Desarrollo Web

Luciano Diamand

Ultima actualización: March 22, 2019.
Actualizaciones del documento y fuentes:
`http://www.diamand.com.ar/training/taller4`
Correcciones, sugerenias, contribuciones y traducciones son bienvenidas!

Derechos de copia

Angular

Luciano Diamand

Introducción a Angular

Angular

- Es un Framework para trabajar del lado del cliente
- Se pueden desarrollar aplicaciones para Escritorio o para Móviles
- También es posible armar Single Page Applications (SPA)
- Usa TypeScript como lenguaje de programación
- La actualización del estado se basa en cambios en los **datos** y no en cambios en el **DOM**
- Utiliza *data binding* para realizar actualizaciones automáticas de la vista siempre que haya cambios en los datos y viceversa
- Es un producto de Google

Angular

- Arquitectura modular
- Código reutilizable mediante componentes
  - nos permite crear nuevos componentes para extender el HTML
- Ruteo incorporado
- Enrutamiento de datos y eventos
- Plantillas
- Directivas
- Tuberías
- Gestión de formularios y validación
- Servicios
- Injección de dependencias
- Servicio HTTP
- Animaciones
- Angular Material
- Pruebas unitarias

Historia

- Versiones:

2010	AngularJS
2016	Angular 2
Dic 2016	Angular 4
Nov 2017	Angular 5
2018	Angular 6
Oct 2018	Angular 7?

- Utilizan versionado semantico

```
6.0.6
Major.Minor.Revision
```

Entorno de desarrollo

- nodejs
- npm
- Angular-CLI
- Visual Studio Code (VS Code)
- Git
- Navegador Web
  - Desarrollo Web
  - Augury

Angular CLI

Luciano Diamand

Introducción

- Angular CLI significa Angular **C**ommand **L**ine **I**interface
- Es una herramienta de línea de comandos que nos permite automatizar el proceso de desarrollo
- Si bien no es necesario utilizar Angular CLI para crear aplicaciones en Angular, es recomendable dado que nos ahorra el tiempo de configurar e instalar las dependencias requeridas, además de estandarizar el formato del proyecto

Introducción cont...

- Con Angular CLI podemos:
  - crear una nueva aplicación en Angular
  - ejecutar un servidor de desarrollo con soporte para la recarga automática para previsualizar la aplicación durante el desarrollo
  - agregar características a una aplicación Angular existente
  - ejecutar las pruebas unitarias de la aplicación
  - ejecutar las pruebas end-to-end (E2E) de la aplicación
  - construir la aplicación para desplegar en producción
- `https://cli.angular.io/`

Instalación de Angular CLI

Instalación

- Angular CLI depende de `node.js`, con lo cual debemos tenerlo instalado antes de comenzar
- `node --version`
- `sudo npm install -g @angular/cli`
- Esto va a instalar el comando `ng` en su sistema
- La opción -**g** significa que Angluar CLI se va a instalar de forma global
- Si desea descargar la última versión de Angular CLI
- `sudo npm install -g @angular/cli@latest`

Comandos disponibles

ng new

- Crea un nuevo proyecto en Angular
- El proyecto es creado por defecto en el directorio actual
- `ng new <nombre-del-proyecto> [opciones]`
- Opciones:
  - `--dry-run` solo muestra los archivos creados y las operaciones realizadas, pero no crea el proyecto (Alias: 'd')
  - `--verbose` muestra informacion adicional (Alias: 'v')
  - `--skip-npm` una vez creado el proyecto, no ejecuta ningún comando `npm`
  - `--skip-git` no crea un repositorio local para el proyecto
  - `--directory` directorio padre donde crear el proyecto

ng doc

- Abre un navegador por defecto con una búsqueda de la palabra clave en la documentación de Angular
- `ng doc <palabra-clave>`

ng generate

- Genera nuevo código dentro del proyecto
- Alias: 'g'
- `ng generate <tipo> [opciones]`
- Tipos válidos:
  - `component <path/to/component-name>` genera un componente
  - `directive <path/to/directive-name>` genera una directiva
  - `class <route/to/route-component>` genera una nueva clase
  - `pipe <path/to/pipe-name>` genera una tubería
  - `service <path/to/service-name>` genera un servicio
- El componente generado posee su propio directorio, a menos que se especifique la opción `--flat`

ng config

- Obtiene/Asigna un valor de la configuración de Angular CLI. El argumento **jsonPath** es una ruta JavaScript válida como `"users[1].userName"`. Si el valor no esta asignado, se mostrara "undefined". Este comando por defecto solo funciona dentro del directorio del proyecto
- `ng config <jsonPath> <value> [options]`
- Opciones:
  - `--global` Obtiene/Asigna el valor en la configuración global (en su directorio home)
- Ejemplo:
  - `ng config projects.prueba.architect`

Verificando la calidad del código

ng lint

- Ejecuta el analizador de código `codelyzer linter` en el proyecto
- `ng lint`
- Utilice el formato de salida con estilo para mostrar los errores de alineación
- `ng lint --format stylish`
- Trate de corregir automáticamente los errores de alineación
- `ng lint --fix`
- Ejemplo de salida:

```
/home/user/example/src/app/event-details/event-details.component.ts:24:4
ERROR: 24:4  semicolon               Missing semicolon
ERROR: 25:1  no-trailing-whitespace  trailing whitespace

Lint errors found in the listed files.


All files pass linting.
```

ng test

- Ejecuta las pruebas unitarias utilizando `karma`
- `ng test [options]`
- Opciones:
  - `--watch` mantiene en ejecución las pruebas. Por defecto true
  - `--browsers`, `--colors`, `--reporters`, `--port`, `--log-level` estos argumentos se pasan directamente a `karma`

ng e2e

- Ejecuta todos las pruebas end-to-end definidas en la aplicación, utilizando protractor
- `ng e2e`

ng version

- Imprime la versión de `angular-cli`, nodejs y del sistema operativo
- `ng version`

```
...
Angular CLI: 6.0.5
Node: 8.12.0
OS: linux x64
Angular: 6.0.3
... animations, common, compiler, compiler-cli, core, forms
... http, language-service, platform-browser
... platform-browser-dynamic, router

Package                                  Version
------------------------------------------------------------
@angular-devkit/architect                0.6.5
@angular-devkit/build-angular            0.6.5
@angular-devkit/build-optimizer          0.6.5
@angular-devkit/core                     0.6.5
@angular-devkit/schematics               0.6.5
@angular/cli                             6.0.5
@ngtools/webpack                         6.0.5
...
```

ng serve

- Este comando levanta un servidor local de desarrollo en el puerto 4200
- Detras de escena lo que sucede es:
  - Angular-CLI carga la configuración desde angular.json
  - Angular-CLI ejecuta Webpack para construir todo el código JavaScript y CSS
  - Angular-CLI inicia **Webpack dev server** para previsualizar el resultado en localhost:4200
- `ng serve [options]`
- Opciones:
  - `--host` podemos asignarle el host
  - `--port` podemos cambiar el puerto por defecto
  - `-o` abre un navegador al iniciar el servidor

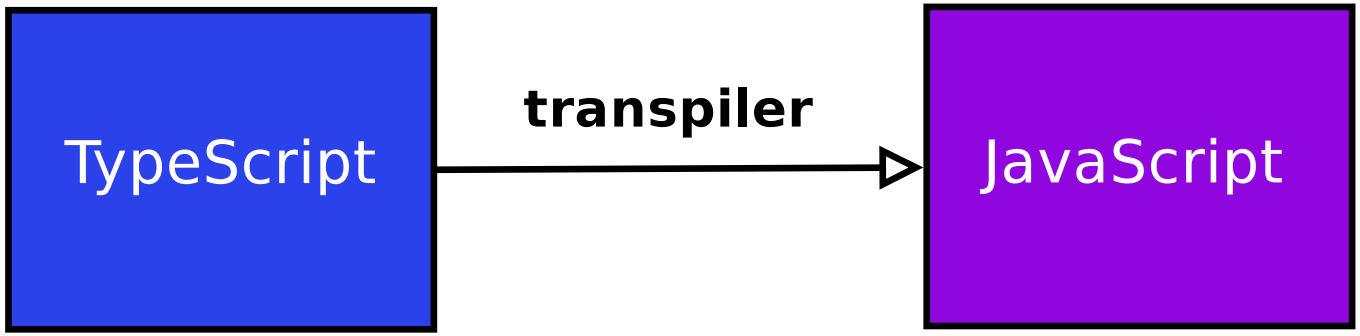
TypeScript

Luciano Diamand

Introducción a TypeScript

Introducción a TypeScript

- TypeScript no es un lenguaje nuevo
- Es una extención del lenguaje ES6 (ECMAScript 6)
- El TypeScript se 'transcompila'
- El **transpiler** toma como entrada el código en *TypeScript* y genera código *JavaScript* como salida el cual es interpretado por todos los navegadores



Características nuevas

- Anotaciones de Tipos
- Clases
- Interfaces
- Funciones flecha
- Decoradores
- Constructores
- Modificadores de acceso
- Importación y exportación de módulos
- Generics
- Utilidades del lenguaje (desestructuración)

Lenguaje TypeScript

- Es un lenguaje Tipado (define el tipo de la variable)
- Es Orientado a Objetos (clases, interfaces, constructores, modificadores de acceso, propiedades, etc.)
- Errores en tiempo de compilación
- Importante Juego de Herramientas de desarrollo
- Los archivos fuente poseen la extension **.ts**
- Para *transpilar* un archivo fuente: `tsc archivo.ts`
- Luego para probar la aplicación: `node archivo.js`

Instalación

- Es posible instalar TypeScript de la siguiente forma:
- `sudo npm install -g typescript`

Pruebas con TypeScript

- Para poder probar los ejemplos podemos utilizar una utilidad llamada **TSUN** (TypeScript Upgraded Node)
- La instalamos de la siguiente manera:
  `sudo npm install -g tsun`
- Ahora podemos ejecutarlo con: `tsun`

Tipos

- El agregado mas importante de TypsScript sobre ES6 es el sistema de tipos
- Lo mejor sobre la verificación de tipos es:
  - Ayuda al escribir código; dado que previene errores en tiempo de compilación
  - Ayuda al leer el código; dado que clarifica las intenciones
- Hay que aclarar que es opcional el usar tipos en TypeScript
- Los tipos básicos de TypeScript son los mismos que usabamos en JavaScript: strings, numbers, booleans, etc.

Definición de variable

- En TypeScript podemos utilizar la palabra reservada `var` para definir una variable (como lo haciamos con JavaScript)
- Pero ahora podemos definirle el tipo:

```
var fullName: string;
```

- Cuando declaramos funciones podemos usar tipos en los argumentos y en los valores de retorno:

```
function greetText(name: string): string {
    return 'Hello ' + name;
}
```

- Los ':' indican que vamos a especificar el tipo de retorno de la función, que en este caso es string
- Intente retornar un entero dentro de la función, que sucede?

Tipos incorporados

- Una cadena almacena texto y se declara usando el tipo *string*
  `var fullName: string = 'James Bond';`
- En TypeScript todos los números son representados como punto flotante. El tipo es *number* `var age: number = 36;`
- El tipo *boolean* almacena ya sea `true` o `false` como valor
  `var married: boolean = true;`
- Los arreglos se declaran de tipo *Array*
- Dado que los arreglos son colecciones, debemos especificar el tipo para los objetos del array
- Podemos especificar el tipo de los items de un arreglo ya sea con `Array<tipo>` o `tipo[]`

```
var jobs: Array<string> = [ 'IBM', 'Microsoft', 'Google' ];
var jobs: string[] = ['Apple', 'Dell', 'HP' ];
```

Tipos incorporados cont...

- Las enumeraciones funcionan nombrando valores numéricos:

```
enum Role { Employee, Manager, Admin };
var role: Role = Role.Employee;
```

- El valor por defecto de inicial para una enum es **0**.
- Es posible cambiar el valor de la siguiente manera:

```
enum Role { Employee = 3, Manager, Admin };
var role: Role = Role.Employee;
```

- El valor de los demas elementos se incrementa a partir de ahí
- También es posible asignar valores independientes:

```
enum Role { Employee = 3, Manager = 5, Admin = 7 };
var role: Role = Role.Employee;
```

- Además, podemos buscar el nombre de una enumeración dada por su valor

```
enum Role { Employee, Manager, Admin };
console.log('Roles:', Role[0], ',', Role[1], 'and', Role[2]);
```

Tipos incorporados cont...

- `any` es el tipo por defecto si omitimos el tipo para una dada variable. Una variable de tipo `any` permite recibir cualquier tipo de valor:

```
var something: any = 'as string';
something = 1;
something = [ 1, 2, 3 ];
```

- Usar `void` implica que no se espera un tipo. Esto se utiliza en general en funciones que no tienen un valor de retorno:

```
function setName(name: string): void {
    this.fullName = name;
}
```

Tipos incorporados cont...

- Una `tupla` permite expresar un arreglo donde el tipo fijo de elementos es conocido, pero no necesariamente el mismo

```
// Declaramos el tipo tupla
let x: [string, number];
// lo inicializamos
x = [ "Hola", 10 ];
// una inicialización incorrecta
x = [ 10, "Hola" ]; // Error
```

- Cuando accedemos a un elemento fuera del juego de índices conocidos, se utiliza una unión:

```
// OK, 'string' puede ser asignado a 'string | number'
x[3] = "mundo";
// OK, 'string' y 'number' ambos tienen 'toString'
console.log(x[5].toString());
// Error, 'boolean' no es de tipo 'string | number'
x[6] = true;
```

Clases

- En JavaScript ES5 la programación orientada a objetos se realizaba utilizando objetos basados en un prototipo. Este modelo no utiliza clases, pero en su lugar se basa en *prototypes*
- Sin embargo, en ES6 se incorporo el soporte para clases en JavaScript
- Para definir una clase usamos la nueva palabra reservada `class` y le damos a nuestra clase un nombre y un cuerpo:

```
class Vehicle {
    // cuerpo
}
```

- las clases pueden tener **propiedades**, **métodos** y **constructores**

Propiedades

- Las propiedades definen los datos asociados a una instancia de la clase
- Cada propiedad en una clase puede tener de forma opcional un tipo
- Ejemplo:

```
class Person {
    first_name: string;
    last_name: string;
    age: number;
}
```

Métodos

- Los métodos son funciones que se ejecutan en el contexto de un objeto. Para invocar un método sobre un objeto, primero necesitamos una instancia de ese objeto
- Para instanciar un objeto utilizamos la palabra reservada `new`
- Ejemplo:

```
class Person {
    first_name: string;
    last_name: string;
    age: number;

    greet() {
        console.log('Hello', this.first_name);
    }
}
```

Métodos cont...

- Dentro de un método es posible acceder al *first_name* de esa persona utilizando la palabra reservada `this`
- Cuando los métodos no declaran un tipo de retorno de forma explicita y devuelven un valor, se asume que pueden retornar cualquier valor (tipo `any`)
- `void` también es un valor válido para `any`
- Ejemplo:

```
// declaramos una variable de tipo Person
var p: Person;

// instanciamos una nueva Person
p = new Person();

// le asignamos un first_name
p.first_name = 'Juan';

// invocamos al metodo greet
p.greet();
```

Métodos cont...

- es posible declarar e instanciar una clase en la misma línea

```
var p: Person = new Person();
```

- Ahora es posible agregar un método a la clase **Person** que retorne un valor. Por ejemplo:

```
class Person {
    ...
    ageInYears(years: number): number {
        return this.age + years;
    }
}
```

Constructores

- Un constructor es un método especial que se ejecuta cuando se crea una nueva instancia de la clase
- Generalmente, el constructor es donde se realizan cualquier tipo de inicialización para los nuevos objetos
- Los constructores se deben llamar `constructor`
- Pueden de forma opcional recibir parametros
- No pueden devolver ningún valor
- En TypeScript la clase puede contener solamente un constructor (esto difiere de ES6 el cual permite tener más de un constructor por clase siempre y cuando tengan diferente número de parámetros)

Constructores cont...

- Cuando una clase no posee un constructor explicitamente definido, se creara uno de forma automática

```
cass Vehicle {
}
var v = new Vehicle();
```

- es lo mismo que:

```
class Vehicle {
    constructor() { }
}
var v = new Vehicle();
```

Constructores cont...

- Un constructor con parámetros sería de la forma:

```
cass Person {
    first_name: string;
    last_name: string;
    age: number;

    constructor(first_name: string, last_name: string, age: number) {
        this.first_name = first_name;
        this.last_name = last_name;
        this.age = age;
    }

    greet() {
        console.log('Hello', this.first_name);
    }

    ageInYears(years: number): number {
        return this.age + years;
    }
}
```

- con lo cual, la creacion de la instancia *inicializada* se reduce a:

```
var p: Person = new Person('Juan', 'Maz', 36);
p.greet();
```

Herencia

- La herencia es una forma de indicar que una clase recibe comportamiento de la clase padre
- Por lo tanto, pueden modificarse, sustituirse o aumentarse aquellos comportamientos (métodos) en la nueva clase
- TypeScript soporta completamente la herencia y se implementa utilizando la palabra reservada `extends`

```
class Report {
    data: Array<string>;

    constructor(data: Array<string>) {
        this.data = data;
    }

    run() {
        this.data.forEach(function(line) { console.log(line); });
    }
}
```

Herencia cont...

- Ahora podemos probar el reporte de la siguiente manera:

```
var r: Report = new Report([ 'First line', 'Second line' ]);
r.run();
```

- Si queremos aplicar herencia sobre la clase `Report`:

```
class TabbedReport extends Report {
    headers: Array<string>;

    constructor(headers: string[], values string[]) {
        super(values);
        this.headers = headers;
    }

    run() {
        console.log(this.headers);
        super.run();
    }
}
```

Herencia cont...

- Ahora podemos probar el nuevo reporte que hereda del anterior:

```
var headers: string[] = [ 'Name' ];
var data: string[] = [ 'Alice Green', 'Paul Pfifer', 'Louis Blakenship' ];
var r: TabbedReport = new TabbedReport(headers, data);
r.run();
```

Utilidades

- ES6 y por extensión TypeScript poseen un número de características sobre sintaxis interesantes a la hora de programar:
- Dos de las mas importantes son:
  - Sintaxis de función de flecha (fat arrow)
  - plantillas de cadena (template strings)

Funciones de flecha

- Las funciones de flecha => son una notación reducida a la hora de escribir funciones
- Ejemplo en ES5:

```
var data = [ 'Alice Green', 'Paul Pfifer', 'Louis Blakenship' ];
data.forEach(function(line) { console.log(line); });
```

- Ahora en TypeScript:

```
var data: string[] = [ 'Alice Green', 'Paul Pfifer', 'Louis Blakenship' ];
data.forEach((line) => console.log(line) );
```

- Los parentesis son opcionales cuando hay un solo parámetro
- La sintaxis de flecha también puede ser utilizada en expresiones:

```
var evens = [ 2, 4, 6, 8 ];
var odds = evens.map(v => v + 1);
```

- o como sentencia:

```
data.forEach( line => {
    console.log(line.toUpperCase());
});
```

Funciones de flecha cont...

- Una caracteristica importante de la sintaxis `=>` es que comparten el mismo `this` con el código que lo envuelve (esto es diferente a como se comportaba en JavaScript)
  - En JavaScript cuando escribimos una función esa función posee su propio `this`

```
var nate = {
    name: "Nate",
    guitars: [ 'Gibson', 'Martin', 'Taylor' ],
    printGuitars: function() {
        var self = this;
        this.guitars.forEach(function(g) {
            // this.name no esta definido, por lo tanto
            // hay que usar self.name
            console.log(self.name + ' plays a ' + g);
        });
    }
};
```

Funciones de flecha cont...

- Dado que la flecha comparte el `this` con el código que encapsula, podemos escribir:

```
var nate = {
    name: 'Nate',
    guitars: [ 'Gibson', 'Martin', 'Taylor' ],
    printGuitars: function() {
        this.guitars.forEach( (g) => {
            console.log(this.name + ' plays a ' + g);
        });
    }
};
```

Plantillas de cadena

- En ES6 se introdujeron las plantillas de cadena. Las características de esta funcionalidad son:
  - Variables dentro de cadenas (sin necesidad de ser concatenadas con $+$) y
  - Cadenas de múltiples líneas

Variables dentro de cadenas

- A esta caracteristica se la llama también "interpolación de cadenas"
- La idea es poder poner variables dentro de las cadenas:

```
var firstName = 'Nate';
var lastName = 'Murray';

// interpolamos una cadena
var greeting = `Hello ${firstName} ${lastName}`;

console.log(greeting);
```

Cadenas de múltiples líneas

- Las cadenas de múltiples líneas son un recurso valioso cuando necesitamos poner cadenas dentro de nuestros templates
- Ejemplo:

```
var template = `
<div>
  <h1>Hello</h1>
  <p>This is a great website</p>
</div>
`
```

Laboratorio práctico - TypeScript Labs

Es momento de jugar con la sintaxis y estructura de TypeScript

- Tipos básicos
- Intefaces
- Funciones
- Tipos
- Clases

Arquitectura de Angular

Luciano Diamand

Vista general de la arquitectura

NgModule

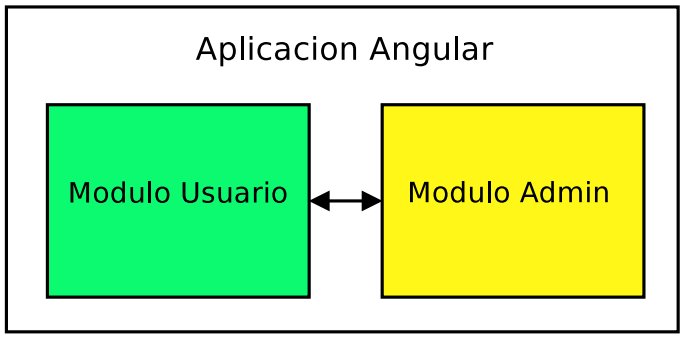
- Los bloques de construcción básicos de una aplicación Angular son los **NgModules**, que proporcionan un contexto de compilación para los componentes
- **NgModules** recopila códigos relacionados en conjuntos funcionales
- Una aplicación Angular se define por un conjunto de **NgModules**
- Una aplicación siempre tiene al menos un módulo raíz que habilita el arranque, y normalmente tiene más módulos con características particulares
- Dichos módulos contienen a los componentes de Angular

NgModule cont...

- Tanto los componentes como los servicios son simples clases, con decoradores que marcan su tipo y proporcionan metadatos que le indican a Angular cómo usarlos
  - Los metadatos en una clase componente la asocian con una plantilla que define la vista. Una plantilla combina HTML ordinario con directivas de Angular y un marcado de enlace que permite a Angular modificar el HTML antes de representarlo
  - Los metadatos para una clase de servicio proporcionan la información que Angular necesita para que esté disponible para los componentes a través de la *inyección de dependencia (DI)*
- Los componentes de una aplicación típicamente definen muchas vistas, ordenadas jerárquicamente. Angular proporciona el servicio de enrutador donde definir rutas de navegación entre vistas. El enrutador proporciona sofisticadas capacidades de navegación en el navegador

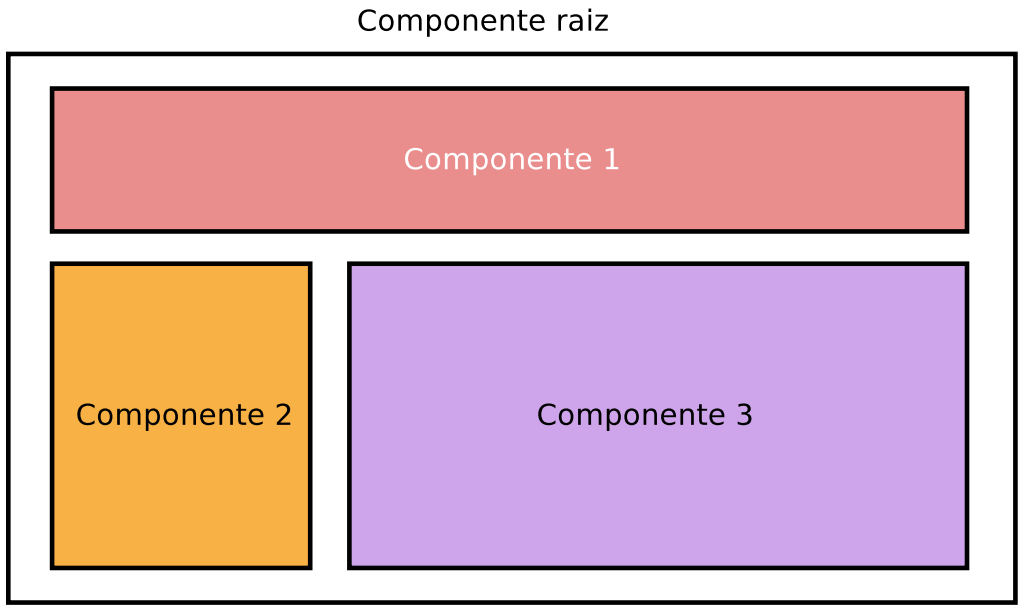
Módulos

- Todas las aplicaciones tienen al menos un módulo que se conoce como módulo **raíz** también llamado **AppModule**. Este módulo provee el mecanismo de arranque que lanza la aplicación
- Cada módulo esta compuesto por **Componentes** y **Servicios** (también tienen otras piezas de Software que después se nombraran)
- En general, un módulo es una característica de la aplicación
- Los módulos pueden importar funcionalidad desde otros **NgModules** y permitir que su propia funcionalidado sea exportada y utilizada por otros **NgModules**



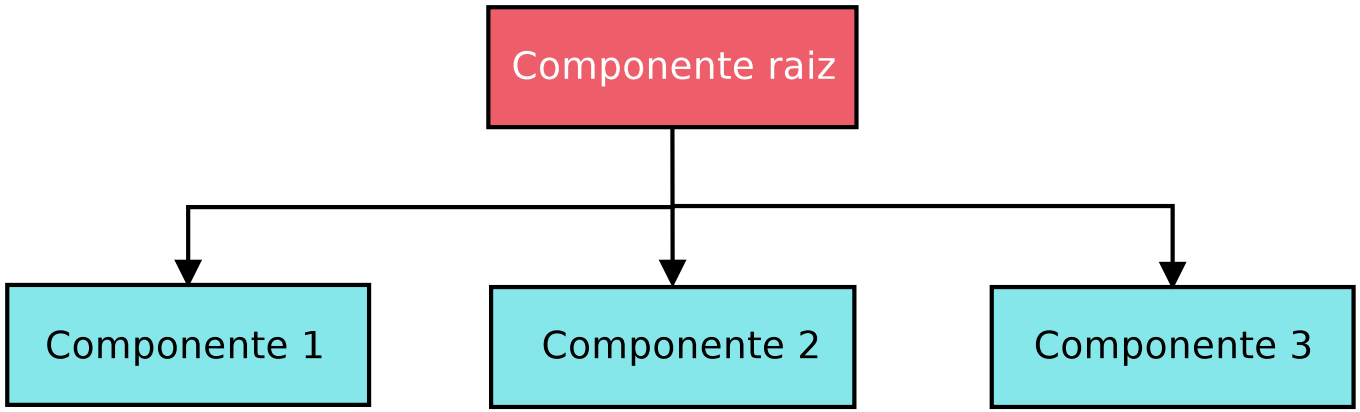
Componentes

- Cada aplicación Angular tiene al menos un componente, el componente **raíz** de la aplicación llamado **AppComponent** que conecta una jerarquía de componentes con el modelo de objeto de documento (DOM) de página
- Cada componente define una clase que contiene datos de aplicación y lógica, y se asocia con una plantilla HTML que define una vista que se mostrará en un entorno destino



Componentes cont...

- Los componentes va a estar anidados dentro del componente **raíz**
- En definitiva, un componente en Angular esta compuesto por:
  - Un plantilla HTML (vista)
  - Una clase en TypeScript (funcionalidad) + Metadatos
  - Un plantilla de estilos CSS
  - Un archivo TypeScript con las pruebas unitarias



La clase de los Componentes

- La clase de los componentes es donde se encuentra la lógica del componente
- El decorador `@Component()` identifica la clase inmediatamente debajo de ella como un componente y proporciona la plantilla y los metadatos específicos del componente relacionado
- A través del decorador también informamos lo siguiente:

selector	es el nombre de la etiqueta que vamos a utilizar para cargar el componente
templateUrl	es el archivo *.html* que contiene la plantilla de la vista
styleUrl	es el archivo *.css* que contienen los estilos CSS

Agregar un nuevo componente

- Para agregar un nuevo componente a la aplicación podemos utilizar Angular CLI:
  `ng generate component <nombre-del-componente>`
- También es posible utilizar la forma abreviada:
  `ng g c <nombre-del-componente>`
- Una vez finalizado la ejecución del comando, tendremos dentro de la carpeta `src/app` un directorio con el nombre `nombre-del-componente`

Agregar un nuevo componente...

- Dicho directorio contendra 4 archivos nuevos generados por `ng`
  - `<nombre-del-componente>.component.ts`
  - `<nombre-del-componente>.component.html`
  - `<nombre-del-componente>.component.spec.ts`
  - `<nombre-del-componente>.component.css`
  - Además se modificaron el archivo `app.module.ts` donde se agregó el `import` de `NombreDelComponente` y también se actualizó el arreglo `declarations` que es donde figuran todos los componentes del módulo raíz

Opciones del decorador `@Component`

- Opciones del selector:
  - Elemento selector:

```
selector: 'app-test'
...
<app-test></app-test>
```

  - Clase selectora:

```
selector: '.app-test'
...
<div class="app-test"></div>
```

  - Atributo selector:

```
selector: '[app-test]'
...
<div app-test></div>
```

Plantillas, directivas y enlace de datos

- Una plantilla combina HTML con el marcado de Angular que puede modificar los elementos HTML antes de que se muestren
- Las directivas de plantilla proporcionan la lógica del programa y el marcado de enlace conecta los datos de la aplicación y el DOM
- Hay dos tipos de enlace de datos:
  - *El enlace de eventos* permite que su aplicación responda a las entradas del usuario en el entorno de destino actualizando los datos de su aplicación
  - *El enlace de propiedades* permite interpolar los valores que se calculan a partir de los datos de su aplicación en el HTML

Plantillas, directivas y enlace de datos

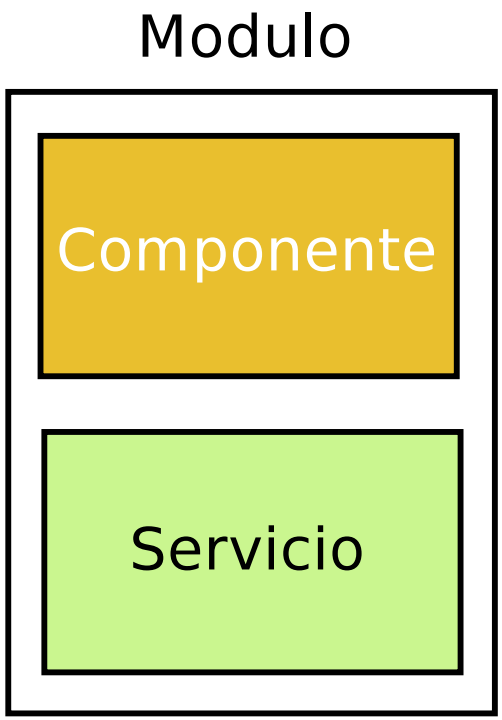
- Antes de que se muestre una vista, Angular evalúa las directivas y resuelve la sintaxis de enlace en la plantilla para modificar los elementos HTML y el DOM, de acuerdo con la lógica y los datos del programa
- Angular admite enlace de datos bidireccional, lo que significa que los cambios en el DOM, como las opciones del usuario, también se reflejan en los datos del programa
- Las plantillas pueden usar tuberías para mejorar la experiencia del usuario al transformar los valores para su visualización
- Por ejemplo, puede usar tuberías para mostrar fechas y valores de moneda que sean apropiados para la configuración regional de un usuario
- Angular proporciona tuberías predefinidas para transformaciones comunes, y también puede definir sus propias tuberías

Servicios

- Para los datos o la lógica que no están asociados con una vista específica y que se desea compartir entre componentes, podemos crear una clase de servicio
- La definición de la clase de servicio está precedida inmediatamente por el decorador `@Injectable()`
- Este decorador proporciona los metadatos que permiten que su servicio se inyecte en los componentes del cliente como una dependencia

Servicios cont...

- La inyección de dependencia (DI) permite mantener las clases de los componentes eficientes y con pocas dependencias
- Los componentes no recuperan datos del servidor, no validan las entradas del usuario ni registran directamente en la consola; delegan tales tareas a los servicios



Entrutamiento

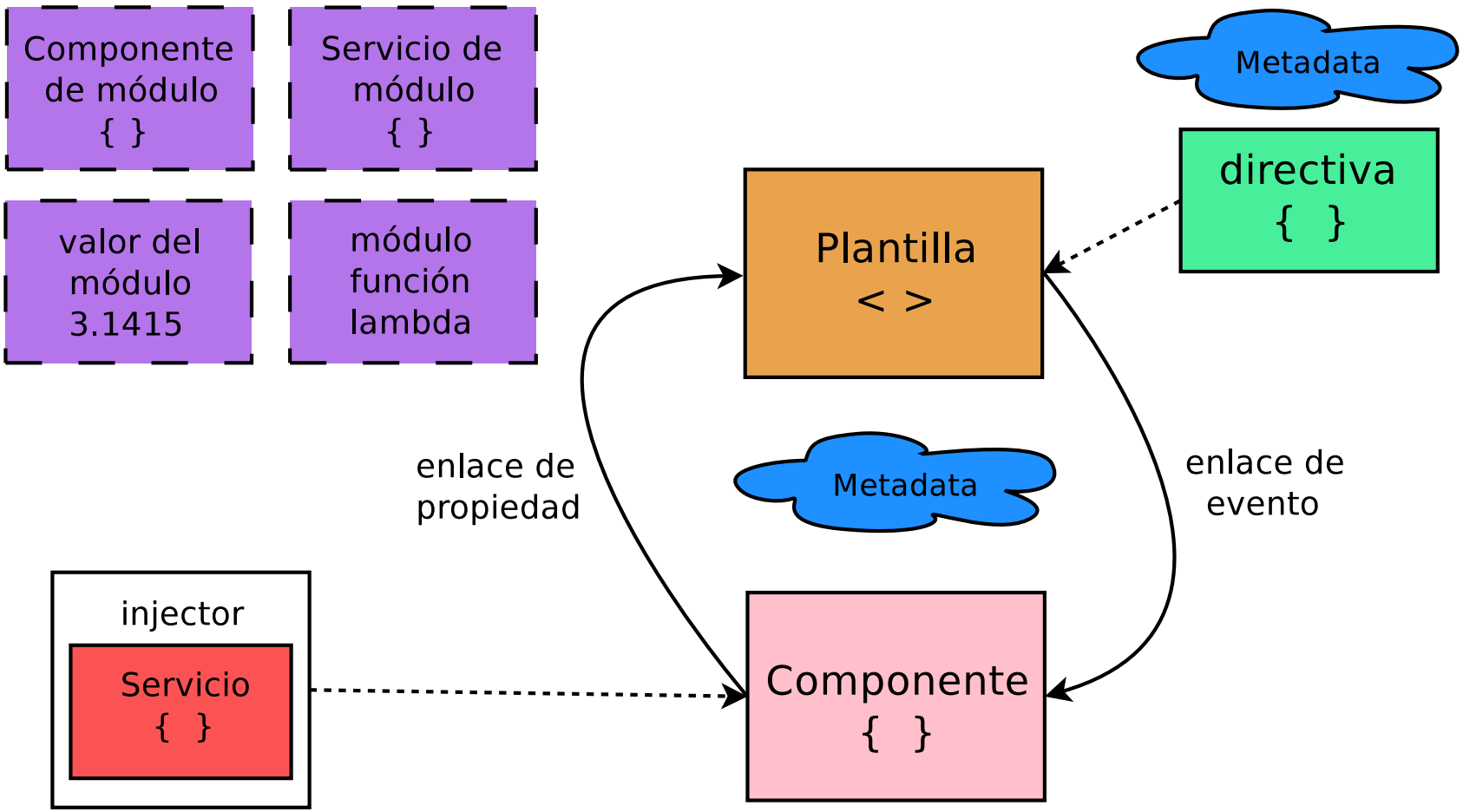
- El módulo de enrutamiento `NgModule` de Angular proporciona un servicio que permite definir una ruta de navegación entre los diferentes estados de la aplicación
- Se basa en las convenciones características de navegación del navegador:
  - Ingrese una URL en la barra de direcciones y el navegador navega a la página correspondiente
  - Haga clic en los enlaces en la página y el navegador navega a una nueva página
  - Haga clic en los botones de avance y retroceso del navegador y el navegador navega hacia atrás y hacia adelante a través del historial de las páginas que ha visitado
- El enrutador asigna rutas de tipo URL a vistas en lugar de páginas

Enrutamiento cont...

- Cuando un usuario realiza una acción, como hacer clic en un enlace, que cargaría una nueva página en el navegador, el enrutador intercepta el comportamiento del navegador y muestra u oculta las jerarquías de vista
- Si el enrutador determina que el estado actual de la aplicación requiere una funcionalidad particular, y el módulo que lo define no se ha cargado, el enrutador puede cargar el módulo a demanda
- El enrutador interpreta una URL de enlace de acuerdo con las reglas de navegación de la aplicación y el estado de los datos
- Puede navegar a nuevas vistas cuando el usuario hace clic en un botón o selecciona de un cuadro desplegable, o en respuesta a algún otro estímulo de cualquier fuente
- El enrutador registra la actividad en el historial del navegador, por lo que los botones de avance y retroceso también funcionan

Enrutamiento cont...

- Para definir reglas de navegación, asocie las rutas de navegación con sus componentes
- Una ruta utiliza una sintaxis similar a una URL que integra los datos de su programa, de la misma manera que la sintaxis de la plantilla integra sus vistas con los datos de su programa
- Luego, puede aplicar la lógica del programa para elegir qué vistas mostrar u ocultar, en respuesta a la entrada del usuario y sus propias reglas de acceso


Componente de módulo { }
Servicio de módulo { }
valor del módulo 3.1415
módulo función lambda
Metadata
directiva { }
Plantilla < >
enlace de propiedad
Metadata
enlace de evento
injector
Servicio { }
Componente { }

Bases del lenguaje

Luciano Diamand

Mostrando datos

Interpolación

- La forma mas simple de mostrar una propiedad de un componente es a través de la **interpolación**
- Para usar interpolación escribimos el nombre de la propiedad en la plantilla de la vista encerrado entre doble llaves `{{ }}`
- Angular reemplaza el nombre con el valor de cadena de la propiedad del compnente correspondiente
- En la plantilla del componente:

```
<h1>{{ title }}</h1>
<p>Heroe con mas heridas: {{ myHero }}</p>
```

Interpolacion cont...

- Ahora, en la clase del componente:

```
@Component({
...
})
export class HeroComponent {
  title = 'Obra Social de Heroes';
  myHero = 'Batman';
}
```

- Angular automaticamente toma los valores de las propiedades `title` y `myHero` del componente e inserta esos valores en el navegador
- Angular actualiza la pantalla cuando estas propiedades cambian

Interpolacion cont...

- De forma mas general, el texto entre las doble llaves es una expresión de plantilla que Angular evalua y luego convierte a cadena

```
<!-- la suma de 1 + 1 es 2 -->
<p>La suma de 1 + 1 es {{ 1 + 1 }}</p>
<p>{{ "Bienvenido" + name }}</p>
<p>{{ name.length }}</p>
<p>{{ name.toUpperCase() }}</p>
<p>{{ greetUser() }}</p>
```

Creando una clase para los datos

- El código del componente anterior define los datos directamente dentro del mismo componente, lo cual no es una buena práctica
- En aplicaciones reales, la mayoría de los enlaces se realiza a objetos mas especializados
- Vamos a mover los datos hacia su propia clase, para ello usamos Angular CLI:

```
ng generate class hero
```

- y editamos el siguiente código `src/app/hero.ts`:

```
export class Hero {
  constructor(public id: number, public name: string) { }
}
```

Sintaxis en plantillas - HTML

- HTML es el lenguaje utilizado en las plantillas de Angular
- Casi todos lo elementos de HTML son válidos dentro de la plantilla
- La excepcion es el elemento `<script>` que esta prohibido, eliminando de esa forma la posibilidad de ataques de injección de script
- En la práctica, `<script>` es ignorado y se visualiza una advertencia en la consola del navegador
- Algunos elementos HTML válidos, no tienen mucho sentido en una plantilla (`<html>`, `<body>` y `<base>`)
- Es posible extender el vocabulario de las plantillas con componentes y directivas que aparecen como nuevos elementos y atributos

Expresiones en plantillas

- Una expresión de plantilla produce un valor
- Angular ejecuta la expresión y la asigna a una propiedad del enlace destino; el destino puede ser un elemento HTML, un componente o una directiva
- En el enlace de propiedades, una expresión de plantilla aparece entre comillas a la derecha del símbolo `=` como `[propiedad]="expresion"`
- Las expresiones de plantilla se escriben en un lenguaje similar a JavaScript, si embargo esta prohibido:
  - Asignaciones (`=`, `+=`, `-=`, `...`)
  - `new`
  - Encadenando expresiones: `;` o `,`
  - Operadores de incremento/decremento (`++`, `--`)

Sentencias en plantillas

- Un sentencia en plantilla responde a un evento lanzado por el destino, como ser un elemento, componente o directiva
- Un sentencia en plantilla aparece entre comillas a la derecha del símbolo = como `(event)="statement"`

```
<button (click)=deleteHero()">Borrar heroe</button>
```

- Las sentencias de plantilla se escriben en un lenguaje similar a JavaScript, donde estan permitidos:
  - Asignaciones = y encadenamiento de expresiones con ; o ,
- Sin embargo esta prohibido:
  - `new`
  - Operadores de incremento/decremento (++, --)
  - Operadores de asignacion += y -=

Enlaces de datos

- El enlace de datos es un mecanismo para coordinar lo que ve el usuario con los valores de los datos de la aplicación
- Los tipos de enlaces se pueden agrupar en tres categorias segun la dirección del flujo de datos:
  - origen-a-vista
  - vista-a-origen
  - vista-a-origen-a-vista

Atributo HTML vs Propiedad DOM

- Los atributos están definidos por HTML
- Las propiedades son definidas por el DOM
  - Algunos atributos HTML tienen una asignación 1:1 a las propiedades. *id* es un ejemplo
  - Algunos atributos HTML no tienen propiedades correspondientes. *colspan* es un ejemplo
  - Algunas propiedades de DOM no tienen atributos correspondientes. *textContent* es un ejemplo
- Como regla general: *Los atributos inicializan las propiedades DOM. Los valores de propiedad pueden cambiar; los valores de los atributos no pueden*

Enlace de propiedades

- Escriba un enlace de propiedad de plantilla para establecer una propiedad de un elemento de vista. El enlace establece la propiedad al valor de una expresión de plantilla.
- El enlace de propiedad más común establece una propiedad de elemento en un valor de propiedad de componente. Un ejemplo es vincular la propiedad `src` de un elemento de imagen a la propiedad *heroImageUrl* de un componente:

```
<img [src]="heroImageUrl" />
```

- Otro ejemplo es deshabilitar un botón cuando el componente dice que no se ha cambiado:

```
<button [disabled]="isUnchanged">Cancelar deshabilitado</button>
```

Enlace de propiedades cont...

- También es posible establecer la propiedad de una directiva:

```
<div [ngClass]="classes">
  [ngClass] enlazado a la propiedad de la clase
</div>
```

- A su vez, se puede configurar la propiedad de modelo de un componente personalizado (una excelente manera para que los componentes padre e hijo se comuniquen):

```
<app-hero-detail [hero]="currentHero"></app-hero-detail>
```

Enlace de atributos

- Puede establecer el valor de un atributo directamente con un enlace de atributo
- Debe utilizar el enlace de atributos cuando no haya ninguna propiedad de elemento para enlazar
- Considere los atributos ARIA, SVG y span de una tabla. Son atributos puros, no corresponden a las propiedades del elemento y no establecen las propiedades del elemento. No hay objetivos de propiedad para enlazar
- Este hecho se vuelve dolorosamente obvio cuando escribes algo como esto

```
<tr><td colspan="{{1 + 1}}">Error</td></tr>
```

Enlace de atributos cont...

- La sintaxis de enlace de atributo se asemeja al enlace de propiedad
- En lugar de una propiedad de elemento entre corchetes, comience con el prefijo `attr`, seguido de un punto `(.)` y el nombre del atributo
- A continuación, establece el valor del atributo, utilizando una expresión que se resuelve en una cadena

```
<table border=1>
  <!--  expression calculates colspan=2 -->
  <tr><td [attr.colspan]="1 + 1">One-Two</td></tr>
  <tr><td>Five</td><td>Six</td></tr>
</table>
```

Enlace de eventos

- Las directivas de enlaces que ha encontrado hasta ahora fluyen datos en una dirección: de un componente a un elemento
- La sintaxis de enlace de eventos consiste en un nombre de evento de destino entre paréntesis a la izquierda de un signo igual y una declaración de plantilla entre comillas a la derecha
- El siguiente enlace de evento escucha los eventos de clic del botón, y llama al método `onSave()` del componente cada vez que se produce un clic:

```
<button (click)="onSave()">Guardar</button>
```

- Algunas personas prefieren la alternativa `on-prefix`, conocida como la forma canónica:

```
<button on-click="onSave()">Guardar</button>
```

Enlace de eventos cont...

- Los eventos de elementos pueden ser los objetivos más comunes, pero Angular mira primero para ver si el nombre coincide con una propiedad de evento de una directiva conocida, como lo hace en el siguiente ejemplo:

```
<!-- `myClick` is an event on the custom `ClickDirective` -->
<div (myClick)="clickMessage=$event" clickable>click with myClick</div>
```

- Si el nombre no coincide con un evento de elemento o una propiedad de salida de una directiva conocida, Angular informa de un error de *"directiva desconocida"*

$event y sentencias de manejo de eventos

- En un enlace de evento, Angular configura un controlador de eventos para el evento objetivo
- Cuando se levanta el evento, el controlador ejecuta la declaración de plantilla. La declaración de plantilla generalmente involucra a un receptor, que realiza una acción en respuesta al evento, como almacenar un valor del control HTML en un modelo
- El enlace transmite información sobre el evento, incluidos los valores de los datos, a través de un objeto de evento llamado $event
- La forma del objeto de evento está determinada por el evento de destino. Si el evento de destino es un evento de elemento DOM nativo, entonces $event es un objeto de evento de DOM, con propiedades como target y target.value.

Enlace de eventos cont...

- Considera este ejemplo:

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

- Este código establece la propiedad de valor del cuadro de entrada enlazando a la propiedad de nombre. Para escuchar los cambios en el valor, el código se enlaza al evento de entrada del cuadro de entrada. Cuando el usuario realiza cambios, se genera el evento de entrada y el enlace ejecuta la declaración dentro de un contexto que incluye el objeto de evento DOM, $event

Propiedades de entrada y salida

- Una propiedad de entrada es una propiedad configurable anotada con un decorador `@Input`
- Los valores fluyen dentro de la propiedad cuando está enlazado a datos con un enlace de propiedad
- Una propiedad de Salida es una propiedad observable anotada con un decorador `@Output`
- La propiedad casi siempre devuelve un `EventEmitter` de Angular
- Los valores salen del componente como eventos enlazados con un enlace de evento.
- Solo puede enlazar a otro componente o directiva a través de sus propiedades de Entrada y Salida

Enlazando a un componente diferente

- También puede enlazar a una propiedad de un componente diferente
- En dichos enlaces, la propiedad del otro componente está a la izquierda de (=)
- En el siguiente ejemplo, la plantilla de AppComponent vincula a los miembros de la clase AppComponent con las propiedades de HeroDetailComponent cuyo selector es 'app-hero-detail'

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

- El compilador de Angular puede rechazar estos enlaces con errores como este

```
Uncaught Error: Template parse errors:
Can't bind to 'hero' since it isn't a known property of 'app
```

Declarar las propiedades de entrada y salida

- En el ejemplo de esta guía, los enlaces a `HeroDetailComponent` no fallan porque las propiedades de enlace de datos están anotadas con los decoradores `@Input()` y `@Output()`

```
@Input()  hero: Hero;
@Output() deleteRequest = new EventEmitter<Hero>();
```

- Alternativamente, puede identificar miembros en las matrices de entradas y salidas de los metadatos de la directiva, como en este ejemplo:

```
@Component({
  inputs: ['hero'],
  outputs: ['deleteRequest'],
})
```

Enlace bidireccional

- A menudo desea mostrar una propiedad de datos y actualizar esa propiedad cuando el usuario realiza cambios
- En el lado del elemento que toma una combinación de establecer una propiedad del elemento específica y escuchar un evento de cambio de elemento
- Angular ofrece una sintaxis especial de enlace de datos bidireccional para este propósito, `[(x)]`
- La sintaxis `[(x)]` combina los paréntesis del enlace de propiedades, `[x]`, con los paréntesis del enlace de eventos, `(x)`
- La sintaxis de `[(x)]` es fácil de demostrar cuando el elemento tiene una propiedad configurable llamada *x* un evento correspondiente llamado `xChange`
- Aquí hay un SizerComponent que se ajusta al patrón. Tiene una propiedad de valor *size* y un evento complementario *sizeChange*:

Enlace bidireccional cont...

```
import { Component, EventEmitter, Input, Output } from '@angular/core';

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

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

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

Enlace bidireccional cont...

- Aquí hay un ejemplo en el que `AppComponent.fontSizePx` está vinculado de manera bidireccional al `SizerComponent`:

```
<app-sizer [(size)]="fontSizePx"></app-sizer>
<div [style.font-size.px]="fontSizePx">Resizable Text</div>
```

- La sintaxis de enlace bidireccional es en realidad solo una simplificación sintáctica para un enlace de propiedad y un enlace de evento. Angular transforma dicha sintaxis en `SizerComponent` en lo siguiente:

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

Laboratorio práctico - Creando componentes

Es el momento de trabajar con componentes

- Creacion de un componente enlazado a datos
- Comunicacion con componentes hijos
- Comunicacion con el componente padre
- Uso de variables locales para interactuar con componentes secundarios
- Agregando estilos a componentes

Directivas estructurales

Luciano Diamand

Directivas estructurales

¿Que son las directivas estructurales?

- Las directivas estructurales son responsables del diseño HTML
- Ellas forman o reforman la estructura del DOM, normalmente agregando, quitando, o manipulando elementos
- Como con otras directivas, aplicamos una directiva estructural al elemento anfitrión. La directiva hace entonces lo que tiene que hacer con el elemento y sus descendientes
- Las directivas estructurales son fáciles de reconocer. Un asterisco (*) precede al nombre de la directiva como en este ejemplo:

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

ng-component

- Angular descompone esta notación en la etiqueta de marca `<ng-template>` que envuelve al elemento anfitrión y sus descendientes

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

Directiva NgIf

- Es posible agregar o quitar un elemento del DOM a través de aplicar la etiqueta `NgIf` al elemento (llamado elemento anfritrión).
- Enlazamos la directiva a una expresión condicional; como por ejemplo `isActive`

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

- Cuando la expresión `isActive` devuelve un valor verdadero, `NgIf` agrega el `HeroDetailComponent` al DOM
- Cuando la expresión es falsa, `NgIf` quita el `HeroDetailComponent` del DOM, destruyendo aquel componente y todos sus sub-componentes

Protección contra null

- La directiva `ngIf` se usa a menudo para proteger contra `null`
- Mostrar/ocultar es inútil como verificación
- Angular lanzará un error si una expresión anidada intenta acceder a una propiedad `null`
- Aquí vemos a `NgIf` custodiando dos `<div>`s. El nombre *currentHero* aparecerá solo cuando haya un currentHero. El *nullHero* nunca se mostrará

```
<div *ngIf="currentHero">Hola, {{ currentHero.name }}</div>
<div *ngIf="nullHero">Hola, {{ nullHero.name }}</div>
```

Operador de navegación segura

- El operador de navegación segura de Angular `(?.)` es una forma fluida y conveniente de protegerse contra valores nulos e indefinidos en rutas de propiedad
- Aquí está protegiendo contra una falla de render de la vista si el *currentHero* es nulo

```
El nombre del héroe actual es {{ currentHero?.name }}
```

- ¿Qué sucede cuando la siguiente propiedad enlazada al título es nula?

```
El título es {{title}}
```

Directiva NgForOf

- `NgForOf` es una directiva de repetición, una forma de presentar una lista de elementos
- Se define un bloque de HTML que define cómo se debe mostrar un solo elemento
- Luego Angular usa ese bloque como plantilla para representar cada elemento de la lista
- Aquí hay un ejemplo de `NgForOf` aplicado a un simple `<div>`:

```
<div *ngFor="let hero of heroes">{{ hero.name }}</div>
```

- También puede aplicar un `NgForOf` a un componente, como en este ejemplo:

```
<app-hero-detail *ngFor="let hero of heroes" [hero]="hero">
</app-hero-detail>
```

Directiva NgForOf cont...

- La cadena asignada a `*ngFor` no es una expresión de plantilla
- Se llama **microsyntax**, es un lenguaje pequeño propio de Angular
- La cadena *"let hero of heroes"* significa:
  Toma a cada heroe del arreglo heroes, guárdalo en la variable de bucle local hero y ponlo a disposición de la plantilla HTML para cada iteración
- Angular descompone esta notación en la etiqueta de marca `<ng-template>` que envuelve al elemento anfitrión y sus descendientes

Variables de entrada de plantilla

- La palabra clave `let` antes de `hero` crea una variable de entrada de plantilla llamada **hero**
- La directiva `NgForOf` itera sobre el arreglo `heroes` que retorna la propiedad `heroes` del componente principal y establece a `hero` en el elemento actual del arreglo durante cada iteración
- Hace referencia a la variable de entrada de `hero` dentro del elemento anfitrión `NgForOf` (y dentro de sus descendientes) para acceder a las propiedades de `hero`
- Aquí está referenciado primero en una interpolación y luego pasado en un enlace a la propiedad de `hero` del componente `<hero-detail>`

```
<div *ngFor="let hero of heroes">{{ hero.name }}</div>

<app-hero-detail *ngFor="let hero of heroes" [hero]="hero">
</app-hero-detail>
```

*ngFor con índice

- La propiedad `index` del contexto de la directiva `NgForOf` devuelve el índice (comenzando en cero) del elemento en cada iteración
- Puede capturar el índice en una variable de entrada de plantilla y utilízala en la plantilla
- El siguiente ejemplo captura el índice en una variable llamada **i** y la muestra junto con el nombre del héroe

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

La directiva NgSwitch

- `NgSwitch` se comporta como la instrucción `switch` de JavaScript
- Puede mostrar un elemento entre varios elementos posibles, basado en una condición
- Angular coloca solo el elemento seleccionado en el DOM
- `NgSwitch` es en realidad un conjunto de tres directivas: `NgSwitch`, `NgSwitchCase` y `NgSwitchDefault` como se ve en este ejemplo:

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

Variables de referencia en plantilla (#var)

- Una variable de referencia en plantilla es a menudo una referencia a un elemento DOM dentro de la plantilla
- También puede ser una referencia a un componente Angular, Directiva o un componente web
- Use el símbolo de cardinal (#) para declarar una variable de referencia

Variables de referencia en plantilla (#var) cont...

- En el siguiente ejemplo, `#phone` declara una variable dentro del elemento `<input>`
- Puede referirse a una variable de referencia en plantilla en cualquier parte del la plantilla
- La variable *phone* declarada en el elemento `<input>` se consume en el elemento `<button>` (en otra parte de la plantilla)

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

<!-- Otros elementos -->

<!-- phone referencia al elemento input;
     pasa `value` al gestor del evento -->
<button (click)="callPhone(phone.value)">Llamar</button>
```

Directivas de atributos incorporados

- Las directivas de atributos escuchan y modifican el comportamiento de otros elementos HTML, atributos, propiedades y componentes
- Por lo general se aplican a los elementos como si fueran atributos HTML, de ahí el nombre.
- Muchos `NgModules` como `RouterModule` y `FormsModule` definen sus propias directivas de atributos
- Las directivas de atributos más utilizadas son:

NgClass	añade o elimina un conjunto de clases CSS
NgStyle	añade o elimina un conjunto de estilos HTML
NgModel	enlace de datos bidireccional a un elemento de formulario HTML

NgClass

- Normalmente controla cómo aparecen los elementos agregando y eliminando clases CSS de forma dinámica
- Puede enlazar a `ngClass` para agregar o eliminar varias clases simultáneamente
- Un enlace de clase es una buena manera de agregar o quitar una clase única

```
<!-- intercambia la clase "special" on/off con la propiedad -->
<div [class.special]="isSpecial">El enlace de la clase es especial</div>
```

NgClass cont...

- Para agregar o eliminar muchas clases CSS al mismo tiempo, la directiva `NgClass` puede ser la mejor opción
- Intente vincular `ngClass` a un objeto clave:valor
- Cada clave del objeto es el nombre de clase CSS; el valor es verdadero si se debe agregar la clase o falso si se debe eliminar

NgClass cont...

- Considere el método `setCurrentClasses` de un componente que establece una propiedad `currentClasses` con un objeto que agrega o elimina tres clases según el estado verdadero/falso de otras tres propiedades del componente:

```
currentClasses: {};

setCurrentClasses() {
  // clases CSS: agreagdo/eliminado a traves del estado de las propiedades
  // del componente
  this.currentClasses =  {
    'saveable': this.canSave,
    'modified': !this.isUnchanged,
     'special':  this.isSpecial
}
```

- Al agregar un enlace de propiedad `ngClass` a *currentClasses*, se establecen las clases del elemento en consecuencia

```
<div [ngClass]="currentClasses">
  This div is initially saveable, unchanged, and special
</div>
```

NgStyle

- Puede establecer estilos en línea dinámicamente, en función del estado del componente
- Con `NgStyle` puede configurar muchos estilos en línea simultáneamente
- Un enlace de estilo es una manera fácil de establecer un valor de estilo único

```
<div [style.font-size]="isSpecial ? 'x-large' : 'smaller'" >
  Este div es x-large o smaller.
</div>
```

- Para establecer muchos estilos en línea al mismo tiempo, la directiva `NgStyle` puede ser la mejor opción
- Intente vincular `ngStyle` a un objeto clave:valor
- Cada clave del objeto es el nombre de estilo; el valor es un valor que sea apropiado para ese estilo

NgStyle cont...

- Considere un método de componente `setCurrentStyles` que establece una propiedad del componente, `currentStyles` con un objeto que define tres estilos, basado en el estado de otras tres propiedades del componente:

```
currentStyles: {};
setCurrentStyles() {
  // estilos CSS: asigna el estado actual de las propiedades del component
  this.currentStyles = {
    'font-style':  this.canSave      ? 'italic' : 'normal',
    'font-weight': !this.isUnchanged ? 'bold'   : 'normal',
    'font-size':   this.isSpecial    ? '24px'   : '12px'
  };
}
```

NgStyle cont...

- El agregar un enlace de propiedad `ngStyle` a `currentStyles`, establece los estilos del elemento acorde:

```
<div [ngStyle]="currentStyles">
  Este div es inicialmente italic, normal weight, y extra large (24px).
</div>
```

Laboratorio práctico - Directivas embebidas

Es el momento de trabajar con las directivas que proporciona Angular

- Repetir datos con ngFor
- Usando el operador de navegacion segura
- Ocultar elementos con ngIf
- Elementos ocultos con hidden
- Ocultar y mostrar elementos con ngSwitch
- Anadiendo estilo con ngClass
- Anadiendo estilo con ngStyle

Servicios en Angular

Luciano Diamand

Visión general

- Los componentes no deben recuperar o guardar datos directamente
- Deben centrarse en presentar datos y delegar el acceso de datos a un servicio
- Los servicios son una excelente manera de compartir información entre clases que no se conocen entre sí
- Para generar un nuevo servicio con Angular CLI:

```
ng generate service empleado
```

Visión general cont...

- Esto creará una clase

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

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

  constructor() { }

}
```

- Para injectar el servicio en un componente debemos:

```
constructor(private empleadoService: EmpleadoService) { }
```

Laboratorio práctico - Creación de servicios

Vamos a crear servicios reutilizables

- Creación e inyección de servicios

Ruteo en Angular

Luciano Diamand

Visión general

- El navegador utiliza un modelo familiar de navegación de aplicaciones:
  - Ingrese una URL en la barra de direcciones y el navegador navega a la página correspondiente
  - Haga clic en los enlaces en la página y el navegador navega a una nueva página
  - Haga clic en los botones de avance y retroceso del navegador y el navegador se desplazará hacia atrás y hacia adelante a través del historial de las páginas que ha visto
- El enrutador de Angular toma prestado de este modelo
- Puede interpretar una URL del navegador como una instrucción para navegar a una vista generada por el cliente
- Puede pasar parámetros opcionales junto con el componente de vista que lo ayude a decidir qué contenido específico presentar

Visión general cont...

- Puede enlazar el enrutador a los enlaces en una página y navegar a la vista de aplicación adecuada cuando el usuario haga clic en un enlace
- Puede navegar de forma imperativa cuando el usuario hace clic en un botón, selecciona de un cuadro desplegable o en respuesta a algún otro estímulo de cualquier fuente
- El enrutador registra la actividad en el diario de historial del navegador para que los botones de avance y retroceso también funcionen

base href

- La mayoría de las aplicaciones de enrutamiento deben agregar el elemento `<base>` al `index.html` como el primer elemento dentro de la etiqueta `<head>` para decirle al enrutador cómo armar las URL de navegación
- Si la carpeta de la aplicación es la raíz de la aplicación, como lo es para las aplicaciones de ejemplo, establece el valor `href` exactamente como se muestra aquí

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

Importar el enrutador

- El enrutador de Angular es un servicio opcional que presenta una vista de componente particular para una URL determinada
- No es parte del núcleo Angular, está en su propio paquete de biblioteca, `@angular/router`
- Luego importe lo que necesite de él como lo haría desde cualquier otro paquete Angular

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

Configuración

- Una aplicación en Angular con enturamiento tiene una instancia única del servicio de Enrutador `Router`
- Cuando cambia la URL del navegador, ese enrutador busca una ruta correspondiente desde la cual puede determinar el componente que se mostrará
- Un enrutador no tiene rutas definidas por defecto
- El siguiente ejemplo crea cinco definiciones de ruta, configura el enrutador a través del método `RouterModule.forRoot` y agrega el resultado a la matriz de importaciones de `AppModule`

Configuración cont...

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

@NgModule({
  imports: [
     RouterModule.forRoot(appRoutes,
       { enableTracing: true } // <-- debugging purposes only
     ) // other imports here
  ],
  ...
})
export class AppModule { }
```

Configuración cont...

- El conjunto de rutas de `appRoutes` describe cómo navegar al pasarlo al método `RouterModule.forRoot` en las importaciones del módulo para configurar el enrutador
- Cada ruta asigna una ruta URL a un componente (No hay barras al inicio de la ruta)
- El enrutador analiza y construye la URL final, lo que le permite utilizar rutas relativas y absolutas cuando navega entre las vistas de la aplicación
- El `:id` en la segunda ruta es un token para un parámetro de ruta
- En una URL como `/hero/42`, "42" es el valor del parámetro id
- El `HeroDetailComponent` correspondiente usará ese valor para encontrar y presentar al héroe cuya *id* es *42*

Configuración cont...

- La propiedad de datos en la tercera ruta es un lugar para almacenar datos arbitrarios asociados con esta ruta específica
- La propiedad de datos es accesible dentro de cada ruta activada
- Se puede utilizar para almacenar elementos como títulos de página, texto de ruta de navegación y otros datos estáticos de solo lectura
- La ruta vacía en la cuarta ruta representa la ruta predeterminada para la aplicación, es el lugar al que va a ir cuando la ruta en la URL está vacía, y que por lo general es al inicio
- Esta ruta por defecto redirige a la ruta de URL `/heroes` y, por lo tanto, mostrará el `HeroesListComponent`

Configuración cont...

- Los `**` en el `path` de la última ruta es un comodín
- El enrutador seleccionará esta ruta si la URL solicitada no coincide con ninguna ruta para las rutas definidas anteriormente en la configuración
- Esto es útil para mostrar una página *"404 - No encontrado"* o redirigir a otra ruta
- El orden de las rutas en la configuración es importante
- El enrutador utiliza una estrategia de victoria en la primera coincidencia al hacer coincidir las rutas, por lo que se deben colocar las rutas más específicas sobre rutas menos específicas
- En la configuración anterior, las rutas con una ruta estática se enumeran primero, seguidas de una ruta de ruta vacía, que coincide con la ruta predeterminada
- La ruta de comodín viene en último lugar porque coincide con todas las URL y debe seleccionarse solo si ninguna otra ruta coincide primero

Configuración cont...

- Si necesita ver qué eventos están sucediendo durante el ciclo de vida de navegación, existe la opción `enableTracing` como parte de la configuración por defecto del router
- Esto registra en la consola cada evento que tuvo lugar en el enrutador durante cada ciclo de vida de la navegación
- Esto solo debe ser usado para la depuración
- Establezca la opción `enableTracing: true` en el objeto que se pasa como el segundo argumento al método `RouterModule.forRoot()`

Salida del Enrutador

- El `RouterOutlet` es una directiva de la biblioteca del enrutador que se utiliza como un componente
- Actúa como una marca de la posición del lugar en la plantilla donde el enrutador debe mostrar los componentes para esa salida

```
<router-outlet></router-outlet>
<!-- Los componentes enrutados van aquí -->
```

- Dada la configuración anterior, cuando la URL del navegador para ésta aplicación, se convierte en `/heroes`, el enrutador hace coincidir esa URL con la ruta `heroes` y muestra `HeroListComponent` como elemento relacionado con el `RouterOutlet`, colocado en la plantilla del componente anfitrión

Enlaces al enrutador

- Ahora ya tenemos rutas configuradas y un lugar para renderizarlas, pero ¿cómo navega?
- La URL podría escribirse directamente desde la barra de direcciones del navegador, pero la mayoría de las veces navega como resultado de alguna acción del usuario, como el clic de una etiqueta de ancla
- Consideremos la siguiente plantilla:

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

Enlaces de enrutador activos

- La directiva `RouterLinkActive` alterna las clases CSS para los enlaces `RouterLink` activos en función del `RouterState` actual
- En cada etiqueta de anclaje, agregamos un enlace de propiedad a la directiva `RouterLinkActive` de la forma `routerLinkActive="..."`
- La expresión de plantilla a la derecha del igual `(=)` contiene una cadena de clases CSS delimitada por espacios que el Enrutador agregará cuando este enlace esté activo (y se eliminará cuando el enlace esté inactivo)
- Establece la directiva `RouterLinkActive` a una cadena de clases como `[routerLinkActive]="'active fluffy'"` o la vincula a una propiedad de componente que devuelve dicha cadena

Enlaces de enrutador activos

- Los enlaces de ruta activos caen en cascada a través de cada nivel del árbol de rutas, por lo que los enlaces de enrutador principal y secundario pueden estar activos al mismo tiempo
- Para anular este comportamiento, puede enlazar el enlace de entrada `[routerLinkActiveOptions]` con la expresión `{exact: true}`
- Al usar `{exact: true}`, un `RouterLink` dado solo estará activo si su URL coincide exactamente con la URL actual

Estado del enrutador

- Después del final de cada ciclo de vida de navegación exitoso, el enrutador crea un árbol de objetos `ActivatedRoute` que conforman el estado actual del enrutador
- Puede acceder al `RouterState` actual desde cualquier lugar de la aplicación utilizando el servicio `Router` y la propiedad `routerState`
- Cada `ActivatedRoute` en `RouterState` proporciona métodos para recorrer el árbol de rutas hacia arriba y hacia abajo para obtener información de las rutas padre, hijo y hermano

Ruta activada

- La ruta y los parámetros están disponibles a través de un servicio de enrutador inyectado llamado `ActivatedRoute`
- Tiene una gran cantidad de información útil que incluye:

Propiedad	Descripción
`url`	Un observable de la ruta, representado como una matriz de cadenas para cada parte de la ruta
`data`	Un observable que contiene el objeto de datos proporcionado para la ruta.
`paramMap`	Un observable que contiene un mapa de los parámetros requeridos y opcionales específicos de la ruta. El mapa admite la recuperación de valores únicos y múltiples del mismo parámetro
`queryParamMap`	Un observable que contiene un mapa de los parámetros de consulta disponibles para todas las rutas. El mapa admite la recuperación de valores únicos y múltiples del parámetro de consulta

Ruta activada

`fragment`	Un observable del fragmento de URL disponible para todas las rutas
`outlet`	El nombre del RouterOutlet utilizado para representar la ruta. Para una salida sin nombre, el nombre de la salida es primario
`routeConfig`	La configuración de ruta utilizada para la ruta que contiene la ruta de origen
`parent`	La ruta activada principal de la ruta cuando esta ruta es una ruta secundaria
`firstChild`	Contiene el primer ActivatedRoute en la lista de rutas secundarias de esta ruta
`children`	Contiene todas las rutas secundarias activadas bajo la ruta actual

Ruta activada cont...

- Dos propiedades más antiguas todavía están disponibles. Son menos capaces que sus reemplazos, se desaniman y pueden ser desaprobados en una futura versión Angular

`params`	Un observable que contiene los parámetros requeridos y opcionales específicos de la ruta. Use paramMap en su lugar
`queryParams`	un observable que contiene los parámetros de consulta disponibles para todas las rutas. Utilice queryParamMap en su lugar

Eventos enrutador

- Durante cada navegación, el enrutador emite eventos de navegación a través de la propiedad `Router.events`. Estos eventos van desde cuando comienza y finaliza la navegación hasta muchos puntos intermedios
- La lista completa de eventos de navegación se muestra en la siguiente tabla

Evento enrutador	Descripción
`NavigationStart`	Un evento activado cuando comienza la navegación
`RouteConfigLoadStart`	Un evento desencadenado antes de que el enrutador cargue la configuración de una ruta
`RouteConfigLoadEnd`	Un evento desencadenado después de que una ruta ha sido cargada perezosamente
`RoutesRecognized`	Un evento activado cuando el enrutador analiza la URL y se reconocen las rutas
`GuardsCheckStart`	Un evento se activa cuando el Enrutador comienza la fase de enrutamiento de Guardias
`ChildActivationStart`	Un evento se activa cuando el enrutador comienza a activar los hijos de una ruta
`ActivationStart`	Un evento se activa cuando el enrutador comienza a activar una ruta

Eventos enrutador cont...

`GuardsCheckEnd`	Un evento se activa cuando el enrutador finaliza la fase de guardias de enrutamiento con éxito
`ResolveStart`	Un evento se activa cuando el enrutador comienza la fase de resolución del enrutamiento
`ResolveEnd`	Un evento se activa cuando el enrutador finaliza la fase de resolución de enrutamiento correctamente
`ChildActivationEnd`	Un evento se activa cuando el enrutador termina de activar los hijos de una ruta
`ActivationEnd`	Un evento se activa cuando el enrutador termina de activar una ruta
`NavigationEnd`	Un evento activado cuando la navegación termina con éxito
`NavigationCancel`	Un evento activado cuando se cancela la navegación. Esto se debe a que una Guardia de ruta devuelve falso durante la navegación
`NavigationError`	Un evento se desencadena cuando falla la navegación debido a un error inesperado
`NavigationError`	An event triggered when navigation fails due to an unexpected error
`Scroll`	Un evento que representa un evento de desplazamiento

Guardias de rutas

- Agregue guardias a la configuración de ruta para manejar estos escenarios
  - Si devuelve verdadero, el proceso de navegación continúa
  - Si devuelve falso, el proceso de navegación se detiene y el usuario se queda en el lugar actual
- El enrutador soporta múltiples interfaces de guarda
  - CanActivate: para mediar la navegación a una ruta
  - CanActivateChild: para mediar la navegación a una ruta secundaria
  - CanDeactivate: para mediar la navegación fuera de la ruta actual y mediar la navegación a una ruta secundaria
  - Resolve: para realizar la recuperación de datos de ruta antes de la activación de la ruta
  - CanLoad: para mediar la navegación a un módulo de características cargado de forma asíncrona

Guardias de rutas cont...

- Puede tener múltiples guardias en cada nivel de una jerarquía de enrutamiento
- El enrutador comprueba primero las guardias `CanDeactivate` y `CanActivateChild`
- Luego verifica los guardias `CanActivate`
- Si el módulo de características se carga de forma asíncrona, la protección de `CanLoad` se comprueba antes de que se cargue el módulo
- Si algún guardia devuelve falso, los guardias pendientes que no se hayan completado se cancelarán y se cancelará toda la navegación

Laboratorio práctico - Enrutamiento y navegación

Es el momento de navegar entre las vistas

- Creando una Ruta
- Usando Parámetros de Ruta
- Enlace a Rutas
- Navegando desde el código
- Cómo evitar que una ruta se active
- Cómo evitar que una ruta se desactive
- Carga previa de datos para un componente

Introducción a formularios en Angular

Luciano Diamand

Visión general

- El manejo de la entrada de usuario con formularios es la pieza clave de muchas aplicaciones
- Las aplicaciones utilizan formularios para permitir a los usuarios iniciar sesión, actualizar un perfil, ingresar información confidencial y realizar muchas otras tareas de ingreso de datos
- Angular proporciona dos enfoques diferentes para manejar la entrada de usuario. Las opciones son:
  - formularios reactivos (basado en modelos)
  - formularios basados en plantillas
- Ambos capturan los eventos de entrada del usuario desde la vista, validan la entrada del usuario, crean un modelo de formulario y un modelo de datos para actualizar, y proporcionan una forma de realizar un seguimiento de los cambios

Visión general cont...

- Los formularios reactivos y los controlados por plantillas procesan y administran los datos del formulario de manera diferente
- Cada uno ofrece diferentes ventajas
- En general:
  - Los formularios reactivos son más robustos: son más escalables, reutilizables y simples de realizar pruebas. Si los formularios son una parte clave de su aplicación, use formularios reactivos
  - Los formularios basados en plantillas son útiles para agregar un formulario simple a una aplicación, como un formulario de registro de lista de correo electrónico. Son fáciles de agregar a una aplicación, pero no escalan tan bien como los formularios reactivos. Si tiene requisitos con formularios y lógica muy básica que se pueden administrar únicamente en la plantilla, utilice formularios controlados por plantillas

Base común

- Tanto los formularios reactivos como los basados en plantillas comparten bloques de construcción subyacentes

  FormControl rastrea el valor y el estado de validación de un control de formulario individual

  FormGroup rastrea los mismos valores y estado para una colección de controles de formulario

  FormArray rastrea los mismos valores y estado para una matriz de controles de formulario

  ControlValueAccessor crea un puente entre las instancias de `FormControl` de Angular y los elementos DOM nativos

Configuración del modelo de formulario

- Tanto los formularios reactivos como los basados en plantillas utilizan un modelo de formulario para rastrear los cambios de valor entre los formularios de Angular y los elementos de entrada
- Los siguientes ejemplos muestran cómo se define y crea el modelo de formulario

Configuración en formularios reactivos

- Aquí hay un componente con un campo de entrada para un solo control implementado usando formularios reactivos

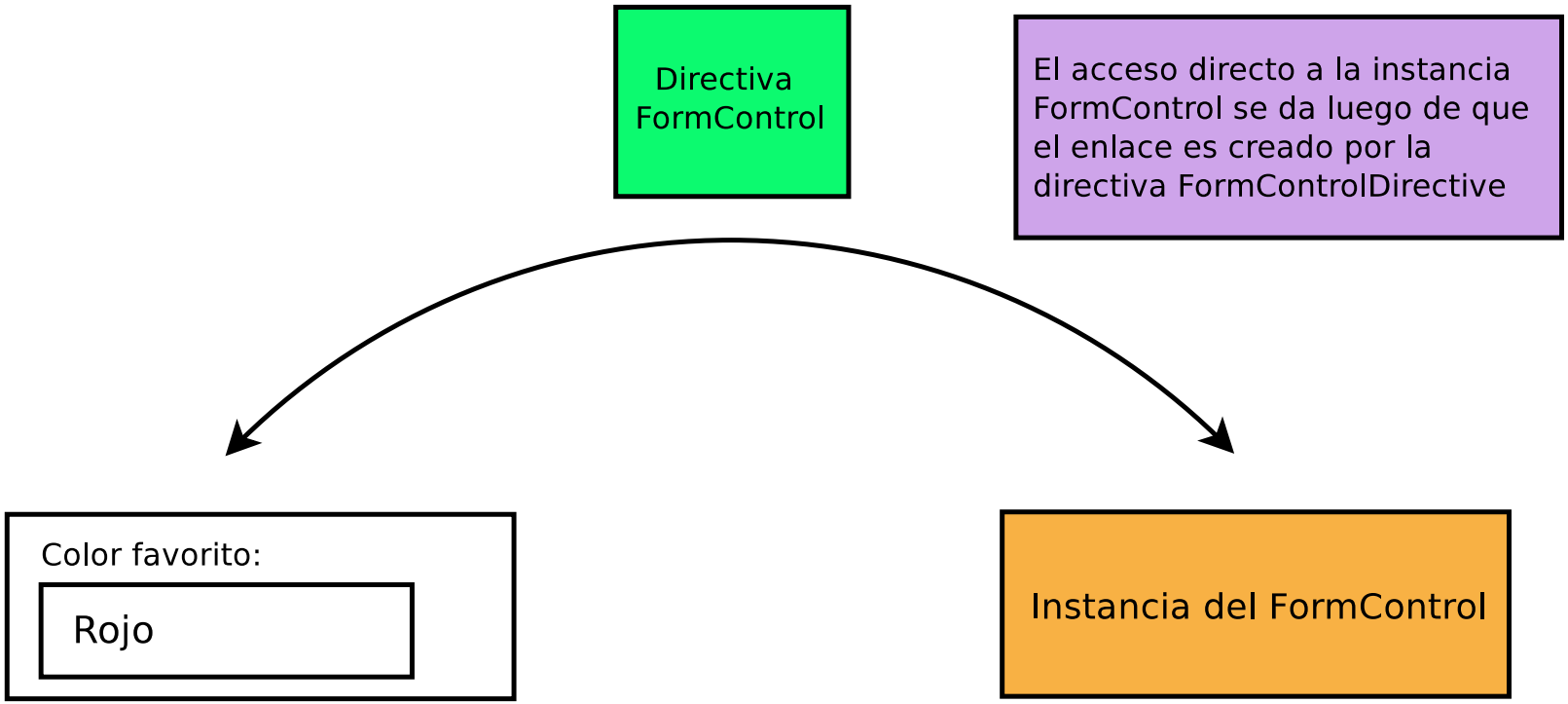
```
import { Component } from '@angular/core';
import { FormControl } from '@angular/forms';

@Component({
  selector: 'app-reactive-favorite-color',
  template: `
    Color favorito:
    <input type="text" [formControl]="favoriteColorControl">
  `
})
export class FavoriteColorComponent {
  favoriteColorControl = new FormControl('');
}
```

Configuración en formularios reactivos cont...

- La *fuente de la verdad* proporciona el valor y el estado del elemento de formulario en un momento dado en el tiempo
- En los formularios reactivos, el modelo del formulario es la *fuente de la verdad*
- En el ejemplo anterior, el modelo de formulario es la instancia de `FormControl`
- Con los formularios reactivos, el modelo de formulario se define explícitamente en la clase de componente
- La directiva de formulario reactivo (en este caso, `FormControlDirective`) vincula la instancia de `FormControl` existente a un elemento de formulario específico en la vista usando un valor de acceso (instancia `ControlValueAccessor`)

Configuración en formas reactivas cont...



Configuración de formularios controlados por plantillas

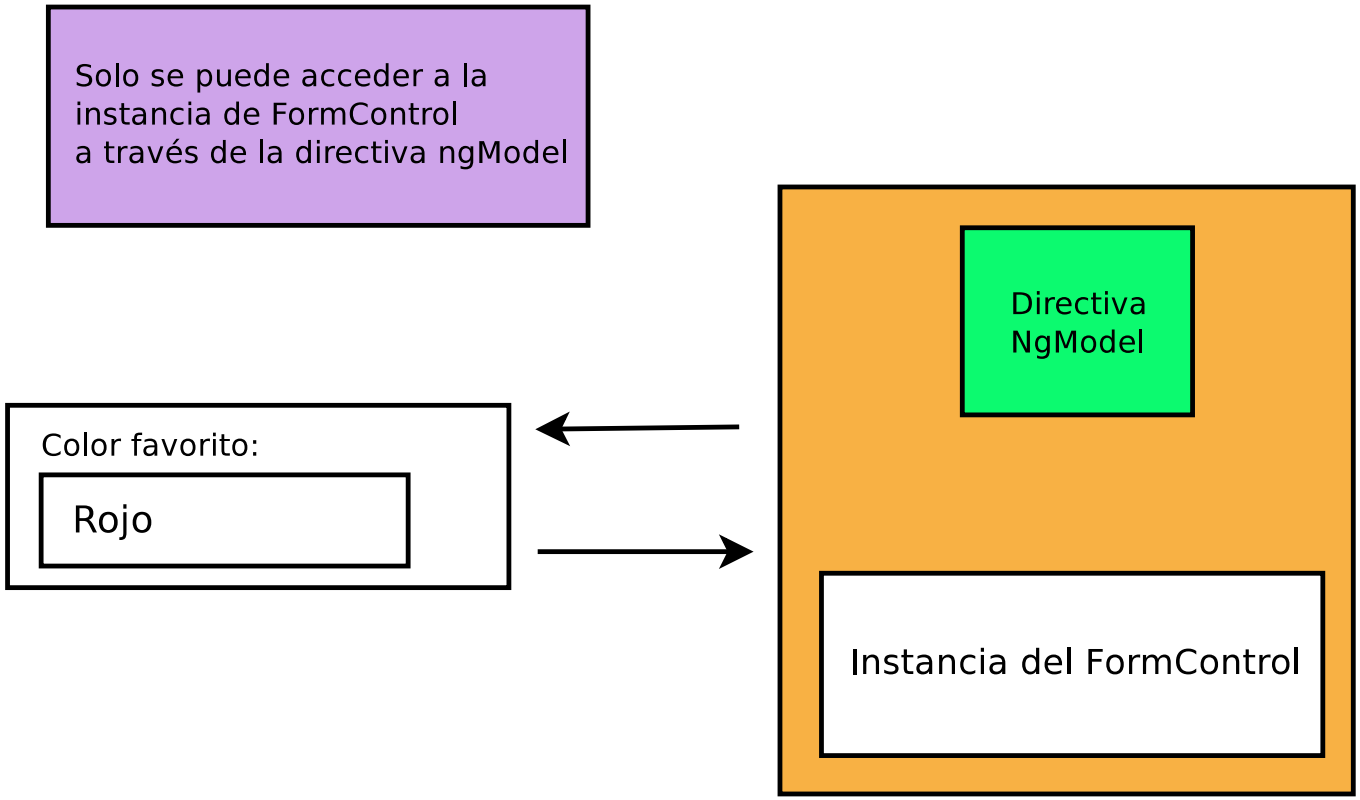
- Este es el mismo componente con un campo de entrada para un solo control implementado utilizando formularios controlados por plantilla

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

@Component({
  selector: 'app-template-favorite-color',
  template: `
    Color favorito:
    <input type="text" [(ngModel)]="favoriteColor">
  `
})
export class FavoriteColorComponent {
  favoriteColor = '';
}
```

Configuración de formularios controlados por plantillas

- En las formas basadas en plantillas, la *fuente de la verdad* es la plantilla



Configuración de formularios controlados por plantillas

- La abstracción del modelo del formulario promueve la simplicidad sobre la estructura
- La directiva del formulario controlado por plantilla `NgModel` es responsable de crear y administrar la instancia de `FormControl` para un elemento de formulario determinado
- Es menos explícito, pero ya no tiene control directo sobre el modelo de formulario

Flujo de datos en formularios

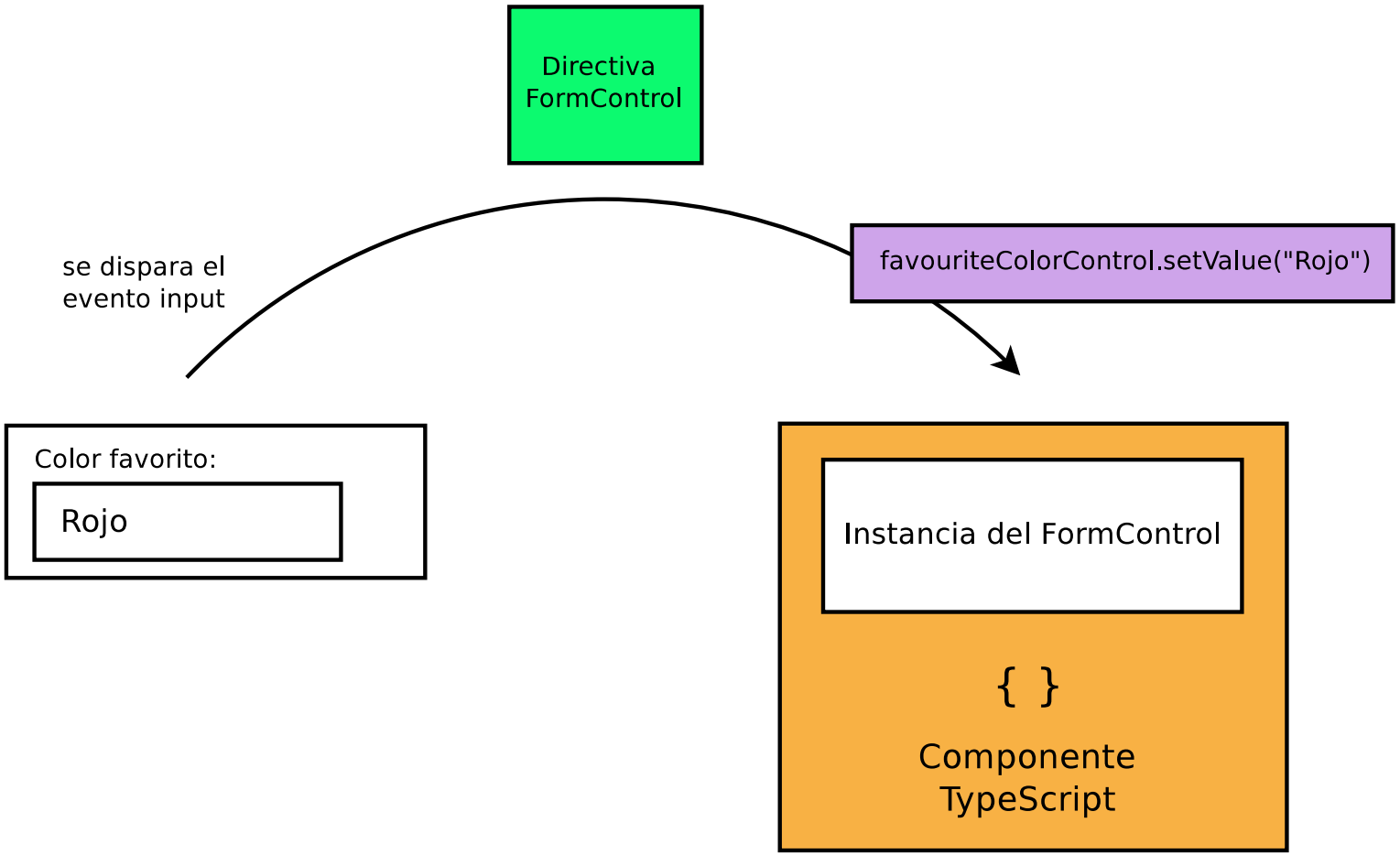
- Al crear formularios en Angular, es importante comprender cómo el framework maneja los datos que fluyen del usuario o de los cambios realizados a través del código
- Los formularios reactivos y los controlados por plantillas siguen dos estrategias diferentes al manejar la entrada de formularios
- Los ejemplos de flujo de datos que se muestran a continuación comienzan con el ejemplo de campo de entrada de color favorito anterior, y luego muestran cómo los cambios en el color favorito se manejan en formularios reactivos en comparación con los formularios controlados por plantilla

Flujo de datos en formularios reactivos

- Como se describió anteriormente, en formularios reactivos, cada elemento de formulario en la vista está directamente vinculado a un modelo de formulario (instancia de `FormControl`)
- Las actualizaciones de la vista al modelo y del modelo a la vista son síncronas y no dependen de la interfaz de usuario representada
- Los diagramas a continuación usan el mismo ejemplo de color favorito para demostrar cómo fluyen los datos cuando se cambia el valor de un campo de entrada desde la vista y luego desde el modelo

Flujo de datos en formularios reactivos cont...

- De la vista al modelo:

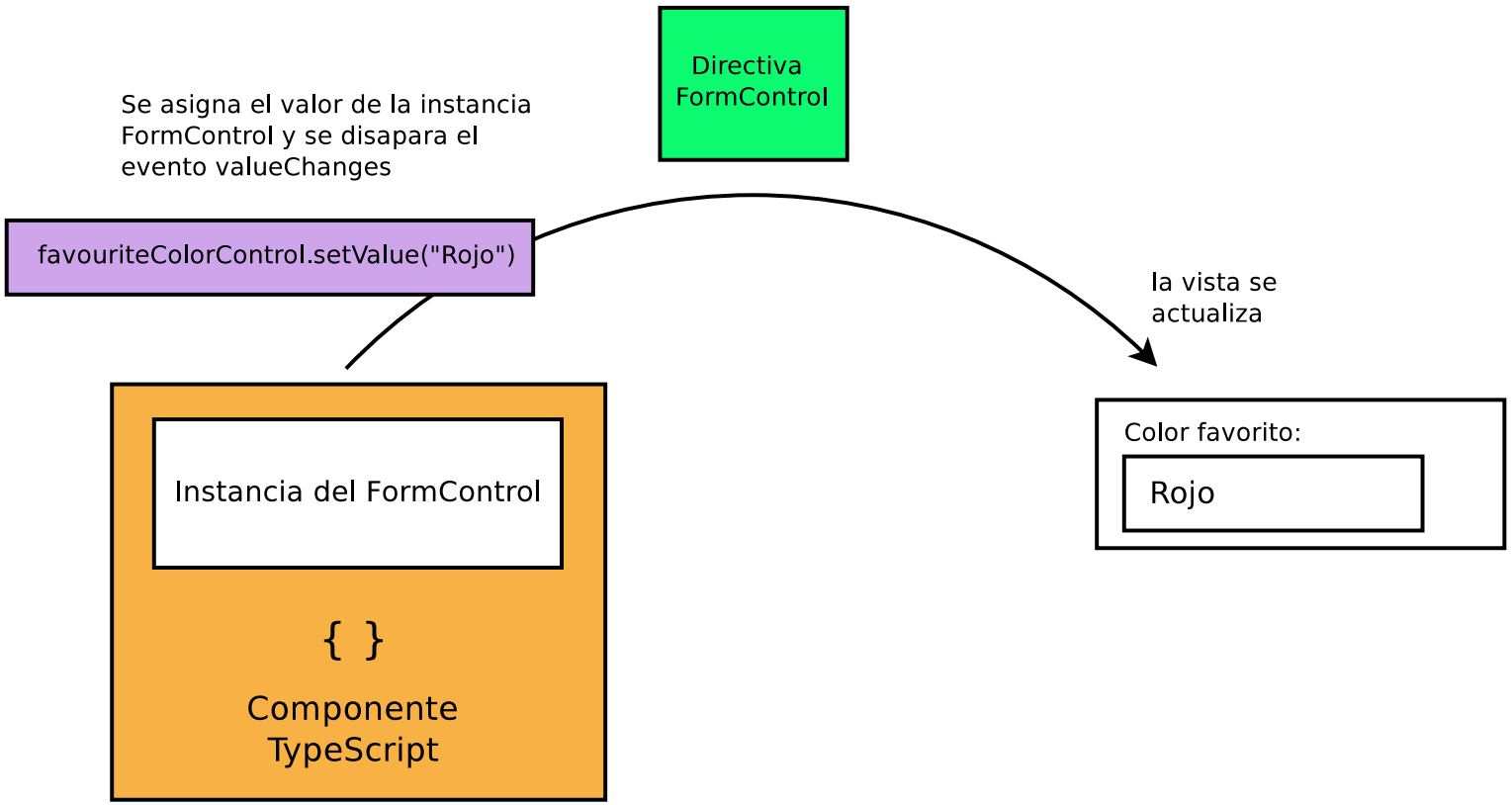


Flujo de datos en formularios reactivos cont...

- Los pasos a continuación describen el flujo de datos de la vista al modelo
  1. El usuario escribe un valor en el elemento de entrada, en este caso, el color favorito Rojo
  2. El elemento de entrada de formulario emite un evento de "entrada" con el último valor
  3. El ControlValueAccessor escucha evntos que se den en el elemento de entrada e inmediatamente retransmite el nuevo valor a la instancia de `FormControl`
  4. La instancia de `FormControl` emite el nuevo valor a través del observable `valueChanges`
  5. Cualquier suscriptor del observable `valueChanges` recibe el nuevo valor

Flujo de datos en formularios reactivos cont...

- Del modelo a la vista:



Flujo de datos en formularios reactivos cont...

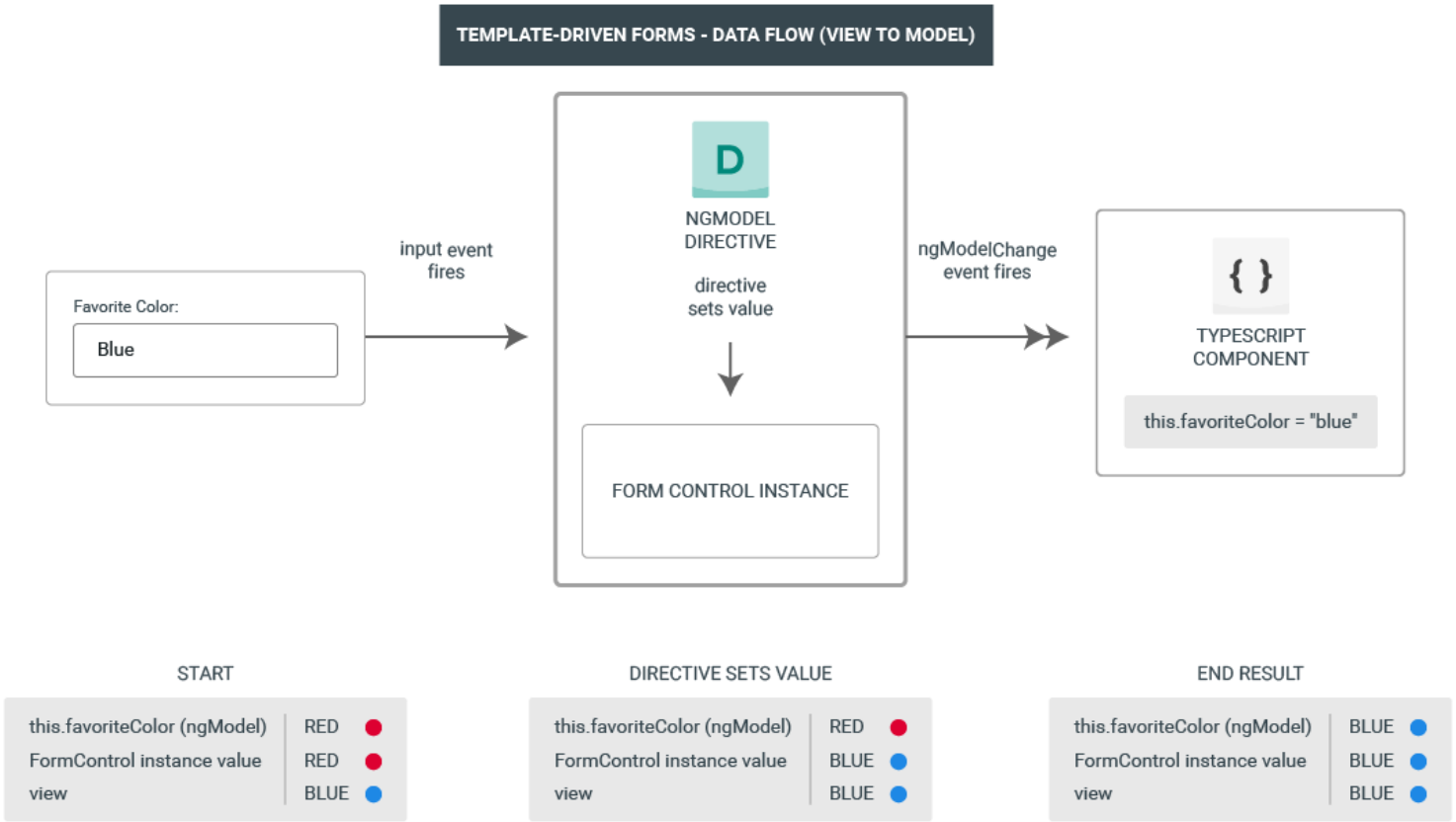
- Los pasos a continuación describen el flujo de datos del modelo a la vista
  1. El usuario llama al método `favoriteColorControl.setValue()`, que actualiza el valor de FormControl
  2. La instancia de `FormControl` emite el nuevo valor a través del observable `valueChanges`
  3. Cualquier suscriptor a los observables de `valueChanges` recibe el nuevo valor
  4. El controlador de valores de acceso en el elemento de entrada de formulario actualiza el elemento con el nuevo valor

Flujo de datos en formularios controlados por plantillas

- En los formularios controlados por plantillas, cada elemento de formulario está vinculado a una directiva que administra el modelo de formulario internamente
- Los diagramas a continuación usan el mismo ejemplo de color favorito para demostrar cómo fluyen los datos cuando se cambia el valor de un campo de entrada desde la vista y luego desde el modelo

Flujo de datos en formularios controlados por plantillas

- De la vista al modelo:

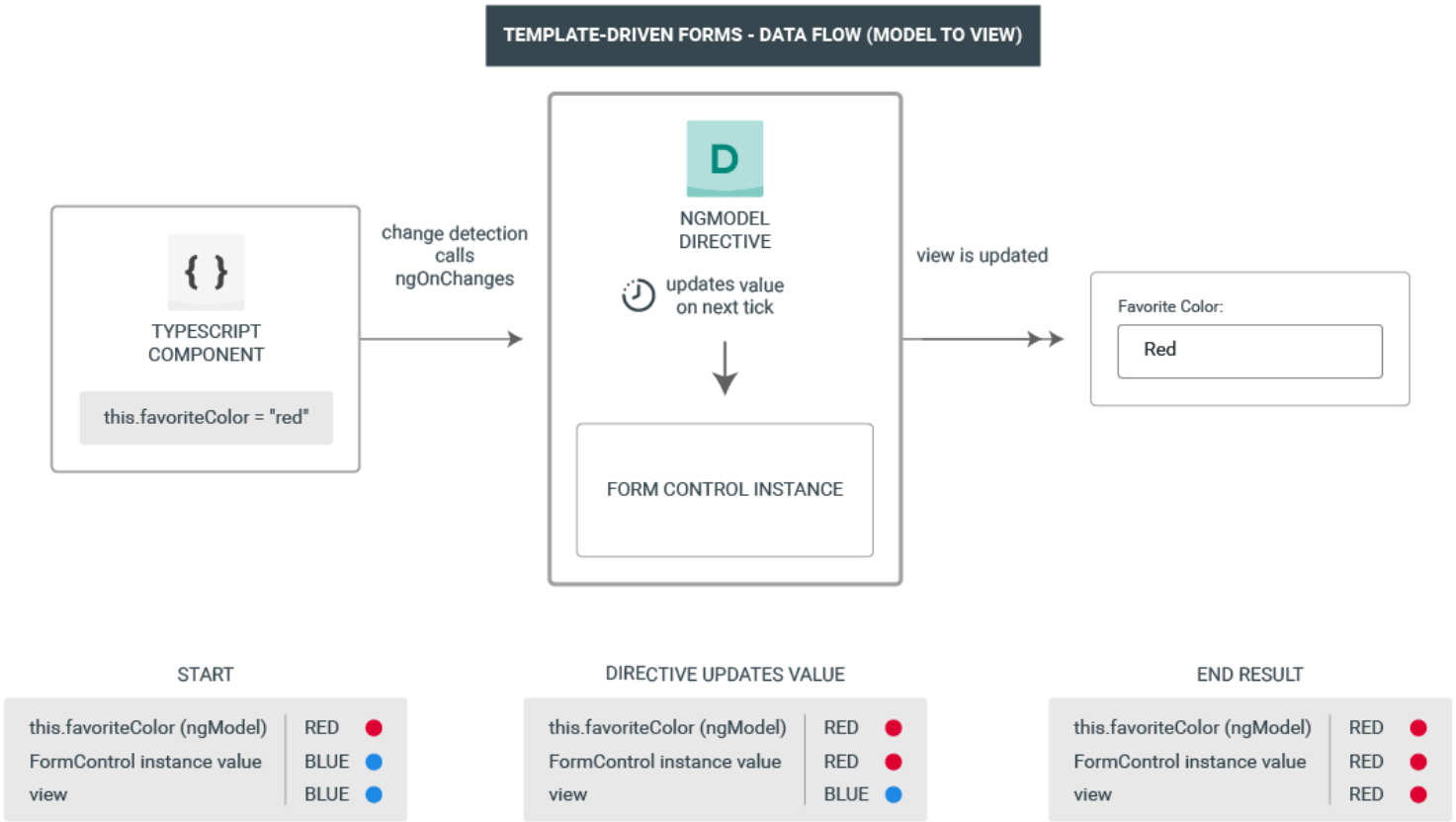


Flujo de datos en formularios controlados por plantillas

- Los pasos a continuación describen el flujo de datos desde la vista al modelo cuando el valor de entrada cambia de Rojo a Azul
  1. El usuario escribe Azul en el elemento de entrada
  2. El elemento de entrada emite un evento de "entrada" con el valor Azul
  3. El valor del control de acceso adjunto a la entrada activa el método `setValue()` en la instancia de `FormControl`
  4. La instancia de `FormControl` emite el nuevo valor a través del observable `valueChanges`
  5. Cualquier suscriptor a los observables de `valueChanges` recibe el nuevo valor
  6. El ControlValueAccessor también llama al método `ngModel.viewToModelUpdate()` que emite un evento `ngModelChange`
  7. Debido a que la plantilla de componente utiliza un enlace bidireccional para la propiedad favoriteColor, la propiedad favoriteColor en el componente se actualiza al valor emitido por el evento `ngModelChange` (Blue)

Flujo de datos en formularios controlados por plantillas

- Del modelo a la vista:



Flujo de datos en formularios controlados por plantillas

- Los pasos a continuación describen el flujo de datos del modelo para ver cuando el color favorito cambia de azul a rojo

1. El valor de favoriteColor se actualiza en el componente
2. Comienza la detección de cambios
3. Durante la detección de cambios, se llama a la función `ngOnChanges` en la instancia de la directiva `NgModel` porque el valor de una de sus entradas ha cambiado
4. El método `ngOnChanges()` pone en cola una tarea asíncrona para establecer el valor de la instancia de `FormControl` interna
5. Se completa la detección de cambios
6. En la siguiente marca, se ejecuta la tarea para establecer el valor de instancia de `FormControl`
7. La instancia de `FormControl` emite el último valor a través de los valores observables de `valueChanges`
8. Cualquier suscriptor a los observables de `valueChanges` recibe el nuevo valor
9. El controlador de valores de acceso actualiza el elemento de entrada de formulario en la vista con el último valor

Mutabilidad

- El método de seguimiento de cambios juega un papel en la eficiencia de su aplicación
  - Los formularios reactivos mantienen el modelo de datos puro al proporcionarlo como una estructura de datos inmutable. Cada vez que se activa un cambio en el modelo de datos, la instancia de `FormControl` devuelve un nuevo modelo de datos en lugar de actualizar el modelo de datos existente. Esto le brinda la capacidad de rastrear cambios únicos en el modelo de datos a través del control observable. Esto proporciona una forma para que la detección de cambios sea más eficiente porque solo necesita actualizarse sobre cambios únicos. También sigue patrones reactivos que se integran con operadores observables para transformar datos.
  - Los formularios controlados por plantillas se basan en la mutabilidad con enlace de datos bidireccional para actualizar el modelo de datos en el componente a medida que se realizan cambios en la plantilla. Debido a que no hay cambios únicos para rastrear en el modelo de datos cuando se usa el enlace de datos bidireccional, la detección de cambios es menos eficiente

Mutabilidad cont...

- La diferencia se demuestra en los ejemplos anteriores utilizando el elemento de entrada de color favorito
  - Con los formularios reactivos, la instancia de `FormControl` siempre devuelve un nuevo valor cuando se actualiza el valor del control
  - Con formularios controlados por plantillas, la propiedad de color favorita siempre se modifica a su nuevo valor

Formularios reactivos

- Registrando el modulo

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

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

Formularios reactivos cont...

- Generando el componente

```
import { Component } from '@angular/core';
import { FormControl } from '@angular/forms';

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

Formularios reactivos cont...

- Registrando el componente en la plantilla

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

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

Formularios reactivos cont...

- Reemplazando el valor desde el código

```
 updateName() {
   this.name.setValue('Nancy');
 }

...

<p>
  <button (click)="updateName()">Update Name</button>
</p>
```

Formularios reactivos cont...

- Agrupando controles de formularios

```
import { Component } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';

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

Formularios reactivos cont...

- Asociando el FormGroup y la vista

```
<form [formGroup]="profileForm">

  <label>
    First Name:
    <input type="text" formControlName="firstName">
  </label>

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

</form>
```

Formularios reactivos cont...

- Almacenando los datos

```
<form [formGroup]="profileForm" (ngSubmit)="onSubmit()">

...

onSubmit() {
  console.info(this.profileForm.value);
}

...

<button type="submit" [disabled]="!profileForm.valid">
  Submit
</button>
```

Formularios reactivos cont...

- Creando grupos anidados

```
import { Component } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';

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

Formularios reactivos cont...

- Creando la plantilla para grupos anidados

```
<div formGroupName="address">
  <h3>Address</h3>

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

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

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

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

Formularios reactivos cont...

- Cambiando los valores del modelo

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

...

<p>
  <button (click)="updateProfile()">
    Update Profile
  </button>
</p>
```

Formularios reactivos cont...

- Generando controles utilizando FormBuilder

```
import { Component } from '@angular/core';
import { FormBuilder } from '@angular/forms';

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

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

Formularios reactivos cont...

- Validación simple

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

...

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

Formularios basados en plantillas

- Generación del modelo

```
export class Hero {

  constructor(
    public id: number,
    public name: string,
    public power: string,
    public alterEgo?: string
  ) {  }

}

...

let myHero =  new Hero(42, 'SkyDog',
   'Fetch any object at any distance', 'Leslie Rollover');
console.log('My hero is called ' + myHero.name);
```

Formularios basados en plantillas cont...

- Generación del componente

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

import { Hero }    from '../hero';

@Component({
  selector: 'app-hero-form',
  templateUrl: './hero-form.component.html',
  styleUrls: ['./hero-form.component.css']
})
export class HeroFormComponent {

  powers = ['Really Smart', 'Super Flexible',
    'Super Hot', 'Weather Changer'];

  model = new Hero(18, 'Dr IQ', this.powers[0], 'Chuck Overstreet');

  submitted = false;

  onSubmit() { this.submitted = true; }

  // TODO: Remove this when we're done
  get diagnostic() { return JSON.stringify(this.model); }
}
```

Formularios basados en plantillas cont...

- Creando un plantilla inicial

```
<div class="container">
  <h1>Hero Form</h1>
  <form>
    <div class="form-group">
      <label for="name">Name</label>
      <input type="text" class="form-control" id="name"
             required>
    </div>

    <div class="form-group">
      <label for="alterEgo">Alter Ego</label>
      <input type="text" class="form-control" id="alterEgo">
    </div>

    <button type="submit" class="btn btn-success">
      Submit
    </button>
  </form>
</div>
```

Formularios basados en plantillas cont...

- Enlace bidireccional con ngModel

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

...

<input type="text" class="form-control" id="name"
       required
       [(ngModel)]="model.name" name="name">
<p>{{model.name}}</p>
```

Formularios basados en plantillas cont...

- Clases asosicadas al estado del componente
- El control fue visitado: `ng-touched`, `ng-untouched`
- El valor del control cambió: `ng-dirty`, `ng-pristine`
- El valor del control es válido: `ng-valid`, `ng-invalid`

Formularios basados en plantillas cont...

- Mostrando mensajes de error

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

...

<div [hidden]="name.valid || name.pristine"
     class="alert alert-danger">
```

Formularios basados en plantillas cont...

- Enviando datos con ngSubmit

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

...

<button type="submit" class="btn btn-success"
        [disabled]="!heroForm.form.valid">
   Submit
</button>
```

Formularios basados en plantillas cont...

- Validaciones

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

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

  <div *ngIf="name.errors.required">
     Name is required.
  </div>
  <div *ngIf="name.errors.minlength">
     Name must be at least 4 characters long.
  </div>
</div>
```

Formularios basados en plantillas cont...

- Validaciones personalizadas

```
import { FormControl } from '@angular/forms'

export function miValidacion(control: FormControl):
       {[key: string]: any} {
   // ... implementación de la validación personalizada
}
```

Laboratorio práctico - Formularios en Angular

Es el momento de trabajar con formularios

- Creacion de un formulario basado en plantillas
- Validar un formulario basado en plantillas
- Creando un formulario reactivo
- Generando formularios utilizando FormBuilder
- Validando un formulario reactivo
- Creando un validador personalizado

Observables

Luciano Diamand

Introducción a Observables

Observables

- Los observables proporcionan soporte para pasar mensajes entre editores y suscriptores en su aplicación
- Los observables ofrecen beneficios significativos sobre otras técnicas para el manejo de eventos, la programación asíncrona y el manejo de múltiples valores
- Los observables son declarativos, es decir, se define una función para la publicación de valores, pero no se ejecuta hasta que un consumidor se suscribe a ella
- El consumidor suscrito recibe notificaciones hasta que la función se completa o hasta que se da de baja

Observables cont...

- Un observable puede entregar múltiples valores de cualquier tipo: literales, mensajes o eventos, según el contexto
- Debido a que, tanto la lógica de configuración como la de desmontaje son manejadas por lo observable, el código de su aplicación solo debe preocuparse por suscribirse para consumir valores, y cuando termina, cancelar la suscripción

Uso básico y términos

- Como editor, se crea una instancia observable que define una función de suscriptor
- Esta es la función que se ejecuta cuando un consumidor llama al método `subscribe()`
- La función de suscriptor define cómo obtener o generar valores o mensajes para ser publicados
- Para ejecutar el observable que se ha creado y comenzar a recibir notificaciones, se debe llamar a su método `subscribe()`, pasando un observador
- Este es un objeto de JavaScript que define los controladores (funciones *callback*) para las notificaciones que recibe
- La llamada `subscribe()` devuelve un objeto `Subscription` que tiene un método `unsubscribe()` al que llama para dejar de recibir notificaciones

Uso básico y términos cont...

```
// Create an Observable that will start listening to geolocation updates
// when a consumer subscribes.
const locations = new Observable((observer) => {
  // Get the next and error callbacks. These will be passed in when
  // the consumer subscribes.
  const {next, error} = observer;
  let watchId;

  // Simple geolocation API check provides values to publish
  if ('geolocation' in navigator) {
    watchId = navigator.geolocation.watchPosition(next, error);
  } else {
    error('Geolocation not available');
  }

  // When the consumer unsubscribes, clean up data ready for next subscription.
  return {unsubscribe() { navigator.geolocation.clearWatch(watchId); }};
});

// Call subscribe() to start listening for updates.
const locationsSubscription = locations.subscribe({
  next(position) { console.log('Current Position: ', position); },
  error(msg) { console.log('Error Getting Location: ', msg); }
});

// Stop listening for location after 10 seconds
setTimeout(() => { locationsSubscription.unsubscribe(); }, 10000);
```

Definiendo observadores

- Un controlador para recibir notificaciones observables implementa la interfaz Observer
- Es un objeto que define los métodos *callback* para manejar los tres tipos de notificaciones que un observable puede enviar:

  next Invocado cero o más veces después de que comience la ejecución (requerido)

  error Un controlador para una notificación de error. Un error detiene la ejecución de la instancia observable (opcional)

  complete Un controlador para la notificación de ejecución completa. (opcional)
- Un objeto observador puede definir cualquier combinación de estos controladores
- Si no proporciona un controlador para un tipo de notificación, el observador ignora las notificaciones de ese tipo

Suscribiendo

- Una instancia observable comienza a publicar valores solo cuando alguien se suscribe a ella
- Usted se suscribe llamando al método `subscribe()` de la instancia, pasando un objeto observador para recibir las notificaciones
- podemos usar algunos métodos de la biblioteca RxJS que crean observables simples de los tipos utilizados con frecuencia:
  - `of (...items)`: devuelve una instancia observable que entrega de forma sincrónica los valores proporcionados como argumentos
  - `from (iterable)`: convierte su argumento en una instancia observable. Este método se usa comúnmente para convertir una matriz en un observable

Suscribiendo cont...

```
  // Create simple observable that emits three values
const myObservable = of(1, 2, 3);

// Create observer object
const myObserver = {
  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'),
};

// Execute with the observer object
myObservable.subscribe(myObserver);
// Logs:
// Observer got a next value: 1
// Observer got a next value: 2
// Observer got a next value: 3
// Observer got a complete notification
```

Suscribiendo cont...

- De forma alternativa, el método `subscribe()` puede aceptar definiciones de funciones *callback* en línea, para `next`, `error` y `complete`

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

Suscribiendo cont...

- En cualquier caso, se requiere un controlador `next`
- Los controladores `error` y `complete` son opcionales
- Tenga en cuenta que una función `next()` podría recibir, por ejemplo, cadenas de mensajes u objetos de eventos, valores numéricos o estructuras, según el contexto
- Como término general, nos referimos a los datos publicados por un observable como un flujo (*stream*)

Creando observables

- Utilice el constructor Observable para crear un flujo observable de cualquier tipo
- El constructor toma como argumento la función suscriptor para ejecutarse cuando se ejecuta el método `subscribe()` del observable
- Una función de suscriptor recibe un objeto `Observer` y puede publicar valores en el método `next()` del observador

Creando observables cont...

```
// This function runs when subscribe() is called
function sequenceSubscriber(observer) {
  // synchronously deliver 1, 2, and 3, then complete
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.complete();

  // unsubscribe function doesn't need to do anything
  // in this because values are delivered synchronously
  return {unsubscribe() {}};
}

// Create a new Observable that will deliver the above
// sequence
const sequence = new Observable(sequenceSubscriber);
```

Creando observables cont...

```
// execute the Observable and print the result of
// each notification
sequence.subscribe({
  next(num) { console.log(num); },
  complete() { console.log('Finished sequence'); }
});

// Logs:
// 1
// 2
// 3
// Finished sequence
```

Manejo de errores

- Debido a que los observables producen valores de forma asíncrona, `try/catch` no detectará los errores de manera efectiva
- En su lugar, hay que manejar los errores especificando una función *callback* de error en el observador
- La producción de un error también hace que el observable limpie las suscripciones y deje de generar valores
- Un observable puede producir valores (llamando a `next`), o puede completar, llamando a `complete` o `error`

```
myObservable.subscribe({
  next(num) { console.log('Next num: ' + num)},
  error(err) { console.log('Received an errror: ' + err)}
});
```

La biblioteca RxJS

La biblioteca RxJS

- La programación reactiva es un paradigma de programación asíncrono relacionado con los flujos de datos y la propagación del cambio
- RxJS (Extensiones reactivas para JavaScript) es una biblioteca para programación reactiva que utiliza elementos observables que facilita la composición de código asíncrono o basado en funciones *callback*
- RxJS proporciona una implementación del tipo `Observable`, que es necesaria hasta que el tipo se convierta en parte del idioma o hasta que los navegadores lo admitan
- La biblioteca también proporciona funciones de utilidad para crear y trabajar con observables

La biblioteca RxJS cont...

- Las funciones de utilidad disponibles se pueden utilizar para:
  - Convertir el código existente para operaciones asíncronas en observables
  - Iterar a través de los valores en un flujo
  - Filtrando de flujos
  - Composición de multiples flujos

Funciones de creación de observables

- RxJS ofrece una serie de funciones que se pueden usar para crear nuevos observables
- Estas funciones pueden simplificar el proceso de creación de elementos observables a partir de eventos, temporizadores, promesas, etc

```
import { fromPromise } from 'rxjs';

// Create an Observable out of a promise
const data = fromPromise(fetch('/api/endpoint'));
// Subscribe to begin listening for async result
data.subscribe({
 next(response) { console.log(response); },
 error(err) { console.error('Error: ' + err); },
 complete() { console.log('Completed'); }
});
```

Funciones de creación de observables cont...

```
  import { interval } from 'rxjs';

// Create an Observable that will publish a value on
// an interval
const secondsCounter = interval(1000);

// Subscribe to begin publishing values
secondsCounter.subscribe(n =>
  console.log(`It's been ${n} seconds since subscribing!`));
```

Funciones de creación de observables cont...

```
import { ajax } from 'rxjs/ajax';

// Create an Observable that will create an AJAX request
const apiData = ajax('/api/data');
// Subscribe to create the request
apiData.subscribe(res =>
    console.log(res.status, res.response));
```

Operadores

- Los operadores son funciones que se basan en los observables para permitir la manipulación sofisticada de colecciones
- Por ejemplo, RxJS define operadores como `map()`, `filter()`, `concat()` y `flatMap()`
- Los operadores toman las opciones de configuración y devuelven una función que toma una fuente observable
- Al ejecutar la función retornada, el operador observa los valores emitidos del observable de origen, los transforma y devuelve un nuevo observable de esos valores transformados

Operadores cont...

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

const nums = of(1, 2, 3);

const squareValues = map((val: number) => val * val);
const squaredNums = squareValues(nums);

squaredNums.subscribe(x => console.log(x));

// Logs
// 1
// 4
// 9
```

Operadores cont...

- Es posible utilizar tuberías para unir a los operadores
- Las tuberías permiten combinar múltiples funciones en una sola función
- La función `pipe()` toma como argumentos las funciones que desea combinar y devuelve una nueva función que, cuando se ejecuta, ejecuta las funciones compuestas en secuencia
- Un conjunto de operadores aplicado a un observable es una receta, es decir, un conjunto de instrucciones para producir los valores que le interesan

Operadores cont...

```
  import { filter, map } from 'rxjs/operators';

const nums = of(1, 2, 3, 4, 5);

// Create a function that accepts an Observable.
const squareOddVals = pipe(
  filter((n: number) => n % 2 !== 0),
  map(n => n * n)
);

// Create an Observable that will run the filter and
// map functions
const squareOdd = squareOddVals(nums);

// Suscribe to run the combined functions
squareOdd.subscribe(x => console.log(x));
```

Operadores cont...

- La función `pipe()` también es un método en el RxJS Observable, por lo que utiliza esta forma más corta para definir la misma operación:

```
import { filter, map } from 'rxjs/operators';

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

// Subscribe to get values
squareOdd.subscribe(x => console.log(x));
```

Manejo de errores

- Además del controlador de `error()` que proporciona en la suscripción, RxJS proporciona el operador `catchError` que le permite manejar los errores conocidos en la receta observable
- Por ejemplo, suponga que tiene un observable que realiza una solicitud de API y se asigna a la respuesta del servidor. Si el servidor devuelve un error o el valor no existe, se produce un error. Si detecta este error y proporciona un valor predeterminado, su flujo continúa procesando los valores en lugar de cometer errores

Manejo de errores cont...

```
import { ajax } from 'rxjs/ajax';
import { map, catchError } from 'rxjs/operators';
// Return "response" from the API. If an error happens,
// return an empty array.
const apiData = ajax('/api/data').pipe(
   map(res => {
     if (!res.response) {
       throw new Error('Value expected!');
     }
     return res.response;
   }),
   catchError(err => of([]))
);

apiData.subscribe({
   next(x) { console.log('data: ', x); },
   error(err) {
     console.log('errors already caught... will not run');
   }
});
```

Reintento fallido observable

- Cuando el operador `catchError` proporciona una ruta de recuperación simple, el operador `retry` permite reintentar una solicitud fallida
- Utilice el operador `retry` antes del operador `catchError`
- El mismo, vuelve a suscribir al observable original, que luego puede volver a ejecutar la secuencia completa de acciones que dieron como resultado el error
- Si esto incluye una solicitud HTTP, reintentará esa solicitud HTTP

Reintento fallido observable cont...

```
import { ajax } from 'rxjs/ajax';
import { map, retry, catchError } from 'rxjs/operators';

const apiData = ajax('/api/data').pipe(
   retry(3), // Retry up to 3 times before failing
   map(res => {
     if (!res.response) {
       throw new Error('Value expected!');
     }
     return res.response;
   }),
   catchError(err => of([]))
 );

 apiData.subscribe({
   next(x) { console.log('data: ', x); },
   error(err) {
     console.log('errors already caught... will not run');
   }
 });
```

HttpClient

Luciano Diamand

Introducción a HttpClient

HttpClient

- La mayoría de las aplicaciones de *front-end* se comunican con servicios de *back-end* a través del protocolo HTTP
- Los navegadores modernos admiten dos API diferentes para realizar solicitudes HTTP
  - la interfaz `XMLHttpRequest`
  - la API `fetch()`
- El `HttpClient` de `@angular/common/http` ofrece una API HTTP cliente simplificada para aplicaciones Angular que se basa en la interfaz `XMLHttpRequest` expuesta por los navegadores
- Los beneficios adicionales de `HttpClient` incluyen características de capacidad de prueba, objetos de solicitud y respuesta tipificados, intercepción de solicitud y respuesta, Apis observables y manejo simplificado de errores

Configuración

- Para poder utilizar `HttpClient`, necesita importar el módulo `HttpClientModule` de Angular
- La mayoría de las aplicaciones lo hacen en el `AppModule` raíz

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

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

Configuración cont...

- Habiendo importado `HttpClientModule` en `AppModule`, puede inyectar `HttpClient` en una clase de servicio como se muestra en el siguiente ejemplo de `ConfigService`

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

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

Obteniendo datos JSON

- Un servicio puede recuperar datos desde el servidor a través del método `get()` de `HttpClient`

```
configUrl = 'assets/config.json';

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

Obteniendo datos JSON cont...

- Luego, en un componente, injectamos el servicio y podemos utilizar el método

```
showConfig() {
  this.configService.getConfig()
      .subscribe((data: any) => this.config = {
    heroesUrl: data['heroesUrl'],
    textfile: data['textfile']
  });
}
```

- Debido a que el método de servicio devuelve un `Observable` de datos de configuración, el componente se suscribe al valor de retorno del método
- La función de *callback* de suscripción copia los campos de datos en el objeto de configuración del componente, que está enlazado a los datos en la plantilla del componente para su visualización

Obteniendo datos JSON cont...

- Es posible decirle a `HttpClient` el tipo de respuesta para que la consulta de la salida sea más fácil y más obvia

```
export interface Config {
  heroesUrl: string;
  textfile: string;
}

...

getConfig() {
  // now returns an Observable of Config
  return this.http.get<Config>(this.configUrl);
}
```

Obteniendo datos JSON cont...

- La función *callback* en el método del componente actualizado recibe un objeto de datos con tipo, que es más fácil y más seguro de consumir

```
config: Config;

showConfig() {
  this.configService.getConfig()
    // clone the data object, using its known Config shape
    .subscribe((data: Config) => this.config = { ...data });
}
```

Leyendo la respuesta completa

- El cuerpo de la respuesta no devuelve todos los datos que pueda necesitar
- A veces, los servidores devuelven encabezados especiales o códigos de estado para indicar ciertas condiciones que son importantes para el flujo de trabajo de la aplicación
- Se puede parar a `HttpClient` la opción `observe` para especificar que se desea la respuesta completa

```
getConfigResponse(): Observable<HttpResponse<Config>> {
  return this.http.get<Config>(
    this.configUrl, { observe: 'response' }
  );
}
```

Leyendo la respuesta completa cont...

- El método `showConfigResponse()` del componente muestra los encabezados de respuesta, así como la configuración:

```
showConfigResponse() {
  this.configService.getConfigResponse()
     // resp is of type `HttpResponse<Config>`
     .subscribe(resp => {
        // display its headers
        const keys = resp.headers.keys();
        this.headers = keys.map(key =>
           `${key}: ${resp.headers.get(key)}`);

     // access the body directly, which is typed as `Config`.
  this.config = { ...resp.body };
  });
}
```

Manejo de errores

- ¿Qué sucede si la solicitud falla en el servidor o si una conexión de red deficiente impide que llegue al servidor?
- `HttpClient` devolverá un objeto de *error* en lugar de una respuesta exitosa
- Es posible manejarlo en el componente agregando una segunda función de *callback* a `.subscribe()`

```
showConfig() {
  this.configService.getConfig()
     .subscribe(
        (data: Config) =>
           this.config = { ...data }, // success path
        error =>
           this.error = error // error path
     );
}
```

Obteniendo detalles del error

- Pueden ocurrir dos tipos de errores:
  - El servidor de *back-end* puede rechazar la solicitud y devolver una respuesta HTTP con un código de estado como 404 o 500. Estas son respuestas de error
  - O tal vez algo podría salir mal del lado del cliente, como un error de red que impide que la solicitud se complete con éxito o que se produzca una excepción en un operador de RxJS. Estos errores producen objetos JavaScript `ErrorEvent`
- El `HttpClient` captura ambos tipos de errores en su `HttpErrorResponse` y puede inspeccionar esa respuesta para descubrir qué sucedió realmente
- La inspección de errores, la interpretación y la resolución es algo que se realiza en el servicio y no en el componente

Manejo de errores

- ¿Qué sucede si la solicitud falla en el servidor o si una conexión de red deficiente impide que llegue al servidor? `HttpClient` devolverá un objeto de error en lugar de una respuesta exitosa
- Puede manejar el componente agregando una segunda función de *callback* a `.subscribe()`

```
private handleError(error: HttpErrorResponse) {
  if (error.error instanceof ErrorEvent) {
      // A client-side or network error occurred. Handle it accordingly.
      console.error('An error occurred:',
      error.error.message);
  } else {
      // The backend returned an unsuccessful response code.
      // The response body may contain clues as to what went wrong,
      console.error(`Backend returned code ${error.status}, ` +
       `body was: ${error.error}`);
  }
      // return an observable with a user-facing error message
    return throwError(
      'Something bad happened; please try again later.');
};
```

Manejo de errores cont...

- Tenga en cuenta que este controlador devuelve un `ErrorObservable` de RxJS con un mensaje de error fácil de usar
- Los consumidores del servicio esperan que los métodos de servicio devuelvan un `Observable` de algún tipo, incluso uno *"malo"*

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

retry()

- A veces el error es transitorio y desaparecerá automáticamente si se intenta nuevamente
- Por ejemplo, las interrupciones de la red son comunes en los escenarios móviles, y volver a intentarlo puede producir un resultado exitoso
- La biblioteca RxJS ofrece varios operadores `retry`
- El más simple se llama retry () y se vuelve a suscribir automáticamente a un observable fallido un número específico de veces
- Volver a suscribirse al resultado de una llamada a un método de `HttpClient` tiene el efecto de volver a emitir la solicitud HTTP
- Inserte el `retry` justo antes del controlador de errores

retry()

```
getConfig() {
  return this.http.get<Config>(this.configUrl)
      .pipe(
        retry(3), // retry a failed request up to 3 times
        catchError(this.handleError) // then handle the error
    );
}
```

Solicitando datos no JSON

- No todas las API devuelven datos JSON

```
getTextFile(filename: string) {
  // The Observable returned by get() is of type Observable<string>
  // because a text response was specified.
  // There's no need to pass a <string> type parameter to get().
  return this.http.get(filename, {responseType: 'text'})
     .pipe(
        tap( // Log the result or error
           data => this.log(filename, data),
           error => this.logError(filename, error)
     )
  );
}
```

- `HttpClient.get()` devuelve una cadena en lugar del JSON predeterminado debido a la opción `responseType`
- El operador `tap` de RxJS permite que el código inspeccione los valores que pasan por lo observable sin alterarlos

Enviando datos al servidor

- Además de recuperar datos del servidor, `HttpClient` admite solicitudes de actualización, es decir, el envío de datos al servidor con otros métodos HTTP como **PUT**, **POST** y **DELETE**

Agregando encabezados

- Muchos servidores requieren encabezados adicionales para operaciones de guardado
- Por ejemplo, pueden requerir un encabezado *"Tipo de contenido"* para declarar explícitamente el tipo MIME del cuerpo de la solicitud
- Otro ejemplo sería que el servidor requiere un token de autorización

```
import { HttpHeaders } from '@angular/common/http';

const httpOptions = {
  headers: new HttpHeaders({
    'Content-Type':  'application/json',
    'Authorization': 'my-auth-token'
  })
};
```

Hacer una solicitud POST

- Las aplicaciones a menudo publican datos en un servidor
- Hacen POST al enviar un formulario
- El método `HttpClient.post()` es similar a `get()` porque tiene un parámetro de tipo (se espera que el servidor devuelva el objeto) y tome una URL de recursos
- Se necesitan dos parámetros más:
  1. los datos a POST en el cuerpo de la solicitud
  2. `httpOptions`: las opciones de método que, en este caso, especifican los encabezados requeridos

Hacer una solicitud POST cont...

```
/** POST: add a new hero to the database */
addHero (hero: Hero): Observable<Hero> {
  return this.http.post<Hero>(this.heroesUrl, hero, httpOptions)
    .pipe(
        catchError(this.handleError('addHero', hero));
    );
}

...

this.heroesService.addHero(newHero)
  .subscribe(hero => this.heroes.push(hero));
```

Hacer una solicitud de DELETE

```
/** DELETE: delete the hero from the server */
deleteHero (id: number): Observable<{}> {
  \\ DELETE api/heroes/42
  const url = `${this.heroesUrl}/${id}`;
    return this.http.delete(url, httpOptions)
      .pipe(
        catchError(this.handleError('deleteHero'))
    );
}

...

this.heroesService.deleteHero(hero.id).subscribe();
```

Hacer una solicitud de PUT

```
/**
 * PUT: update the hero on the server.
 * Returns the updated hero upon success.
 */
updateHero (hero: Hero): Observable<Hero> {
  return this.http.put<Hero>(this.heroesUrl, hero, httpOptions)
    .pipe(
      catchError(this.handleError('updateHero', hero))
  );
}
```

Laboratorio práctico - Comunicación con el servidor

Es hora de contactar al servidor

- Convirtiendo un observable en una promesa
- Realizar una solicitud `GET` de HTTP
- Tratar con flujos de datos
- Usando Parámetros de Cadena de Consulta

Pruebas unitarias

Luciano Diamand

Inicio rápido

Introducción

- Código mal escrito, una funcionalidad defectuosa y malas prácticas de refactorización pueden llevar a aplicaciones poco confiables
- Escribir buenas pruebas ayuda a detectar este tipo de problemas y evita que afecten negativamente a la aplicación
- Angular fue escrito desde cero para ser comprobable
- Las pruebas en Angular son sencillas y formales

Tipos de Pruebas

- Las pruebas se pueden dividir en tres categorías diferentes:
  1. **Pruebas de extremo a extremo**: son pruebas en las que se desea imitar a un usuario real que visita su sitio web. Cada prueba contiene una serie de eventos de usuario simulados
  2. **Pruebas de integración**: Normalmente, cada prueba se centrará en una función. La prueba llama a la función de destino con un conjunto de parámetros y luego verifica que los resultados coincidan con los valores esperados
  3. **Pruebas unitarias**: son las mismas que las pruebas de integración, excepto que toman medidas adicionales para asegurarse de que sean independientes del contexto
- **Protractor** es la herramienta que se utiliza para las pruebas de extremo a extremo, mientras que **Karma** maneja las de integración y las pruebas unitarias

Tipos de Pruebas

- Algunos ejemplos de estas pruebas son:
  1. **Pruebas de extremo a extremo**: Por ejemplo, se simula un usuario que vaya a la `http://mysite.com/home` y haga clic en el botón con el ID 'mi-botón'). Luego, se verifica cuales son los resultados esperados (por ejemplo, después de 200 ms aparecerá una nueva ventana que dice *"gracias"*)
  2. **Pruebas de integración**: Por ejemplo, cuando se prueba un servicio que usa `HttpClient` para llamar a una API de back-end. La prueba incluiría la llamada a la API
  3. **Pruebas unitarias**: Por ejemplo, cuando prueba un servicio que usa `HttpClient` para llamar a una API de back-end. La misma usaría un `Mock` para reemplazar a `HttpClient`, de modo que el código ejecutado por la prueba esté restringido solo a la función de destino

Escribiendo pruebas con Jasmine

- Jasmine es un framework de desarrollo guiado por comportamiento (BDD) muy utilizado a la hora de probar aplicaciones en JavaScript
- BDD es una metodología que permite describir en un formato entendible la prueba a realizar y de esa forma personas no tecnicas pueden comprendan de que se trata

Escribiendo pruebas con Jasmine cont...

- Las funciones esenciales de Jasmine son:
  - **describe**: Se utiliza para agrupar una serie de pruebas. Este grupo de pruebas es conocido como una *suite* de prueba

    ```
    describe('cadena describiendo las pruebas', callback);
    ```

    Se pueden tener tantas funciones `describe` como desee. El número de funciones `describe` depende de cómo desea organizar las pruebas. También es posible anidar tantas funciones `describe` como desee
  - **it**: Se utiliza para crear una prueba específica, que generalmente va dentro de una función `describe`

    ```
    it('cadena describiendo la prueba', callback);
    ```

    Se crea una prueba dentro de la función `it` al poner una aserción dentro de la función callback. La aserción se crea utilizando la función `expect`

Escribiendo pruebas con Jasmine cont...

- **expect**: Esta función es la que confirma que la prueba funciona. Estas líneas de código también se conocen como *aserción* dado que afirman que algo es verdadero. En Jasmine, la afirmación se divide en dos partes: en la función `expect` y en la función de comparación. La función `expect` es donde se pasa el valor obtenido; por ejemplo, un valor booleano. La función de comparación es donde se pasa el valor esperado. Algunas de las funciones de comparación que se incluyen son `toBe()`, `toContain()`, `toThrow()`, `toEqual()`, `toBeTruthy()` y `toBeNull()`. Tenga en cuenta que cuando escribe sus afirmaciones, debe tratar de tener una sola afirmación por prueba. Cuando hay múltiples aserciones, cada aserción debe ser verdadera para que la prueba pase
- `https://jasmine.github.io/tutorials/your_first_suite`

Escribiendo pruebas con Jasmine cont...

- Por ejemplo si necesitamos probar la siguiente funcion:

```
function helloWorld() {
  return 'Hello world!';
}
```

- Podemos escribir una prueba en Jasmin:

```
describe('Hello world', () => {
  it('says hello', () => {
    expect(helloWorld())
        .toEqual('Hello world!');
  });
});
```

Escribiendo pruebas con Jasmine cont...

- `expect(array).toContain(member);`
- `expect(fn).toThrow(string);`
- `expect(fn).toThrowError(string);`
- `expect(instance).toBe(instance);`
- `expect(mixed).toBeDefined();`
- `expect(mixed).toBeFalsy();`
- `expect(mixed).toBeNull();`
- `expect(mixed).toBeTruthy();`
- `expect(mixed).toBeUndefined();`
- `expect(mixed).toEqual(mixed);`
- `expect(mixed).toMatch(pattern);`
- `expect(number).toBeCloseTo(number, decimalPlaces);`
- `expect(number).toBeGreaterThan(number);`
- `expect(number).toBeLessThan(number);`
- `expect(number).toBeNaN();`
- `expect(spy).toHaveBeenCalled();`
- `expect(spy).toHaveBeenCalledTimes(number);`
- `expect(spy).toHaveBeenCalledWith(...arguments);`

Escribiendo pruebas con Jasmine cont...

- En general, se desea tener un archivo de prueba para todos y cada uno de los archivos de código (que no sean pruebas) de la aplicación
- Se debe adoptar un esquema de nomenclatura común para que las herramientas de compilación y los encargados de ejecutarlas puedan seleccionar entre archivos de prueba y archivos que no lo sean
- Uno de los esquemas de nomenclatura de archivos de prueba más comunes es: `<archivo>.spec.js`, donde, si tiene un archivo de código llamado `app.js`, el archivo que contiene todas las pruebas para `app.js` se llamaría `app.spec.js`
- Es muy posible que los archivos de prueba se encuentren en un directorio independiente del resto del código generalmente llamado 'test' (este es el caso de las pruebas E2E)

Configuración y limpieza

- A veces, para probar una característica necesitamos realizar alguna configuración, tal vez necesite crear algunos objetos de prueba
- También es posible que tengamos que realizar algunas actividades de limpieza una vez que hayamos finalizado las pruebas, como por ejemplo, eliminar algunos archivos del disco rigido
- Estas actividades se denominan configuración y limpieza y Jasmine tiene algunas funciones de ayuda:
  - **beforeAll**: Esta función se llama una vez, antes de que se ejecuten todas las especificaciones descritas en el conjunto de pruebas
  - **afterAll**: Esta función se llama una vez que todas las especificaciones de un conjunto de pruebas han finalizado
  - **beforeEach**: Esta función se llama antes de la ejecución de la especificación de la prueba
  - **afterEach**: Esta función se llama después de ejecutar cada especificación de prueba

Configuración y limpieza

- Podríamos usar estas funciones así:

```
describe('Hello world', () => {

  let expected = "";

  beforeEach(() => {
    expected = "Hello World";
  });

  afterEach(() => {
    expected = "";
  });

  it('says hello', () => {
    expect(helloWorld())
        .toEqual(expected);
  });
});
```

Configuracion y limpieza cont...

- Debe utilizar las funciones `before`, `beforeEach`, `after` y `afterEach` para configurar el contexto apropiado para las pruebas y eliminar el código redundante
- Idealmente, solo debe haber un par de líneas de código dentro de cada función `it()`
- Además debe tener en cuenta que el encargado de ejecutar las pruebas puede usar el primer parámetro de las funciones `describe()` y `it()` para informar lo que se está haciendo
- Por ejemplo, cuando se ejecuta esa especificación, algunos ejecutores de la prueba pueden generar:

```
Chromium 70.0.3538 (Ubuntu 0.0.0) ConfigService should be created FAILED
```

- Por lo tanto, hay que asegurarse de que los valores de cadena sean descriptivos

Karma

- La ejecución manual de las pruebas de Jasmine actualizando una pestaña del navegador repetidamente en diferentes navegadores cada vez que editamos el código puede resultar aburrido
- `Karma` es una herramienta que nos permite configurar disitnos navegadores y ejecutar las pruebas de Jasmine sobre dichos navegadores desde la línea de comandos
- Los resultados de las pruebas también se muestran en la línea de comando
- `Karma` también puede analizar los archivos de código en busca de cambios y volver a ejecutar las pruebas automáticamente
- No es necesario conocer los aspectos internos de cómo funciona `Karma`. Angular CLI, se encarga de la configuración de forma automática

Angular CLI

- Al crear proyectos Angular utilizando Angular CLI, se crean y ejecutan pruebas unitarias predeterminadas utilizando `Jasmine` y `Karma`
- Cada vez que generamos un artefacto nuevo utilizando Angular CLI, también creamos un archivo de especificaciones de Jasmine con el mismo nombre que el archivo de código principal, pero que termina en `.spec.ts`
- Para correr todas las pruebas de nuestra aplicación, simplemente ejecutamos `ng test` en la raíz de nuestro proyecto
- Las puerbas se ejecutaran con Jasmine a través de Karma
- Observa los cambios en nuestros archivos de desarrollo, agrupa todos los archivos del desarrollador y vuelve a ejecutar las pruebas automáticamente

Pruebas deshabilitadas o enfocadas

- Puede deshabilitar las pruebas sin necesidad de comentarlas con solo agregar la `x` delante de la función `describe`:

```
xdescribe('Hello world', () => {
  it('says hello', () => {
    expect(helloWorld())
        .toEqual('Hello world!');
  });
});
```

Pruebas deshabilitadas o enfocadas cont...

- A la inversa, también puede centrarse en pruebas específicas anteponiendo la letra `f` a la función `describe`:

```
fdescribe('Hello world', () => {
  it('says hello', () => {
    expect(helloWorld())
        .toEqual('Hello world!');
  });
});
```

Probando una clase

- Vamos a hacer las primeras pruebas sobre una clase, dado que en Angular todos los artefactos son clases en si
- Supongamos que tenemos una clase simple llamada `AuthService`, que es un servicio que luego vamos a inyectar utilizando DI de Angular

```
export class AuthService {
  isAuthenticated(): boolean {
    return !!localStorage.getItem('token');
  }
}
```

Probando una clase cont...

- Para probar esta clase, creamos un archivo de prueba llamado `auth.service.spec.ts` que se encuentra junto a nuestro archivo `auth.service.ts`:

```
import {AuthService} from './auth.service';

describe('Service: Auth', () => {

});
```

Probando una clase cont...

- Queremos ejecutar nuestras especificaciones de prueba contra instancias nuevas de `AuthService`, de modo que vamos a usar las funciones `beforeEach` y `afterEach` para configurar y limpiar las instancias:

```
describe('Service: Auth', () => {
  let service: AuthService;

  beforeEach(() => {
    service = new AuthService();
  });

  afterEach(() => {
    service = null;
    localStorage.removeItem('token');
  });
});
```

Probando una clase cont...

- Ahora creamos algunas especificaciones de prueba, en la primera vamos a comprobar si la función `isAuthenticated` devuelve `true` cuando hay un token

```
it('isAuthenticated debe retornar true cuando hay token', () => {
  localStorage.setItem('token', '1234');
  expect(service.isAuthenticated()).toBeTruthy();
});
```

- También queremos probar el caso inverso, cuando no hay token, la función debe devolver falso:

```
it('isAuthenticared debe retornar false cuando no hay token', () => {
  expect(service.isAuthenticated()).toBeFalsy();
});
```

Pruebas con Mocks y Spies

- Supongamos que tenemos un `LoginComponent` que funciona con el `AuthService` que probamos anteriormente, de esta manera:

```
import {Component} from '@angular/core';
import {AuthService} from "./auth.service";

@Component({
  selector: 'app-login',
  template: `<a [hidden]="noLogin()">Login</a>`
})
export class LoginComponent {

  constructor(private auth: AuthService) {
  }

  noLogin() {
    return this.auth.isAuthenticated();
  }
}
```

Pruebas con Mocks y Spies

- Inyectamos el `AuthService` en el `LoginComponent` y el componente muestra un botón Iniciar sesión si `AuthService` dice que el usuario no está autenticado.
- El `AuthService` es el mismo que el ejemplo anterior:

```
export class AuthService {
  isAuthenticated(): boolean {
    return !!localStorage.getItem('token');
  }
}
```

Mocking con clases falsas

- Podemos crear un `AuthService` falso llamado `MockedAuthService` que simplemente devuelve el valor que necesitemos para nuestra prueba
- Incluso podemos eliminar la importación de `AuthService` si queremos que realmente no haya dependencia
- Ahora, el `LoginComponent` se prueba de forma aislada:

```
import {LoginComponent} from './login.component';

class MockAuthService {
  authenticated = false;

  isAuthenticated() {
    return this.authenticated;
  }
}
```

Mocking con clases falsas cont...

```
describe('Component: Login', () => {

  let component: LoginComponent;
  let service: MockAuthService;

  beforeEach(() => {
    service = new MockAuthService();
    component = new LoginComponent(service);
  });

  afterEach(() => {
    service = null;
    component = null;
  });


  it('noLogin retorna true cuando el usuario esta autenticado', () => {
    service.authenticated = true;
    expect(component.noLogin()).toBeTruthy();
  });

  it('noLogin retorna false cuando el usuario no esta autenticado', () => {
    service.authenticated = false;
    expect(component.noLogin()).toBeFalsy();
  });
});
```

Mocking sustituyendo funciones

- A veces, crear una copia falsa completa de una clase real puede ser complicado, lento e innecesario.
- En su lugar, podemos simplemente extender la clase y anular una o más funciones específicas para que devuelvan las respuestas de prueba que necesitamos, de esta manera:

```
class MockAuthService extends AuthService {
  authenticated = false;

  isAuthenticated() {
    return this.authenticated;
  }
}
```

Mocking sustituyendo funciones cont...

- En la clase anterior, `MockAuthService` extiende `AuthService`. Tendrá acceso a todas las funciones y propiedades que existen en `AuthService`, pero solo anulará la función `isAuthenticated` para que podamos controlar fácilmente su comportamiento y aislar nuestra prueba de `LoginComponent`.
- El resto de la suite de prueba que utiliza el Mock a través de funciones sustituidas es igual a la versión de prueba que utilizamos antes con clases falsas

Simulacro usando una instancia real con Spy

- Un `spy` es una característica de Jasmine que permite tomar una clase, función u objeto y simularla de manera que pueda controlar lo que obtiene de las funciones
- Vamos a volver a escribir nuestra prueba para usar un `Spy` en una instancia real de `AuthService`, así:

```
import {LoginComponent} from './login.component';
import {AuthService} from "./auth.service";

describe('Component: Login', () => {

  let component: LoginComponent;
  let service: AuthService;
  let spy: any;

  beforeEach(() => {
    service = new AuthService();
    component = new LoginComponent(service);
  });
```

Simulacro usando una instancia real con Spy cont...

```
  afterEach(() => {
    service = null;
    component = null;
  });

  it('noLogin retorna true cuando el usuario esta autenticado', () => {
    spy = spyOn(service, 'isAuthenticated').and.returnValue(true);
    expect(component.noLogin()).toBeTruthy();
    expect(service.isAuthenticated).toHaveBeenCalled();

  });

  it('noLogin retorna false cuando el usuario no esta autenticado', () => {
    spy = spyOn(service, 'isAuthenticated').and.returnValue(false);
    expect(component.noLogin()).toBeFalsy();
    expect(service.isAuthenticated).toHaveBeenCalled();
  });
});
```

Simulacro usando una instancia real con Spy cont...

- Al usar la función `spy` de Jasmine, podemos hacer que una función devuelva lo que necesitemos para la prueba:

```
spyOn(service, 'isAuthenticated').and.returnValue(false);
```

- En nuestro ejemplo anterior, hacemos que la función `isAuthenticated` devuelva `false` o `true` en cada especificación de prueba de acuerdo con nuestras necesidades

Angular TestBed

- El banco de pruebas de Angular (ATB) es un framework de Angular de nivel superior que nos permite probar fácilmente los comportamientos que dependen del propio framework de Angular
- Con ATB podemos crear componentes, manejar la inyección, probar el comportamiento asíncrono e interactuar con nuestra aplicación de forma sencilla

Angular TestBed cont...

- Vamos a demostrar cómo usar el ATB con un ejemplo
- Convertiremos el componente que probamos con Jasmine a uno que usa el ATB

```
import {TestBed, ComponentFixture} from '@angular/core/testing';
import {LoginComponent} from './login.component';
import {AuthService} from "./auth.service";

describe('Component: Login', () => {

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [LoginComponent],
      providers: [AuthService]
    });
  });
});
```

Angular TestBed cont...

- En la función `beforeEach` para nuestro conjunto de pruebas, configuramos un módulo de prueba utilizando la clase `TestBed`
- Esto crea un módulo angular de prueba que podemos utilizar para crear instancias de componentes, realizar la inyección de dependencia, etc.
- Se configura exactamente de la misma manera que configuramos un `NgModule` normal. En este caso, pasamos el `LoginComponent` en las declaraciones y el `AuthService` en los proveedores

Angular TestBed cont...

- Una vez que el ATB está configurado, podemos usarlo para crear instancias de componentes y resolver dependencias

```
import {TestBed, ComponentFixture} from '@angular/core/testing';
import {LoginComponent} from './login.component';
import {AuthService} from "./auth.service";

describe('Component: Login', () => {

  let component: LoginComponent;
  let fixture: ComponentFixture<LoginComponent>;
  let authService: AuthService;
```

Angular TestBed cont...

```
  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [LoginComponent],
      providers: [AuthService]
    });

    // create component and test fixture
    fixture = TestBed.createComponent(LoginComponent);

    // get test component from the fixture
    component = fixture.componentInstance;

    // UserService provided to the TestBed
    authService = TestBed.get(AuthService);

  });
});
```

Angular TestBed cont...

- Ahora que configuramos el `TestBed` y obtuvimos el componente y el servicio, podemos ejecutar las mismas especificaciones de prueba que antes:

```
it('noLogin retorna true cuando el usuario esta autenticado', () => {
  spyOn(authService, 'isAuthenticated').and.returnValue(true);
  expect(component.onLogin()).toBeTruthy();
  expect(authService.isAuthenticated).toHaveBeenCalled();
});

it('noLogin retorna false cuando el usuario no esta autenticado', () => {
  spyOn(authService, 'isAuthenticated').and.returnValue(false);
  expect(component.onLogin()).toBeFalsy();
  expect(authService.isAuthenticated).toHaveBeenCalled();
});
```

Probando Formularios guiados por modelos

- Para probar los formularios, vamos a definir un `LoginComponent`

```
@Component({
  selector: 'app-login',
  template: `
<form (ngSubmit)="login()"
      [formGroup]="form">
  <label>Email</label>
  <input type="email"
         formControlName="email">
  <label>Password</label>
  <input type="password"
         formControlName="password">
  <button type="submit">Login</button>
</form>
`
})
export class LoginComponent {
  @Output() loggedIn = new EventEmitter<User>();
  form: FormGroup;

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

Probando Formularios guiados por modelos cont...

```
  ngOnInit() {
    this.form = this.fb.group({
      email: ['', [
        Validators.required,
        Validators.pattern("[^ @]*@[^ @]*")]],
      password: ['', [
        Validators.required,
        Validators.minLength(8)]],
    });
  }

  login() {
    console.log(`Login ${this.form.value}`);
    if (this.form.valid) {
      this.loggedIn.emit(
          new User(
              this.form.value.email,
              this.form.value.password
          )
      );
    }
  }
}
```

Probando Formularios guiados por modelos cont...

- Nuestra prueba quedaría de la siguiente forma:

```
describe('Component: Login', () => {

  let component: LoginComponent;
  let fixture: ComponentFixture<LoginComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [ReactiveFormsModule, FormsModule],
      declarations: [LoginComponent]
    });

   // create component and test fixture
   fixture = TestBed.createComponent(LoginComponent);

   // get test component from the
   fixture component = fixture.componentInstance;
   component.ngOnInit();
   });
});
```

Probando Formularios guiados por modelos cont...

- La primera especificación de prueba que podemos hacer es verificar que un formulario en blanco no es válido
- Dado que estamos usando formularios controlados por modelos, podemos verificar la propiedad `valid` en el modelo de formulario:

```
it('form invalid when empty', () => {
  expect(component.form.valid).toBeFalsy();
});
```

Probando Formularios guiados por modelos cont...

- También podemos verificar si los campos individuales son válidos
- Por ejemplo, el campo de correo electrónico inicialmente debería ser inválido

```
it('email field validity', () => {
  let email = component.form.controls['email'];
  expect(email.valid).toBeFalsy();
});
```

Probando Formularios guiados por modelos cont...

- Además de verificar si el campo es válido, también podemos ver qué validadores específicos fallan a través de la propiedad `email.errors`
- Dado que es obligatorio y el campo de correo electrónico no se ha establecido, es de esperar que el validador requerido falle

```
it('email field validity', () => {
   let errors = {};
   let email = component.form.controls['email'];
   errors = email.errors || {};
   expect(errors['required']).toBeTruthy();
});
```

- Debido a que los errores contienen una clave `required` y tiene un valor, esto significa que el validador requerido ha fallado

Probando Formularios guiados por modelos cont...

- Podemos establecer algunos datos en nuestra caja de texto llamando a `setValue(...)`:

```
email.setValue("test");
```

- Si configuramos el campo de correo electrónico en una prueba, debería fallar el validador de patrones, ya que se espera que el correo electrónico contenga una `@`
- Luego podemos verificar si el validador de patrones está fallando

```
email.setValue("test");
errors = email.errors || {};
expect(errors['pattern']).toBeTruthy();
```

Probando Formularios guiados por modelos cont...

- Podemos probar también el envío de formularios
- Dado que la directiva `ngSubmit` tiene su propio conjunto de pruebas, es seguro asumir que la expresión `(ngSubmit)=login()` funciona como se espera
- Entonces, para probar el envío de formularios con formularios controlados por modelos, podemos simplemente llamar a la función `login()` en nuestro controlador, de esta manera

```
component.login();
```

Probando Formularios guiados por modelos cont...

```
it('submitting a form emits a user', () => {
    expect(component.form.valid).toBeFalsy();
    component.form.controls['email'].setValue("test@test.com");
    component.form.controls['password'].setValue("123456789");
    expect(component.form.valid).toBeTruthy();

    let user: User;
    // Subscribe to the Observable and store
    // the user in a local variable.
    component.loggedIn.subscribe((value) =>
    user = value);

    // Trigger the login function
    component.login();

    // Now we can check to make sure
    // the emitted value is correct
    expect(user.email).toBe("test@test.com");
    expect(user.password).toBe("123456789");
});
```

Probando HttpClient

- Aquí hay un ejemplo de servicio angular que utiliza la clase `HttpClient`

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

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

Probando HttpClient cont...

- Si queremos probar dicha clase, debemos proveer al constructor de una instancia de `HttpClient`
- Angular se asegura de que solo tengamos una instancia de `HttpClient` en toda la aplicación
- Como resultado, no podemos simplemente crear una nueva instancia de `HttpClient` y pasarla a nuestro servicio:

```
//...
//We can not do that, there will be an error
beforeEach(() => {
   const httpClient = new HttpClient();
   const authService = new AuthService(httpClient);
});
//...
```

Probando HttpClient cont...

- Como solución lo vamos a inyectar como lo hace Angular

```
import { TestBed, inject } from '@angular/core/testing';
import { HttpClientTestingModule } from '@angular/common/http/testing';
import { AuthService } from './auth.service';

describe('AuthService', () => {
  beforeEach(() => {
     TestBed.configureTestingModule({
        imports: [HttpClientTestingModule],
        providers: [AuthService]
     });
  });

  it('should be initialized', inject([AuthService],
    (authService: AuthService) => {
       expect(authService).toBeTruthy();
  }));
});
```

Probando HttpClient cont...

- En este ejemplo estamos utilizando algunas utilidades nuevas:
  - **HttpClientTestingModule**: es análogo a `HttpClientModule`, pero con fines de prueba
  - **inject**: es una función de utilidad angular que inyecta servicios en la función de prueba. Requiere dos parámetros: un arreglo de servicios que queremos inyectar e instancias de esos servicios
  - **TestBed.configureTestingModule**: es identico a `@NgModule`, pero para la inicialización de la prueba

Probando HttpClient cont...

- Si ahora agregamos un metodo de login a nuestro servicio:

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { User } from '../models/user';
import { Observable } from 'rxjs/Observable';

@Injectable()
export class AuthService {
  private apiUrl = 'https://example.com/login';
  constructor(private http: HttpClient) {}

  public onLogin(user: User): Observable<Object> {
    return this.http.post(this.apiUrl, user);
  }
}
```

Probando HttpClient cont...

- `HttpTestingController` es un controlador para ser inyectado en las pruebas, que permite falsear y vaciar las solicitudes
- En si lo que hace es vigilar la URL que se llamará, la intercepta y devuelve una respuesta "falsa"
- Llamamos a la API donde le pasamos un objeto que contiene una URL que se interceptara y luego creamos una respuesta "falsa"
- Además de `expectOne`, existen otros métodos como, `expectNone` y `verify`

Probando HttpClient cont...

```
import { TestBed, inject } from '@angular/core/testing';
import {
  HttpClientTestingModule,
  HttpTestingController
} from '@angular/common/http/testing';
import { AuthService } from './auth.service';
import { User } from '../models/user';

describe('AuthService', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [AuthService]
    });
  });

  it(
    'should be initialized',
    inject([AuthService], (authService: AuthService) => {
      expect(authService).toBeTruthy();
    })
  );
```

Probando HttpClient cont...

```
it(
  'should perform login correctly',
  inject(
    [AuthService, HttpTestingController],
    (authService: AuthService, backend: HttpTestingController) => {
      const user = new User('test@example.com', 'testpassword');
      authService.onLogin(user).subscribe(
        (data: any) => {
          expect(data.success).toBe(true);
          expect(data.message).toBe('login was successful');
        },
        (error: any) => {}
      );

      backend
        .expectOne({
          url: 'https://example.com/login'
        })
        .flush({
          success: true,
          message: 'login was successful'
        });
    }
  )
);
```

Probando HttpClient cont...

```
  it(
    'should fail login correctly',
    inject(
      [AuthService, HttpTestingController],
      (authService: AuthService, backend: HttpTestingController) => {
        const user = new User('test@example.com', 'wrongPassword');
        authService.onLogin(user).subscribe(
          (data: any) => {
            expect(data.success).toBe(false);
            expect(data.message).toBe('email and password combination is wrong');
          },
          (error: any) => {}
        );

        backend
          .expectOne({
            url: 'https://example.com/login'
          })
          .flush({
            success: false,
            message: 'email and password combination is wrong'
          });
      }
    )
  );
});
```

Mejores prácticas

- Una lista de las mejores prácticas:
  - Utilice `beforeEach()` para configurar el contexto para sus pruebas
  - Asegúrate de que las descripciones de cadena que utiliza en `describe()` e `it()` tengan sentido como salida
  - Utilice `after()` y `afterEach()` para limpiar sus pruebas si hay algún estado inválido
  - Si alguna de las pruebas tiene más de 10 líneas de código, es posible que deba refactorizar la prueba
  - Si encuentra el mismo código para muchas pruebas, refactorice el código común en una función auxiliar

Laboratorio práctico - Pruebas de Unidad de su código

Es hora de verificar el codigo

- Escribe una prueba simple
- Prueba un servicio
- Usa un objeto espía de Jasmine
- Mock de una llamada http
- Use toHaveBeenCalledWith

Pipes

Luciano Diamand

Introducción a Pipes

Introducción a Pipes

- Cada aplicación comienza con lo que parece una tarea simple: obtener datos, transformarlos y mostrarlos a los usuarios
- Obtener datos puede ser tan simple como crear una variable local o tan complejo como transmitirlos a través de un WebSocket
- Una vez que llegan los datos, puede enviar sus valores de cadena en bruto directamente a la vista, pero eso rara vez hace que la experiencia del usuario sea buena
- Por ejemplo, en la mayoría de los casos de uso, los usuarios prefieren ver una fecha en un formato simple como *"el 15 de abril de 1988"* en lugar del formato de fecha (sin formato) *"el 15 de abril de 1988 a las 00:00:00 GMT-0700 (hora del Pacífico)"*.

Introducción a Pipes cont...

- Claramente, algunos valores generan una experiencia de usuario más satisfactoria al aplicarles formato
- Se puede notar que muchas de las mismas transformaciones se dan repetidamente, tanto dentro de la aplicación como a través de distintas aplicaciones
- Es posible pensar en dichas transformaciones si fueran estilos, de hecho, es posible aplicarlas en plantillas HTML mientras se definen los estilos
- Los **pipes** de Angular son una forma de escribir transformaciones de valores para la visualización de datos con formato que se puede declarar en las plantillas HTML

Usando Pipes

- Un pipe toma datos como entrada y los transforma en una salida deseada
- El siguiente ejemplo usa pipes para transformar la propiedad de cumpleaños de un componente en una fecha amigable

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

@Component({
  selector: 'app-hero-birthday',
  template: `<p>El cumpleaños del heroe es {{ birthday | date }}</p>`
})
export class HeroBirthdayComponent {
  birthday = new Date(1988, 3, 15); // April 15, 1988
}
```

- Dentro de la expresión de interpolación se pasa el valor de la propiedad `birthday` del componente a través del operador pipe `(|)` a la función `date` de la derecha. Todos los pipes funcionan de esta manera

Pipes incorporados

- Angular viene con algunos pipes pre-definidos:
  - CurrencyPipe
  - DatePipe
  - DecimalPipe
  - I18nPluralPipe
  - I18nSelectPipe
  - JsonPipe
  - KeyValuePipe
  - LowerCasePipe
  - PercentPipe
  - SlicePipe
  - TitleCasePipe
  - UpperCasePipe

Ejemplo de DatePipe

```
<div class="card">
   <div class="card-body">
      <h4 class="card-title">Date Pipe</h4>
      <p ngNonBindable>{{today | date}}</p>
      <p>{{today | date}}</p>
      <hr>
      <p ngNonBindable>{{today | date:'fullDate'}}</p>
      <p>{{today | date:'fullDate'}}</p>
      <hr>
      <p ngNonBindable>{{today | date:'shortTime'}}</p>
      <p>{{today | date:'shortTime'}}</p>
      <hr>
      <p ngNonBindable>{{today | date:'full'}}</p>
      <p>{{today | date:'full'}}</p>
      <hr>
      <p ngNonBindable>{{today | date:'yyyy-MM-dd HH:mm a z'}}</p>
      <p>{{today | date:'yyyy-MM-dd HH:mm a z'}}</p>
   </div>
</div>
```

Ejemplo de DecimalPipe

```
<div class="card">
   <div class="card-body">
      <h4 class="card-title">Decimal Pipe</h4>
      <p ngNonBindable>{{123456.14 | number}}</p>
      <p>{{123456.14 | number}}</p>
      <hr>
      <p ngNonBindable>{{1.3456 | number:'3.1-2'}}</p>
      <p>{{1.3456 | number:'3.1-2'}}</p>
      <hr>
      <p ngNonBindable>{{123456789 | number:'1.2-2'}}</p>
      <p>{{123456789 | number:'1.2-2'}}</p>
      <hr>
      <p ngNonBindable>{{1.2 | number:'4.5-5'}}</p>
      <p>{{1.2 | number:'4.5-5'}}</p>
   </div>
</div>
```

Ejemplo de CurrencyPipe

```
<div class="card">
   <div class="card-body">
      <h4 class="card-title">Currency Pipe</h4>
      <p ngNonBindable>{{1234567.5555 | currency}}</p>
      <p>{{1234567.555 | currency}}</p>
      <hr>

      <p ngNonBindable>{{1234567 | currency:'INR'}}</p>
      <p>{{1234567 | currency:'INR'}}</p>
      <hr>

      <p ngNonBindable>{{1234567 | currency:'CAD': 'symbol' :'1.2-5'}}</p>
      <p>{{1234567 | currency:'CAD':'symbol' :'1.2-5'}}</p>
      <hr>

      <p ngNonBindable>{{1234567 | currency:'CAD': 'symbol-narrow' :'1.2-5'}}</p>
      <p>{{1234567 | currency:'CAD':'symbol-narrow' :'1.2-5'}}</p>
      <hr>

      <p ngNonBindable>{{1234567.555 | currency:'INR':'symbol':'1.0-0'}}</p>
      <p>{{1234567.555 | currency:'INR':'symbol':'1.0-0'}}</p>
  </div>
</div>
```

Parametrizando Pipes

- Un pipe puede aceptar cualquier número de parámetros opcionales para ajustar su salida
- Para agregar parámetros a un pipe, se escribe el nombre del pipe seguido de dos puntos `(:)` y luego el valor del parámetro *(como la moneda: 'EUR')*.
- Si el pipe acepta varios parámetros, separe los valores con dos puntos (como por ejemplo, `slice: 1: 5`)
- Vamos a modificar la plantilla de cumpleaños para dar al pipe `date` un parámetro de formato
- Luego de formatearlo, el cumpleaños del héroe del 15 de abril, se presenta como `15/04/88`:

```
<p>El cumpleaños del heroe es el {{ birthday | date:"MM/dd/yy" }}</p>
```

Parametrizando Pipes cont...

- El valor del parámetro puede ser cualquier expresión de plantilla válida (consulte la sección *Expresiones de plantilla* de la página *Sintaxis de plantilla*), como un literal de cadena o una propiedad de componente
- En otras palabras, puede controlar el formato mediante un enlace a una propiedad de la misma manera que controla el valor de cumpleaños
- Vamos a modificar el componente agregando un parámetro que vincule el formato del pipe a la propiedad `format` del componente

```
template: `
  <p>El cumpleaños del heroe es {{ birthday | date:format }}</p>
  <button (click)="toggleFormat()">Cambiar Formato</button>
`
```

Parametrizando Pipes cont...

- También agregamos un botón a la plantilla y vinculamos su evento clic al método `toggleFormat()` del componente
- Ese método alterna la propiedad de formato del componente entre una forma corta `('shortDate')` y una forma más larga `('fullDate')`

```
export class HeroBirthday2Component {
  birthday = new Date(1988, 3, 15); // April 15, 1988
  toggle = true; // start with true == shortDate

  get format() { return this.toggle ? 'shortDate' : 'fullDate'; }
  toggleFormat() { this.toggle = !this.toggle; }
}
```

Encadenando Pipes

- Puede encadenar pipes en combinaciones potencialmente útiles
- En el siguiente ejemplo, para mostrar el cumpleaños en mayúsculas, `birthday` se encadena a `DatePipe` y a `UpperCasePipe`
- El cumpleaños se muestra como *15 de abril de 1988*

```
El cumpleaños del heroe es {{ birthday | date | uppercase }}
```

- El siguiente ejemplo, muestra *VIERNES, 15 DE ABRIL DE 1988*, encadena los mismos pipes que el anterior, pero también pasa un parámetro a la fecha

```
El cumpleaños del heroe es {{  birthday | date:'fullDate' | uppercase}}
```

Pipes Personalizados

- Es posible escribir pipes personalizados
- El siguiente ejemplo es de pipe personalizado llamado `ExponentialStrengthPipe` que puede aumentar los poderes de un héroe:

```
import { Pipe, PipeTransform } from '@angular/core';
/*
 * Raise the value exponentially
 * Takes an exponent argument that defaults to 1.
 * Usage:
 *   value | exponentialStrength:exponent
 * Example:
 *   {{ 2 | exponentialStrength:10 }}
 *   formats to: 1024
*/
@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements PipeTransform {
  transform(value: number, exponent: string): number {
    let exp = parseFloat(exponent);
    return Math.pow(value, isNaN(exp) ? 1 : exp);
  }
}
```

Pipes Personalizados cont...

- Esta definición de pipe revela los siguientes puntos clave:
  - Un **pipe** es una clase decorada con metadatos de `pipe`
  - La clase de pipe implementa el método de transformación de la interfaz `PipeTransform` que acepta un valor de entrada seguido de parámetros opcionales y devuelve el valor transformado
  - Habrá un argumento adicional al método de transformación para cada parámetro pasado al pipe
  - En el ejemplo anteriro, el pipe tiene un parámetro: el exponente
  - Para decirle a Angular que se trata de un pipe, aplique el decorador `@Pipe`, que se importa desde la biblioteca `@angular/core`
  - El decorador `@Pipe` le permite definir el nombre del pipe que usará dentro de las expresiones de la plantilla. Debe ser un identificador de JavaScript válido. En este caso el nombre del pipe es `exponentialStrength`

Pipes Personalizados cont...

- Se utilizan de la misma forma que las tuberías integradas
- Se deben registrar los pipes personalizados, de no hacerlo, Angular informará con un error
- Se debe incluir en las declaraciones del arreglon de `AppModule`
- El generador de **Angular CLI** registra el pipe automáticamente

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

@Component({
  selector: 'app-power-booster',
  template: `
    <h2>Power Booster</h2>
    <p>Super power boost: {{ 2 | exponentialStrength: 10 }}</p>
  `
})
export class PowerBoosterComponent { }
```

Pipes y detección de cambios

- Angular busca cambios en los valores vinculados a los datos a través de un proceso de detección de cambios que se ejecuta después de cada evento DOM: cada pulsación de tecla, movimiento del mouse, marca de tiempo y respuesta del servidor
- Esto podría ser costoso. Angular se esfuerza por reducir el costo siempre que sea posible y apropiado
- Angular elige un algoritmo de detección de cambios más simple y rápido cuando utiliza un pipe

Pipes y detección de cambios cont...

- Como ejemplos, agregamos un `FlyingHeroesPipe` a `*ngFor` que filtra la lista de héroes solo a aquellos héroes que pueden volar

```
<div *ngFor="let hero of (heroes | flyingHeroes)">
  {{ hero.name }}
</div>
```

- Aquí está la implementación de `FlyingHeroesPipe`, que sigue el patrón de pipes personalizadas descritas anteriormente

```
import { Pipe, PipeTransform } from '@angular/core';

import { Flyer } from './heroes';

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

Pipes y detección de cambios cont...

```
New hero:
  <input type="text" #box
    (keyup.enter)="addHero(box.value); box.value=''"
    placeholder="hero name">
  <button (click)="reset()">Reset</button>
  <div *ngFor="let hero of heroes">
    {{ hero.name }}
  </div>

...

export class FlyingHeroesComponent {
  heroes: any[] = [];
  canFly = true;
  constructor() { this.reset(); }

  addHero(name: string) {
    name = name.trim();
    if (!name) { return; }
    let hero = {name, canFly: this.canFly};
    this.heroes.push(hero);
  }

  reset() { this.heroes = HEROES.slice(); }
}
```

Pipes y detección de cambios cont...

- Aunque no está obteniendo el comportamiento que desea, no es un bug de Angular
- Angular está utilizando un algoritmo diferente de detección de cambios que ignora los cambios en la lista o en cualquiera de sus elementos
- Un héroe se agrega de la siguiente manera:

```
this.heroes.push(hero);
```

- El héroe se agrega a un arreglo de héroes. La referencia al arreglo no cambió. Eso es todo lo que Angular verifica, por lo tanto al ser la misma matriz y no haber ningún cambio, no se actualiza la pantalla

Pipes y detección de cambios cont...

- Para arreglar esto, hay que crear un arreglo con el nuevo héroe agregado y asignar eso a los héroes. Esta vez Angular detecta que la referencia del arreglo ha cambiado y ejecuta el pipe, actualizando la la pantalla con el nuevo arreglo, que incluye al nuevo héroe
- Si muta la matriz, no se invoca ningún pipe y la pantalla no se actualiza
- Si reemplaza la matriz, el pipe se ejecuta y la pantalla se actualiza

Pipes y detección de cambios cont...

- Angular proporciona enlaces al ciclo de vida del componente para la detección de cambios
- `OnChanges` es una interfaz y tiene una declaración del método `ngOnChanges()`
- En un componente primario-secundario, el componente secundario declara la propiedad `@Input()` para obtener valores del componente principal
- Cada vez que el componente principal cambia el valor de las propiedades utilizadas en el componente secundario decorado con `@Input()`, el método `ngOnChanges()` creado en el componente secundario se ejecuta automáticamente
- El método `ngOnChanges()` utiliza `SimpleChanges` como argumento que proporciona valores nuevos y anteriores de los valores de entrada después de los cambios
- Si cambiamos solo los valores de las propiedades de un objeto de de entrada, el método `ngOnChanges()` no se ejecutará

Laboratorio práctico - Manipulación de datos con pipes

- Usando el pipe `lowercase`
- Usando el pipe de fecha con parámetros
- Creando un pipe personalizado
- Usando la función del ciclo de vida `ngOnChanges`
- Filtrando datos
- Ordenando datos

Directivas

Luciano Diamand

Visión general

Visión general

- Vamos a ver como crear Directivas personalizadas
- En realidad ya estuvimos creando directivas dado que los Componentes son en si Directivas
- Los componentes tienen todas las características de las Directivas, pero también tienen una vista, es decir, tienen una plantilla y algo de HTML que se inyecta en el DOM cuando lo usamos
- Otra diferencia entre componentes y directivas es que un solo elemento HTML solo puede tener un único componente asociado, sin embargo, un solo elemento puede tener varias directivas asociadas

Decorador de directiva

- Creamos directivas anotando una clase con el decorador `@Directive`
- En el ejemplo, vamos a crear una directiva `CardHoverDirective` decorada con `@Directive`
- Además la vamos a asociar al nombre `ccCardHover`

```
<div class="card card-block" ccCardHover>...</div>
```

- En la clase:

```
import { Directive } from '@angular/core';
.
.
.
@Directive({
  selector:"[ccCardHover]"
})
class CardHoverDirective { }
```

Selector de atributos

- El código anterior es muy similar a cuando escribiamos un componente, la primera diferencia notable es que el selector está envuelto con `[]`
- Para comprender por qué hacemos esto, primero debemos entender que el atributo selector utiliza reglas de coincidencia de CSS para hacer coincidir un componente/directiva con un elemento HTML
- En CSS para que coincida con un elemento específico, simplemente escribimos el nombre del elemento `p {...}`
- Esta es la razón por la que anteriormente, cuando definimos el selector en la directiva `@Component`, solo escribimos el nombre del elemento, que coincide con un elemento del mismo nombre

Selector de atributos cont...

- Si escribimos el selector como `.ccCardHover`

```
import { Directive } from '@angular/core';
.
.
.
@Directive({
  selector:".ccCardHover"
})
class CardHoverDirective { }
```

Selector de atributos

- Entonces esto asociaría la directiva con cualquier elemento que tenga una clase `ccCardHover`:

```
<div class="card card-block ccCardHover">...</div>
```

- Si queremos asociar la directiva a un elemento que tiene un determinado atributo, en CSS envolvemos el nombre del atributo con `[]`, y es por eso que el selector se llama `[ccCardHover]`

Constructor de directiva

- Lo siguiente que haremos es agregar un constructor a nuestra directiva:

```
import { ElementRef } from '@angular/core';
.
.
.
class CardHoverDirective {
  constructor(private el: ElementRef) {
  }
}
```

- Cuando se crea la directiva, Angular puede inyectar una instancia de un objeto llamado `ElementRef` al constructor

Constructor de directiva cont...

- El `ElementRef` le da a la directiva acceso directo al elemento DOM sobre el cual está adjunto
- Podemos usarlo, por ejemplo, para cambiar el color de fondo del elemento
- `ElementRef` en sí mismo es un contenedor para el elemento DOM real al que podemos acceder a través de la propiedad `nativeElement`

```
el.nativeElement.style.backgroundColor = "gray";
```

Constructor de directiva cont...

- Sin embargo, esto supone que nuestra aplicación siempre se ejecutará en el entorno de un navegador
- Angular se ha creado desde cero para trabajar en diferentes entornos, incluido el servidor a través de `node` y en un dispositivo móvil nativo
- Por lo tanto, el equipo de Angular ha proporcionado una forma independiente de la plataforma para establecer propiedades en nuestros elementos a través del `Renderer`

Constructor de directiva cont...

```
import { Renderer2 } from '@angular/core';
.
.
.
class CardHoverDirective {
  constructor(private el: ElementRef,
              private renderer: Renderer2) {
   renderer.setElementStyle(el.nativeElement,
      'backgroundColor', 'gray');
  }
}
```

Constructor de directiva cont...

- Utilizamos la inyección de dependencia (DI) para inyectar el `Renderer2` dentro del constructor de la directiva
- En lugar de configurar el color de fondo directamente a través del elemento DOM, lo hacemos a través del `Renderer2`

Laboratorio práctico - Creación de directivas

- Crear una directiva
- Escuchar cambios de parámetros de ruta de una página parametrizada

HttpClient

Luciano Diamand

Introducción a HttpClient

ontinuationsoap

- La mayoría de las aplicaciones de *front-end* se comunican con servicios de *back-end* a través del protocolo HTTP
- Los navegadores modernos admiten dos API diferentes para realizar solicitudes HTTP
  - la interfaz `XMLHttpRequest`
  - la API `fetch()`
- El `HttpClient` de `@angular/common/http` ofrece una API HTTP cliente simplificada para aplicaciones Angular que se basa en la interfaz `XMLHttpRequest` expuesta por los navegadores
- Los beneficios adicionales de `HttpClient` incluyen características de capacidad de prueba, objetos de solicitud y respuesta tipificados, intercepción de solicitud y respuesta, Apis observables y manejo simplificado de errores