Introducción a Angular

Rubén Gómez Olivencia

2025-09-08

- Github: https://github.com/yuki

Puedes encontrar la última versión de este libro en formato **HTML** en el siguiente link, así como otros libros que he creado. Para descargar el código fuente en formato **Markdown** visita el repositorio en GitHub.

Información

Por favor, ponte en contacto conmigo si encuentras algún fallo, falta de ortografía o quieres mejorar de alguna manera este libro. Gracias.

I Introducción a Angular

II Conceptos fundamentales

Introducción a Angular

1. Introducción

Angular es un *framework front-end* desarrollado por **Google** para construir **aplicaciones web modernas, escalables y mantenibles**. Utiliza **TypeScript** como lenguaje principal y está basado en una arquitectura por **componentes**, lo que permite dividir la aplicación en piezas reutilizables y fáciles de gestionar.

Una de las ventajas de usar Angular es la creación de **SPA** (*single-page applications*). Las SPA permiten que el usuario interactúe con la aplicación reescribiendo partes de la web actual con datos obtenidos del servidor. De esta manera, permite construir interfaces dinámicas y estructuradas.

1.1. Historia

Es importante conocer un poco la historia de Angular, ya que originalmente, la primera versión se llamaba AngularJS debido a que estaba basado en Javascript.

Año	Versión	Características destacadas
2010	AngularJS (1.x)	Basado en JavaScript y patrón MVC
2016	Angular 2	Reescritura completa, TypeScript
2016-2023	Angular 4 → 15	Mejora del rendimiento y CLI
2024	Angular 16-17	Standalone Components, Signals
2025	**Angular 20**	Mejoras de rendimiento y nuevas sintaxis

Tal como se puede ver, desde la versión 2 se reescribió para hacer uso de TypeScript.

1.2. Arquitectura básica de Angular

Una aplicación Angular se compone principalmente de:

- **Componentes** (`Component`): representan las pantallas o partes visuales. En la documentación oficial existe mucha información acerca de cómo funcionan, por lo que es recomendable ir a ella para ver todos los detalles.
- **Servicios** (`Service`): contienen la lógica o manipulan datos.
- **Plantillas (*templates*)**: definen el HTML de cada componente.
- **Rutas (*Routing*)**: gestionan la navegación entre páginas.

Estos elementos trabajan juntos siguiendo el patrón **MVC** (Modelo - Vista - Controlador) adaptado al entorno web.

En las últimas versiones de Angular los componentes por defecto son *standalone* (facilita la migración a otras aplicaciones) mientras que anteriormente eran de tipo **NgModule**.

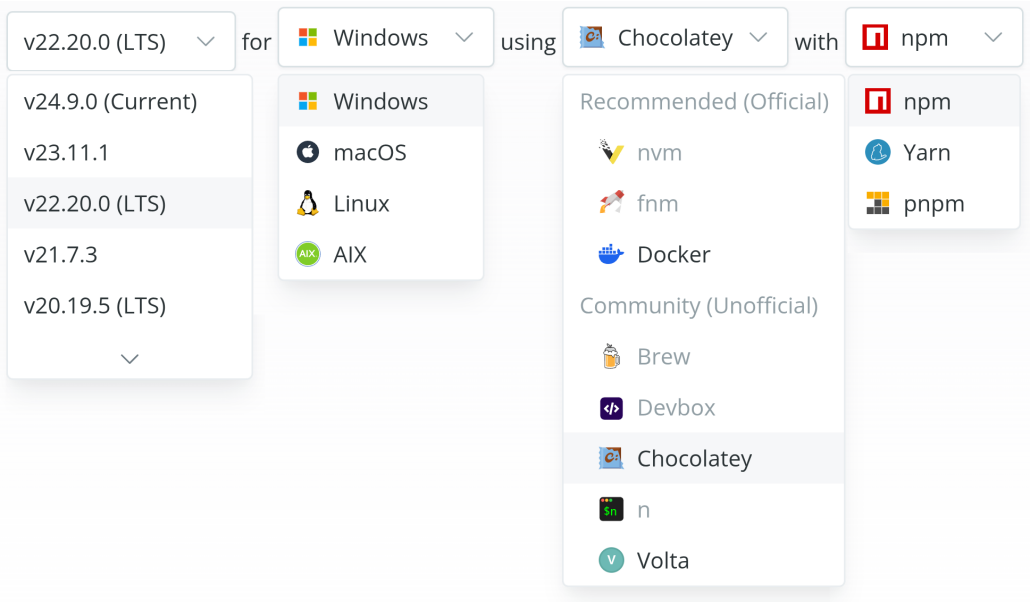
2. Cómo desarrollar en Angular

Para poder desarrollar una aplicación con Angular necesitamos tener instalado:

- Node.js: es un entorno de ejecución que nos permite ejecutar código JavaScript fuera de un navegador. Muy conocido en entornos de desarrollo web gracias al gestor de paquetes npm.
- **Angular CLI**: Es la herramienta oficial de Angular, que cuenta con un interfaz de línea de comandos que nos va a permitir crear el esqueleto de una aplicación a través de un asistente.

2.1. Instalar Node

Desde la web para descargar Node, tenemos distintas opciones a elegir:



Opciones de descargas

- **Versión**: Node.js cuenta con distintas versiones, que varían en su ciclo de vida y de mantenimiento. Algunas de las versiones son nombradas como **LTS** (*Long Term Support*, con soporte a largo plazo), ideales para usar en servidores.
- **Sistema operativo**: Debemos elegir el sistema donde queremos realizar la instalación.
- **Sistema de instalación**: Dependiendo del sistema operativo, podemos elegir distintas maneras de instalar Node. Podemos destacar:
  - **NVM**: *node versión manager*, es un gestor de versiones de Node. Nos permite tener distintas versiones instaladas en el mismo equipo, por lo que es ideal si desarrollamos con él.
  - **Docker**: para desarrollar usando imágenes Docker.
  - **Brew**: Es un gestor para instalar aplicaciones en MacOS, para no tener que ir a cada web para

descargarlos. También nos facilita poder actualizarlas desde el propio gestor.

- **Chocolatey**: Es un gestor para poder instalar aplicaciones en Windows desde la línea de comandos.

- **Gestor de paquetes**: Junto con Node se pueden instalar distintos gestores de paquetes, que después usaremos durante el desarrollo de nuestras aplicaciones.
  - **npm**: es el gestor de paquetes utilizado por defecto con Node.js. Originalmente creado por Isaac Z. Schlueter, hoy en día está bajo el amparo de Github.
  - **yarn**: la alternativa más extendida a *npm*, creado en una colaboración entre Facebook, Google y otras compañías.

Aparte de estos sistemas más "modernos", en la propia web se puede descargar un instalador clásico para cada sistema.

2.2. Instalar Angular CLI

Tras instalar Node, vamos a instalar el cliente de consola de Angular. Para ello usaremos el gestor de paquetes **npm**:

Instalar Angular CLI

```
ruben@vega:~$ npm install -g @angular/cli

ruben@vega:~$ npm -g ls
/home/ruben/.nvm/versions/node/v24.8.0/lib
├── @angular/cli@20.3.6
└── npm@11.6.0

ruben@vega:~$ ng version
Angular CLI: 20.3.6
Node: 24.8.0
Package Manager: npm 11.6.0
OS: linux x64
```

De esta manera tenemos instalado, **de manera global**, el CLI de Angular, por lo que ya podremos crear nuestra primera aplicación.

2.3. Desarrollar con Docker

Para no depender del sistema operativo, y poder realizar cambios de versiones en cualquier momento, se va a explicar cómo desarrollar usando Docker y Visual Studio Code como IDE (en inglés *Integrated Development Environment*, o entorno integrado de desarrollo).

Si estamos en un entorno Windows, se recomienda realizar lo siguiente desde un **WSL**, teniendo en cuenta el rendimiento de los sistemas de ficheros.

Crear contenedor para desarrollo

```
ruben@vega:~$ docker run  -it --name angular -p 4200:4200 \
--entrypoint /bin/bash -v "$(pwd):/workdir" --workdir /workdir node:24
root@a1d346824085:/workdir#
```

Los parámetros explicados:

- `-p 4200:4200`: por defecto es el puerto que se usa con Angular.
- `--entrypoint /bin/bash`: por defecto, la imagen ejecuta como *entrypoint* el comando `node`, y de esta manera lo sobreescribimos para usar `/bin/bash` al entrar en el contenedor.
- `-v "\$(pwd):/workdir"`: para enlazar el directorio en el que estamos como volumen persistente de datos, y configurado en la ruta `/workdir` dentro del contenedor.
- `--workdir /workdir`: el directorio que usamos al entrar en el contenedor.

Para poder desarrollar con Visual Studio Code dentro del contenedor vamos a necesitar la extensión remote development. Una vez instalada, nos aparecerá un nuevo icono en el lateral del IDE y podremos ver los contenedores que tenemos arrancados:

Con esta extensión podemos conectarnos al contenedor, se nos abrirá una nueva ventana del IDE, y desde ahí podremos abrir el directorio `/workdir` que está dentro del contenedor.

¡Cuidado!

Trabajar dentro de un contenedor no nos permitirá usar el emulador de Android/iOS de manera sencilla.

3. Crear primera aplicación

Vamos a crear una primera aplicación para ver cómo funciona el asistente de generación del proyecto:

Crear aplicación

```
ruben@vega:~$ ng new pruebas
```

Nos realiza las siguientes preguntas:

- **Formato de hojas de estilo**: A la hora de desarrollar con Angular podemos hacer uso de distintos formatos en los que queremos escribir nuestros estilos:
  - **CSS**: El sistema de hojas de estilo en cascada (CSS) tradicional en el desarrollo web.
  - **Sass (SCSS)**: Del inglés *syntactically awesome style sheets*, es un lenguaje pre-procesado que tras ser interpretado genera CSS. Nació en 2006 para cubrir carencias que tenía CSS como variables, reutilización de código…
  - **Sass (Indented)**: Es una variante del caso anterior utilizando un metalenguaje.
  - **Less**: Otro sistema pre-procesado para generar hojas de estilo, que nació en 2009 como alternativa a Sass.
- **Habilitar SSR y SSG**: Por defecto no se habilita. Se recomienda leer los siguientes enlaces para decidir si activar estas opciones o no:
  - **SSR**: *Server-Side Rendering* permite generar HTML en el lado del servidor para la petición inicial de una ruta, de esta manera permite contenido dinámico para bueno SEO.
  - **SSG**: *Static Site Generation* pre-renderiza las páginas en HTML estáticos cuando se "compila" el proyecto. El servidor envía HTML pre-compilado para la carga inicial de la página
- Crear aplicación ***zoneless*** sin **zone.js**: Por defecto la respuesta es "no". Zone.js usa eventos y tareas *async* como indicadores para cuando los estados de la aplicación puedan ser modificados.
- Uso de **utilidades de Inteligencia Artificial** en el proyecto: con la expansión de la IA, al generar un nuevo proyecto podemos hacer que se configure para distintos LLM.

Para no tener que responder a estas preguntas el CLI nos permite añadir parámetros y valores al crear el proyecto. Se puede ver con `ng new --help`.

3.1. Estructura del proyecto

Una vez creado el proyecto podremos comprobar cómo se ha creado una estructura de directorios y ficheros donde estará nuestra aplicación. Es interesante comprobar cómo también nos ha generado un repositorio GIT con el primer commit ya generado.

Dentro de la estructura generada podemos destacar:

- `angular.json`: Es el fichero de configuración de nuestra aplicación. A pesar de no entender algunos aspectos de la configuración es conveniente echar un ojo, ya que otros son auto-explicativos.
- `node_modules`: El directorio donde se guardan las dependencias que necesita nuestro proyecto. Este directorio está ignorado en GIT.
- `package.json`: Es el fichero que contiene qué dependencias necesita nuestro proyecto, indicando la versión, si es para desarrollo o producción, y los scripts necesarios para ejecutar/"compilar"/testear nuestra aplicación.

- `public`: Es el directorio donde podemos guardar *assets* públicos.
- `src`: El directorio donde se va a guardar el código de nuestra aplicación. Vamos a detallar algunos de los ficheros que hay dentro de este directorio.
  - `index.html`: Es la plantilla principal del HTML que se va a generar alrededor de nuestra aplicación.
  - `styles.scss`: Son los estilos generales de nuestra aplicación. Más adelante hablaremos de ello.
  - `app`: Es el directorio donde van a estar los componentes y el código de nuestra aplicación. En el siguiente tema se detallará más.
    - `app.routes.ts`: Es el fichero de rutas a las que se puede acceder en nuestra aplicación. Más adelante se explicará cómo funcionan las rutas.

3.2. Ejecutar aplicación

Para ejecutar el proyecto recién creado, entramos al directorio y ejecutamos `ng serve`:

Ejecutar aplicación

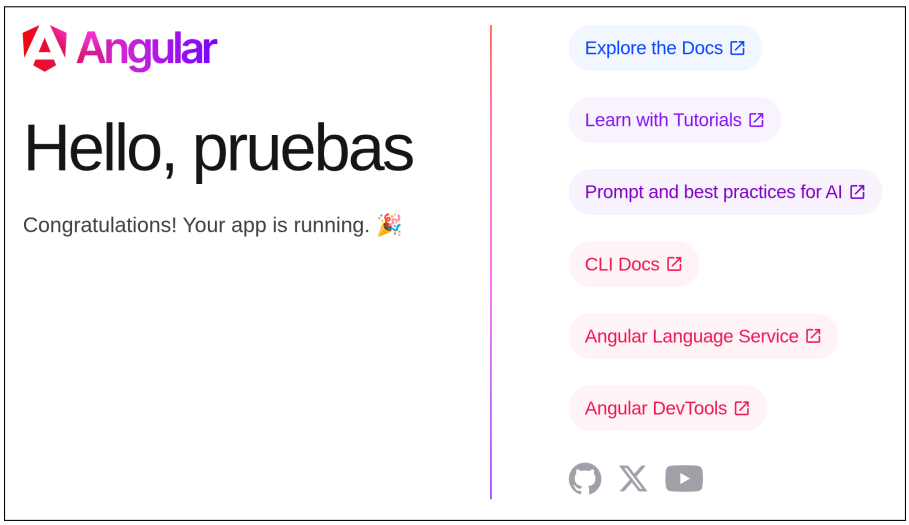
```
ruben@vega:~/pruebas $ ng serve --host 0.0.0.0
Initial chunk files | Names         | Raw size
main.js             | main          | 47.75 kB |
styles.css          | styles        | 96 bytes |
polyfills.js        | polyfills     | 95 bytes |

                    | Initial total | 47.94 kB

Application bundle generation complete.
[1.336 seconds] - 2025-10-23T16:19:11.364Z

Watch mode enabled. Watching for file changes...
NOTE: Raw file sizes do not reflect development server
per-request transformations.
  -  Local:   http://localhost:4200/
  -  Network: http://172.17.0.2:4200/
  -  press h + enter to show help
```

Si abrimos nuestro navegador en la dirección indicada http://localhost:4200, veremos lo siguiente:



4. Continuar desarrollo de una aplicación

Si queremos continuar el desarrollo de una aplicación en otro equipo, o un proyecto creado hace tiempo que tenemos en GitHub, es necesario crear un entorno de desarrollo es similar a lo visto anteriormente, con matices.

Para crear un nuevo entorno, los pasos a dar serían los siguientes:

1. Descargar código fuente del proyecto
2. Asegurar que tenemos instalado el Angular/Cli
3. Instalar dependencias dentro del directorio del proyecto

Instalar dependencias

```
root@025c365f282c:/workdir/proyecto# npm install
```

4. Ejecutar aplicación

Tal como se puede ver, la única diferencia respecto a crear un proyecto nuevo es el punto de instalar las dependencias, ya que sin estas nuestra aplicación no podrá funcionar.

5. Poner en producción

Cuando necesitemos poner nuestra aplicación en producción Angular tiene un asistente que nos permite realizar el despliegue en distintos servicios como Firebase, GitHub Pages o Amazon S3.

Es importante entender cuáles son los pasos que hay que realizar para poner nuestra aplicación en producción:

1. **Construir los ficheros de la aplicación**: El código TypeScript, junto con las vistas HTML y su CSS, se transpila y genera los ficheros finales necesarios de nuestra aplicación. Estos ficheros son estáticos.
   - Este proceso se puede realizar también con el comando `ng build` cuando queramos.

2. **Desplegar aplicación**: Es necesario desplegar los ficheros detrás de un servidor web. Se puede realizar de dos maneras:
    - Usar una **plataforma** (Firebase, GitHub Pages, Amazon S3, …) donde desplegar nuestra aplicación. Para ello se debe añadir la dependencia necesaria y usar el asistente `ng deploy` para desplegar la aplicación.
    - Usar un **servidor web propio**. Para ello debemos seguir las recomendaciones necesarias para configurar nuestro servidor.

Conceptos fundamentales

1. Componentes

Los **componentes** son la unidad básica de construcción en Angular. Cada parte visible de nuestra webapp (botones, tarjetas, secciones de una página, layouts…) puede representarse como un componente.

Información

Es conveniente pensar nuestra aplicación diferenciando componentes, de esta manera los podremos reutilizar en caso de que sea necesario.

1.1. Estructura de un componente

Un componente está formado por los siguientes ficheros:

- `.css` / `.scss`: Estilos propios del componente, para que de esta manera sea independiente a otros. Dependiendo del formato elegido al generar el proyecto, serán pre-procesados para generar un CSS válido.
- `.html`: Es la plantilla del componente, en formato HTML, donde se mostrará lo que queremos visualizar.
- `.spec.ts`: Fichero para realizar pruebas unitarias.
- `.ts`: Contiene la lógica y la clase del componente.

Al crear nuestra aplicación se ha generado el componente principal llamado **app-root** que está dentro del directorio `src/app/`.

En el fichero `src/app/app.ts` del proyecto tendremos:

Componente App

```
@Component({
  selector: 'app-root',
  imports: [RouterOutlet],
  templateUrl: './app.html',
  styleUrl: './app.scss'
})
```

Podemos destacar:

- ***selector***: es la manera que usaremos para posteriormente visualizar nuestro componente.
- ***imports***: posibles importaciones que tendrá nuestro componente. Más adelante hablaremos de ello.
- ***templateUrl***: fichero de plantilla HTML. En lugar de ser un fichero, podría ser código HTML directamente, pero para componentes grandes perderíamos facilidad de modificación.
- ***styleUrl***: fichero de estilos. De nuevo, se podrían poner aquí los estilos.

Ejercicio

Comprueba el contenido de los ficheros `app.html`, `app.scss`, `app.specs.ts` y `app.ts`. Modifica **app** para que en el HTML aparezca tu nombre en color rojo.

1.2. Vistas con plantillas

Angular usa un **sistema de enlace de datos (*data binding*)** que permite conectar la lógica con la vista, pudiendo mostrar variables en la vista o llamar a funciones de la clase desde la vista.

Desde **Angular 16**, para las variables que van a cambiar en el tiempo, se hace uso de signals para rastrear cómo y dónde se cambian los estados de una aplicación. De esta manera el *framework* optimiza las actualizaciones en el renderizado de las vistas.

Las `signals` pueden contener cualquier tipo de dato, que puede ser leído a través de la función *getter*, y pueden ser tanto de lectura-escritura o de sólo lectura.

1.2.1. Interpolación

De esta manera podemos mostrar de manera sencilla las propiedades de un componente en su vista (*template*).

Modificar app.ts

```ts
export class App {
  contador = signal(0);
}
```

Modificar app.html

```html
<p>{{contador()}}</p>
```

1.2.2. Propiedades y atributos dinámicos

Para poder modificar propiedades y atributos del DOM (*Document Object Model*) del *template* HTML, debemos especificarlo entre corchetes `[]`. En el siguiente ejemplo el botón va a aparecer como deshabilitado debido a que así se ha especificado desde la clase.

Modificar app.ts

```ts
export class App {
  contador = signal(0);
  public isDisabled = signal(false);
}
```

Modificar home.page.html

```html
<button [disabled]="isDisabled()">
  Pulsa
</button>
```

Pregunta

¿Qué sucede al poner `isDisabled` a `true`?

1.2.3. *Event listeners*

Para manejar interacción desde la vista, a través d un *event binding*, podemos llamar a un método de la clase desde la vista, de esta manera se permite la interacción del usuario.

Siguiendo con el ejemplo anterior, se va a crear otro botón que al pulsarlo modifique la propiedad de la clase que hacía que estuviese deshabilitado.

Modificar app.ts

```
incrementar() {
  this.contador.update((n) => n + 1);
}
```

Modificar home.page.html

```
<button [disabled]="isDisabled()"
        (click)="incrementar()">
  Pulsa
</button>
```

1.2.4. Control de datos

En las vistas también podremos realizar un control de los datos que se visualizan.

1.2.4.1. Condicionales

Para poder controlar qué se visualiza en la vista, al igual que sucede en el *backend*, se puede realizar un control condicional con `@if`,`@else` o `@else if`. También existe el bloque `@switch` para condicionales de muchas opciones.

Control en la vista

```
@if (isGod()) {
  <h2>God settings</h2>
  <!-- ... -->
} @else if (isAdmin()){
  <h2>Admin settings</h2>
  <!-- ... -->
} @else{
  <h2>User settings</h2>
  <!-- ... -->
}
```

Control en la vista

```
@switch (userPermissions) {
  @case ('admin') {
    <app-admin-dashboard />
  }
  @case ('reviewer') {
    <app-reviewer-dashboard />
  }
  @case ('editor') {
    <app-editor-dashboard />
  }
  @default {
    <app-viewer-dashboard />
  }
}
```

¡Cuidado!

En versiones antiguas de Angular existían `NgIf` y `NgFor`, pero están **obsoletas**.

1.2.4.2. Bucles

Para realizar repeticiones en la vista, como en el caso de una lista, podemos hacer uso del bucle `@for`. En la documentación nos muestra que en el bucle podemos hacer uso de variables implícitas que siempre están disponibles como:

- `\$count`: Número de items en la colección que estamos iterando.
- `\$index`: Índice actual de la lista (empieza en `0`).
- `\$first`: `True` si el elemento actual es el primero de la lista.
- `\$last`: `True` si el elemento actual es el último.
- `\$even`: `True` si el índice actual es impar.
- `\$odd`: `True` si el índice actual es par.

Modificar app.ts

```
frutas = ['Manzana', 'Pera',
          'Naranja', 'Banana',
          'Kiwi'];
```

Modificar app.html

```
@for (fruta of frutas; track fruta;
      let idx = $index) {
  <p class="{{$even ? 'even':''}}">
    {{fruta}} + {{idx}}
  </p>
}
```

La palabra reservada `track` sirve para generar una **clave única** para cada elemento de la colección. De esta manera cuando la colección cambie Angular tendrá un sistema eficiente para actualizar el DOM.

1.2.5. *Pipes*

Las *pipes* (o "tuberías") son un operador especial que se puede usar en las vistas para transformar datos de manera declarativa. Existen funciones ya predefinidas y podemos crear nuestras funciones. Este sistema está basado en los *pipes* de Unix y su carácter " | "".

Modificar app.ts

```
import { DatePipe,
    DecimalPipe,
    CurrencyPipe,
  } from '@angular/common';
```

Uso de pipes en la vista

```
<!--Jan 1, 2025 -->
<p>{{"2025-01-01" | date}}</p>
<!--1,234.0000 -->
<p>{{ 1234 | number:'1.4' }}</p>
<!--€1,234.00 -->
<p>{{ 1234 | currency:'EUR' }}</p>
```

¡Cuidado!

Faltan los `imports` en el `@Component`.

1.2.6. Variables

Podemos crear variables en la vista para poder ser usadas, y así quizá favorecer la sintaxis. Para poder declarar la variable se hará uso de `@let`.

Uso de variables en la vista

```
@let i = 1;
<p>{{i+i}}</p> <!--2 -->
```

1.3. Crear un componente

Para crear un componente nuevo vamos a hacer uso del CLI. Para ver todas las opciones se puede usar `ng generate --help`

Crear componente

```
ruben@vega:~/pruebas $ ng generate component prueba
CREATE src/app/prueba/prueba.scss (0 bytes)
CREATE src/app/prueba/prueba.spec.ts (528 bytes)
CREATE src/app/prueba/prueba.ts (186 bytes)
CREATE src/app/prueba/prueba.html (21 bytes)
```

Tal como se puede ver en la salida del comando, nos ha generado los cuatro ficheros dentro del directorio `src/app/prueba`. Cada componente es independiente y se mantiene en su directorio.

Información

Podemos crear componentes dentro de una estructura jerárquica de directorios propia, para ordenar los componentes por características: `ng generate componente user/profile`

1.4. Uso de componentes

Para visualizar un componente creado debemos añadirlo a la vista donde queramos que aparezca, pudiendo ser la vista principal o la de otro componente.

Para añadirlo, usaremos el nombre que le hayamos dado en el apartado ***selector*** al componente, y lo añadiremos como una etiqueta HTML más.

Añadir un componente a una vista

```
<app-prueba />
```

1.4.1. Recibir parámetros

Imaginemos que tenemos una aplicación que contiene varios listados, que por aspecto visual son iguales. Podemos generar un único componente que reciba la lista como parámetro y la visualice por pantalla. De esta manera, no duplicaremos código, y si queremos cambiar todos los listados, sólo debemos cambiar un único componente.

Generamos los datos en el componente padre y llamamos al componente hijo pasándole el listado como parámetro:

Modificar app.ts

```
export class App {
  listado = [1, 2, 3];
}
```

Modificar app.html

```
<app-prueba [lista]="listado"/>
```

Y ahora en el componente "hijo", debemos leer el parámetro mediante `input()`, indicando el tipo de datos que esperamos recibir, en este caso `number[]`.

Puede ser interesante inicializar el valor, de ahí que dentro del `input()` hayamos puesto `[3,2,1]`, por si no recibe el parámetro desde el padre.

Modificar prueba.ts

```
export class Prueba {
  lista = input<number[]>([3,2,1]);
}
```

Modificar prueba.html

```
@if (lista().length > 0) {
  @for (item of lista();track item){
    <p>{{item}}</p>
  }
} @else {
  <p>Sin elementos en la lista</p>
}
```

Ejercicio

Pasa la variable `contador` como parámetro y comprueba cómo se modifica en ambos componentes al pulsar el botón en App.

Si queremos que un **parámetro sea obligatorio**, lo haríamos con `input.required()`.

Información

La función `input` realmente hace uso de señales **de sólo lectura**.

1.4.2. Parámetros *model*

Los parámetros `model()` son *inputs* especiales que permiten propagar cambios de vuelta al componente padre. De esta manera, si desde el componente hijo necesitamos actualizar un valor del padre, al realizar el

cambio el padre recibirá el cambio.

Confirmamos en el componente padre que tenemos un valor que le pasamos al hijo. Desde el componente padre también podemos modificar ese valor con un botón:

Modificar app.ts

```
export class HomePage {
  contador = signal(0);
  incrementar() {
    this.contador.update(
      (n) => n + 1
    );
  }
}
```

Modificar app.html

```
<p>{{contador()}}</p>
<button [disabled]="isDisabled()"
        (click)="incrementar()">
    Pulsa
</button>
<app-prueba
    [lista]="listado"
    [(contador)]="contador"/>
```

Ahora desde el componente hijo vamos a recibir el valor mediante `model()`:

Modificar prueba.ts

```
export class Prueba {
  contador = model<number>(0);

  incrementar() {
    this.contador.update((n) => n+1);
  }
}
```

Modificar incrementar.page.html

```
<p>{{contador()}}</p>

<button (click)="incrementar()">
    Pulsa
</button>
```

1.5. Proyección de contenido con ng-content

Es posible que necesitemos crear componentes que sólo van a ser contenedores de contenido, en los que quizá no exista ninguna lógica aparte del aspecto visual. Para estos casos existe `ng-content`.

Desde el componente padre llamamos al componente hijo añadiendo contenido, y en el hijo simplemente ponemos `ng-content` para visualizarlo.

Llamada al componente hijo

```
<custom-card>
  <p>Contenido proyectado</p>
</custom-card>
```

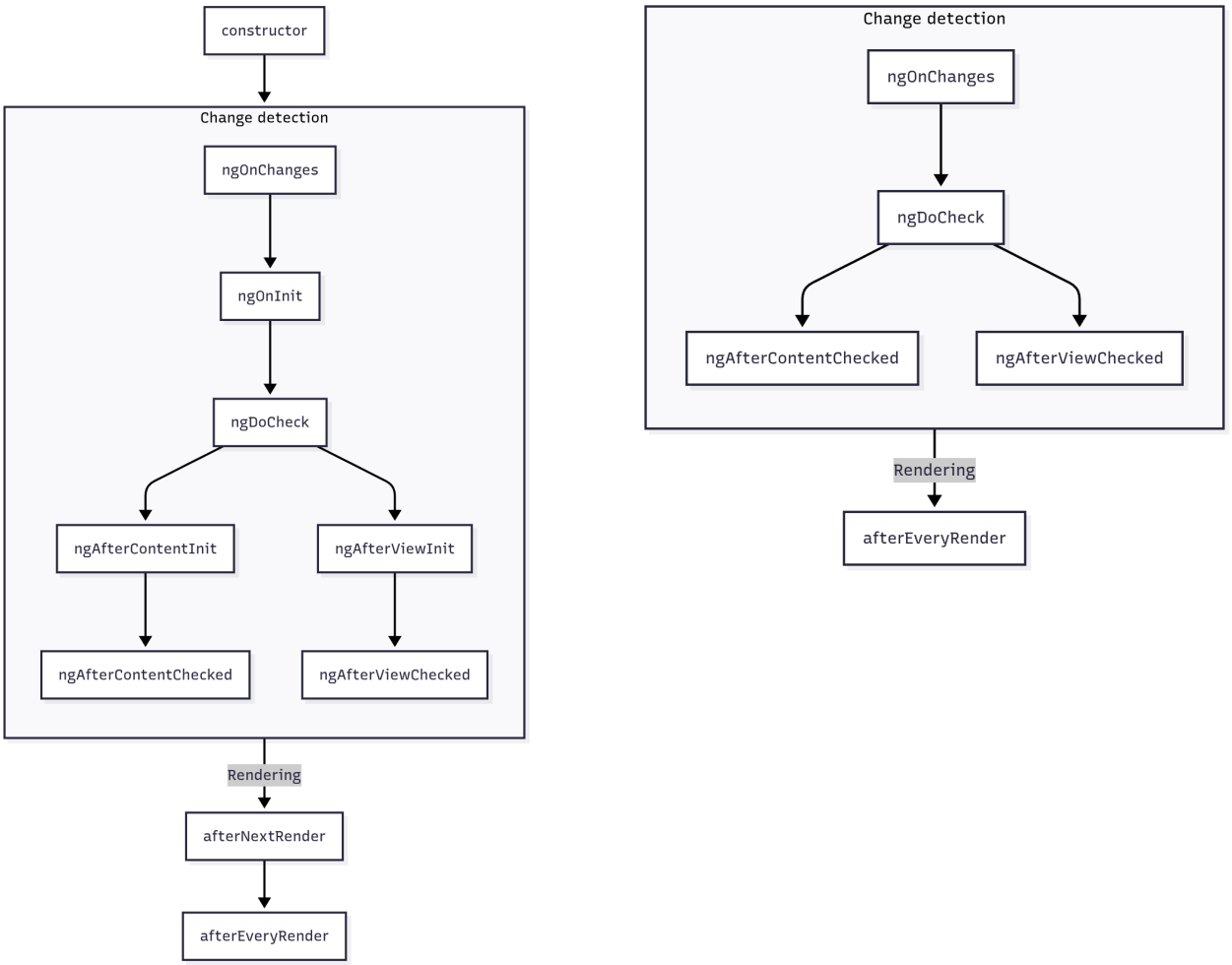
En el componente hijo

```
<div class="card-shadow">
  <ng-content />
</div>
```

1.6. Ciclo de vida de un componente

Cada componente Angular pasa por una serie de fases o *hooks* desde que se crea hasta que se destruye, lo que es conocido como el *lifecycle* del componente.

El ciclo de vida varía ligeramente al crearse el componente o durante sus actualizaciones posteriores. La explicación para todos los *hooks* se pueden encontrar en la documentación oficial.



Lifecycle de componente. Fuente: Angular.dev

Vamos a especificar las funciones más importantes que se ejecutan durante la creación/ejecución del componente. Tal como se puede ver en el dibujo previo, el ciclo de vida de los componentes varía en función de si es recién creado o si detecta alguna actualización:

Hook	Cuándo se ejecuta	Uso principal
`constructor`	Al crearse el componente	La función estándar de la clase TypeScript
`ngOnInit()`	Al inicializar el componente	Cargar datos o inicializar variables. Sólo se ejecuta una vez, antes de inicializar el *template*.
`ngOnChanges()`	Cuando cambian propiedades de entrada (`@Input`)	Reaccionar a cambios desde el componente padre.

Continúa en la siguiente página

`ngOnDestroy()`	Antes de eliminar el componente (ocultarse o navegar a otra página)	Limpiar recursos o detener suscripciones.

2. Rutas y navegación

Al navegar a una URL (del inglés *Uniform Resource Locator*) con nuestro navegador, éste realiza una petición de red al servidor web y el retorno suele ser una página HTML. Al hacer click sobre un enlace, el ciclo vuelve a empezar.

Angular permite crear aplicaciones de una sola página (SPA) y esto hace que el comportamiento citado sea diferente. En este caso, lo habitual suele ser realizar la petición al servidor web a la primera página, `index.html`, y el **enrutador de Angular**, desde el lado cliente, controla el cambio de URL.

El enrutamiento en Angular se compone de tres partes principales:

- Las rutas definen qué componente se muestra cuando un usuario visita una URL específica.
- Los ***outlets*** son marcadores de posición en las plantillas que cargan y renderizan componentes dinámicamente según la ruta activa.
- Los enlaces permiten a los usuarios navegar entre las diferentes rutas de la aplicación sin recargar la página completa.

2.1. Configuración básica del router

Para usar el sistema de rutas, debemos asegurar que `provideRouter` está en la configuración principal (`app.config.ts`) y se importa un conjunto de **rutas**, en este caso a través del fichero `app.routes.ts`.

Configuración de la aplicación

```
import { ApplicationConfig, provideBrowserGlobalErrorListeners,
         provideZoneChangeDetection } from '@angular/core';
import { provideRouter } from '@angular/router';
import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [
    provideBrowserGlobalErrorListeners(),
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes)
  ]
};
```

En el componente principal `app` deberíamos tener la configuración y en su template lo siguiente:

Componente principal

```
import { RouterOutlet }
  from '@angular/router';

@Component({
  imports: [RouterOutlet],
  // ...
})
```

Plantilla del componente

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

La directiva `<router-outlet/>` es un marcador de posición que indica la ubicación donde el enrutador debe renderizar el componente para la URL actual. Con los siguientes apartados se entenderá mejor

2.2. Ejemplo de rutas y navegación básica

Para entender cómo se crean las rutas necesarias de nuestra aplicación, vamos a suponer que nuestra aplicación cuenta con las siguientes URLs:

- `/`: página principal de la aplicación.
- `/home`: página principal del usuario.
- `/login`: página para poder loguearnos en la aplicación.

Para ello, en el fichero de rutas (`app.routes.ts`) añadiremos lo siguiente:

Ejemplo de rutas

```
import { Routes } from '@angular/router';
import { Home } from './home/home';
import { Login } from './user/login/login';

export const routes: Routes = [
    {
        path: 'home',
        component: Home,
        title: 'Home Page'
    },
    {
        path: 'login',
        component: Login,
        title: 'Login Page'
    }
];
```

El apartado `title` de la configuración es opcional y sirve para modificar el título de la página, para que así sea más accesible.

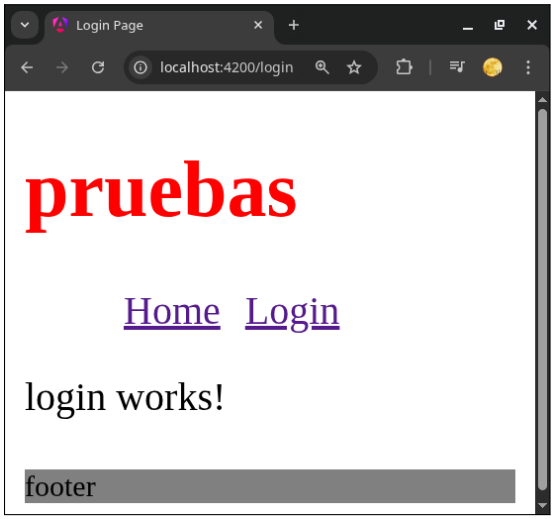
Ejercicio

Crea los dos componentes necesarios: `Home` y `Login` .

Para poder navegar entre las rutas creadas se va a usar `RouterLink`, por lo que en el template de nuestro componente principal vamos a crear unos enlaces de la siguiente manera:

Vista principal

```
<ul>
  <li>
    <a routerLink="/home">Home</a>
  </li>
  <li>
    <a routerLink="/login">Login</a>
  </li>
</ul>
<router-outlet />
<div id="footer">footer</div>
```



Y añadiendo un poco de CSS para pruebas, nuestra vista debería quedar tal como aparece en la imagen.

Información

Al navegar a `/login` el sistema de rutas carga el componente **Login** en la posición de `<router-outlet />`. Lo mismo al navegar a `/home`.

2.3. Definir parámetros en la URL

En las aplicaciones web actuales es habitual tener URLs parametrizadas, que son rutas dinámicas que posibilitan mostrar datos diferentes teniendo en cuenta los parámetros de la URL.

Vamos a hacer un ejemplo con URLs del estilo:

- `/user/:id/:social`: nos mostrará la red `:social` del usuario `:id`.
- `/user/:id`: donde `:id` es el parámetro y que nos mostrará la página del usuario concreto.

Debemos añadir las rutas para que llamen a los componentes correspondientes:

Nuevas rutas

```
export const routes: Routes = [
    {
        path: 'user/:id/:social',
        component: Social
    },
```

```
    {
        path: 'user/:id',
        component: UserProfile
    },
];
```

Y ahora en los componentes correspondientes se debe realizar la lectura de los parámetros para poder ser usados. Para ello se va a usar ActivatedRoute

Componente

```
import { ActivatedRoute } from '@angular/router';
// ...
export class UserProfile {
  readonly id: string | null;
  private route = inject(ActivatedRoute);

  constructor() {
    this.id = this.route.snapshot.paramMap.get('id');
  }
}
```

Ejercicio

Es momento de realizar lo mismo para el componente **Social**.

2.4. Ruta *Wildcard*

Para manejar URLs inexistentes, se debe usar la ruta comodín `**`, y **debe colocarse la última** dentro del array de rutas en el fichero `app.routes.ts`.

Rutas

```
export const routes: Routes = [
  //...
  {
      path: '**',
      component: NotFound
  },
];
```

2.5. Orden de las rutas

Aunque no se ha dicho explícitamente, **las rutas deben tener un orden concreto** para que el funcionamiento sea el esperado. El router de Angular evalúa las rutas **de arriba hacia abajo**, siguiendo el orden en que aparecen en el array de configuración.

Cuando el usuario navega a una URL, Angular busca la **primera ruta que coincide** con ese patrón, y deja de buscar. Es por eso que es importante **poner las rutas más específicas antes**.

Información

Las rutas deben seguir un orden: de más específicas a menos.

II

Comunicación cliente-servidor

1. Arquitectura cliente-servidor en aplicaciones web

Las mayoría de aplicaciones web *front-end* necesitan comunicarse con un servidor que ofrecen datos a través de APIs REST, normalmente a través del protocolo HTTP.

Separar nuestra aplicación diferenciando entorno cliente/*front end* y servidor/*back end* ofrece una serie de ventajas, entre las que podemos destacar:

- **Separación de código**: La ventaja más apreciable es la separación entre el código que afecta a la interfaz de usuario (ventanas/páginas, interacción, lógica de presentación…) y la lógica de negocio (con el acceso a la base de datos, autenticación, servicios externos, …).
- **Escalabilidad**: en caso de necesidad, se puede escalar el *back end* sin afectar a la aplicación, ya que el interfaz ya está preparado para realizar peticiones a un *backend*. El *frontend* también se puede cachear en un CDN para no cargar al servidor.
- **Desarrollo en paralelo**: Si el proyecto de la aplicación se diseña de manera adecuada, el desarrollo se puede realizar en paralelo, lo que suele acortar la duración del proyecto.
- **Separación en el despliegue**: Al realizarse el desarrollo en paralelo, el despliegue a los servidores de test/QA y producción se pueden realizar de manera separada, y adecuándose a los ciclos separados.
- **Ventajas en mantenimiento y evolución**: Similar al punto anterior, cada parte puede mantenerse y evolucionar sin afectar a la otra.
- **Reutilización del *backend***: Con el mismo *backend* podemos crear distintas aplicaciones *front end*: web, dispositivo móvil Android, iOS, smartTV…
- **Uso de diferentes tecnologías**: Al separar los desarrollos podemos hacer uso de la tecnología más adecuada para cada parte. Esto da una flexibilidad que en aplicaciones "monolíticas" no suele haber.
- **Posibilidad de reemplazo**: Si un día se decide realizar un cambio en la tecnología, al realizar el cambio en un apartado, no afecta al otro.
- **Especialización de perfiles**: A la hora de buscar profesionales también resulta útil, ya que podemos buscar perfiles muy especializados en las necesidades específicas de cada apartado.

2. Inyección de dependencias

La inyección de dependencias es un patrón de diseño para organizar y compartir código a lo largo de nuestra aplicación y componentes haciendo uso de "**servicios**". Las ventajas que nos ofrece son:

- **Mejora del mantenimiento del código**: Permite una separación más clara de responsabilidades, lo que facilita la refactorización y reduce la duplicación de código.
- **Escalabilidad**: La funcionalidad modular se puede reutilizar en múltiples contextos y permite una escalabilidad más sencilla.

- **Mejores pruebas**: Permite que las pruebas unitarias utilicen fácilmente dobles de prueba en situaciones donde no es práctico usar una implementación real.

Al igual que al definir un componente, los **servicios** se componen de lo siguiente:

- Un decorador de TypeScript que declara la clase como un servicio de Angular mediante `@Injectable` y permite definir qué parte de la aplicación puede acceder al servicio a través de la propiedad `providedIn` (que normalmente es `'root'`), permitiendo así el acceso al servicio desde cualquier punto de la aplicación.
- Una clase de TypeScript que define el código que estará disponible cuando se inyecte el servicio.

Entre los tipos de servicios más comunes se incluyen:

- **Clientes de datos**: Simplifican el proceso de realizar solicitudes a un servidor para la recuperación y modificación de datos.
- **Gestión de estado**: Definen el estado compartido entre varios componentes o páginas.
- **Autenticación y autorización**: Gestionan la autenticación de usuarios, el almacenamiento de tokens y el control de acceso.
- **Registro y gestión de errores**: Establecen una API común para registrar o comunicar los estados de error al usuario.
- **Gestión y distribución de eventos**: Gestionan eventos o notificaciones que no están asociados a un componente específico, o bien, distribuyen eventos y notificaciones a los componentes, siguiendo el patrón observador.
- **Funciones de utilidad**: Ofrecen funciones de utilidad reutilizables, como el formateo, la validación o los cálculos de datos.

2.1. Crear servicio

Para crear un servicio el CLI de Angular nos ofrece el siguiente comando:

Crear servicio

```
ruben@vega:~/pruebas $ ng generate service calculadora
```

Y nos ha creado el fichero `src/app/calculadora.ts` que contiene:

Servicio recién creado

```
import { Injectable } from '@angular/core';
@Injectable({
  providedIn: 'root'
})
export class Calculadora { }
```

Tal como se puede ver, tenemos:

- ***Import*** del *decorator* `@Injectable()` que es parte del *core* de Angular.
- Al usar `@Injectable({ providedIn: 'root' })` en el servicio:
  - Crea una única instancia (***singleton***) para toda tu aplicación.
  - La hace disponible en todas partes sin necesidad de configuración adicional.
  - Habilita la eliminación de código muerto (*tree-shaking*) para que el servicio solo se incluya en el paquete JavaScript generado si realmente se utiliza.

¡Atención!

`providedIn: 'root'` cubre la mayoría de los casos pero existen muchos escenarios posibles.

En la clase creada en el servicio se pueden crear las funciones que necesitemos.

Ejercicio

Crea una función “sumar” que reciba dos parámetros y devuelva el valor.

2.2. Inyectar servicio en un componente

Para inyectar nuestro servicio en un componente debemos hacer uso de `inject()` e importar el servicio.

Inyectar servicio

```
import { Component, inject } from '@angular/core';
import { Calculadora } from '../calculadora';
//...
export class Prueba {
  calculadora = inject(Calculadora);
  suma = this.calculadora.sumar(2, 3);
}
```

Tal como se puede ver, se ha hecho la inyección del servicio y se ha llamado a la función sumar que se ha creado en el servicio.

Información

Un servicio se puede inyectar en otro servicio: documentación.

3. Cómo usar una API REST

Para realizar las peticiones HTTP a una API (*Application Programming Interface*) necesitamos usar HTTPClient, que se proporciona a través de provideHttpClient. Lo habitual es configurar este *provider* en la configuración general de nuestra aplicación, `src/app/app.config.ts` y aseguramos que contiene:

Añadimos el nuevo provider

```
//...
import { provideHttpClient } from '@angular/common/http';
//...
export const appConfig: ApplicationConfig = {
  providers: [
    //...
    provideHttpClient(),
  ]
};
```

Ahora ya podemos inyectar `HTTPClient{}` dentro de nuestro componente, servicio u otra clase.

3.1. Crear servicio para API

Vamos a crear un nuevo servicio llamado **"Cliente"** para realizar las peticiones a la API en las que se han incluido dos funciones. Para el ejemplo se ha utilizado la API de eventos culturales de OpenData Euskadi:

Nuevo servicio

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

@Injectable({
  providedIn: 'root'
})
export class Cliente {
  private http = inject(HttpClient);
  private apiUrl = "https://api.euskadi.eus/culture/events/v1.0";
  data = null;
  page = 0;
  elements = 20;

  get_events(): Observable<any[]> {
    this.page = this.page + 1;
    return this.http.get<any>(this.apiUrl+
        '/events?_elements='+this.elements+
```

```
        '&_page='+this.page);
  }

  get_event(id:number): Observable<any[]> {
    this.page = this.page + 1;
    return this.http.get<any>(this.apiUrl+
        '/events/'+id);
  }
}
```

En el código anterior se ha hecho uso de any como tipo de objeto de la respuesta, pero lo ideal sería tener en cuenta el modelo de datos recibido y tenerlo en cuenta.

3.2. Usar servicio de API

Una vez tengamos nuestro servicio creado, es momento de hacer uso de él y ver que funciona.

Ejercicio

Crea una ruta y un componente para la URL `/events`.

En este nuevo componente vamos a hacer que al ir a la URL correspondiente se realice la llamada para obtener todos los eventos. La idea es que al cargar el componente, a través de la función `constructor()` se realiza la llamada a la función del servicio que obtiene todos los eventos, y también lo manda a la consola de log para asegurar que está funcionando.

Por otro lado, en la plantilla se recorre el objeto obtenido en formato JSON y mostramos a modo de ejemplo el nombre del evento. Para ello, modifica la clase y la plantilla:

Uso del servicio

```
export class Events {
  cliente = inject(Cliente);
  data: any = signal(null);

  constructor() {
    this.cliente.get_events()
      .subscribe((response) => {
        this.data.set(response);
        console.log(this.data());
    });
  }
}
```

Plantilla

```
@if (data()) {
  @for (event of data().items;
    track event.id;
    let idx = $index) {
    <h3>
      <a href="/events/{{event.id}}">
      {{event.nameEs}}
      </a>
    </h3>

    <hr/>
  }
}
```

Con esto ya tendríamos el listado de todos los eventos recogidos a través de una función que llama a una

API externa.

3.3. Uso de *state* para reutilizar datos

Dependiendo de cómo funcione la API a la que llamemos, es posible que hayamos obtenido los objetos completos, por lo que quizá no sea necesario volver a llamar a la API en ciertas situaciones.

Vamos a continuar con el ejemplo y visualizar un evento concreto. Al obtener el listado de eventos, pueden pasar dos cosas:

- **Obtener sólo el título del evento y su identificador único**: obtendríamos un listado de títulos e IDs, por lo que al querer visualizar un evento, tendríamos que volver a llamar a la API para obtener los datos de ese evento concreto.
- **Obtener eventos y datos completos del listado**: al obtener el listado de eventos, obtendríamos para cada evento todos los datos completos. **Este es el caso de OpenData Euskadi**, por lo que tenemos los datos de cada evento.

Pregunta

¿Pero qué pasa si cargamos directamente la URL `/events/:id`?

Estamos ante dos situaciones distintas:

1. Si venimos de la navegación de la aplicación, tenemos los datos del evento, por lo que podemos visualizarlos directamente.
2. Si venimos de la recarga de la web o de un acceso directo, no tenemos los datos, sólo tenemos el `:id`, por lo que hay que realizar la petición a la API.

Desde el componente **Events**, donde obtenemos todos, vamos a realizar un pequeño cambio para usar`router.navigate` y ***state*** para pasar el objeto.

Crear función

```
export class Events {
  //...
  goto_event(event: any) {
    this.router.navigate(
      ['/events', event.id],
      { state: { event } }
    );
  }
}
```

Plantilla

```
<li>
  <a (click)="goto_event(event)">
    +Info
  </a>
</li>
```

Ejercicio

Crea otra ruta `/events/:id` y el componente para visualizar un evento.

En el nuevo componente vamos a comprobar lo siguiente:

- Obtenemos la ruta actual y comprobamos si tenemos el ***state*** del evento.
  - Eso significaría que venimos del otro componente.
  - Asignamos a la variable local el ***state*** para visualizarlo en la plantilla.
- En caso contrario:
  - Obtenemos el `id` de la URL
  - Llamamos a la función del servicio de la API para obtener el evento correspondiente.

Componente Event

```
import { Component, inject, signal } from '@angular/core';
import { ActivatedRoute, Router } from '@angular/router';
import { Cliente } from '../cliente';
//...
export class Event {
  private router = inject(Router);
  private route = inject(ActivatedRoute);
  cliente = inject(Cliente);
  event = signal<any | null>(null);

  constructor() {
    const nav = this.router.currentNavigation();
    const stateEvent = nav?.extras?.state?.['event'];

    if (stateEvent) {
      this.event.set(stateEvent);
      console.log("Event from state:", stateEvent);
    } else {
      // cogemos el parámetro de la ruta
      const id = Number(this.route.snapshot.paramMap.get('id'));
      this.cliente.get_event(id).subscribe({
        next: (data) => this.event.set(data),
        error: (err) => console.error('Error cargando evento', err)
      });
    }
  }
}
```

3.4. Generar cliente API automáticamente

A la hora de crear una API habría que hacer uso de la especificación OpenAPI (**OAS**), para generar los archivos de interfaz que son legibles por máquinas para describir, producir, consumir y visualizar servicios web REST. Si la API que generamos sigue el estándar, generar un servicio que lo consuma puede ser automático, y lo mismo sucede al revés.

Si miramos la web de eventos culturales de OpenData Euskadi, que hemos usado previamente, veremos que tiene el logotipo **OAS**. El enlace donde podemos ver la especificación de la API es el siguiente: cultural-events.json. Partiendo de este fichero vamos a generar nuestro cliente automáticamente.

Tenemos que instalar el paquete OpenAPI Generator con nuestro gestor de paquetes **npm**:

Instalamos OpenAPI Generator

```
ruben@vega:~/pruebas $ npm install @openapitools/openapi-generator-cli -g
```

Ahora podemos generar en nuestro proyecto Angular el nuevo servicio que automáticamente generará las funciones para llamar a la API:

Instalamos OpenAPI Generator

```
ruben@vega:~/pruebas $ openapi-generator-cli generate \
 -i https://opendata.euskadi.eus/.../cultural-events.json \
 -g typescript-angular -o src/app/api
```

Al comando se le pasan los siguientes parámetros:

- `generate`: para generar el servicio.
- `-i`: se le indica el fichero o URL que contiene la especificación OpenAPI. En el ejemplo anterior se ha acortado la URL por motivos de espacio.
- `-g`: qué generador vamos a usar, en este caso [typescript-angular]
- `-o`: dónde vamos a guardar el resultado

Si ahora queremos hacer uso del nuevo servicio, en el componente **Events** vamos a cambiar todo lo que habíamos añadido del servicio **Cliente** (el antiguo servicio) para que aparezca el nuevo servicio:

Uso del servicio

```
import { EventsService } from '../api';

export class Events {
  //...
  private eventsService = inject(EventsService)

  constructor() {
    this.eventsService.findEvents().subscribe((response) => {
      this.data = response;
      console.log(this.data);
    });
  }
}
```

Como se puede apreciar, los cambios son menores, importar el nuevo servicio, inyectarlo y hacer uso de las nuevas funciones auto-generadas.

Información

Para conocer qué funciones tiene el servicio, se puede generar la documentación con `-g html2`.

IV

Introducción a Typescript

1. Introducción

TypeScript (TS) es un lenguaje de código abierto desarrollado por Microsoft que se basa en **JavaScript** (realmente es un superconjunto de este), añadiendo **tipado estático** y características propias de lenguajes orientados a objetos como clases, interfaces o módulos.

El código que generamos se **transpila a JavaScript** (el código fuente se convierte a código JavaScript), por lo que puede ejecutarse en cualquier navegador o entorno que soporte JS.

Hoy en día se usa para crear aplicación tanto en entorno cliente como para entorno servidor (para React.js, Node.js…). Desde la versión 2 de Angular está desarrollado en TypeScript, ya que la primera versión (llamada AngularJS), se desarrollaba con JavaScript.

Hay una amplia documentación del lenguaje y en la web del W3School también existe un tutorial completo. A continuación sólo se van a explicar ciertos conceptos básicos para tener una guía de referencia básica.

1.1. Ventajas de usar TypeScript

Entre las ventajas que nos podemos encontrar al usar TypeScript, se pueden destacar las siguientes:

- **Detección de errores en tiempo de desarrollo** gracias al tipado.
- **Código más legible y mantenible**.
- **Autocompletado y ayuda contextual** en el editor (por ejemplo, VS Code).
- **Compatibilidad total con JavaScript**: cualquier código JS válido también lo es en TS.

2. Declaración de variables

Para declarar variables existen diferentes maneras en TypeScript:

- `var` o sin asignación: **no recomendado**
  - Es la forma tradicional de declarar variables en JavaScript.
  - Tiene **ámbito de función**, no de bloque.
  - Puede **redefinirse y reasignarse** libremente.
  - Puede provocar errores difíciles de detectar.
- `let`:
  - Introducido con **ES6** (2015).
  - Tiene **ámbito de bloque**: solo existe dentro del bloque donde se define.
  - Puede **reasignarse**, pero **no re-declararse** en el mismo bloque.
  - Es la opción más usada para variables que cambian de valor.

- `const`
  - También introducido con **ES6**.
  - Tiene **ámbito de bloque**, igual que `let`.
  - **No puede reasignarse** (su valor no puede cambiar).
  - Ideal para valores que deben permanecer constantes.

	var	let	const
Ámbito (scope)	Función	Bloque	Bloque
Re-declaración permitida	Sí	**No**	**No**
Re-asignación permitida	Sí	Sí	No
Uso recomendado	**Evitarlo**	Para variables que cambian	Para valores fijos o referencias

En el siguiente apartado veremos ejemplos de cómo se declaran las variables.

3. Tipos de datos

En la documentación oficial se explican todos los tipos de datos que se pueden utilizar en TypeScript. A continuación se van a detallar algunos de ellos

3.1. Tipos simples

TypeScript cuenta con tres tipos de datos simples:

- **`string`**: para representar cadenas de texto
- **`number`**: para tipos numéricos. No existe distinción como en otros lenguajes para *int* o *float*, aunque sí existe para **bigint**. También soporta números en binario, octal y hexadecimal:
- **`boolean`**: para tipos que sólo pueden ser *true* o *false*.

Existen otros tipos de datos especiales:

- **`any`**: desactiva el control de tipos. Se recomienda evitarlo salvo en casos especiales (uso de código que no se ha escrito en TypeScript, o librerías de terceros).
- **`unknown`**: es la contrapartida a *any*, ya que aunque puede ser de cualquier tipo, mantiene el tipado seguro.

- **undefined**: se puede asignar a cualquier tipo, pero es un tipo propio.
- **null**: igual que el anterior.

Ejemplos de declaración de variables y tipos

```
const nombre: string = "Rubén";
const pi: number = 3.1415;
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
let big: bigint = 100n;
let activado: boolean = true;
```

3.2. Tipos complejos

Existen otros tipos de datos, también habituales en otros lenguajes de programación.

3.2.1. Arrays

Como en otros lenguajes, podemos crear *arrays* del **mismo tipo de datos**. A la hora de declararlo, en cambio, se puede hacer de tres maneras:

Declarar un array

```
// se infiere que el tipo es number[]
let prueba = [1, 2, 3];
// define un array con elementos de tipo number
let prueba2: number[] = [1,2, 3];
// usa la clase Array, indicando que contendrá datos number
let prueba3: Array<number> = [1, 2, 3];
```

3.2.2. Tuplas

Una ***tuple*** (o tupla) es un tipo de array con una largura y un tipo de datos para cada posición predefinida. Se pueden nombrar las posiciones, para ayudar a la comprensión, pero no afecta al funcionamiento

Declarar una tuple

```
// definir la tupla
let ejemplo: [number, boolean, string];
// inicializar con datos correctos
ejemplo = [5, false, 'Mi texto'];
let direccion: [calle: string, n: number];
```

3.2.3. Objetos propios: *type* e *interface*

Al igual que pasa en JavaScript, podemos crear objetos propios, siendo una colección de parejas clave-valor. La ventaja es que podremos especificar el tipo de datos para cada propiedad del objeto. Otra característica de TypeScript es que podemos crear objetos con propiedades opcionales

Crear un objeto propio

```
const coche: { marca: string, year?: number } = {
  marca: "Toyota"
};
car.year = 2000;
```

La propiedad *year* es opcional porque le hemos puesto el símbolo de interrogante `?`. Al ser opcional, durante la asignación inicial podemos no asignarle valor.

Para poder reutilizar un tipo de objeto, podemos usar `interface`, lo que también nos permite crear nuevos objetos extendiendo otros ya creados.

Crear y extender un objeto

```
interface Rectangle {
  height: number,
  width: number
}
interface ColoredRectangle extends Rectangle {
  color: string
}
const coloredRectangle: ColoredRectangle = {
  height: 20,
  width: 10,
  color: "red"
};
```

Con `type` podemos crear tipos personalizados, que pueden tener un valor prefijado.

Crear y extender un objeto

```
type Rol = "admin" | "usuario" | "invitado";
let rolActual: Rol = "usuario";
```

La palabra reservada `type` también puede utilizarse para crear otros tipos de objetos. Se recomienda leer los siguientes enlaces para poder ver todas las posibilidades:

- w3cshcools: explica las diferencias entre `type` e `interface`.

- *cheatsheets* explicando las posibilidades de `type` e `interface`.

¡Atención!

Los objetos declarados con `type` e `interface` tienen ciertas diferencias, explicadas en la documentación.

3.2.4. Enum

Los enum son un tipo de clase especial que representa un grupo de constantes, y que no se pueden modificar. Pueden ser de tipo *string* o numéricos.

Los *enum* numéricos los podemos inicializar o pueden inicializarse solos empezando por 0.

Enum inicializado

```
enum Direction {
  Up = 1,
  Down,
  Left,
  Right,
}
```

Sin inicializar

```
enum Direction {
  Up,
  Down,
  Left,
  Right,
}
```

Enum de texto

```
enum Direction {
  Up = "UP",
  Down = "DOWN",
}
```

El resto de constantes tendrán un valor auto-incremental partiendo del valor inicial (sea 0, o el que hayamos puesto). En el caso de crear un *enum* de tipo *string*, cada miembro debe ser inicializado con un texto.

4. Estructuras básicas

A continuación se van a explicar las estructuras básicas, que son iguales que en JavaScript.

4.1. Estructuras condicionales

A continuación las estructuras condicionales más habituales. Como en otros lenguajes de programación, se pueden hacer uso de los operadores lógicos **&&** (AND), **||** (OR) y **!** (NOT). También existe la posibilidad de utilizar el **operador ternario**: `condition ? siTrue : siFalse`.

A la hora de manejar una gran cantidad de condiciones, en lugar de usar `if-else`, lo ideal es hacer uso del sistema `switch-case`.

También tenemos las palabras reservadas `break` para salir de la función y `default` que se ejecutará cuando no coincide con ninguno de los casos expresados.

Condicional if

```
let num: number = 20;
if (num > 18) {
  console.log("Mayor");
}
```

Condicional if-else

```
let num: number = 10;
if (num > 18) {
  console.log("Mayor");
} else {
```

```
  console.log("Menor");
}
```

Condicional else if

```
let num: number = 20;
if (num < 18) {
  console.log("Pequeño");
} else if (num > 65) {
  console.log("Muy mayor");
}
```

Estructura switch-case

```
let num: number = 18;
switch(num){
  case 7:
    console.log("Siete");
    break;
  default:
    return "Otro numero";
}
```

4.2. Bucles

Existen distintos tipos de bucles. El bucle `for` es utilizado cuando conocemos previamente el número de veces que queremos itera, y también se puede utilizar para recorrer un array.

Dentro de los bucles, y utilizando una condicional, podemos usar `continue` si queremos omitir parte del bucle y saltar a la siguiente iteración.

Bucle for

```
for (let i = 0; i < 5; i++) {
  console.log(`Contador ${i}`);
}
```

Bucle for y array

```
const frutas: string[] = ["Manzana", "Pera"];
for (let i = 0; i < frutas.length; i++) {
  console.log(`Fruta ${i + 1}: ${frutas[i]}`);
}
```

Bucle for…of y array

```
const frutas: string[] =
      ["Manzana", "Pera"];
for (const f of frutas) {
  console.log(`Fruta: ${f}`);
}
```

Bucle for…in

```
const p = {
  name: "Ruben",
  job: "Teacher"
};
for (const key in p) {
  console.log(`${key}:
      ${p[key as keyof typeof p]}`
  );
}
```

Existe la variante `for…of` que nos permite iterar sobre los elementos de colecciones, como son los Arrays (`[]`), Strings (`“”`), Mapas (`Map`) y Conjuntos (`Set`). Y para poder iterar entre las propiedades de un objeto existe `for…in`, de esta manera tendremos acceso a sus posibles *key* y *values*.

También podemos hacer uso de bucles `while` y `do-while`.

Bucle while

```
while (condición) {
  // Código
}
```

Bucle do-while

```
do {
  // Código
} while (condición);
```

5. Funciones y parámetros

A la hora de crear funciones debemos especificar los parámetros y los tipos de objetos que tienen. Los parámetros pueden ser:

- **Tener valores prefijados**: se puede especificar el valor por defecto de un parámetro, por lo que se puede llamar a la función sin especificarlo.
- **Opcionales**: los parámetros pueden ser opcionales, si se especifica con el símbolo de interrogante `?`.

En caso de que las funciones devuelvan un dato, también hay que especificar el tipo de datos que va a devolver. En caso de que no devuelva ningún tipo de datos, se usará `void`.

Enum de texto

```
function sumar(a: number, b: number = 10): number {
  return a + b;
}
function saludar(nombre: string, mensaje?: string): void {
  console.log(mensaje ? `${mensaje}, ${nombre}` : `Hola, ${nombre}`);
}
console.log(sumar(2));     // 12
console.log(sumar(3, 5)); // 8
saludar("Ana");            // Hola, Ana
saludar("Bob", "Buenas"); // Buenas, Bob
```

6. Clases y métodos

TypeScript permite usar clases, constructores y métodos, como en otros lenguajes de programación orientado a objetos tradicionales. La sintaxis de una clase, los objetos y los métodos se realiza de manera similar a otros lenguajes.

Ejemplo de clase

```
class Persona {
  private nombre: string;
  private edad: number;
```

```
  public constructor(nombre: string, edad: number) {
    this.nombre = nombre;
    this.edad = edad;
  }

  saludar() {
    console.log(`Hola, soy ${this.nombre} y tengo ${this.edad} años.`);
  }
}

const persona1 = new Persona("Bob", 30);
persona1.saludar();
```

Los atributos y los métodos de una clase pueden tener distinta "**visibilidad**":

- **public**: El valor por defecto. Se permite el acceso desde cualquier lugar.
- **private**: Permite el acceso a los miembros de la clase sólo desde la propia clase.
- **protected**: Permite el acceso al miembro de la clase desde sí mismo y desde cualquier clase que lo herede.

6.1. Herencia

Como en otros lenguajes de programación orientada a objetos, también se permite la herencia de una clase "padre". Se permite expandir la funcionalidad con:

- `implements`: en la documentación se explica su uso, que sirve para comprobar que se satisface una `interface` concreta.
- `extends`: para extender una clase padre. La nueva clase contiene todos los atributos y métodos de la clase original, pudiendo añadir nuevos miembros.

7. Módulos e importaciones

TypeScript organiza el código en **módulos**. Cada archivo puede **exportar** variables, funciones o clases, e **importarlas** en otros archivos.

Ejemplo de clase exportada

```
// archivo: operaciones.ts
export function multiplicar(a: number, b: number): number {
  return a * b;
}
```

```
// archivo: main.ts
import { multiplicar } from './operaciones';

console.log(multiplicar(4, 5)); // 20
```

De esta manera es cómo funciona Angular para importar componentes, servicios o módulos.