FORMACIÓN EN NUEVAS TECNOLOGÍAS

Angular 7 Avanzado

1

ICONO TRAINING

Formación en Nuevas Tecnologías

- www.iconotc.com
- linkedin.com/company/icono-training-consulting
- training@iconotc.com

FORMADOR

Javier Martín Antón

Consultor / formador en tecnologías del área de programación y desarrollo.

¡Síguenos en las Redes Sociales!

2

ANGULAR 7 AVANZADO

DURACIÓN

- 25 horas

LUGAR/FECHAS /HORARIO:

- Salamanca. Del 1 al 5 de Abril de 2019. De 15:00 hs a 20:00 hs

CONTENIDO:

- Introducción:
  - ECMAScript
  - TypeScript
  - NodeJS
  - NPM
  - NVM
  - Angular 7
  - Angular CII
- Angular 7 Avanzado
  - Componentes avanzados (Modales, login, spinner, etc)
  - Formularios avanzados (validaciones, etc.)
  - Routin y Navegación avanzados (Guards)
  - Servicios (HttpClient, Observables y RxJS)
  - Internacionalización (i18n)
  - Test unitarios
  - Publicación PROD

3

Angular Avanzado

4

Temario

- Arquitectura avanzada y mejores practicas
  - Ciclo de vida y detección de cambios
  - Optimización y rendimiento
- RxJS (Reactive Programming)
  - Programación Reactiva
  - Patrón Observable
  - Patrón Promise
  - Creación de Observables y observadores
- Acceso al servidor
  - Patrón Observable (RxJS).
  - HttpModule y JsonpModule
  - Servicios RESTFul
  - JSONP
- Enrutamiento y navegación
  - RouterModule
  - Definición de rutas
  - Paso de parámetros
  - Navegación
- Creación de directivas
  - Directivas atributo
  - Validaciones personalizadas
  - Directivas estructurales
- Formularios
  - Formularios basados en plantillas
  - Formularios basados en modelos
  - Validaciones
- UI Avanzado
  - Animaciones
  - Responsive Web Design
  - Angular UI Bootstrap
  - Material Design
  - Otros Módulos
- Conceptos avanzados
  - Internacionalización (I18N)
  - Publicación de módulos con NPM
  - Migración desde AngularJS a Angular
  - Módulo ngUpgrade: Aplicaciones hibridas AngularJS y Angular 2+
- Testing
  - Utilidades
  - Técnicas de pruebas y TDD
  - Test unitarios
  - Test E2E
    - Protractor
    - Selenium
- Despliegue

5

RXJS (REACTIVE PROGRAMMING)

6

Reactive Programming

- La Programación Reactiva (o Reactive Programming) esta orientada a programar con flujos de datos asíncronos. Estos flujos se pueden observar y reaccionar en consecuencia.
- RxJS es una biblioteca 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.
- ReactiveX combina el patrón Observer con el patrón Iterator y la programación funcional con las colecciones para gestionar secuencias de eventos de una forma ideal .
- 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 permite coordinar cuando cómputo ocurre en, por ejemplo, setTimeout, requestAnimationFrame u otros.

7

Observables

- Los Observables proporciona soporte para pasar mensajes entre publicador o editor y 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.

8

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

9

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 completado, entonces no se podrá entregar nada más después.
- Cuando se suscribe, obtiene una Suscripción, que representa la ejecución en curso. Basta con llamar a unsubscribe() para cancelar la ejecución.

10

Publicador - Suscriptor

- Para crear un publicador, se crea una instancia de la clase Observable en cuyo constructor se le suministra la función de publicadora que define cómo obtener o generar 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( { notify => console.log("Observer got a next value: " + notify),
    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();
```


11

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


12

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.

13

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(x => console.log(x));
// …
subscription.unsubscribe();
```

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

```
var childSubscription = observable2.subscribe(x => console.log('second: ' + x));
subscription.add(childSubscription);
subscription.unsubscribe();
```


14

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.

15

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 subject = new Subject();
subject.subscribe({ next: (v) => console.log('observerA: ' + v) });
subject.subscribe({ next: (v) => console.log('observerB: ' + v) });
let observable = from([1, 2, 3]);
observable.subscribe(subject); //de unidifusión a multidifusión
```


16

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 el 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);
```


17

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);
```


18

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);
```


19

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().

20

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
```


21

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();
```


22

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 para vincular varios operadores juntos. Las tuberías 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.

23

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

24

Operadores

```
const squareOdd = of(1, 2, 3, 4, 5)
  .pipe(
    filter(n => n % 2 !== 0),
    map(n => n * n)
  );
squareOdd.subscribe(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(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);
  });
```


25

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.

26

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.

27

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

28

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);
```


29

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.

30

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 aparecen nombrados con un signo "$" como sufijo.
- Esto puede ser útil al escanear a través del código en búsqueda de valores observables.
- Además, si desea que una propiedad almacene el valor más reciente de un observable, puede ser conveniente simplemente usar el mismo nombre con o sin el "$".

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.
- El servicio $q es un servicio de AngularJS que contiene toda la funcionalidad de las promesas (está basado en la implementación Q de Kris Kowal).
- La librería JQuery incluye el objeto $.Deferred desde la versión 1.5.
- Las promesas se han incorporado a los objetos estándar de JavaScript en la versión 6.

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.

33

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();
 }
```


34

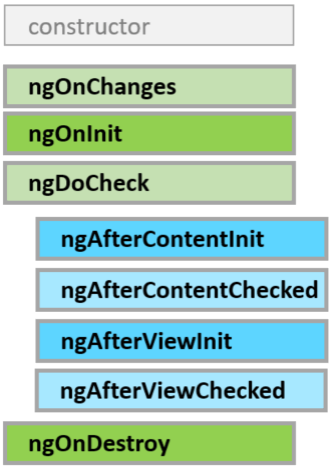
Invocar promesas

- 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.

ARQUITECTURA AVANZADA Y MEJORES PRACTICAS

Ciclo de vida



- Cada componente tiene un ciclo de vida gestionado por el Angular.
- Angular lo crea y pinta, 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.

37

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 Angular inicialice las vistas del componente y sus hijos.
ngAfterViewChecked	Responder después de que Angular chequee las vistas del componente y sus hijos.
ngOnDestroy	Limpiar justo antes de que Angular destruya el componente.

38

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.

39

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) { … }
```


40

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.

41

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.

42

AfterContent

- ngAfterContentInit() y ngAfterContentChecked() se invocan después de que Angular proyecta contenido externo en el componente, 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.

43

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.

44

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.

45

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 si no se hace.

46

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.

47

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 que se supone que 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.

48

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.

49

zone.js

- Zone.js 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.

50

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

- 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
```


55

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.

56

ChangeDetectionStrategy.OnPush

- En esta estrategia, 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.

57

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();
});
```


58

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.

61

ENRUTADO

62

Introducción

- El enrutado permite tener una aplicación de una sola página, pero que es capaz de representar URL distintas, simulando lo que sería una navegación a través páginas web, pero sin salirnos nunca de la página inicial. Esto permite:
  - **Memorizar rutas profundas dentro de nuestra aplicación**. Podemos contar con enlaces que nos lleven a partes internas (deeplinks), de modo que no estemos obligados a entrar en la aplicación a través de la pantalla inicial.
  - Eso **facilita también el uso natural del sistema de favoritos** (o marcadores) del navegador, así como el historial. Es decir, gracias a las rutas internas, seremos capaces de guardar en favoritos un estado determinado de la aplicación. A través del uso del historial del navegador, para ir hacia delante y atrás en las páginas, podremos navegar entre pantallas de la aplicación con los botones del navegador.
  - **Mantener y cargar módulos en archivos independientes**, lo que reduce la carga inicial y permite la carga perezosa.

63

Rutas internas

- En las URL, la “almohadilla“, el carácter “#“, sirve para hacer rutas a anclas internas: zonas de una página.
- Cuando se pide al navegador que acceda a una ruta creada con “#“ éste no va a recargar la página (cargando un nuevo documento que pierde el contexto actual), lo que hará es buscar el ancla que corresponda y mover el scroll de la página a ese lugar.
  - http://example.com/index.html
  - http://example.com/index. html#/seccion
  - http://example.com/index. html#/pagina_interna
- Es importante fijarse en el patrón "#/", sirve para hacer lo que se llaman "enlaces internos" dentro del mismo documento HTML.
- En el caso de Angular no habrá ningún movimiento de scroll, pues con Javascript se detectará el cambio de ruta en la barra de direcciones para intercambiar la vista que se está mostrando.
- Los navegadores HTML5 modernos admiten history.pushState, una técnica que cambia la ubicación y el historial de un navegador sin activar una solicitud de página del servidor. El enrutador puede componer una URL "natural" que es indistinguible de una que requeriría una carga de página.

64

Hash Bag

- Dado que los robots indexadores de contenido de los buscadores no siguen los enlaces al interior de la pagina (que asumen como ya escaneada), el uso del enrutado con # que carga dinámicamente el contenido impide el referenciado en los buscadores.
- Para indicarle al robot que debe solicitar el enlace interno se añade una ! después de la # quedando la URL:

```
http://www.example.com/index.html#!ruta
```


65

Angular Router

- El Angular Router ("router") toma prestado el modelo deeplinks. Puede interpretar una URL del navegador como una instrucción para navegar a una vista generada por el cliente y pasar parámetros opcionales en la ruta al componente para decidir qué contenido específico se quiere manejar.
- El Angular router es un servicio opcional que presenta la vista de un componente en particular asociado a una determinada URL.
- No es parte del núcleo Angular. Es un paquete de la biblioteca, @angular/router, a importar en el módulo principal como se haría con cualquier otro modulo Angular.

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

- La aplicación tendrá un único router. Cuando la URL del navegador cambia, el router busca una correspondencia en la tabla de rutas para determinar el componente que debe mostrar.
- Las aplicaciones de enrutamiento deben agregar un elemento <base> index.html al principio de la etiqueta <head> para indicar al enrutador cómo componer las URL de navegación.

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


66

Tabla de rutas

- La tabla de ruta es un conjunto de objetos Route.
- Toda ruta tiene una propiedad path con la ruta que se utiliza como patrón de coincidencia. Puede ser:
  - Única: *path: 'mi/ruta/particular'*
  - Parametrizada: *path: 'mi/ruta/:id'*
  - Vacía (solo una vez): *path: ''*
  - Todas las demás (solo una vez): *path: '**'*
- Dado que la búsqueda se realiza secuencialmente la tabla debe estar ordenada de rutas mas especificas a las mas generales.
- La ruta puede estar asociada a:
  - Un componente: *component: MyComponent*
  - Otra ruta (redirección): *redirectTo: '/otra/ruta'*
  - Otro módulo (carga perezosa): **loadChildren: 'ruta/otro/modulo'**
- Adicionalmente se puede indicar:
  - outlet: Destino de la ruta.
  - pathMatch: **prefix** | full
  - children: Subrutas
  - data: datos adicionales
  - Servicios canActivate, canActivateChild, canDeactivate, canLoad

67

Registrar enrutamiento

```
const routes: Routes = [
  { path: '', component: HomeComponent, pathMatch: 'full' },
  { path: 'path/:routeParam', component: MyComponent1 },
  { path: 'staticPath', component: MyComponent2 },
  { path: 'oldPath', redirectTo: '/newPath' },
  { path: 'path', component: MyComponent3 , data: { message: 'Custom' } },
  { path: '**', component: ErrorComponent },
];

@NgModule({
  imports: [ BrowserModule,
             RouterModule.forRoot(routes)
         ],
  // ...
})
export class AppModule { }
```


68

Directivas

- Punto de entrada por defecto:
  <**router-outlet**></router-outlet>
- Punto de entrada con nombre, propiedad outlet de la ruta:
  <router-outlet **name="aux"**></router-outlet>
- Generador de referencias href:
  <a **routerLink**="/path">
  <button **routerLink**="/">Go to home</button>
  <a [routerLink]="['/path', routeParam]">
  <a [routerLink]="['/path', { matrixParam: 'value' }]">
  <a [routerLink]="['/path']" [queryParams]="{ page: 1 }">
  <a [routerLink]="['/path']" fragment="anchor">
- Nombre de la Class CSS asociada a ruta actual:
  <a [routerLink]="['/path']" **routerLinkActive**="active"
  [routerLinkActiveOptions]="{exact: true}">

69

Trabajar con rutas

- Importar clases:
  import { Router, ActivatedRoute } from '@angular/router';
- Inyectar dependencias:
  constructor(private route: ActivatedRoute, private router: Router) { }
- Decodificar parámetros y ruta:
  - Valor actual (Instantánea):
    let id = this.route.snapshot.params['id'];
  - Detección de cambios (observable)
    route.url.map(segments => segments.join(''));
    let sid = this.route.queryParamMap.pipe(map(p => p.get('sid') || 'None'));
    this.token = this.route.fragment.pipe(map(f => f || 'None'));
- Navegación desde el código:
  this.router.navigate(['/ruta/nueva/1']);

70

Decodifica ruta

```
ngOnInit() {
  let id = this.route.snapshot.params['id'];
  if (id) {
    if (this.route.snapshot.url.slice(-1)[0].path === 'edit') {
      this.vm.edit(+id);
    } else {
      this.vm.view(+id);
    }
  } else if (this.route.snapshot.url.slice(-1)[0].path === 'add') {
    this.vm.add();
  } else {
    this.vm.load();
  }
 }
```


Parámetros observables

```
private obs$: any;
ngOnInit() {
  this.obs$ = this.route.paramMap.subscribe(
    (params: ParamMap) => {
    const id = +params.get('id'); // (+) converts string 'id' to a number
    if (id) {
      this.vm.edit(id);
    } else {
      this.router.navigate(['/404.html']);
    }
   });
 }
 ngOnDestroy() { this.obs$.unsubscribe(); }
```

Eventos de ruta

Evento	Desencadenado
NavigationStart	cuando comienza la navegación.
RoutesRecognized	cuando el enrutador analiza la URL y las rutas son reconocidas.
RouteConfigLoadStart	antes de que empiece la carga perezosa.
RouteConfigLoadEnd	después de que una ruta se haya cargado de forma perezosa.
NavigationEnd	cuando la navegación termina exitosamente.
NavigationCancel	cuando se cancela la navegación: un guardián devuelve falso.
NavigationError	cuando la navegación falla debido a un error inesperado.

- Durante cada navegación, el Router emite eventos de navegación a través de la propiedad observable Router.events a la se le puede agregar suscriptores para tomar decisiones basadas en la secuencia de eventos.

```
this.router.events.subscribe(ev => {
    if(ev instanceof NavigationEnd) { this.inicio = (ev as NavigationEnd).url == '/'; }
  });
```


73

Guardianes de rutas

- Los guardianes controlan el acceso a las rutas por parte del usuario o si se le permite abandonarla:
  - CanActivate: verifica el acceso a la ruta.
  - CanActivateChild: verifica el acceso a las ruta hijas.
  - CanDeactivate: verifica el abandono de la ruta, permitiendo descartar acciones pendientes que se perderán.
  - CanLoad: verifica el acceso a un módulo que requiere carga perezosa.
- Para crear un guardián hay que crear un servicio que implemente el correspondiente interfaz y registrarlo en cada ruta a controlar.

```
@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private authService: AuthService, private router: Router) {}
  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
          boolean { return this.authService.isAutenticated;  }
}
{ path: 'admin', component: AdminComponent, canActivate: [AuthGuard], },
```


74

Obtener datos antes de navegar

- En algunos casos es interesante resolver el acceso a datos antes de navegar para poder redirigir en caso de no estar disponibles.
- Servicio:

```
@Injectable({ providedIn: 'root', } )
export class DatosResolve implements Resolve<any> {
  constructor(private dao: DatosDAOService, private router: Router) {}
  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<any> {
    return this.dao.get(+route.paramMap.get('id')).pipe(
      take(1),
      map(data => {
        if (data) { return data; } else { // id not found
          this.router.navigate(['/404.html']);
          return null;
        }
      }),
      catchError(err => { this.router.navigate(['/404.html']); return empty(); })
    );
  }
}
```


Obtener datos antes de navegar

- En la tabla de rutas:

```
{ path: 'datos/:id', component: DatosViewComponent,
    resolve: {elemento: DatosResolve} },
```

- En el componente:

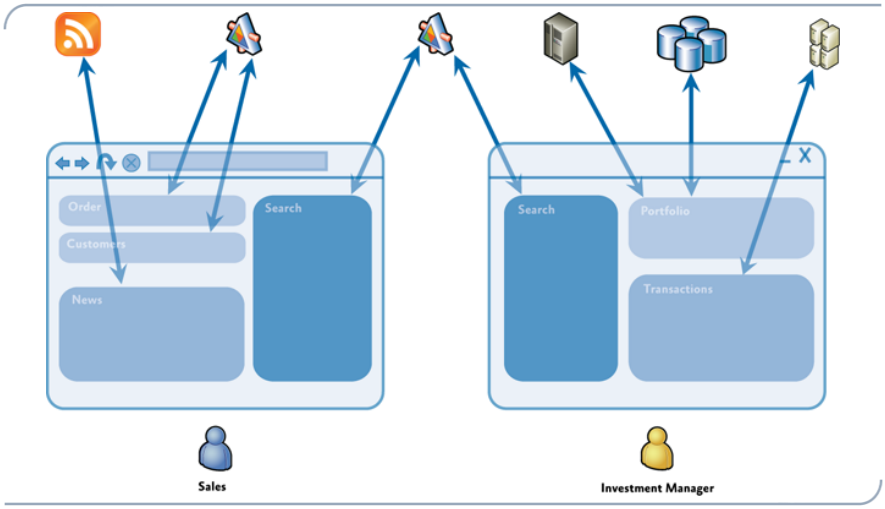
```
ngOnInit() {
  this.route.data.subscribe((data: { elemento: any }) => {
    let e = data.elemento;
    // …
  });
```

Mantenimiento de estado

- Uno de los posibles aspectos problemáticos del enrutado es que los componentes se instancian con cada vista donde se estén usando, cada vez que se carga la ruta.
- Por este motivo todos los datos que se inicializan y se vuelven a poner a sus valores predeterminados cuando carga cualquier la vista.
- El mantenimiento de estado consiste en guardar los datos desde que el componente se destruye hasta que se vuelve a crear y se restaura el estado, situación antes de destruirse.
- A través de los servicios se puede implementar el mantenimiento de estado de los componentes en caso de ser necesario.
  - El patrón singleton, utilizado en la inyección de dependencias, asegura que sólo existe una instancia de ellos en el modulo, por lo que no pierden su estado, y, si hay varios componentes que dependen de un mismo servicio, todos recibirán la misma instancia del objeto.

77

Patrón Composite View



78

outlet

- Punto de entrada con nombre, propiedad outlet de la ruta:
  <router-outlet name="aside"></router-outlet>
- En la tabla de rutas:
  { path: 'mypath', component: MyComponent, outlet: 'aside' }
- Generador de referencias secundarias:
  <a [routerLink]="[{ outlets: { aside: ['mypath'] } }]">...
- Ruta múltiple:
  http://example.com/mainpath(aside:mypath)
- Navegación desde el código:
  this.router.navigate([{ outlets: { aside: ['mypath'] } }]);
- Eliminar el contenido:
  this.router.navigate([{ outlets: { aside: null } }]);

79

Lazy loading

- La carga diferida de módulos (lazy loading) se controla con el router:

```
export const routes: Routes = [
  // …
  { path: 'admin',
    loadChildren: './admin/admin.module#AdminModule' },
];
```

- El localizador loadChildren de carga perezosa es una cadena, no un tipo.
- La cadena identifica tanto el módulo de archivo (módulo TypeScript) como el módulo de clase (módulo Angular), este último separado del anterior por una #.
  ./admin/admin.module#AdminModule
  equivale a una importación en app.module
  import { AdminModule } from './admin/admin.module';

80

Lazy loading

- El módulo LazyLoad no debe estar entre los imports del @NgModule del AppModule.
- WebPack crea un bundle por cada módulo enrutado como loadChildren identificado por un número y la sub extensión .chunk
  - 0.chunk.js, 1.chunk.js, 2.chunk.js, …
- El módulo LazyLoad debe tener una tabla de rutas con la ruta vacía que indica el componente inicial del módulo.

```
export const routes: Routes = [
   { path: '', component: AdminMainComponent },
   { path: 'users', component: UsersComponent },
   { path: 'roles', component: RolesComponent },
];
```

- La tabla de rutas, en el módulo LazyLoad se debe importar a través del forChild:

```
@NgModule({  imports: [ RouterModule.forChild(routes), // …
```


81

ACCESO AL SERVIDOR

82

Servicio HttpClient (v 4.3)

- El servicio HttpClient permite hacer peticiones AJAX al servidor.
- Encapsula el objeto XMLHttpRequest (Level 2), pero está integrado con Angular como un servicio (con todas las ventajas de ellos conlleva) y notifica a Angular que ha habido un cambio en el modelo de JavaScript y actualiza la vista y el resto de dependencias adecuadamente.
- El HttpClient es un servicio opcional y no es parte del núcleo Angular. Es parte de la biblioteca @angular/common/http, con su propio módulo.
- Hay que importar lo que se necesita de él en el módulo principal como se haría con cualquier otro modulo Angular. No es necesario registrar el proveedor.

```
import { HttpClientModule, HttpClientJsonpModule } from
    '@angular/common/http';
@NgModule({
  imports: [ HttpModule, HttpClientJsonpModule, // …
  ],  // …
})
export class AppModule { }
```


HttpClient (v 4.3)

- Evolución del Http anterior:
  - Atajos a los verbos HTTP con parámetros mínimos.
  - Respuestas tipadas.
  - Acceso al cuerpo de respuesta síncrono y automatizado, incluido el soporte para cuerpos de tipo JSON.
  - JSON como valor predeterminado, ya no necesita ser analizado explícitamente.
  - Los interceptores permiten que la lógica de middleware sea insertada en el flujo.
  - Objetos de petición/respuesta inmutables
  - Eventos de progreso para la carga y descarga de la solicitud.
  - Protección ante XSRF

Servicio HttpClient

- Método general:

  request(first: string|HttpRequest<any>, url?: string, options: {...}): Observable<any>
- Atajos:

  get(url: string, options: {...}): Observable<any>
  post(url: string, body: any, options: {...}): Observable<any>
  put(url: string, body: any, options: {...}): Observable<any>
  delete(url: string, options: {...}): Observable<any>
  patch(url: string, body: any, options: {...}): Observable<any>
  head(url: string, options: {...}): Observable<any>
  options(url: string, options: {...}): Observable<any>
  jsonp<T>(url: string, callbackParam: string): Observable<T>

85

Opciones adicionales

- **body**: Para peticiones POST o PUT
- **headers**: Colección de cabeceras que acompañan la solicitud
- **observe**: **'body'** | 'events' | 'response'
- **params**: Coleccion de pares nombre/valor del QueryString
- **reportProgress**: true activa el seguimiento de eventos de progreso
- **responseType**: 'arraybuffer'|'blob'|**'json'**|'text',
- **withCredentials**: true indica el envío de credenciales

86

Peticiones al servidor

- Solicitar datos al servidor:

```
this.http.get(`ws/entidad/${Id} `)
    .subscribe(
      datos => this.listado = datos,
      error => console.error(`Error: ${error}`)
    );
```

- Enviar datos al servidor (ws/entidad/edit?id=3):

```
this.http.post('ws/entidad', body, {
  headers: new HttpHeaders().set('Authorization', 'my-auth-token'),
  params: new HttpParams().set('id', '3'),
})
.subscribe(
  data => console.log('Success uploading', data),
  error => console.error('Error: ${error}')
);
```


87

JSON como valor predeterminado

- El tipo más común de solicitud para un backend es en formato JSON.

```
return this.http.get(this.baseUrl); // .map(response => response.json());
```

- Se puede establecer el tipo de datos esperado en la respuesta:

```
this.http.get<MyModel>(this.baseUrl).subscribe(data => {
// data is an instance of type MyModel
});
```

- Para obtener la respuesta completa:

```
this.http.get<MyModel>(this.baseUrl, {observe: 'response'})
.subscribe(resp => {
// resp is an instance of type HttpResponse
// resp.body is an instance of type MyModel
});
```

- Para solicitar datos que no sean JSON:

```
return this.http.get(this.baseUrl, {responseType: 'text'});
```


88

HttpResponse<T>

- **type**: HttpEventType.Response o HttpEventType.ResponseHeader.
- **ok**: Verdadero si el estado de la respuesta está entre 200 y 299.
- **url**: URL de la respuesta.
- **status**: Código de estado devuelto por el servidor.
- **statusText**: Versión textual del `status`.
- **headers**: Cabeceras de la respuesta.
- **body**: Cuerpo de la respuesta en el formato establecido.

89

Manejo de errores

- Hay que agregar un controlador de errores a la llamada de .subscribe():

```
this.http.get<MyModel>(this.baseUrl).subscribe(
  resp => { ... },
  err => { console.log('Error !!!');  }
 });
```

- Para obtener los detalles del error:

```
(err: HttpErrorResponse) => {
    if (err.error instanceof Error) {
      // A client-side or network error occurred. Handle it accordingly.
      console.log('An error occurred:', err.error.message);
    } else {
      // The backend returned an unsuccessful response code.
      // The response body may contain clues as to what went wrong,
      console.log(`Backend returned code ${err.status}, body was: ${err.error}`);
    }
  }
```


90

HttpErrorResponse

- **type**: HttpEventType.Response o HttpEventType.ResponseHeader.
- **url**: URL de la respuesta.
- **status**: Código de estado devuelto por el servidor.
- **statusText**: Versión textual del `status`.
- **headers**: Cabeceras de la respuesta.
- **ok**: Falso
- **name**: 'HttpErrorResponse'
- **message**: Indica si el error se ha producido en la solicitud (status: 4xx, 5xx) o al procesar la respuesta (status: 2xx)
- **error**: Cuerpo de la respuesta.

Tratamiento de errores

- Una forma de tratar los errores es simplemente reintentar la solicitud.
- Esta estrategia puede ser útil cuando los errores son transitorios y es poco probable que se repitan.
- RxJS tiene el operador .retry(), que automáticamente resubscribe a un Observable, reeditando así la solicitud, al encontrarse con un error.

```
import {retry, catchError} from 'rxjs/internal/operators';
// …
this.http.get<MyModel>(this.baseUrl).pipe(
   retry(3),
   catchError(err => …)
   ).subscribe(…);
```

Retroceso exponencial

- El retroceso exponencial es una técnica en la que se vuelve a intentar una API después de la falla, lo que hace que el tiempo entre reintentos sea más prolongado después de cada fallo consecutiva, con un número máximo de reintentos después de los cuales se considera que la solicitud ha fallado.

```
function backoff(maxTries, ms) {
 return pipe(
  retryWhen(attempts => range(1, maxTries)
    .pipe(zip(attempts, (i) => i), map(i => i * i), mergeMap(i =>  timer(i * ms))
    ) ) );
}

ajax('/api/endpoint')
  .pipe(backoff(3, 250))
  .subscribe(data => handleData(data));
```


93

Servicio RESTFul

```
import { HttpClient } from '@angular/common/http';
import {Observable} from 'rxjs';
import { environment } from 'src/environments/environment';

export abstract class RESTDAOService<T, K> {
  protected baseUrl = environment.apiURL;
  constructor(protected http: HttpClient, entidad: string, protected option = {}) {
          this.baseUrl += entidad; }
  query(): Observable<T> { return this.http.get<T>(this.baseUrl, this.option); }
  get(id: K): Observable<T> { return this.http.get<T>(this.baseUrl + '/' + id, this.option);
          }
  add(item: T): Observable<T>  { return this.http.post<T>(this.baseUrl, item,
          this.option); }
  change(id: K, item: T): Observable<T> {return this.http.put<T>(this.baseUrl + '/' + id,
          item, this.option);}
  remove(id: K): Observable<T> { return this.http.delete<T>(this.baseUrl + '/' + id,
          this.option); }
}
```


94

Seguridad

- La ejecución de aplicaciones JavaScript puede suponer un riesgo para el usuario que permite su ejecución.
- Por este motivo, los navegadores restringen la ejecución de todo código JavaScript a un entorno de ejecución limitado.
- Las aplicaciones JavaScript no pueden establecer conexiones de red con dominios distintos al dominio en el que se aloja la aplicación JavaScript.
- Los navegadores emplean un método estricto para diferenciar entre dos dominios ya que no permiten ni subdominios ni otros protocolos ni otros puertos.
- Si el código JavaScript se descarga desde la siguiente URL: http://www.ejemplo.com
- Las funciones y métodos incluidos en ese código no pueden acceder a:
  - **https**://www.ejemplo.com/scripts/codigo2.js
  - http://www.ejemplo.com**:8080**/scripts/codigo2.js
  - http://**scripts**.ejemplo.com/codigo2.js
  - http://**192.168.0.1**/scripts/codigo2.js
- La propiedad document.domain se emplea para permitir el acceso entre subdominios del dominio principal de la aplicación.

CORS

- Un recurso hace una solicitud HTTP de origen cruzado cuando solicita otro recurso de un dominio distinto al que pertenece y, por razones de seguridad, los exploradores restringen las solicitudes HTTP de origen cruzado iniciadas dentro de un script.
- XMLHttpRequest sigue la política de mismo-origen, por lo que solo puede hacer solicitudes HTTP a su propio dominio. Para mejorar las aplicaciones web, los desarrolladores pidieron a los proveedores de navegadores que permitieran a XMLHttpRequest realizar solicitudes de dominio cruzado.
- El Grupo de Trabajo de Aplicaciones Web del W3C recomienda el nuevo mecanismo de Intercambio de Recursos de Origen Cruzado (CORS, Cross-origin resource sharing). Los servidores deben indicar al navegador mediante cabeceras si aceptan peticiones cruzadas y con que características:
  - "Access-Control-Allow-Origin", "*"
  - "Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept"
  - "Access-Control-Allow-Methods", "GET, POST, PUT, DELETE“
- Soporte: Chrome 3+ Firefox 3.5+ Opera 12+ Safari 4+ Internet Explorer 8+

  Para habilitar CORS en servidores específicos consultar http://enable-cors.org

Proxy a un servidor de back-end

- Se puede usar el soporte de proxy en el servidor de desarrollo del webpack para desviar ciertas URL a un servidor backend.
- Crea un archivo proxy.conf.json junto a angular.json.

```
{
  "/api": {
    "target": "http://localhost:4321",
    "pathRewrite": { "^/api": "/ws" },
    "secure": false,
    "logLevel": "debug"
  }
}
```

- Cambiar la configuración del Angular CLI en angular.json:

```
"architect": {
  "serve": {
    "builder": "@angular-devkit/build-angular:dev-server",
    "options": {
      ...
      "proxyConfig": "proxy.conf.json"
    },
```


101

Desactivar la seguridad de Chrome

- Pasos para Windows:
  - Localizar el acceso directo al navegador (icono) y crear una copia como “Chrome Desarrollo”.
  - Botón derecho -> Propiedades -> Destino
  - Editar el destino añadiendo el parámetro al final. ej: "C:\Program Files (x86)\Google\Chrome\Application\chrome.exe" --disable-web-security
  - Aceptar el cambio y lanzar Chrome
- Para desactivar parcialmente la seguridad:

  --allow-file-access

  --allow-file-access-from-files

  --allow-cross-origin-auth-prompt
- Referencia a otros parametros:
  - http://peter.sh/experiments/chromium-command-line-switches/

102

JSONP (JSON con Padding)

- JSONP es una técnica de comunicación utilizada en los programas JavaScript para realizar llamadas asíncronas a dominios diferentes. JSONP es un método concebido para suplir la limitación de AJAX entre dominios por razones de seguridad. Esta restricción no se aplica a la etiqueta <script> de HTML, para la cual se puede especificar en su atributo src la URL de un script alojado en un servidor remoto.
- En esta técnica se devuelve un objeto JSON envuelto en la llamada de una función (debe ser código JavaScript valido), la función ya debe estar definida en el entorno de JavaScript y se encarga de manipular los datos JSON.

```
miJsonCallback ({"Nombre":"Carmelo","Apellidos":"Cotón"});
```

- Por convención, el nombre de la función de retorno se suele especificar mediante un parámetro de la consulta, normalmente, utilizando jsonp o callback como nombre del campo en la solicitud al servidor.

```
<script type="text/javascript"
   src="http://otrodominio.com/datos.json?callback=
   miJsonCallback"></script>
```


JSONP (JSON con Padding)

- Si no se importa el módulo HttpClientJsonpModule, las solicitudes de Jsonp llegarán al backend con el método JSONP, donde serán rechazadas.

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

@Injectable()
export class WikipediaService {
 constructor(private http: HttpClient) {}
 search (term: string) {
   let wikiUrl =
        `http://en.wikipedia.org/w/api.php?search=${term}&action=opensearch&for
        mat=json`;
   return this.http.jsonp(wikiUrl, `callback`).subscribe(
       datos => this.items  = datos, error => console.error(`Error: ${error}`)
    );
 }
}
```

Eventos

- Los eventos trabajan en un nivel más bajo que las solicitudes. Una sola solicitud puede generar múltiples eventos.
- Tipos de eventos:
  - **Sent**: La solicitud fue enviada.
  - **ResponseHeader**: Se recibieron el código de estado de respuesta y los encabezados.
  - **UploadProgress**: Se ha recibido un evento de progreso de subida.
  - **DownloadProgress**: Se ha recibido un evento de progreso de descarga.
  - **Response**: Se ha recibido la respuesta completa incluyendo el cuerpo.
  - **User**: Un evento personalizado de un interceptor o un backend.

Eventos de progreso

- A veces las aplicaciones necesitan transferir grandes cantidades de datos, como por ejemplo subir ficheros, y esas transferencias pueden tomar mucho tiempo.
- Es una buena práctica para la experiencia de usuario proporcionar información sobre el progreso de tales transferencias.

```
this.http.post('/upload/file', file, { reportProgress: true,  })
  .subscribe(event => {
    if (event.type === HttpEventType.UploadProgress) {
      const percentDone = Math.round(100 * event.loaded / event.total);
      console.log(`File is ${percentDone}% uploaded.`);
    } else if (event instanceof HttpResponse) {
      console.log('File is completely uploaded!');
    }
  });
```

Interceptores

- Una característica importante de HttpClient es la interceptación: la capacidad de declarar interceptores que se sitúan entre la aplicación y el backend.
- Cuando la aplicación hace una petición, los interceptores la transforman antes de enviarla al servidor.
- Los interceptores pueden transformar la respuesta en su camino de regreso antes de que la aplicación la vea.
- Esto es útil para múltiples escenarios, desde la autenticación hasta el registro.
- Cuando hay varios interceptores en una aplicación, Angular los aplica en el orden en que se registraron.

Crear un interceptor

- Los interceptores son servicios que implementan el interfaz HttpInterceptor, que requiere el método intercept:

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

- next siempre representa el siguiente interceptor en la cadena, si es que existe, o el backend final si no hay más interceptores.
- La solicitud es inmutable para asegurar que los interceptores vean la misma petición para cada reintento. Para modificarla es necesario crear una nueva con el método clone():

```
return next.handle(req.clone({url: req.url.replace('http://', 'https://')}));
```

- Se registran como un servicio múltiple sobre HTTP_INTERCEPTORS en el orden deseado:

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

Modificar la petición

```
import { HttpEvent, HttpInterceptor, HttpHandler, HttpRequest } from
        '@angular/common/http';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {
 constructor(private auth: AuthService) { }

 intercept(req: HttpRequest<any>, next: HttpHandler):
        Observable<HttpEvent<any>> {
  if (!req.withCredentials || ! this.auth.isAutenticated)
       { return next.handle(req); }
  const authReq = req.clone({ headers: req.headers.set('Authorization',
        this.auth.AuthorizationHeader) });
  return next.handle(authReq);
 }
}
```


109

Modificar la respuesta

```
import { HttpEvent, HttpInterceptor, HttpHandler, HttpRequest, HttpResponse } from
          '@angular/common/http';
import { tap, finalize } from 'rxjs/operators';
@Injectable()
export class LoggingInterceptor implements HttpInterceptor {
 intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const started = Date.now();
    let ok: string;
    return next.handle(req)
      .pipe(
        tap(
          event => ok = event instanceof HttpResponse ? 'succeeded' : '',
          error => ok = 'failed'
        ),
        finalize(() => {
          console.log(`${req.method} "${req.urlWithParams}" ${ok} in ${Date.now() - started} ms.`);
        })
      );
  }
}
```


110

Protección ante XSRF

- Cross-Site Request Forgery (XSRF) explota la confianza del servidor en la cookie de un usuario. HttpClient soporta el mecanismo “Cookie-to-Header Token” para prevenir ataques XSRF.
  - El servidor debe establecer un token en una cookie de sesión legible en JavaScript, llamada XSRF-TOKEN, en la carga de la página o en la primera solicitud GET. En las solicitudes posteriores, el cliente debe incluir el encabezado HTTP X-XSRF-TOKEN con el valor recibido en la cookie.
  - El servidor puede verificar que el valor en la cookie coincida con el del encabezado HTTP y, por lo tanto, asegúrese de que sólo el código que se ejecutó en su dominio pudo haber enviado la solicitud.
  - El token debe ser único para cada usuario y debe ser verificable por el servidor. Para mayor seguridad se puede incluir el token en un resumen de la cookie de autenticación de su sitio.
- Para establecer nombres de cookies / encabezados personalizados:

```
imports: [ // …
  HttpClientXsrfModule.withConfig({
        cookieName: 'My-Xsrf-Cookie',  headerName: 'My-Xsrf-Header',
  }),]
```

- El servicio backend debe configurarse para establecer la cookie y verificar que el encabezado está presente en todas las solicitudes elegibles.
- Angular solo lo aplica para peticiones a rutas relativas que no sean GET o HEAD.

111

JWT: JSON Web Tokens

https://jwt.io

- JSON Web Token (JWT) es un estándar abierto (RFC-7519) basado en JSON para crear un token que sirva para enviar datos entre aplicaciones o servicios y garantizar que sean válidos y seguros.
- El caso más común de uso de los JWT es para manejar la autenticación en aplicaciones móviles o web. Para esto cuando el usuario se quiere autenticar manda sus datos de inicio del sesión al servidor, este genera el JWT y se lo manda a la aplicación cliente, posteriormente en cada petición el cliente envía este token que el servidor usa para verificar que el usuario este correctamente autenticado y saber quien es.
- Se puede usar con plataformas IDaaS (Identity-as-a-Service) como Auth0 que eliminan la complejidad de la autenticación y su gestión.
- También es posible usarlo para transferir cualquier datos entre servicios de nuestra aplicación y asegurarnos de que sean siempre válido. Por ejemplo si tenemos un servicio de envío de email otro servicio podría enviar una petición con un JWT junto al contenido del mail o cualquier otro dato necesario y que estemos seguros que esos datos no fueron alterados de ninguna forma.

112

AuthService

```
@Injectable({providedIn: 'root'})
export class AuthService {
  private isAuth = false;
  private authToken: string = '';
  private name = '';
  constructor() {
    if (localStorage && localStorage.AuthService) {
      const rslt = JSON.parse(localStorage.AuthService);
      this.isAuth = rslt.isAuth;
      this.authToken = rslt.authToken;
      this.name = rslt.name;
    }
  }
  get AuthorizationHeader() { return this.authToken;  }
  get isAutenticated() { return this.isAuth; }
  get Name() { return this.name; }
  login(authToken: string, name: string ) {
    this.isAuth = true;
    this.authToken = authToken;
    this.name = name;
    if (localStorage) { localStorage.AuthService = JSON.stringify({isAuth, authToken, name}); }
  }
  logout() {
    this.isAuth = false;
    this.authToken = '';
    this.name = '';
    if (localStorage) { localStorage.removeItem('AuthService'); }
  }
}
```


113

LoginService

```
@Injectable({providedIn: 'root'})
export class LoginService {
  constructor(private http: HttpClient, private auth: AuthService) { }
  get isAutenticated() { return this.auth.isAutenticated;  }
  get Name() { return this.auth.Name;  }
  login(usr: string, pwd: string) {
    return new Observable(observable =>
      this.http.post('http://localhost:4321/login', { name: usr, password: pwd })
        .subscribe(
          data => {
            if(data['success']) { this.auth.login(data['token'], data['name']); }
            observable.next(this.auth.isAutenticated);
          },
          (err: HttpErrorResponse) => { observable.error(err); }
        )
    );
  }
  logout() { this.auth.logout(); }
}
```


114

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.
- 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.
- La recomendación es que el único sitio donde se puede manipular el árbol DOM debe ser en las directivas, para que entre dentro del ciclo de vida de compilación, binding y renderización del HTML que sigue el Angular.
- Aunque un componente es técnicamente una directiva-con-plantilla, conceptualmente son dos elementos completamente diferentes.
- Las directivas son clases decoradas con @Directive.

Características de las directivas

- Las directivas se clasifican en dos tipos:
  - directivas estructurales: que alteran el diseño mediante la adición, eliminación y sustitución de elementos en DOM.
  - directivas atributos: que alteran la apariencia o el comportamiento de un elemento existente.
- Prefijos en Directivas
  - El equipo de Angular 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 camel sigue el formato de los identificadores en JavaScript. Es el nombre que se usa en la documentación de Angular y el que se usa cuando se crea una nueva directiva.

119

Registrar

- Globalmente:

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

- En el contenedor:

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


120

@Directive

- **selector**: Selector CSS que indica a Angular que debe ejecutar la directiva cuando se encuentra un elemento con ese nombre en el HTML.
- **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

121

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; }
}

<button (winConfirm)="vm.delete(p.id)" winConfirmMessage="¿Estás seguro?">Borrar</button>
```


122

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.

123

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

```
@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>();
```


124

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()" >

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; }
```

Eventos del DOM

- Se podría llegar al DOM con JavaScript estándar y adjuntar manualmente los controladores de eventos pero hay por lo 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 pérdidas 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) { … }
```


Directivas estructurales

- 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 la etiqueta de plantilla 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.

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" >
```


129

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.

130

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(): void

- Destruye todas las vistas del contenedor.

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


131

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 a 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.
- 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');
  }
}
```


132

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 vacio 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.

133

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.

134

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>
<span class="errorMsg" [hidden]="!edad?.errors?.naturalNumber">No es un número entero positivo.</span>
```


Validaciones personalizadas

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

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

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

```
<input type="password" ... #pwd="ngModel">
<input type="password" name="valPwd" id="valPwd" [(ngModel)]="valPwd" #valPwd="ngModel"
        validateEqual="pwd">
<div *ngIf="valPwd?.errors?. validateEqual">Tiene que ir en mayúsculas.</div>
```

Validación HTML5

```
const URL_REGEXP = /^[a-z][a-z\d.+-]*:\/*(?:[^:@]+(?::[^:@]+)?@)?(?:[^\s:/?#]+|\[[a-f\d:]+])(?::\d+)?(?:\/[^?#]*)?(?:\?[^#]*)?(?:#.*)?$/i;
const EMAIL_REGEXP = /^(?=.{1,254}$)(?=.{1,64}@)[-!#$%&'*+/0-9=?A-Z^_`a-z{|}~]+(\.[-!#$%&'*+/0-9=?A-Z^_`a-z{|}~]+)*@[A-Za-z0-9]([A-Za-z0-9-
    ]{0,61}[A-Za-z0-9])?(\.[A-Za-z0-9]([A-Za-z0-9-]{0,61}[A-Za-z0-9])?)*$/;
const NUMBER_REGEXP = /^\s*(-|\+)?(\d+|(\d*(\.\d*)))([eE][+-]?\d+)?\s*$/;

@Directive({
  selector: '[type=url],[type=email],[type=number]',
  providers: [
    { provide: NG_VALIDATORS, useExisting: forwardRef(() => TypeValidatorDirective), multi: true }
  ]
})
export class TypeValidatorDirective implements Validator {
  constructor( private elem: ElementRef) { }
  validate(control: AbstractControl): { [key: string]: any } {
    const valor = control.value;
    const dom = this.elem.nativeElement;
    const tipo = dom.type.toLowerCase();
    if (valor) {
      if (dom.validity) {
        return dom.validity.typeMismatch ? { 'validationMessage': dom.validationMessage, 'typeMismatch': true } : null;
      }
      switch (tipo) {
        case 'url': return URL_REGEXP.test(valor) ? null : { 'typeMismatch': true };
        case 'email': return EMAIL_REGEXP.test(valor) ?  null : { 'typeMismatch': true };
        case 'number': return NUMBER_REGEXP.test(valor) ?  null : { 'typeMismatch': true };
      }
    }
    return null;
  }
}
```


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;
```


141

*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'};
}
```


142

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 }
```

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 esperan 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 son 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 genere 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

147

Contextos de saneamiento y seguridad

- La desinfección es la inspección de un valor que no es de confianza, convirtiéndolo en un valor que 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 <script 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.

148

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, puede decirle a Angular que 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 su aplicación.
- Para marcar un valor como confiable, ha 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.

149

FORMULARIOS

150

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 basados en plantillas

- Los formularios basados en plantillas delegan en la declaración de la plantilla la definición del formulario y el enlazado de datos por lo que requieren un código mínimo.
- Se colocan los controles de formulario HTML (<input>, <select>, <textarea>) en la plantilla del componente y se enlazan a las propiedades del modelo de datos suministrado por el componente, utilizando directivas como ngModel.
- No es necesario crear objetos de control de formulario Angular en la clase del componente dado que las directivas Angular los crean automáticamente, utilizando la información de los enlaces de datos.
- No es necesario traspasar los datos del modelo a los controles y viceversa dado que Angular lo automatiza con la directiva ngModel. Angular actualiza el modelo de datos mutable con los cambios de usuario a medida que suceden.
- Aunque esto significa menos código en la clase de componente, los formularios basados en plantillas son asíncronos, lo que puede complicar el desarrollo en escenarios más avanzados.

Encapsulación

- Angular encapsula automáticamente cada formulario y sus controles en un heredero de AbstractControl.
- Expone una serie de propiedades para gestionar los cambios, la validación y los errores:
  - pristine / dirty: Booleanos que indican si el formulario o control no ha sido o ya ha sido modificado por el usuario.
  - touched / untouched: Booleanos que indican si el formulario o control ha sido o no tocado por el usuario.
  - disabled / enabled: Booleanos que indican si el formulario o control esta deshabilitado o habilitado.
  - valid / invalid: Booleanos que indican si el formulario o control es valido o invalido.
  - pending : Booleano que indica si la validación está pendiente.
  - status: Cadena que indica el estado: VALID, INVALID, PENDING, DISABLED.
  - errors: Contiene como propiedades las validaciones que han fallado.
  - hasError() y getError() (v2.2): acceden a un error concreto.
  - value: valor del control

153

Avisos visuales

- Angular esta preparado para cambiar el estilo visual de los controles según su estado, utilizando class de CSS.
- Para personalizar el estilo (en la hojas de estilos):

```
.ng-valid[required], .ng-valid.required, .ng-pending {
  border-left: 5px solid #42A948; /* green */
}
.ng-invalid:not(form) {
  border-left: 5px solid #a94442; /* red */
}
```

- En caso de estar definidos Angular aplicará automáticamente los siguientes estilos:

Estado	Clase si es cierto	Clase si es falso
El control ha sido visitado	ng-touched	ng-untouched
El valor ha sido modificado	ng-dirty	ng-pristine
El valor es válido	ng-valid	ng-invalid

154

Directivas

- **ngForm**: Permite crear variables referencia al formulario.

```
<form name="miForm" #miDOMForm>
<form name="miForm" #miForm="ngForm">
        <button type="submit" [disabled]="miForm.invalid">
```

- **ngModel**: Permite el enlazado bidireccional y la creación de variables referencia.

```
<input type="text" name="nombre" required [(ngModel)]="model.nombre"
        #nombre="ngModel">
<div [hidden]="nombre.valid">Obligatorio</div>
<input type="text" #miDOMElement>
```

- **ngModelGroup**: Permite crear variables referencia de un grupo de controles de formulario.

```
<fieldset ngModelGroup="direccion" #direccionCtrl="ngModelGroup">
    <input name="calle" [ngModel]="direccion.calle" minlength="2">
    <input name="localidad" [ngModel]="direccion.localidad" required>
</fieldset >
```


Tipos de Validaciones

- required: el campo es requerido, aplicable a cualquier <input> , <select> o <textarea>, con la directiva required o [required] (vincula al modelo).

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

- minlength: El campo debe tener un nº mínimo de caracteres.
- maxlength: El campo debe tener un nº máximo de caracteres.
- pattern: El campo debe seguir una expresión regular, applicable a <input> o a <textarea>.

```
<input type="text" [(ngModel)]="model.nombre"
  name="nombre" minlength="3" maxlength="50"
  pattern="/^[a-zA-Z]*$/" >
```

Tipos de Validaciones (v.2+)

- min: El campo debe tener un valor mínimo
- max: El campo debe tener un valor máximo

```
<input type="number"
  [(ngModel)]="model.edad" name="edad"
  min="18" max="99" >
```

- email: El campo debe tener el formato de un correo electrónico. Deberemos indicar en el <input> que type="email".

```
<input type="email"
  [(ngModel)]="model.correo" name="correo">
```


Comprobando las validaciones

- valid y invalid indican a nivel global si pasa la validación. Para saber que validaciones han fallado, la colección errors dispone de una propiedad por cada tipo de validación.

```
nombre.errors?.required
nombre.errors?.pattern
```

- Si no hay validaciones o si se cumplen todas las validaciones, la propiedad errors es nula por lo que es necesario utilizar el operador de navegación segura (?.) para evitar problemas
- Las propiedades del objeto errors se denominan de la misma forma que la validación que ha fallado y suministran información complementaria.
- La comprobación se puede realizar a nivel de control individual o de formulario completo.
- V 2.2: hasError y getError simplifican el acceso a los errores.
- Con el atributo HTML novalidate en la etiqueta <form> se evita que el navegador haga sus propias validaciones que choquen con las de Angular.

```
<form name="miForm" novalidate >
```

Notificación de errores

- Con la directiva *ngIf y con [hidden] se puede controlar cuando se muestran los mensajes de error.

```
<span class="error"
   [hidden]="!nombre?.errors?.maxlength">El tamaño
   m&aacute;ximo debe ser 50</span>
<span class="error"
   [hidden]="!nombre.hasError('required')">Es
   obligatorio</span>
```

- Para resumir en un mensaje varias causas de error:

```
<span class="error"
   [hidden]="!nombre?.errors?.minlength &&
   !nombre?.errors?.maxlength">El nombre debe tener
   entre 3 y 50 letras</span>
```

- Para no mostrar los mensajes desde el principio sin dar oportunidad al usuario a introducir los datos:

```
<span [hidden]="nombre.valid || miForm.pristine" …
```


159

Enviar formulario

- La propiedad disabled permite deshabilitar el botón de envío del formulario mientras no sea valido.

```
<button (click)="enviar()"
   [disabled]="miForm.invalid">Enviar</button>
```

- La directiva ngSubmit permite interceptar el evento de envío del formulario:

```
<form #miForm="ngForm" (ngSubmit)="onSubmit(miForm)">
    <input type="submit"
       [disabled]="miForm.invalid">Enviar</button>
onSubmit(miForm: NgForm) {
  if (miForm.valid) {
    …
  } else {
    alert("Hay datos inválidos");
  }
}
```


160

Consultas a la plantilla

- Los formularios basados en plantillas delegan la creación de sus controles de formulario y el enlazado en directivas, por lo que requieren un mínimo de código y acoplamiento entre plantilla y clase del componente.
- 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.
- Dado que los formularios basados en plantillas son asíncronos hay que controlar el ciclo de vida para evitar errores:

```
miForm: NgForm;
@ViewChild('miForm') currentForm: NgForm;

ngAfterViewChecked() {
 this.formChanged();
}
formChanged() {
 if (this.currentForm === this.miForm) { return; }
 this.miForm = this.currentForm;
 // ...
}
```


161

Validaciones personalizadas

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

@Directive({
 selector: '[upperCase]',
 providers: [{provide: NG_VALIDATORS, useExisting: UpperCaseValidatorDirective, multi: true}]
})
export class UpperCaseValidatorDirective implements Validator {
 validate(control: AbstractControl): {[key: string]: any} {
  const valor = control.value;
  if (valor) {
   return valor !== valor.toUpperCase() ? {'upperCase': {valor}} : null;
  } else {
   return null;
  }
 }
}
<input type="text" name="nombre" [(ngModel)]="model.nombre" #nombre="ngModel" upperCase>
<div *ngIf="nombre?.errors?.upperCase">Tiene que ir en mayúsculas.</div>
```


162

Validaciones personalizadas

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

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

      if (valor) {
        return (!cntrlBind || valor !== cntrlBind.value) ? { 'validateEqual': `${valor} <> ${cntrlBind.value}` } : null;
      }
      return null;
   }
}
<input type="password" ... #pwd="ngModel">
<input type="password" name="valPwd" id="valPwd" [(ngModel)]="valPwd" #valPwd="ngModel"
        validateEqual="pwd">
<div *ngIf="valPwd?.errors?. validateEqual">Tiene que ir en mayúsculas.</div>
```


Formularios Reactivos

- Las 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 de control de formulario Angular (AbstractControl) en la clase de 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 de formulario y reaccionar a 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 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 de los cuales 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 crear formularios.

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

- 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.forEach(r => fa.push(
  new FormGroup({ role: new FormControl(r.role , Validators.required) })
));
this.miForm = new FormGroup({
  user: new FormControl(''),
  password: // ...
  roles: fa
});
```


Agregaciones

- Añadir un nuevo elemento a la agregación:

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

- Borrar 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
});
```


182

Validaciones personalizadas

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

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

- Para un FormGroup:

```
function passwordMatchValidator(g: FormGroup) {
  return g.get('passwordValue').value ===
            g.get('passwordConfirm').value ? null : {'mismatch': true};
}
```


183

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 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();
```


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
  });
```

FormBuilder

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

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

- Mediante una estructura se crea el formulario:

```
const fa = this.fb.array([]);
this.model.roles.forEach(r => fa.push(this.fb.group({ role: [r.role,
        Validators.required] })));
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: fa
});
```


186

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).

```
<div formGroupName="password" >
```

- 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>
```


187

Mostrar errores

- Errores en los controles del formulario:

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

- 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}}
```


188

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}}
```


189

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 }}
```


190

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 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 async() o en fakeAsync() para evitar buscar valores en el formulario que aún no están disponibles. Con los formularios reactivo, todo está disponible tal y como se espera que sea.

191

ANIMACIONES

Introducción

- El movimiento es un aspecto importante en el diseño de aplicaciones web modernas. Las buenas interfaces de usuario se transfieren entre los estados con atractivas animaciones que llaman la atención donde se necesita. Las animaciones bien diseñadas pueden hacer que una interfaz de usuario no solo sea más divertida sino también más fácil de usar.
- El sistema de animación de Angular le permite crear animaciones que se ejecutan con el mismo tipo de rendimiento nativo que se encuentra en las animaciones de CSS puro. También se puede integrar estrechamente la lógica de animación con el resto del código de la aplicación, para facilitar el control.
- Las animaciones Angular se crean sobre la Web Animations API estándar y se ejecutan de forma nativa en navegadores compatibles.
- *Para navegadores antiguos requiere activar el polyfills web-animations-js (npm install --save web-animations-js) en polyfills.ts.*
- Hay que importarlo en el módulo principal como se haría con cualquier otro modulo Angular.

```
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
@NgModule({ imports: [[ BrowserModule, BrowserAnimationsModule,  // …
```

Definición de Animaciones

- Las animaciones se pueden definir dentro de los metadatos de @Component

```
@Component({
  animations: [
    trigger('triggerName', [
      state('inactive', style({ … })),
      state('active',   style({ … })),
      transition('inactive => active', animate('100ms ease-in')),
      transition('active => inactive', animate('100ms ease-out'))
    ])
  ]
})
```

- Se puede definir trigger (desencadenadores de animación) en los metadatos del componente. Se utilizan animaciones para la transición entre dos estados (valores).
- Usando la sintaxis [@triggerName], se adjunta la animación a uno o más elementos en la plantilla del componente y se vincula al contendor de estado lógico.

```
<p [@triggerName]="model.state" >…</p>
```


Estados y transiciones

- Las animaciones Angular se definen como estados lógicos y transiciones entre estados.
- Un estado de animación es un valor cadena que se define en el código de la aplicación.
- La fuente del estado puede ser un atributo o propiedad de objeto o puede ser un valor calculado en un método. Lo importante es que se pueda vincular en la plantilla del componente.
- Se pueden definir estilos para cada valor de estado especificando los propiedades de estilo finales del estado y se aplican al elemento una vez que ha realizado la transición a ese estado y se conserva mientras permanezca en ese estado.
- Básicamente se están definiendo varios estilos para el mismo elemento que se aplican dependiendo del estado.

```
state('active', style({ color: '#cfd8dc', transform: 'scale(1.1)'})),
```

Propiedades animatables y unidades

- Dado que el soporte de animación de Angular se basa en Web Animations, se puede animar cualquier propiedad que el navegador considere animable, lo que incluye posiciones, tamaños, transformaciones, colores, bordes y muchos otras (https://www.w3.org/TR/css-transitions-1/#animatable-properties).
- Para propiedades que tienen un valor numérico, se puede definir una unidad proporcionando el valor como una cadena con el sufijo apropiado:
  - '50px'
  - '3em'
  - '100%'
- Si no proporciona una unidad, Angular asume como valor predeterminado los px:
  - 50 es lo mismo que decir '50px'
- Se puede usar el * como valor de propiedad para que en tiempo de ejecución el valor de la propiedad se sustituya por el valor actual antes de comenzar la animación.

Transiciones

- Después de definir estados, se pueden definir transiciones entre los estados: cambio del valor original de estado lógico a un nuevo valor.
- Cada transición controla el tiempo de cambio entre un conjunto de estilos y el siguiente:

```
transition('inactive => active', animate('100ms ease-in')),
transition('active => inactive', animate('100ms ease-out'))
```

- Si varias transiciones tienen la misma configuración de tiempo, puede combinarlas en la misma definición:

```
transition('inactive => active, active => inactive', animate(100))
```

- Cuando ambas direcciones de una transición tienen el mismo tiempo se puede usar la sintaxis abreviada <=>:

```
transition('inactive <=> active', animate(100))
```

Transiciones

- El estado * coincide con cualquier estado de animación permitiendo definir transiciones que se aplican independientemente del estado en que se encuentre la animación:

```
transition('* => active', animate('100ms ease-in')),
transition('active => *', animate('100ms ease-out'))
```

- El estado especial **void**, sin estado, se aplica cuando el elemento no está presente en una vista, porque aún no se ha agregado o se ha eliminado, permite definir animaciones de entrada y salida.

```
transition('void => active', animate('100ms ease-in')),
transition('active => void', animate('100ms ease-out')),
transition('void => *', animate('100ms ease-in')),
transition('* => void', animate('100ms ease-out'))
```

- Estas dos últimas transiciones comunes tienen sus propios alias:

```
transition(':enter', [ ... ]); // void => *
transition(':leave', [ ... ]); // * => void
```

- Los alias :increment y :decrement se pueden usar para iniciar una transición cuando el **valor numérico** del estado aumenta o disminuye.

198

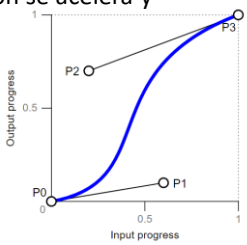
Temporización

- Hay tres propiedades de temporización que pueden fijar la cadencia para cada transición animada: la duración, la demora y la velocidad. Todas se combinan en una única cadena de temporización.
- La duración controla el tiempo que tarda la animación en ejecutarse de principio a fin. Se puede definir de tres maneras:
  - Como un número aritmético, en milisegundos: 100
  - En una cadena, en milisegundos: '100ms'
  - En una cadena, en segundos: '0.1s'
- La demora controla la cantidad de tiempo entre el activador de animación y el comienzo de la transición. Se puede definir agregándolo a la misma cadena después de la duración expresado en milisegundos o segundos:
  - Espera 100ms y luego ejecútala en 200ms: '0.2s 100ms‘
- La función easing especifica la velocidad de la animación para hacerla más realista.

199

Función easing

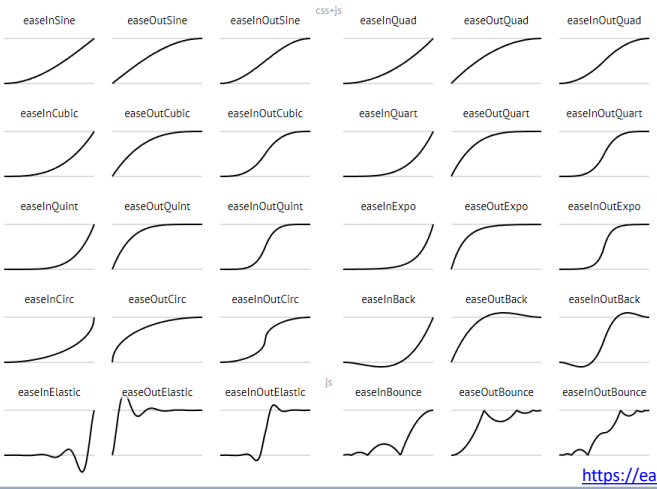
- Un objeto real no comienza su movimiento instantáneamente y de manera constante (lineal). Cuando abrimos un cajón, primero le damos aceleración, y después lo frenamos. Cuando algo se cae, primero baja rápidamente y después de alcanzar el suelo rebota.
- La función de cubic-bezier controla cómo la animación se acelera y desacelera durante su tiempo de ejecución.
  - cubic-bezier(P0, P1, P2, P3)
- Funciones predefinidas (CSS3):
  - ease: equivale a cubic-bezier(0.25, 0.1, 0.25, 1).
  - ease-in: equivale a cubic-bezier(0.42, 0, 1, 1).
  - ease-out: equivale a cubic-bezier(0, 0, 0.58, 1).
  - ease-in-out: equivale a cubic-bezier(0.42, 0, 0.58, 1).



- Se establecen en la cadena como segundo o tercer parámetro:
  animate("200ms ease-in")
  animate("5s 10ms cubic-bezier(.17,.67,.88,.1)")

200

Función easing



https://easings.net/es

201

Estilo durante una animación

- También puede aplicar un estilo durante una animación pero no mantenerla después de que termine la transición.
- Se deben definir en el transition dichos estilos en línea:

```
transition('void => *', [
    style({transform: 'translateX(-100%)'}),
    animate(100)
  ]),
transition('* => void', [
    animate(100, style({transform: 'translateX(100%)'}))
  ])
```


Keyframes

- Las animaciones de varios pasos con fotogramas clave (keyframes) van más allá de una simple transición, son una animación más intrincada que pasa por uno o más estilos intermedios al hacer la transición entre el estilo inicial y el estilo final.
- Para cada fotograma clave se especifica un estilo y un desplazamiento que defina en qué punto de la animación se aplica ese fotograma clave. El desplazamiento es un número entre 0 (que marca el comienzo de la animación) y 1 (que marca el final).
- Los desplazamientos no están definidos en términos de tiempo absoluto, son medidas relativas a la línea de tiempo final de la animación, basado en la combinación de desplazamientos de fotogramas clave, duración, retraso y velocidad.

```
animate(300, keyframes([
    style({opacity: 0, transform: 'translateX(-100%)', offset: 0}),
    style({opacity: 1, transform: 'translateX(15px)',  offset: 0.3}),
    style({opacity: 1, transform: 'translateX(0)',     offset: 1.0})
  ]))
```

Grupos paralelos de animaciones

- Es posible que desee configurar diferentes tiempos para animar las diferentes propiedades del estilo y que ocurra en paralelo.
- Para ello se pueden usar los grupos de animación, cada animación del grupo se aplica al mismo elemento en paralelo, pero se ejecutan independientemente unas de otras:

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


Deshabilitar animaciones

- Se puede colocar un enlace de control de animación especial llamado @.disabled en un elemento que establecerá si se deshabilitará las animaciones para cualquier animación situada dentro del elemento, así como cualquier animación en el elemento en sí.
- Cuando sea verdadero, el enlace @.disabled evitará que se reproduzcan todas las animaciones.

```
<p [@.disabled]="model.isDisabled"
```

- Para deshabilitar todas las animaciones en toda la aplicación: si se desactivan en componente principal todos los componentes internos también tendrán sus animaciones desactivadas, se desactivan todas.

```
class AppComponent {
  @HostBinding('@.disabled') public animationsDisabled = true;
}
```

Eventos

- Se activa una devolución de llamada cuando se inicia una animación y también cuando finaliza.
- Esto permite personalizar lo que se quiere hacer cuando empieza una animación y cuando termina.

```
<p [@triggerName]="model.state"
    (@triggerName.start)="animationStarted($event)"
    (@triggerName.done)="animationDone($event)"
>…</p>
```

- Reciben un AnimationEvent que contiene propiedades útiles tales como fromState, toState, totalTime, phaseName, element, triggerName y disabled.
- Si están desactivadas las animaciones, las devoluciones de llamada de la animación se seguirán disparando de forma normal, solo que sin tiempos de espera.

RESPONSIVE WEB DESIGN

Diseño Adaptativo

- Es un enfoque de diseño destinado a la elaboración de sitios/aplicaciones para proporcionar un entorno óptimo de:
  - Lectura Fácil
  - Navegación correcta con un número mínimo de cambio de tamaño
  - Planificaciones y desplazamientos
- Con la irrupción multitud de nuevos dispositivos y que el acceso a internet se realiza ya mayoritariamente desde dispositivos diferentes a los tradicionales ordenadores ha obligado a seguir dicho enfoque en las aplicaciones WEB.
- Contempla la definición de múltiples elementos antes de la realización de la programación real.
  - Elementos de página en las unidades de medidas correctas
  - Imágenes flexibles
  - Utilización de CSS dependiendo de la aplicación

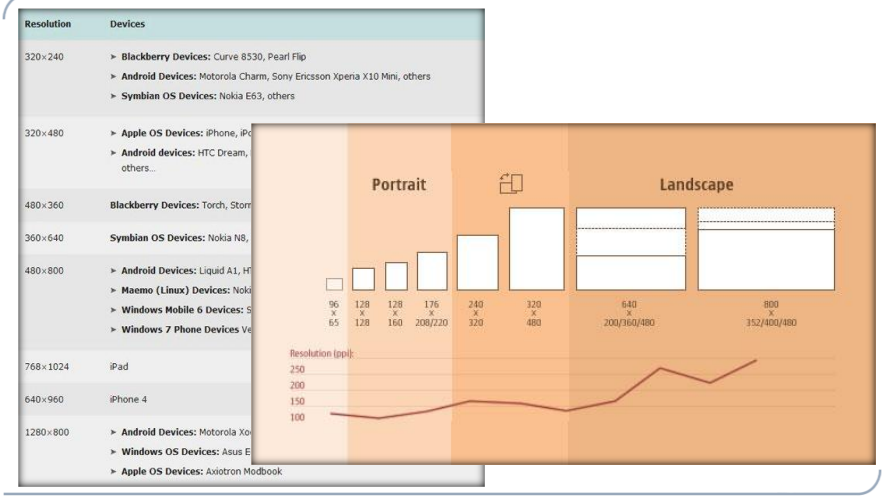
208

Resolución

- Los dispositivos móviles tienen una característica distintiva, y es su resolución de pantalla.
- Es necesario conocer cuales son las resoluciones más comunes en este tipo de dispositivos móviles, de los gadgets más utilizados, etc.
- Las resoluciones van cambiando de forma muy rápida y en dispositivos nuevos

209

Resolución



210

Resolución

- También deberemos tener en cuenta la resoluciones de otros dispositivos como:
  - Tablets
  - TV SmartTV
  - Pizarras electrónicas, etc

Pizarras	**10.1" y 11.6" (2560x1440, 1920x1080, 1366x768), 17" (1920x1080)**
PC	**12" (1280x800), 14" (1920x1080, 1366x768), 15.6" (1920x1080)**
Family hub	**23" (1920x1080), 27" (2560x1440)**

211

Orientación de Página

- La orientación del papel es la forma en la que una página rectangular está orientada y es visualizada.
- Los dos tipos más comunes son:
  - Landscape (Horizontal)
  - Portrait (Vertical)

212

Recomendaciones de Diseño

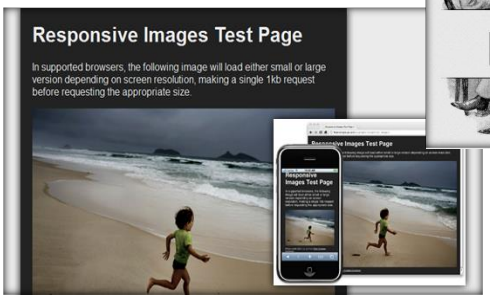
1. Utilizar porcentajes y "ems" como unidad de medida en lugar de utilizar los valores determinados como definición de pixel.

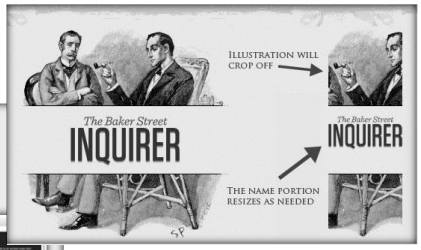
Las ems son unidades relativas, así que más exactamente 1 em equivale al cien por cien del tamaño inicial de fuente.

213

Recomendaciones de Diseño

2. Determinar el tamaño y definición de las imágenes a utilizar
   - Recortar, Ajustar, etc





214

Recomendaciones de Diseño

3. El contenido y funcionalidad BÁSICA debe de ser accesible por todos los navegadores

215

Recomendaciones de Diseño

4. Definición correcta de contenidos en función del dispositivo

216

Ventajas

- Soporte de dispositivos móviles.
- Con una sola versión en HTML y CSS se cubren todas las resoluciones de pantalla.
- Mejora la experiencia de usuario.
- Se reducen los costos de creación y mantenimiento cuando el diseño de las pantallas es similar entre dispositivos de distintos tamaños.
- Evita tener que desarrollar aplicaciones especificas para cada sistema operativo móvil.
- Facilita la referenciación y posicionamiento en buscadores, versión única contenido/página.

217

HTML5: viewport

- Hace referencia a la región visible del navegador, o sea, la parte de la página que está visualizándose actualmente en el navegador.
- Podemos redimensionar la ventana del navegador para reducir el tamaño del viewport y simular que se trata de una pantalla y dispositivo más pequeño.

```
<meta name="viewport" content="initial-scale=1, width=device-width">
```

- Los parámetros de comportamiento para el viewport son:
  - width: Indica un ancho para el viewport.
  - height: Indica un alto para el viewport.
  - initial-scale: Escala inicial con la que se visualiza la página web.
  - minimum-scale: Escala mínima a la que se puede reducir al hacer zoom.
  - maximum-scale: Escala máxima a la que se puede aumentar al hacer zoom.
  - user-scalable: Posibilidad de hacer zoom en la página web.

CSS3: Consultas de medios

- Las consultas de medios en CSS3 extiende la idea de CSS2: en lugar de buscar un tipo de dispositivo, se mira la capacidad del dispositivo.
- Las consultas de medios se pueden utilizar para comprobar muchas cosas, tales como:
  - anchura y la altura de la ventana gráfica
  - anchura y la altura del dispositivo
  - orientación (es la tableta / teléfono en modo horizontal o vertical)
  - resolución

```
@media not|only mediatype and (expressions) {
  CSS-Code;
}
<link rel="stylesheet" media="mediatype and|not|only (expressions)" href="print.css">
```

- Las consultas de medios son una técnica popular para la entrega de una hoja de estilo a medida para tabletas, iPhone y Androids.

```
@media screen and (min-width: 480px) {
  body {
    background-color: lightgreen;
  }
}
```

Twitter Bootstrap

- Twitter Bootstrap es un framework o conjunto de herramientas de software libre para diseño de sitios y aplicaciones web adaptativas (diseño).
- Bootstrap se puede descargar y usar de forma totalmente gratuita.
- Contiene plantillas de diseño con tipografía, formularios, botones, cuadros, menús de navegación y otros elementos de diseño basado en HTML y CSS, así como, extensiones de JavaScript opcionales adicionales.
- Bootstrap es el framework libre de cliente más rápido y fácil para el desarrollo web adaptativo apto para dispositivos móviles
- Permite la creación de sitios web que se ajustan a sí mismos automáticamente para que se visualicen correctamente en todos los dispositivos, desde pequeños teléfonos a grandes televisores.
- Todos los plugins JavaScript de Bootstrap requieren la librería jQuery para funcionar, por lo que se deberá incluir.

Material Design

- Material Design es un concepto, una filosofía, unas pautas enfocadas al diseño utilizado en Android, pero también en la web y en cualquier plataforma: Un diseño adaptado para todo tipo de pantallas.
- Fue desarrollado por Google y anunciado en la conferencia Google I/O celebrada el 25 de junio de 2014. Material se integró en Android Lollipop como reemplazo de Holo. La filosofía también se aplicó en Google Drive y Google Docs, se ha extendido Gmail y Google Calendar proporcionando una experiencia consistente en todas las plataformas.
- Material Design recibe su nombre por estar basado en objetos materiales. Piezas colocadas en un espacio (lugar) y con un tiempo (movimiento) determinado. Es un diseño donde la profundidad, las superficies, los bordes, las sombras y los colores juegan un papel principal.
- Es una manera de intentar aproximarse a la realidad, algo que en un mundo donde todo es táctil y virtual es difícil. Quiere guiarse por las leyes de la física, donde las animaciones sean lógicas, los objetos se superpongan pero no puedan atravesarse el uno al otro y demás. Elementos ordenados e imágenes claras. Luz y sombras dan sensación de jerarquía. El movimiento es la mejor forma de guiar al usuario.
- El equipo de Angular a desarrollado http://material.angular.io con los componentes Material Design construidos para y con Angular.

CONCEPTOS AVANZADOS

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, 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:
  - 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 xi18n --output-path i18n
```

  - 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

- El atributo i18n Angular marca un contenido como traducible y hay utilizarlo en cada etiqueta o atributo cuyo contenido literal deba ser traducido.

  ```
  <h1 i18n>Hello!</h1> <img [src]="logo" i18n-title title="Logo" /> <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 xi18n --i18n-format=xlf2
  - ng xi18n --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 message.ts (por ejemplo):

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

- Crea un duplicado por idioma siguiendo el mismo convenio message.es.ts.

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

- Sustituir los literales del código por:

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


228

Localización: Configuración

```
"build": {
  "configurations": {
    "production": { … }
    "es": {
      "aot": true,
      "outputPath": "dist/avanzado/es/",
      "i18nFile": "src/i18n/messages.es.xlf",
      "i18nFormat": "xlf",
      "i18nLocale": "es",
      "fileReplacements": [
        {
          "replace": "src/i18n/messages.ts",
          "with": "src/i18n/messages.es.ts"
        }
      ]
    }
    …
"serve": {
  "configurations": {
    "production": { … }
    "es": {
      "browserTarget": "avanzado:build:es"
    }
```


229

Localización : Configuración

- Duplicar la entrada "production" dentro de "build" : "configurations" y añadir:

```
"production": { … }
"es": {
  "baseHref": "/es/",
  "outputPath": "dist/myApp/es/",
  "i18nFile": "src/i18n/messages.es.xlf",
  "i18nFormat": "xlf",
  "i18nLocale": "es",
  "fileReplacements": [{
     "replace": "src/i18n/messages.ts",
     "with": "src/i18n/messages.es.ts"
   }],
  …
}
```

- Hacer lo mismo en "serve": "configurations": {

```
"production": { … }
"es": {
  "browserTarget": "myApp:build:es"
}
```


230

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>
```


231

Publicación de módulos con NPM

- 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 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. ng build my-lib --prod
  2. cd dist/my-lib
  3. 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

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

235

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.

236

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">
```


237

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">
```


238

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 " />
```


239

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.

240

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});
 }
```


TEST

Ingeniería de Software

- El JavaScript es un lenguaje muy poco apropiado para trabajar en un entorno de calidad de software.
- En descargo del lenguaje JavaScript y de su autor, Brendan Eich, hay que decir que los problemas que han forzado esta evolución del lenguaje (así como las críticas ancestrales de la comunidad de desarrolladores) vienen dados por lo que habitualmente se llama "morir de éxito".
- Jamás se pensó que un lenguaje que Eich tuvo que montar en 12 días como una especie de "demo" para Mozilla, pasase a ser omnipresente en miles de millones de páginas Web.
- O como el propio Hejlsberg comenta:

  "JavaScript se creó –como mucho- para escribir cien o doscientas líneas de código, y no los cientos de miles necesarias para algunas aplicaciones actuales."

Principios fundamentales

- Las pruebas exhaustivas no son viables
- El proceso de pruebas no puede demostrar la ausencia de defectos
- Las pruebas no garantizan ni mejoran la calidad del software
- Las pruebas tienen un coste
- Hay que ejecutar las pruebas bajo diferentes condiciones
- Inicio temprano de pruebas

Desarrollo Guiado por Pruebas (TDD)

- El Desarrollo Guiado por Pruebas, es una técnica de programación (definida por KentBeck); consistente en desarrollar primero el código que pruebe una característica o funcionalidad deseada antes que el código que implementa dicha funcionalidad.
- El objetivo a lograr es que no exista ninguna funcionalidad que no esté avalada por una prueba.
- Lo primero que hay que aprender de TDD son sus reglas básicas:
  - No añadir código sin escribir antes una prueba que falle
  - Eliminar el Código Duplicado empleando Refactorización

Ritmo TDD

- TDD invita a seguir una serie de tareas ordenadas, que a menudo se denomina ritmo TDD, y que se basa en los siguientes pasos:
  1. Escribir una prueba que demuestre la necesidad de escribir código.
  2. Escribir el mínimo código para que el código de pruebas compile
  3. Implementar exclusivamente la funcionalidad demandada por las pruebas
  4. Mejorar el código (Refactoring) sin añadir funcionalidad
  5. Volver al primer paso
- Este ritmo permite formalizar las tareas que se han de realizar para conseguir un código fácil de mantener, bien diseñado y que se puede probar automáticamente.

Las tres partes del test: AAA

- Arrange (Preparar):
  - Inicializa objetos y establece el valor de los datos que se pasa al método en pruebas de tal forma que los resultados sean predecibles.
- Act (Actuar)
  - Invoca al método a probar con los parámetros preparados.
- Assert (Afirmar)
  - Comprobar si la acción del método probado se comporta de la forma prevista. Puede tomar la forma de:
    - Aserción: Es una afirmación que se hace sobre el resultado y puede ser cierta o no.
    - Expectativa: Es la expresión del resultado esperado que puede cumplirse o no.

Arrange (Preparar)

- **Fixture**: Es el término se utiliza para hablar de los datos de contexto de las pruebas, aquellos que se necesitan para construir el escenario que requiere la prueba.
- **Dummy**: Objeto que se pasa como argumento pero nunca se usa realmente. Normalmente, los objetos dummy se usan sólo para rellenar listas de parámetros.
- **Fake**: Objeto que tiene una implementación que realmente funciona pero, por lo general, usa una simplificación que le hace inapropiado para producción (como una base de datos en memoria por ejemplo).
- **Stub**: Objeto que proporciona respuestas predefinidas a llamadas hechas durante los tests, frecuentemente, sin responder en absoluto a cualquier otra cosa fuera de aquello para lo que ha sido programado. Los stubs pueden también grabar información sobre las llamadas.
- **Mock**: Objeto preprogramado con expectativas que conforman la especificación de cómo se espera que se reciban las llamadas. Son más complejos que los stubs aunque sus diferencias son sutiles.

Beneficios de TDD

- Reducen el número de errores y bugs ya que éstos, aplicando TDD, se detectan antes incluso de crearlos.
- Facilitan entender el código y que, eligiendo una buena nomenclatura, sirven de documentación.
- Facilitan mantener el código:
  - Protege ante cambios, los errores que surgen al aplicar un cambio se detectan (y corrigen) antes de subir ese cambio.
  - Protegen ante errores de regresión (rollbacks a versiones anteriores).
  - Dan confianza.
- Facilitan desarrollar ciñéndose a los requisitos.
- Ayudan a encontrar inconsistencias en los requisitos
- Ayudan a especificar comportamientos
- Ayudan a refactorizar para mejorar la calidad del código (Clean code)
- A medio/largo plazo aumenta (y mucho) la productividad.

249

Herramientas y tecnologías

Tecnología	Propósito
Jasmine	El marco de trabajo Jasmine proporciona todo lo necesario para escribir pruebas unitarias. Cuenta con una página HTML que ejecuta pruebas en el navegador.
Utilidades Angular de pruebas	Las utilidades Angular de pruebas crean casos de prueba para el código de la aplicación Angular bajo prueba. Permiten añadir y controlar partes de la aplicación a medida que interactúan dentro del entorno Angular.
Karma	El lanzador de pruebas Karma es ideal para escribir y ejecutar pruebas unitarias, mientras se desarrolla la aplicación. Puede ser una parte integral de los procesos de desarrollo e integración continua del proyecto.
Protractor	Permite escribir y ejecutar pruebas de extremo a extremo (E2E), para explorar la aplicación tal y como los usuarios la experimentan. En las pruebas E2E: en un proceso se ejecuta la aplicación real y en un segundo proceso se ejecuta las pruebas Protractor, que simulan el comportamiento del usuario y comprueban que la aplicación responde en el navegador tal y como se esperaba.
Selenium WebDriver	El Selenium es un conjunto de herramientas para automatizar los navegadores web, un robot que simula la interacción del usuario con el navegador.

250

DOCUMENTADOR

Documentación

- Hacer la documentación del código fuente puede llegar a ser muy tedioso, todos los programadores prefieren ir directo al grano, escribir su código y pasar de largo esta aburrida tarea. Por fortuna, actualmente existen un montón de herramientas para agilizar la documentación del código sin tener que dedicarle más tiempo del imprescindible.
- JSDoc es una sintaxis para agregar comentarios con documentación al código fuente de JavaScript.
- La sintaxis JSDoc es similar a la sintaxis de Javadoc, usado para documentar el código de Java, pero se ha especializado para trabajar con la sintaxis de JavaScript, es más dinámico y, por tanto único, ya que no es totalmente compatible con Javadoc. Sin embargo, como Javadoc, JSDoc permite al programador crear Doclets y Taglets que luego se pueden traducir en formatos como HTML o RTF.

```
/**
 * Create a dot.
 * @param {number} x - The x value.
 * @param {number} y - The y value.
 * @param {number} width - The width of the dot, in pixels.
 */
constructor(x, y, width) {
```

JSDoc

Etiqueta	Descripción
@author	nombre del autor.
@constructor	indica el constructor.
@deprecated	indica que ese método es deprecated.
@exception	sinónimo de @throws.
@param	parámetros de documentos y métodos.
@private	indica que el método es privado.
@return	indica que devuelve el método.
@see	Indica la asociación con otro objeto.
@throws	Indica la excepción que puede lanzar un método.
@version	indica el número de versión o librería.

253

Documentador

- JDOC
  - http://usejsdoc.org/index.html
  - https://github.com/jsdoc3/jsdoc
  - http://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=881:guia-de-estilo-javascript-comentarios-proyectos-jsdoc-param-return-extends-ejemplos-cu01192e&catid=78&Itemid=206
- Doxygen
  - http://www.stack.nl/~dimitri/doxygen/
- Compodoc
  - https://compodoc.github.io/website/
- Typedoc
  - https://typedoc.org

254

TEST UNITARIOS

JSLint, JSHint y TSLint

- Los analizadores de código son herramientas que realizan la lectura del código fuente y devuelve observaciones o puntos en los que tu código puede mejorarse desde la percepción de buenas prácticas de programación y código limpio.
- JSHint es un analizador online de código JavaScript (basado en el JSLint creado por Douglas Crockford) que nos permitirá mostrar puntos en los que tu código no cumpla unas determinadas reglas establecidas de “código limpio“.
- El funcionamiento de JSHint es el siguiente: toma nuestro código, lo escanea y, si encuentra un problema, devuelve un mensaje describiéndolo y mostrando su ubicación aproximada.
- Para descargar e instalar:
  - npm install -g jshint
- Existen “plug-in” para la mayoría de los entornos de desarrollo (http://jshint.com). Se puede automatizar con GRUNT o GULP.

Jasmine

- Jasmine es un framework de desarrollo dirigido por comportamiento (behavior-driven development, BDD) para probar código JavaScript.
  - No depende de ninguna otra librería JavaScript.
  - No requiere un DOM.
  - Tiene una sintaxis obvia y limpia para que se pueda escribir pruebas fácilmente.
- Prácticamente se ha convertido en el estándar de facto para el desarrollo con JavaScript.
- Para su instalación “standalone”, descargar y descomprimir:
  - https://github.com/jasmine/jasmine/releases
- Mediante npm:
  - npm install -g jasmine

SpecRunner.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Jasmine Spec Runner v2.5.0</title>
  <link rel="shortcut icon" type="image/png" href="lib/jasmine-2.5.0/jasmine_favicon.png">
  <link rel="stylesheet" href="lib/jasmine-2.5.0/jasmine.css">
  <script src="lib/jasmine-2.5.0/jasmine.js"></script>
  <script src="lib/jasmine-2.5.0/jasmine-html.js"></script>
  <script src="lib/jasmine-2.5.0/boot.js"></script>
  <script type="text/javascript" src="angular.js"></script>
  <script type="text/javascript" src="angular-mocks.js"></script>
  <!-- include source files here... -->
  <script src="src/Player.js"></script>
  <script src="src/Song.js"></script>
  <!-- include spec files here... -->
  <script src="spec/SpecHelper.js"></script>
  <script src="spec/PlayerSpec.js"></script>
</head>
<body></body>
</html>
```

Suites

- Una “suite” es un nombre que describe al género o sección qué se va a pasar por un conjunto de pruebas unitarias, además es una herramienta que es el núcleo que se necesita para poder tener un orden en el momento de crear las pruebas.
- Las “suites” se crean con la función **describe**, que es una función global y con la cual se inicia toda prueba unitaria, además consta con dos parámetros:

```
describe("Una suite es sólo una función", function() {
//…
});
```

- El primer parámetro es una cadena de caracteres donde se define el nombre descriptivo de la prueba unitaria.
- El segundo parámetro es una función con el código que ejecutará la prueba de código.
- Se pueden anidar “suites” para estructurar conjuntos complejos y facilitar la legibilidad y la búsqueda, basta con crear un describe dentro de otro.

259

Especificaciones

- Una especificación contiene una o más expectativas (algo que se espera) que ponen a prueba el estado del código. Una expectativa de Jasmine es una afirmación que debe ser verdadera pero puede ser falsa.
- Una especificación con todas las expectativas verdaderas es una especificación que pasa la prueba, pero con una o más falsas es una especificación que falla.
- Las especificaciones se definen dentro de una Suite llamando a la función global del Jasmine **it** que, al igual que describe, recibe una cadena y una función. La cadena es el título de la especificación y la función es la especificación o prueba.
  - it("y así es una especificación", function() {
  - //…
  - });
- **describe** y **it** son funciones: pueden contener cualquier código ejecutable necesario para implementar la prueba y las reglas de alcance de JavaScript se aplican, por lo que las variables declaradas en un describe están disponibles para cualquier bloque it dentro de la suite.

260

Expectativas

- Las expectativas se construyen con la función expect que obtiene un valor real de una expresión y lo comparan mediante una función Matcher con un el valor esperado (constante).

  expect(valor actual).matchers(valor esperado);
- Los matchers son funciones que implementan comparaciones booleanas entre el valor actual y el esperado, ellos son los responsables de informar a Jasmine si la expectativa se cumple o es falsa.
- Cualquier matcher puede evaluarse como una afirmación negativa mediante el encadenamiento a la llamada expect de un not antes de llamar al matcher.

  expect(valor actual)**.not**.matchers(valor esperado);
- También existe la posibilidad de escribir matchers personalizados para cuando el dominio de un proyecto consiste en afirmaciones específicas que no se incluyen en los ya definidos.

261

Matchers

- .toEqual(y); verifica si ambos valores son iguales ==.
- .toBe(y); verifica si ambos objetos son idénticos ===.
- .toMatch(pattern); verifica si el valor pertenece al patrón establecido.
- .toBeDefined(); verifica si el valor está definido.
- .toBeUndefined(); verifica si el valor es indefinido.
- .toBeNull(); verifica si el valor es nulo.
- .toBeNaN(); verifica si el valor es NaN.
- .toBeCloseTo (n, d); verifica la precisión matemática (número de decimales).
- .toContain(y); verifica si el valor actual contiene el esperado.

262

Matchers

- .toBeTruthy(); verifica si el valor es verdadero.
- .toBeFalsy(); verifica si el valor es falso.
- .toBeLessThan(y); verifica si el valor actual es menor que el esperado.
- .toBeLessThanOrEqual (y); verifica si el valor actual es menor o igual que el esperado.
- .toBeGreaterThan(y); verifica si el valor actual es mayor que el esperado.
- .toBeGreaterThanOrEqual (y); verifica si el valor actual es mayor o igual que el esperado.
- .toThrow(); verifica si una función lanza una excepción.
- .toThrowError(e); verifica si una función lanza una excepción específica.

263

Forzar fallos

- La función fail(msg) hace que una especificación falle. Puede llevar un mensaje de fallo o error de un objeto como un parámetro.

```
describe("Una especificación utilizando la función a prueba", function() {
  var foo = function(x, callBack) {
    if (x) {
      callBack();
    }
  };
  it("no debe llamar a la devolución de llamada", function() {
    foo(false, function() {
      fail("Devolución de llamada ha sido llamada");
    });
  });
});
```


264

Montaje y desmontaje

- Para montar el escenario de pruebas suele ser necesario definir e inicializar un conjunto de variables. Para evitar la duplicidad de código y mantener las variables inicializadas en un solo lugar además de mantener la modularidad, Jasmine suministra las funciones globales :
  - beforeAll(fn) se ejecuta solo una vez antes empezar a ejecutar las especificaciones del "describe".
  - beforeEach(fn) se ejecuta antes de cada especificación dentro del "describe".
  - afterEach(fn) se ejecuta después de cada especificación dentro del "describe".
  - afterAll(fn) se ejecuta solo una vez después de ejecutar todas las especificaciones del "describe".

```
describe("operaciones aritméticas", function(){
    var cal;
    beforeEach(function(){ calc = new Calculadora(); });
    it("adicion", function(){ expect(calc.suma(4)).toEqual(4); });
    it("multiplicacion", function(){ expect(calc.multiplica(7)).toEqual(0); });
    // …
```

- Otra manera de compartir las variables entre una beforeEach, it y afterEach es a través de la palabra clave this. Cada expectativa beforeEach/it/afterEach tiene el mismo objeto vacío this que se restablece de nuevo a vacío para de la siguiente expectativa beforeEach/it/afterEach.

Desactivación parcial

- Las Suites se pueden desactivar renombrando la función describe por xdescribe. Estas suites y las especificaciones dentro de ellas se omiten cuando se ejecuta y por lo tanto sus resultados no aparecerán entre los resultados de la prueba.
- De igual forma, las especificaciones se desactivan renombrando it por xit, pero en este caso aparecen en los resultados como pendientes (pending).
- Cualquier especificación declarada sin un cuerpo función también estará marcada pendiente en los resultados.

```
it('puede ser declarada con "it", pero sin una función');
```

- Y si se llama a la función de pending en cualquier parte del cuerpo de las especificaciones, independientemente de las expectativas, la especificación quedará marcada como pendiente. La cadena que se pasa a pending será tratada como una razón y aparece cuando termine la suite.

```
it('se puede llamar a "pending"en el cuerpo de las especificaciones', function() {
    expect(true).toBe(false);
    pending('esto es por lo que está pendiente');
});
```

Ejecución de pruebas específicas

- En determinados casos (desarrollo) interesa limitar las pruebas que se ejecutan. Si se pone el foco en determinadas suites o especificaciones solo se ejecutaran las pruebas que tengan el foco, marcando el resto como pendientes.
- Las Suites se enfocan renombrando la función describe por fdescribe. Estas suites y las especificaciones dentro de ellas son las que se ejecutan.
- De igual forma, las especificaciones se enfocan renombrando it por fit.
- Si se enfoca una suite que no tiene enfocada ninguna especificación se ejecutan todas sus especificaciones, pero si tiene alguna enfocada solo se ejecutaran las que tengan el foco.
- Si se enfoca una especificación se ejecutara independientemente de que su suite esté o no enfocada.
- La funciones de montaje y desmontaje se ejecutaran si la suite tiene alguna especificación con foco.
- Si ninguna suite o especificación tiene el foco se ejecutaran todas las pruebas normalmente.

267

Espías

- Jasmine tiene funciones dobles de prueba llamados espías.
- Un espía puede interceptar cualquier función y hacer un seguimiento a las llamadas y todos los argumentos.

```
beforeEach(function() {
  fnc = spyOn(calc, 'suma');
  prop = spyOnProperty(calc, 'pantalla', 'set')
});
```

- Un espía sólo existe en el bloque describe o it en que se define, y se eliminará después de cada especificación.
- Hay comparadores (matchers) especiales para interactuar con los espías.
  - .toHaveBeenCalled() pasará si el espía fue llamado.
  - .toHaveBeenCalledTimes(n) pasará si el espía se llama el número de veces especificado.
  - .toHaveBeenCalledWith(…) pasará si la lista de argumentos coincide con alguna de las llamadas grabadas a la espía.
  - .toHaveBeenCalledBefore(esperado): pasará si el espía se llama antes que el espía pasado por parámetro.

268

Seguimiento de llamadas

- El proxi del espía añade la propiedad calls que permite:
  - all(): Obtener la matriz de llamadas sin procesar para este espía.
  - allArgs(): Obtener todos los argumentos para cada invocación de este espía en el orden en que fueron recibidos.
  - any(): Comprobar si se ha invocado este espía.
  - argsFor(índice): Obtener los argumentos que se pasaron a una invocación específica de este espía.
  - count(): Obtener el número de invocaciones de este espía.
  - first(): Obtener la primera invocación de este espía.
  - mostRecent(): Obtener la invocación más reciente de este espía.
  - reset(): Restablecer el espía como si nunca se hubiera llamado.
  - saveArgumentsByValue(): Establecer que se haga un clon superficial de argumentos pasados a cada invocación.

```
spyOn(foo, 'setBar');
expect(foo.setBar.calls.any()).toEqual(false);
foo.setBar();
expect(foo.setBar.calls.count()).toBy(1);
```


269

Cambiar comportamiento

- Adicionalmente el proxi del espía puede añadir los siguientes comportamiento:
  - callFake(fn): Llamar a una implementación falsa cuando se invoca.
  - callThrough(): Llamar a la implementación real cuando se invoca.
  - exec(): Ejecutar la estrategia de espionaje actual.
  - identity(): Devolver la información de identificación para el espía.
  - returnValue(valor): Devolver un valor cuando se invoca.
  - returnValues(... values): Devolver uno de los valores especificados (secuencialmente) cada vez que se invoca el espía.
  - stub(): No haga nada cuando se invoca. Este es el valor predeterminado.
  - throwError(algo): Lanzar un error cuando se invoca.

```
spyOn(foo, "getBar").and.returnValue(745);
spyOn(foo, "getBar").and.callFake(function(arguments, can, be, received) {
  return 745;
});
spyOn(foo, "forbidden").and.throwError("quux");
```


270

Karma

- Karma es una herramienta de línea de comandos JavaScript que se puede utilizar para generar un servidor web que carga el código fuente de la aplicación y ejecuta sus pruebas.
- Puede configurar Karma para funcionar contra una serie de navegadores, que es útil para estar seguro de que la aplicación funciona en todos los navegadores que necesita soportar.
- Karma se ejecuta en la línea de comandos y mostrará los resultados de sus pruebas en la línea de comandos una vez que se ejecute en el navegador.
- Karma es una aplicación NodeJS, y debe ser instalado a través de npm:
  - npm install karma karma-jasmine jasmine-core karma-chrome-launcher --save-dev
  - npm install -g karma-cli

Cobertura de código

- Con Angular CLI podemos ejecutar pruebas unitarias y crear informes de cobertura de código. Los informes de cobertura de código nos permiten ver que parte del código ha sido o no probada adecuadamente por nuestras pruebas unitarias.
- Para generar el informe de cobertura:

  ng test --watch=false --code-coverage
- Una vez que las pruebas se completen, aparecerá una nueva carpeta /coverage en el proyecto. Si se abre el archivo index.html en el navegador se debería ver un informe con el código fuente y los valores de cobertura del código.
- Usando los porcentajes de cobertura del código, podemos establecer la cantidad de código (instrucciones, líneas, caminos, funciones) que debe ser probado. Depende de cada organización determinar la cantidad de código que deben cubrir las pruebas unitarias.

Cobertura de código

- Para establecer un mínimo de 80% de cobertura de código cuando las pruebas unitarias se ejecuten en el proyecto, se debe configurar en karma.conf.js

```
coverageIstanbulReporter: {
  reports: [ 'html', 'lcovonly' ],
  fixWebpackSourcePaths: true,
  thresholds: { statements: 80, lines: 80, branches: 80, functions: 80  }
}
```

- Para generar siempre el informe de cobertura de código (sin usar --code-coverage), se debe configurar en angular.json

```
"test": {
   "options": {
    "codeCoverage": true
   }
  }
```


Depuración de pruebas

1. Seleccionar la ventana del navegador Karma.
2. Hacer clic en el botón DEBUG; se abre una nueva pestaña del navegador que permite volver a ejecutar las pruebas.
3. Abrir “Herramientas de Desarrollo” del navegador.
4. Seleccionar la sección de código fuentes.
5. Abrir el archivo con el código de prueba.
6. Establecer un punto de interrupción en la prueba.
7. Actualizar el navegador, que se detiene en el punto de interrupción.

Pruebas Angular

- Angular, por defecto, se encuentra alineado con la calidad de software.
- Cuando Angular-CLI crea un nuevo proyecto:
  - Descarga TSLint, Jazmine, Karma y Protractor (e2e)
  - Configura el entorno de pruebas
  - Habilita un servidor Karma de pruebas continuas (puerto: 9876)
- Así mismo, cuando genera un nuevo elemento, crea el correspondiente fichero de pruebas, usando .spec como sub extensión.
- Para ejecutar el servidor de pruebas:
  - **ng test** *--code-coverage (alias: -cc)*
  - ng test --single-run (alias: -sr)
- No se debe cerrar la instancia de Chrome mientras duren las pruebas.
- Angular suministra una serie de clases, funciones, mock y módulos específicos para las pruebas, comúnmente denominadas Utilidades Angular para pruebas.
- Permite la creación tanto de pruebas unitarias aisladas como casos de prueba que interactúan dentro del entorno Angular.

278

Pruebas Unitarias Aisladas

- Las Pruebas Unitarias Aisladas examinan una instancia de una clase por sí misma sin ninguna dependencia Angular o de los valores inyectados.
- Se crea una instancia de prueba de la clase con new, se le suministran los parámetros al constructor y se prueba la superficie de la instancia.
- Se pueden escribir pruebas unitarias aisladas para pipes y servicios.
- Aunque también se puede probar los componentes de forma aislada, las pruebas aisladas no revelan cómo interactúan entre si los elementos Angular. En particular, no pueden revelar cómo una clase de componente interactúa con su propia plantilla o con otros componentes.

279

Pruebas Unitarias Aisladas

- De servicios sin dependencias
  - let srv = new MyService();
- De servicios con dependencias
  - let srv = new MyService(new OtherService());
- De Pipes
  - let pipe = new MyPipe();
  - expect(pipe.transform('abc')).toBe('Abc');
- De la clase del componente:
  - let comp = new MyComponent();
- De la clase del componente con dependencias :
  - let comp = new MyComponent(new MyService());

Utilidades Angular para pruebas

- Para realizar pruebas dentro del contexto de Angular, las Utilidades Angular para pruebas cuentan con las clases TestBed, ComponentFixture, DebugElement y By, así como varias funciones de ayuda para sincronizar, inyectar, temporizar, …
- Para probar los componentes, lo mas correcto es crear dos juegos de pruebas, a menudo en el mismo archivo de especificaciones.
  - Un primer conjunto de pruebas aisladas que examinan la corrección de la clase del componente.
  - Un segundo conjunto de pruebas que examina como se comporta el componente dentro del Angular, como interactúa con las plantillas, si actualiza el DOM y colabora con el resto de la aplicación.

TestBed

- TestBed representa un módulo Angular para la prueba, proporciona el medio ambiente del módulo para la clase que desea probar. Extrae el componente a probar desde su propio módulo de aplicación y lo vuelve a conectar al módulo de prueba construido a medida, de forma dinámica, específicamente para una serie de pruebas.
- El método configureTestingModule reemplaza a la anotación @NgModule en la declaración del módulo. Recibe un objeto @NgModule que puede tener la mayoría de las propiedades de metadatos de un módulo normal de Angular.
- El estado base incluye la configuración del módulo de prueba predeterminado con las declaraciones (componentes, directivas y pipes) y los proveedores (servicios inyectables) necesarios para el entorno de prueba.
- El método configureTestingModule se suele invocar dentro de un método beforeEach de modo que TestBed pueda restablecer el estado base antes de cada ejecuciones de pruebas.

Preparación de la prueba

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { DebugElement } from '@angular/core';
import { By } from '@angular/platform-browser';

import { MyComponent } from './my.component';

describe('Prueba de MyComponent', () => {
  let fixture: ComponentFixture<MyComponent>;
  let comp:    MyComponent;
  let de:      DebugElement;
  let tag:     HTMLElement;

  beforeEach(() => {
    TestBed.configureTestingModule({ declarations: [ MyComponent ],  });
    // …
   }
```

Pruebas poco profundas

- En TestBed.configureTestingModule hay que declarar todas las dependencias del componente a probar: otros componentes, pipes y directivas propias, proveedores de los servicios utilizados, incluso los módulos importados.

```
TestBed.configureTestingModule({
  imports: [MyCoreModule],
  declarations: [MyComponent, OtherComponent, MyPipe, MyDirective, ...],
})
```

- Agregando NO_ERRORS_SCHEMA (en @angular/core) a los metadatos del esquema del módulo de prueba se indica al compilador que ignore los elementos y atributos no reconocidos. Ya no es necesario declarar los elementos de plantilla irrelevantes.

```
TestBed.configureTestingModule({
  declarations: [ MyComponent ],
  schemas:      [ NO_ERRORS_SCHEMA ]
});
```


284

ComponentFixture y DebugElement

- Un ComponentFixture es un contexto (fixture) que envuelve el componente creado en el entorno de prueba.

```
fixture = TestBed.createComponent(MyComponent);
```

- El fixture proporciona acceso a si mismo, a la instancia del componente y al envoltorio del elemento DOM del componente.

```
comp = fixture.componentInstance; // instancia del componente
```

- El DebugElement es un envoltorio del elemento DOM del componente o de los elementos localizados.

```
de = fixture.debugElement;
tag = de.nativeElement;
```

***NOTA**: Una vez ejecutado el método createComponent se cierra la configuración del TestBed, si se intenta cambiar la configuración dará un error.*

285

Consultar DebugElement

- La clase By es una utilidad Angular de pruebas para consultar el árbol del DOM. El método estático By.css utiliza un selector CSS estándar para generar un predicado (función que devuelve un valor booleano) que filtra de la misma manera que un selector de jQuery.
- Un predicado de consulta recibe un DebugElement y devuelve true si el elemento cumple con los criterios de selección.
- La clase DebugElement dispone de métodos para, utilizando una función de predicado, buscar en todo el árbol DOM del fixture:
  - El método query devuelve el primer elemento que satisface el predicado.
  - El método queryAll devuelve una matriz de todos los DebugElement que satisfacen el predicado.
- La propiedad nativeElement de DebugElement obtiene el elemento DOM.

```
let tag: HTMLElement = fixture.debugElement.query(By.css('#myId')).
    nativeElement;
```


286

Consultas By

- Recuperar un componente con elemento HTML:

```
tag  = fixture.debugElement.query(By.css('my-component'));
```

- Recuperar el valor de la primera etiqueta:

```
it('should render title in a h1 tag', async(() => {
  fixture.detectChanges();
  const tag = fixture.debugElement.query(By.css('h1'));
  expect(tag.nativeElement.textContent).toContain('Welcome to app!!');
 }));
```

- Recuperar los elementos de un listado:

```
it('renders the list on the screen', () => {
  fixture.detectChanges();
  const li = fixture.debugElement.queryAll(By.css('li'));
  expect(li.length).toBe(2);
 });
```


287

triggerEventHandler

- El método DebugElement.triggerEventHandler permite simular que el elemento ha lanzado un determinado evento.

```
fixture.debugElement.triggerEventHandler('click', eventData);
```

- Según sea el evento, eventData representa el objeto event del DOM o el valor emitido. En algunos casos el eventData es obligatorio y con una determinada estructura.
- Dado que los eventos están vinculados a comandos, suele ser preferible invocar directamente a los métodos comando de la clase componente.
- Puede ser necesario para probar la interacción de determinadas directivas con el elemento y para probar las vinculaciones @HostListener.

Detección de cambios

- La prueba puede decir a Angular cuándo realizar la detección de cambios, lo que provoca el enlace de datos y la propagación de las propiedades al elemento DOM.

```
fixture.detectChanges()
```

- Cuando se desea que los cambios se propaguen automáticamente sin necesidad de invocar fixture.detectChanges():

```
import { ComponentFixtureAutoDetect } from '@angular/core/testing';
// …
TestBed.configureTestingModule({
  declarations: [ MyComponent ],
  providers: [
    { provide: ComponentFixtureAutoDetect, useValue: true }
  ]
})
```

- El servicio ComponentFixtureAutoDetect responde a las actividades asíncronas como la resolución de la promesa, temporizadores y eventos DOM. Pero una actualización directa, síncrona de una propiedad de componente es invisible. La prueba debe llamar fixture.detectChanges() manualmente para desencadenar otro ciclo de detección de cambios.

Inyección de dependencias

```
TestBed.configureTestingModule({
    declarations: [ MyComponent ],
    providers:    [ MyService ]
  });
```

- Un componente bajo prueba no tiene por que ser inyectado con servicios reales, por lo general es mejor si son dobles de pruebas (stubs, fakes, spies o mocks), dado que el propósito de la especificación es probar el componente y no el servicio o servicios reales que pueden ser el origen del error.

```
providers:    [ {provide: MyService, useValue: MyServiceFake },
        { provide: Router, useClass: RouterStub} ]
```

- Si la pruebas necesita tener acceso al servicio inyectado, la forma más segura es obtener el servicio desde el fixture.

```
srv = fixture.debugElement.injector.get(MyService);
```

- También se puede obtener el servicio desde TestBed:

```
srv = TestBed.get(MyService);
```


inject

- La función inject es una de las utilidades Angular de prueba.
- Inyecta servicios en la función especificación donde se los puede alterar, espiar y manipular.
- La función inject tiene dos parámetros:
  - Una matriz de tokens de inyección de dependencias Angular.
  - Una función de prueba cuyos parámetros corresponden exactamente a cada elemento de la matriz de tokens de inyección.

```
it('demo inject', inject([Router], (router: Router) => {
  // ...
}));
```

- La función inject utiliza el inyector del módulo TestBed actual y sólo puede devolver los servicios proporcionados a ese nivel. No devuelve los servicios de los proveedores de componentes.

***NOTA**: Una vez ejecutado el método createComponent se cierra la configuración del TestBed, si se intenta cambiar la configuración dará un error.*

Sobre escritura de @Component

- En algunos casos, sobre todo en la inyección de dependencias a nivel de componentes es necesario sobreescibir la definición del componente.
- La estructura MetadataOverride establece las propiedades de @Component a añadir, modificar o borrar:

  type MetadataOverride = { add?: T; remove?: T; set?: T; };
- Donde T son las propiedades de la anotación:

  selector?: string;

  template?: string; ó templateUrl?: string;

  providers?: any[];

  …

292

Sobre escritura de @Component

- El método overrideComponent recibe el componente a modificar y los metadatos con las modificaciones:

```
TestBed.configureTestingModule({
     declarations: [ MyComponent, MyChildComponent],
     providers:    [ { provide: Router, useClass: RouterStub} ]
  })
  .overrideComponent(MyChildComponent, {
    set: {
      providers: [
        { provide: MyService, useClass: MyServiceSpy }
      ]
    }
  });
```


293

Creación asíncrona

- Si el componente tiene archivos externos de plantillas y CSS, que se especifican en las propiedades templateUrl y styleUrls, supone un problema para las pruebas.
- El método TestBed.createComponent es síncrono.
- Sin embargo, el compilador de plantillas Angular debe leer los archivos externos desde el sistema de archivos antes de que pueda crear una instancia de componente. Eso es una actividad asincrónica.
- El método compileComponents devuelve una promesa para que se puedan realizar tareas adicionales inmediatamente después de que termine.
- La función async es una de las utilidades Angular de prueba que esconde la mecánica de ejecución asíncrona, envuelve una función de especificación en una zona de prueba asincrónica, la prueba se completará automáticamente cuando se finalicen todas las llamadas asíncronas dentro de esta zona.

294

Preparación de la prueba asíncrona

```
import { TestBed, async } from '@angular/core/testing';
// …
describe('AppComponent', () => {
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ AppComponent  ],
    }).compileComponents();
  }));

  it('should create the app', async(() => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.debugElement.componentInstance;
    expect(app).toBeTruthy();
  }));
```


295

Inyección de servicios asíncronos

- Muchos servicios obtienen los valores de forma asíncrona. La mayoría de los servicios de datos hacen una petición HTTP a un servidor remoto y la respuesta es necesariamente asíncrona.
- Salvo cuando se estén probando los servicios asíncronos, las pruebas no deben hacer llamadas a servidores remotos. Las pruebas deberían emular este tipo de llamadas.
- Disponemos de las siguientes técnicas:
  - Sustituir el servicio asíncrono por un servicio síncrono.
  - Sustituir el método asíncrono por un espía.
  - Crear una zona de pruebas asíncrona.
  - Crear una falsa zona de pruebas asíncrona.

Espías

- Los espías de Jasmine permiten interceptar y sustituir métodos de los objetos.
- Mediante el espía se sustituye el método asíncrono de tal manera que cualquier llamada al mismo recibe una promesa resuelta de inmediato con un valor de prueba (stub).
- El espía no invoca el método real, por lo tanto no entra en contacto con el servidor.
- En lugar de crear un objeto de servicio sustituto, se inyecta el verdadero servicio y se sustituye el método crítico con un espía Jasmine.

Espías

```
beforeEach(() => {
  TestBed.configureTestingModule({
    declarations: [ MyComponent ],
    providers:    [ MyService ],
  });
  fixture = TestBed.createComponent(MyComponent);
  comp    = fixture.componentInstance;
  srv = fixture.debugElement.injector.get(MyService);
  spy = spyOn(srv, 'myAsyncMethod')
      .and.returnValue(Promise.resolve('result value'));
  // ...
});
it('Prueba asíncrona con espia', () => {
  // ...
  fixture.detectChanges();
  expect(...).matcher(...);
  expect(spy.calls.any()).toBe(true, 'myAsyncMethod called');
  expect(spy.calls.count()).toBe(1, 'stubbed method was called once');
  expect(myService.myAsyncMethod).toHaveBeenCalled();
});
```


298

whenStable

- En algunos casos, la prueba debe esperar a que la promesa se resuelva en el siguiente ciclo de ejecución del motor de JavaScript.
- En este escenario la prueba no se tiene acceso directo a la promesa devuelta por la llamada del método asíncrono dado que está encapsulada en el interior del componente, inaccesibles desde la superficie expuesta.
- Afortunadamente, la función async genera una zona de prueba asincróna, que intercepta todas las promesas emitidas dentro de la llamada al método asíncrono sin importar dónde se producen.
- El método ComponentFixture.whenStable devuelve su propia promesa que se resuelve cuando todas las actividades asíncronas pendientes dentro de la prueba se han completado (cuando sea estable).

```
it('Prueba asíncrona cuando sea estable', async(() => {
  fixture.detectChanges();
  fixture.whenStable().then(() => {
    fixture.detectChanges();
    expect(...).matcher(...);
  });
}));
```


299

fakeAsync

- La función fakeAsync, otra de las utilidades Angular de prueba, es una alternativa a la función async. La función fakeAsync permite un estilo de codificación secuencial mediante la ejecución del cuerpo de prueba en una zona especial de ensayo propia de fakeAsync, haciendo que la prueba aparezca como si fuera síncrona.
- Se apoya en la función tick (), que simula el paso del tiempo hasta que todas las actividades asincrónas pendientes concluyen, similar al wait en concurrencia. Sólo se puede invocar dentro del cuerpo de fakeAsync, no devuelve nada, no hay promesa que esperar.

```
it('Prueba asíncrona cuando con fakeAsync', fakeAsync(() => {
  fixture.detectChanges();
  tick();
  fixture.detectChanges();
  expect(...).matcher(...);
 }));
```


300

Componentes contenidos

- Para probar los componentes simulando que están contenidos en una plantilla es necesario crear un componente Angular para la prueba (wrapper):

```
@Component({
  template: `<my-component [myInput]="MyInput"
      (myOutput)="onMyOutput($event)"></my-component>`
})
class TestHostComponent {
  @ViewChild(MyComponent) myComponent: MyComponent;
  MyInput: any = null;
  MyOutput: any;
  onMyOutput(rslt: any) { this.MyOutput = rslt; }
}
```


301

Componentes contenidos

- Para posteriormente instanciarlo:

```
beforeEach( async(() => {
  TestBed.configureTestingModule({
    declarations: [ MyComponent, TestHostComponent ],
  }).compileComponents();
}));

beforeEach(() => {
  // create TestHostComponent instead of MyComponent
  fixture  = TestBed.createComponent(TestHostComponent);
  testHost = fixture.componentInstance;
  tag  = fixture.debugElement.query(By.css('my-component'));
  fixture.detectChanges(); // trigger initial data binding
});
```


Entradas y Salidas

- Entrada: Se modifican las entradas a través del componente de pruebas y se comprueba que las modificaciones se reflejan en el componente contenido.

```
it('input test', () => {
  testHost.MyInput = '666';
  fixture.detectChanges();
  expect(testHost.myComponent.getInit()).toBe('666');
});
```

- Salida: Se interactúa con el componente contenido para que se disparen los eventos de salida y se comprueba en el componente de pruebas que las modificaciones se han reflejado en el.

```
it('output test', () => {
  testHost.myComponent.exec();
  fixture.detectChanges();
  expect(testHost.MyOutput).toBe('666');
});
```

Pruebas de observables

- Para poder probar los observables es necesario:
  - Crear una zona asíncrona
  - Convertir el observable en una promesa
  - Definir las expectativas dentro del .then de la promesa.

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

it('get http', async(inject([HttpClient], (http: HttpClient) => {
  http.get(url)
    .toPromise().then(
      data => { expect(data).toBeTruthy(); },
      err => { fail(); }
    );
})));
```


Dobles de prueba

- La regla de oro de las pruebas unitarias, es que una unidad (unit) tiene que ser testeada sin utilizar ninguna de sus dependencias.
- Eso quiere decir que si una unidad tiene dependencias hay que reemplazarlas por mocks.
- Un ejemplo de mock de un servicio sería:

```
class MyServiceSpy {
  getData = jasmine.createSpy('getData').and.callFake(() => {
    return of([{ id: 0, name: 'Uno'}, { id: 1, name: 'Dos' }]);
  });
}
```

- Para crear las especificación:

```
it('fetches all data', () => {
  // …
  expect(instance.names.length).toBe(2);
  expect(MyService.getData).toHaveBeenCalled();
});
```

Doble de prueba Observable

```
import { of, Observable } from 'rxjs';

export class DAOServiceMock {
  constructor(private listado: Array<any>) { }
  query(): Observable<any> { return of(this.listado); }
  get(id: number) { return of(this.listado[0]); }
  add(item: any) { return of(item); }
  change(id: number, item: any) { return of(item); }
  remove(id: number) { return of(id); }
}

{provide: MyDAOService, useValue: new DAOServiceMock([{ id: 0,
      name: 'Uno'}, { id: 1, name: 'Dos' }])}
```


306

Prueba de peticiones HTTP

- La biblioteca de pruebas HTTP de Angular está diseñada siguiendo un patrón de pruebas donde la especificación empieza haciendo las solicitudes.
- Después de eso, las pruebas esperan a que ciertas solicitudes hayan sido o no realizadas, se cumplan determinadas afirmaciones contra esas solicitudes y, finalmente, se proporcionan respuestas "descargando" cada solicitud esperada, lo que puede activar más solicitudes nuevas, etc.
- Al terminar, se pueden verificar que la aplicación no ha hecho peticiones inesperadas.
- Para disponer de la biblioteca de pruebas HTTP:

```
imports: [ HttpClientTestingModule, ],
```


307

Prueba de peticiones HTTP

- El módulo instala un mock que sustituye el acceso real al servidor.
- HttpTestingController es el controlador que se inyecta en las pruebas, permite la inspección y el volcado de las solicitudes.

```
it('query', inject([DAOService, HttpTestingController], (dao: DAOService,
httpMock: HttpTestingController) => {
  dao.query().subscribe(
    data => { expect(data.length).toEqual(2); },
    data => { fail(); }
  );
  const req = httpMock.expectOne('http://localhost:4321/data');
  expect(req.request.method).toEqual('GET');
  req.flush([{ name: 'Data' }, { name: 'Data2' }]);
  httpMock.verify();
 }));
```


Enrutado

- Para disponer del enrutador se utilizará el módulo:

```
imports: [ // …
    RouterTestingModule.withRoutes([
      {path: '', component: HomeCmp},
      {path: 'simple', component: SimpleCmp}
    ])]
```

- Para crear y registrar un sustituto del Router:

```
class RouterStub {
  navigateByUrl(url: string) { return url; }
  navigate(commands: Array<any>) { return url; }
}
{ provide: Router, useClass: RouterStub },
```

- Para interceptar las llamadas al Router:

```
it('Demo ROUTER', inject([Router], (router: Router) => { // ...
  const spy = spyOn(router, 'navigateByUrl');
  // ...
  const navArgs = spy.calls.first().args[0];
  expect(navArgs).toBe(...);
}));
```

http://www.protractortest.org/

TEST E2E

Introducción

- A medida que las aplicaciones crecen en tamaño y complejidad, se vuelve imposible depender de pruebas manuales para verificar la corrección de las nuevas características, errores de captura y avisos de regresión.
- Las pruebas unitarias son la primera línea de defensa para la captura de errores, pero a veces las circunstancias requieran la integración entre componentes que no se pueden capturar en una prueba unitarias .
- Las pruebas de extremo a extremo (E2E: end to end) se hacen para encontrar estos problemas.
- El equipo de Angular ha desarrollado Protractor que simula las interacciones del usuario con el interfaz (navegador) y que ayudará a verificar el estado de una aplicación Angular.
- Protractor es una aplicación Node.js para ejecutar pruebas de extremo a extremo que también están escritas en JavaScript y se ejecutan con el propio Node.
- Protractor utiliza WebDriver para controlar los navegadores y simular las acciones del usuario.

Introducción

- Protractor utiliza Jasmine para su sintaxis prueba.
- Al igual que en las pruebas unitarias, el archivo de pruebas se compone de uno o más bloques describe de it que describen los requisitos de su aplicación.
- Las bloques it están hechos de comandos y expectativas .
- Los comandos indican a Protractor que haga algo con la aplicación, como navegar a una página o hacer clic en un botón.
- Las expectativas indican a Protractor afirmaciones sobre algo acerca del estado de la aplicación, tales como el valor de un campo o la URL actual.
- Si alguna expectativa dentro de un bloque it falla, el ejecutor marca en it como "fallido" y continúa con el siguiente bloque.
- Los archivos de prueba también pueden tener bloques beforeEach y afterEach, que se ejecutarán antes o después de cada bloque it, independientemente de si el bloque pasa o falla.

312

Instalación

- Se utiliza npm para instalar globalmente Protractor:
  - npm install -g protractor
- Esto instalará dos herramientas de línea de comandos, protractor y WebDriver-manager, para asegurarse de que está funcionando.
  - protractor –versión
- El WebDriver-Manager es una herramienta de ayuda para obtener fácilmente una instancia de un servidor en ejecución Selenium. Para descargar los binarios necesarios:
  - webdriver-manager update
- Para poner en marcha el servidor:
  - webdriver-manager start
- Las pruebas Protractor enviarán solicitudes a este servidor para controlar un navegador local, el servidor debe estar en funcionamiento durante todo el proceso de pruebas.
- Se puede ver información sobre el estado del servidor en:
  - http://localhost:4444/wd/hub

313

Configuración y Ejecución

- Se configura un fichero con las pruebas a realizar:

```
// Fichero: e2e.conf.js
  exports.config = {
  framework: 'jasmine',
  seleniumAddress: 'http://localhost:4444/wd/hub',
  specs: ['test/*.e2e.js'],
  multiCapabilities: [
     //{ browserName: 'firefox' },
     { browserName: 'chrome' }
   ]
};
```

- Se lanzan las pruebas (con Selenium Server en ejecución):
  - protractor e2e.conf.js

Elementos Globales

- browser: Envoltura alrededor de una instancia de WebDriver, utilizado para la navegación y la información de toda la página.
  - El método browser.get carga una página.
  - Protractor espera que Angular esté presente la página, por lo que generará un error si la página que está intentando cargar no contiene la biblioteca Angular.
- element: Función de ayuda para encontrar e interactuar con los elementos DOM de la página que se está probando.
  - La función element busca un elemento en la página.
  - Se requiere un parámetro: una estrategia de localización del elemento dentro de la página.
- by: Colección de estrategias elemento localizador.
  - Por ejemplo, los elementos pueden ser encontrados por el selector CSS, por el ID, por el atributo ng-model, …
- protractor: Espacio de nombres de Angular que envuelve el espacio de nombres WebDriver.
  - Contiene variables y clases estáticas, tales como protractor.Key que se enumera los códigos de teclas especiales del teclado.

Visión de conjunto

- Protractor exporta la función global element, que con un localizador devolverá un ElementFinder .
- Esta función recupera un solo elemento, si se necesita recuperar varios elementos, la función element.all obtiene la colección de elementos localizados.
- El ElementFinder tiene un conjunto de métodos de acción, tales como click(), getText(), y sendKeys().
- Esta es la forma principal para interactuar con un elemento (etiqueta) de la página y obtener información de respuesta de el.
- Cuando se buscan elementos en Protractor todas las acciones son asíncronas:
  - Por debajo, todas las acciones se envían al navegador mediante el protocolo SON Webdriver Wire Protocol.
  - El navegador realiza la acción tal y como un usuario lo haría de forma nativa o manual.

316

Localizadores

- Un localizador de Protractor dice cómo encontrar un cierto elemento DOM.
- Los localizadores más comunes son:
  - by.css('.myclass')
  - by.id('myid')
  - by.model('name')
  - by.binding('bindingname')
- Los localizadores se pasan a la función element:
  - var tag = element(by.css('some-css'));
  - var arr = element.all(by.css('some-css'));
- Aunque existe una notación abreviada para CSS similar a jQuery:
  - var tag = $('some-css');
- Para encontrar subelementos o listas de subelementos:
  - var uno = element(by.css('some-css')).element(by.tagName('tag-within-css'));
  - var varios = element(by.css('some-css')).all(by.tagName('tag-within-css'));

317

ElementFinder

- La función element() devuelve un objeto ElementFinder.
- El ElementFinder sabe cómo localizar el elemento DOM utilizando el localizador que se pasa como un parámetro, pero en realidad no lo ha hecho todavía.
- No va a ponerse en contacto con el navegador hasta que un método de acción sea llamado.
- ElementFinder permite invocar acciones como si se produjesen directamente en el navegador.
- Dado que todas las acciones son asíncronas, todos los métodos de acción devuelven una promesa.
- Las acciones sucesivas se encolan y se mandan la navegador ordenadamente.
- Para acciones que deban esperar se usan las promesas:

```
element(by.model('nombre')).getText().then(function(text) {
  expect(text).toBe("MUNDO");
 });
```


La prueba

```
describe('Primera prueba con Protractor', function() {
  it('introducir nombre y saludar', function() {
    browser.get('http://localhost:4200/');
    var txt = element(by.model('vm.nombre'));
    txt.clear();
    txt.sendKeys('Mundo');
    browser.sleep(5000);
    element(by.id('btnSaluda')).click();
    expect(element(by.binding('vm.msg')).getText()).
       toEqual('Hola Mundo');
    browser.sleep(5000);
  });
});
```

Selenium

- El Selenium es un conjunto de herramientas para automatizar los navegadores web, robot que simula la interacción del usuario con el navegador, originalmente pensado como entorno de pruebas de software para aplicaciones basadas en la web.
- Como principales herramientas Selenium cuenta con:
  - Selenium IDE: una herramienta para grabar y reproducir secuencias de acciones con el navegador que permite crear pruebas sin usar un lenguaje de scripting para pruebas.
  - Selenium Core: API para escribir pruebas automatizadas y de regresión en un amplio número de lenguajes de programación populares incluyendo Java, C#, Ruby, Groovy, Perl, Php y Python.
  - WebDriver: interfaces que permite ejecutar las pruebas de forma nativa usando la mayoría de los navegadores web modernos en diferentes sistemas operativos como Windows, Linux y OSX.
  - Selenium Grid: Permite ejecutar muchas pruebas de un mismo grupo en paralelo o pruebas en múltiples entornos. Tiene la ventaja que un conjunto de pruebas muy grande puede dividirse en varias maquinas remotas para una ejecución más rápida o si se necesitan repetir las mismas pruebas en múltiples entornos.

320

Selenium IDE

- Es el entorno de desarrollo integrado para pruebas con Selenium que permite grabar, editar y depurar fácilmente las pruebas.
- Solo está disponible como una extensión de Firefox y Chrome.
- Se pueden desarrollar automáticamente scripts al crear una grabación y de esa manera se puede editar manualmente con sentencias y comandos para que la reproducción de nuestra grabación sea correcta
- Los scripts se generan en un lenguaje de scripting especial para Selenium a menudo denominado Selanese.
- Selanese provee comandos que dicen al Selenium que hacer y pueden ser:
  - **Acciones**: son comandos que generalmente manipulan el estado de la aplicación, ejecutan acciones sobre objetos del navegador, como hacer click en un enlace, escribir en cajas de texto o seleccionar de una lista de opciones. Muchas acciones pueden ser llamadas con el sufijo "AndWait" que indica la acción hará que el navegador realice una llamada al servidor y que se debe esperar a una nueva página se cargue.
  - **Descriptores de acceso**: examinan el estado de la página y almacenan los resultados en variables.
  - **Aserciones**: son como descriptores de acceso, pero las muestras confirman que el estado de la solicitud se ajusta a lo que se esperaba, verifican la presencia de un texto en particular o la existencia de elementos.

321

Selenium IDE

- Dispone de una selección inteligente de campos usando ID, nombre, Xpath o DOM según se necesite.
- Para la depuración permite la configuración de los puntos de interrupción, iniciar y detener la ejecución de un caso de prueba desde cualquier punto dentro del caso de prueba e inspeccionar la forma en el caso de prueba se comporta en ese punto.
- ~~Permite exportar los casos de prueba a Java, C# y Ruby, actuando como embriones en la creación de los casos de prueba para WebDriver.~~
- Selenium IDE dispone de un amplio conjunto de extensiones adicionales que ayudan o simplifican la elaboración de los casos de pruebas.

WebDriver

```
@BeforeClass
public static void setUpClass() throws Exception {
        System.setProperty("webdriver.chrome.driver", "C:/Archivos/…/chromedriver.exe");
}
@Before
public void setUp() throws Exception {
  driver = new ChromeDriver();
  baseUrl = "http://localhost/";
  driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
}
@Test
public void testLoginOK() throws Exception {
  driver.get(baseUrl + "/login.php");
  driver.findElement(By.id("login")).sendKeys("admin");
  driver.findElement(By.id("password")).sendKeys("admin");
  driver.findElement(By.cssSelector("input[type=\"submit\"]")).click();
  try {
    assertEquals("", driver.findElement(By.cssSelector("img[title=\"Main Menu\"]")).getText());
  } catch (Error e) {
    verificationErrors.append(e.toString());
  }
}
```

Maven

```
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-java</artifactId>
  <version>3.13.0</version>
</dependency>
```

Ejecutar en línea de comandos

- Requiere tener instalado NodeJS (https://nodejs.org)
- Instalar CLI
  - npm install -g selenium-side-runner
- Instalar los WebDriver:
  - Crear una carpeta y referenciarla en el PATH del sistema.
  - Descargar los drivers (https://www.seleniumhq.org/download/)
  - Copiarlos a la carpeta creada.
- Para ejecutar las suites de pruebas:
  - selenium-side-runner project.side project2.side *.side
- Para ejecutar en diferentes navegadores:
  - selenium-side-runner *.side -c "browserName=Chrome"
  - selenium-side-runner *.side -c "browserName=firefox"

327

Proceso de Prueba

- Descomponer la aplicación web existente para identificar qué probar
- Identificar con qué navegadores probar
- Elige el mejor lenguaje para ti y tu equipo.
- Configura Selenium para que funcione con cada navegador que te interese.
- Escriba pruebas de Selenium mantenibles y reutilizables que serán compatibles y ejecutables en todos los navegadores.
- Cree un circuito de retroalimentación integrado para automatizar las ejecuciones de prueba y encontrar problemas rápidamente.
- Configura tu propia infraestructura o conéctate a un proveedor en la nube.
- Mejora drásticamente los tiempos de prueba con la paralelización
- Mantente actualizado en el mundo Selenium.

328

DESPLIEGUE

Pruebas

- Son imprescindibles en entornos de calidad: permite ejecutar las pruebas unitarias, las pruebas de extremo a extremo o comprobar la sintaxis.
- Para comprobar la sintaxis: Hay que ejecutar el analizador con el comando:
  - ng lint
- Para ejecutar tests unitarios: Se puede lanzar los tests unitarios con karma con el comando:
  - ng test
- Para ejecutar tests e2e: Se puede lanzar los tests end to end con protractor con el comando:
  - ng e2e

Polyfill

- Angular se basa en los últimos estándares de la plataforma web. Dirigirse a una gama tan amplia de navegadores es un reto porque no todos son compatibles con todas las funciones de los navegadores modernos.
- Un Polyfill puede ser un segmento de código o un plugin que permite tener las nuevas funcionalidades u objetos de HTML5 y ES2015 en aquellos navegadores que nativamente no lo soportan.
- Es necesario activar (des-comentar) en el fichero polyfills.ts las funcionalidades utilizadas.

```
/** IE9, IE10 and IE11 requires all of the following polyfills. **/
import 'core-js/es6/symbol';
import 'core-js/es6/object';
// …
import 'core-js/es6/set';
// …
```

- Este archivo incorpora los polyfill obligatorios y muchos opcionales. Algunos polyfill opcionales se tendrá que instalar con npm.

Compilación

- Una aplicación Angular consiste principalmente en componentes y sus plantillas HTML. Debido a que los componentes y las plantillas proporcionadas por Angular no pueden ser entendidas directamente por el navegador, las aplicaciones de Angular requieren un proceso de compilación antes de que puedan ejecutarse en el navegador.
- Angular ofrece dos formas de compilar la aplicación:
  - Just-in-Time (JIT), que compila la aplicación en el navegador en tiempo de ejecución (la predeterminada cuando ejecuta los comandos de CLI ng build o ng serve).
  - Ahead-of-Time (AOT), que compila la aplicación antes de que la descargue el navegador.
- El compilador Ahead-of-Time (AOT) convierte el código TypeScript y el HTML de la aplicación Angular en un eficiente código JavaScript, durante la fase de despliegue antes de que el navegador descargue y ejecute la aplicación. Las ventajas de la compilación AOT son:
  - Representación más rápida: Con AOT, el navegador descarga una versión precompilada de la aplicación. El navegador carga directamente el código ejecutable para que pueda procesar la aplicación de inmediato, sin esperar a compilarla primero.
  - Menos peticiones asíncronas: El compilador incluye las plantillas HTML externas y las hojas de estilo CSS dentro de la aplicación JavaScript, eliminando solicitudes ajax separadas para esos archivos de origen.
  - Tamaño de descarga del framework de Angular más pequeño: No es necesario descargar el compilador Angular si la aplicación ya está compilada. El compilador ocupa aproximadamente la mitad de Angular, por lo que omitirlo reduce drásticamente la carga útil de la aplicación.
  - Detectar antes errores de plantillas: El compilador AOT detecta e informa de los errores de enlace de la plantilla durante el proceso de compilación antes de que los usuarios puedan verlos.
  - Mejor seguridad: AOT compila plantillas y componentes HTML en archivos JavaScript mucho antes de que se sirvan al cliente. Sin plantillas para leer y sin el riesgo de evaluación de HTML o JavaScript del lado del cliente, hay menos oportunidades para ataques de inyección.

Despliegue

- El despliegue más simple posible
  1. Generar la construcción de producción.
  2. Copiar todo dentro de la carpeta de salida (/dist por defecto) a una carpeta en el servidor.
  3. Configurar el servidor para redirigir las solicitudes de archivos faltantes a index.html.
- Construye la aplicación en la carpeta /dist
  - ng build
  - ng build --dev
- Paso a producción, construye optimizándolo todo para producción
  - ng build --prod
  - ng build --prod --env=prod
  - ng build --target=production --environment=prod
- Precompila la aplicación
  - ng build --prod –aot
- Cualquier servidor es candidato para desplegar una aplicación Angular. No se necesita un motor del lado del servidor para componer dinámicamente páginas de aplicaciones porque Angular lo hace en el lado del cliente.
- La aplicaciones Angular Universal y algunas funcionalidades especiales requieren una configuración especial del servidor.

333

Gracias por vuestra participación

¡Seguimos en contacto!

www.iconotc.com

334