San Cristobal Seguros

Luciano Diamand

Ultima actualización: October 12, 2023.
Correcciones, sugerenias, contribuciones y traducciones son bienvenidas!

Javascript

Luciano Diamand

Historia

Historia

- Lenguaje de programación desarrollado por Brendan Eich
- Se creó en Netscape como una herramienta de secuencias de comandos para manipular páginas Web en el navegador Netscape Navigator

Historia cont.

- Desde su lanzamiento en 1995, `JavaScript` ha pasado por muchos cambios
- Al comienzo sólo se utilizaba para agregar elementos interactivos o "dar vida" a las páginas web: clics en botones, eventos sobre enlaces, validación de formularios, etc.
- Más tarde, `JavaScript` se hizo más robusto con `DHTML` y `AJAX`
- Hoy en día, con la aparición de `Node.js`, `JavaScript` se ha convertido en un lenguaje que trascendió las fronteras del navegador Web para permitir crear aplicaciones del lado del servidor o de escritorio
- Podemos usar `JavaScript` para programar robots (`http://johnny-five.io/`) o crear herramientas de línea de comandos (`https://www.npmjs.com/package/commander`)

Historia cont.

- La evolución ha sido guiada por un grupo de personas y de empresas que usan `JavaScript`, proveedores de navegadores
- El comité a cargo de guiar los cambios en `JavaScript` a lo largo de los años es la Asociación Europea de Fabricantes de Computadoras (ECMA)
- Los cambios en el lenguaje son impulsados por la comunidad
- Cualquiera puede presentar una propuesta al comité ECMA
- La responsabilidad del comité ECMA es administrar y priorizar estas propuestas para decidir qué se incluye en cada especificación

Historia cont.

- El primer lanzamiento de `ECMAScript` fue en 1997, `ECMAScript1`
- Esto fue seguido en 1998 por `ECMAScript2`
- `ECMAScript3` apareció en 1999, agregando expresiones regulares, manejo de cadenas y más
- La versión `ECMAScript4` nunca fue lanzada
- En 2009, se lanzó `ECMAScript5` (ES5), que incorpora características nuevas como: métodos de arreglos, propiedades de objeto y soporte de biblioteca para JSON

Historia cont.

- Después del lanzamiento de ES6 o ES2015 en 2015, la especificación ECMAScript se ha movido a una cadencia de lanzamiento anual, y las versiones del lenguaje ES2016, ES2017, ES2018, ES2019 y ES2020 ahora se identifican por año de lanzamiento
- Todo lo que se proponga agregar y que formará parte de la especificación de `JavaScript` se lo denomina `ESNext`
- En ES5 y versiones posteriores, los programas pueden optar por el *modo estricto* de JavaScript en el que se han corregido una serie de errores de lenguaje iniciales
- ES6 se lanzó en 2015 y agregó nuevas características importantes, incluida la sintaxis de clases y módulos

Más información

- Historia (`https://en.wikipedia.org/wiki/JavaScript`)
- Modo estricto (`https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Strict_mode`)
- Transición a modo estricto (`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode/Transitioning_to_strict_mode`)
- La guerra de los navegadores (`https://www.emezeta.com/articulos/browser-wars-la-historia-de-la-guerra-de-navegadores`)
- Estado de JS 2021 (`https://2021.stateofjs.com/`)
- El estado de Octoverse 2021 (`https://octoverse.github.com/`)

Caracteristicas

Caracteristicas

- JavaScript es un lenguaje:
  - **Interpretado**: Es un lenguaje de programación para el que la mayoría de sus implementaciones ejecuta las instrucciones directamente, sin una previa compilación del programa a instrucciones en lenguaje máquina
  - **Hilo único**: Los lenguajes de programacion pueden ser de un único hilo o multi hilo. JavaScript posee un único hilo de ejecución
  - **Tipado dinámico**: Un lenguaje de programación tiene un sistema de tipos dinámico cuando el tipo de dato de una variable puede cambiar en tiempo de ejecución. La mayoría de los lenguajes de tipado dinámico son lenguajes interpretados
  - **Inferencia de tipos**: Al decir `a = 5` o `a = "Hola mundo"`, `JavaScript` es capaz de inferir el tipo de dato de una variable a partir del valor que se le está asignando

Introducción cont.

- **Tipado débil**: Un lenguaje se considera de tipado fuerte cuando ante una operación entre dos tipos de datos incompatibles, arroja un error (durante la compilación o la ejecución, dependiendo de si se trata de un lenguaje compilado o interpretado), en lugar de convertir implícitamente alguno de los dos tipos
- **Multiparadigma**: La Programación Multiparadigma es una práctica que emerge como resultado de la co-existencia de los paradigmas Orientado a Objetos, Procedural, Declarativo y Funcional buscando mejorar la producción en el desarrollo de proyectos. Según lo describe Bjarne Stroustrup, permiten crear "programas usando más de un estilo de programación"

Más información

- Tipos en JavaScript (`https://developer.mozilla.org/es/docs/Web/JavaScript/Data_structures`)
- Paradigmas en JavaScript (`https://dev.to/alamin__yusuf/paradigms-in-javascript-1m31`)
- Event loop en JavaScript (`https://www.becomebetterprogrammer.com/is-node-js-single-threaded-or-multi-threaded-and-why/`)

Node.js

Introducción

- `Node.js` es un entorno de tiempo de ejecución de `JavaScript` de código abierto y multiplataforma
- `Node.js` ejecuta el motor V8 de JavaScript (que es el núcleo de Google Chrome) fuera del navegador, esto permite que `Node.js` sea muy eficaz
- Una aplicación en `Node.js` se ejecuta en un solo proceso, sin crear un nuevo hilo para cada solicitud
- `Node.js` proporciona un conjunto de primitivas de E/S asincrónicas en su biblioteca estándar que evita que el código JavaScript se bloquee
- En general, las bibliotecas en `Node.js` se escriben utilizando paradigmas que no bloquean, lo que hace que el comportamiento de bloqueo sea la excepción en lugar de la norma
- Se liberan dos versiones al año (Abril y Octubre)
- Las versiones pares se promueven a `LTS` luego de 6 meses (`https://github.com/nodejs/release`)

Introducción cont.

- Cuando `Node.js` realiza una operación de E/S, como leer de la red, acceder a una base de datos o al sistema de archivos, en lugar de bloquear el subproceso y desperdiciar ciclos de CPU en espera, `Node.js` reanudará las operaciones cuando regrese la respuesta
- Esto permite que `Node.js` maneje miles de conexiones simultáneas con un solo servidor sin presentar la carga de administrar la concurrencia de subprocesos, lo que podría ser una fuente importante de errores

Npm

- `node.js` cuenta con un gestor de paquetes llamado `npm`
- `npm` realiza dos tareas:
  - Es un repositorio en línea para la publicación de proyectos Node.js de código abierto
  - Es una utilidad de línea de comandos para interactuar con dicho repositorio que ayuda en la instalación de paquetes, la gestión de versiones y la gestión de dependencias
- `npm` con su estructura simple ayudó a que proliferar el ecosistema de `Node.js`
- El registro de `npm` alberga más de 1000000 de paquetes de código abierto que se pueden usar libremente

Instalar Node.js

- `Node.js` se puede instalar de diferentes maneras
- Los paquetes oficiales para todas las plataformas principales están disponibles en `https://nodejs.dev/en/download/`
- Una forma muy conveniente de instalar `Node.js` es a través de un administrador de paquetes. En este caso, cada sistema operativo tiene el propio
- Otros administradores de paquetes para MacOS, Linux y Windows se enumeran en `https://nodejs.dev/en/download/`

Node version manager (nvm)

- `nvm` es una forma popular de ejecutar `Node.js`
- `nvm` es un administrador de versiones para `Node.js`, diseñado para ser instalado por el usuario e invocado a través del `shell`
- `nvm` funciona en cualquier `shell` compatible con `POSIX` (`sh`, `dash`, `ksh`, `zsh`, `bash`), en particular en estas plataformas: `unix`, `macOS` y Windows Subsystem for Linux `WSL`
- Permite cambiar fácilmente la versión de `Node.js` e instalar nuevas versiones para intentar revertir fácilmente cuando algo falla
- `nvm` provee un script que automatiza la descarga en instalación de `nvm`

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.4/install.sh | bash
```

- o

```
wget -qO- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.4/install.sh | bash
```

- Las version de `Node.js` quedaran instaladas en el directorio `~/.nvm/versions/node/`

Node version manager (nvm) cont.

- Para instalar una versión particular

```
nvm install 18
```

- Para usar una versión particular

```
nvm use 18
```

- Para instalar la última versión LTS

```
nvm install --lts
```

- Para desinstalar una versión particular

```
nvm uninstall
```

- Para cambiar la versión por defecto al abrir la terminal

```
nvm alias default <version>
```

Usando Node.js

- Una vez instalado tendrá acceso al programa ejecutable de `Node.js` en la línea de comandos
- Si no está seguro de si `Node.js` está instalado en su máquina, puede abrir una ventana de Terminal o Símbolo del sistema y escribir:
  ```
  node -v
  ```
- La documentación de la API esta disponible en `https://nodejs.org/api/`
- `Node.js` define y utiliza los siguientes índices de estabilidad
  - **0** - **Discontinuado**: el uso de esta API esta desaconsejado
  - **1** - **Experimental**: no se considera estable y puede estar sujeta a cambios no compatibles hacia atrás
  - **2** - **Estable**: esta API es estable y asegura compatibilidad

Diferencias entre Node.js y el Navegador

- Tanto el navegador como `Node.js` utilizan JavaScript como lenguaje de programación
- La creación de aplicaciones que se ejecutan en el navegador es algo completamente diferente a la creación de una aplicación `Node.js`
- A pesar de que en ambos casos se utiliza JavaScript, existen algunas diferencias clave que hacen que la experiencia sea radicalmente diferente
- Las aplicaciones en `Node.js` traen consigo una gran ventaja: la comodidad de programar todo, el frontend y el backend, en un solo lenguaje tanto en el cliente como en el servidor

Diferencias entre Node.js y el Navegador cont.

- En el navegador, la mayoría de las veces se interactua con el DOM u otras API de la plataforma web como las cookies. Esto no existe en Node.js
- En Nodejs no existen el `document`, `window` y todos los demás objetos que proporciona el navegador
- En el navegador, no tenemos todas las buenas API que proporciona Node.js a través de sus módulos, como la funcionalidad de acceso al sistema de archivos
- Otra gran diferencia en Node.js es el control sobre el entorno
- En comparación con el entorno del navegador, donde no puede darse el lujo de elegir qué navegador usarán sus visitantes, esto es muy conveniente
- Esto significa que puede escribir todo el JavaScript ES6-7-8-9 moderno que admite su versión de Node.js

Diferencias entre Node.js y el Navegador cont.

- Dado que JavaScript se mueve tan rápido, pero los navegadores pueden ser un poco lentos para actualizar, a veces en la web se queda atascado con el uso de versiones anteriores de JavaScript/ECMAScript
- Puede usar Babel para transformar su código para que sea compatible con ES5 antes de enviarlo al navegador, pero en Node.js, no necesitará eso
- Otra diferencia es que Node.js es compatible con los sistemas de módulos CommonJS y ES (desde Node.js v12), mientras que en el navegador comenzamos a ver que se implementa el estándar de módulos ES.
- En la práctica, esto significa que puede usar tanto `require()` como `import` en Node.js, mientras que en el Navegador está limitado a `import`
- Referencia: `https://nodejs.dev/learn`

Recorrido por JavaScript

- Podemos utilizar la herramineta **Read Eval Print Loop** (`REPL` de `Node.js` para realizar pruebas rápidas. En la terminal ejecutamos`node`:

```
Welcome to Node.js v18.17.0.
Type ".help" for more information.
> .help
.break    Sometimes you get stuck, this gets you out
.clear    Alias for .break
.editor   Enter editor mode
.exit     Exit the REPL
.help     Print this help message
.load     Load JS from a file into the REPL session
.save     Save all evaluated commands in this REPL session to a file

Press Ctrl+C to abort current expression, Ctrl+D to exit the REPL
```

Hola Mundo

- Para probar bloques de código pequeños puede utilzar tanto el entorno interactivo de `Node.js` o la consola de Navegador
- Para códigos más complejos puede utilzar directamente un editor de texto
- Desde allí, puede copiar y pegar en la consola de JavaScript o en una sesión de `Node.js`, o puede guardar su código en un archivo (la extensión de nombre de archivo tradicional para código JavaScript es `.js`) y luego ejecutar ese archivo de código JavaScript con Node:

  ```
  node snippet.js
  ```

- Si usa Node de manera no interactiva, no imprimirá automáticamente el valor del código que ejecute

Más información

- Sitio Web de Node.js (`https://nodejs.dev/`)
- Npm (`https://nodejs.dev/en/learn/an-introduction-to-the-npm-package-manager/`)
- Nvm (`https://github.com/nvm-sh/nvm`)

Herramientas de desarrollo

Configuración mínima

- La forma más simple de probar algunas líneas de JavaScript es a través del Navegador
- Para ello podemos utilizar las **DevTools** que vienen incorporadas con la mayoría de los Navegadores
- Tambien podemos usar herramientas en linea como ser `https://jsbin.com`
- Otra opcion para probar el código en JavaScript es utilizar un servidor Web
- Una opción de servidor Web escrito JavaScript es `http-server`. El siguiente comando permite instalar `http-server` en nuestra maquina
  `npm install http-server -g`
- Luego, para iniciar el servidor, ejecutamos el comando `http-server`

DevTools

Tenemos distintas formas de acceder al *DevTools*:

- Vía menú: **Más herramientas** > **Herramientas para desarrolladores**
- Vía menú contextual: **Click derecho en cualquier lugar de la página** > **Inspeccionar**
- Vía atajos de teclado:
  - MacOS: **Cmd** + **Alt** + **i**
  - Windows o Linux: **Ctrl** + **Shift** + **i** o **F12**

DevTools cont.

Dentro de las herramientas de desarrollo, encontraremos las siguientes secciones:

- **Device**: Permite previsualizar el comportamiento de una Web en distintas resoluciones de pantalla. Muy útil para emular el comportamiento en móviles
- **Elements**: Permite acceder a los elementos que conforman a una página Web. Es posible manipular los argumentos y de esa forma modificar contenido y estilos
- **Console**: Registra el contenido de logs de la Web y permite interactuar con ella y con el código JS que contiene
- **Sources**: Permite depurar el código y conectarnos a nuestro sistema de archivos para usar el depurador como un editor en tiempo real
- **Network**: Registra el detalle de cada petición y respuesta de un sitio Web. Además, permite emular conexiones de red

DevTools cont.

- **Performance**: Permite grabar todos los eventos que ocurren durante el ciclo de vida de un sitio Web
- **Memory**: Nos permite identificar memory leaks en nuestras aplicaciones. Nos permite visualizar en tiempo real el consumo de Heap de una página
- **Application**: Aquí podemos consultar todos los recursos del navegador que utiliza un sitio Web: `cookies`, `IndexedDB`, `Service Workers`, `storages`, etc.
- **Security**: Sirve para consultar el nivel de seguridad de un sitio web: `HTTPS`, `certificados`, etc.
- **Lighthouse**: Nos permite realizar auditorías sobre un sitio Web: Performance, SEO, Accesibilidad, PWA y mejores prácticas

La consola

- La consola es, quizás, la herramienta más sobre-utilizada para depurar código y entender qué está pasando en él
- Hay que tener en cuenta que en entornos productivos hay que limitar al máximo posible el uso de logs

```
// Comprueba que si aserción es verdadera
console.assert('Padre' > 'Hijo');

// Limpia la consola
console.clear();

// Lista de forma interactiva las propiedades de un objeto
console.dir(window);

// Muestra una representación HTML del objeto
console.dirxml(document.body);
```

La consola cont.

- Nos permite agrupar elementos

```
const hobbits = ['Frodo Bolsón', 'Bilbo Bolsón', 'Samsagaz Gamyi', 'Peregrin Tuk'];
// Permite agrupar diferentes valores
console.group('The Hobbits');
hobbits.forEach(hobbit => console.log(hobbit));
// Indica el fin del grupo
console.groupEnd();
```

- También podemos medir el timpo que toma ejecutar un conjunto de sentencias

```
// Establece un contador y le da un identificador
// Muy útil para obtener métricas de rendimiento
console.time('Mide el tiempo de este bloque');
// código a medir el rendimiento
console.timeEnd('Mide el tiempo de este bloque');
```

La consola cont.

- Podemos crear tablas con información

```
const heroes = [
  { nombre: 'Bruce', apellidos: 'Wayne', identidad: 'Batman'},
  { nombre: 'Diana', apellidos: 'prince', identidad: 'WonderWoman' },
  { nombre: 'Clark', apellidos: 'Kent', identidad: 'Superman' }
];

// Nos permite ver de forma muy visual el contenido de un array de objetos
console.table(heroes);

// Además permite filtrar que columnas visualizar
console.table(heroes, ['identidad']);
```

La consola cont.

- La consola está orientada a funcionar por niveles
- Existen cuatro categorías de salida que se pueden generar, utilizando los métodos:
  - `console.debug`: Texto en color negro sin icono
  - `console.info`: Texto de color azul con icono
  - `console.warn`: Texto de color amarillo con icono
  - `console.error`: Texto de color rojo con icono
- Cada uno de ellos muestran resultados que lucen diferentes en el registro
- Se pueden utilizar los controles de filtro del navegador para ver únicamente los tipos de salida de interés

La consola cont.

- Esto, bien usado, es muy útil cuando queremos monitorear el estado de nuestro código

```
// Muestra el mensaje sólo cuando lo forzamos
console.debug('Soy tímido')

// Saca un mensaje por consola
console.log('We love you console.log!');

// Muestra un mensaje informativo
console.info('A ti también console.info...');

// Muestra una advertencia por consola
console.warn('Hay que lavarse las manos y estornudar en el codo');

// Muestra un error por consola
console.error('Batman vs Superman');
```

La consola cont.

- No hay que olvidar que la consola, además de util, puede ser tan bonita como queramos
- Además, algo que la diferencia de otras funciones de JavaScript, es que admite "*placeholding*" de variables de forma nativa

%s	Cadena de texto
%d / %i	Entero
%f	Decimal
%o	DOM
%O	Objeto JS
%c	CSS

```
const parrafos = document.getElementsByTagName('p');
console.log('DOM: %o', parrafos);

const estilos = 'background: #00b5ac; color: #fff; font-size: 18px;';
const egocentrico = { 'nombre': 'Yo', 'Apellido': 'Mismo' };

console.log('%c Objeto: %O', estilos, egocentrico);
```

La consola cont.

- También podemos usar `console.log('num: ', num)` en lugar de `console.log('num: ' + num)`
- Mientras que con el uso del `+` va a concatenar el resultado en una sola cadena, el uso de la `,` nos permite agregar una descripción y también visualizar el contenido de la variable en caso de que sea un objeto

Variables

Declaracion de variables

- Antes de ES2015, la única forma de declarar variables era utilizando la palabra clave `var`
- Ahora tenemos distintas opciones que mejoran el control sobre las variables
  - La palabra clave `const`
  - La palabra clave `let`
- Son case sensitive, es decir, las mayúsculas son importantes

Reglas sintácticas

```
var variable1;
let variable2;
const variable3 = 2;

// puede contener letras, números, _ o $
var $jquery, _private;

// No puede empezar con números o contener espacios
var 50cent = 1, esto no vale = 1;

// Son case sensitive
var variable = 2, VARIABLE = 2;
// Mayoritariamente se usa camelCase, pero también se usa snake_case
var camelCase = 2, snake_case = 2;
// Se recomienda declarar una variable por línea
var camelCase = 2;
var snake_case = 3;
```

La palabra clave const

- Una constante es una variable que no puede ser reemplazada. Una vez declarada, no es posible cambiar su valor
- JavaScript introdujo constantes en ES6
- Antes de las constantes, disponiamos solo de variables, las cuales pueden cambiar su contenido:

```
var pizza = true;
pizza = false;
console.log(pizza); // false
```

- Si intentamos cambiar el valor de una constante va a generar un error por consola:

```
const pizza = true;
pizza = false; // Uncaught TypeError: Assignment to constant variable
```

La palabra clave let

- JavaScript posee ahora variables de alcance lexico
- En JavaScript podemos crear bloques de código utilizando llaves
- En funciones, dichas llaves limitan el alcance de cualquier variable declarada con var
- Por otro lado, consideremos las sentencias `if/else`
- Si tenemos en mente otros lenguajes, podemos asumir de que estos bloques también van a bloquear el alcance de la variable
- Este no es el caso, hasta que llego `let`

La palabra clave let cont.

- Si una variable se define dentro de un bloque `if/else`, dicha variable no queda limitada solamente a dicho bloque:

```
var topic = 'JavaScript';
if (topic) {
  var topic = 'React';
  console.log('block', topic); // block React
}
console.log('global', topic); // global React
```

- La variable `topic` dentro del bloque, limpia el valor de `topic` fuera del bloque

La palabra clave let cont.

- Con la palabra clave `let`, podemos delimitar el alcance de una variable al bloque de código donde fue declarada
- Usando `let` protejemos el valor de la variable global

```
var topic = 'JavaScript';
if (topic) {
  let topic = 'React';
  console.log('block', topic); // block React
}
console.log('global', topic); // global JavaScript
```

- El valor de `topic` fuera del bloque no se modifica

Alcance de las variables

- El alcance se refiere a desde dónde es accesible una variable dentro de un algoritmo (también puede ser una función cuando trabajamos con ámbitos de función)
- Hay variables locales y globales

```
var myVariable = 'global';
myOtherVariable = 'global';
function myFunction() {
  var myVariable = 'local';
  return myVariable;
}
function myOtherFunction() {
  myOtherVariable = 'local';
  return myOtherVariable;
}
console.log(myVariable); // global
console.log(myFunction()); // local
console.log(myOtherVariable); // global
console.log(myOtherFunction()); // local
console.log(myOtherVariable); // local
```

Alcance de las variables cont.

- En las estructuras de repeción las llaves no bloquean el alcance de una variable:

```
var div, container = document.getElementById('container');
for (var i = 0; i < 5; i++) {
  div = document.createElement('div');
  div.onclick = function() {
    alert('This is box #' + i);
  };
  container.appendChild(div);
}
```

Tipos

Tipos

- Los tipos de datos disponibles en JavaScript son los siguientes:
  - undefined
  - Object
  - boolean
  - number
  - string
  - Function
  - symbol (ES6)

Tipos cont.

- Para conocer el tipo de una variable, tenemos dos opciones:

```
// como obtener el tipo de una variable
typeof variable;
// comprobar el tipo de una variable
variable instanceof Clase;

// undefined
typeof undefined;
typeof estaVariableQueNoHeInicializadoAun;

// object
typeof null;
typeof { key: 2 };
typeof [15, 4];
typeof new Date();

// boolean
typeof true;
typeof false;
typeof new Boolean(true);
```

Tipos cont.

```
// string
typeof 'hola';

// function
typeof function (){};

// number
typeof 1;
typeof 1.3;
typeof NaN;
typeof Infinity;

// symbol
typeof Symbol();
typeof Symbol('mi simbolo');
```

Objetos y arreglos

- Desde ES2016, la sintaxis de JavaScript ha admitido formas creativas para operar sobre el contendido de los objetos y de los arreglos:
  - Desestructuración de objetos
  - Desestructuración de Arrays
  - Mejora de objetos literales
  - El operador de propagación

Desestructuración de objetos

- La desestructuración por asignacion permite acceder localmente a los campos dentro de un objeto y declarar que valores se utilizaran
- Considere el objeto `sandwich`. Tiene cuatro propiedades, pero solo queremos usar dos de ellas. Podemos buscar *bread* y *meat* para ser utilizadas localmente:

```
const sandwich = {
  bread: 'dutch crunch',
  meat: 'tuna',
  cheese: 'swiss',
  toppings: ['lettuce', 'tomato', 'mustard']
};

const { bread, meat } = sandwich;

console.log(bread, meat); // dutch crunch tuna
```

- El código extrae *bread* y *meat* del objeto y crea variables locales

Desestructuración de objetos cont.

- También podemos desestructurar los argumentos de funciones
- Considere una función que registraría el nombre de una persona como *lord*:

```
const lordify = function (regularPerson) {
  console.log(regularPerson.firstname + ' of Canterbury');
};

const regularPerson = {
  firstname: 'Bill',
  lastname: 'Wilson'
};

lordify(regularPerson); // Bill of Canterbury
```

Desestructuración de objetos cont.

- En lugar de usar la sintaxis **notación punto** para acceder a las propiedades de los objetos, podemos desestructurar los valores que necesitamos de `regularPerson`:

```
const lordify = function ({ firstname }) {
  console.log(firstname + ' of Canterbury');
};
const regularPerson = {
  firstname: 'Bill',
  lastname: 'Wilson'
};
lordify(regularPerson); // Bill of Canterbury
```

Desestructuración de objetos cont.

- En el siguiente ejemplo, el objeto `regularPerson` tiene un objeto anidado en la propiedad `spouse`:

```
const regularPerson = {
  firstname: 'Bill',
  lastname: 'Wilson',
  spouse: {
    firstname: 'Phil',
    lastname: 'Wilson'
  }
};
```

Desestructuración de objetos cont.

- Si quisiéramos hacer *lord* el nombre de `spouse`, deberíamos ajustar los argumentos desestructurados de la función:

```
const lordify = function ({ spouse: { firstname } }) {
  console.log(firstname + ' of Canterbury');
};
lordify(regularPerson); // Phil of Canterbury
```

Desestructuración arreglos

- Los valores también se pueden desestructurar a partir de arreglos
- Podemos asignar el primer valor de un arreglo a un nombre de variable

```
const [firstAnimal] = ['Horse', 'Mouse', 'Cat'];

console.log(firstAnimal); // Horse
```

- También podemos pasar por alto valores innecesarios con la coincidencia de posiciones usando comas
- La coincidencia de posiciones ocurre cuando las comas reemplazan a los elementos que deben omitirse
- Usando el mismo ejemplo anterior, podemos acceder al último valor reemplazando los dos primeros valores con comas:

```
const [, , thirdAnimal] = ['Horse', 'Mouse', 'Cat'];

console.log(thirdAnimal); // Cat
```

Mejora de objetos literales

- La mejora de objetos literales es lo opuesto a la desestructuración
- Es el proceso de reestructurar o volver a armar el objeto
- Por ejemplo, podemos tomar variables del alcance global y agregarlas a un objeto:

```
const name = 'Tallac';
const elevation = 9738;

const funHike = { name, elevation };

console.log(funHike); // {name: 'Tallac', elevation: 9738}
```

- `name` y `elevation` ahora son propiedades del objeto `funHike`

Mejora de objetos literales cont.

- También podemos crear métodos de objetos con mejora o reestructuración de objetos literales:

```
const name = 'Tallac';
const elevation = 9738;
const print = function() {
  console.log('Mt. ' + this.name + ' is ' + this.elevation + ' feet tall');
};

const funHike = { name, elevation, print };

funHike.print(); // Mt. Tallac is 9738 feet tall
```

Mejora de objetos literales cont.

- Al definir métodos de objetos, ya no es necesario usar la palabra clave de `function`:

```
// Antes
var skier = {
  name: name,
  sound: sound,
  powderYell: function() {
    var yell = this.sound.toUpperCase();
    console.log(yell + ' ' + yell + ' ' + yell + '!!!!');
  },
  speed: function(mph) {
    this.speed = mph;
    console.log('speed:', mph);
  }
};
```

Mejora de objetos literales cont.

```
// Ahora
const skier = {
  name,
  sound,
  powderYell() {
    let yell = this.sound.toUpperCase();
    console.log(yell + ' ' + yell + ' ' + yell + '!!!');
  },
  speed(mph) {
    this.speed = mph;
    console.log('speed:', mph);
  }
};
```

El operador de propagación

- El operador de propagación (...) nos permite realizar varias tareas diferentes
- Podemos utilizarlo para combinar el contenido de arreglos
- Por ejemplo, si tuviéramos dos arreglos, podríamos hacer un tercer arreglo que combine los dos arreglos en uno:

```
const peaks = ['Tallac', 'Ralston', 'Rose'];
const canyons = ['Ward', 'Blackwood'];
const tahoe = [...peaks, ...canyons];

console.log(tahoe); // [ 'Tallac', 'Ralston', 'Rose', 'Ward', 'Blackwood' ]
```

El operador de propagación cont.

- Usando el arreglo `peaks` del ejemplo anterior, imaginemos que queremos tomar el último elemento del arreglo en lugar del primero
- Podríamos usar el método `Array.reverse` para invertir el arreglo en combinación con la desestructuración de la matriz

```
const peaks = ['Tallac', 'Ralston', 'Rose'];
const [last] = peaks.reverse();

console.log(last); // Rose
console.log(peaks); // [ 'Rose', 'Ralston', 'Tallac']
```

- La función `reverse` en realidad ha alterado o mutado el arreglo

El operador de propagación cont.

- Utilizando el operador de propagación, no tenemos que mutar el arreglo original. En su lugar, podemos crear una copia de la matriz y luego invertirla:

```
const peaks = ['Tallac', 'Ralston', 'Rose'];
const [last] = [...peaks].reverse();

console.log(last); // Rose
console.log(peaks); // [ 'Tallac', 'Ralston', 'Rose' ]
```

- Dado que usamos el operador de propagación para copiar el arreglo, el arreglo `peaks` queda intacto y se puede usar más adelante en su forma original

El operador de propagación cont.

- El operador de propagación también se puede usar para obtener los elementos restantes del arreglo:

```
const lakes = ['Donner', 'Marlette', 'Fallen Leaf', 'Cascade'];

const [first, ...others] = lakes;

console.log(others); // [ 'Marlette', 'Fallen Leaf', 'Cascade' ]
```

El operador de propagación cont.

- También podemos usar la sintaxis de tres puntos para agrupar los argumentos de una función como un arreglo
- Ejemplo: la siguiente función toma una cantidad `n` de argumentos usando el operador de propagación, luego usa esos argumentos para imprimir algunos mensajes de la consola:

```
function directions(...args) {
  let [start, ...remaining] = args;
  let [finish, ...stops] = remaining.reverse();

  console.log('drive through ' + args.length + ' towns');
  console.log('start in ' + start);
  console.log('the destination is ' + finish);
  console.log('stopping ' + stops.length + ' times in between');
}

directions('Truckee', 'Tahoe City', 'Sunnyside', 'Homewood', 'Tahoma');
```

El operador de propagación cont.

- El operador de propagación también se puede usar para objetos

```
const morning = {
  breakfast: 'oatmeal',
  lunch: 'peanut butter and jelly'
};

const dinner = 'mac and cheese';

const backpackingMeals = {
  ...morning,
  dinner
};

console.log(backpackingMeals);

// {
//   breakfast: 'oatmeal',
//   lunch: 'peanut butter and jelly',
//   dinner: 'mac and cheese'
// }
```

Control de flujo

Control de flujo

- Los programas en JavaScript son un conjunto de sentencias que se van a ejecutar
- De forma predeterminada, el intérprete de JavaScript ejecuta estas sentencias de forma secuencial (una tras otra en el orden en que se escriben)
- Para alterar este orden de ejecución predeterminado, JavaScript tiene una serie de sentencias o estructuras de control:
  - **Condicionales:** sentencias como `if` y `switch` hacen que el intérprete de JavaScript ejecute u omita otras sentencias dependiendo del valor de una expresión
  - **Repetición:** sentencias como `while` y `for` ejecutan otras sentencias repetitivamente
  - **Saltos:** sentencias como `break`, `return` y `throw` hacen que el intérprete salte a otra parte del programa

Estructuras condicionales

- Las sentencias condicionales ejecutan u omiten otras sentencias según el valor de una expresión específica
- Estas sentencias son los puntos de decisión dentro del código y, a veces, también se conocen como "`ramas`".

Sentencia if

- La sentencia `if` es la sentencia de control fundamental que permite a JavaScript tomar decisiones o, más precisamente, ejecutar sentencias condicionalmente
- Esta declaración tiene dos formas. La primera es:

```
if (expresion)
    sentencia
```

- Se evalúa la expresión y si el valor resultante es verdadero, se ejecuta la instrucción
- Si la expresión es falsa, la sentencia no se ejecuta

```
if (username == null)   // Si el username es null o undefined,
    username = 'John Doe'; // lo definimos
```

Sentencia if cont.

- Los paréntesis alrededor de la expresión son una parte necesaria de la sintaxis de la instrucción `if`
- La sintaxis de JavaScript requiere una sentencia única después de la palabra clave `if`
- Se puede usar un bloque de sentencias para combinar varias sentencias en una
- La segunda forma de la sentencia `if` introduce una cláusula `else` que se ejecuta cuando expresión es falsa
- Su sintaxis es:

```
if (expresion)
    sentencia1
else
    sentencia2
```

- En este caso la sentencia `if` ejecuta la *sentencia1* si la expresión es verdadera y ejecuta la *sentencia2* si la expresión es falsa

Sentencia else if

- La sentencia `if/else` evalúa una expresión y ejecuta uno de dos fragmentos de código, según el resultado
- ¿Qué pasa cuando es necesario ejecutar uno de muchos bloques de código?
- Una forma de hacer esto es con una sentencia `else if`

```
if (n === 1) {
    // bloque de codigo #1
} else if (n === 2) {
    // bloque de codigo #2
} else if (n === 3) {
    // bloque de codigo #3
} else {
    // Si todo lo anterior es falso, bloque de codigo #4
}
```

Sentencia switch

- Como vimos, se puede usar el idioma `else if` para realizar una bifurcación multidireccional
- Esta no es la mejor solución cuando todas las ramas dependen del valor de una misma expresión
- En ese caso, es un desperdicio evaluar repetidamente esa expresión en múltiples sentencias `if`
- La instrucción `switch` maneja exactamente esta situación
- La palabra clave `switch` va seguida de una expresión entre paréntesis y un bloque de código entre llaves

```
switch (expresion) {
    sentencias
}
```

Sentencia switch cont.

- Varios lugares en el bloque de código están etiquetadas con la palabra clave `case` seguida de una expresión y dos puntos
- Cuando se ejecuta `switch`, calcula el valor de expresión y luego busca una etiqueta `case` cuya expresión evalúe como el mismo valor (donde la uniformidad está determinada por el operador `===`)
- Si encuentra uno, comienza a ejecutar el bloque de código en la declaración etiquetada `case`
- Si no encuentra un `case` con un valor coincidente, busca una declaración etiquetada como `default`
- Si no hay una etiqueta `default`, la sentencia `switch` omite el bloque de código por completo

Sentencia switch cont.

```
switch(n) {
  case 1:
    // Codigo del bloque 1
    break;
  case 2:
    // Codigo del bloque 2
    break;
  case 3:
    // Codigo del bloque 3
    break;
  default:
    // Codigo del bloque 4
    break;
}
```

Sentencias de repetición

- Las sentencias de repetición son aquellas que desvían el flujo de ejecucion sobre sí mismo para repetir partes de su código
- JavaScript tiene cinco sentencias de repeticion:
  - `while`
  - `do/while`
  - `for`
  - `for/of` (con su variante `for/await`)
  - `for/in`

Sentencia while

- la instrucción `while` es la estructura de repeticion básica de JavaScript
- Tiene la siguiente sintaxis:

```
while (expresion)
    sentencia
```

- Para ejecutar una instrucción `while`, el intérprete primero evalúa la expresión
- Si el valor de la expresión es falso, el intérprete se salta la sentencia que sirve como cuerpo del ciclo y pasa a la siguiente sentencia en el programa
- Si, por el contrario, la expresión es verdadera, el intérprete ejecuta la sentencia, saltando nuevamente a la parte superior del bucle, evaluando la expresión de nuevo

Sentencia do/while

- El ciclo `do/while` es como el ciclo `while`, excepto que la expresión del ciclo se prueba en la parte inferior del ciclo en lugar de en la parte superior
- Esto significa que el cuerpo del bucle siempre se ejecuta al menos una vez
- La sintaxis es:

```
do
    sentencia
while (expresion);
```

- El ciclo `do/while` se usa con menos frecuencia que su primo `while`

Sentencia do/while cont.

- ¿Cual es la diferencia entre `while` y `do/while`?

Sentencia for

- La mayoría de las estructuras de repeticion tienen una variable contador de algún tipo
- Esta variable se inicializa antes de que comience la estructura de repeticion y se verifica antes de cada iteración del ciclo
- Finalmente, la variable de contador se incrementa o se actualiza de alguna manera al final del cuerpo del ciclo, justo antes de que la variable se vuelva a probar
- En este tipo de estructura de repeticion, la inicialización, la verificación y la actualización son las tres manipulaciones cruciales de una variable de bucle
- La declaración `for` codifica cada una de estas tres manipulaciones como expresiones y convierte esas expresiones en una parte explícita de la sintaxis del bucle:

```
for (inicializacion; prueba; incremento)
    sentencia
```

- *inicialización*, *verificación* e *incremento* son tres expresiones (separadas por punto y coma) que son responsables de inicializar, probar e incrementar la variable de ciclo respectivamente

Sentencia for cont.

- La forma más sencilla de explicar cómo funciona la estructura `for` es mostrar la estructura `while` equivalente

```
inicializacion;
while (prueba) {
    sentencia;
    incremento;
}
```

- A veces múltiples variables cambian con cada iteración del ciclo
- Esta situación es el único lugar donde el operador *coma* se usa comúnmente en JavaScript

Sentencia for cont.

- Proporciona una forma de combinar múltiples expresiones de inicialización e incremento en una sola expresión adecuada para usar en una estructura `for`

```
let i, j, sum = 0;
for (i = 0, j = 10; i < 10; i++, j--) {
    sum += i * j;
}
```

- Cualquiera de las tres expresiones puede omitirse de un ciclo for, pero se requieren los **;**
- Si omite la expresión de verificación, el ciclo se repite por siempre
- `for(;;)` es otra forma de escribir un ciclo infinito, como `while(true)`

Sentencia for cont.

```
for (let i = 0; i < 10; i += 1) {
  console.log(i)
}

// a veces nos olvidamos que también es posible decrementar
for (let i = 10; i > 0; i -= 1) {
  console.log(i)
}

// las expresiones en for no tienen por qué ser individuales
for (let i = 10, j = 0; i > j; i -= 1) {
  console.log(i, j)
}
```

Sentencia for/of

- ES6 define una nueva instrucción de repeticion: `for/of`
- Este nuevo tipo utiliza la palabra clave `for`, pero es un tipo de estructura de repeticion completamente diferente al bucle `for` normal
- El bucle `for/of` funciona con **objetos iterables**:
  - arreglos
  - strings
  - Set
  - Map
- Podemos usar `for/of` para recorrer los elementos de un arreglo de números y calcular su suma:

```
let data = [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ], sum = 0;
for (let element of data) {
    sum += element;
}
```

Sentencia for/of con objetos

- Los objetos no son (por defecto) iterables
- Intentar usar `for/of` en un objeto regular arroja un `TypeError` en tiempo de ejecución:

```
let o = { x: 1, y: 2, z: 3 };
for (let element of o) { // Throws TypeError because o is not iterable
    console.log(element);
}
```

- Si desea iterar a través de las propiedades de un objeto, puede usar la estructura `for/of` con `Object.keys()`

```
let o = { x: 1, y: 2, z: 3 };
let keys = '';
for (let k of Object.keys(o)) {
    keys += k;
}
```

Sentencia for/of con cadenas

- Si estámos interesados tanto en las claves como en los valores de las propiedades de un objeto, podemos usar `for/of` con `Object.entries()` y la asignación de desestructuración:

```
let pairs = '';
for (let [k, v] of Object.entries(o)) {
    pairs += k + v;
}
```

- `Object.entries()` devuelve un arreglo de arreglos, donde cada arreglo interno representa un par clave/valor para una propiedad del objeto
- Usamos la asignación de desestructuración en este ejemplo de código para desempaquetar esos arreglos internos en dos variables individuales

Sentencias for/of con cadenas

- Las cadenas son iterables carácter por carácter en ES6:

```
let frequency = {};
for (let letter of 'mississippi') {
    if (frequency[letter]) {
        frequency[letter]++;
    } else {
        frequency[letter] = 1;
    }
}
```

- Las cadenas se iteran por el código Unicode, no por carácter UTF-16

Sentencia for/of con Set y Map

- Las clases integradas de ES6 `Set` y `Map` son iterables
- Cuando itera un `Set` con `for/of`, el cuerpo del bucle se ejecuta una vez para cada elemento del conjunto
- Se podría usar un código como el siguiente para imprimir las palabras únicas en una cadena de texto:

```
let text = 'Na na na na na na na na Batman!';
let wordSet = new Set(text.split(' '));
let unique = [];
for (let word of wordSet) {
    unique.push(word);
}
```

Sentencia for/of con Set y Map cont.

- Los `Map` son un caso interesante porque el iterador de un objeto `Map` no itera las claves del mapa o los valores del mapa, sino los pares clave/valor
- Cada vez que se realiza la iteración, el iterador devuelve una matriz cuyo primer elemento es una clave y cuyo segundo elemento es el valor correspondiente
- Dado un `Map m`, podría iterar y desestructurar su clave/valor pares como este:

```
let m = new Map([[1, 'one']]);
for (let [key, value] of m) {
    key
    value
}
```

Sentencia for/in

- Un bucle `for/in` se parece mucho a un bucle `for/of`
- Mientras que un bucle `for/of` requiere un objeto iterable después de `of`, un bucle `for/in` funciona con cualquier objeto después de `in`
- El bucle `for/of` es nuevo en ES6, pero `for/in` ha sido parte de JavaScript desde el principio (es por eso que tiene una sintaxis que suena más natural)
- La instrucción `for/in` recorre los nombres de propiedad de un objeto específico. La sintaxis se ve así:

```
for (variable in objeto)
    sentencia
```

Sentencia for/in

- `Variable` es la declaración de una variable
- `Objeto` es una expresión que se evalúa como un objeto
- `Sentencia` es la sentencia o bloque de sentencias que sirve como cuerpo del bucle

```
for(let p in o) {  // Asignar nombres de propiedades de o a la variable p
  console.log(o[p]); // Imprimir el valor de cada propiedad
}
```

Operador ternario

- `<condición lógica> ? true : false`

```
let edad = prompt('¿Cuantos años tienes?', 0);
```

- Aquí se modifica el mensaje en base a la variable edad

```
const mensaje = edad >= 18 ? 'Perfecto, puedes conducir' : 'No puedes conducir!';
```

- Se pueden ejecutar múltiples expresiones. No es nada recomendable, pero poder, se puede :-)

```
let mensaje, edad = prompt('¿Cuantos años tienes?', 0);
edad >= 18 ? (
  mensaje = 'Perfecto, puedes pasar',
  console.log(mensaje)
) : (
  mensaje = 'No vayas tan rápido!',
  alert(mensaje)
);
```

Características adicionales

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 utilizar 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>
`
```

Funciones

Creando funciones

- Cada vez que deseamos realizar algún tipo de tarea repetible con JavaScript, podemos usar una función
- Algunas de las diferentes opciones de sintaxis que se pueden usar para crear una función son las siguientes:
  - Declaraciones de funciones
  - Expresiones de función
  - Función flecha

Declaraciones de funciones

- La declaración o definición de una función comienza con la palabra clave `function`, seguida del nombre de la función
- Las sentencias JavaScript que forman parte de la función se definen entre llaves

```
function logCompliment() {
  console.log("You're doing great!");
}
```

- Una vez definida la función, se invocará o llamará utilizando el nombre que se le asignó:

```
logCompliment();
```

Expresiones de función

- En este caso solo implica crear la función como una variable:

```
const logCompliment = function() {
  console.log("You're doing great!");
};
logCompliment();
```

Declaraciones de funciones frente a expresiones

- Algo a tener en cuenta al decidir entre una declaración de función y una expresión de función es que las declaraciones de función se elevan mientras que las expresiones de función no
- Por lo tanto, puede invocar una función antes de escribir la declaración de la función

```
// Invocando la función antes de que sea declarada
hey();
// Declaración de la función
function hey() {
  alert("hey!");
}
```

- Esto funciona dado que la función se eleva o sube a la parte superior del alcance del archivo

Declaraciones de funciones frente a expresiones cont.

- No es posible invocar una función creada por una expresión de función antes de la expresión
- Si probamos el mismo ejercicio con una expresión de función causará un error:

```
// Invocando la función antes de que sea declarada
hey();
// Expresión de función
const hey = function() {
  alert("hey!");
};
TypeError: hey is not a function
```

- Obviamente, este es un pequeño ejemplo, pero este `TypeError` puede surgir ocasionalmente al importar archivos y funciones en un proyecto

Pasaje de argumentos

- Si queremos proporcionar variables dinámicas a la función, podemos pasar parámetros con nombre a una función simplemente agregándolos a los paréntesis
- Comencemos agregando una variable `firstName`:

```
const logCompliment = function(firstName) {
  console.log("You're doing great, " + firstName);
};
logCompliment("Molly");
```

- Ahora, cuando llamamos a la función `logCompliment`, el valor `firstName` enviado como argumento se agregará al mensaje de la consola

Retorno de funciones

- La función `logCompliment` utiliza la consola para visualizar un mensaje
- En general una mejor práctica es definir una función que devuelva un valor
- Una declaración de `return` especifica el valor retornado por la función

```
const createCompliment = function(firstName, message) {
  return firstName + ': ' + message;
};
createCompliment("Molly", "You're so cool");
```

Parámetros predeterminados

- Los parámetros predeterminados se incluyen en la especificación ES6, por lo que, en caso de que no se proporcione un valor para el argumento, se utilizará el valor predeterminado
- Por ejemplo, podemos configurar cadenas predeterminadas para los parámetros `name` y `activity`:

```
function logActivity(name = "Shane McConkey", activity = "skiing") {
  console.log(name + ' loves ' + activity);
}
```

- Si no se proporcionan argumentos a la función `logActivity`, se ejecutará correctamente utilizando los valores predeterminados

Parámetros predeterminados cont.

- Si no se proporcionan argumentos a la función `logActivity`, se ejecutará correctamente con los valores predeterminados
- Los argumentos predeterminados pueden ser de cualquier tipo, no solo cadenas

```
const defaultPerson = {
  name: {
    first: "Shane",
    last: "McConkey"
  },
  favActivity: "skiing"
};

function logActivity(person = defaultPerson) {
  console.log(person.name.first + ' loves ' + person.favActivity);
}
```

Función flecha

- Las funciones flecha son una característica nueva y útil de ES6
- Con funciones flecha, puede crear funciones sin usar la palabra clave de `function`
- Además, a menudo no es necesario usar la palabra clave `return`
- Consideremos una función que toma `name` y devuelve una cadena, convirtiendo a la persona en un `lord`

```
const lordify = function(firstName) {
  return firstName + ' of Canterbury';
};
console.log(lordify("Dale")); // Dale of Canterbury
console.log(lordify("Gail")); // Gail of Canterbury
```

- Con una función flecha, podemos simplificar enormemente la sintaxis:

```
const lordify = firstName => firstName + ' of Canterbury';
```

Función flecha cont.

- Con la función flecha, ahora tenemos la declaración de una función completa en una sola línea
- Si la función solo toma un argumento, podemos eliminar los paréntesis alrededor de los argumentos
- Para más de un argumento se deben usar los paréntesis:

```
// Función clásica
const lordify = function(firstName, land) {
  return firstName ' + of + ' land;
};

// Función flecha
const lordify = (firstName, land) => firstName + ' of ' + land;

console.log(lordify("Don", "Piscataway")); // Don of Piscataway
console.log(lordify("Todd", "Schenectady")); // Todd of Schenectady
```

Función flecha cont.

- Si el cuerpo de la función está compuesta por varias sentencias, debemos utilizar llaves:

```
const lordify = (firstName, land) => {
  if (!firstName) {
    throw new Error("A firstName is required to lordify");
  }

  if (!land) {
    throw new Error("A lord must have a land");
  }

  return firstName + ' of ' + land;
};

console.log(lordify("Kelly", "Sonoma")); // Kelly of Sonoma
console.log(lordify("Dave")); // ! JAVASCRIPT ERROR
```

Devolviendo objetos

- ¿Qué sucede si queremos devolver un objeto? Consideremos una función llamada `person` que construye un objeto basado en parámetros `firstName` y `lastName`:

```
const person = (firstName, lastName) =>
  {
    first: firstName,
    last: lastName
  }

console.log(person("Brad", "Janson"));
```

- Tan pronto como ejecute esto, verá el error:
  `Uncaught SyntaxError: Unexpected token :`

Devolviendo objetos cont.

- Para solucionar esto, simplemente envuelva el objeto que está devolviendo entre paréntesis:

```
const person = (firstName, lastName) => ({
  first: firstName,
  last: lastName
});

console.log(person("Flad", "Hanson"));
```

Gestión de la memoria

Introducción

- La mayoría de las veces, probablemente podamos desarrollar aplicaciones JavaScript sin saber nada sobre la administración de memoria
- Después de todo, el motor de JavaScript se encarga de la gestión de memoria por nosotros
- Sin embargo, en algún momento pueden aparecer problemas, como fugas de memoria, que solo puede resolver si sabe cómo funciona la asignación de memoria

Ciclo de vida de la memoria

- En JavaScript, cuando creamos variables, funciones, etc ..., el motor JS asigna memoria y la libera una vez que ya no se necesita
- La asignación de memoria es el proceso de reservar espacio en la memoria, mientras que la liberación de memoria libera espacio dejandola disponible para otro propósito
- Cada vez que asignamos una variable o creamos una función, la memoria utilizada para ese propósito pasa por las siguientes etapas:
  - **Asignar memoria**: JavaScript se encarga de esto por nosotros: asigna la memoria que necesitaremos para el objeto que creamos
  - **Usar memoria**: Usar la memoria es algo que hacemos explícitamente en nuestro código: leer y escribir en la memoria no es más que leer o escribir desde o hacia una variable
  - **Liberar memoria**: Este paso también es manejado por el motor de JavaScript. Una vez que se libera la memoria asignada, se puede usar para un nuevo propósito
- "*Objetos*" en el contexto de la gestión de memoria no solo incluye objetos JS sino también funciones

El Heap y el Stack de memoria

- ¿Dónde se almacena toda esa memoria que se asigna?
- Los motores de JavaScript tienen dos lugares donde pueden almacenar datos:
  - El **Heap** de memoria
  - El **Stack** de memoria
- El Heap y el Stack son dos estructuras de datos que el motor utiliza para diferentes propósitos

Stack: asignación de memoria estática

- Una pila es una estructura de datos que utiliza JavaScript para almacenar datos estáticos
- Los datos estáticos son datos en los que el motor conoce el tamaño
- En JavaScript, esto incluye valores primitivos (cadenas, números, booleanos, undefined y null) y referencias que apuntan a objetos y funciones

```
const male = true
const name = 'John Doe'
const age = 24
const adult = true
```

Stack: asignación de memoria estática cont.

- Dado que el motor sabe que el tamaño no cambiará, asignará una cantidad fija de memoria para cada valor
- El proceso de asignación de memoria justo antes de la ejecución se conoce como asignación de memoria estática
- Debido a que el motor asigna una cantidad fija de memoria para estos valores, existe un límite de cuán grandes pueden ser los valores primitivos
- Los límites de estos valores y el tamaño del Stack varían según el navegador

Heap: asignación de memoria dinámica

- El Heap es un espacio diferente para almacenar datos donde JavaScript almacena objetos y funciones
- A diferencia del Stack, el motor no asigna una cantidad fija de memoria para estos objetos. En cambio, se asignará más espacio según sea necesario
- La asignación de memoria de esta manera también se denomina asignación de memoria dinámica

Stack vs Heap

- Para obtener una descripción general, estas son las características de los dos tipos de almacenamiento en una aplicación JavaScript:
  - **Pila**
    - Valores primitivos y referencias
    - El tamaño se conoce en tiempo de compilación
    - Asigna una cantidad fija de memoria
  - **Heap**
    - Objetos y funciones
    - El tamaño se conoce en tiempo de ejecución
    - Sin límite por objeto

Ejemplos

- JavaScript asigna memoria para el siguiente objeto en el Heap

```
const person = {
  name: 'John',
  age: 24,
};
```

- Los arreglos también son objetos, por lo que se almacenan en el Heap

```
const hobbies = ['hiking', 'reading'];
```

- Los valores primitivos son inmutables, lo que significa que en lugar de cambiar el valor original, JavaScript crea uno nuevo.

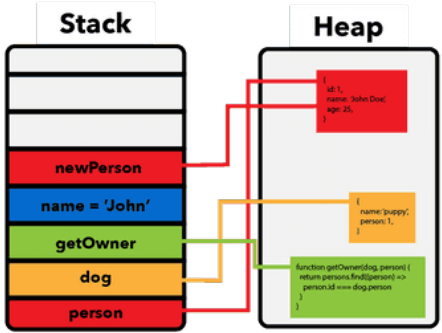
```
let name = 'John'; // reserva memoria para la cadena
const age = 24; // reserva memoria para el número

name = 'John Doe'; // reserva memoria para una nueva cadena
const firstName = name.slice(0,4); // reserva memoria para una nueva cadena
```

Referencias en JavaScript

- Todas las variables se almacenan en el Stack
- En caso de que sea un valor no primitivo, el Stack contiene una referencia al objeto en el Heap
- La memoria del Heap no está ordenada de ninguna manera en particular, por lo que debemos mantener una referencia a ella desde el Stack

Referencias en JavaScript cont.

```
const person = {
  id: 1,
  name: 'John',
  age: 25,
}

const dog = {
  name: 'puppy',
  personId: 1,
}

function getOwner(dog, persons) {
  return persons.find((person) =>
    person.id === dog.person
  )
}

const name = 'John';

const newPerson = person;
```



Recolección de basura

- Una vez que el motor de JavaScript reconoce que una determinada variable o función ya no es necesaria, libera la memoria que ocupaba
- El encargado de realizar esta tarea es el **recolector de basura**
- Existen distintos algoritmos para la recoleccion de basura:
  - La recolección de basura de conteo de referencias
  - El algoritmo de marca y barrido

Paso por Valor y por referencia

Introducción

- Dependiendo del tipo de dato que pasamos como argumento a una función, podemos diferenciar dos comportamientos:
  - Paso por valor: Se crea una copia local de la variable dentro de la función
  - Paso por referencia: Se maneja directamente la variable, los cambios realizados dentro de la función le afectarán también fuera
- Tradicionalmente:
  - Los tipos simples se pasan por valor: enteros, flotantes, cadenas, booleanos...
  - Los tipos compuestos se pasan por referencia: Listas, diccionarios, conjuntos...

Paso por valor de valores primitivos

- En el paso por valor, la función se llama pasando directamente el valor de la variable como argumento
- La función que recibe el argumento, crea una copia local de dicho valor
- Por lo tanto, cambiar el valor del argumento dentro de la función no afecta el valor de la variable que se paso desde fuera de la función
- Es importante tener en cuenta que en JavaScript, todos los argumentos de función siempre se pasan por valor
- Es decir, JavaScript copia los valores de las variables de paso en argumentos dentro de la función

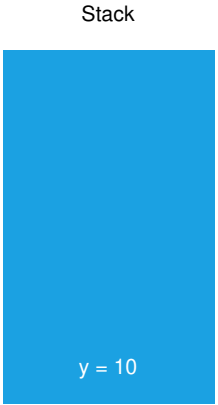
Paso por valor de valores primitivos cont.

- Veamos el siguiente ejemplo

```
function square(x) {
  x = x * x;
  return x;
}

let y = 10;
let result = square(y);

console.log(result);
console.log(y);
```

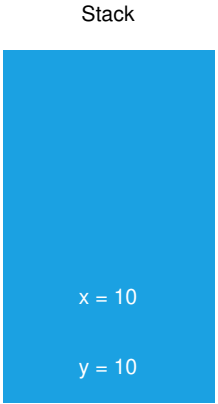
Paso por valor de valores primitivos cont.

- En primer lugar definimos una función `square()` que acepta un argumento `x`
- La función asigna el cuadrado de `x` al argumento `x`
- A continuación, declaramos la variable `y` e inicializamos su valor en `10`



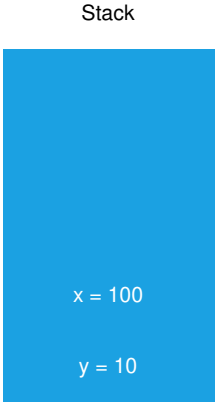
Paso por valor de valores primitivos cont.

- Luego, pasamos la variable `y` a la función `square()`
- Al pasar la variable `y` a la función `square()`, JavaScript copia el valor `y` a la variable `x`



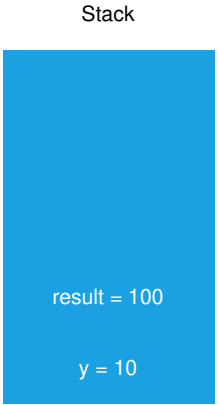
Paso por valor de valores primitivos cont.

- Luego de esto, la función `square()` cambia el valor de la variable `x`
- Sin embargo, el cambio no afecta el valor de la variable `y` porque `x` e `y` son variables independientes



Paso por valor de valores primitivos cont.

- Finalmente, el valor de la variable `y` no cambia después de que se completa la ejecución de la función `square()`



Paso por referencia

- Si JavaScript usara el paso por referencia, la variable `y` cambiaría a `100` después de la llamada a la función
- En el paso por referencia, se invoca a una función pasando directamente la **referencia/dirección** de la variable como argumento
- Cambiar el argumento dentro de la función afecta a la variable que se paso desde fuera de la función
- En JavaScript, los objetos y las matrices se pasan por referencia

Pasaje por referencia cont.

```
function callByReference(varObj) {
  console.log('Inside Call by Reference Method');
  varObj.a = 100;
  console.log(varObj);
}

let varObj = {
  a: 1
};

console.log('Before Call by Reference Method');
console.log(varObj);

callByReference(varObj);

console.log('After Call by Reference Method');
console.log(varObj);
```

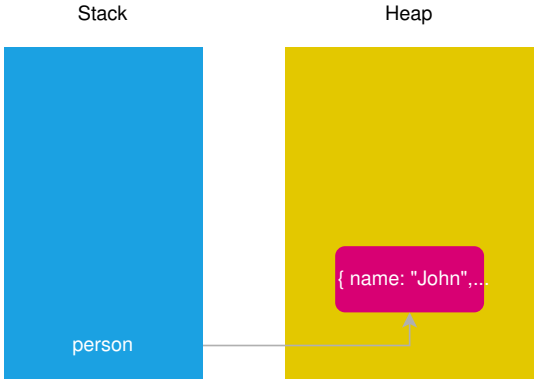
Paso por valor de los valores de referencia

- No es tan obvio ver que los valores de referencia en realidad se pasan también por valor
- Por ejemplo:

```
let person = {
  name: 'John',
  age: 25,
};

function increaseAge(obj) {
  obj.age += 1;
}

increaseAge(person);

console.log(person)
```

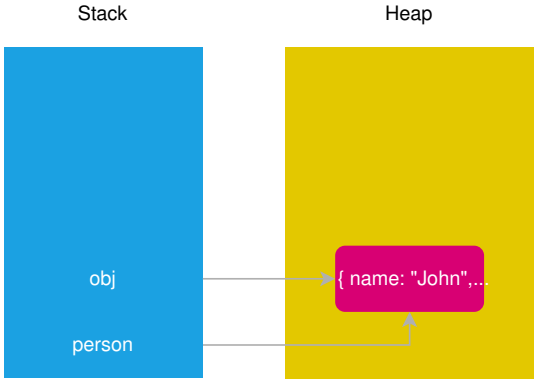
Paso por valor de los valores de referencia cont.

- Primero, definimos la variable `person` que hace referencia a un objeto con dos propiedades `name` y `age`:



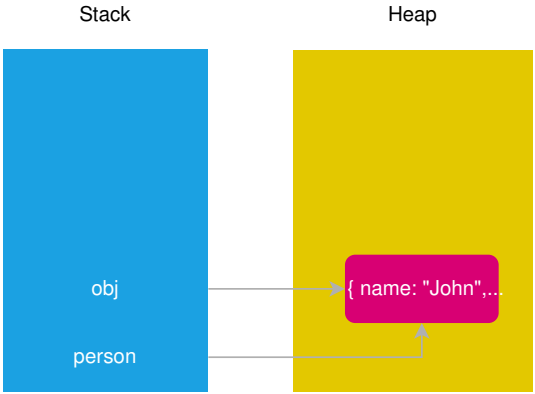
Paso por valor de los valores de referencia cont.

- A continuación, definimos la función `increaseAge()` que acepta un objeto `obj` y aumenta la propiedad `age` del argumento `obj` en uno
- Luego, pasamos el objeto `person` a la función `increaseAge()`:



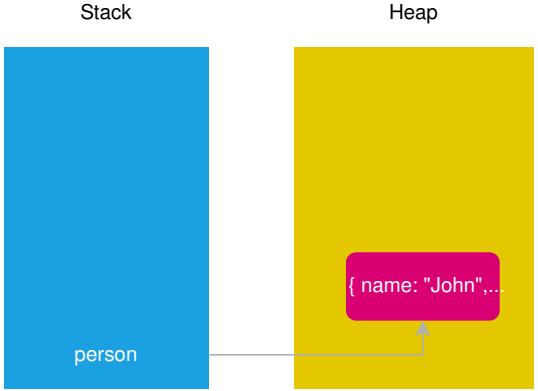
Paso por valor de los valores de referencia cont.

- Internamente, el motor JavaScript crea una referencia `obj` y hace que esta variable haga referencia al mismo objeto al que hace referencia la variable `person`
- Después de eso, se incrementa la propiedad de `age` en uno dentro de la función de `increaseAge()` a través de la variable `obj`



Paso por valor de los valores de referencia cont.

- Finalmente, accediendo al objeto a través de la referencia `person`:



Paso por valor de los valores de referencia cont.

- Al parecer JavaScript pasa un objeto por referencia dado que el cambio en el objeto se refleja fuera de la función. Sin embargo, éste no es el caso
- De hecho, cuando pasa un objeto a una función, está pasando la referencia de ese objeto, no el objeto real
- Por lo tanto, la función puede modificar las propiedades del objeto a través de su referencia
- Sin embargo, hay que tener en cuenta que no es posible cambiar la referencia original que se paso a la función dentre de la propia función

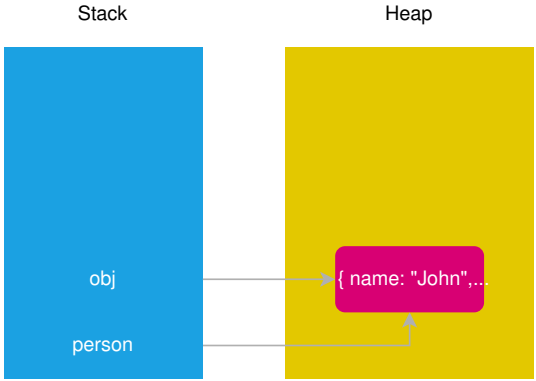
Paso por valor de los valores de referencia cont.

- Por ejemplo:

```
let person = {
  name: 'John',
  age: 25,
};

function increaseAge(obj) {
  obj.age += 1;

  // reference another object
  obj = { name: 'Jane', age: 22 };
}

increaseAge(person);

console.log(person);
```

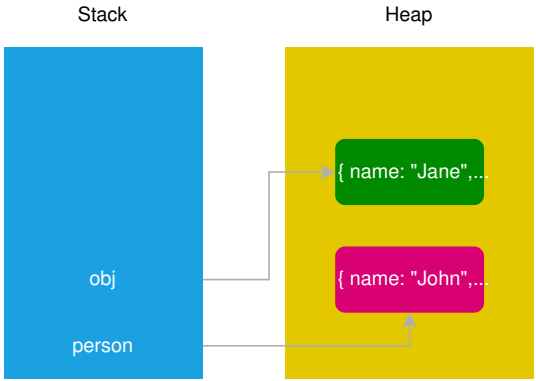
Paso por valor de los valores de referencia cont.

- En este ejemplo, la función de `increaseAge()` cambia la propiedad `age` a través del argumento `obj`:



Paso por valor de los valores de referencia cont.

- Pero, dentro de la función modifica la referencia de `obj` hacia otro objeto:



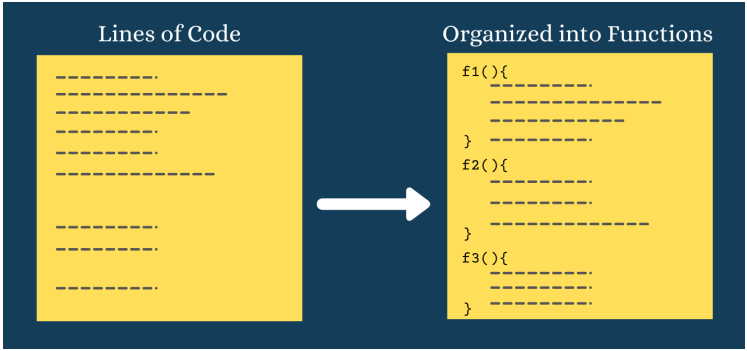
Paso por valor de los valores de referencia cont.

- Sin embargo, la referencia de la persona aún refiere al objeto original cuya propiedad `age` cambió a `26`
- En otras palabras, la función de `increaseAge()` no cambia la referencia de `person`

Motor de JavaScript

Funciones como unidad fundamental

- En JavaScript, podemos crear y modificar una función, usarla como argumento, devolverla desde otra función y asignarla a una variable
- Todas estas habilidades nos permiten usar funciones en todas partes y de esa forma agrupar código lógicamente



Funciones como unidad fundamental cont.

- Como vimos anteriormente, si necesitamos decirle al motor de JavaScript que ejecute funciones lo hacemos de la siguiente manera:

```
  // Definimos la función
function f1() {
    // Hacemos algo
    // Hacemos algo nuevamente
    // Nuevamente
    // Y asi sucesivamente...
}

// Invocamos la función
f1();
```

- De forma predeterminada, cada línea de una función se ejecuta secuencialmente, una línea a la vez
- Lo mismo es aplicable incluso cuando invocamos múltiples funciones en el código, nuevamente línea por línea

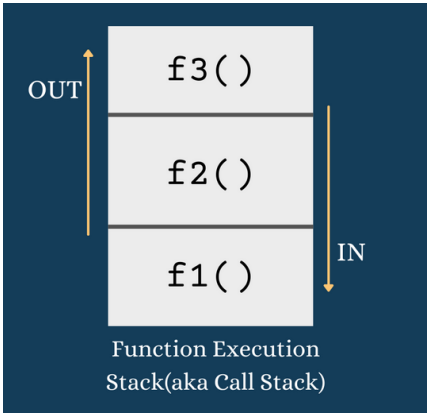
Pila de ejecución de funciones

Entonces, ¿qué sucede cuando definimos una función y luego la invocamos?

- El motor de JavaScript mantiene una estructura de datos de tipo Stack llamada **pila de ejecución de funciones**
- El propósito del Stack es rastrear la función actual en ejecución, es decir:
  - Cuando el motor de JavaScript invoca una función, la agrega al Stack y comienza la ejecución
  - Si la función actualmente en ejecución llama a otra función, el motor agrega la segunda función al Stack y comienza a ejecutarla
  - Una vez que termina de ejecutar la segunda función, el motor la saca del Stack
  - El control vuelve a reanudar la ejecución de la primera función desde el punto en que la dejó la última vez
  - Una vez terminada la ejecución de la primera función, el motor la saca del Stack
  - Continúa de la misma manera hasta que no haya nada que poner en el Stack

Pila de ejecución de funciones cont.

La pila de ejecución de funciones también se conoce como **pila de llamadas**



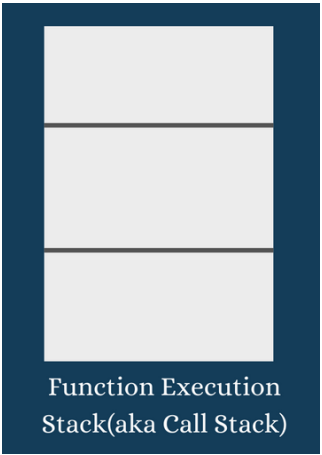
Pila de ejecución de funciones cont.

Veamos un ejemplo de tres funciones que se ejecutan una por una:

```
function f1() {
  // algo de código
}
function f2() {
  // algo de código
}
function f3() {
  // algo de código
}

// Invocamos las funciones una a la vez
f1();
f2();
f3();
```

Pila de ejecución de funciones cont.

Ahora veamos qué sucede con la pila de ejecución de funciones:



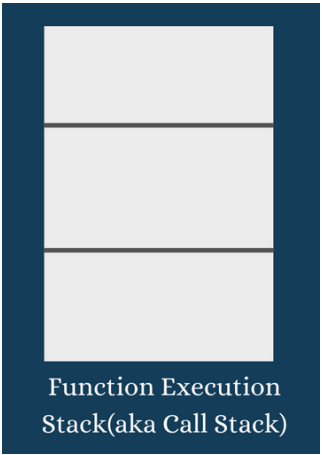
Pila de ejecución de funciones cont.

Ahora veamos un ejemplo más complejo. Aquí hay una función `f3()` que invoca otra función `f2()` que a su vez invoca otra función `f1()`.

```
function f1() {
  // algo de código
}
function f2() {
  f1();
}
function f3() {
  f2();
}

f3();
```

Pila de ejecución de funciones cont.

Veamos qué está pasando con la pila de ejecución de funciones:



Pila de ejecución de funciones cont.

- En primer lugar `f3()` entra en el Stack, invocando o la tra función `f2()`
- Ahora `f2()` se ejecuta mientras `f3()` permanece en el Stack
- Luego, la función `f2()` invoca `f1()`
- `f1()` entra en el Stack junto con `f2()` y `f3()`
- Primero, `f1()` termina de ejecutarse y sale del Stack. Justo después de que termine `f2()`, y finalmente `f3()`
- Por lo tanto, todo lo que sucede dentro de la pila de ejecución de funciones es secuencial
- Esta es la parte síncrona de JavaScript. El subproceso principal de JavaScript se asegura de que se ocupa de todo lo que hay en la pila antes de que comience a buscar en cualquier otro lugar

Cómo funcionan las API y las promesas del navegador

- Por lo general, ejecutar instrucciones en secuencia funciona bien
- Pero a veces es posible que necesite obtener datos del servidor o ejecutar una función con un retraso, o algo que no está previsto que ocurra AHORA
- Entonces, deseamos que el código se ejecute de forma asincrónica
- En estas circunstancias, es posible que no queramos que el motor de JavaScript detenga la ejecución del otro código secuencial
- El motor de JavaScript necesita administrar las cosas de manera más eficiente en este caso

Cómo funcionan las API y las promesas del navegador cont.

Podemos clasificar la mayoría de las operaciones de JavaScript como asincrónicas con dos disparadores principales:

- Eventos o funciones de la API del navegador o API web: Estos incluyen métodos como `setTimeout` o controladores de eventos como hacer click, pasar el mouse por encima, desplazarse y muchos más
- Promesas: Un objeto JavaScript que nos permite realizar operaciones asíncronas

Cómo manejar las API del navegador o API web

- Las API del navegador como `setTimeout` y los controladores de eventos se basan en funciones **callback**
- Una función callback se ejecuta cuando se completa una operación asincrónica
- Aquí hay un ejemplo de cómo funciona una función `setTimeout`:

```
function printMe() {
  console.log('print me');
}

setTimeout(printMe, 2000);
```

- La función `setTimeout` ejecuta la función después de que haya transcurrido una cierta cantidad de tiempo
- En el código anterior, el texto *print me* se muestra por consola después de un retraso de 2 segundos

Cómo manejar las API del navegador o API web cont.

Ahora supongamos que tenemos algunas líneas más de código justo después de la función `setTimeout` como esta:

```
function printMe() {
  console.log('print me');
}

function test() {
  console.log('test');
}

setTimeout(printMe, 2000);
test();
```

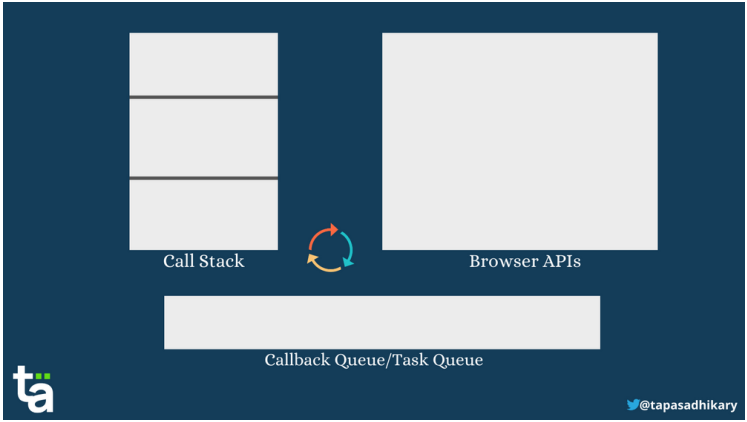
¿Cuál será la salida en este caso?

Cola de tareas

- JavaScript mantiene una cola de funciones callback también llamada cola de tareas
- Una estructura de datos de tipo cola es FIFO (Primero en entrar, primero en salir)
- Entonces, la función callback que primero ingresa a la cola tiene la oportunidad de salir primero. Pero la pregunta es:
  - ¿Cuándo el motor de JavaScript lo pone en la cola?
  - ¿Cuándo el motor de JavaScript lo saca de la cola?
  - ¿Adónde va cuando sale de la cola?

Cola de tareas cont.

La imagen muestra el Stack de llamadas regular. Hay dos secciones adicionales para rastrear si una API del navegador (como `setTimeout`) se activa y pone en cola la función callback de esa API



Cola de tareas cont.

- El motor de JavaScript sigue ejecutando las funciones en la pila de llamadas
- Dado que no coloca la función callback directamente en la pila, no existe ningún código esperando/bloqueando la ejecución en la pila
- El motor crea un ciclo para buscar en la cola periódicamente para encontrar lo que necesita extraer de allí
- Extrae una función callback de la cola al Stack de llamadas cuando el Stack esté vacío
- Ahora, la función callback se ejecuta generalmente como cualquier otra función en la pila
- El ciclo continúa y es lo que se conoce como el ciclo de eventos o `event loop`

Cola de tareas cont.

Resumiendo:

- Cuando se produce una llamada a la API de navegador, las funciones callback se agregan a la cola
- El código continua ejecutandose en la pila
- El ciclo de eventos comprueba si hay una función callback en la cola
- Si hay alguna función en la cola y el Stack está vacío, movemos la función callback de la cola al Stack y la ejecutamos
- Continúe el ciclo

Cola de tareas cont.

Veamos cómo funciona con el siguiente código:

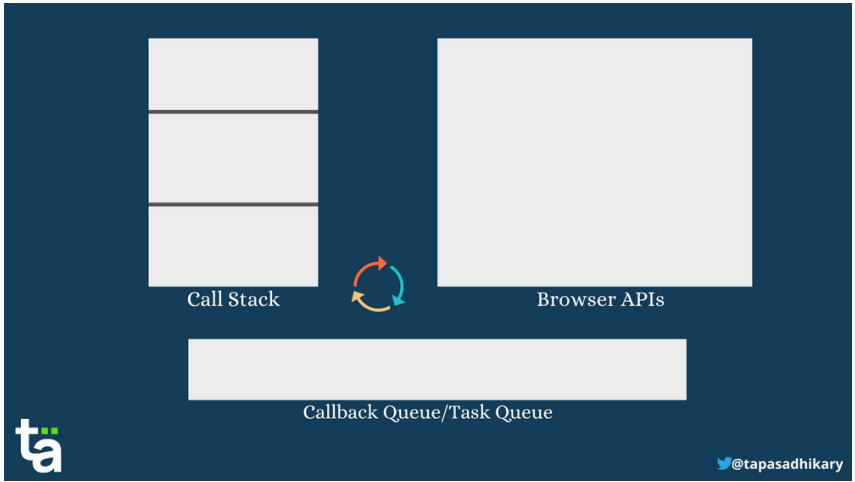
```
function f1() {
  console.log('f1');
}

function f2() {
  console.log('f2');
}

function main() {
  console.log('main');

  setTimeout(f1, 0);

  f2();
}

main();
```

Cola de tareas cont.

El código ejecuta una función `setTimeout` con una función callback `f1()`. A la función callback le hemos dado un retraso de `cero`. Esto significa que esperamos que la función `f1()` se ejecute inmediatamente. Luego de `setTimeout` ejecutamos otra función `f2()`
¿Cuál será el resultado en este caso?

Cola de tareas cont.

Ahora, veamos en un flujo paso a paso para el código anterior.



Cola de tareas cont.

La secuencia de pasos sería la siguiente:

- La función `main()` entra en el Stack de llamada
- Tiene una llamada a la consola para imprimir la palabra `main`, que se ejecuta y sale del Stack
- Se invoca a la API del navegador `setTimeout`
- La función callback se coloca en la cola de callbacks
- En el Stack, la ejecución ocurre como de costumbre, por lo que `f2()` entra en el Stack. Se ejecuta la llamada a la consola de `f2()`. Ambos salen del Stack
- El `main()` también sale del Stack
- El ciclo de eventos reconoce que en el Stack está vacío y que hay una función callback en la cola
- La callback `f1()` se mueve al Stack. Comienza su ejecución. Se ejecuta la llamada por consola y `f1()` también sale del Stack
- En este punto, no hay nada más en el Stack y en la cola para ejecutar

Gestión de promesas

- Aquí hay un ejemplo de una promesa en JavaScript:

```
const promise = new Promise((resolve, reject) =>
  resolve('I am a resolved promise');
);
```

Gestión de promesas cont.

- Después de ejecutar la promesa, podemos manejar el resultado usando el método `.then()` y cualquier error con el método `.catch()`

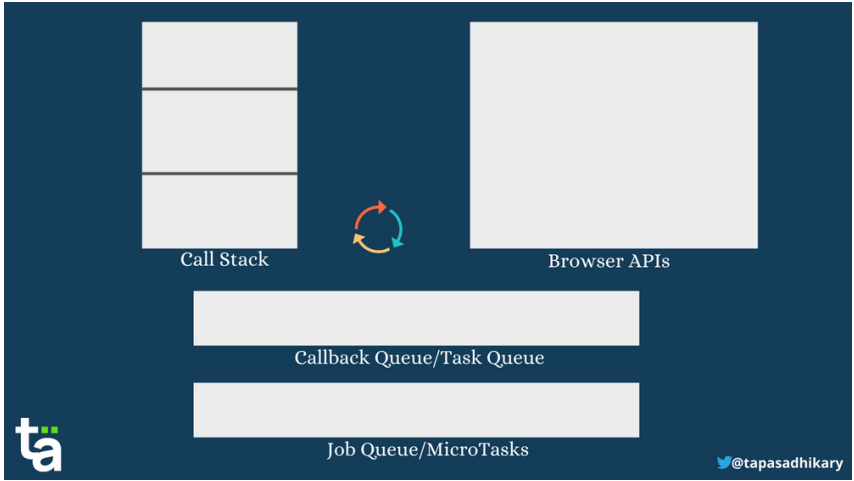
```
promise.then(result => console.log(result));
```

- Por ejemplo, utilizamos promesas cada vez que llamamos a la función `fetch()` para obtener algunos datos del backend
- El motor de JavaScript no usa la misma cola de callbacks que hemos visto anteriormente para las API del navegador
- Utiliza otra cola especial llamada **Cola de trabajos**

Cola de trabajos

- Cada vez que se produce una promesa en el código, la función a ejecutora entra en la cola de trabajos
- El ciclo de eventos examinar ambas colas, pero da prioridad a los elementos de la cola de trabajos sobre los elementos de la cola de callbacks cuando el Stack está libre
- El elemento de la cola de callbacks se denomina **macro-tarea**, mientras que el elemento de la cola de trabajos se denomina **micro-tarea**
- Todo el flujo de trabajo quedaría así:
  - Para cada ciclo del ciclo de eventos, se completa una tarea fuera de la cola de callbacks
  - Una vez que se completa esa tarea, el ciclo de eventos revisa la cola de trabajos. Completa todas las micro-tareas en la cola de trabajos antes de buscar lo siguiente
  - Si ambas colas obtuvieron entradas en el mismo momento, la cola de trabajo tiene preferencia sobre la cola de callbacks

Cola de trabajos cont.



Cola de trabajos cont.

Veamos un ejemplo para entender mejor esta secuencia:

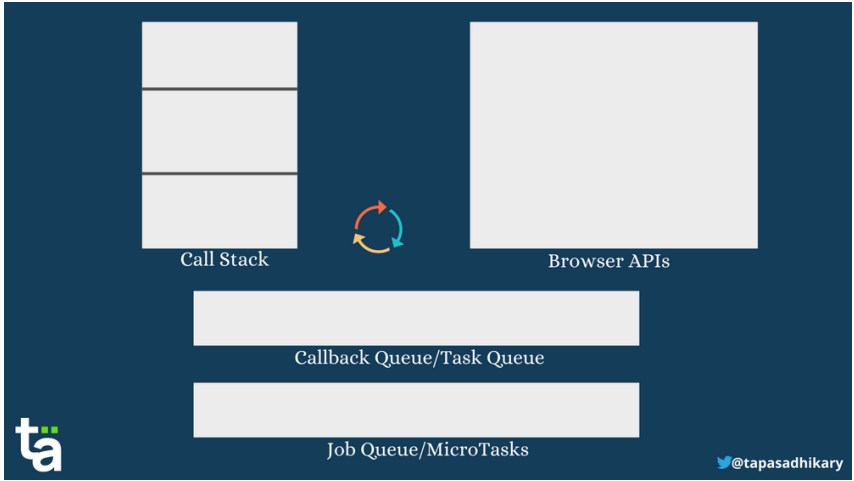
```
function f1() {
    console.log('f1');
}

function f2() {
    console.log('f2');
}

function main() {
    console.log('main');

    setTimeout(f1, 0);

    new Promise((resolve, reject) =>
        resolve('I am a promise')
    ).then(resolve => console.log(resolve))

    f2();
}

main();
```

Cola de trabajos cont.

- En el código anterior, tenemos una función `setTimeout()` como antes, pero hemos introducido una promesa justo después
- La salida debería ser la siguiente:

```
main
f2
I am a promise
f1
```

Cola de trabajos cont.



Más información

- https://nodejs.dev/learn/the-nodejs-event-loop

JavaScript Asincrónico

Introducción

- Hay diferentes formas de manejar el código asincrónico
  - Callbacks
  - Promesas
  - Async/Await

Callbacks

- En JavaScript, las funciones son objetos, por lo tanto podemos pasar objetos a funciones como parámetros
- También podemos pasar funciones como parámetros a otras funciones y llamarlas dentro de las funciones
- Los callbacks son funciones que se pasan como argumento a otra función
- Tomemos un ejemplo de función callback:

```
function printString() {
  console.log("Tom");
  setTimeout(function() { console.log("Jacob"); }, 3000);
  console.log("Mark");
}

printString();
```

Callbacks cont.

- Si ese fuera código sincrónico, habríamos encontrado el siguiente resultado

```
Tom
Jacob
Mark
```

- Pero `setTimeout` es una función asíncrona, entonces la salida del código anterior será:

```
Tom
Mark
Jacob
```

Callbacks cont.

- Hay un método incorporado en JavaScript llamado `setTimeout`, que llama a una función o evalúa una expresión después de un período de tiempo determinado (en milisegundos).
- En otras palabras, la función que muestra el mensaje por consola se llama después de que algo sucedió (después de que pasaron 3 segundos para este ejemplo), pero no antes
- En este caso, el callback es la función que se pasa como argumento a `setTimeout`

Callback utilizando función flecha

- También puede escribir la misma función callback anterior como una función flecha:

```
function printString(){
    console.log("Tom");
    setTimeout(() =>  { console.log("Jacob"); }, 3000);
   console.log("Mark");
}

printString();
```

Callback utilizando función flecha cont.

- La salida será la misma que la anterior
- El problema con las callbacks es que pueden llegar a crear algo llamado *"Infierno de callbacks"*
- Básicamente, se comienzan a anidar funciones dentro de funciones dentro de funciones, y comienza a ser muy difícil leer el código
- Una forma de solucionar dicho problema es utilizar `Promises`, las cuales manejan las callbacks de forma mas prolija

Callback utilizando función flecha cont.

- Infierno de callbacks:

```
function pyramidOfDoom() {
  setTimeout(() => {
    console.log(1);
    setTimeout(() => {
      console.log(2);
      setTimeout(() => {
        console.log(3);
      }, 500);
    }, 2000);
  }, 1000);
}
```

Promesas

- Una promesa en JavaScript es similar a una promesa en la vida real
- Cuando hacemos una promesa en la vida real, es una garantía de que vamos a hacer algo en el futuro (las promesas solo se pueden hacer para el futuro)
- Una promesa tiene dos resultados posibles: se cumplirá cuando llegue el momento o no se cumplirá

Promesas cont.

- Esto mismo pasa en las promesas de JavaScript
- Cuando definimos una promesa en JavaScript, se resolverá cuando llegue el momento, o se rechazará
- Una promesa se usa para manejar el resultado asíncrono de una operación
- JavaScript está diseñado para no esperar a que un bloque de código asíncrono se ejecute por completo antes de que puedan ejecutarse otras partes síncronas del código
- Con Promises, podemos diferir la ejecución de un bloque de código hasta que se complete una solicitud asíncrona
- De esta manera, otras operaciones pueden seguir ejecutándose sin interrupción

Estados de las promesas

- En primer lugar, una Promesa es un objeto
- Hay 3 estados por los que puede pasar el objeto Promesa:
  - **Pendiente**: Estado Inicial, antes de que la Promesa tenga éxito o falle
  - **Resuelto**: Se completa la promesa
  - **Rechazado**: Promesa fallida, lanza un error
- Por ejemplo, cuando solicitamos datos del servidor mediante una Promesa, estará en modo pendiente hasta que recibamos nuestros datos
- Si conseguimos obtener la información del servidor, la Promesa se resolverá con éxito. Pero si no obtenemos la información, entonces la Promesa estará en estado rechazado

Creando una promesa

- Para crear una promesa usamos el constructor del objeto Promise
- La promesa tiene dos parámetros, uno para el éxito (`resolve`) y otro para el fracaso (`reject`):

```
const myPromise = new Promise((resolve, reject) => {
  // ejecutor
});
```

- Vamos a crear una promesa:

```
const myFirstPromise = new Promise((resolve, reject) => {
  const condition = true;
  if(condition) {
    setTimeout(function(){
      resolve("Promise is resolved!"); // fulfilled
    }, 300);
  } else {
    reject('Promise is rejected!');
  }
});
```

Creando una promesa cont.

- En la promesa anterior, si la condición es verdadera, resuelve la promesa devolviendo "La promesa está resuelta"
- De lo contrario, devuelva un error "La promesa es rechazada"
- Ahora veamos como utilizarla

Usando la promesa

- Para usar la Promesa que creamos anteriormente, usamos el método `then()` para resolverla y el método `catch()` para rechazarla

```
myFirstPromise.then((successMsg) => {
  console.log(successMsg);
})
.catch((errorMsg) => {
  console.log(errorMsg);
});
```

- O definiendo la promesa dentro de una función:

```
const demoPromise = function() {
  myFirstPromise
    .then((successMsg) => {
      console.log("Success:" + successMsg);
    })
    .catch((errorMsg) => {
      console.log("Error:" + errorMsg);
    });
}
```

Usando la promesa cont.

- En la promesa que creamos, la condición es `true`, y llamamos a `demoPromise()`, luego por consola veremos la salida:

```
Success: Promise is resolved!
```

- Entonces, si la promesa es rechazada, saltará al método `catch()` y esta vez veremos un mensaje diferente en la consola

```
Error: Promise is rejected!
```

¿Qué es el encadenamiento?

- A veces necesitamos llamar a múltiples solicitudes asincrónicas
- Luego, después de que se resuelva (o rechace) la primera Promesa, comenzará un nuevo proceso al que podemos adjuntarlo directamente mediante un método llamado encadenamiento.
- Entonces creamos otra promesa:

```
const helloPromise  = function() {
  return new Promise(function(resolve, reject) {
    const message = `Hi, How are you!`;

    resolve(message);
  });
}
```

¿Qué es el encadenamiento? cont.

- Encadenamos esta promesa a nuestra operación anterior "myFirstPromise" así:

```
const demoPromise = function() {

  myFirstPromise
  .then(helloPromise)
  .then((successMsg) => {
      console.log("Success:" + successMsg);
  })
  .catch((errorMsg) => {
      console.log("Error:" + errorMsg);
  })
}

demoPromise();
```

- Dado la condición es `true`, la salida por consola es:

```
Success: Hi, How are you!
```

- Una vez que la `helloPromise` se encadena con `.then`, el subsiguiente `.then` utilizara los datos del anterior

Async/Await

- `Await` es básicamente lo que se conoce como **azúcar sintáctico** para Promesas
- Usando `Async/Await` hace que el código asincrónico se parezca más al código sincrónico, que es más fácil de entender para los humanos
- Sintaxis de `Async` y `Await`:

```
async function printMyAsync() {
  await printString("one");
  await printString("two");
  await printString("three");
}
```

Async/Await cont.

- Puede ver que usamos la palabra clave `async` para la función contenedora `printMyAsync`
- Esto le permite a JavaScript saber que estamos usando la sintaxis `async/await`, y es necesario si desea usar `Await`
- Esto significa que no puede usar Await a nivel global
- Siempre necesita una función contenedora, o podemos decir que await solo se usa con una función asíncrona
- La palabra clave `await` se usa en una función `async` para garantizar que todas las promesas retornadas en la función asíncrona estén sincronizadas
- `Await` elimina el uso de callbacks en `.then()` y `.catch()`
- Al usar `async` y `await`, `async` se antepone al devolver una promesa, `await` se antepone al llamar a una promesa
- `try` y `catch` también se utilizan para obtener el valor de rechazo de una función asíncrona

Async/Await cont.

- Tomemos un ejemplo para comprender Async y Await con nuestro `demoPromise`:

```
async function demoPromise() {
  try {
    let message1 = await myFirstPromise;
    let message2 = await helloPromise();
    console.log(message2);
  } catch(error) {
      console.log("Error:" + error.message);
  }
}
```

Laboratorio práctico - JavaScript Labs

Es momento de jugar con la sintaxis y estructura de JavaScript

- Arreglos
- Control de flujo
- Nuevas características de ES6
- Promesas
- Async/await

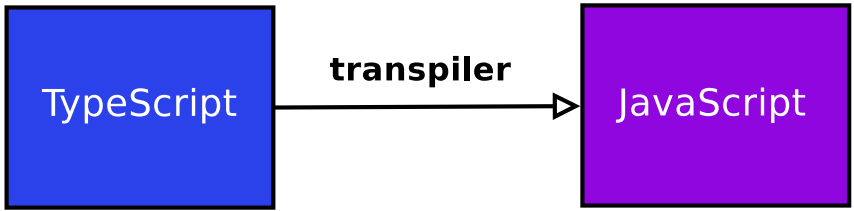
Typescript

Luciano Diamand

Introducción

Introducción

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



Características

- Anotaciones de Tipos
- Clases
- Interfaces
- Decoradores
- Constructores
- Modificadores de acceso
- Importación y exportación de módulos
- Enums
- Tuplas
- Uniones
- Generics

Caracteristicas cont.

Qué agrega TypeScript:

- Tipos: Con los tipos, tenemos que ser mucho más explícitos sobre cómo funcionan las cosas y podemos evitar muchos errores inesperados e innecesarios al usar tipos. Además de eso, también podemos usar un IDE, que tenga compatibilidad integrada con TypeScript, que puede captar esos tipos y brindarnos un mejor autocompletado y errores integrados, que se muestran incluso antes de que compilemos el código
- Funciones de JavaScript de próxima generación: Podemos usar ciertas características de JavaScript de próxima generación, que podemos escribir y usar en nuestros archivos TypeScript y luego se compilarán en código JS con implementaciones alternativas que incluso funcionarán en navegadores más antiguos

Caracteristicas cont.

- Caracteristicas que no son de JavaScript: interfaces, generics: Estas son características que se pueden compilar en JS, y que nos ayudan durante el desarrollo y brindan errores más claros y a evitar errores comunes
- Metaprogramación - Decoradores: También nos brinda ciertas funciones de metaprogramación, como los decoradores: un tipo especial de declaración que se puede adjuntar a una declaración de clase, método, elemento de acceso, propiedad o parámetro

Caracteristicas cont.

- TS es altamente configurable: puede ajustarse muy bien a nuestros requisitos, hacerlo más estricto, más flexible o simplemente asegurarnos de que se comporte de la manera que queremos
- Herramientas modernas: Con los IDEs modernos, como VSCode, también obtenemos soporte para proyectos TypeScript como la verificación de tipo y el autocompletado

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 podemos usar el comando `tsc` (typescript compiler):

  ```
  tsc archivo.ts
  ```
- Luego podemos probar la aplicación:

  ```
  node archivo.js
  ```

¿Por qué usamos TypeScript?

- Consideremos el siguiente ejemplo de JavaScript:

```
function add(num1, num2) {
  return num1 + num2;
}

console.log(add('2', '3'));
```

- Si invocamos la función con números se realizará la suma de los mismos
- Sin embargo, es posible invocar la función pasando números como cadenas o valores de cualquier tipo, lo que podría resultar en un comportamiento no deseado

¿Por qué usamos TypeScript? cont.

- En este ejemplo, JavaScript concatenará esas dos cadenas, que no es lo que queríamos hacer: el resultado será `23` en lugar de `5` (pasamos dos cadenas, evaluadas por el operador de suma)
- Esto no arrojará ningún error en tiempo de ejecución, es más bien un error lógico el cual podría generar grandes problemas con la aplicación que estamos desarrollando
- Por supuesto, tenemos varias formas o estrategias de mitigarlo
  - Podríamos agregar verificaciones `if`
  - Validar y corregir la entrada del usuario, etc.
- Todo esto está bien, pero también podríamos detectar errores durante el desarrollo, lo cual es posible con TypeScript
- TypeScript es lenguaje que nos ayuda a escribir código mejorado

Debilidad de JavaScript: tipado débil

- Como vimos anteriormente, JavaScript posee un tipado débil
- Consideremos otro ejemplo (omitiendo `.html` ya que solo contiene el formulario):

```
const button = document.querySelector('button');
const input1 = document.getElementById('num1');
const input2 = document.getElementById('num2');

function add(num1, num2) {
  return num1 + num2;
}

button.addEventListener('click', function() {
  console.log(add(input1.value, input2.value));
})
```

- Al ejecutar este código con los números `10` y `5`, esperaríamos obtener un resultado de `15`. En su lugar, vemos `105`

Debilidad de JavaScript: tipado débil cont.

- ¿De dónde viene este error?. Llenamos nuestro formulario y en el evento `click`, pasamos dos valores y los sumamos
- Hay que recordar que el tipo de retorno de nuestro `input1.value` es **siempre** una `cadena`, que luego se pasa a nuestra función `add`, y no importa si la entrada es de tipo `number`, el valor recuperado siempre es sera `cadena`
- Por lo tanto, dado que ambos valores son cadenas, no se suman, sino que siempre se concatenan
- Este es un problema con JavaScript con el que TypeScript puede ayudarnos
- Podríamos agregar una sentencia `if` y verificar si esos valores son números y luego devolver la suma o `else` convertirlos en un número anteponiendo un `+`: `return +num1 + +num2`
- La "*solución*" anterior funciona, pero requiere que escribiamos código adicional

Instalación

Instalación

- Para comenzar a trabajar con TypeScript vamos a instalarlo
- Para ello vamos a la sección de descargas (`https://www.typescriptlang.org/download`) en donde disponemos de distintas formas de instalación
- Vamos a realizar la instalación a traves de `npm` de forma global
  `sudo npm install -g typescript`
- Una vez realizada la instalación de TypeScript, estará disponible de forma global en nuestra máquina
- Además se instalará el compilador de TypeScript `tsc`

Proyecto TypeScript

- Los archivos TypeScript se pueden compilar con el comando `tsc <archivo>.ts`
- Es tedioso compilar varios archivos `.ts` en un proyecto grande
- Por lo tanto, TypeScript proporciona otra opción para compilar todos o ciertos archivos `.ts` del proyecto
- Puede proporcionar opciones de compilación que se pueden almacenar en el archivo `tsconfig.json`
- Para iniciar un nuevo proyecto de TypeScript, ingrese al directorio raíz de su proyecto y en una ventana de terminal ejecute `tsc --init`
- Este comando generará un archivo de `tsconfig.json` con un mínimo de opciones de configuración, similar al siguiente

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "strict": true,
    "esModuleInteop": true,
    "forceConsistentCasingInFileNames": true
  }
}
```

Archivo tsconfig.json

- El detalle de las opciones disponibles se puede consultar en: `https://www.typescriptlang.org/docs/handbook/tsconfig-json.html`
- Un detalle del mínimo de opciones generadas es (con su valor predeterminado entre paréntesis):
  - target (es5): especifica la versión de destino de ECMAScript (JavaScript) a la que se adherirá el JavaScript generado: es3, es5, es2015, es2016, es2017, es2018, es2019 o esnext
  - module (commonjs): especifica el sistema de carga de módulos que se usará: none, commonjs, amd, system, umd, es2015 o esnext
  - strict (true): habilita todas las opciones estrictas de comprobación de tipo
  - esModuleInterop (true): permite la generación de código de interoperabilidad para habilitar la interoperabilidad entre los módulos CommonJS y ES a través de la creación de objetos de espacio de nombres para todas las importaciones

Pruebas con TypeScript

- Para poder probar de forma simple programas en TypeScript podemos utilizar una herramienta llamada **TSUN** (TypeScript Upgraded Node)
- La instalamos de la siguiente manera: `sudo npm install -g tsun`
- Ahora podemos ejecutarlo con: `tsun`
- Tambien hay otras opciones como ser `ts-node`:
  `sudo npm install -g ts-node`
- O de forma on-line `http://typescriptlang.org`
- Si estamos usando `nvm` no es necesario hacer `sudo`

Usando TypeScript

Usando TypeScript

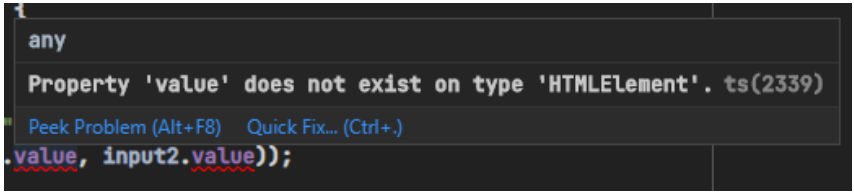
- Para compilar TypeScript podemos ejecutar el comando `tsc <archivo.ts>`
- Simplemente tenemos que crear un archivo nuevo con extensión `.ts` (que es la extensión `TypeScript`)
- Si estamos usando una estructura junto con el archivo de configuración `tsconfig.json` es suficiente con ejecutar `tsc`
- Vamos a probar el código del ejemplo anterior sobre "*la debilidad de JS*" y analizar la salida

Usando TypeScript cont.

- Incluso podemos intentar compilar inmediatamente este código, lo que nos arrojará un error:

```
error TS2339: Property 'value' does not exist on type 'HTMLElement'.
```

- Eso no solo se muestra en la compilación, también nos mostrará este error en nuestro IDE:



- TypeScript, nos obliga a ser más explícitos con nuestro código

Usando TypeScript cont.

- En aras de la exhaustividad del ejemplo anterior, proporcionamos algunos comentarios:

```
/**
 * Dado que estamos tratando con HTML, y ya nos hemos asegurado de
 * que nuestros elementos existen, podemos decirle a TS que nuestro
 * .value nunca será null (porque existe) agregando un signo de
 * exclamación al final del elemento asignado:
 *
 * const input1 = document.getElementById("num1")!;
 *
 * En este ejemplo, tenemos que decirle a TS qué tipo de elemento es,
 * por lo que lo convertiremos al tipo HTMLInputElement. Esta es la
 * sintaxis de TypeScript y podemos usarla en un archivo .ts.
 */
const button = document.querySelector("button");
const input1 = document.getElementById("num1")! as HTMLInputElement;
const input2 = document.getElementById("num2")! as HTMLInputElement;
```

Usando TypeScript cont.

```
/**
 * La ventaja adicional es que podemos definir tipos.
 * Al principio, TS nos dirá que no sabe de qué tipo es el argumento (num: any),
 * y sería bueno agregarle un tipo para que sepamos con qué estamos tratando.
 *
 * Mediante el uso de la sintaxis específica, que comprende el compilador de TS,
 * podemos definir el tipo de argumentos. Hacemos esto agregando dos puntos después
 * de un argumento y especificando un tipo
 */
function add(num1: number, num2: number) {
  return num1 + num2;
}

/**
 * Nuestro IDE nos advertira por la falta de coincidencia de tipos.
 * value es de tipo cadena y estamos tratando de pasar una cadena como argumento
 * a un parámetro que debería ser de tipo numérico.
 *
 * Este error también aparecerá cuando intentemos compilar el siguiente addEventListener.
 * TypeScript entiende qué tipo obtenemos de un InputElement, por lo que no podemos
 * pasar esto a la función add(), dado que espera un número. Podemos
 * convertirlo rápidamente en un número anteponiendo un signo más.
 */
button.addEventListener("click", function() {
  console.log(add(+input1.value, +input2.value));
})
```

Usando TypeScript cont.

- Ahora podemos compilar este código con `tsc filename.ts`
- Se compilará con éxito y obtendremos un archivo `.js` a cambio
- Vemos que obtuvimos casi el mismo código que antes, pero en Vanilla JS
- Cuando compilamos el código, esas funciones solo se usan para evaluar nuestro código para encontrar posibles errores y luego se eliminan
- Es cierto, también tuvimos que escribir código adicional, pero nos vimos obligados a hacerlo de una manera más limpia, lo que da como resultado un código mejor y menos propenso a errores

Tipos

Tipos

- El agregado mas importante de TypsScript sobre ES6 es el sistema de tipos
- La verificación de tipos:
  - Ayuda al escribir código; dado que previene errores en tiempo de compilación
  - Ayuda al leer el código; dado que clarifica las intenciones
- Hay que mencionar que el uso de tipos en Typescript es *opcional*
- Los tipos básicos de TypeScript son los mismos que usabamos en JavaScript: `string`, `number`, `boolean`, 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;
```

Funciones

- Cuando declaramos funciones también 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`
- Si intentamos retornar un entero dentro de la función, ¿que sucede?

Funciones

- En Typescript, el compilador espera que una función reciba el número exacto y el tipo de argumentos definidos en la firma de la función
- Si la función espera tres parámetros, el compilador comprueba que el usuario haya pasado valores para los tres parámetros, es decir, comprueba si hay coincidencias exactas

```
function greet(greeting: string, name: string): string {
  return greeting + ' ' + name + '!';
}

greet('Hello', 'Steve');
greet('Hi');
greet('Hi', 'Bill', 'Gates');
```

Parámetros opcionales en funciones

- Todos los parámetros opcionales deben seguir a los parámetros requeridos y deben estar al final

```
function greet(greeting: string, name?: string): string {
  return greeting + ' ' + name + '!';
}

greet('Hello', 'Steve');
greet('Hi');
greet('Hi', 'Bill', 'Gates');
```

- En el ejemplo anterior, el nombre del segundo parámetro está marcado como opcional con un signo `?`
- Por lo tanto, la función `greet` acepta uno o dos parámetros y devuelve una cadena de saludo
- Si no especificamos el segundo parámetro, su valor será `undefined`

Sobrecarga de funciones

- Typescript proporciona el concepto de sobrecarga de funciones
- Puedemos tener múltiples funciones con el mismo nombre pero diferentes tipos de parámetros y tipos de retorno
- Sin embargo, el número de parámetros debe ser el mismo

```
function add(a: string, b: string): string;

function add(a: number, b: number): number;

function add(a: any, b: any): any {
  return a + b;
}

add('Hello ', 'Steve');
add(10, 20);
```

- No se admite la sobrecarga de funciones con diferentes números de parámetros

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 de elementos, 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' ];
```

Tipo any

- Typescript tiene verificación de tipo y verificación en tiempo de compilación
- Sin embargo, no siempre tenemos conocimiento previo sobre el tipo de algunas variables, especialmente cuando hay valores ingresados por el usuario de bibliotecas de terceros
- En tales casos, necesitamos un tipo que pueda tratar con contenido dinámico
- `any` es el tipo por defecto si omitimos el tipo para una variable dada
- Una variable de tipo `any` permite recibir cualquier tipo de valor:

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

let arr: any[] = [ 'John', 212, true ];
arr.push('Smith');
console.log(arr);
```

Tipo void

- Similar a lenguajes como Java, `void` se usa donde no hay ningún tipo de datos
- Por ejemplo, en tipo de retorno de funciones que no devuelven ningún valor

```
function sayHi(): void {
  console.log('Hi!');
}

let speech: void = sayHi();
console.log(speech);
```

- No tiene sentido asignar `void` a una variable, ya que solo `null` o `undefined` se puede asignar a `void`

Tipo never

- TypeScript introduce un nuevo tipo `never`, que indica los valores que nunca ocurrirán
- El tipo `never` se usa cuando se está seguro de que algo nunca va a ocurrir
- Por ejemplo, se escribe una función que no volverá a su punto final o que siempre genera una excepción

```
function throwError(errorMsg: string): never {
  throw new Error(errorMsg);
}

function keepProcessing(): never {
  while (true) {
    console.log('I always does something and never ends.');
  }
}
```

Tupla

- TypeScript introduce un nuevo tipo de dato llamado `tupla`
- Permite expresar un arreglo donde el tipo de elementos es fijo y 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

let user: [ number, string, boolean, number, string ];
user = [ 1, 'Steve', true, 20, 'Admin' ];
```

Tupla cont.

- Es posible declarar un arreglo de tuplas:

```
let employees: [ number, string ][];
employee = [ [ 1, 'Steve' ], [ 2, 'Bill'], [ 3, 'Jeff' ]];
```

- y agregar elemento a la tupla:

```
var employee: [ number, string ] = [ 1, 'Steve' ];
employee.push(2, 'Bill');
console.log(employee);
```

Enums

- Existen tres tipos de enumeraciones:
  - Numéricas
  - Cadenas
  - Heterogeneas

Enums numéricas

- Las *enumeraciones* funcionan nombrando valores numéricos:

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

- El valor por defecto de inicio 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]);
```

Enums numéricas cont.

- El valor de un miembro de enumeración puede ser constante o calculado
- La siguiente enumeración incluye miembros con valores calculados:

```
enum PrintMedia {
  Newspaper = 1,
  Newsletter = getPrintedMediaCode('newsletter'),
  Magazine = Newsletter * 3,
  Book = 10
}

function getPrintedMediaCode(mediaName: string): number {
  if (mediaName === 'newsletter') {
    return 5;
  }
  return 0;
}
```

Enums de cadenas

- Las enumeraciones de cadenas son similares a las enumeraciones numéricas, excepto que sus valores son cadenas

```
enum PrintMedia {
  Newspaper = 'NEWSPAPER',
  Newsletter = 'NEWSLETTER',
  Magazine = 'MAGAZINE',
  Book = 'BOOK'
}

console.log(PrintMedia.Newspaper);
console.log(PrintMedia['Magazine']);
```

- La diferencia entre enumeraciones numéricas y de cadena es que los valores de enumeración numéricos se incrementan automáticamente, mientras que los valores de enumeración de cadena deben inicializarse individualmente

Enums de cadenas

- Las enumeraciones heterogéneas son enumeraciones que contienen cadenas y valores numéricos

```
enum Status {
  Active = 'ACTIVE',
  Deactivate = 1,
  Pending
}
```

Unión

- Typescript nos permite usar más de un tipo de datos para una variable o un parámetro de función
- Esto se llama tipo de unión:

```
let code: (string | number);
code = 123;
code = 'ABC';
code = false;

let empId: string | number;
empId = 111;
empId = 'E111';
empId = true;
```

Unión cont.

```
function displayType(code: (string | number)) {
  if (typeof(code) === 'number')
    console.log('Code is number.');
  else
    console.log('Code is string.');
}

displayType(123);
displayType('ABC');
displayType(true);
```

Inferencia de tipos

- No es obligatorio anotar el tipo
- TypeScript infiere tipos de variables cuando no hay información explícita disponible en forma de anotaciones de tipo
- Los tipos son inferidos por el compilador de TypeScript cuando:
  - Las variables se inicializan
  - Los valores predeterminados se establecen para los parámetros.
  - Se determinan los tipos de devolución de función
- Por ejemplo:

```
var a = 'some text';
var b = 123;
a = b;
```

- El código anterior muestra un error porque al inferir tipos, TypeScript infirió el tipo de variable `a` como cadena y la variable `b` como número

Aserción de tipos

- Hay dos formas de hacer una aserción de tipo en TypeScript:
  - Usando la sintaxis <>:

```
let code: any = 123;
let employeeCode = <number> code;
```

  - Usando la palabra clave as:

```
let code: any = 123;
let employeeCode = code as number;
```

- Permite establecer el tipo de un valor y decirle al compilador que no lo infiera
- Tal situación puede ocurrir cuando podría estar transfiriendo código de JavaScript y podría conocer un tipo de variable más preciso que el que está asignado actualmente
- Es similar a la conversión de tipos en otros lenguajes como Java. Sin embargo, a diferencia de Java, no existe un efecto de tiempo de ejecución de la aserción de tipo en Typescript

Generics

Usando generics

- Al escribir programas, uno de los aspectos más importantes es construir componentes reutilizables
- Esto asegura que el programa sea flexible y escalable a largo plazo
- Los genéricos ofrecen una forma de crear componentes reutilizables
- Además proporcionan una manera de hacer que los componentes funcionen con cualquier tipo de datos y no se restrinjan a un tipo de datos particular
- Por lo tanto, los componentes se pueden llamar o usar con una variedad de tipos de datos
- Los genéricos en TypeScript son similares a los genéricos de Java

Usando generics cont.

- Veamos por qué necesitamos genéricos usando el siguiente ejemplo.

```
function getArray(items: any []): any[] {
  return new Array().concat(items);
}
let myNumArr = getArray([100, 200, 300]);
let myStrArr = getArray(['Hello', 'World']);
myNumArr.push(400);
myStrArr.push('Hello TypeScript');
myNumArr.push('Hi');
myStrArr.push(500);
console.log(myNumArr);
console.log(myStrArr);
```

Usando generics cont.

- En el último ejemplo, la función `getarray()` acepta una arreglo de tipo `any`
- Crea un nuevo arreglo de tipo `any` donde le concatena elementos y lo devuelve
- Como usamos type `any` para nuestros argumentos, podemos pasar cualquier tipo de arreglo a la función
- Sin embargo, esto puede no ser el comportamiento deseado. Es posible que deseemos agregar números al arreglo de números o cadenas al arreglo de cadenas, pero no números al arreglo de cadenas o viceversa
- Para resolver esto, TypeScript introdujo generics
- Generics utiliza la variable de tipo `<T>`, un tipo especial de variable que denota tipos
- La variable de tipo recuerda el tipo que proporciona el usuario y trabaja solo con ese tipo en particular
- A esto se le llama preservar la información de tipo

Usando generics cont.

- La función anterior se puede reescribir como una función genérica como se muestra a continuación:

```
function getArray<T>(items: T[]): T[] {
  return new Array<T>().concat(items);
}
let myNumArr = getArray<number>([100, 200, 300]);
let myStrArr = getArray<string>(['Hello', 'World']);
myNumArr.push(400);
myStrArr.push('Hello TypeScript');
myNumArr.push('Hi');
myStrArr.push(500);
```

Variables de tipo multiple

- Podemos especificar múltiples variables de tipo con diferentes nombres como se muestra a continuación:

```
function displayType<T, U>(id: T, name: U): void {
  console.log(typeof(id) + ', ' + typeof(name));
}

displayType<number, string>(1, 'Steve');
```

- El tipo genérico también se puede utilizar con otros tipos no genéricos:

```
function displayType<T>(id: T, name: string): void {
  console.log(typeof(id) + ', ' + typeof(name));
}

displayType<number>(1, 'Steve');
```

Métodos y propiedades de tipo generics

- Cuando usamos variables de tipo para crear componentes genéricos, TypeScript nos obliga a usar solo métodos generales que están disponibles para cada tipo

```
function displayType<T, U>(id: T, name: U): void {
  id.toString();
  name.toString();

  id.toFixed();
  name.toUpperCase();

  console.log(typeof(id) + ', ' + typeof(name));
}
```

- En el ejemplo anterior, las llamadas a los métodos `id.toString()` y `name.toString()` son correctas porque el método `toString()` está disponible para todos los tipos
- Sin embargo, no se pueden llamar métodos específicos de tipo como `toFixed()` para el tipo `number` o `toUpperCase()` para el tipo `string`
- El compilador dará un error

Laboratorio práctico - TypeScript Labs

Es momento de jugar con la sintaxis y estructura de TypeScript

- Tipos básicos
- Funciones
- Enums
- Unions
- Generics

Módulos en JavaScript

Introducción

- Un módulo es una pieza de código dentro de un archivo que se puede llamar y usar desde otros archivos
- Un diseño modular es lo opuesto a tener todo el código de tu proyecto en un solo archivo
- Al desarrollar proyectos de gran tamaño, es muy útil dividir nuestro código en módulos por las siguientes razones:
  - Para dividir características y funciones en diferentes archivos, lo que ayuda a la visualización y organización del código
  - El código tiende a ser más fácil de mantener y menos propenso a errores y fallas cuando está claramente organizado
  - Los módulos se pueden usar y reutilizar fácilmente en diferentes archivos y partes de nuestro proyecto, sin necesidad de volver a escribir el mismo código

Tipos de módulos

- Existen diferentes formas de implementar módulos
- Como JavaScript se creó para ser un lenguaje de secuencias de comandos para sitios web, al principio no poseía la característica de módulos
- A medida que el lenguaje y el ecosistema crecieron, los desarrolladores comenzaron a ver la necesidad de agregar módulos
- Así se desarrollaron diferentes opciones y bibliotecas para agregar esta función a JavaScript
- De las opciones disponibles, solo vamos a revisar a `CommonJS` y `ESmodules`, que son los más recientes y ampliamente utilizados

Módulos CommmonJS

- CommonJS es un conjunto de estándares utilizados para implementar módulos en JavaScript
- El proyecto fue iniciado por el ingeniero de Mozilla Kevin Dangoor en 2009.
- CommonJS se usa principalmente en aplicaciones JS del lado del servidor con Node, ya que los navegadores no admiten el uso de CommonJS
- Node solía admitir solo CommonJS para implementar módulos, pero hoy en día también admite ESmodules, que es un enfoque más moderno.

Módulos CommmonJS cont.

- En primer lugar vamos a crear un archivo `main.js` con una simple función:

```
const testFunction = () => {
    console.log('En la funcion main');
}

testFunction();
```

- Supongamos ahora que queremos que se llame a otra función desde nuestro archivo principal, pero no queremos la función en él, ya que no es parte de nuestra característica principal
- Para esto, vamos a crear un archivo `mod1.js` y vamos a agregarle el siguiente código:

```
const mod1Function = () => console.log('Mod1 esta disponible!');
module.exports = mod1Function;
```

Módulos CommmonJS cont.

- `module.exports` es la palabra clave que usamos para declarar todo lo que queremos exportar desde ese archivo
- Para usar esta función en nuestro archivo `main.js`, podemos hacerlo así:

```
let mod1Function = require('./mod1.js');

const testFunction = () => {
    console.log('Estoy en la funcion main');
    mod1Function();
}

testFunction();
```

- Declaramos lo que queramos usar y luego lo asignamos al `require` del archivo que queremos usar

Módulos CommmonJS cont.

- Si quisiéramos exportar más de una funcionalidad o elemento de un solo módulo, podemos hacerlo así:

```
const mod1Function = () => console.log('Mod1 is alive!');
const mod1Function2 = () => console.log('Mod1 is rolling, baby!');

module.exports = { mod1Function, mod1Function2 };
```

- En el archivo `main.js` podemos usar ambas funciones así:

```
({ mod1Function, mod1Function2 } = require('./mod1.js'));

const testFunction = () => {
    console.log('Estoy en la funcion main');
    mod1Function();
    mod1Function2();
}

testFunction();
```

Módulos ES

- `ESmodules` es un estándar que se introdujo con ES6 (2015)
- La idea fue estandarizar el funcionamiento de los módulos JS e implementar estas funciones en los navegadores (que anteriormente no admitían módulos)
- `ESmodules` es un enfoque más moderno que actualmente es compatible con el navegador y las aplicaciones del lado del servidor con Node

Módulos ES cont.

- Vamos a nuestro archivo `package.json` y agregamos `"type": "module"`, así:

```
{
  "name": "modulestestapp",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "type": "module"
}
```

- Si no hacemos esto e intentamos implementar ESmodules en Node, obtendremos un error como el siguiente:

```
(node:29568) Warning: To load an ES module, set "type": "module" in the package.json or use the
...
SyntaxError: Cannot use import statement outside a module
```

Módulos ES cont.

- Ahora repitamos el ejemplo anterior
- En nuestro archivo `main.js` tendremos el siguiente código:

```
// main.js
import { mod1Function } from './mod1.js';

const testFunction = () => {
    console.log('En la funcion main');
    mod1Function();
}

testFunction();
```

- Y en `mod1.js` tendremos:

```
// mod1.js
const mod1Function = () => console.log('Mod1 esta disponible!');
export { mod1Function };
```

Módulos ES cont.

- Observemos que en lugar de `require` estamos usando `import` y en lugar de `module.exports` estamos usando `export`
- La sintaxis es diferente pero el comportamiento es muy similar
- Nuevamente, si quisiéramos exportar más de una funcionalidad del mismo archivo, podríamos hacerlo así:

```
// main.js
import { mod1Function, mod1Function2 } from './mod1.js';

const testFunction = () => {
    console.log('Im the main function');
    mod1Function();
    mod1Function2();
}

testFunction();
// mod1.js
const mod1Function = () => console.log('Mod1 esta disponible!');
const mod1Function2 = () => console.log('Otra funcion de Mod1!');

export { mod1Function, mod1Function2 };
```

Módulos ES cont.

- Otra característica disponible en `ESmodules` es el cambio de nombre de importación, que se puede hacer así:

```
// main.js
import { mod1Function as funct1, mod1Function2 as funct2 } from './mod1.js';

const testFunction = () => {
    console.log('En la funcion main');
    funct1();
    funct2();
}

testFunction();
```

- Observamos que la palabra clave `as` aparece luego de cada función, y luego cambiamos el nombre que queramos
- Más adelante en nuestro código, podemos usar ese nuevo nombre en lugar del nombre original que tiene la importación

Módulos ES cont.

- También podríamos importar todas las exportaciones juntas y agruparlas en un objeto, de la siguiente forma:

```
// main.js
import * as mod1 from './mod1.js';

const testFunction = () => {
    console.log('En la funcion main');
    mod1.mod1Function();
    mod1.mod1Function2();
}

testFunction();
```

- Esto puede ser útil en los casos en que, a lo largo de nuestro código, queremos ser explícitos acerca de dónde proviene cada importación
- Las funciones ahora se llaman como `mod1.mod1Function()`

Módulos ES cont.

- Con la palabra clave `default` podemos establecer una exportación predeterminada para un módulo determinado. Como esto:

```
// mod1.js
const mod1Function = () => console.log('Mod1 esta disponible!');
const mod1Function2 = () => console.log('Otra funcion de Mod1!');

export default mod1Function;
export { mod1Function2 };
```

- ¿Y qué significa tener una exportación predeterminada? Bueno, significa que no tenemos que desestructurarlo cuando lo importamos. Podemos usarlo así:

```
// main.js
import mod1Function, { mod1Function2 } from './mod1.js';

const testFunction = () => {
    console.log('En la funcion main');
    mod1Function();
    mod1Function2();
}

testFunction();
```

Módulos ES cont.

- Incluso podemos cambiar el nombre de la importación sin usar la palabra clave as, ya que JavaScript *"sabe"* que si no estamos desestructurando nos referiremos a la importación predeterminada

```
// main.js
import lalala, { mod1Function2 } from './mod1.js';

const testFunction = () => {
    console.log('En la funcion main');
    lalala();
    mod1Function2();
}

testFunction();
```

Programación Orientada a Objetos

Luciano Diamand

Introducción

Conceptos basicos de POO

- La programación orientada a objetos (POO) es un paradigma de programación que se basa en el concepto de clases y objetos
- Se utiliza para estructurar un programa de software en piezas fundamentales llamadas clases, que se utilizan para crear instancias individuales
- La programación orientada a objetos es adecuada para programas de gran tamaño, complejos y que se actualizan o mantienen activamente
- `Object` representa uno de los tipos de datos de JavaScript
- Se utiliza para almacenar colecciones con clave/valor o entidades más complejas
- Los objetos se pueden crear de dos formas:
  - Utilizando el constructor `Object()`:

```
var myObj = new Object();
myObj.key = value;
```

  - Utilizando la sintaxis declarativa (literal):

```
var myObj = {
  key: value
  // ...
};
```

Beneficios de la Programación Orientada a Objetos

- Algunos de los beneficios de la Programación Orientada a Objetos son los siguientes:
  - Desarrollo en terminos del dominio del problema
  - Bajo acoplamiento a través del encapsulamiento
  - Alta cohesión
  - Reutilización de código a través de la herencia
  - Flexibilidad a través del polimorfismo

Principios de la Programación Orientada a Objetos

- **Abstracción**: Proceso de clasificación de las propiedades que intervienen en la solución
- **Encapsulamiento**: Permite dividir un programa en componentes más pequeños e independientes. Cada componente es autónomo e independiente de los demás componentes
- **Herencia**: Reutilizar código entre clases
- **Polimorfismo**: Los objetos pueden tomar diferentes comportamientos dependiendo del contexto

Clases

- Una clase es una plantilla para la creación de objetos según un modelo predefinido
- Las clases se utilizan para representar entidades o conceptos
- Están compuestas por miembros:
  - Propiedades
  - Métodos

Objetos

- Instancia de una clase
- Entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad (métodos), los mismos que consecuentemente reaccionan a eventos
- Se corresponden con los objetos reales del mundo que nos rodea, o con objetos internos del sistema (del programa)

Abstracción

- La abstracción es el procedimiento mediante el cual el programador captura las características y comportamientos esenciales de un elemento
- Esta abstracción nos permite crear una pieza de software que emula al objeto del mundo real

Encapsulamiento

- Se denomina encapsulamiento a la reunión de atributos pertenecientes a una misma clase, al mismo nivel de abstracción, suele confundirse con el principio de ocultación, básicamente porque este último depende del encapsulamiento
- El principio de ocultación consiste en garantizar que los elementos privados de una clase no puedan ser accedidos por otra. Es decir, proteger información específica de cualquier otro tipo de objeto diferente a la instancia actual, incluso si son de la misma clase.
- Las tres características de un encapsulamiento eficaz son:
  - Abstracción
  - Ocultamiento de la implementación
  - División de la responsabilidad

Herencia

- La herencia es un mecanismo de los lenguajes de programación orientados a objetos, por medio del cual una clase deriva de otra (se define en base a otra) de manera que extiende su funcionalidad
- La definición de la nueva clase obtiene automáticamente todos los atributos, comportaminetos e implementaciones que contenga la clase existente
- La clase de la que se hereda se suele denominar clase base, clase padre, superclase, clase ancestro

Polimorfismo

- Se basa en la herencia
- En terminos de programación permite que un solo nombre de clase o de método represente un código diferente seleccionado mediante algún mecanismo automático
- Un nombre puede tomar diferentes formas y puesto que puede representar código diferente, también puede representar muchos comportamientos distintos
- Pensemos en el termino *abrir*. Podriamos abrir una puerta, una caja, una ventana o una cuenta en el banco. Se puede aplicar a muchos objetos distintos en la realidad. Cada objeto lo interpretara a su manera (se abrira de distintas formas)

Clases

POO basado en prototipos

- Conceptualmente, en la programación orientada a objetos basada en clases, primero creamos una clase que sirva como un "modelo" para los objetos y luego creamos objetos basados este modelo
- Para construir tipos de objetos más específicos, creamos clases "secundarias"; es decir, hacemos algunos cambios en el plano y usamos el nuevo plano resultante para construir los objetos más específicos
- Para una analogía del mundo real, si tuviera que construir una silla, primero crearíamos un plano en papel y luego fabricaría sillas basadas en este plano
- El modelo aquí es la clase y las sillas son los objetos
- Si quisiera construir una mecedora, tomaría el plano, haría algunas modificaciones y fabricaría mecedoras utilizando el nuevo plano

POO basado en prototipos cont.

Ahora llevemos este ejemplo al mundo de los prototipos:

- No creamos los planos o clases aquí, solamente creamos el objeto
- Tomamos un poco de madera y cortamos una silla. Esta silla, un objeto real, puede funcionar completamente como silla y servir también como prototipo para futuras sillas
- En el mundo de los prototipos, construimos una silla y simplemente creamos "clones" de ella
- Si deseamos construir una mecedora, todo lo que tenemos que hacer es elegir una silla que haya fabricado anteriormente y colocarle dos mecedoras
- Ahora podemos disfrutar de esa mecedora, o tal vez usarla como prototipo para crear más mecedoras

Clases

- En JavaScript ES5 la programación orientada a objetos se realizaba utilizando objetos basados en *prototypes* en lugar de utilizar *clases*
- 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 {
  firstName: string;
  lastName: 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 {
  firstName: string;
  lastName: string;
  age: number;

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

Métodos cont.

- Dentro de un método es posible acceder al *firstName* 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 firstName
p.firstName = 'Juan';

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

Métodos cont.

- Es posible declarar e instanciar un objeto de 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 realiza cualquier tipo de inicialización de las propiedades del nuevo objeto
- Los constructores se deben llamar `constructor`
- Pueden de forma opcional recibir parametros
- No pueden devolver ningún valor (no tienen tipo de retorno)

Constructores cont.

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

```
class 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:

```
class Person {
  firstName: string;
  lastName: string;
  age: number;

  constructor(firstName: string, lastName: string, age: number) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
  }

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

  // ... otros métodos
}
```

Constructores cont.

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

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

Abstracción

Abstracción

- Los objetos solo revelan mecanismos internos que son relevantes para el uso de otros objetos, ocultando cualquier código de implementación innecesario
- Esta funcionalidad es muy poderosa en el desarrollo y TypeScript nos proporciona varias formas de manipular la visibilidad de los miembros de un objeto de clase
- Al ejemplo anterior, hemos agregado el modificador `public` en todos los atributos de `Person`
- De forma predeterminada, todos los atributos son siempre `public`

```
// Clase Person
class Person {
   public name: string = '';
   public age: string = '';

   public greetings() {
     return this.name + ' ' + this.age;
   }
}
```

Modificadores de acceso

- Podemos utilizar los siguientes modificadores de acceso para controlar la visibilidad de los atributos
  - public
  - protected
  - private
- Además, podemos usar `readonly`, lo que evitará las asignaciones al campo fuera del constructor

```
class Person {
  public readonly credentials: string = '';
  public name: string = '';
  public department: string = '';

  constructor(value: string) {
    this.credentials = value;
  }

  public setName(name: string): void {
    if (!this.credentials) return;
    this.name = name;
  }
```

Modificadores de acceso cont.

- En el ejemplo anterior, si intentamos crear una nueva instancia de `Person` sin proporcionar `credentials`, recibiremos una advertencia y no compilará
- Lo primero que debemos notar es que la visibilidad de las credenciales es `readonly`, lo que significa que solo se puede actualizar el valor dentro del constructor

```
// warining An argument for 'value' was not provided.
const person1 = new Person();

// En este caso funcionará
const person1 = new Person('123456');
```

Encapsulamiento

Encapsulamiento

- La implementación y el estado de cada objeto se mantienen de forma privada dentro de un límite definido o clase
- Otros objetos no tienen acceso a esta clase o la autoridad para realizar cambios, pero solo pueden llamar a una lista de funciones o métodos públicos
- Esta característica de ocultación de datos proporciona una mayor seguridad del programa y evita la corrupción de datos no deseada

Encapsulamineto cont.

- Por lo tanto, una buena práctica es definir a los atributos como `private` y que no se puedea acceder a ellos desde fuera de la clase
- Tenga en cuenta que solo podemos acceder a campos públicos fuera de la definición de la clase
- Si cambiamos el nivel de acceso de los campos de la clase `Person` a `private`:

```
// Creamos el objeto Preson person1
const person1 = new Person('123456');

person1. // esto enumerará automáticamente todas las propiedades accesibles
// Note que no se puede acceder a los atributos privados fuera de la definición de la clase
```

Encapsulamineto cont.

- En el siguente ejemplo, hemos definido dos clases de objetos y hemos creado una instancia de cada uno
- El principio de encapsulación establece que la nueva instancia de `motor1` no puede acceder a los atributos de `person1`
- Si intentaramos acceder, obtendriamos una advertencia de este tipo

```
La propiedad 'edad' no existe en el tipo 'Motor'
```

Encapsulamiento cont.

```
// Clase persona
class Person {
   name: string = '';
   age: number = 0;
}

// Clase motor
class Motor {
  make: string = '';
  model: string = '';
  color: string = '';
}

// Creamos una instancia de cada una
const person1 = new Person();
const motor1 = new Motor();

// Advertencia: Property 'age' does not exist on type 'Motor'.
motor1.age();
```

Propiedades getters y setters

- Hay dos tipos de propiedades de objeto
  - Propiedades de los datos: Ya sabemos cómo trabajar con ellas. Todas las propiedades que hemos estado usando hasta ahora eran propiedades de datos
  - Propiedad accesor o mutador: Son esencialmente funciones que se ejecutan al obtener o establecer un valor, pero parecen propiedades normales para un código externo

Propiedades getters y setters cont.

- Las propiedades de acceso están representadas por los métodos `getter` y `setter`
- En un objeto literal, se denotan por `get` y `set`:

```
let obj = {
  get propName() {
    // getter, el código ejecutado al obtener obj.propName
  },

  set propName(value) {
    // setter, el código ejecutado al configurar obj.propName = value
  }
};
```

Propiedades getters y setters cont.

- Supongamos tener un objeto que represente a una persona con su nombre y apellido
- Ahora queremos agregar una propiedad `fullName` que devuelva la concatenación de nombre y apellido
- Podríamos implementarla con un `get`/`set`:

```
let user = {
  name: "John",
  surname: "Smith",

  get fullName() {
    return `${this.name} ${this.surname}`;
  }
};

console.log(user.fullName); // John Smith
```

Herencia

Herencia

- La herencia es una forma de indicar que una clase "*recibe*" o hereda el comportamiento de otra clase llamada *clase base* o *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`
- Se pueden asignar relaciones de subclases entre objetos, lo que permite a los desarrolladores reutilizar una lógica común manteniendo una jerarquía única

Herencia cont.

- Creemos una nueva clase encargada de generar reportes:

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

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

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

- Ahora podemos probar la clase `Report` de la siguiente manera:

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

Herencia cont.

- Si queremos aplicar herencia sobre la clase `Report` hacemos un `extends` sobre la misma:

```
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 `TabbedReport` 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();
```

Polimorfismo

Polimorfismo

- Los objetos pueden tomar más de una forma dependiendo del contexto
- El programa determinará qué significado o uso es necesario para cada ejecución de ese objeto, reduciendo la necesidad de duplicar código
- El polimorfismo se basa en la sustitución

Ejemplo

Analicemos el siguente ejemplo y veamos como podemos mejorarlo:

```
class Person {
  public name: string = '';
  public role: string = '';
}

class BasketballPlayer extends Person {
  public setName(name: string) {
    this.name = name
    this.role = 'BasketBall Player';
  }
  public getName() {
    return `User name: ${this.name} Role: ${this.role}`;
  }
}
```

Polimorfismo cont.

```
class GolfPlayer extends Person {
  public setName(name: string) {
    this.name = name;
    this.role = 'Golf Player';
  }
  public getName() {
    return `User name: ${this.name} Role: ${this.role}`;
  }
}

const person1 = new BasketballPlayer();
const person2 = new GolfPlayer();
person1.setName('Kevin Odongo');
person2.setName('Kevin Odongo');
console.log(person1.getName());
console.log(person2.getName());
```

Interfaces

Tipos de objetos

- En JavaScript, la forma fundamental en la que agrupamos y transmitimos datos es a través de objetos
- En TypeScript, los representamos a través de tipos de objetos
- Como hemos visto, pueden ser anónimos:

```
function greet(person: { name: string; age: number }) {
  return 'Hello ' + person.name;
}
```

Tipos de objetos cont.

- También pueden ser nombrados usando una interfaz

```
interface Person {
  name: string;
  age: number;
}

function greet(person: Person) {
  return 'Hello ' + person.name;
}
```

- o un alias de tipo

```
type Person = {
  name: string;
  age: number;
};

function greet(person: Person) {
  return "Hello " + person.name;
}
```

Tipos de objetos cont.

- En los tres ejemplos anteriores, hemos escrito funciones que toman objetos que contienen la propiedad `name` (que debe ser una cadena) y la propiedad `age` (que debe ser un número).

Extendiendo tipos

- Es bastante común tener tipos que pueden ser versiones más específicas de otros tipos
- Por ejemplo, podríamos tener un tipo `BasicAddress` que describa los campos necesarios para enviar cartas y paquetes

```
interface BasicAddress {
  name?: string;
  street: string;
  city: string;
  country: string;
  postalCode: string;
}
```

Extendiendo tipos

- En algunas situaciones, eso es suficiente, pero las direcciones a menudo tienen un número de departamento asociado si el edificio tiene varios departamentos
- Entonces podemos describir una `AddressWithUnit`

```
interface AddressWithUnit {
  name?: string;
  unit: string;
  street: string;
  city: string;
  country: string;
  postalCode: string;
}
```

Extendiendo tipos cont.

- Esto funciona, pero la desventaja es que tuvimos que repetir todos los demás campos de `BasicAddress` cuando nuestros cambios eran puramente aditivos
- En su lugar, podemos extender el tipo `BasicAddress` original y simplemente agregar los nuevos campos que son exclusivos de `AddressWithUnit`

```
interface BasicAddress {
  name?: string;
  street: string;
  city: string;
  country: string;
  postalCode: string;
}

interface AddressWithUnit extends BasicAddress {
  unit: string;
}
```

Extendiendo tipos cont.

- La palabra clave `extends` en una interfaz nos permite copiar efectivamente miembros de otros tipos con nombre y agregar los miembros nuevos que queramos
- Esto puede ser útil para reducir la cantidad de declaraciones repetitivas de tipo que tenemos que escribir y para señalar la intención de que varias declaraciones diferentes de la misma propiedad puedan estar relacionadas
- Por ejemplo, `AddressWithUnit` no necesitaba repetir la propiedad `street` y, dado que `street` se origina en `BasicAddress`, el lector sabrá que esos dos tipos están relacionados de alguna manera
- Las interfaces también pueden extenderse desde múltiples tipos

```
interface Colorful {
  color: string;
}

interface Circle {
  radius: number;
}

interface ColorfulCircle extends Colorful, Circle {}
```

Intersección de tipos

- Las interfaces nos permitieron construir nuevos tipos a partir de otros tipos al extenderlos
- TypeScript proporciona otra construcción llamada intersección de tipos que se usa principalmente para combinar tipos de objetos existentes
- Una intersección de tipo se define mediante el operador &

```
interface Colorful {
  color: string;
}
interface Circle {
  radius: number;
}

type ColorfulCircle = Colorful & Circle;
```

Intersección de tipos cont.

- Aquí, hemos cruzado `Colorful` y `Circle` para producir un nuevo tipo que tiene todos los miembros de `Colorful` y `Circle`

```
function draw(circle: Colorful & Circle) {
  console.log(`Color was ${circle.color}`);
  console.log(`Radius was ${circle.radius}`);
}

// okay
draw({ color: "blue", radius: 42 });

// oops
draw({ color: "red", raidus: 42 });
```

Interfaces vs intersecciones

- Acabamos de ver dos formas de combinar tipos que son similares, pero que en realidad son sutilmente diferentes
- Con las interfaces, podríamos usar una cláusula `extends` para extendernos desde otros tipos, y pudimos hacer algo similar con las intersecciones y nombrar el resultado con un alias de tipo
- La diferencia principal entre los dos es cómo se manejan los conflictos, y esa diferencia suele ser una de las razones principales por las que elegiría uno sobre el otro entre una interfaz y un alias de tipo de un tipo de intersección.

Tipos de objetos genéricos

- Imaginemos un tipo `Box` que puede contener cualquier valor: cadenas, números, jirafas, lo que sea

```
interface Box {
  contents: any;
}
```

- En este momento, la propiedad de contenido se escribe como `any`, lo que funciona, pero puede provocar problemas en el futuro

Tipos de objetos genéricos cont.

- En su lugar, podríamos usar `unknown`, pero eso significaría que en los casos en los que ya conocemos el tipo de contenido, necesitaríamos hacer verificaciones preventivas o usar aserciones de tipo propensas a errores

```
interface Box {
  contents: unknown;
}

let x: Box = {
  contents: "hello world",
};

// we could check 'x.contents'
if (typeof x.contents === "string") {
  console.log(x.contents.toLowerCase());
}

// or we could use a type assertion
console.log((x.contents as string).toLowerCase());
```

Tipos de objetos genéricos cont.

- Un enfoque de tipo seguro sería, en cambio, crear diferentes tipos de `Box` para cada tipo de contenido

```
interface NumberBox {
  contents: number;
}

interface StringBox {
  contents: string;
}

interface BooleanBox {
  contents: boolean;
}
```

- Pero eso significa que tendremos que crear diferentes funciones, o sobrecargas de funciones, para operar en estos tipos

```
function setContents(box: StringBox, newContents: string): void;
function setContents(box: NumberBox, newContents: number): void;
function setContents(box: BooleanBox, newContents: boolean): void;
function setContents(box: { contents: any }, newContents: any) {
  box.contents = newContents;
}
```

Tipos de objetos genéricos cont.

- Esto es mucho codigo repetitivo. Además, más adelante podríamos necesitar introducir nuevos tipos y sobrecargas. Esto es frustrante, ya que nuestros tipos de `Box` y sobrecargas son todas iguales
- En su lugar, podemos crear un tipo de `Box` genérico que declare un parámetro `Type`

```
interface Box<Type> {
  contents: Type;
}
```

- Podemos leer esto como "*Un Box de Type es algo cuyo contenido tiene tipo Type*"
- Más adelante, cuando nos referimos a `Box`, tenemos que dar un argumento de tipo en lugar de `Type`

```
let box: Box<string>;
```

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
  ...
  2019 Angular 8
  ...
  2023 Angular 16

- Utilizan versionado semantico

  `14.1.1`
  `Major.Minor.Revision`

Entorno de desarrollo

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

Angular CLI

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 poder 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`
- El comando anterior va a instalar `ng` en el 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 espacio de trabajo y un proyecto inicial en Angular
- El espacio de trabajo 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-install` 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: 13.2.4
Node: 16.10.0
Package Manager: npm 7.24.0
OS: linux x64

Angular:
...

Package                      Version
------------------------------------------------------
@angular-devkit/architect    0.1302.4 (cli-only)
@angular-devkit/core         13.2.4 (cli-only)
@angular-devkit/schematics   13.2.4 (cli-only)
@schematics/angular          13.2.4 (cli-only)
...
```

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 (un nuevo constructor experimental se utiliza en lugar de Webpack basado en `esbuild`)
- `ng serve [options]`
- Opciones:
  - `--host` podemos asignarle el host
  - `--port` podemos cambiar el puerto por defecto
  - `-o` abre un navegador al iniciar el servidor

Arquitectura de Angular

NgModule

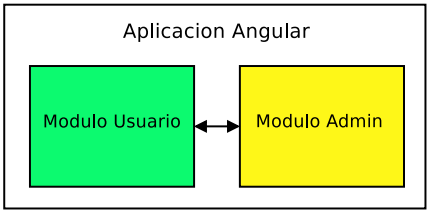
- Los bloques de construcción básicos de una aplicación Angular son los **NgModules**, que proporcionan un contexto de compilación para los componentes
- **NgModules** recopila códigos relacionados en conjuntos funcionales
- Una aplicación Angular se define por un conjunto de **NgModules**
- Una aplicación 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

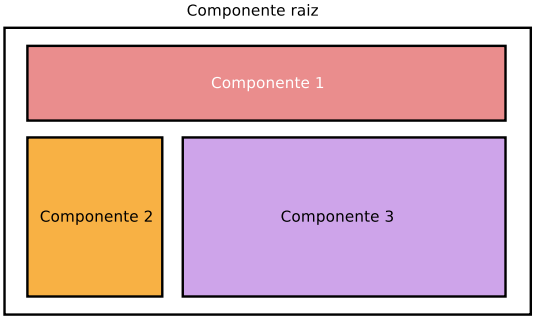
Módulos

- 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**



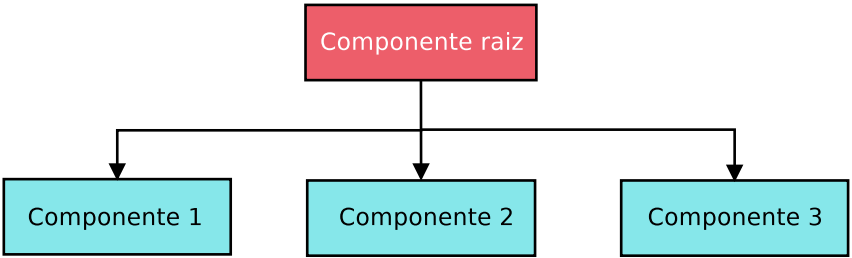
Componentes

- 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 van 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 finalizada la ejecución del comando, tendremos dentro de la carpeta `src/app` un directorio con el nombre `nombre-del-componente`

Agregar un nuevo componente cont.

- 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

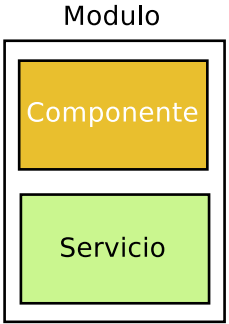
- Antes de que se muestre una vista, Angular evalúa las directivas y resuelve la sintaxis de enlace en la plantilla para modificar los elementos HTML y el DOM, de acuerdo con la lógica y los datos del programa
- Angular admite enlace de datos bidireccional, lo que significa que los cambios en el DOM, 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

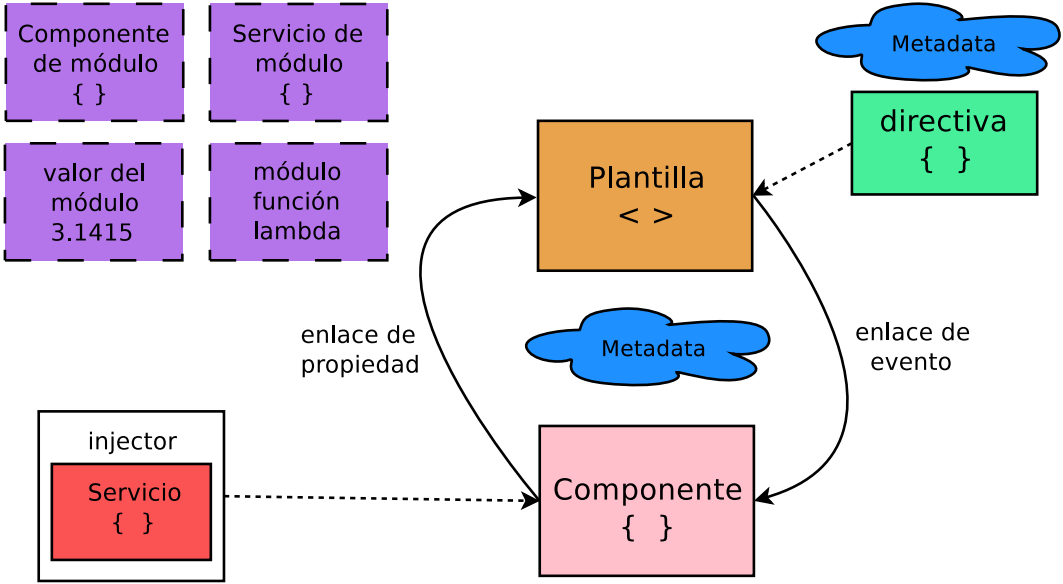
- El módulo de enrutamiento `NgModule` de Angular proporciona un servicio que permite definir una ruta de navegación entre los diferentes estados de la aplicación
- Se basa en las convenciones características de navegación del navegador:
  - Ingrese una URL en la barra de direcciones y el navegador navega 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 { }

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-hero-detail'
```

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
- Del lado del elemento debe tomar una combinación de establecer una propiedad de 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

¿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 componente
  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

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

Visión general

- El navegador utiliza un modelo familiar a la navegación de aplicaciones:
  - Ingresamos una URL en la barra de direcciones y el navegador nos llevara a la página correspondiente
  - Hacemos clic en los enlaces en la página y el navegador nos llevara a una nueva página
  - Hacemos 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 hemos visitado
- El enrutador de Angular toma como base este modelo
- Podemos interpretar una URL del navegador como una instrucción para ir a una vista generada por el cliente
- Podemos pasar parámetros opcionales junto con el componente de vista que ayude a decidir qué contenido específico presentar

Visión general cont.

- Podemos vincular el enrutador a los enlaces en una página y navegar a la vista adecuada cuando el usuario haga clic en un enlace
- Podemos 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 origen
- El enrutador registra la actividad en el historial del navegador para que los botones de avance y retroceso también funcionen

base href

- La mayoría de las aplicaciones que utilizan enrutamiento deben agregar el elemento `<base>` al `index.html` como primer elemento dentro de la etiqueta `<head>` para decirle al enrutador cómo armar las URL de navegación
- Si el directorio de la aplicación es también la raíz de la aplicación, como lo es para las aplicaciones de ejemplo, establecemos el valor `href` en

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

Importar el enrutador

- El enrutador de Angular es un servicio opcional que presenta la vista de un componente particular para una URL determinada
- No es parte del núcleo Angular, está en su propio paquete de biblioteca, `@angular/router`
- Por lo tanto podemos importar lo que necesitemos como lo haríamos con otro paquete de 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, el enrutador busca una ruta correspondiente desde la cual puede determinar el componente que se mostrará
- Un enrutador no tiene rutas definidas por defecto
- El método estático `forRoot()` crea un módulo que contiene todas las directivas, las rutas y el servicio de enrutamiento
- El siguiente ejemplo crea cinco definiciones de ruta, configura el enrutador a través del método `RouterModule.forRoot` y agrega el resultado al arreglo de importaciones del módulo `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 } // <-- solo con propósitos de depuración
    ) // otros imports
  ],
  ...
})
export class AppModule { }
```

Configuración cont.

- El conjunto de rutas `appRoutes` describe cómo será la navegación y es el argumento al método `RouterModule.forRoot` dentro de la importación del módulo
- Cada ruta asigna una 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 al navegar 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*
- `pathMatch` se utiliza para especificar la estrategia de comparación que puede ser `full` o `prefix`

Configuración cont.

- La propiedad `data` de la tercer ruta es un lugar donde almacenar datos arbitrarios asociados con esta ruta específica
- La propiedad `data` es accesible dentro de cada ruta activa
- 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 de la cuarta ruta representa la ruta predeterminada para la aplicación. Es el lugar al que se dirije cuando la ruta de 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 componente `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 selección de 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 vacía, que coincide con la ruta predeterminada
- La ruta comodín viene en último lugar porque coincide con todas las URL y debe seleccionarse solo si ninguna otra ruta coincide

Configuración cont.

- Si necesitamos ver qué eventos están sucediendo durante el ciclo de vida de navegación, tenemos la opción `enableTracing` como parte de la configuración por defecto del enrutador
- Esto registra en consola cada evento que tuvo lugar en el enrutador durante cada ciclo de vida de la navegación y debería utilizarse solo para la depuración
- Establecemos la opción `enableTracing: true` en el objeto que se pasa como segundo argumento al método `RouterModule.forRoot()` (ver ejemplo más arriba)

Salida del Enrutador

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

```
<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 mostrará `HeroListComponent` como elemento relacionado con el `RouterOutlet`, colocado en la plantilla del componente anfitrión

Enlaces al enrutador

- Ahora ya tenemos las rutas configuradas y un lugar para renderizarlas, ¿cómo las navegamos?
- La URL podría escribirse directamente desde la barra de direcciones del navegador, pero la mayoría de las veces navegamos como resultado de alguna acción del usuario, como el clic de una etiqueta `<a>`
- 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` modifica las clases CSS para los enlaces `RouterLink` activos en función del `RouterState` actual
- En cada etiqueta de de enlace, agregamos un enlace propiedad a la directiva `RouterLinkActive` con 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 cont.

- 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 vincular 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 del enrutador que puede ser inyectado llamado `ActivatedRoute`
- Tiene una gran cantidad de información útil que incluye:

Propiedad	Descripción
`url`	Un observable de la ruta, representado como un arreglo 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 activas bajo la ruta actual

Ruta activada cont.

- Dos propiedades más antiguas todavía están disponibles. Tienen menos características que sus reemplazos, se desaconseja su uso y pueden ser removidas 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 pasando por muchos estados intermedios
- La lista completa de eventos de navegación se muestra en la siguiente tabla

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

Eventos enrutador cont.

`GuardsCheckEnd`	Un evento disparado cuando el enrutador finaliza la fase de guarda de enrutamiento con éxito
`ResolveStart`	Un evento disparado cuando el enrutador comienza la fase de resolución del enrutamiento
`ResolveEnd`	Un evento disparado cuando el enrutador finaliza la fase de resolución de enrutamiento correctamente
`ChildActivationEnd`	Un evento disparado cuando el enrutador termina de activar los hijos de una ruta
`ActivationEnd`	Un evento disparado cuando el enrutador termina de activar una ruta
`NavigationEnd`	Un evento disparado cuando la navegación termina con éxito
`NavigationCancel`	Un evento disparado cuando se cancela la navegación. Esto se debe a que una Guardia de ruta devuelve falso durante la navegación
`NavigationError`	Un evento disparado cuando falla la navegación debido a un error inesperado
`Scroll`	Un evento disparado que representa un evento de desplazamiento

Guarda de rutas

- Agregamos una guarda a la configuración de la ruta para manejar los siguientes escenarios
  - Si devuelve `true`, el proceso de navegación continúa
  - Si devuelve `false`, el proceso de navegación se detiene y el usuario 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 guarda en cada nivel de una jerarquía de enrutamiento
- El enrutador comprueba primero las guarda `CanDeactivate` y `CanActivateChild`
- Luego verifica los guarda `CanActivate`
- Si el módulo con características particulares de la aplicación se carga de forma asíncrona, la protección de `CanLoad` se comprueba antes de que se cargue el módulo
- Si alguna guarda devuelve `false`, las guardas pendientes que no se hayan completado se cancelarán y se cancelará toda la navegación

Resolve

- Interfaz que las clases pueden implementar para ser un proveedor de datos
- Se puede usar una clase de proveedor de datos con el enrutador para resolver datos durante la navegación
- La interfaz define un método `resolve()` que se invoca justo después del evento del enrutador `ResolveStart`
- El enrutador espera a que se resuelvan los datos antes de que finalmente se active la ruta

```
interface Resolve<T> {
  resolve(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<T> | Promise<T> | T
}
```

Resolve cont.

- El siguiente ejemplo implementa un método `resolve()` que recupera los datos necesarios para activar la ruta solicitada

```
@Injectable({ providedIn: 'root' })
export class HeroResolver implements Resolve<Hero> {
  constructor(private service: HeroService) { }

  resolve(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<Hero> | Promise<Hero> | Hero {
    return this.service.getHero(route.paramMap.get('id'));
  }
}
```

Resolve cont.

- Luego, el `Resolver` se proporciona como parte del objeto `Route` en la configuración del enrutador:

```
@NgModule({
  imports: [
    RouterModule.forRoot([
      {
        path: 'detail/:id',
        component: HeroDetailComponent,
        resolve: {
          hero: HeroResolver
        }
      }
    ])
  ],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

Resolve cont.

- Podemos acceder a los datos recuperados desde `HeroComponent`:

```
@Component({
 selector: "app-hero",
 templateUrl: "hero.component.html",
})
export class HeroComponent {

 constructor(private activatedRoute: ActivatedRoute) { }

 ngOnInit() {
   this.activatedRoute.data.subscribe(({ hero }) => {
     // hacemos algo con los datos obtenidos ...
   });
 }
}
```

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

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 de forma única en la plantilla, utilice formularios controlados por plantillas

Diferencias claves

- La siguiente tabla resume las diferencias clave entre los formularios reactivos y los basados en plantillas

	Reactivos	Basados en plantilla
Configuración	Explícito, creado en la clase del componente	Implícito, creado por directivas
Modelo de datos	Estructurado e inmutable	No estructurado y mutable
Flujo de datos	Sincrónico	Asincrónico
Validación de formulario	Funciones	Directivas

Clases bases comunes a ambos formularios

- 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

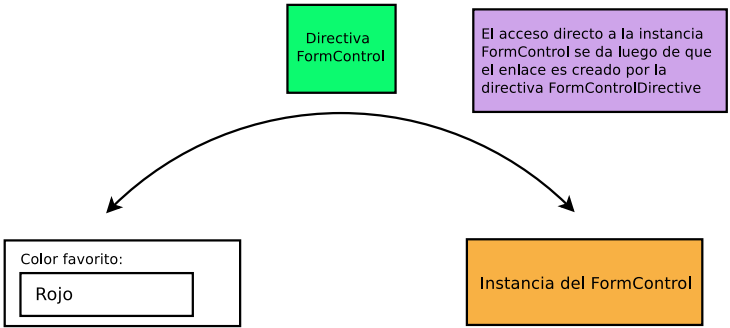
- Aquí hay un componente con un campo de entrada `input` para un solo control implementado usando formularios reactivos

```
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.

- En los formularios reactivos, el modelo del formulario es la *fuente de la verdad*, proporciona el valor y el estado del elemento del formulario en cualquier momento del tiempo, a través de la directiva `FormControl` en el elemento `input`
- Con los formularios reactivos, el modelo de formulario se define explícitamente en la clase del 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 (instancia `ControlValueAccessor`)

Configuración en formas reactivas cont.



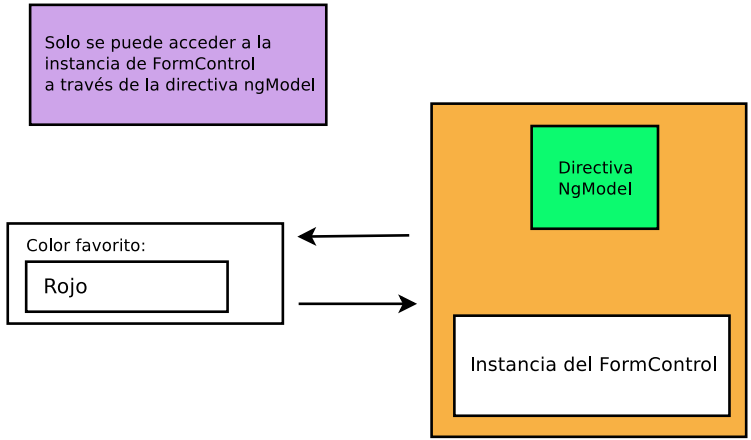
Configuración de formularios controlados por plantillas

- Este es el mismo componente con un campo de entrada `input` 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 los formularios basadas en plantillas, la *fuente de la verdad* es la propia plantilla
- No tenemos acceso directo de forma codificada a la instancia `FormControl`



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 `FormControl` para un elemento de formulario determinado
- Es menos explícito, pero no tiene control directo sobre el modelo de formulario

Flujo de datos en formularios

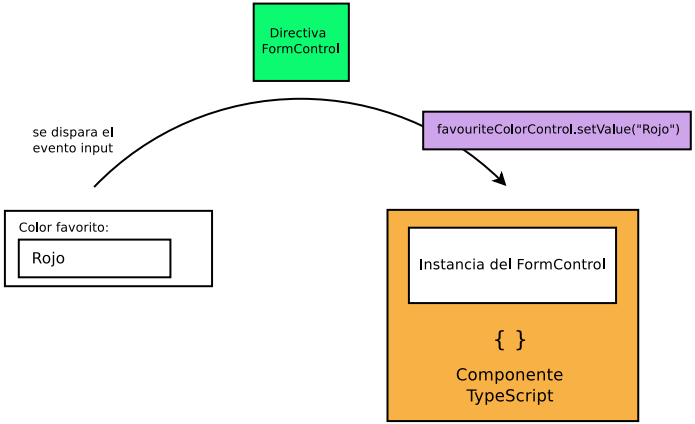
- Cuando una aplicación contiene un formulario, Angular debe mantener la vista sincronizada con el modelo del componente y el modelo del componente sincronizado con la vista
- A medida que los usuarios cambian valores y realizan selecciones a través de la vista, los nuevos valores deben reflejarse en el modelo de datos
- De manera similar, cuando la lógica del programa cambia los valores en el modelo de datos, esos valores deben reflejarse en la vista
- Los formularios reactivos y los controlados por plantillas siguen dos estrategias diferentes al manejar la entrada de formularios

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 como es renderizada la interfaz de usuario
- 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:

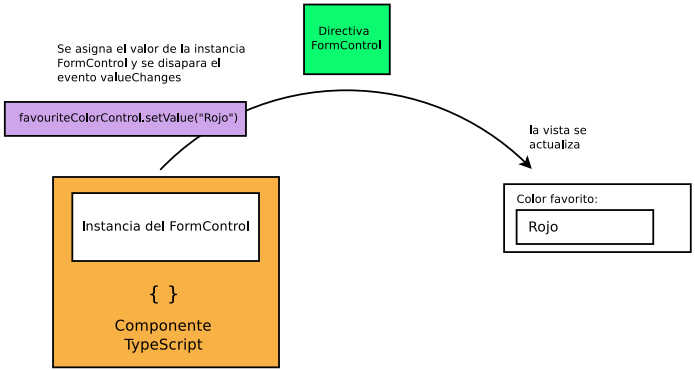


Flujo de datos en formularios reactivos cont.

- 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 del formulario emite un evento "`input`" con el último valor
  3. El `ControlValueAccessor` escucha eventos 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.

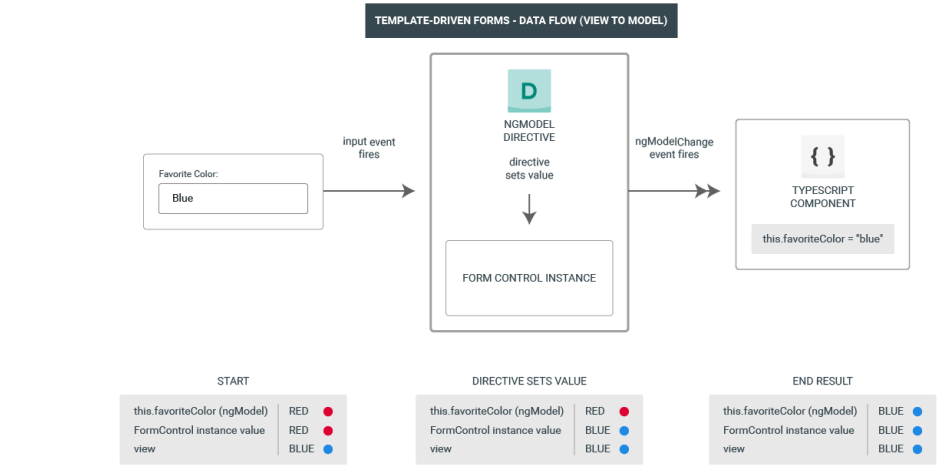
- Los pasos a continuación describen el flujo de datos del modelo a la vista en un cambio que se realiza de forma codificada
  1. El usuario llama al método `favoriteColorControl.setValue()`, que actualiza el valor de `FormControl`
  2. La instancia de `FormControl` emite el nuevo valor a través del observable `valueChanges`
  3. Cualquier suscriptor a los observables de `valueChanges` recibe el nuevo valor
  4. El `ControlValueAccessor` en el elemento de entrada `input` del 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:

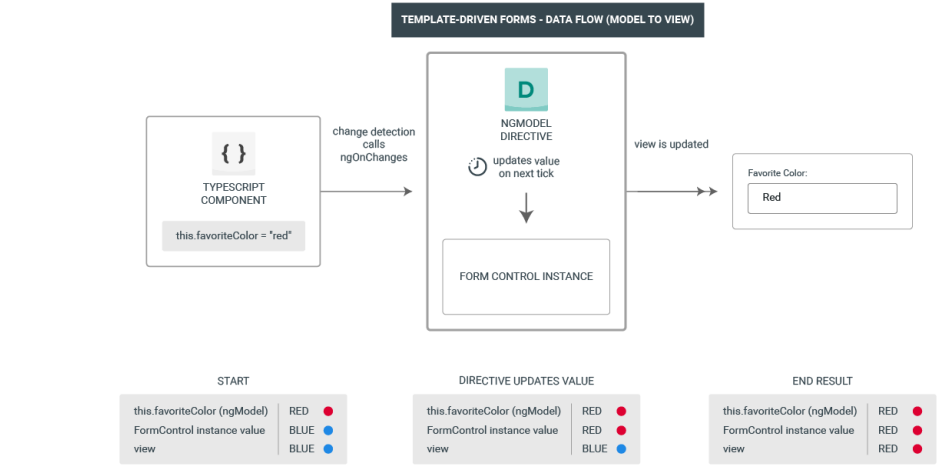


Flujo de datos en formularios controlados por plantillas

- 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 "`input`" con el valor *Azul*
  3. El `ControlValueAccessor` adjunto a la entrada actíva 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` del 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 del ciclo de vida `ngOnChanges` en la instancia de la directiva `NgModel` dado que 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 tick, 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 `ControlValueAccessor` actualiza el elemento de entrada de formulario en la vista con el último valor de `favoriteColor`

Mutabilidad

- 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
- 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 en determinar cuándo se requieren actualizaciones

Mutabilidad cont.

- La diferencia se demuestra en los ejemplos anteriores utilizando el elemento `input` del `favoriteColor`
  - 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 `favoriteColor` siempre se modifica a su nuevo valor

Validación de los formularios

- La validación es una parte integral de la gestión de cualquier conjunto de formularios
- Ya sea que esté verificando los campos obligatorios o consultando una API externa para un nombre de usuario existente, Angular proporciona un conjunto de validadores integrados, así como la capacidad de crear validadores personalizados
- Formularios reactivos: Definir validadores personalizados como funciones que reciben un control para validar
- Formularios basados en plantillas: Vinculado a directivas de plantilla, y debe proporcionar directivas de validación personalizadas que envuelvan funciones de validación

Formularios reactivos

- Registrando el modulo

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

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

Formularios reactivos cont.

- Hay tres pasos a seguir para usar controles de formulario
  - Registre el módulo de formularios reactivos en la aplicación `ReactiveFormsModule`. Este módulo declara las directivas de formulario reactivo que necesita para usar formularios reactivos
  - Genere una nueva instancia de `FormControl` y guárdela en el componente
  - Registre el `FormControl` en la plantilla

Formularios reactivos cont.

```
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">
  Enviar
</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(): void {
  this.profileForm.patchValue({
    firstName: 'Nancy',
    address: {
      street: '123 Drew Street'
    }
  });
}

...

<p>
  <button (click)="updateProfile()">
    Actualizar perfil
  </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">
      Enviar
    </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 de estilos 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">
  Nombre requerido
</div>

...
```

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">
   Enviar
</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">
     El nombre es requerido
  </div>
  <div *ngIf="name.errors.minlength">
     El nombre debe tener al menos 4 caracteres de largo
  </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

Observables

- Proporcionan el soporte para pasar mensajes entre editores y suscriptores dentro de la aplicación
- Ofrecen beneficios significativos sobre otras técnicas para el manejo de eventos, la programación asíncrona y el manejo de múltiples valores
- 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 el desacople son gestionadas por lo observable, el código de la 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 se llama para dejar de recibir notificaciones

Uso básico y términos cont.

```
// Cree un Observable que comenzará a escuchar actualizaciones de geolocalización
// cuando un consumidor se suscriba
const locations = new Observable((observer) => {
  // Obtenga las callback next y error. Estos se transmitirán cuando el consumidor
  // se suscriba
  const {next, error} = observer;
  let watchId;

  // Verificamos el soporte para la API de geolocalización la cual nos proporciona
  // valores para publicar
  if ('geolocation' in navigator) {
    watchId = navigator.geolocation.watchPosition(next, error);
  } else {
    error('Geolocation not available');
  }

  // Cuando el consumidor se dé de baja, limpie los datos para prepararlos para la
  // próxima suscripción.
  return {unsubscribe() { navigator.geolocation.clearWatch(watchId); }};
});

// Llame a subscribe() para comenzar a escuchar actualizaciones
const locationsSubscription = locations.subscribe({
  next(position) { console.log('Posicion actual: ', position); },
  error(msg) { console.log('Error obteniendo la posicion: ', msg); }
});

// Deja de escuchar la ubicación después de 10 segundos
setTimeout(() => { locationsSubscription.unsubscribe(); }, 10000);
```

Definiendo observadores

- Un controlador, para recibir notificaciones observables, implementa la interfaz `Observer`
- Es un objeto que define 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
- Nos subscribimos 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 mas frecuentemente utilizados:
  - `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 un arreglo en un observable

Suscribiendo cont.

```
// Crea un observable simple que emita tres valores
const myObservable = of(1, 2, 3);

// Crea un objeto observador
const myObserver = {
  next: x => console.log('El Observer obtuvo el proximo valor: ' + x),
  error: err => console.error('El Observer obtuvo un error: ' + err),
  complete: () => console.log('El Observer obtuvo una notificacion de completado'),
};

// Ejecutar con el objeto observador
myObservable.subscribe(myObserver);

// Logs:
// El Observer obtuvo el proximo valor: 1
// El Observer obtuvo el proximo valor: 2
// El Observer obtuvo el proximo valor: 3
// El Observer obtuvo una notificacion de completado
```

Suscribiendo cont.

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

```
myObservable.subscribe(
  x => console.log('El Observer obtuvo el proximo valor: ' + x),
  err => console.error('El Observer obtuvo un error: ' + err),
  () => console.log('El Observer obtuvo una notificacion de completado')
);
```

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

- Utilizamos el constructor `Observable` para crear un flujo observable de cualquier tipo
- El constructor toma como argumento la función suscriptor para ejecutarse cuando se invoca 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.

```
// Esta función se ejecuta cuando se llama subscribe()
function sequenceSubscriber(observer) {
  // entregue sincrónicamente 1, 2 y 3, luego complete
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.complete();

  // la función de cancelación de suscripción no necesita hacer
  // nada en este caso porque los valores se entregan sincrónicamente
  return { unsubscribe() {} };
}

// Cree un nuevo Observable que entregue la secuencia anterior
const sequence = new Observable(sequenceSubscriber);
```

Creando observables cont.

```
// ejecutar el Observable e imprimir el resultado de cada notificación
sequence.subscribe({
  next(num) { console.log(num); },
  complete() { console.log('Secuencia finalizada'); }
});

// Logs:
// 1
// 2
// 3
// Secuencia finalizada
```

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('Proximo numero: ' + num)},
  error(err) { console.log('Se recibio un error: ' + 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 y que facilita la composición de código asíncrono o basado en funciones *callback*
- RxJS proporciona una implementación del tipo `Observable` que es necesaria mientras dicho tipo no sea 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 de operaciones asíncronas existentes en observables
  - Iterar a través de los valores en un flujo
  - Mapear valores a diferentes tipos
  - Filtrar flujos de datos
  - Componer multiples flujos de datos

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 { from } from 'rxjs';
import fetch from 'node-fetch';

// Crear un Observable a partir de una promesa
const data = from(fetch('/api/endpoint'));
// Nos Suscribimos para comenzar a escuchar el resultado asíncrono
data.subscribe({
  next(response) { console.log(response); },
  error(err) { console.error('Error: ' + err); },
  complete() { console.log('Completado'); }
});
```

Funciones de creación de observables cont.

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

// Crear un Observable que publicará un valor en un intervalo
const secondsCounter = interval(1000);

// Nos suscribimos para comenzar a publicar valores
secondsCounter.subscribe(n =>
  console.log(`Pasaron ${n} segundos desde la suscripcion!`)
);
```

Funciones de creación de observables cont.

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

// Cree un Observable que genere una solicitud AJAX
const apiData = ajax('/api/data');
// Suscríbase para crear la solicitud
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()`, entre otros
- Los operadores toman las opciones de configuración y devuelven una función que toma como fuente un 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
- Desde la versión 7 de RxJS todos los operadores están expuestos en `'rxjs'`. Si estamos trabajando con la versión 6, estos se encuentra en `'rxjs/operators'`

Operadores cont.

```
import { of } from 'rxjs';
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 todas las funciones compuestas en secuencia
- Un conjunto de operadores aplicado a un observable es una receta, es decir, un conjunto de instrucciones de como producir los valores que nos interesan

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

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

// Suscríbase para obtener los valores
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 un pipe de observables
- Por ejemplo, supongamos que tienemos un observable que realiza una solicitud a una API y que obtiene la respuesta del servidor. Si el servidor devuelve un error o el valor que no existe, se produce un error. Si detectamos este error y proporcionamos un valor predeterminado, el 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';
// Devuelve "respuesta" de la API. Si ocurre un error, devuelve una matriz vacía.
const apiData = ajax('/api/data').pipe(
  map(res => {
    if (!res.response) {
      throw new Error('Se espera un valor!');
    }
    return res.response;
  }), catchError(err => of([]))
);

apiData.subscribe({
  next(x) { console.log('dato: ', x); },
  error(err) {
    console.log('errores detectados... no se ejecuta');
  }
});
```

Reintento fallido

- Asi como el operador `catchError` proporciona una forma de recuperación simple, el operador `retry` permite reintentar una solicitud fallida
- Utilizamos 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 cont.

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

const apiData = ajax('/api/data').pipe(
  retry(3), // Vuelva a intentarlo hasta 3 veces antes de fallar
  map(res => {
    if (!res.response) {
      throw new Error('Valor esperado!');
    }
    return res.response;
  }), catchError(err => of([]))
);

apiData.subscribe({
  next(x) { console.log('dato: ', x); },
  error(err) {
    console.log('errores detectados... no se ejecuta');
  }
});
```

Subject

- Es el tipo especial de `Observable` que permite enviar los datos a otros componentes o servicios
- Permite la multidifusión de valores a muchos observadores
- Hay diferentes tipos de `Subject` en Angular:
  - async subject
  - behavior subject
  - replay subject

Subject cont.

```
import { Subject, interval } from "rxjs";

const clock$ = interval(1000);
const sub = new Subject();

clock$.subscribe(sub);

sub.subscribe({ next: v => console.log('A: ', v) });

setTimeout(() => {
  sub.subscribe({ next: v => console.log('B: ', v) });
}, 3000);
```

BehaviorSubject

- El `BehaviorSubject` es el `Subject` más común en angular
- Representa el valor actual y se importa de la biblioteca RxJS
- Es similar al `Subject` pero la diferencia es que podemos establecer el valor inicial
- Siempre devolverá un valor, incluso si no se han emitido datos del suscriptor

BehaviorSubject cont.

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

let bSubject = new BehaviorSubject('a');

bSubject.next('b');

bSubject.subscribe(value => {
  console.log('Subscription got', value);
});

bSubject.next('c');
bSubject.next('d');
```

BehaviorSubject cont.

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

let subject = new Subject();

subject.next('b');

subject.subscribe(value => {
  console.log('Subscription got', value);
});

subject.next('c');
subject.next('d');
```

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` (`XHR`)
  - 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 pruebas unitarias, objetos de solicitud y respuesta tipificados, intercepción de solicitud y respuesta, APIs observables y manejo simplificado de errores

Configuración

- Para poder utilizar `HttpClient`, necesitamos 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`, podemos inyectar `HttpClient` en una clase de servicio como vemos 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()` del `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 `getConfig` del servicio devuelve un `Observable` con los 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() {
  // ahora devuelve un Observable de Config
  return this.http.get<Config>(this.configUrl);
}
```

Obteniendo datos JSON cont.

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

```
config: Config;

showConfig() {
  this.configService.getConfig()
    // clonamos el objeto de datos, usando el tipo Config
    .subscribe((data: Config) => this.config = { ...data });
}
```

Leyendo la respuesta completa

- El cuerpo de la respuesta no devuelve todos los datos que tal vez lleguemos a 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 pasar 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()
    // respuesta de tipo `HttpResponse<Config>`
    .subscribe(resp => {
      // muestra los encabezados
      const keys = resp.headers.keys();
      this.headers = keys.map(key =>
        `${key}: ${resp.headers.get(key)}`);

      // accedemos al cuerpo de forma directa (tipo `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({
      next: (data: Config) =>
        this.config = { ...data },
      error: error =>
        this.error = error
    });
}
```

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) {
    // Ocurrio un error del lado del cliente o error de red
    console.error('Ocurrio un error:', error.error.message);
  } else {
    // El Backend devolvio un codigo de respuesta insatisfactorio
    // El cuerpo de la respuesta puede contener informacion sobre lo sucedido
    console.error(`El Backend devolvio el codigo ${error.status}, ` +
       `cuerpo: ${error.error}`);
  }
  // Devuelve un observable con el mensaje de error
  return throwError('Algo malo sucedio; pruebe mas tarde.');
};
```

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), // reintenta un pedido fallido hasta 3 veces
      catchError(this.handleError) // luego manejamos el error
   );
}
```

Solicitando datos no JSON

- No todas las API devuelven datos JSON

```
getTextFile(filename: string) {
  // El tipo retornado por el get() del Observable es del tipo
  // Observable<string> dado que se especifico el tipo de respuesta
  // No hay necesidad de pasar el parametro <string> a get()
  return this.http.get(filename, {responseType: 'text'})
    .pipe(
      tap( // Log el resultado o el 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: borra un hero del servidor */
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: actualiza un hero en el server.
 * Devuelve el hero actualizado.
 */
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

Pipes

Introducción a Pipes

- Cada aplicación comienza con lo que parece una tarea simple: obtener datos, transformarlos y mostrarlos a los usuarios
- Obtener los 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, podemos enviar los valores en bruto directamente a la vista (lo cual hace que la experiencia del usuario no 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 `Date` (sin formato) *"el 15 de abril de 1988 a las 00:00:00 GMT-0700 (hora del Pacífico)"*.

Introducción a Pipes cont.

- Como podemos ver, algunos valores generan una experiencia de usuario más satisfactoria al aplicarles un 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 como 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 `birthday` 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(1973, 2, 23); // Marzo 23, 1973
}
```

- 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:
  - DatePipe
  - DecimalPipe
  - CurrencyPipe
  - AsyncPipe
  - I18nPluralPipe
  - I18nSelectPipe
  - JsonPipe
  - KeyValuePipe
  - UpperCasePipe
  - LowerCasePipe
  - PercentPipe
  - SlicePipe
  - TitleCasePipe

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, separamos los valores con dos puntos (como por ejemplo, `slice: 1: 5`)
- Vamos a modificar la plantilla de cumpleaños para darle al pipe `date` un parámetro de formato
- Luego de formatearlo, el cumpleaños del héroe del 23 de Marzo, se presentara como `23/03/73`:

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

Parametrizando Pipes cont.

- El valor del parámetro puede ser cualquier expresión de plantilla válida, como un literal de cadena o una propiedad de componente
- En otras palabras, podemos controlar el formato mediante un enlace a una propiedad de la misma manera que controlamos 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(1973, 2, 23); // Marzo 23, 1973
  toggle = true; // inicia en verdadero => 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 *23 DE MARZO DE 1973*

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

- El siguiente ejemplo, muestra *VIERNES, 23 DE MARZO DE 1973*, encadena los mismos pipes que el anterior, pero también pasa un parámetro a la fecha

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

Ejemplo de i18nPlural

- Definimos el objeto que vamos a pasar como parámetro:

```
pluralMap = {
  '=0': '0 Días',
  '=1': '1 Día',
  'other': '# Días'
};
```

- Luego lo utilizamos en la plantilla:

```
<ul>
  <li *ngFor="let unit of units">
    {{ unit.daysRemaining | i18nPlural: pluralMap }}
  </li>
</ul>
```

Pipes Personalizados

- Es posible crear pipes personalizados utilizando Angular CLI:
  `ng g pipe <nombre>`
- El siguiente ejemplo es de pipe personalizado llamado `ExponentialStrengthPipe` que puede aumentar los poderes de un héroe:

```
import { Pipe, PipeTransform } from '@angular/core';
/*
 * Aumenta el valor de forma exponencial
 * Toma un exponente como argumento que por defecto es 1.
 * Uso:
 *   valor | exponentialStrength: exponente
 * Ejemplo:
 *   {{ 2 | exponentialStrength: 10 }}
 *   formats to: 1024
*/
@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements PipeTransform {
  transform(value: number, exponent = 1): number {
    return Math.pow(value, exponent);
  }
}
```

Pipes Personalizados cont.

```
<h2>Power Boost Calculator</h2>
<label for="power-input">Normal power: </label>
<input id="power-input" type="text" [(ngModel)]="power">
<label for="boost-input">Boost factor: </label>
<input id="boost-input" type="text" [(ngModel)]="factor">
<p>
  Super Hero Power: {{ power | exponentialStrength: factor }}
</p>
```

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 `transform` 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 `transform` 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, aplicamos el decorador `@Pipe`, que se importa desde la biblioteca `@angular/core`
  - El decorador `@Pipe` nos permite definir el nombre del pipe que usaremos 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 los `pipes` integrados
- Se deben registrar los pipes personalizados, de no hacerlo, Angular informará con un error
- Se debe incluir en el arreglo de las declaraciones de `AppModule`
- El generador de `pipes` 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: pulsación de una 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 a solo 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.

```
  <p>Nuevo heroe:</p>
  <input type="text" #box
    (keyup.enter)="addHero(box.value); box.value=''"
    placeholder="hero name">
  <button (click)="reset()">Limpiar</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 estamos obteniendo el comportamiento que deseamos, 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 el misma arreglo 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 el arreglo, no se invoca ningún pipe y la pantalla no se actualiza
- Si reemplazamos el arreglo, 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

Visión general

- Ya vimos algunas directivas integradas, ahora vamos a ver como crear directivas personalizadas
- Los componentes también son Directivas
- Los componentes tienen todas las características de las Directivas, pero a diferencia de una Directiva, tienen una vista asociada (una plantilla)
- Otra diferencia entre componentes y directivas es que un único elemento HTML puede tener un único componente asociado, sin embargo, un único elemento HTML puede tener varias directivas asociadas

Creación de una directiva personalizada

- Para crear una directiva propia, podemos:
  - Utilizar Angular CLI: `ng generate directive <nombre>`
  - Crearla de forma manual

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 vamos a asociar la directiva 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 cont.

- 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 debemos hacer 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 agrega 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.setStyle(el.nativeElement, 'backgroundColor', 'gray');
  }
}
```

Constructor de directiva cont.

- Utilizamos la inyección de dependencias (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

Angular Material

Luciano Diamand

Vista previa

Instalar Angular Material

- Instalar material angular, CDK angular y las animaciones de angular
- El comando `ng add` de Angular CLI actualizará el proyecto Angular con las dependencias correctas, realizará cambios de configuración y ejecutará el código de inicialización

```
ng add @angular/material
```

Configurar animaciones

- Una vez que el paquete de animaciones esté instalado, importamos `BrowserAnimationsModule` en su aplicación para habilitar la compatibilidad con animaciones

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

@NgModule({
  ...
  imports: [BrowserAnimationsModule],
  ...
})
export class PizzaPartyAppModule { }
```

Importar los módulos de componentes

- Importamos `NgModule` para cada componente que deseamos usar:

```
import {MatButtonModule, MatCheckboxModule} from '@angular/material';

@NgModule({
  ...
  imports: [MatButtonModule, MatCheckboxModule],
  ...
})
export class PizzaPartyAppModule { }
```

Incluyendo un tema

- Se requiere incluir un tema para aplicar todos los estilos básicos y de tema a su aplicación
- Para comenzar con un tema preconstruido, incluya uno de los temas preconstruidos de Angular Material globalmente en su aplicación
- Si estamos utilizando Angular CLI, podemos agregar lo siguiente a su `styles.css`:

```
@import "~@angular/material/prebuilt-themes/indigo-pink.css";
```

Soporte de gestos

- Algunos componentes (`mat-slide-toggle`, `mat-slider`, `matTooltip`) dependen de `HammerJS` para los gestos
- Para obtener el conjunto completo de funciones de estos componentes, se debe cargar `HammerJS` en la aplicación
- Puede agregar `HammerJS` a su aplicación a través de `npm`, un CDN (como el CDN de Google) o directamente desde su aplicación
- Para instalarlo con `npm`

```
npm install --save hammerjs
```

- Después de la instalación, podemos importarlo en el punto de entrada de su aplicación (por ejemplo, `src/main.ts`)

```
import 'hammerjs';
```

Añadir iconos de Material (opcional)

- Si deseamos utilizar el componente `mat-icon` con los iconos oficiales de Material Design, cargamos la fuente del icono en el `index.html`

```
<link href="https://fonts.googleapis.com/icon?family=Material+Icons" re
```

Componentes de Material

- Angular Material comprende una gama de componentes que implementan patrones de interacción comunes de acuerdo con la especificación Angular Material
  - Controles de formularios
  - Navegación
  - Diseño
  - Botones e indicadores
  - Popups y modales
  - Tablas de datos

Controles de formularios

Controles de formularios

- Controles que recopilan y validan la entrada del usuario
  - Input
  - Autocomplete
  - Checkbox
  - Datepicker
  - Form field
  - Radio button
  - Select
  - Slider
  - Slide toggle

Input

- El módulo esta disponible en:
  `import \{ MatInputModule \} from '@angular/material/input';`
- Directiva que permite que una entrada nativa funcione dentro de MatFormField:
  `matInput`

```
<mat-form-field class="example-form-field" appearance="fill">
  <mat-label>Clearable input</mat-label>
  <input matInput type="text" [(ngModel)]="value">
  <button *ngIf="value" matSuffix mat-icon-button aria-label="Clear" (c
    <mat-icon>close</mat-icon>
  </button>
</mat-form-field>
```

Autocomplete cont.

- Comenzamos agregando un `matInput` regular a la plantilla
- Supongamos que estamos utilizando la directiva `formControl` de `ReactiveFormsModule` para rastrear el valor de la entrada

Autocomplete cont.

```
<mat-form-field>
  <input type="text" matInput [formControl]="myControl">
</mat-form-field>


<mat-autocomplete>
  <mat-option *ngFor="let option of options" [value]="option">
    {{ option }}
  </mat-option>
</mat-autocomplete>
```

Checkbox

- `<mat-checkbox>` proporciona la misma funcionalidad que un nativo `<input type="checkbox">` mejorado con estilos y animaciones de Material Design
- La etiqueta de la casilla de verificación se proporciona como contenido del elemento `<mat-checkbox>`. La etiqueta se puede colocar antes o después de la casilla de verificación configurando la propiedad `labelPosition` en '`before`' o '`after`'
- Si no desea que la etiqueta aparezca junto a la casilla de verificación, puede usar `aria-label` o `aria-labelledby` para especificar una etiqueta adecuada
- Usar con `@angular/forms`
- `<mat-checkbox>` es compatible con `@angular/forms` y es compatible con ambos `FormsModule` y `ReactiveFormsModule`
- Estado indeterminado
- `<mat-checkbox>` admite un estado indeterminado, similar al `<input type="checkbox">` nativo. Si bien la propiedad `indeterminate` del `checkbox` es verdadera, se representará como indeterminada independientemente

Navigation

Navigation

- Menús, sidenavs y barras de herramientas que organizan su contenido
  - Menu
  - Sidenav
  - Toolbar

Layout

Layout

- Bloques de construcción esenciales para presentar su contenido
  - Card
  - Divider
  - Expansion Panel
  - Grid list
  - List
  - Stepper
  - Tabs
  - Tree

Botones e indicadores

Botones e indicadores

- Botones, conmutadores, indicadores de estado y progreso
  - Button
  - Button Toggle
  - Badge
  - Chips
  - Icon
  - Progress Spinner
  - Progress Bar
  - Ripples

Popups y Modales

Popups y Modales

- Componentes flotantes que se pueden mostrar u ocultar dinámicamente
  - Buttom Sheet
  - Dialog
  - Snackbar
  - Tooltip

Tablas de datos

Tablas de datos

- Herramientas para mostrar e interactuar con datos tabulares
  - Paginator
  - Sort header
  - Table

Pruebas unitarias

Introducción

- El 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 (e2e)**: son pruebas en las que se desea imitar a un usuario real que visita la aplicación. 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 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
- **Cypress** 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**: 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**: 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**: cuando se 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()`
  - Tengamos en cuenta que cuando escribimos las afirmaciones, debemos 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 necesitemos 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.

- Debemos 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 las cadenas sean descriptivos

Karma

- La ejecución manual de las pruebas de Jasmine teniendo que actualizar la pestaña del navegador repetidamente y para diferentes navegadores cada vez que editamos el código puede resultar aburrido, repetitivo y consumir mucho tiempo
- `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`, dado que, 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
- Karma observa los cambios en nuestros archivos de desarrollo, y si detecta un cambio y vuelve a ejecutar las pruebas automáticamente

Pruebas deshabilitadas o enfocadas

- Podemos deshabilitar las pruebas sin necesidad de comentarlas
- Para ello basta con agregar una `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 podemos centrarnos 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 la mayoría de 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(() => {
    TestBed.configureTestingModule({});
    service = TestBed.inject(AuthService);
  });

  afterEach(() => {
    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 cont.

- 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);
  });

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

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

Autenticación y Autorización

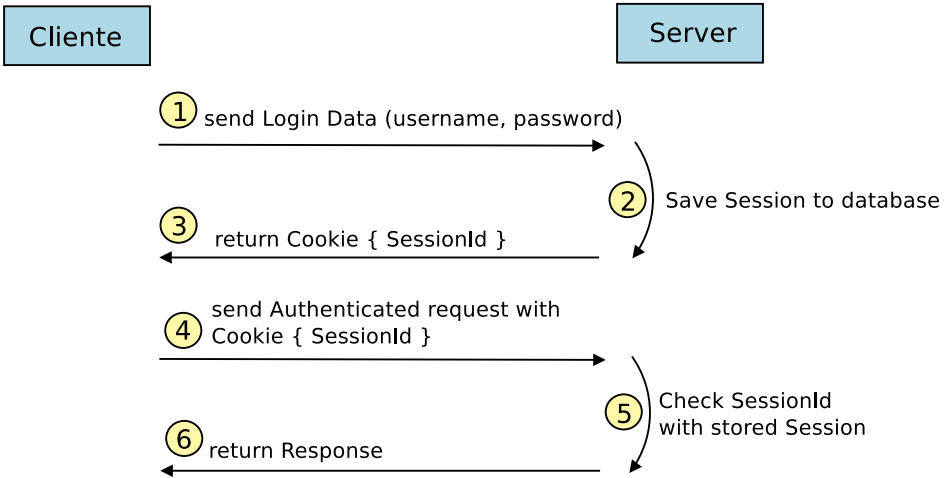
Luciano Diamand

JSON Web Token (JWT)

Introducción a JSON Web Token (JWT)

- Las cookies llevan tiempo utilizándose para autentificar a los usuarios de Internet; y funcionan muy bien para ciertas aplicaciones
- Sin embargo, hay casos en donde es necesaria una mayor flexibilidad
- `JSON Web Token`, es un nuevo estándar abierto adoptando cada vez más por sitios web y aplicaciones importantes

Autenticación basada en sesión



¿Qué es un JSON Web Token?

- Un `JSON Web Token` es un token de acceso estandarizado en el `RFC 7519` que permite el intercambio seguro de datos entre dos partes
- Contiene toda la información importante sobre una entidad, lo que implica que no hace falta consultar una base de datos ni que la sesión tenga que guardarse en el servidor (sesión sin estado)
- Por este motivo, los `JWT` son especialmente populares en los procesos de autentificación
- Con este estándar es posible cifrar mensajes cortos, dotarlos de información sobre el remitente y demostrar si este cuenta con los derechos de acceso requeridos
- Los propios usuarios solo entran en contacto con el token de manera indirecta: por ejemplo, al introducir el nombre de usuario y la contraseña en una interfaz
- La comunicación como tal entre las diferentes aplicaciones se lleva a cabo en el lado del cliente y del servidor

¿Cómo se estructura un JSON Web Token?

- Un `JWT` consta de tres partes, todas ellas codificadas en `Base64` y separadas por un punto:

  `HEADER.PAYLOAD.SIGNATURE`

- El cliente adjunta el `JWT` en el encabezado `x-access-token` o `Authorization`:

  ```
  x-access-token: HEADER.PAYLOAD.SIGNATURE
  ```

  o

  ```
  Authorization: Bearer HEADER.PAYLOAD.SIGNATURE
  ```

JWT: Encabezado

- Contiene el tipo de token y el algoritmo de la firma y/o cifrado utilizados, por ejemplo: `{ "alg": "HS256", "typ": "JWT" }`
- Es recomendable introducir `JWT` como tipo, que hace referencia al mime type `application/jwt` de la IANA
- En el ejemplo anterior, el header indica que `HMAC-SHA256`, abreviado como `HS256`, se utiliza para firmar el token
- Otros métodos de cifrado típicos son `RSA`, con `SHA-256 (RS256)`, y `ECDSA`, con `SHA-256 (ES256)`
- No se recomienda prescindir del cifrado, aunque sí se puede especificar `none` si los datos no requieren un nivel de protección alto
- Los posibles valores están estandarizados por `JSON-Web-Encryption` según el `RFC 7516`

Payload

- Contiene la información real que se transmitirá a la aplicación
- La información se proporciona como pares key/value (clave-valor); las claves se denominan `claims` en `JWT`
- Hay tres tipos diferentes de `claims`:
  - Los `claims` registrados son los que figuran en el IANA JSON Web Token Claim Register y cuyo propósito se establece en un estándar: ejemplos `iss`, el emisor, `aud`, de audiencia y `exp`, tiempo de expiración
  - Se utilizan nombres de claim cortos para abreviar el token lo máximo posible
  - Los `claims` privados están destinados a los datos que intercambiamos especialmente con nuestras propias aplicaciones
  - Si bien los `claims` públicos contienen información como nombre o correo electrónico, los claims privados son más concretos
  - Todos los claims son opcionales, por lo que no es obligatorio utilizar todos los claims registrados
- Un payload podría estructurarse, por lo tanto, de la siguiente manera:

```
{ "sub": "123", "name": "Alicia", "exp": 30 }
```

Firma

- Se crea utilizando la codificación Base64 del `header` y del `payload`, así como el método de firma o cifrado especificado
- La estructura viene definida por `JSON Web Signature` (`JWS`), un estándar establecido en el `RFC 7515`
- Para que la firma sea eficaz, es necesario utilizar una clave secreta que solo conozca la aplicación original
- La firma verifica que el mensaje no se ha modificado por el camino
- Si el token está firmado con una clave privada, también garantiza que el remitente del `JWT` sea el correcto

¿Cómo funciona un JSON Web Token?

- El inicio de sesión de usuario ejemplifica la función del `JSON Web Token`
- Antes de utilizar el `JWT`, hay que establecer una clave secreta
- Una vez que el usuario ha introducido correctamente sus credenciales, el `JWT` se devuelve con la clave y se guarda localmente
- La transmisión debe realizarse a través de `HTTPS` para que los datos estén mejor protegidos.
- De esta manera, cada vez que el usuario accede a recursos protegidos, como a una `API` o a una ruta protegida, el user agent utiliza el `JWT` como parámetro (por ejemplo, `JWT` para peticiones `GET`) o como header de autorización (para `POST`, `PUT`, `OPTIONS` y `DELETE`)
- La otra parte puede descifrar el `JSON Web Token` y ejecutar la solicitud si la verificación se realiza correctamente

¿En qué casos se utiliza JSON Web Token?

- `JSON Web Token` ofrece varias ventajas en comparación con el método tradicional de autentificación y autorización con cookies, por lo que se utiliza en las siguientes situaciones:
  - Aplicaciones `REST`: En las aplicaciones `REST`, el `JWT` garantiza la ausencia de estado enviando los datos de autentificación directamente con la petición.
  - Intercambio de recursos de origen cruzado: `JSON Web Token` envía información mediante el llamado cross-origin resource sharing, lo cual le da una gran ventaja sobre las cookies, que no suelen enviarse con este procedimiento.
  - Uso de varios frameworks: `JSON Web Token` está estandarizado y puede utilizarse una y otra vez. Cuando se emplean múltiples frameworks, los datos de autentificación pueden compartirse más fácilmente.

Ejemplo práctico de JWT

Veamos un ejemplo de JWT

- El encabezado es de la siguiente forma:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

- El payload del JSON Web Token podría tener el siguiente aspecto:

```
{
  "sub": "1234567890",
  "name": "ldiamand",
  "iat": 1516239022
}
```

- Para lograr la estructura real del JWT (tres partes separadas por puntos), el header y el payload deben codificarse con Base64

Ejemplo práctico de JWT cont.

- La codificación del header se realizaría del siguiente modo:

```
base64Header = base64Encode(header);
// eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

- Lo mismo debe hacerse para el payload:

```
base64Payload = base64Encode(payload);
// eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6ImxkaWFtYW5kIiwiaWFOIjoxNTE2MjM5MDIyfQ
```

- Ahora creamos la firma. En el header, indicamos que se firme con `HMAC-SHA256`:

```
signature = HS256(base64Header + '.' + base64Payload, 'secret');
// R4UihRfDiIkZlb_JjgsmFNMsXIyXKiCAiSltJIpQH6w
```

- Por último, estas tres partes deben unirse con puntos entre ellas:

```
Token = base64Header + '.' + base64Payload + '.' + signature;
// eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6ImxkaWFtYW5kIiwiaWFO
```

- La mayoría de los lenguajes de programación actuales proporcionan bibliotecas para generar `JWT`, por lo que no es necesaria la conversión manual

Autenticación con JWT en Angular

Luciano Diamand

Grupo San Cristóbal

Guarda de autenticación

- La guarda de autenticación es una protección de la ruta de Angular que se usa para evitar que los usuarios no autenticados accedan a rutas restringidas
- Se hace mediante la implementación de la interfaz `CanActivate` que permite que la protección decida si una ruta se puede activar con el método `canActivate()`
- Si el método devuelve `true`, la ruta está activada (permitira continuar); de lo contrario, si el método devuelve `false`, la ruta está bloqueada
- El guarda de autenticación usa el servicio de autenticación para comprobar si el usuario ha iniciado sesión; si lo ha hecho, devuelve `true` desde el método `canActivate()`; de lo contrario, devuelve `false` y redirige al usuario a la página de inicio de sesión

Guarda de autenticación cont.

```
@Injectable({ providedIn: 'root' })
export class AuthGuard implements CanActivate {
  constructor(
    private router: Router,
    private authenticationService: AuthenticationService
  ) { }

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
    const currentUser = this.authenticationService.currentUserValue;
    if (currentUser) {
      // El usuario esta logeado
      return true;
    }

    // No logeado, por lo tanto es redirigido a la pagina de login
    this.router.navigate(['/login'], { queryParams: { returnUrl: state.url } });
    return false;
  }
}
```

Interceptor de errores HTTP

- El interceptor de errores intercepta las respuestas HTTP de la API para verificar si hubo algún error
- Si hay una respuesta `401` no autorizada, el usuario se desconecta automáticamente de la aplicación, todos los demás errores se vuelven a enviar al servicio de llamadas para que se pueda mostrar una alerta con el error en la pantalla
- Se implementa usando la clase `HttpInterceptor` incluida en `HttpClientModule`, al extender la clase `HttpInterceptor` podemos crear un interceptor personalizado para capturar todas las respuestas de error del servidor en una sola ubicación
- Los interceptores HTTP se agregan a la linea de solicitudes en la sección de proveedores del archivo `app.module.ts`

Interceptor de errores HTTP cont.

```
@Injectable()
export class ErrorInterceptor implements HttpInterceptor {
  constructor(private authenticationService: AuthenticationService) { }

  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    return next.handle(request).pipe(catchError(err => {
      if (err.status === 401) {
        // llama a logout si recibimos una respuesta 401 del API
        this.authenticationService.logout();
        location.reload(true);
      }

      const error = err.error.message || err.statusText;
      return throwError(error);
    }));
  }
}
```

Interceptor JWT

- El interceptor JWT intercepta las solicitudes HTTP de la aplicación para agregar un token de autenticación JWT al encabezado de autorización si el usuario ha iniciado sesión
- Se implementa usando la clase `HttpInterceptor` incluida en `HttpClientModule`, al extender la clase `HttpInterceptor` puede crear un interceptor personalizado para modificar las solicitudes http antes de que se envíen al servidor
- Los interceptores HTTP se agregan a línea de solicitudes en la sección de proveedores del archivo `app.module.ts`

Interceptor JWT

```
@Injectable()
export class JwtInterceptor implements HttpInterceptor {
  constructor(private authenticationService: AuthenticationService) { }

  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    // add authorization header with jwt token if available
    let currentUser = this.authenticationService.currentUserValue;
    if (currentUser && currentUser.token) {
      request = request.clone({
        setHeaders: {
          Authorization: `Bearer ${currentUser.token}`
        }
      });
    }

    return next.handle(request);
  }
}
```

Modelo del usuario

- El modelo de usuario es una clase que define las propiedades de un usuario

```
export class User {
    id: number;
    username: string;
    password: string;
    firstName: string;
    lastName: string;
    token?: string;
}
```

Servicio de autenticación

- El servicio de autenticación se utiliza para iniciar sesión y cerrar sesión en la aplicación Angular, notifica a otros componentes cuando el usuario inicia y cierra sesión y permite el acceso al usuario actualmente conectado
- Los `Subject` y `Observables` de RxJS se utilizan para almacenar el objeto `currentUser` y notificar a otros componentes cuando el usuario inicia y cierra sesión en la aplicación
- Los componentes de Angular pueden invocar a `subscribe()` sobre la propiedad `currentUser` para recibir notificaciones de cambios, y las notificaciones se envían cuando se llama al método `this.currentUserSubject.next()` en los métodos `login()` y `logout()`, pasando el argumento a cada suscriptor
- El método `login()` envía las credenciales de usuario a la API a través de una solicitud HTTP POST para la autenticación
- Si tiene éxito, el objeto de usuario que incluye un token de autenticación JWT se almacena en `localStorage` para mantener al usuario conectado entre actualizaciones de página

Servicio de autenticación cont.

- Luego, el objeto de usuario se publica para todos los suscriptores con la llamada a `this.currentUserSubject.next(user)`
- El `constructor()` del servicio inicializa `currentUserSubject` con el objeto `currentUser` del `localStorage`, lo que permite al usuario permanecer conectado entre actualizaciones de página o después de cerrar el navegador
- Luego, la propiedad `currentUser` se establece en `this.currentUserSubject.asObservable()`; lo que permite que otros componentes se suscriban a `currentUser`, pero no les permite publicar en `currentUserSubject`, por lo que solo se puede iniciar y cerrar sesión en la aplicación a través del servicio de autenticación
- El getter de `currentUserValue` permite que otros componentes obtengan fácilmente el valor del usuario conectado actualmente sin tener que suscribirse a `currentUser`
- El método `logout()` elimina el objeto de usuario actual del `localStorage` y publica un valor `null` en `currentUserSubject` para notificar a todos los suscriptores que el usuario ha cerrado sesión

Servicio de autenticación cont.

```
@Injectable({ providedIn: 'root' })
export class AuthenticationService {
  private currentUserSubject: BehaviorSubject<User>;
  public currentUser: Observable<User>;

  constructor(private http: HttpClient) {
    this.currentUserSubject = new BehaviorSubject<User>(
        JSON.parse(localStorage.getItem('currentUser')));
    this.currentUser = this.currentUserSubject.asObservable();
  }

  public get currentUserValue(): User {
    return this.currentUserSubject.value;
  }
```

Servicio de autenticación cont.

```
  login(username: string, password: string) {
    return this.http.post<any>(`${environment.apiUrl}/users/authenticate`, { username, password })
      .pipe(map(user => {
      // Almacena los detalles del usuario y el token JWT para mantener
      // al usuario logeado incluso entre actualizaciones de las páginas
      localStorage.setItem('currentUser', JSON.stringify(user));
      this.currentUserSubject.next(user);
      return user;
    }));
  }

  logout() {
    // Elimina al usuario del localStorage para cerrar sesión
    localStorage.removeItem('currentUser');
    this.currentUserSubject.next(null);
  }
}
```

Componente de Login

- El componente de inicio de sesión utiliza el servicio de autenticación para iniciar sesión en la aplicación
- Si el usuario ya ha iniciado sesión, se le redirigirá automáticamente a la página de inicio

```
@Component({ templateUrl: 'login.component.html' })
export class LoginComponent implements OnInit {

  loginForm: FormGroup;
  loading = false;
  submitted = false;
  returnUrl: string;
  error = '';

  constructor(
    private formBuilder: FormBuilder,
    private route: ActivatedRoute,
    private router: Router,
    private authenticationService: AuthenticationService
  ) {
```

Componente de Login cont.

```
    // redirige al Inicio si ya está logeado
    if (this.authenticationService.currentUserValue) {
      this.router.navigate(['/']);
    }
  }

  ngOnInit() {
    this.loginForm = this.formBuilder.group({
      username: ['', Validators.required],
      password: ['', Validators.required]
    });

    // get return url from route parameters or default to '/'
    this.returnUrl = this.route.snapshot.queryParams['returnUrl'] || '/';
  }

  // convenience getter for easy access to form fields
  get f() { return this.loginForm.controls; }

  onSubmit() {
    this.submitted = true;
```

Componente de Login cont.

```
    // stop here if form is invalid
    if (this.loginForm.invalid) {
      return;
    }

    this.loading = true;
    this.authenticationService.login(this.f.username.value, this.f.password.value)
      .pipe(first())
      .subscribe(
        data => {
          this.router.navigate([this.returnUrl]);
        },
        error => {
          this.error = error;
          this.loading = false;
        });
  }
}
```

Componente de la aplicación

- Suscribe al Observable `currentUser` al servicio de autenticación para que pueda mostrar/ocultar de forma reactiva la barra de navegación principal cuando el usuario inicia/cierra sesión en la aplicación
- El componente de la aplicación contiene un método `logout()` que se llama desde el enlace de cierre de sesión en la barra de navegación principal para cerrar la sesión del usuario y redirigirlo a la página de inicio de sesión

Componente de la aplicación cont.

```
@Component({ selector: 'app', templateUrl: 'app.component.html' })
export class AppComponent {
  currentUser: User;

  constructor(
    private router: Router,
    private authenticationService: AuthenticationService
  )
  {
    this.authenticationService.currentUser.subscribe(x => this.currentUser = x);
  }

  logout() {
    this.authenticationService.logout();
    this.router.navigate(['/login']);
  }
}
```

Módulo de aplicación

- Aquí agregaremos los interceptores de Error y JWT

```
@NgModule({
    imports: [
      ...
    ],
    declarations: [
      ...
    ],
    providers: [
        { provide: HTTP_INTERCEPTORS, useClass: JwtInterceptor, multi: true },
        { provide: HTTP_INTERCEPTORS, useClass: ErrorInterceptor, multi: true },

        // proveedor utilizado para crear el backend falso
        fakeBackendProvider
    ],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

Configuración del entorno de desarrollo

- La configuración del entorno de desarrollo contiene variables necesarias para ejecutar la aplicación en desarrollo
- Se accede a la configuración del entorno importando el objeto del entorno en cualquier servicio Angular del componente con la línea `import {environment}` desde `'@environments/environment'` y accediendo a las propiedades del objeto `environment`

```
export const environment = {
  production: false,
  apiUrl: 'http://localhost:4000'
};
```

Configuración del entorno de producción

- La configuración del entorno de producción contiene variables necesarias para ejecutar la aplicación en producción
- Esto le permite crear la aplicación con una configuración diferente para cada entorno diferente (por ejemplo, producción y desarrollo) sin actualizar el código de la aplicación
- Cuando compilamos la aplicación para producción con el comando `ng build --prod`, el archivo de salida `environment.ts` se reemplaza por `environment.prod.ts`

```
export const environment = {
  production: true,
  apiUrl: 'https://api.server.com'
};
```

Últimas diapositivas

Luciano Diamand

Gracias!

Y que el Código Fuente te acompañe