Introducción a Angular

UD2: Desarrollo de aplicaciones basado en componentes con Angular

Objetivos de aprendizaje

- Aprender a crear una nueva aplicación Angular
- Comprender las principales partes que tenemos en una aplicación Angular
- Organizar una aplicación y crear componentes personalizados
- Entender las distintas formas de comunicación entre componentes y plantillas y las directivas básicas

Indice

Introducción a Angular

UD2: Desarrollo de aplicaciones basado en componentes con Angular

Indice

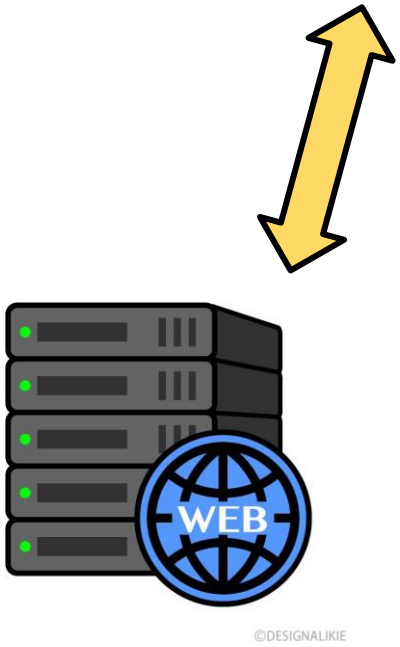
Recordemos

- Al desarrollar en Web dividimos nuestro código en dos partes:
  - Parte servidora
    - Código que se ejecuta en un equipo remoto (servidor).
    - Funciones principales es el manejo de datos (acceso a bases de datos), autentificación, seguridad…
    - El usuario no tiene acceso al código de esta parte
  - Parte cliente
    - Código que se ejecuta en mi propio dispositivo (navegador Web)
- El diálogo entre cliente y servidor se establece a través del protocolo HTTP

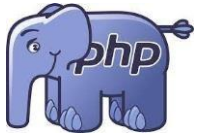
Modelo Web

PETICIÓN HTTP

RESPUESTA HTTP



Tecnologías Web

Lado Cliente 	
Lado Servidor 	       

Frameworks

- Un framework es un conjunto de utilidades probadas y validadas que me permiten solucionar problemas que me encuentro de manera común al realizar una aplicación.
  - Organización de archivos, separación de funcionalidad, manejo de rutas, seguridad, acceso a datos, manejo de usuarios….
- Cada plataforma de desarrollo suele tener su propio framework asociado.

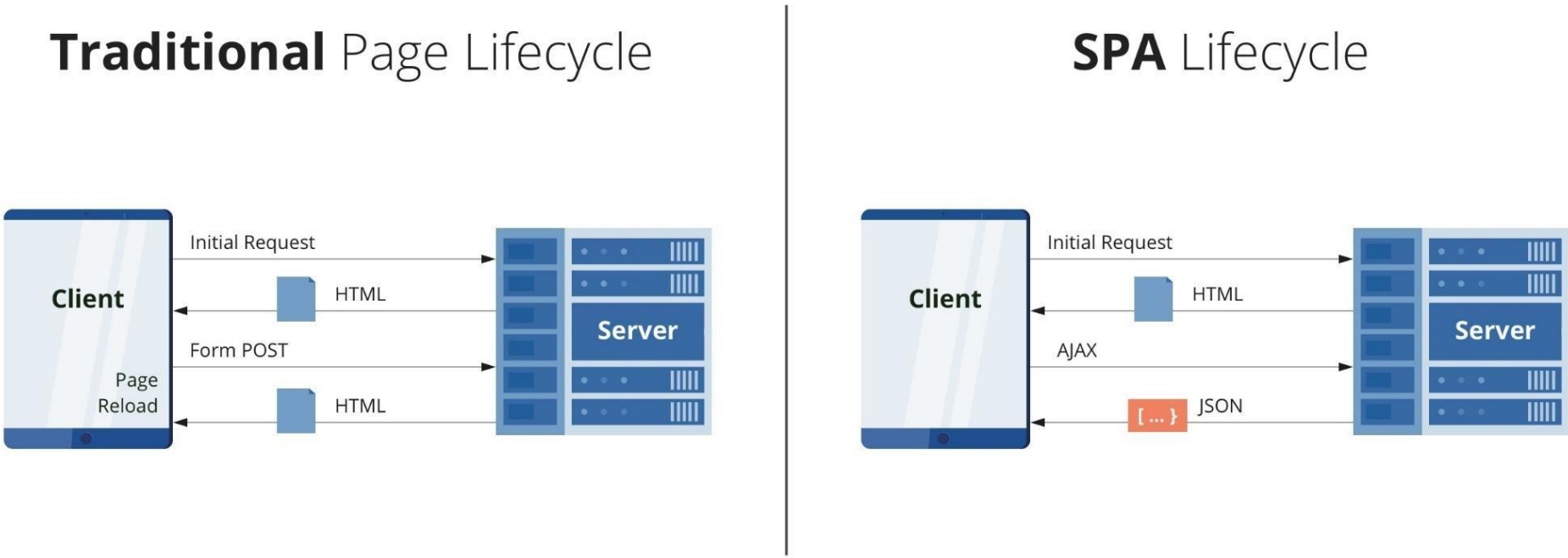
Frameworks JS

- Actualmente está muy extendido el uso de 3 frameworks JS
- Son similares pero cada uno tiene sus particularidades y curva de aprendizaje

¿Qué es una SPA?

- SPA = Single Page Application
- Aplicación Web donde todo el contenido se carga en la primera petición
  - Primera carga más lenta pero mejora la experiencia de usuario durante la navegación

SPA vs MPA


Traditional Page Lifecycle
Client
Page Reload
Initial Request
HTML
Form POST
HTML
Server
SPA Lifecycle
Client
Initial Request
HTML
AJAX
[... }
JSON
Server

Angular

- Framework para aplicaciones Web SPA
- Código abierto
- Mantenido por Google
- Lanzado en 2016 (Angular 2)
- Ultima versión estable: 20.3.12 (noviembre 2025)

EN ESTA PARTE DEL CURSO VAMOS A USAR LA VERSIÓN 20

Últimas versiones

	Angular 19	Angular 20
Fecha de lanzamiento	Noviembre 2024	Mayo 2025
Principales novedades	• Standalone components/directives/pipes por defecto • Hydratación incremental (SSR) • Nuevas APIs de señales (“linkedSignal”, etc)	• API de Signals ahora estable • Modo “zoneless” de detección de cambios en preview • Mejoras de rendimiento, bundle más pequeños, SSR mejorado

¿Por qué la versión 20?

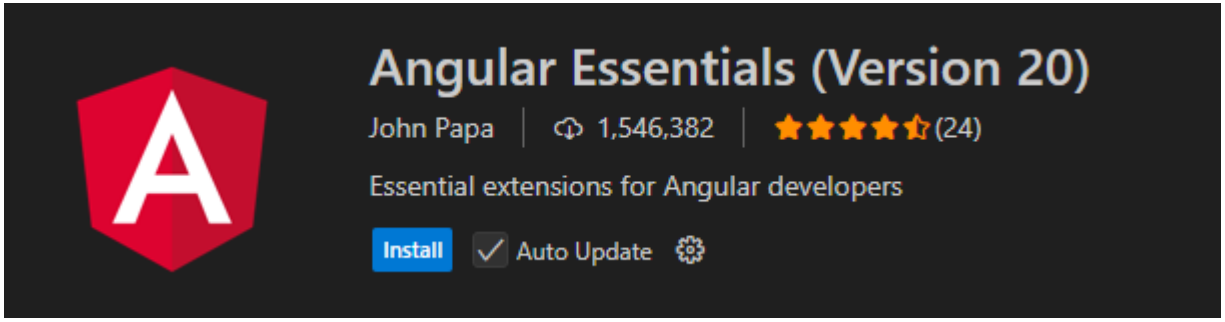
- Introduce el modelo moderno basado en *standalone components*.
- Simplifica la estructura del proyecto
  - Elimina la necesidad de módulos
  - Permite un desarrollo más rápido, limpio y fácil de mantener.
- Versión recomendada por el propio equipo de Angular

Características de Angular

- Velocidad y rendimiento
  - Optimiza el código
  - Multiplataforma
  - Carga solo lo que necesitamos
- Productividad
  - Creamos elementos de manera rápida
  - Angular CLI
  - Integración con la mayoría de editores de texto

Preparando el entorno

- Conjunto de Extensiones “Angular Essentials” de VSCode



Preparando el entorno

- Otras extensiones:
  - Angular Language Service
  - Angular Snippets
  - Angular Schematics
  - Angular 2 Inline
  - Auto Close Tag
  - Auto import
  - Auto Rename Tag
  - Error Lens
  - Paste JSON as Code
  - TypeScript Importer - optional
  - Editor Config for VSCode
  - Better Comments
  - Tailwind CSS IntelliSense
  - Pretty TypeScript Errors

Preparando el entorno

- Extensión Angular Dev Tools de Google Chrome

Angular CLI

- Es una herramienta en línea de comandos para trabajar con Angular
- Documentación
- Instalación (global):

```
npm install -g @angular/cli@20
```

- Comprobación:

```
ng version
```

Creación de un proyecto Angular

```
ng new NombreProyecto
```

- Instala paquetes y dependencias necesarios para nuestro proyecto
- Se nos hacen algunas preguntas para comenzar

Pregunta	Escogemos
Tipo de estilo que queremos añadir	**SCSS**
(SSR) (SSG/Prerendering)	**No**
zoneless	**No**
IA	**No**

Inspeccionando el código

Archivo/carpeta	Explicación
`package.json`	Gestión de las dependencias de proyecto. Debería incluir las librerías de Angular y otras dependencias
`node_modules`	Carpeta con las dependencias
`public/`	Carpeta para **archivos estáticos** accesibles directamente por el navegador. (Imágenes, Iconos, pdf…)
`src`	La carpeta **más importante del proyecto**. Contiene **todo el código fuente** que programas tú.
`main.ts`	punto de arranque (con bootstrapApplication)
`editconfig`	Configura reglas comunes para todos los editores: • Indentación • Uso de espacios/tabuladores • Codificación UTF-8

`angular.json`	Configuración de la aplicación Angular

`tsconfig.json`
`tsconfig.app.json`
`tsconfig.spec.json`
	Configuración de TypeScript

Inspeccionando el código (II)

Archivo/carpeta	Explicación
`src`	Carpeta donde se encuentra el código de nuestra aplicación
`src/index.html`	Base de nuestra aplicación
`src/main.ts`	Arranque de nuestra aplicación (punto de salida)
`styles.css`	Estilos a nivel de aplicación
`assets`	Documentación, archivos estáticos, imágenes, archivos de JS…

Lanzando el proyecto

```
ng serve
```

- Hace una precompilación de nuestro proyecto
- Lo ejecuta en un servidor local bajo Node.js
- Se indica la dirección por defecto donde está levantado
- Opciones:

```
ng serve --port puerto
```

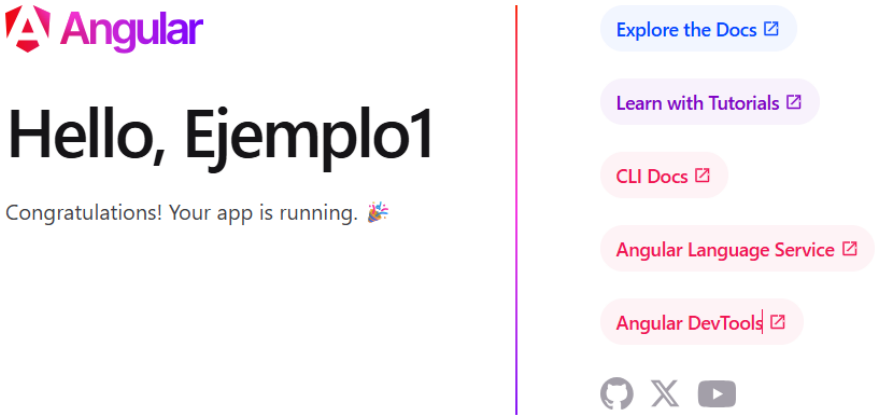
Puerto a medida

```
ng serve --open
```

Abre en el navegador

Lanzando el proyecto

- Página predeterminada en http://localhost:4200



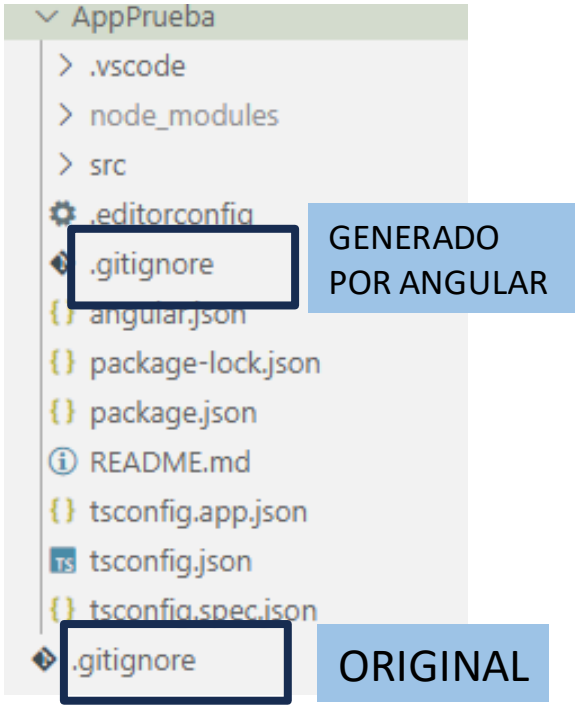
npm y GitHub

- Nuestro repositorio git debe tener añadida la carpeta `node_modules` en `.gitignore`
- Cuando subamos nuestro proyecto al repositorio de GitHub, se subirá el archivo package.json que indica las dependencias pero no las propias dependencias
- Cuando recuperemos el proyecto desde GitHub, tenemos que descargar de nuevo las dependencias con el comando

```
npm install
```

Angular y Git

- Cuando creamos un proyecto Angular, si no estamos dentro de un repositorio Git se pre-inicializa uno.
- Se genera un archivo `.gitignore` con las rutas que se deberían ignorar cuando hacemos commit
- Teóricamente deberíamos tener un solo .gitignore pero git admite varios anidados



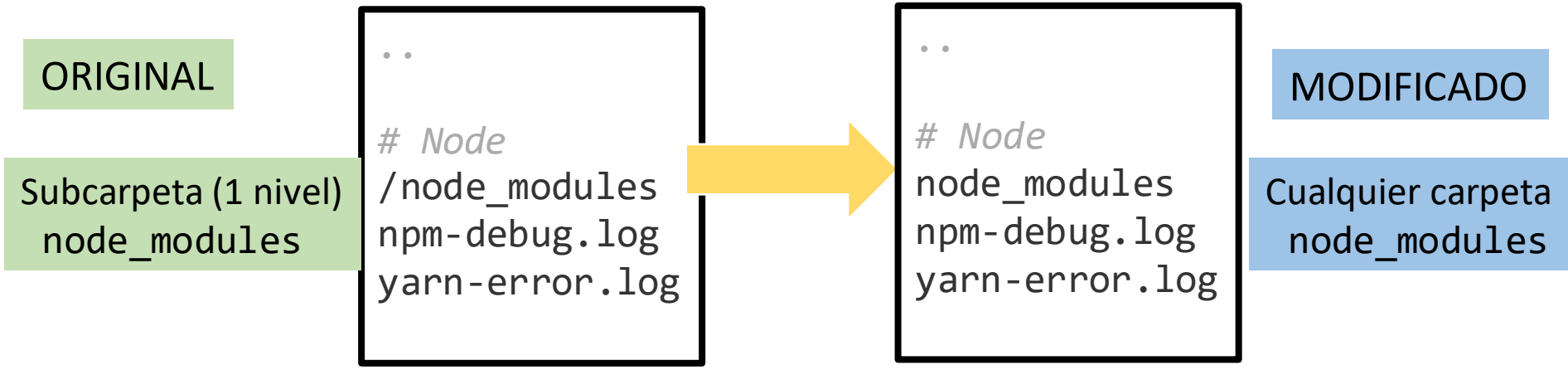
Contenido de `.gitignore`

```
# Compiled output
/dist
/tmp
/out-tsc
/bazel-out

# Node
/node_modules
npm-debug.log
yarn-error.log

# IDEs and editors
.idea/
.project
.classpath
.c9/
*.launch
.settings/
*.sublime-workspace
```

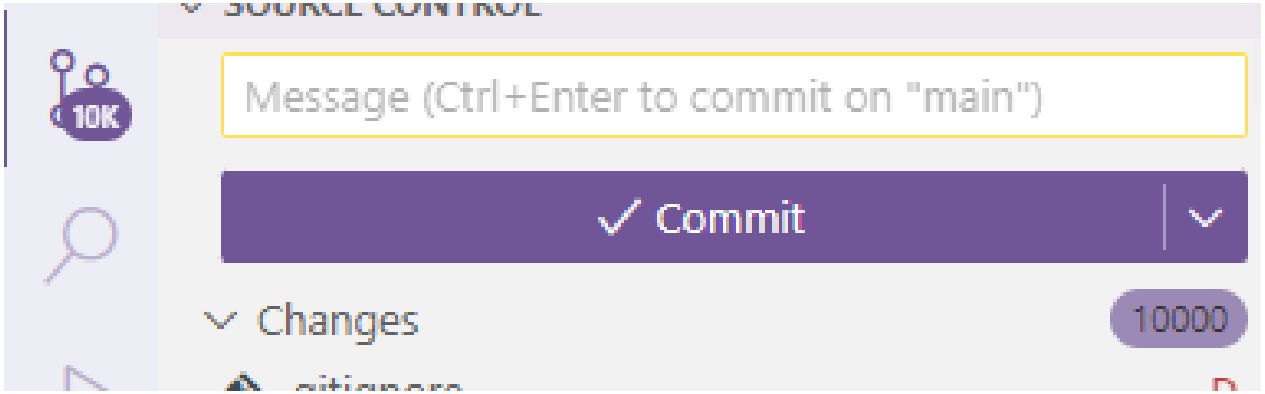
```
# Visual Studio Code
.vscode/*
!.vscode/settings.json
!.vscode/tasks.json
!.vscode/launch.json
!.vscode/extensions.json
.history/*

# Miscellaneous
/.angular/cache
.sass-cache/
/connect.lock
/coverage
/libpeerconnection.log
testem.log
/typings

# System files
.DS_Store
Thumbs.db
```

Angular y Git (II)

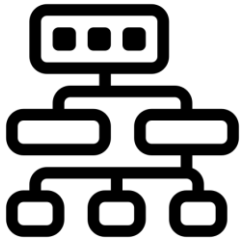
- También podemos copiar el contenido a nuestro `.gitignore`, **adaptando las rutas** y luego borrar el original



Angular y Git



INDICATIVO DE QUE ESTOY HACIENDO ALGO MAL




Arranque de una app Angular

UD2: Desarrollo de aplicaciones basado en componentes con Angular

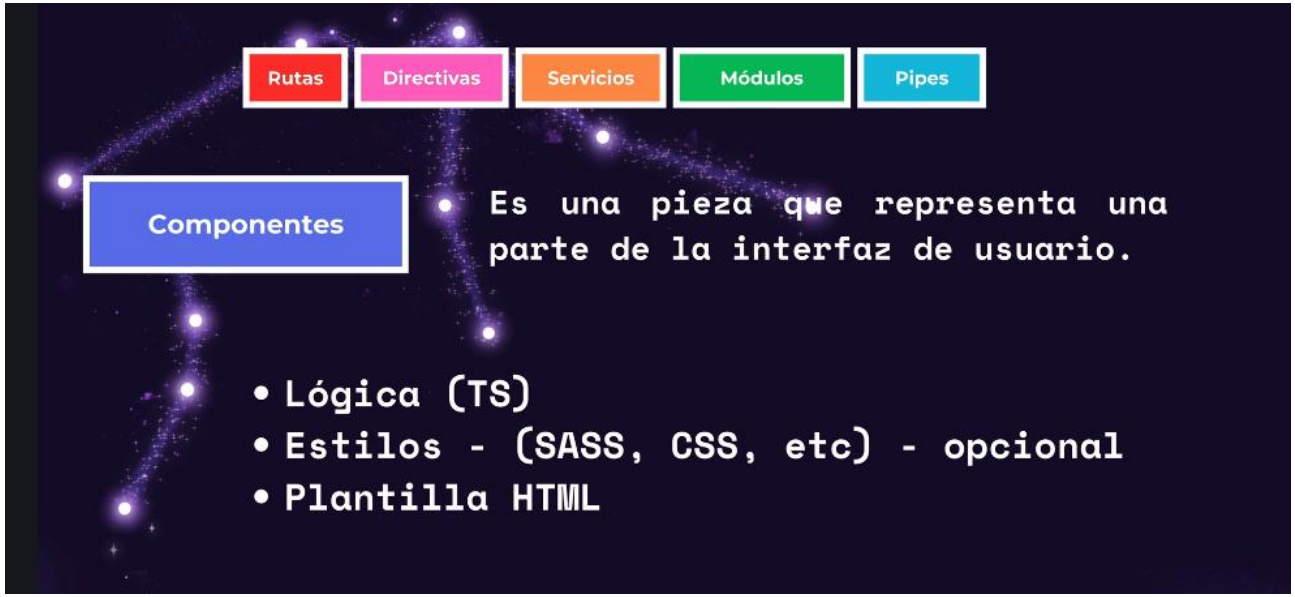
Indice

Estructura básica de un proyecto Angular

- Angular 20 la estructura se organiza principalmente mediante **componentes standalone.**
- Cada **componente standalone**:
  - Representa una unidad funcional de la aplicación.
  - Puede importar directamente otros componentes, servicios, directivas, pipes, formularios, etc.
- La aplicación ya **no necesita un módulo raíz**.
  - En su lugar, se inicia con bootstrapApplication() cargando el componente principal de la app.

Bloques fundamentales

Bloques fundamentales. Componentes

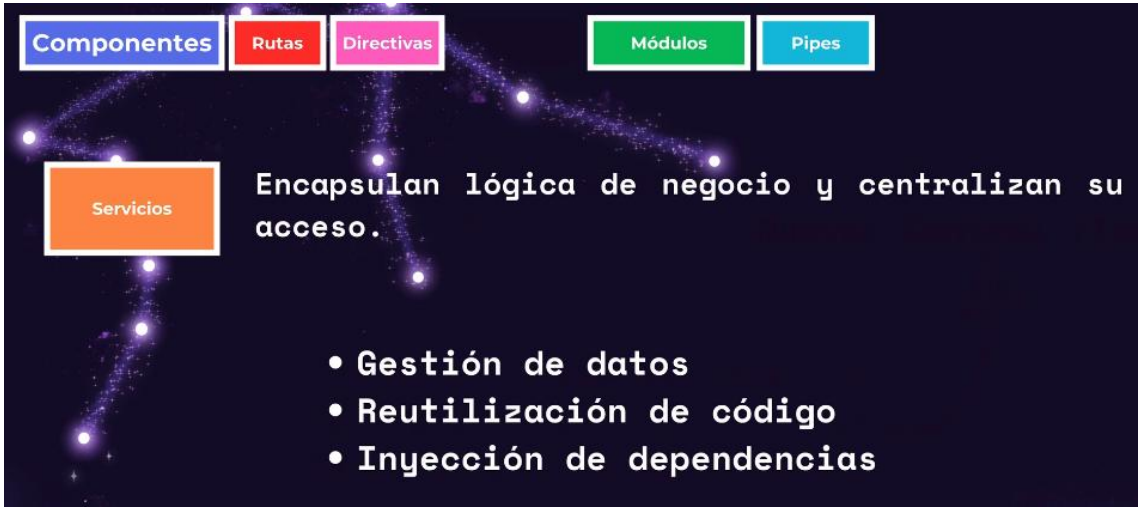


Bloques fundamentales. Rutas

Bloques fundamentales. Directivas

- Relacionados con atributos, cambios de clases, cambios de estilo…

Bloques fundamentales. Servicios

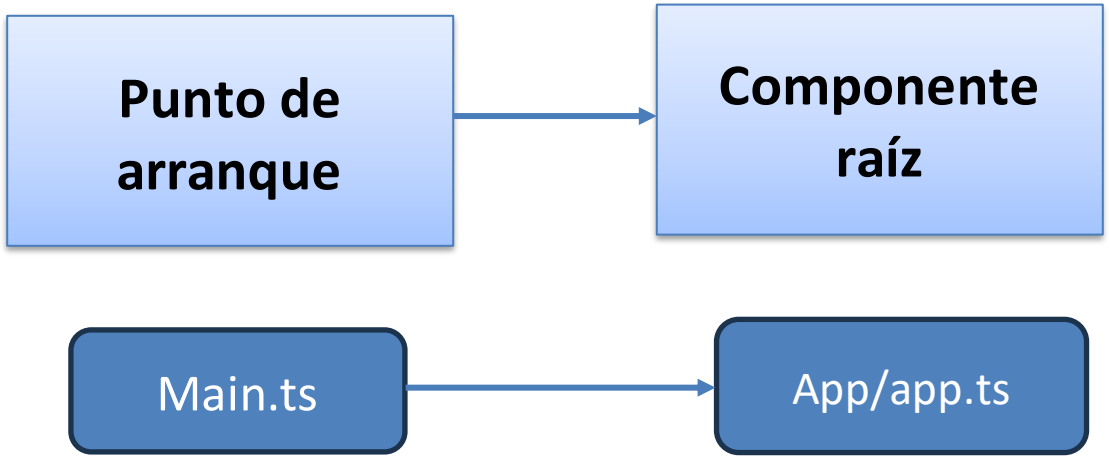


Bloques fundamentales. Módulos

Bloques fundamentales. Módulos

- Por ejemplo si tienes un número y quieres que aparezca en moneda, pues le pasas un pipe.

Arranque de nuestra aplicación



Arranque de nuestra aplicación

```
import { bootstrapApplication } from ' @angular/platform-browser';
import { appConfig } from './app/app.config';
import { App } from './app/app';

bootstrapApplication(App, appConfig)
  .catch((err) => console.error(err));
```

Main.ts

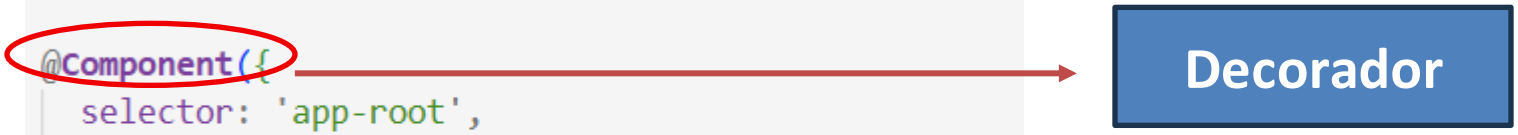
- Este es un archivo que nos va a servir para decirle a Angular cómo queremos que corra la aplicación.

- El componente raíz es App y tiene un appConfig, que es un archivo de configuración y nos permite definir cómo corre la aplicación.

App.ts

- No es más que una clase con un decorador que transforma mi clase a un componente.

```ts
import { Component, signal } from '@angular/core';
import { RouterOutlet } from '@angular/router';

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



Decoradores

- Si tenemos un decorador aplicado sobre una clase le estamos dando una funcionalidad distinta
- Uso:
  - Antes de definir la clase mencionamos el decorador con @
  - Dicho decorador puede incluir parámetros

```
@DecoradorClase
 class miClase{
```

```
@DecoradorClase(parámetros)
 class miClase{
```

Utilizaremos decoradores ya definidos

Tarea 1:

- Cambia el nombre mostrado y pon el tuyo
- Muestra tu apellido en otra variable
- Cambia el logo

Ejercicio 1a. Primer componente desde cero

- Creamos un proyecto “Contador”
- Borrar pruebas y el css
- Creamos carpeta pages
- Dentro de pages creamos fichero /counter/counter.ts
- Creamos la clase Counter
- Establecemos el decorador que la cataloga como un componente
- Definimos una ruta para ese componente (por ejemplo, la raíz).
- Creamos una variable en la clase counter inicializada a 10
- Creamos un método que incremente en un valor que se le pasa por parámetro el contador.
- Creamos dentro de la clase una template y mostramos el contador en html y creamos 3 botones: sumar 1, restar 1 y resetear que devuelva el contador a su valor inicial.
- Una vez que se muestran los campos extraer el código a un fichero html.

Señales

- Un señal es una variable reactiva que guarda un valor y avisa automáticamente a Angular para que actualice la vista cuando ese valor cambia.
- Lectura y escritura sencilla: se lee como una función (signal()) y se actualiza con .set() o .update().
- Renderizado automático: cuando cambia su valor, Angular vuelve a pintar solo lo necesario sin usar zone.js ni ChangeDetectorRef.

```
export class Counter {

  counter = 10;
  counterSignal = signal(10);
```

Ejercicio 1b. Practicar lo aprendido hasta el momento

- Descarga el código y enunciado de Moodle y realiza el ejercicio propuesto.

Señales computadas. Readonly signals

- Señal cuyo valor se calcula automáticamente a partir de otras señales, sin que tú tengas que actualizarla manualmente (no se puede hacer).

ts

```ts
name = signal("Ironman");
age = signal(45);

heroDescription = computed (()=> `${this.name()} - ${this.age()}`);
```

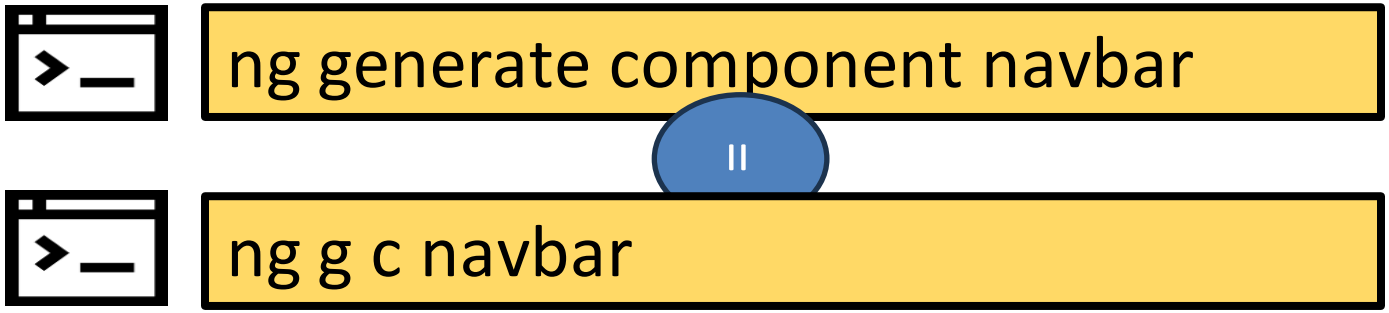
html

```html
<td>Método:</td>
<dd>{{ heroDescription() }}</dd>
```

Resumen:

- Carpeta app contiene todo lo que se usa en la aplicación
- Pages para componentes que ocupan toda la pantalla.
- Las páginas se referencian en routes.ts
- App.html contiene el router-oulet, que se encarga de mostrar un componente particular en función de la ruta en la que nos encontramos.
- Index.html, toda nuestra aplicación está montada en app-root
- Señales es una forma de cambiar el estado de la aplicación.

Crear componentes



RouterLink

Es **una directiva del router de Angular** que sirve para **navegar entre rutas de la aplicación** desde el HTML **sin recargar la página**.

Funciona como un enlace (href) pero gestionado por Angular.

html

```html
<nav>
  <a routerLink="/">Contador</a>
  <a routerLink="/hero">Heroe</a>
</nav>
```

ts

```ts
@Component({
  selector: 'app-navbar',
  imports: [RouterLink],
  templateUrl: './navbar.html',
  styleUrl: './navbar.scss',
})
export class Navbar {

}
```

Actividad

Crear un componente navbar con dos enlaces a la página contador y héroe.

Nota 1: Todos los componentes se colocarán en la ruta:

app/components

Nota 2: aplica este css:

```
nav {
    display: flex;
    justify-content: space-around;
    background-color: #212121;
    color: white;
    padding: 10px;
  }

 nav a {
    color: white;
    text-decoration: none;
  }
```

Rutas por defecto

Podemos establecer una ruta a la que redirigiremos el flujo de navegación en caso de no esté definida esa ruta:

app.routes.ts

```
{
   path: 'ruta1',
   component: MiComponente,
 },
 {
   path: '**',
   redirectTo: ''
 }
```

RouterLinkActive

- **routerLinkActive** es una directiva de Angular que te permite aplicar una clase CSS automáticamente a un elemento cuando la ruta asociada al routerLink está activa.
- Sirve para marcar visualmente **qué página está actualmente seleccionada** en un menú o barra de navegación.

css

```
.active {
  font-weight: bold;
  color: red;
}
```

html

```
<a routerLink="/"
routerLinkActive="active">Contador</a>
```

- Si queremos que afecte solo al path exacto y no a los que contenga ese path :

html

```
<a routerLink="/"
routerLinkActive="active"
[routerLinkActiveOptions]="{exact:true}"
>Contador</a>
```

Tarea

- Crear componente dragonball
- Crear ruta para componente
- Agregarlo a navbar
- Crear un interface Personaje con campos id, name y power.

Directivas

UD2: Desarrollo de aplicaciones basado en componentes con Angular

Plantillas y vistas

- **Plantilla**. Es un archivo HTML donde mediante sintaxis Angular podemos alterar el DOM de la aplicación
  - Esto se hace mediante unas instrucciones especiales llamadas **directivas**
  - **Las plantillas mezclan HTML y directivas**
- **Vista**. Es lo que finalmente ve el navegador

Algunas directivas comunes

Directiva	Descripción
`for`	Interacción con una lista
`If`	Muestra/oculta una etiqueta
`(evento)`	Definición de evento
`{{propiedad}}` `[objecto]`	Permiten enlazar con los datos a mostrar en el DOM
`<app-usuario>`	Selector de otro componente

DATA BINDING

Directiva For

- Trabaja recorriendo un array, generando tantas etiquetas como elementos tiene el listado

```
public entradas:any[]=[
  {titulo:"Entrada 1",resumen:"Contenido de la entrada 1"},
  {titulo:"Entrada 2",resumen:"Contenido de la entrada 2"},
  {titulo:"Entrada 3",resumen:"Contenido de la entrada 3"}
];
```

COMPONENTE

```
<h2>Listado</h2>
<p> Listado de usuarios</p>
<ul>
     @for (ingredient of ingredientList; track ingredient.name) {
     <li>{{ ingredient.quantity }} - {{ ingredient.name }}</li>
     }
</ul>
```

PLANTILLA

If

- Muestra/oculta en función del valor de un atributo

```
@if (isAdmin) {
      <button>Borrar</button>
    }
```

El componente solo se muestra el botón si se cumple condición

Tarea

- Seguimos con el componente dragonball
- Crear un interface Personaje con campos id, name y power.

```
interface Personaje {
  id: number;
  name: string;
  power: number
}
```

- Crear una señal personajes con un array de 3 personajes.
- Pintar en html los personajes usando un for
- Pintar solo los personajes que tengan un poder > 500 usando if

Ejercicio repaso:

Descargar ejercicio desde Moodle.

ngClass- ngStyle - Alternativas

- **ngClass** se usa para agregar o eliminar clases de un elemento basado en una condición.
  - Alternativa a) Binding directo de clases. Binding directo a la propiedad class para manipular las clases de un elemento directamente.

```
<div [class.active]="isActive" [class.hidden]="isHidden"></div>
```

ngClass- ngStyle - Alternativas

- Alternativa b) Uso de un objeto con ngClass.

```
<div [ngClass]="{'active': isActive, 'hidden': isHidden}"></div>
```

ngClass- ngStyle – Alternativas

- **ngClass** se utiliza para aplicar estilos en línea de forma condicional
  - Alternativa a) Usar el binding directo a la propiedad style para aplicar estilos de manera condicional.

```
<div [style.color]="isActive ? 'red' : 'green'"></div>
```

ngClass- ngStyle – Alternativas

- Alternativa b) Uso de un objeto con ngStyle.

```
<div [ngStyle]="{
       'color': isActive ? 'red' : 'green',
        'font-size': size + 'px'}">
</div>
```

Inputs

- conecta un campo de texto con un **signal**

```
<input
  type="text"
  placeholder="Nombre"
  [value]="name()"
  (input)="name.set(txtName.value)"
  #txtName
/>
```

Tarea

- Hacer lo mismo con power
- Hacer una función que al pulsar sobre el botón pinte en consola el nombre y el power
- Modificar la función para que cree un nuevo personaje y lo agregue al listado.
- Crear otra función que limpie los valores de power y name y llámalo des de la función anterior.

Ampliación ejercicio repaso

Sección de contacto.

- Mostrar en tiempo real lo que escribe el usuario
- Colorear los input según si están vacíos o llenos usando la propiedad css border.
- Cambiar el tamaño del campo Mensaje en función de la longitud del mismo (propiedad height).
- Desactivar el botón si falta algún campo

Comunicación entre componentes

- la comunicación entre componentes padre → hijo y hijo → padre se realiza mediante la nueva API basada en funciones:
  - input() reemplaza a @Input()
  - output() reemplaza a @Output()
- Estas funciones hacen el código más simple, más seguro y totalmente compatible con signals.

Comunicación Padre → Hijo con input()

- El componente hijo recibe datos del padre mediante input().
  Ejemplo hijo (personaje-hijo.component.ts):

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

@Component({
  selector: 'personaje-hijo',
  standalone: true,
  template: `El héroe es {{ nombre() }}`
})
export class PersonajeHijoComponent {
  nombre = input.required<string>();
}
```

En el padre:

```html
<personaje-hijo [nombre]="heroe.nombre"></personaje-hijo>
```

Comunicación Padre → Hijo con input()

Cómo funciona:

- input() crea una signal de solo lectura.
- El hijo recibe el valor.
- Para leerlo siempre se usa:

```
nombre()
```

(con paréntesis porque es una signal).

Comunicación Hijo → Padre con output()

- Cuando el hijo necesita enviar un dato al padre (un clic, una puntuación, un valor...), se usa output().

  Hijo:

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

@Component({
  selector: 'personaje-hijo',
  standalone: true,
  template: `<button (click)="emitir()">Enviar al padre</button>`
})
export class PersonajeHijoComponent {

  mensaje = input<string>('Hola padre!');
  notificar = output<string>();

  emitir() {
    this.notificar.emit(this.mensaje());
  }
}
```

Comunicación Hijo → Padre con output()

Padre:

```html
<personaje-hijo
  [mensaje]="'Kamehameha!'"
  (notificar)="recibirMensaje($event)">
</personaje-hijo>
```

html

```ts
recibirMensaje(texto: string) {
  console.log('Mensaje del hijo:', texto);
}
```

TS

Cómo funciona:

- output() crea un EventEmitter simplificado.
- El hijo llama a .emit() para enviar datos.
- El padre escucha el evento igual que un evento nativo.

Ejercicio ampliación:

Ampliar el formulario de Dragon Ball

- En el componente donde ya se crea un personaje (nombre, poder), añade un nuevo campo "Valoración del personaje" de tipo <select> con valones de 1 a 5, que se guarde al pulsar el botón Agregar.
- Crear un nuevo componente que muestre los tres personajes mejor valorados y que se muestre en la página junto con el formulario y el listado.

¿Qué es un Servicio?

Un servicio es una clase que contiene lógica reutilizable que NO debería estar dentro de los componentes.

Se usa para:

- Compartir datos entre componentes
- Acceder a una API (HTTP)
- Mantener estados (usando signals)
- Encapsular reglas de negocio
- Guardar datos en localStorage
- Realizar cálculos o utilidades

Los servicios se inyectan mediante Dependency Injection.

¿Por qué Servicios?

- Separan lógica y presentación
- El código es más limpio y mantenible
- Permiten compartir datos entre componentes
- Ideal para trabajar con signals y APIs
- Facilitan testing y escalabilidad

Crear un Servicio

```
ng generate service servicios/heroes
```

Angular crea:

```
heroes.service.ts
```

Servicios. Estructura Básica

```
@Injectable({
  providedIn: 'root'
})
export class HeroesService { }
```

Inyección del Servicio en un Componente

```
servicio = inject(HeroesService);

heroes = this.servicio.heroes;
```

LocalStorage en Angular

LocalStorage es un almacenamiento local del navegador, persistente (no se borra al recargar), que guarda pares clave–valor en formato texto.

- Guardar preferencias del usuario
- Sesiones simples
- Datos como favoritos, configuraciones, tokens (no recomendable)
- Carritos de compra
- Estado de la app fuera de memoria

Métodos principales de localStorage

```
localStorage.setItem('clave', 'valor');
localStorage.getItem('clave');
localStorage.removeItem('clave');
localStorage.clear();
```

LocalStorage + Signals

Service.ts

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

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

  // 1. Leer del localStorage al iniciar
  favoritos = signal<string[]>(this.cargar());

  private cargar() {
    const data = localStorage.getItem('favoritos');
    return data ? JSON.parse(data) : [];
  }

  private guardar() {
    localStorage.setItem('favoritos', JSON.stringify(this.favoritos()));
  }
    ......
```

LOCALSTORAGE + SIGNALS + EFFECTS

Angular 20 permite crear **efectos reactivos** que se ejecutan automáticamente cuando cambian uno o varios signals.

Esto es perfecto para:

- Guardar datos en localStorage automáticamente
- Lanzar peticiones HTTP cuando cambia un estado
- Sincronizar partes de la app

¿Qué es un effect() en Angular?

Un se ejecuta automáticamente cada vez que un signal dentro de él cambia.

```
effect(() => {
  console.log('El valor ha cambiado:', contador());
});
```

Servicio con LocalStorage automático usando effect():

- Carga datos desde localStorage al iniciar
- Mantiene un signal con la lista
- effect() guarda automáticamente en localStorage cada vez que cambie la lista

```ts
@Injectable({
  providedIn: 'root'
})
export class FavoritosService {

  // 1. Inicializamos el signal con los datos del localStorage
  favoritos = signal<string[]>(this.cargar());

  constructor() {

    // 3. Guardar automáticamente en localStorage cuando el signal cambia
    effect(() => {
      const datos = this.favoritos();
      localStorage.setItem('favoritos', JSON.stringify(datos));
      console.log('Guardado en localStorage:', datos);
    });

  }

  // 2. Cargar desde localStorage
  private cargar(): string[] {
    const datos = localStorage.getItem('favoritos');
    return datos ? JSON.parse(datos) : [];
  }

    ...
```

Peticiones http

Para activar las peticiones http en angular es necesario hacer la provisión en app.config.ts

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

Peticiones http

Inyectar HttpClient en Angular 20:

```
private http = inject(HttpClient);
```

Hacer petición:

```
cargando = signal(false);
error = signal<string | null>(null);

this.cargando.set(true);

this.http.get(url).subscribe({
  next: resp => {
    this.heroes.set(resp);
    this.cargando.set(false);
  },
  error: () => {
    this.error.set('Error al cargar datos');
    this.cargando.set(false);
  }
});
```

Ejercicio:

- Duplicamos el servicio dragonball llamándolo heroes
- Vamos a hacer una petición a la api:
  - https://akabab.github.io/superhero-api/api/all.json
- Creamos un método cargarPersonajes que haga la petición y hacemos la llamada desde la página que muestra el listado.
- Utilizamos parte de los datos devueltos de héroes para rellenar nuestros personajes:

```
const listaPersonajes: Personaje[] = resp
  .slice(0, 10)
  .map((h: any, index: number) => ({
    id: index + 1, // Generamos un id incremental
    name: h.name, // Nombre del héroe
    power: Number(h.powerstats?.power ?? 0), // Poder (o 0 si falta)
  }));
```