ayudar a convertir los proyectos existentes a las nuevas API autónomas. El schema tiene como objetivo transformar la mayor cantidad de código posible automáticamente, pero puede requerir algunas correcciones manuales. Antes de migrar, hay que asegurarse de que el proyecto:
  - Está usando Angular 15.2.0 o posterior.
  - No tiene ningún error de compilación.
  - Está en una rama limpia de Git y todo el trabajo esta confirmado.
- Se ejecuta el esquema con el siguiente comando:
  - ng generate @angular/core:standalone
- El proceso de migración se compone de tres pasos por lo que se tendrá que ejecutarlo varias veces y verificar manualmente que el proyecto se compila y se comporta como se esperaba:
  1. "Convert all components, directives and pipes to standalone"
  2. "Remove unnecessary NgModule classes"
  3. "Bootstrap the project using standalone APIs"

Rendimiento: Server-Side Rendering

ANGULAR UNIVERSAL (SSR)

Angular Universal

- Una aplicación Angular normal se ejecuta en el navegador, representando páginas en el DOM en respuesta a las acciones del usuario. Angular Universal permite ejecutar la aplicación Angular del navegador en el servidor con un mínimo de configuración, modificaciones y restricciones (isomorfas).
- La representación del lado del servidor (SSR) es un proceso que implica representar páginas en el servidor, lo que da como resultado un contenido HTML inicial que contiene el estado inicial de la página. Una vez que el contenido HTML se entrega a un navegador, Angular inicializa la aplicación y utiliza los datos contenidos en el HTML.
- Las principales ventajas de SSR en comparación con la renderización del lado del cliente (CSR) son:
  - Rendimiento mejorado: SSR puede mejorar el rendimiento de las aplicaciones web al entregar HTML completamente renderizado al cliente, que se puede analizar y mostrar incluso antes de descargar el JavaScript de la aplicación. Esto puede resultar especialmente beneficioso para los usuarios con conexiones de ancho de banda reducido o dispositivos móviles.
  - Core Web Vitals mejorado: SSR da como resultado mejoras de rendimiento que se pueden medir utilizando estadísticas Core Web Vitals (CWV, Métricas web esenciales), como la reducción del primer procesamiento de imagen con contenido (FCP) y con contenido más grande (LCP), así como el cambio de diseño acumulativo (CLS).
  - Mejor SEO: SSR puede mejorar la optimización del motor de búsqueda (SEO) de las aplicaciones web al facilitar que los motores de búsqueda rastreen e indexen el contenido de la aplicación.

Terminología

- Renderización
  - Renderización del cliente (CSR: Client-side rendering): Renderizar una app en un navegador mediante JavaScript para modificar el DOM.
  - Renderización del servidor (SSR: Server-side rendering): Se procesa una app universal o del cliente en HTML en el servidor.
  - Renderización previa (SSG: Static Site Generation): Ejecutar una aplicación del cliente en el tiempo de compilación para generar su estado inicial como código HTML estático.
  - Hidratación o rehidratación: Inicia vistas JavaScript en el cliente de modo que vuelvan a usar los datos y el árbol del DOM del HTML renderizado por el servidor.
- Rendimiento
  - Tiempo hasta el primer byte (TTFB): Se considera el tiempo que transcurre entre el clic en un vínculo y la primera parte del contenido que llega.
  - First Contentful Paint (FCP): Es el momento en que se hace visible el contenido solicitado (cuerpo del artículo, etcétera).
  - Interaction to Next Paint (INP): Se trata de una métrica representativa que evalúa si una página responde de forma coherente y rápida a las entradas del usuario.
  - Tiempo de bloqueo total (TBT): Es una métrica proxy para INP, que calcula la cantidad de tiempo que se bloqueó el subproceso principal durante la carga de la página.

Estructura de directorios de soporte

- src
- .editorconfig
- .gitignore
- angular.json
- package-lock.json
- package.json
- README.md
- **server.ts**
- tsconfig.json
- tsconfig.app.json
- tsconfig.spec.json

- Para crear una nueva aplicación con SSR:
  - ng new --ssr
- Para agregar SSR a un proyecto existente:
  - ng add @angular/ssr

El archivo server.ts configura un servidor Node.js Express y la representación del lado del servidor de la aplicación Angular. Utiliza CommonEngine del módulo @angular/ssr para el renderizado del front-end de la aplicación Angular. Angular CLI creará una implementación inicial del servidor centrada en la representación del lado del servidor de su aplicación Angular que se puede ampliar para admitir otras funciones, como servicios REST (API), redirecciones, activos estáticos y más.

Estructura de directorios de aplicación

- app
  - app.component.css
  - app.component.html
  - app.component.spec.ts
  - app.component.ts
  - app.config.ts
  - **app.config.server.ts**
  - app.routes.ts
- assets
- favicon.ico
- index.html
- main.ts
- **main.server.ts**
- styles.css

- main.ts: Arranque del módulo principal de la aplicación Angular utilizando la configuración de app.config.ts.
- **main.server.ts**: Arranque desde el servidor del módulo principal de la aplicación Angular utilizando la configuración de app.config.server.ts.
- app/app.config.ts: Configuración de la aplicación, permite definir o importar los proveedores de servicios (externalización de main.ts), añadiendo:
  - provideClientHydration()
- **app/app.config.server.ts**: Configuración de la aplicación desde el servidor, permite definir o importar los proveedores de servicios propios del servidor que mezcla con las definiciones de app.config.ts (externalización de main.server.ts).
  - provideServerRendering()

Filtrar las URL de solicitud en el servidor

- El servidor web debe distinguir las solicitudes de páginas de aplicaciones de otros tipos de solicitudes. No es tan simple como interceptar una solicitud a la dirección raíz / dado que el navegador podría solicitar una de las rutas de la aplicación, como /dashboard, /home o /detail:12, ficheros estáticos o peticiones REST. De hecho, si la aplicación solo fuera procesada por el servidor, cada enlace de aplicación en el que se haga clic llegaría al servidor como una petición URL destinada al enrutador.
- Afortunadamente, las rutas de aplicaciones tienen algo en común: sus URL carecen de extensiones de archivo. Si utilizamos enrutamiento, podemos reconocer fácilmente los tres tipos de solicitudes y manejarlas de manera diferente:
  - Solicitud de datos: peticiones REST, URL fáciles de reconocer porque comienzan con /api o similares.
  - Navegación de la aplicación: solicitudes cuyas URL, sin extensión de archivo, están mapeadas en el enrutador de Angular.
  - Activo estático: todas las demás solicitudes, normalmente con extensión.
- El comportamiento básico se maneja automáticamente cuando se utiliza el esquema ssr, salvo las solicitudes de datos que aparecen comentadas.

Compatibilidad con SSR

- Debido a que una aplicación universal no se ejecuta en el navegador, es posible que algunas de las API y capacidades del navegador falten en el servidor. Las aplicaciones no pueden hacer uso de objetos globales específicos del navegador como window, document, navigator o location así como ciertas propiedades de HTMLElement. Las aplicaciones universales utilizan el paquete Angular platform-server (a diferencia de platform-browser), que proporciona implementaciones de servidor de DOM, XMLHttpRequest y otras funciones de bajo nivel que no dependen de un navegador.
- Angular proporciona algunas abstracciones inyectables sobre estos objetos, como Location o DOCUMENT, que puede sustituir adecuadamente estas API. Si Angular no lo proporciona, es posible escribir nuevas abstracciones que deleguen a las API del navegador mientras está en el navegador y a una implementación alternativa mientras se está en el servidor (también conocido como shimming).
- El servicio Renderer2 proporciona una abstracción que permite indicar las manipulaciones DOM pero que se puede utilizar con seguridad incluso cuando el acceso directo a elementos nativos no es posible.
- De manera similar, sin eventos de mouse o teclado, una aplicación del lado del servidor no puede depender de que un usuario haga clic en un botón para mostrar un componente. La aplicación debe determinar qué representar basándose únicamente en la solicitud entrante del cliente. Este es un buen argumento para hacer que la aplicación enrutable.
- En general, el código que usa el API del navegador sólo debe ejecutarse en el navegador, no en el servidor. Esto se puede aplicar a través de los hooks del ciclo de vida afterRender y afterNextRender, que sólo se ejecutan en el navegador y se omiten en el servidor.

JAMStack

- JAMStack es un término genérico para referirse a una forma de desarrollar aplicaciones web: Pila JavaScript, API y Markup.
- Jamstack es fundamentalmente una filosofía, una metodología, es un enfoque, una forma de pensar, un conjunto de principios y mejores prácticas. Es una arquitectura, no se trata realmente de tecnologías específicas, no existe un instalador de Jamstack, ni es impulsado por una gran empresa, ni existe algún organismo de normalización que lo controle o defina.
- JAMStack es estática en el sentido de que el contenido es servido como archivos estáticos (Markup) pero los datos utilizados pueden ser dinámicos (JavaScript, API). El HTML se renderiza previamente en archivos estáticos, este código se distribuye a una CDN que es una red global de servidores, y las tareas que alguna vez se procesaron y administraron en el lado del servidor ahora se realizan a través de API y microservicios.
- Componentes de Jamstack
  - JavaScript: las funcionalidades dinámicas son manejadas por JavaScript. No se exige que se use algún framework, ni hay alguna restricción sobre diseñar tu aplicación de alguna manera en particular. JavaScript es lo que hace que un sitio de Jamstack sea dinámico, permite que los activos estáticos cambien dinámicamente a través de la manipulación DOM.
  - APIs: Jamstack no requiere una necesidad real de construir un backend o servidor y se basa principalmente en APIS de terceros, en cambio complementa la tecnología sin servidores y se puede usar con funciones sin servidor, como funciones de Netlify o similares. Los desarrolladores usan API para obtener los datos que necesitan para crear e implementar aplicaciones y sitios Jamstack estáticos. Las API se utilizan a menudo en el cliente cuando desea acceder a contenido dinámico desde el navegador.
  - Markup: abarca lenguajes de plantillas como Markdown por ejemplo y lenguajes de serialización de datos como YAML y JSON. Específicamente, el Markup o marcado en Jamstack se refiere al marcado o contenido “prerenderizado” servido en su forma final, HTML.

Hidratación

- La hidratación es el proceso que restaura la aplicación renderizada del lado del servidor en el cliente. Esto incluye cosas como reutilizar las estructuras DOM renderizadas por el servidor, conservar el estado de la aplicación, transferir los datos de la aplicación que ya fueron recuperados por el servidor y otros procesos.
- La hidratación mejora el rendimiento de la aplicación al evitar trabajo adicional para recrear nodos DOM. En cambio, Angular intenta hacer coincidir los elementos DOM existentes con la estructura de la aplicación en tiempo de ejecución y reutiliza los nodos DOM cuando es posible. Esto da como resultado una mejora del rendimiento que se puede medir utilizando estadísticas Core Web Vitals (CWV, Métricas web esenciales), como la reducción del primer procesamiento de imagen con contenido (FCP) y con contenido más grande (LCP), así como el cambio de diseño acumulativo (CLS). Mejorar estos números también afecta aspectos como el posicionamiento SEO.
- Sin la hidratación habilitada, las aplicaciones Angular renderizadas en el lado del servidor destruirán y volverán a renderizar el DOM en el cliente, lo que puede provocar un parpadeo visible en la interfaz de usuario y puede afectar negativamente a las métricas del contenido más grande (LCP) y el cambio de diseño acumulativo (CLS).

Hidratación

- Se puede habilitar manualmente la hidratación proveyendo el servicio provideClientHydration() importado desde @angular/platform-browser en el bootstrapApplication o en el módulo principal.
  providers: [provideServerRendering(),]
- Mientras se ejecuta la aplicación en modo de desarrollo, se puede confirmar que la hidratación está habilitada en la consola del navegador, que debería mostrar un mensaje que incluye estadísticas relacionadas con la hidratación, como la cantidad de componentes y nodos hidratados.
- La hidratación impone algunas limitaciones a la aplicación que no están presentes sin la hidratación habilitada:
  - No es compatible con las implementaciones personalizada ni la implementación "noop" de Zone.js
  - La aplicación debe tener la misma estructura DOM generada tanto en el servidor como en el cliente
  - Angular omitiría la hidratación de los componentes que usan bloques i18n
- La hidratación depende de una señal de Zone.js cuando se vuelve estable dentro de una aplicación, para que Angular puede iniciar el proceso de serialización en el servidor o la limpieza posterior a la hidratación en el cliente para eliminar los nodos DOM que no fueron reclamados. Proporcionar una implementación personalizada o la implementación "noop" de Zone.js puede dar lugar a una sincronización diferente del evento onStable, lo que desencadena la serialización o la limpieza demasiado pronto o demasiado tarde.

Limitaciones impuestas por la hidratación

- La aplicación debe tener la misma estructura DOM generada tanto en el servidor como en el cliente, el proceso de hidratación espera que el árbol DOM tenga exactamente la misma estructura en ambos lugares. Esto también incluye espacios en blanco y nodos de comentarios que Angular produce durante el renderizado en el servidor, que deben estar presentes en el HTML generado por el proceso de renderizado del lado del servidor.
- El HTML producido por la operación de renderizado del lado del servidor no debe alterarse entre el servidor y el cliente.
- Muchos CDN intentarán optimizar la aplicación eliminando los nodos del DOM que considere innecesarios, incluidos los nodos de comentarios. Los nodos de comentarios son una parte esencial del funcionamiento de Angular y son fundamentales para que funcione la hidratación. Se deberá desactivar esta función del CDN para garantizar que su aplicación se cargue e hidrate.
- Si hay una discrepancia entre las estructuras del árbol DOM del servidor y del cliente, el proceso de hidratación encontrará problemas al intentar hacer coincidir lo que se esperaba con lo que realmente está presente en el DOM. Los componentes que realizan manipulación directa de DOM utilizando API de DOM nativas son los culpables más comunes.

Limitaciones impuestas por la hidratación

- El proceso de hidratación generara errores si los componentes manipulan el DOM directamente, usando innerHTML o outerHTML, accediendo a document para consultar elementos específicos e inyectar nodos adicionales usando appendChild o separando nodos DOM y para moverlos a otras ubicaciones.
- Esto se debe a que Angular desconoce estos cambios de DOM y no puede resolverlos durante el proceso de hidratación: Angular espera una determinada estructura, pero encontrará una estructura diferente al intentar hidratarse. Esta discrepancia implica un fallo de hidratación y generará un error de discrepancia de DOM.
- Es mejor refactorizar los componentes para evitar este tipo de manipulación DOM, utilizando las API de Angular para realizar este trabajo, si es posible. Si no se puede refactorizar este comportamiento, el atributo ngSkipHydration omite la hidratación de componentes particulares.
- También se generará un error de discrepancia de DOM si las plantillas de los componente que no tienen una estructura HTML válida: <table> sin un <tbody>, dentro de un <p>, <a> dentro de un <h1> o de otro <a>, … Los validadores de sintaxis HTML ayudan a comprobarlo y asegúrarse de que el HTML de las plantillas utilice una estructura válida.

Omitir la hidratación

- Hay bibliotecas de terceros que dependen de la manipulación DOM para poder renderizar. Estas bibliotecas funcionaron sin hidratación, pero pueden provocar errores de discrepancia en DOM cuando la hidratación está habilitada. Hay componente propios de también dependen de la manipulación DOM y que no se han podido refactorizar o no se ha encontrado una solución compatible con la hidratación.
- En ambos casos, como solución alternativa, se puede agregar el atributo ngSkipHydration a la etiqueta de un componente para omitir la hidratación de todo el componente:

```
<dom-cmp ngSkipHydration />
```

- También se puede configurar la definición del componente:

```
@Component({
  :
  host: {ngSkipHydration: 'true'},
})
class DomCmp {
```

- ngSkipHydration obligará a Angular a omitir la hidratación en todo el componente y en **todos sus hijos**, lo que significa que se destruirá y se volverá a renderizar todo su contenido. Esto solucionará los problemas de renderizado, pero se perderán los beneficios de la hidratación, también en los hijos.

Comportamiento de HttpClient en la hidratación

- Cuando SSR está habilitado, HttpClient las respuestas se almacenan en caché mientras se ejecutan en el servidor. Después de eso, esta información se serializa y se transfiere a un navegador como parte del HTML inicial enviado desde el servidor. En un navegador, HttpClient comprueba si tiene datos en la memoria caché y, de ser así, los reutiliza en lugar de realizar una nueva solicitud HTTP durante la representación inicial de la aplicación. HttpClient deja de usar el caché una vez que una aplicación se vuelve estable mientras se ejecuta en un navegador.
- El almacenamiento en caché se realiza de forma predeterminada para las solicitudes GET y HEAD. Se puede configurar esta caché usando withHttpTransferCacheOptions al proveer la hidratación.

```
bootstrapApplication(AppComponent, {
  providers: [
    provideClientHydration(
      withHttpTransferCacheOptions({
        includePostRequests: true,
      }),
    ),
  ],
});
```


Renderización previa (SSG)

- La renderización previa, comúnmente conocida como generación de sitios estáticos (SSG), representa el método mediante el cual las páginas se generan en archivos HTML estáticos durante el proceso de construcción (build).
- La renderización previa amplia los beneficios de rendimiento que la renderización del lado del servidor (SSR): logra un tiempo reducido hasta el primer byte (TTFB), lo que en última instancia mejora la experiencia del usuario, requiere menos proceso del servidor (espacio x proceso). La distinción clave radica en su enfoque: las páginas se muestran como contenido estático ya generado y no hay representación basada en solicitudes.
- Cuando los datos necesarios para la renderización del lado del servidor son consistentes para todos los usuarios, la estrategia de renderización previa surge como una alternativa valiosa. En lugar de renderizar páginas dinámicamente para cada solicitud del usuario, el renderizado previo adopta un enfoque proactivo al renderizarlas por adelantado durante el proceso de construcción.

Renderización previa (SSG)

- Por defecto el builder procesa la configuración del enrutador de Angular para encontrar todas las rutas no parametrizadas y preprocesarlas: crea en dist/*app*/browser/ un directorio por ruta donde introduce un index.html con la página ya renderizada.
- Se pueden pre renderizar las rutas parametrizadas, pero deben enumerarse en un archivo de texto (una ruta absoluta, con los parámetros resueltos, en una línea separada) y referenciar dicho archivo en el angular.json usando la opción routesFile.

```
# param-routes.txt
/contactos/1
/contactos/1/edit

# angular.json
"architect": {
 "build": {
    :
    "options": {
      "prerender": { "routesFile": "param-routes.txt" },
      :
```