

Angular Avanzado

Pronoide

Version 18.0.0 2024-11-18

Table of Contents

1. Transclusión	1
1.1. Lab: Transclusión	2
2. Decorador ViewChild	10
2.1. Lab: ViewChild	11
3. Componentes dinámicos	16
3.1. Lab: Componentes dinámicos	17
4. Progressive Web Apps (PWAs)	28
4.1. Beneficios de usar una PWA	29
4.2. App Manifest	30
4.3. Service Workers	31
4.4. Lab: Progressive Web Apps (PWAs)	33
4.5. Notificaciones Push	61
4.6. Lab: Notificaciones Push en PWAs	63
5. Angular Universal	81
5.1. Server Side Rendering	82
5.2. Lab: Server Side Rendering	83
5.3. Static Site Generation (SSG o Prerendering)	84
5.4. Lab: Prerendering	85
6. Ngrx	101
6.1. Actions	102
6.2. Reducers	103
6.3. Selectors	104
6.4. Lab: Contador con Ngrx	105
6.5. Effects	114
7. Internacionalización	115
7.1. Lab: internacionalización	116
7.2. Lab: Internacionalización con ngx-translate	126

Chapter 1. Transclusión

En Angular podemos utilizar la **transclusión**, que consiste en poder pasarle a un componente un código HTML que pueden ser etiquetas de HTML u otros componentes de Angular para utilizarlo dentro del componente.

Normalmente esto lo usamos cuando un mismo componente vamos a querer que muestre distintas cosas internamente, es decir, que inicialmente a la hora de construirlo no sabemos exactamente que van a querer mostrar aquellos usuarios que usen este componente.

Un ejemplo podría ser un carrusel en el que una vez podemos mostrar imágenes, otra vez podemos usarlo para mostrar componentes Card, otra lo usamos para mostrar textos...

Para poder usar la transclusión, lo único que tenemos que hacer es pasar el contenido a mostrar dentro del componente entre las etiquetas de este (la de apertura y la de cierre). Y luego indicaremos en que parte del HTML del componente se tiene que inyectar este contenido con la etiqueta **ng-content**.

Además, se nos permite pasar más de un elemento, y como tal, podemos indicar distintas posiciones para ellos utilizando un **ng-content** principal, y luego otros secundarios.

Los secundarios llevan un atributo **select** con un selector CSS que apunta a que parte de los elementos que se inyectan se tienen que poner en la posición de esta etiqueta.

1.1. Lab: Transclusión

En este laboratorio vamos a ver como crear un componente Acordeon en el que vamos a proyectar el contenido a mostrar desde el exterior.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-18-transclusion-lab
? Which stylesheet format would you like to use? CSS
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation
(SSG/Prerendering)? N
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado, creamos un componente y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-18-transclusion-lab
$ ng g c acordeon
$ ng s
```

Empezaremos creando la estructura del componente acordeon.

En la plantilla vamos a tener dos secciones, una para la cabecera y la otra para el cuerpo del acordeón.

/angular-18-transclusion-lab/src/app/acordeon/acordeon.component.html

```
<div>
  <div>
    <h3>Aquí va el título</h3>
  </div>
  <div>
    Aquí va el contenido
  </div>
</div>
```

El título lo vamos a recibir desde el componente superior con un **@Input**, por lo que vamos a añadir esta propiedad en el TypeScript del componente **Acordeon**.

/angular-18-transclusion-lab/src/app/acordeon/acordeon.component.ts

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

@Component({
  selector: 'app-acordeon',
  standalone: true,
  imports: [],
  templateUrl: './acordeon.component.html',
```

```

    styleUrls: './acordeon.component.css'
  })
export class AcordeonComponent implements OnInit {
  @Input() titulo: string = 'Título'

  constructor() { }

  ngOnInit(): void {
  }

}

```

La idea de este acordeón es que al pulsar sobre la cabecera se despliegue el contenido o se colapse, por lo que vamos a necesitar otra propiedad para controlar si hay que mostrarlo abierto o cerrado.

/angular-18-transclusion-lab/src/app/acordeon/acordeon.component.ts

```

import { Component, Input, OnInit } from '@angular/core';

@Component({
  selector: 'app-acordeon',
  standalone: true,
  imports: [],
  templateUrl: './acordeon.component.html',
  styleUrls: './acordeon.component.css'
})
export class AcordeonComponent implements OnInit {
  @Input() titulo: string = 'Título'
  estaCerrado: boolean = false

  constructor() { }

  ngOnInit(): void {
  }

}

```

Vamos a añadir una función para ir cambiando el valor de **estaCerrado** y que así se pueda desplegar y colapsar el contenido.

/angular-18-transclusion-lab/src/app/acordeon/acordeon.component.ts

```

import { Component, Input, OnInit } from '@angular/core';

@Component({
  selector: 'app-acordeon',
  standalone: true,
  imports: [],
  templateUrl: './acordeon.component.html',
  styleUrls: './acordeon.component.css'
})

```

```

})
export class AcordeonComponent implements OnInit {
  @Input() titulo: string = 'Título'
  estaCerrado: boolean = false

  constructor() { }

  ngOnInit(): void {
  }

  toggleAcordeon() {
    this.estaCerrado = !this.estaCerrado
  }
}

```

Esta función que acabamos de añadir se va a ejecutar cuando pulsemos sobre la cabecera del Acordeon, por lo que vamos a añadirle el evento **click** en la plantilla.

También cambiaremos el contenido de la cabecera para mostrar el título que vamos a recibir desde el componente superior.

/angular-18-transclusion-lab/src/app/acordeon/acordeon.component.html

```

<div>
  <div (click)="toggleAcordeon()">
    <h3>{{titulo}}</h3>
  </div>
  <div>
    Aquí va el contenido
  </div>
</div>

```

El siguiente paso es añadir la etiqueta **ng-content** dentro del **div** que mostrará el contenido. De esta forma, vamos a poder proyectar el contenido que queremos mostrar desde fuera, haciendo que lo que se muestre no tenga la misma estructura siempre.

/angular-18-transclusion-lab/src/app/acordeon/acordeon.component.html

```

<div>
  <div (click)="toggleAcordeon()">
    <h3>{{titulo}}</h3>
  </div>
  <div>
    <ng-content></ng-content>
  </div>
</div>

```

Hasta aquí, ya podemos ir a probar el componente, a ver si se muestran distintos elementos dentro

de el.

En el componente **App** vamos a utilizar este componente, una vez para mostrar una lista de datos, y otra para mostrar un párrafo.

Estos dos elementos se los vamos a pasar entre las etiquetas de apertura y cierre de **app-acordeon**, ya que es lo que se va a mostrar donde hemos puesto el **ng-content**.

/angular-18-transclusion-lab/src/app/app.component.html

```
<app-acordeon titulo="Lista de productos">
  <ul>
    <li>Albahaca</li>
    <li>Queso parmesano</li>
    <li>Pechuga de pollo</li>
    <li>Tomates</li>
  </ul>
</app-acordeon>

<app-acordeon titulo="Frase de Big Bang Theory">
  <p>¿El amor está en el aire? Erróneo. El nitrógeno, el oxígeno y el dióxido de carbono están en el aire.</p>
</app-acordeon>
```

Ahora tenemos que importar el componente del **Acordeon** en el **AppComponent**.

/angular-18-transclusion-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { AcordeonComponent } from './acordeon/acordeon.component';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    AcordeonComponent,
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
```

Ya deberíamos de poder ver el contenido de los dos acordeones.

Ahora vamos a añadirle unas clases y estilos al componente para que mejore su apariencia y que se oculte o muestre el contenido al pulsar sobre la cabecera de estos.

Vamos a añadirle las siguientes clases fijas al Acordeon.

```
/angular-18-transclusion-lab/src/app/acordeon/acordeon.component.html
```

```
<div class="acordeon">
  <div class="acordeon-heading" (click)="toggleAcordeon()">
    <h3>{{titulo}}</h3>
  </div>
  <div class="acordeon-content">
    <ng-content></ng-content>
  </div>
</div>
```

Y también le vamos a añadir unas clases **cerrado** y **abierto**, pero esta vez con la directiva **ngClass**, ya que estas se van a activar o desactivar dependiendo del valor de la propiedad **estaCerrado**. Primero la importaremos en el componente.

```
/angular-18-transclusion-lab/src/app/acordeon/acordeon.component.ts
```

```
import { NgClass } from '@angular/common';
import { Component, Input, OnInit } from '@angular/core';

@Component({
  selector: 'app-acordeon',
  standalone: true,
  imports: [
    NgClass,
  ],
  templateUrl: './acordeon.component.html',
  styleUrls: ['./acordeon.component.css']
})
export class AcordeonComponent implements OnInit {
  @Input() titulo: string = 'Título'
  estaCerrado: boolean = false

  constructor() { }

  ngOnInit(): void {
  }

  toggleAcordeon() {
    this.estaCerrado = !this.estaCerrado
  }
}
```

```
/angular-18-transclusion-lab/src/app/acordeon/acordeon.component.html
```

```
<div class="acordeon">
  <div class="acordeon-heading" (click)="toggleAcordeon()">
    <h3>{{titulo}}</h3>
  </div>
```

```

<div class="acordeon-content" [ngClass]="{cerrado: estaCerrado, abierto: !estaCerrado}">
  <ng-content></ng-content>
</div>
</div>

```

Ahora nos vamos al archivo del CSS para añadir los estilos de todas estas clases.

/angular-18-transclusion-lab/src/app/acordeon/acordeon.component.css

```

.acordeon {
  border: 1px solid black;
  border-radius: 5px;
}

.acordeon-heading {
  text-align: center;
  border-bottom: 2px solid black;
  cursor: pointer;
}

.acordeon-content {
  overflow: hidden;
}

.acordeon-content.cerrado {
  height: 0;
}

.acordeon-content.abierto {
  height: auto;
  padding: 20px;
}

```

Después de hacer esto, ya deberíamos de poder desplegar y colapsar los acordeones.

Imaginemos que ahora queremos proyectar otro tipo de contenido dentro del acordeón, pero en otro lugar distinto.

Vamos a hacerlo metiendo este contenido entre las etiquetas de apertura y cierre, al igual que antes, pero esta vez le vamos a poner algo identificativo como una clase.

Vamos a añadir otro párrafo con el autor de la frase del segundo acordeón.

/angular-18-transclusion-lab/src/app/app.component.html

```

<app-acordeon titulo="Lista de productos">
  <ul>
    <li>Albahaca</li>
    <li>Queso parmesano</li>
    <li>Pechuga de pollo</li>
    <li>Tomates</li>
  </ul>

```

```

</app-acordeon>

<app-acordeon titulo="Frase de Big Bang Theory">
  <p>¿El amor está en el aire? Erróneo. El nitrógeno, el oxígeno y el dióxido de carbono están en el aire.</p>
  <p class="ac-footer">Sheldon Cooper</p>
</app-acordeon>

```

Para mostrar esto nuevo en alguna parte de la plantilla del Acordeon, vamos a añadirle otra caja dentro del contenido y dentro de esta pondremos otro **ng-content**, pero esta vez con una propiedad **select** cuyo valor será el selector del elemento que se va a proyectar y que queremos pintar en este otro **ng-content**.

/angular-18-transclusion-lab/src/app/acordeon/acordeon.component.html

```

<div class="acordeon">
  <div class="acordeon-heading" (click)="toggleAcordeon()">
    <h3>{{titulo}}</h3>
  </div>
  <div class="acordeon-content" [ngClass]="{cerrado: estaCerrado, abierto: !estaCerrado}">
    <ng-content></ng-content>
    <div class="acordeon-footer">
      <ng-content select=".ac-footer"></ng-content>
    </div>
  </div>
</div>

```

Por último, vamos a añadirle los estilos para esta nueva clase que hemos añadido.

/angular-18-transclusion-lab/src/app/acordeon/acordeon.component.css

```

.acordeon {
  border: 1px solid black;
  border-radius: 5px;
}

.acordeon-heading {
  text-align: center;
  border-bottom: 2px solid black;
  cursor: pointer;
}

.acordeon-content {
  overflow: hidden;
}

.acordeon-content.cerrado {
  height: 0;
}

.acordeon-content.abierto {
  height: auto;
}

```

```
padding: 20px;  
}  
  
.acordeon-footer {  
text-align: right;  
font-style: italic;  
}
```

Ya deberíamos ver este footer dentro del segundo acordeón, y con los estilos aplicados.

Chapter 2. Decorador ViewChild

El decorador ViewChild es un decorador de propiedad, por lo que se le asigna a las propiedades de los componentes.

Con este decorador podemos acceder al primer elemento o directiva que hace match con el selector que se le pasa como parámetro.

El decorador acepta como selectores principalmente los siguientes, aunque hay alguno más:

- Clases de componentes y directivas: `@ViewChild(MiComponente) prop1: MiComponente.`
- Referencias o variables de plantilla: `@ViewChild('miRef') prop2: any`

Los valores a los que apunta este decorador se asignan después del método del ciclo de vida `ngAfterViewInit`, por lo que si necesitamos acceder a las propiedades que devuelve al cargar el componente, tendremos que hacerlo dentro de este método en lugar de hacerlo en el `ngOnInit`.

2.1. Lab: ViewChild

En este laboratorio vamos a ver como crear un modal y acceder a sus propiedades y métodos desde el exterior con el decorador `@ViewChild`.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-18-viewchild-lab
? Which stylesheet format would you like to use? CSS
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation
(SSG/Prerendering)? N
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado, creamos un componente y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-18-viewchild-lab
$ ng g c modal
$ ng s
```

Vamos a empezar por implementar el modal.

Dentro de la plantilla, vamos a tener dos **divs**, uno que hará de backdrop (la sombra que aparece por detrás de la caja del modal) y el otro para la propia caja que va a tener el contenido.

`/angular-18-viewchild-lab/src/app/modal/modal.component.html`

```
<div class="backdrop">
  <div id="modal">
    </div>
  </div>
```

Dentro del modal, vamos a añadir un botón para cerrarlo y la etiqueta **ng-content** con la que inyectaremos el contenido a mostrar desde el exterior, usando la transclusión.

`/angular-18-viewchild-lab/src/app/modal/modal.component.html`

```
<div class="backdrop">
  <div id="modal">
    <ng-content></ng-content>
    <button type="button">Cerrar</button>
  </div>
</div>
```

Ahora vamos a añadir en el TypeScript del modal una propiedad **hidden** con la que vamos a controlar si se tiene que mostrar o no el modal.

También vamos a añadir dos métodos, **abrir** y **cerrar** para cambiar el valor de la propiedad anterior.

/angular-18-viewchild-lab/src/app/modal/modal.component.ts

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

@Component({
  selector: 'app-modal',
  standalone: true,
  imports: [],
  templateUrl: './modal.component.html',
  styleUrls: ['./modal.component.css'
})
export class ModalComponent implements OnInit {
  hidden: boolean = true

  constructor() { }

  ngOnInit(): void {
  }

  abrir(): void {
    this.hidden = false
  }

  cerrar(): void {
    this.hidden = true
  }
}
```

Ahora pondremos dos eventos click para llamar a **cerrar**, uno en el backdrop y otro en el botón.

/angular-18-viewchild-lab/src/app/modal/modal.component.html

```
<div class="backdrop" (click)="cerrar()">
  <div id="modal">
    <ng-content></ng-content>
    <button type="button" (click)="cerrar()">Cerrar</button>
  </div>
</div>
```

También vamos a añadir la directiva **@if** sobre el backdrop para eliminar todo el contenido cuando se tiene que ocultar el modal.

/angular-18-viewchild-lab/src/app/modal/modal.component.html

```
@if (!hidden) {
  <div class="backdrop" (click)="cerrar()">
    <div id="modal">
```

```

<ng-content></ng-content>
<button type="button" (click)="cerrar()">Cerrar</button>
</div>
</div>
}

```

Vamos a añadir unos estilos para el modal y el backdrop en su CSS.

/angular-18-viewchild-lab/src/app/modal/modal.component.css

```

div.backdrop {
  position: fixed;
  background-color: rgba(0, 0, 0, 0.8);
  width: 100vw;
  height: 100vh;
  display: flex;
  justify-content: center;
  align-items: center;
}

div#modal {
  width: 40%;
  margin: 0 auto;
  background-color: white;
  padding: 20px;
  border-radius: 10px;
}

```

Pues ahora nos vamos a ir al componente App, en el que vamos a pintar este modal, con algo de contenido dentro, el cual tenemos que pasar entre las dos etiquetas de **app-modal** para inyectarlo en el **ng-content** que hemos puesto dentro.

/angular-18-viewchild-lab/src/app/app.component.html

```

<app-modal>
  <h4>Hakuna matata, vive y se feliz</h4>
</app-modal>

```

También añadiremos un botón para poder abrir el modal.

/angular-18-viewchild-lab/src/app/app.component.html

```

<app-modal>
  <h4>Hakuna matata, vive y se feliz</h4>
</app-modal>

<button type="button" (click)="abrirModal()">Abrir modal</button>

```

Dentro de su TypeScript tenemos que añadir la función **abrirModal** y tenemos que importar el

componente del modal para poder utilizarlo.

/angular-18-viewchild-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { ModalComponent } from './modal/modal.component'

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    ModalComponent,
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  abrirModal() {

  }
}
```

Para poder abrir el modal desde esta función, necesitamos poder acceder al componente del modal, y para ello vamos a utilizar el decorador **ViewChild**.

Este decorador necesita recibir como parámetro el nombre de una variable de plantilla que tenemos que poner en la etiqueta del componente, o la otra opción es pasarle el nombre de la clase del componente.

Vamos a usar la de la variable de plantilla, por lo que se la vamos a añadir en la etiqueta **app-modal**.

/angular-18-viewchild-lab/src/app/app.component.html

```
<app-modal #modal>
  <h4>Hakuna matata, vive y se feliz</h4>
</app-modal>

<button type="button" (click)="abrirModal()">Abrir modal</button>
```

Una vez tenemos la variable de plantilla, vamos a añadir el decorador y le pasaremos como parámetro **modal**, y con el declararemos una propiedad con el mismo nombre.

/angular-18-viewchild-lab/src/app/app.component.ts

```
import { Component, ViewChild } from '@angular/core';
import { ModalComponent } from './modal/modal.component'

@Component({
```

```

    selector: 'app-root',
    standalone: true,
    imports: [
      ModalComponent,
    ],
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css'
  })
export class AppComponent {
  @ViewChild('modal') modal!: ModalComponent;

  abrirModal() {
  }
}

```

Solo nos queda llamar al método **abrir** del modal dentro del método **abrirModal** del componente App.

/angular-18-viewchild-lab/src/app/app.component.ts

```

import { Component, ViewChild } from '@angular/core';
import { ModalComponent } from './modal/modal.component'

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    ModalComponent,
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'
})
export class AppComponent {
  @ViewChild('modal') modal!: ModalComponent;

  abrirModal() {
    this.modal.abrir()
  }
}

```

Si entramos en la aplicación <http://localhost:4200/>, ya debemos de poder abrir y cerrar el modal.

Chapter 3. Componentes dinámicos

Cuando usamos componentes dentro de las aplicaciones de Angular, estos los ponemos en las plantillas de otros componentes, por lo tanto, tienen una posición fija, dada por el lugar en el que ponemos la etiqueta de estos.

Pues en Angular podemos utilizar otro tipo de componentes, estos son los **componentes dinámicos**, y la diferencia con los anteriores es que no tienen esa posición fija, es decir, no los vamos a añadir en nuestra aplicación poniendo su etiqueta dentro de la plantilla de otro componente.

Estos componentes se crean mediante código, y se van a pintar en un sitio u otro en tiempo de ejecución.

Por ejemplo, un ejemplo de este tipo de componentes son el contenido de los modales, ya que en lugar de crear distintos modales que pinten distintos componentes, podríamos crear un único modal en el que vamos a pintar los componentes de forma dinámica.

Para utilizar este tipo de componentes, lo primero que necesitamos es una forma de acceder a la posición donde se van a añadir. Para ello, lo normal es crear una directiva e injectarle **ViewContainerRef**.

El **ViewContainerRef** es una referencia a un contenedor en el que podremos añadir y eliminar estos componentes dinámicos.

Esta directiva tendremos que añadirla sobre un elemento (puede ser también un **ng-template**), y desde el componente al cual pertenezca la plantilla sobre la que se ha usado esta directiva, nos vamos a encargar de indicar que tiene que pintar.

Esto lo haremos accediendo al **host**, elemento en el que hemos puesto la directiva, y llamando a los métodos de la instancia de **ViewContainerRef**. Estos métodos son:

- **createComponent**: crea el componente que se le pasa como parámetro en la posición del host.
- **clear**: limpia el contenido que hay pintado en el host.

3.1. Lab: Componentes dinámicos

En este laboratorio vamos a mostrar una lista de audios y videos que se cargarán como componentes dinámicos dentro de un modal según pulsemos sobre ellos.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-18-componentes-dinamicos-lab
? Which stylesheet format would you like to use? CSS
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? N
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado, creamos tres componentes, una directiva y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-18-componentes-dinamicos-lab
$ ng g c modal
$ ng g c video
$ ng g c audio
$ ng g d host
$ ng s
```

Lo primero que vamos a hacer es crear en la carpeta de **public** dos carpetas, una de **videos** y otra de **audios**, y después descargarnos 3 videos y 3 audios de las siguientes páginas para guardarlos en las carpetas:

- Audios: <https://licensing.jamendo.com/es/catalogo>
- Videos: <https://www.pexels.com/videos>

Ahora nos vamos a ir al TypeScript del componente App, donde vamos a crear una interfaz **IMedia** para definir el tipo de los videos y audios y que propiedades van a tener.

Estos items tendrán:

- src: la ruta hasta el archivo.
- titulo: el título que vamos a mostrar.
- tipo: un string audio o video para saber de que tipo es el elemento que vamos a reproducir.

/angular-18-componentes-dinamicos-lab/src/app/app.component.ts

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

interface IMedia {
  src: string,
  titulo: string,
  tipo: 'video' | 'audio'
}
```

```

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
}

```

Ahora vamos a crear dos propiedades, una, **audios** con un array en el que vamos a añadir los datos para los audios que hemos descargado, y haremos lo mismo con **videos**.

/angular-18-componentes-dinamicos-lab/src/app/app.component.ts

```

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

interface IMedia {
  src: string,
  titulo: string,
  tipo: 'video' | 'audio'
}

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  videos: Array<IMedia> = [
    { src: 'videos/video1.mp4', titulo: 'Puesta de sol silueta - Maksim Goncharenok', tipo: 'video' },
    { src: 'videos/video2.mp4', titulo: 'Vela a la lluvia - Beytlik', tipo: 'video' },
    { src: 'videos/video3.mp4', titulo: 'Flores al viento - Anastasia Ilina-Makarova', tipo: 'video' },
  ]
  audios: Array<IMedia> = [
    { src: 'audios/audio1.mp3', titulo: 'Dance Endlessly - TimTaj', tipo: 'audio' },
    { src: 'audios/audio2.mp3', titulo: 'Feel Good - MatFix', tipo: 'audio' },
    { src: 'audios/audio3.mp3', titulo: 'Epic Trailer - Dmitry Taras', tipo: 'audio' },
  ]
}

```

En la plantilla vamos a mostrar estas dos listas y les pondremos un evento **click** a cada item para llamar a una función **mostrarMedia** a la que le pasaremos el objeto sobre el que hemos pulsado.

/angular-18-componentes-dinamicos-lab/src/app/app.component.html

```

<ul>
  @for (video of videos; track $index) {
    <li (click)="mostrarMedia(video)">{{video.titulo}}</li>
  }

```

```

</ul>

<ul>
  @for (audio of audios; track $index) {
    <li (click)="mostrarMedia(audio)">{{audio.titulo}}</li>
  }
</ul>

```

Ahora vamos a añadir el componente del modal al que le pondremos una variable de plantilla para poder acceder después a el.

Y mediante transclusión le vamos a inyectar una plantilla con la directiva **HostDirective**.

/angular-18-componentes-dinamicos-lab/src/app/app.component.html

```

<app-modal #modal>
  <ng-template appHost></ng-template>
</app-modal>

<ul>
  @for (video of videos; track $index) {
    <li (click)="mostrarMedia(video)">{{video.titulo}}</li>
  }
</ul>

<ul>
  @for (audio of audios; track $index) {
    <li (click)="mostrarMedia(audio)">{{audio.titulo}}</li>
  }
</ul>

```

Vamos a poner en el TypeScript la función de **mostrarMedia** y con el decorador **ViewChild** vamos a buscar tanto el modal como la directiva. Además tendremos que importar el componente del modal y la directiva HostDirective.

/angular-18-componentes-dinamicos-lab/src/app/app.component.ts

```

import { Component, ViewChild } from '@angular/core';
import { HostDirective } from './host.directive';
import { ModalComponent } from './modal/modal.component';

interface IMedia {
  src: string,
  titulo: string,
  tipo: 'video' | 'audio'
}

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    ModalComponent,

```

```

    HostDirective,
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'
})
export class AppComponent {
  videos: Array<IMedia> = [
    { src: 'videos/video1.mp4', titulo: 'Puesta de sol silueta - Maksim Goncharenok', tipo: 'video' },
    { src: 'videos/video2.mp4', titulo: 'Vela a la lluvia - Beytlik', tipo: 'video' },
    { src: 'videos/video3.mp4', titulo: 'Flores al viento - Anastasia Ilina-Makarova', tipo: 'video' },
  ]
  audios: Array<IMedia> = [
    { src: 'audios/audio1.mp3', titulo: 'Dance Endlessly - TimTaj', tipo: 'audio' },
    { src: 'audios/audio2.mp3', titulo: 'Feel Good - MatFix', tipo: 'audio' },
    { src: 'audios/audio3.mp3', titulo: 'Epic Trailer - Dmitry Taras', tipo: 'audio' },
  ]
  @ViewChild(HostDirective) host!: HostDirective
  @ViewChild('modal') modal!: ModalComponent;

  mostrarMedia(itemMedia: IMedia): void {
  }
}

```

Antes de continuar por aquí, vamos a inyectar el **ViewContainerRef** en la directiva.

/angular-18-componentes-dinamicos-lab/src/app/host.directive.ts

```

import { Directive, ViewContainerRef } from '@angular/core';

@Directive({
  selector: '[appHost]',
  standalone: true
})
export class HostDirective {

  constructor(
    public viewContainerRef: ViewContainerRef,
  ) { }

}

```

También tenemos que cambiar el contenido de los componentes **AudioComponent** y **VideoComponent** a los que les declararemos una propiedad **src** que usaremos para mostrar estos items en las etiquetas **audio** y **video**.

/angular-18-componentes-dinamicos-lab/src/app/video/video.component.ts

```

import { Component, Input, OnInit } from '@angular/core';

@Component({
  selector: 'app-video',
  standalone: true,

```

```

imports: [],
templateUrl: './video.component.html',
styleUrl: './video.component.css'
})
export class VideoComponent implements OnInit {
  @Input() src: string = ''
  constructor() { }

  ngOnInit(): void {
  }
}

```

/angular-18-componentes-dinamicos-lab/src/app/video/video.component.html

```
<video [src]="src" controls width="200"></video>
```

/angular-18-componentes-dinamicos-lab/src/app/audio/audio.component.ts

```

import { Component, Input, OnInit } from '@angular/core';

@Component({
  selector: 'app-audio',
  standalone: true,
  imports: [],
  templateUrl: './audio.component.html',
  styleUrls: ['./audio.component.css'
})
export class AudioComponent implements OnInit {
  @Input() src: string = ''

  constructor() { }

  ngOnInit(): void {
  }
}

```

/angular-18-componentes-dinamicos-lab/src/app/audio/audio.component.html

```
<audio [src]="src" controls></audio>
```

El siguiente paso es crear el modal. Dentro del TypeScript vamos a añadir la propiedad **hidden** que la usaremos para pintar o eliminar el modal y dos métodos para cambiar el valor de esta propiedad.

/angular-18-componentes-dinamicos-lab/src/app/modal/modal.component.ts

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

```

@Component({
  selector: 'app-modal',
  standalone: true,
  imports: [],
  templateUrl: './modal.component.html',
  styleUrls: ['./modal.component.css'
})
export class ModalComponent implements OnInit {
  hidden: boolean = true

  constructor() { }

  ngOnInit(): void {
  }

  abrir(): void {
    this.hidden = false
  }

  cerrar(): void {
    this.hidden = true
  }
}

```

En la plantilla, pondremos dos divs, en el primero el **@if** con la propiedad **hidden** para controlar cuando mostrar el modal.

En el div interno vamos a añadir el **ng-content** para pintar el contenido que se le va a inyectar, y un botón con el que llamaremos al método que cierra el modal.

/angular-18-componentes-dinamicos-lab/src/app/modal/modal.component.html

```

@if (!hidden) {
  <div class="backdrop">
    <div id="modal">
      <ng-content></ng-content>
      <button type="button" (click)="cerrar()">Cerrar</button>
    </div>
  </div>
}

```

Y ahora la ponemos unos estilos para que se vea mejor.

/angular-18-componentes-dinamicos-lab/src/app/modal/modal.component.css

```

div.backdrop {
  position: fixed;
  background-color: rgba(0, 0, 0, 0.8);
  width: 100vw;
  height: 100vh;
}

```

```

display: flex;
justify-content: center;
align-items: center;
}

div#modal {
width: 40%;
margin: 0 auto;
background-color: white;
padding: 20px;
border-radius: 10px;
}

```

Pues ya tenemos todos los elementos necesarios para añadir los videos y audios dentro del modal como componentes dinámicos.

Esto lo vamos a hacer en el método **mostrarMedia** que habíamos puesto en el componente App.

Vamos a empezar por acceder al **ViewContainerRef** de la directiva **Host** y llamaremos a **clear** para eliminar el contenido anterior si lo hubiese.

/angular-18-componentes-dinamicos-lab/src/app/app.component.ts

```

import { Component, ViewChild } from '@angular/core';
import { HostDirective } from './host.directive';
import { ModalComponent } from './modal/modal.component';

interface IMedia {
  src: string,
  titulo: string,
  tipo: 'video' | 'audio'
}

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    ModalComponent,
    HostDirective,
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  videos: Array<IMedia> = [
    { src: 'videos/video1.mp4', titulo: 'Puesta de sol silueta - Maksim Goncharenok', tipo: 'video' },
    { src: 'videos/video2.mp4', titulo: 'Vela a la lluvia - Beytlik', tipo: 'video' },
    { src: 'videos/video3.mp4', titulo: 'Flores al viento - Anastasia Ilina-Makarova', tipo: 'video' },
  ]
  audios: Array<IMedia> = [
    { src: 'audios/audio1.mp3', titulo: 'Dance Endlessly - TimTaj', tipo: 'audio' },
    { src: 'audios/audio2.mp3', titulo: 'Feel Good - MatFix', tipo: 'audio' },
    { src: 'audios/audio3.mp3', titulo: 'Epic Trailer - Dmitry Taras', tipo: 'audio' },
  ]
  @ViewChild(HostDirective) host!: HostDirective
}

```

```

@ViewChild('modal') modal!: ModalComponent;

mostrarMedia(itemMedia: IMedia): void {
  const viewContainerRef = this.host.viewContainerRef
  viewContainerRef.clear()
}

}

```

Ahora vamos a obtener el tipo del componente que queremos crear en el modal.

/angular-18-componentes-dinamicos-lab/src/app/app.component.ts

```

import { Component, ViewChild } from '@angular/core';
import { HostDirective } from './host.directive';
import { ModalComponent } from './modal/modal.component';
import { VideoComponent } from './video/video.component';
import { AudioComponent } from './audio/audio.component';

interface IMedia {
  src: string,
  titulo: string,
  tipo: 'video' | 'audio'
}

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    ModalComponent,
    HostDirective,
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'
})
export class AppComponent {
  videos: Array<IMedia> = [
    { src: 'videos/video1.mp4', titulo: 'Puesta de sol silueta - Maksim Goncharenok', tipo: 'video' },
    { src: 'videos/video2.mp4', titulo: 'Vela a la lluvia - Beytlik', tipo: 'video' },
    { src: 'videos/video3.mp4', titulo: 'Flores al viento - Anastasia Ilina-Makarova', tipo: 'video' },
  ]
  audios: Array<IMedia> = [
    { src: 'audios/audio1.mp3', titulo: 'Dance Endlessly - TimTaj', tipo: 'audio' },
    { src: 'audios/audio2.mp3', titulo: 'Feel Good - MatFix', tipo: 'audio' },
    { src: 'audios/audio3.mp3', titulo: 'Epic Trailer - Dmitry Taras', tipo: 'audio' },
  ]
  @ViewChild(HostDirective) host!: HostDirective
  @ViewChild('modal') modal!: ModalComponent;

  mostrarMedia(itemMedia: IMedia): void {
    const viewContainerRef = this.host.viewContainerRef
    viewContainerRef.clear()
    const tipoCmp = itemMedia.tipo === 'video' ? VideoComponent : AudioComponent
  }
}

```

El siguiente paso es crear el componente, para lo que llamaremos al método **createComponent** del

viewContainerRef pasándole como parámetro el tipo del componente que queremos crear.

/angular-18-componentes-dinamicos-lab/src/app/app.component.ts

```
import { Component, ViewChild } from '@angular/core';
import { HostDirective } from './host.directive';
import { ModalComponent } from './modal/modal.component';
import { VideoComponent } from './video/video.component';
import { AudioComponent } from './audio/audio.component';

interface IMedia {
  src: string,
  titulo: string,
  tipo: 'video' | 'audio'
}

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    ModalComponent,
    HostDirective,
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  videos: Array<IMedia> = [
    { src: 'videos/video1.mp4', titulo: 'Puesta de sol silueta - Maksim Goncharenok', tipo: 'video' },
    { src: 'videos/video2.mp4', titulo: 'Vela a la lluvia - Beytlik', tipo: 'video' },
    { src: 'videos/video3.mp4', titulo: 'Flores al viento - Anastasia Ilina-Makarova', tipo: 'video' },
  ]
  audios: Array<IMedia> = [
    { src: 'audios/audio1.mp3', titulo: 'Dance Endlessly - TimTaj', tipo: 'audio' },
    { src: 'audios/audio2.mp3', titulo: 'Feel Good - MatFix', tipo: 'audio' },
    { src: 'audios/audio3.mp3', titulo: 'Epic Trailer - Dmitry Taras', tipo: 'audio' },
  ]
  @ViewChild(HostDirective) host!: HostDirective
  @ViewChild('modal') modal!: ModalComponent;

  mostrarMedia(itemMedia: IMedia): void {
    const viewContainerRef = this.host.viewContainerRef
    viewContainerRef.clear()
    const tipoCmp = itemMedia.tipo === 'video' ? VideoComponent : AudioComponent
    const componente: any = viewContainerRef.createComponent(tipoCmp)
  }
}
```

Ahora accedemos a la propiedad **src** a través de la instancia del componente que hemos creado para asignarle la ruta donde se encuentra el audio o video que se va a reproducir.

/angular-18-componentes-dinamicos-lab/src/app/app.component.ts

```
import { Component, ViewChild } from '@angular/core';
import { HostDirective } from './host.directive';
import { ModalComponent } from './modal/modal.component';
import { VideoComponent } from './video/video.component';
```

```

import { AudioComponent } from './audio/audio.component';

interface IMedia {
  src: string,
  titulo: string,
  tipo: 'video' | 'audio'
}

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    ModalComponent,
    HostDirective,
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  videos: Array<IMedia> = [
    { src: 'videos/video1.mp4', titulo: 'Puesta de sol silueta - Maksim Goncharenok', tipo: 'video' },
    { src: 'videos/video2.mp4', titulo: 'Vela a la lluvia - Beytlik', tipo: 'video' },
    { src: 'videos/video3.mp4', titulo: 'Flores al viento - Anastasia Ilina-Makarova', tipo: 'video' },
  ]
  audios: Array<IMedia> = [
    { src: 'audios/audio1.mp3', titulo: 'Dance Endlessly - TimTaj', tipo: 'audio' },
    { src: 'audios/audio2.mp3', titulo: 'Feel Good - MatFix', tipo: 'audio' },
    { src: 'audios/audio3.mp3', titulo: 'Epic Trailer - Dmitry Taras', tipo: 'audio' },
  ]
  @ViewChild(HostDirective) host!: HostDirective
  @ViewChild('modal') modal!: ModalComponent;

  mostrarMedia(itemMedia: IMedia): void {
    const viewContainerRef = this.host.viewContainerRef
    viewContainerRef.clear()
    const tipoCmp = itemMedia.tipo === 'video' ? VideoComponent : AudioComponent
    const componente: any = viewContainerRef.createComponent(tipoCmp)
    componente.instance.src = itemMedia.src
  }
}

```

Y por último, solo nos queda abrir el modal. Como tenemos acceso a el por el **ViewChild**, vamos a llamar al método **abrir** que habíamos añadido dentro de el.

/angular-18-componentes-dinamicos-lab/src/app/app.component.ts

```

import { Component, ViewChild } from '@angular/core';
import { HostDirective } from './host.directive';
import { ModalComponent } from './modal/modal.component';
import { VideoComponent } from './video/video.component';
import { AudioComponent } from './audio/audio.component';

interface IMedia {
  src: string,
  titulo: string,
  tipo: 'video' | 'audio'
}

```

```

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    ModalComponent,
    HostDirective,
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  videos: Array<IMedia> = [
    { src: 'videos/video1.mp4', titulo: 'Puesta de sol silueta - Maksim Goncharenok', tipo: 'video' },
    { src: 'videos/video2.mp4', titulo: 'Vela a la lluvia - Beytlik', tipo: 'video' },
    { src: 'videos/video3.mp4', titulo: 'Flores al viento - Anastasia Ilina-Makarova', tipo: 'video' },
  ]
  audios: Array<IMedia> = [
    { src: 'audios/audio1.mp3', titulo: 'Dance Endlessly - TimTaj', tipo: 'audio' },
    { src: 'audios/audio2.mp3', titulo: 'Feel Good - MatFix', tipo: 'audio' },
    { src: 'audios/audio3.mp3', titulo: 'Epic Trailer - Dmitry Taras', tipo: 'audio' },
  ]
  @ViewChild(HostDirective) host!: HostDirective
  @ViewChild('modal') modal!: ModalComponent;

  mostrarMedia(itemMedia: IMedia): void {
    const viewContainerRef = this.host.viewContainerRef
    viewContainerRef.clear()
    const tipoCmp = itemMedia.tipo === 'video' ? VideoComponent : AudioComponent
    const componente: any = viewContainerRef.createComponent(tipoCmp)
    componente.instance.src = itemMedia.src
    this.modal.abrir()
  }
}

```

Y con esto ya deberíamos de poder abrir el modal y ver el video o escuchar un audio al pulsar sobre ellos en el listado que se está mostrando.

Chapter 4. Progressive Web Apps (PWAs)

Las **Progressive Web Apps (PWA)** son aplicaciones web con funcionalidades extra que les permite comportarse como si fueran aplicaciones móviles. Lo que buscan es que la experiencia de usuario sea igual que con las aplicaciones nativas, y para ello tienen que cumplir las siguientes características:

- De confianza: la aplicación se tiene que cargar instantáneamente al igual que ocurre con las aplicaciones nativas, incluso cuando tengamos una mala conexión a internet o esta sea nula.
- Rápida: tiene que tardar poco en responder al usuario.
- Adictiva: tiene que parecer que es una aplicación nativa, que tenga características propias de las nativas.

Para convertir nuestras aplicaciones de Angular en PWAs tendremos que añadir en nuestro proyecto la librería de `@angular/pwa`.

4.1. Beneficios de usar una PWA

Lo bueno de crear las PWAs es que vamos a tener lo mejor de las páginas web y lo mejor de las aplicaciones a la vez.

Las PWAs nos van a permitir tener nuestra aplicación instalada en nuestros dispositivos móviles, por lo tanto ya no es necesario tener que entrar al navegador del móvil para entrar en ella, sino que tendremos un ícono en nuestro **homescreen** y podremos acceder desde el.

Este ícono no es un acceso directo a nuestra web, sino que al darle se abrirá como una aplicación nativa. Podemos verla en el listado de aplicaciones del dispositivo como si hubiéramos instalado el **apk**.

Cuando entramos en alguna aplicación nativa y no tenemos conexión, la aplicación se carga aunque luego no puedas usar todas las funcionalidades que tiene. Estas aplicaciones tienen que permitirnos usarlas tanto con internet como sin el.

La distribución de estas aplicaciones es mucho más rápida que las de las aplicaciones nativas, ya que no tenemos que esperar a que los marketplaces nos validen la aplicación para que los usuarios puedan empezar a descargarla. En este caso, son **URLs** por tanto con entrar en la página web ya podemos instalarla en nuestro dispositivo y a partir de ahí, la podremos usar como si de una aplicación nativa se tratase.

Son mucho más fáciles de descubrir que las aplicaciones nativas, ya que no es necesario entrar en los marketplaces para buscar algo que instalar, sino que podemos ir navegando por internet, y en el momento que entremos en una web y nos guste la podremos instalar, habiéndola podido probar antes, que esto no se puede hacer con las nativas.

Las webs suelen ocupar mucho menos que las aplicaciones nativas porque estas suelen llevar instaladas librerías que al final es muy probable que no se vayan a usar en la aplicación, mientras que en las PWAs solo vamos a tener las librerías que vamos a usar para construirla.

También podemos tener características propias de las aplicaciones nativas como pueden ser las notificaciones push o el funcionamiento de estas sin internet.

Todos estos beneficios, los vamos a conseguir añadiendo las siguientes funcionalidades a nuestra aplicación web:

- App Manifest
- Service Workers
- Notificaciones Push
- App Shell

4.2. App Manifest

Para poder instalar la aplicación en el dispositivo móvil, necesitamos el archivo de **App Manifest** que es el archivo donde se le da información extra al navegador y que permitirá que se instale la aplicación en los dispositivos móviles y se muestre como una app nativa.

En este archivo vamos a poder añadir las siguientes propiedades:

- **name**: indica el nombre de la aplicación que se va a mostrar por ejemplo en la pantalla de **splash**.
- **short_name**: aquí le indicamos el nombre de la aplicación que queremos mostrar junto al icono en el dispositivo una vez que la tengamos instalada.
- **description**: una descripción sobre la aplicación para mostrarla en sitios donde se pueda mostrar más información sobre la PWA.
- **start_url**: aquí le indicamos la URL de la pantalla que se tiene que abrir al abrir la aplicación.
- **display**: esta propiedad indica como se tiene que mostrar la aplicación una vez que la abramos. Los posibles valores son:
 - **standalone**: se muestra como una aplicación nativa.
 - **fullscreen**: pensado para juegos o aplicaciones que se tienen que ver en pantalla completa.
 - **browser**: se muestra como en el navegador.
- **orientation**: aquí le vamos a indicar la orientación en la que se tiene que mostrar la aplicación, y los posibles valores son **portrait** y **landscape**.
- **background_color**: indica el color que se va a aplicar a la pantalla de **splash**.
- **theme_color**: indica el color que se va a mostrar el toolbar, en la pantalla del gestor de aplicaciones...
- **icons**: esta propiedad recibe como valor un array de iconos que representan nuestra aplicación. Estos iconos son objetos JS en los que vamos a poner las siguientes propiedades:
 - **src**: ruta hasta el ícono.
 - **sizes**: tamaño del ícono (512x512).
 - **type**: el tipo de archivo (**image/png**).

Dentro de los proyectos de angular, este archivo es el **public/manifest.webmanifest** y se importa en el **index.html**.

4.3. Service Workers

Los **Services Workers** son archivos JS que se ejecutan en segundo plano, y no necesitan que la página web esté abierta para que sigan funcionando.

Al poder ejecutarlos sin necesidad de que la tengamos la web abierta, esto nos da la posibilidad de recibir notificaciones push, pero también nos permite hacer más cosas.

Los service workers actúan como proxy, y cada petición que se va a realizar para descargarnos algún archivo desde el servidor (al abrir la página web nos descargamos varios archivos) pasa por él.

Para que los service workers funcionen, la aplicación se tiene que servir usando el protocolo HTTPS para evitar que nos hagan un **man in the middle** y lean todos los paquetes que pasan por ahí. Existe una excepción, y es si utilizamos el **localhost**.

Los service workers interactúan con nuestra aplicación mediante eventos, pero no todos los tipos de eventos. Al estar ejecutándose en segundo plano, no puede acceder al DOM y por tanto no puede escuchar los eventos relacionados con él. Algunos de los eventos que vamos a poder detectar son:

- Fetch: se activa cuando se realiza una petición HTTP para pedir algún recurso al servidor.
- Interacciones con las notifications: se activan cuando interactuamos con las notificaciones que hemos recibido, por ejemplo al cerrarlas o al pulsar alguna de las opciones que nos muestran.
- Push Notifications: se activa cuando recibimos una mensaje Push.
- Eventos del ciclo de vida: se activa con algunos eventos del ciclo de vida de los service workers.



Los **service workers** no funcionan con el servidor de desarrollo de Angular, es decir el comando **ng s**, por lo que tenemos que utilizar alguna otra librería como **http-serve**.

<https://angular.io/guide/service-worker-getting-started#serving-with-http-server>

Dentro de Angular, no nos tenemos que preocupar por tener que crear la lógica que hace uso de los service workers, sino que esto lo controlaremos desde un archivo de configuración JSON, haciendo que todo sea mucho más sencillo.

Este archivo se crea al añadir la librería de **@angular/pwa**, y lo podemos encontrar en **ngsw-config.json**.

Las opciones que podemos llenar son las siguientes:

- **index**: se especifica el archivo que se sirve como página inicial, es decir, nuestro **index.html**.
- **assetGroups**: assets o recursos estáticos que son parte de la aplicación. Se pueden cargar desde el mismo origen, algún CDN o URLs externas. Aquí definimos grupos de assets, cada uno con su propia configuración. Dentro de cada grupo de assets nos encontramos:
 - **name**: este nombre es muy importante ya que identifica el grupo de assets entre las distintas versiones de la configuración (por ejemplo a la hora de actualizar).

- **installMode:** indica como se tienen que cachear los assets inicialmente.
 - Si usamos **prefetch** le estamos indicando que cachee todos los recursos al arrancar la aplicación de tal forma que esto siempre estará disponible incluso cuando estemos sin conexión.
 - En caso de utilizar **lazy**, le estaríamos indicando que solo cachee los recursos cuando se pidan, de tal forma que si nos quedamos sin conexión y no se había pedido hasta ahora un recurso, no tendremos acceso a él hasta que volvamos a tener conexión.
- **updateMode:** tiene las mismas opciones que el `installMode`. Esta opción indica como se tienen que cachear los recursos cuando se descubre una nueva versión de la aplicación. El **prefetch** cachea inmediatamente todos los assets que hayan cambiado, y el **lazy** (solo funciona si en `installMode` hemos usado `lazy` también) espera a cachear las nuevas versiones de los recursos cuando se pidan estos otra vez.
- **resources:** tenemos dos opciones:
 - **files** para indicar que archivos del fichero de distribución queremos cachear.
 - **urls** para indicar que archivos que se encuentran en URLs externas queremos cachear.
- **dataGroups:** al contrario que los assets, los datos no están versionados, son items dinámicos como los datos que obtenemos de una API, así que tienen sus propias opciones a la hora de cachearlos. Las opciones que encontramos dentro de estos objetos son:
 - **name:** al igual que en los assets, aquí va el nombre con el que identificamos los datos cacheados aquí.
 - **urls:** es una lista de patrones de URLs a cachear. Cuando se realice una petición HTTP, si coincide con alguno de los patrones entonces se cachea.
 - **version:** es un número que solo cambiaremos cuando la API a la que pedimos los datos se haya actualizado y los cambios realizados sobre ella sean incompatibles con la antigua API.
 - **cacheConfig:** en este objeto de configuración es donde vamos a establecer las políticas de la caché:
 - **maxSize:** el número máximo de entradas en la caché.
 - **maxAge:** el tiempo máximo que se van a mantener las entradas en la caché. Utilizaremos sufijos para indicar la unidad de tiempo (d - días, h - horas, m - minutos, s - segundos y u - milisegundos). Por ejemplo, **6h30s**.
 - **timeout:** el tiempo máximo que el SW va a esperar a que se obtenga la respuesta a una petición. Si se termina este tiempo, entonces se devuelve la respuesta cacheada. Se usa el mismo formato que en **maxAge**.
 - **strategy:** el tipo de estrategia de cacheado a seguir:
 - **freshness:** pide el recurso a la API o servidor, y si no puede obtenerlo o se termina el tiempo de espera entonces devuelve el cacheado. Esta es una buena estrategia para esos datos que cambian frecuentemente, ya que siempre los tendremos lo más actualizados posible.
 - **performance:** devuelve lo de la cache, y si no está entonces pide el recurso al servidor. Se suele usar para recursos que no cambian demasiado a lo largo del tiempo.

4.4. Lab: Progressive Web Apps (PWAs)

En este laboratorio vamos a crear una pequeña aplicación para publicar y listar ofertas de trabajo que vamos a convertir en una PWA con cacheado de assets y datos.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-18-pwa-lab
? Would you like to add Angular routing? N
? Which stylesheet format would you like to use? CSS
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado, crearemos unos componentes, un servicio y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-18-pwa-lab
$ ng g c inicio
$ ng g c nueva-oferta
$ ng g c detalle-oferta
$ ng g c oferta
$ ng g s ofertas
$ ng s
```

Vamos a empezar creando las rutas por las que vamos a poder navegar en la aplicación:

- / → Inicio
- /nueva-oferta → Formulario para crear una nueva oferta
- /ofertas/:id → Información completa de una oferta dado su identificador

Por tanto, vamos al archivo **app.routes.ts** en la carpeta **app** en el que vamos a declarar estas rutas.

/angular-18-pwa-lab/src/app/app.routes.ts

```
import { Routes } from '@angular/router';
import { InicioComponent } from './inicio/inicio.component';
import { NuevaOfertaComponent } from './nueva-oferta/nueva-oferta.component';
import { DetalleOfertaComponent } from './detalle-oferta/detalle-oferta.component';

export const routes: Routes = [
  { path: '', component: InicioComponent },
  { path: 'nueva-oferta', component: NuevaOfertaComponent },
  { path: 'ofertas/:id', component: DetalleOfertaComponent },
];
```

Ahora vamos al archivo de configuración **app.config.ts** para añadir el provider de Http Client que vamos a utilizar.

/angular-18-pwa-lab/src/app/app.config.ts

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

import { routes } from './app.routes';
import { provideHttpClient } from '@angular/common/http';

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

Con esto, ya podemos centrarnos en ir desarrollando los componentes y el servicio.

Vamos a empezar por añadir el **router-outlet** y los enlaces para navegar por las rutas en el componente App.

/angular-18-pwa-lab/src/app/app.component.html

```
<ul>
  <li>
    <a [routerLink]="/">Inicio</a>
  </li>
  <li>
    <a [routerLink]="/nueva-oferta">Crear oferta</a>
  </li>
</ul>

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

Como estamos utilizando la directiva **routerLink** para la navegación, necesitamos importarla en el componente.

/angular-18-pwa-lab/src/app/app.component.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    RouterOutlet,
    RouterLink,
  ],
  templateUrl: './app.component.html',
```

```

        styleUrl: './app.component.css'
    })
export class AppComponent {
}

```

Ahora vamos a ir al servicio a crear los 3 métodos que vamos a necesitar:

- `getOfertas`: petición GET para obtener todas las ofertas de trabajo

`/angular-18-pwa-lab/src/app/ofertas.service.ts`

```

import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { map, Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class OfertasService {
  URL: string = 'https://fluentjobs-fa22e.firebaseio.com/ejemplo-pwa/ofertas'

  constructor(private http: HttpClient) { }

  getOfertas(): Observable<any> {
    return this.http.get(`${this.URL}.json`)
      .pipe(
        map((objOfertas: any) => {
          const arrOfertas = []

          for (let id in objOfertas) {
            arrOfertas.push({...objOfertas[id], id})
          }

          return arrOfertas
        })
      )
  }
}

```

- `getOferta`: petición GET para obtener una oferta de trabajo dado su identificador

`/angular-18-pwa-lab/src/app/ofertas.service.ts`

```

import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { map, Observable } from 'rxjs';

@Injectable({

```

```

    providedIn: 'root'
})
export class OfertasService {
  URL: string = 'https://fluentjobs-fa22e.firebaseio.com/ejemplo-pwa/ofertas'

  constructor(private http: HttpClient) { }

  getOfertas(): Observable<any> {
    return this.http.get(`.${this.URL}.json`)
      .pipe(
        map((objOfertas: any) => {
          const arrOfertas = []

          for (let id in objOfertas) {
            arrOfertas.push({...objOfertas[id], id})
          }

          return arrOfertas
        })
      )
  }

  getOferta(id: string): Observable<any> {
    return this.http.get(`.${this.URL}/${id}.json`)
  }

}

```

- `crearOferta`: petición POST para crear una nueva oferta de trabajo

`/angular-18-pwa-lab/src/app/ofertas.service.ts`

```

import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { map, Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class OfertasService {
  URL: string = 'https://fluentjobs-fa22e.firebaseio.com/ejemplo-pwa/ofertas'

  constructor(private http: HttpClient) { }

  getOfertas(): Observable<any> {
    return this.http.get(`.${this.URL}.json`)
      .pipe(
        map((objOfertas: any) => {
          const arrOfertas = []

          for (let id in objOfertas) {

```

```

        arrOfertas.push({...objOfertas[id], id})
    }

    return arrOfertas
})
)
}

getOferta(id: string): Observable<any> {
    return this.http.get(`.${this.URL}/${id}.json`)
}

crearOferta(nuevaOferta: any): Observable<any> {
    return this.http.post(`.${this.URL}.json`, nuevaOferta)
}
}

```

El siguiente paso es crear el componente **Oferta** que usaremos para listar en la página de **Inicio** cada una de las ofertas.

En este componente solo vamos a mostrar la imagen de la empresa y el título de esta.

El título lo envolveremos con la etiqueta **a** para poder navegar al detalle de la oferta al pulsar sobre el.

/angular-18-pwa-lab/src/app/oferta/oferta.component.html

```

<div>
    <img [src]="oferta.urlImagen" alt="Logo de {{oferta.empresa}}">
    <a [routerLink]=["/ofertas", oferta.id]>
        <h3>{{oferta.titulo}}</h3>
    </a>
</div>

```

Ese objeto **oferta** al cual estamos accediendo a la imagen, título, id y empresa lo obtendremos desde el exterior (desde el componente de **Inicio**), por lo que tenemos que declarar la propiedad con un **@Input**.

También hay que importar el **RouterLink** ya que lo estamos utilizando en la plantilla.

/angular-18-pwa-lab/src/app/oferta/oferta.component.ts

```

import { Component, Input, OnInit } from '@angular/core';
import { RouterLink } from '@angular/router';

@Component({
    selector: 'app-oferta',
    standalone: true,
    imports: [
        RouterLink,

```

```

  ],
  templateUrl: './oferta.component.html',
  styleUrls: ['./oferta.component.css'
})
export class OfertaComponent implements OnInit {
  @Input() oferta: any = {}
  constructor() { }

  ngOnInit(): void {
  }
}

```

Ahora nos vamos a ir al componente de Inicio, donde vamos a declarar la lista de ofertas que pediremos desde el servicio que hemos creado anteriormente.

/angular-18-pwa-lab/src/app/inicio/inicio.component.ts

```

import { Component, OnInit } from '@angular/core';
import { OfertasService } from '../ofertas.service';

@Component({
  selector: 'app-inicio',
  standalone: true,
  imports: [],
  templateUrl: './inicio.component.html',
  styleUrls: ['./inicio.component.css'
})
export class InicioComponent implements OnInit {
  listaOfertas: Array<any> = []

  constructor(private ofertas: OfertasService) { }

  ngOnInit(): void {
    this.ofertas.getOfertas()
      .subscribe((datos: Array<any>) => {
        this.listaOfertas = datos
      })
  }
}

```

Como nos estamos suscribiendo al **getOfertas**, vamos a crear una propiedad para guardar la suscripción, y nos vamos a desuscribir en el método **ngOnDestroy**.

/angular-18-pwa-lab/src/app/inicio/inicio.component.ts

```

import { Component, OnDestroy, OnInit } from '@angular/core';
import { OfertasService } from '../ofertas.service';
import { Subscription } from 'rxjs';

@Component({

```

```

        selector: 'app-inicio',
        standalone: true,
        imports: [],
        templateUrl: './inicio.component.html',
        styleUrls: ['./inicio.component.css']
    })
export class InicioComponent implements OnInit, OnDestroy {
    listaOfertas: Array<any> = []
    ofertasSubs: Subscription | null = null

    constructor(
        private ofertas: OfertasService
    ) { }

    ngOnInit(): void {
        this.ofertasSubs = this.ofertas.getOfertas()
            .subscribe((datos: Array<any>) => {
                this.listaOfertas = datos
            })
    }
    ngOnDestroy() {
        this.ofertasSubs?.unsubscribe()
    }
}

```

Tenemos que importar el componente **OfertaComponent** ya que lo vamos a utilizar en la plantilla.

/angular-18-pwa-lab/src/app/inicio/inicio.component.ts

```

import { Component, OnDestroy, OnInit } from '@angular/core';
import { OfertasService } from '../ofertas.service';
import { Subscription } from 'rxjs';
import { OfertaComponent } from '../oferta/oferta.component';

@Component({
    selector: 'app-inicio',
    standalone: true,
    imports: [
        OfertaComponent,
    ],
    templateUrl: './inicio.component.html',
    styleUrls: ['./inicio.component.css'
})
export class InicioComponent implements OnInit, OnDestroy {
    listaOfertas: Array<any> = []
    ofertasSubs: Subscription | null = null

    constructor(
        private ofertas: OfertasService
    ) { }

```

```

ngOnInit(): void {
  this.ofertasSubs = this.ofertas.getOfertas()
    .subscribe((datos: Array<any>) => {
      this.listaOfertas = datos
    })
}

ngOnDestroy() {
  this.ofertasSubs?.unsubscribe()
}
}

```

Ya podemos iterar la **listaOfertas** en la plantilla y crear un componente **Oferta** por cada una de las que haya en la lista.

/angular-18-pwa-lab/src/app/inicio/inicio.component.html

```

<h2>Ofertas de trabajo</h2>

@for (oferta of listaOfertas; track $index) {
  <app-oferta [oferta]="oferta"></app-oferta>
}

```

Ahora nos vamos a ir al componente **DetalleOferta** en el que vamos a mostrar toda la información de cada oferta.

/angular-18-pwa-lab/src/app/detalle-oferta/detalle-oferta.component.html

```

<h2>{{oferta.titulo}}</h2>
<img [src]="oferta.urlImagen" alt="Logo de {{oferta.empresa}}">
<p>{{oferta.descripcion}}</p>
<p>Empresa: {{oferta.empresa}}</p>
<p>Salario: {{oferta.salario | currency:'EUR'}}</p>
<p>Ciudad: {{oferta.ciudad | titlecase}}</p>

```

Ahora dentro del TypeScript tenemos que obtener la oferta, primero sacando el identificador del parámetro de la ruta usando el servicio de **ActivatedRoute**, y después teniendo ya el id, llamando al método **getOferta** del servicio de **Ofertas**.

También importaremos los pipes que hemos utilizado en la plantilla.

/angular-18-pwa-lab/src/app/detalle-oferta/detalle-oferta.component.ts

```

import { CurrencyPipe, TitleCasePipe } from '@angular/common';
import { Component, OnInit } from '@angular/core';
import { OfertasService } from '../ofertas.service';
import { ActivatedRoute, ParamMap } from '@angular/router';

```

```

@Component({
  selector: 'app-detalle-oferta',
  standalone: true,
  imports: [
    CurrencyPipe,
    TitleCasePipe,
  ],
  templateUrl: './detalle-oferta.component.html',
  styleUrls: ['./detalle-oferta.component.css'
})
export class DetalleOfertaComponent implements OnInit {
  oferta: any = {}

  constructor(
    private ofertas: OfertasService,
    private route: ActivatedRoute,
  ) { }

  ngOnInit(): void {
    this.route.paramMap
      .subscribe((params: ParamMap) => {
        const id = params.get('id')
        if (id) {
          this.ofertas.getOferta(id)
            .subscribe((datos: any) => {
              this.oferta = datos
            })
        }
      })
  }
}

```

Por último, vamos a abrir el TypeScript del componente de **NuevaOferta** donde vamos a definir un formulario con los campos para llenar una oferta de trabajo:

- título
- descripción
- empresa
- salario
- ciudad
- urlImagen

Y como vamos a utilizar un formulario reactivo, también vamos a dejar importado el **ReactiveFormsModule**.

/angular-18-pwa-lab/src/app/detalle-oferta/detalle-oferta.component.ts

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

```

import { FormControl, FormGroup, ReactiveFormsModule } from '@angular/forms';
import { OfertasService } from '../ofertas.service';
import { Router } from '@angular/router';

@Component({
  selector: 'app-nueva-oferta',
  standalone: true,
  imports: [
    ReactiveFormsModule,
  ],
  templateUrl: './nueva-oferta.component.html',
  styleUrls: ['./nueva-oferta.component.css']
})
export class NuevaOfertaComponent implements OnInit {
  formOferta: FormGroup

  constructor(
    private ofertas: OfertasService,
    private router: Router,
  ) {
    this.formOferta = new FormGroup({
      titulo: new FormControl(''),
      descripcion: new FormControl(''),
      empresa: new FormControl(''),
      salario: new FormControl(0),
      ciudad: new FormControl(''),
      urlImagen: new FormControl(''),
    })
  }

  ngOnInit(): void {
  }
}

```

Ahora vamos a añadir una función **guardar** en la que llamaremos al método **crearOferta** del servicio de **Ofertas** pasándole los datos obtenidos del formulario, y guardaremos la suscripción en una propiedad para desuscribirnos en el método **ngOnDestroy**.

/angular-18-pwa-lab/src/app/detalle-oferta/detalle-oferta.component.ts

```

import { Component, OnDestroy, OnInit } from '@angular/core';
import { FormControl, FormGroup, ReactiveFormsModule } from '@angular/forms';
import { OfertasService } from '../ofertas.service';
import { Router } from '@angular/router';
import { Subscription } from 'rxjs';

@Component({
  selector: 'app-nueva-oferta',
  standalone: true,
  imports: [

```

```

        ReactiveFormsModule,
    ],
    templateUrl: './nueva-oferta.component.html',
    styleUrls: ['./nueva-oferta.component.css']
})
export class NuevaOfertaComponent implements OnInit, OnDestroy {
  formOferta: FormGroup
  nuevaOfertaSubs: Subscription | null = null

  constructor(
    private ofertas: OfertasService,
    private router: Router,
  ) {
    this.formOferta = new FormGroup({
      titulo: new FormControl(''),
      descripcion: new FormControl(''),
      empresa: new FormControl(''),
      salario: new FormControl(0),
      ciudad: new FormControl(''),
      urlImagen: new FormControl(''),
    })
  }

  ngOnInit(): void {
  }

  guardar(): void {
    this.nuevaOfertaSubs = this.ofertas.crearOferta(this.formOferta.value)
      .subscribe(() => {
        this.router.navigate(['/'])
      })
  }

  ngOnDestroy(): void {
    this.nuevaOfertaSubs?.unsubscribe()
  }
}

```

El siguiente paso es crear el formulario en la plantilla y enlazarlo con el que acabamos de crear en el TypeScript.

`/angular-18-pwa-lab/src/app/nueva-oferta/nueva-oferta.component.html`

```

<h2>Nueva oferta de trabajo</h2>

<form [formGroup]="formOferta" (ngSubmit)="guardar()">
  <div>
    <label for="titulo">Título</label>
    <input type="text" id="titulo" formControlName="titulo">
  </div>

```

```

<div>
  <label for="descripcion">Descripción</label>
  <input type="text" id="descripcion" formControlName="descripcion">
</div>
<div>
  <label for="empresa">Empresa</label>
  <input type="text" id="empresa" formControlName="empresa">
</div>
<div>
  <label for="salario">Salario</label>
  <input type="number" id="salario" formControlName="salario">
</div>
<div>
  <label for="ciudad">Ciudad</label>
  <select id="ciudad" formControlName="ciudad">
    <option value="madrid">Madrid</option>
    <option value="barcelona">Barcelona</option>
    <option value="sevilla">Sevilla</option>
    <option value="valencia">Valencia</option>
    <option value="full-remoto">Full Remoto</option>
  </select>
</div>
<div>
  <label for="urlImagen">Url imagen</label>
  <input type="text" id="urlImagen" formControlName="urlImagen">
</div>
<button type="submit">Guardar</button>
</form>

```

Con esto ya tenemos la aplicación, ahora nos toca transformarla en una PWA.

El primer paso es añadir la librería que se encarga de esto con el siguiente comando:

```
$ ng add @angular/pwa
```

```
The package @angular/pwa@18.2.21 will be installed and executed.
Would you like to proceed? Y
```

Con este comando se ha generado el archivo del service worker (**ngsw-config.json**) y el archivo del manifest (**manifest.webmanifest**) que son los que nos interesan modificar.

Vamos a empezar cambiando alguna opción del archivo de **manifest** para modificar los estilos de la pantalla de splash.

El nombre de la aplicación lo vamos a cambiar a **Jobs, tu futuro en la palma de tu mano**. Este nombre es el que aparece al abrir la aplicación en un dispositivo móvil en la pantalla de splash.

También vamos a cambiar el **short_name** a **jobs**. Este otro nombre es el que aparece debajo del icono de la aplicación.

Y el último cambio que vamos a hacer es el de **theme_color** y **background_color** para cambiar los colores que se muestran en la pantalla de splash y el fondo de la aplicación.

/angular-18-pwa-lab/src/manifest.webmanifest

```
{  
  "name": "Jobs, tu futuro en la palma de tu mano",  
  "short_name": "jobs",  
  "theme_color": "#cccccc",  
  "background_color": "#000000",  
  "display": "standalone",  
  "scope": "./",  
  "start_url": "./",  
  "icons": [  
    {  
      "src": "icons/icon-72x72.png",  
      "sizes": "72x72",  
      "type": "image/png",  
      "purpose": "maskable any"  
    },  
    {  
      "src": "icons/icon-96x96.png",  
      "sizes": "96x96",  
      "type": "image/png",  
      "purpose": "maskable any"  
    },  
    {  
      "src": "icons/icon-128x128.png",  
      "sizes": "128x128",  
      "type": "image/png",  
      "purpose": "maskable any"  
    },  
    {  
      "src": "icons/icon-144x144.png",  
      "sizes": "144x144",  
      "type": "image/png",  
      "purpose": "maskable any"  
    },  
    {  
      "src": "icons/icon-152x152.png",  
      "sizes": "152x152",  
      "type": "image/png",  
      "purpose": "maskable any"  
    },  
    {  
      "src": "icons/icon-192x192.png",  
      "sizes": "192x192",  
      "type": "image/png",  
      "purpose": "maskable any"  
    },  
    {  
      "src": "icons/icon-312x312.png",  
      "sizes": "312x312",  
      "type": "image/png",  
      "purpose": "maskable any"  
    }  
  ]  
}
```

```
        "src": "icons/icon-384x384.png",
        "sizes": "384x384",
        "type": "image/png",
        "purpose": "maskable any"
    },
    {
        "src": "icons/icon-512x512.png",
        "sizes": "512x512",
        "type": "image/png",
        "purpose": "maskable any"
    }
]
```

El otro cambio que vamos a realizar es el de cambiar los iconos por unos propios de nuestra aplicación.

Para ello, vamos a coger la siguiente imagen y vamos a subirla en la página <https://manifest-gen.netlify.app/>, la cual nos va a generar el archivo manifest con la carpeta de iconos.



A nosotros solo nos interesa la carpeta de iconos, por lo que vamos a copiar esa carpeta y la vamos a pegar dentro de **public**, sobreescribiendo todos los iconos que venían por defecto.

Una vez hecho esto, vamos a comprobar que nuestra aplicación ya es una PWA y que podemos instalarla en nuestro dispositivo móvil.

El primer paso para poder instalarla en nuestro dispositivo móvil es generar el proyecto para producción con el siguiente comando:

```
$ ng build
```

Una vez construido el proyecto, vamos a entrar en la carpeta **dist/angular-18-pwa-lab/browser** y vamos a levantar un servidor utilizando la dependencia **http-serve**.

```
$ cd dist/angular-18-pwa-lab/browser
$ npx http-serve
```

Ya está levantada la aplicación en <http://localhost:8080/>. Si accedemos a la página veremos nuestra aplicación.

Ahora queremos verla en nuestro dispositivo móvil para ver el mensaje que nos permitirá instalarla dentro de este como si fuese una aplicación.

Para ello vamos a acceder, desde Google Chrome, a <chrome://inspect/#devices> para poder conectar nuestro móvil al <http://localhost:8080/> de nuestro ordenador y así poder ver la misma web en los dos sitios.

Tenemos que habilitar la **Depuración USB** del dispositivo móvil desde **Ajustes > Herramientas del desarrollador** y conectar el móvil al ordenador.

Al hacer los pasos anteriores, debe de salirnos un popup en nuestro móvil pidiendo que le demos acceso a nuestro ordenador para conectarse.

Depuración USB habilitada

Toca aquí para desactivar la depuración USB

¿Permitir depuración por USB?

La huella digital de tu clave RSA es:

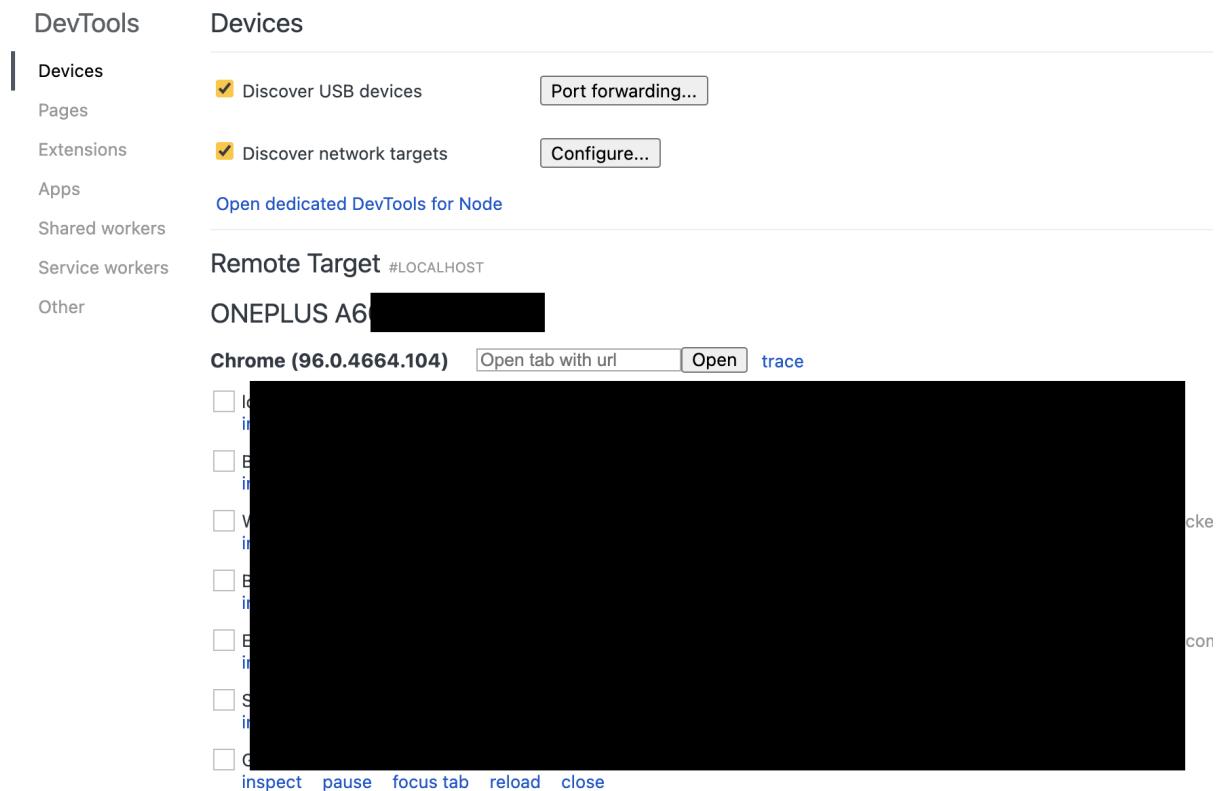
A: [REDACTED] B:
: [REDACTED] .0

Permitir siempre desde este ordenador

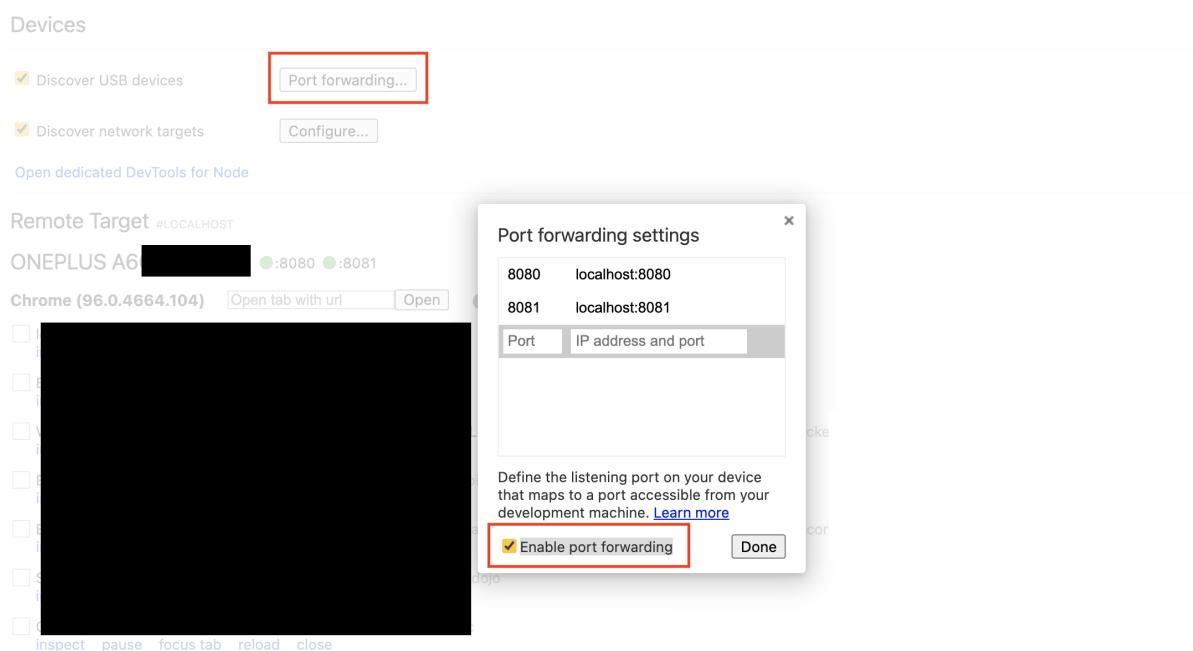
CANCELAR PERMITIR



Una vez aceptado el permiso, en la página de los dispositivos de Chrome (<chrome://inspect/#devices>), debería de salir el nombre de nuestro dispositivo y al entrar en el navegador de nuestro móvil, también deberíamos de poder ver la lista de páginas que tenemos abiertas en este.



Debemos de asegurarnos de que tenemos activado el **Port forwarding** pulsando sobre dicho botón y seleccionando la opción de **Enable port forwarding**.



Ya podemos entrar en el navegador del dispositivo móvil en la URL <http://localhost:8080/>, donde debería de salirnos la aplicación que se está sirviendo en ese mismo puerto dentro del ordenador.

Deberíamos de poder ver la aplicación en el móvil. Y si queremos instalar la aplicación, debemos

darle a los 3 puntos que se encuentra en la esquina superior derecha y seleccionar **Añadir a la pantalla de inicio**.

Al pulsar sobre la alerta se abre un popup del sistema y nos pregunta si queremos instalar la aplicación. Vamos a darle a **Instalar**.



- [Inicio](#)
- [Crear oferta](#)

Ofertas de trabajo

Empresa 1

Diseñador/a Gráfico/a in Madrid

Instalar aplicación



Jobs, tu futuro en la palma de tu mano

<http://localhost:8080>

[Cancelar](#)

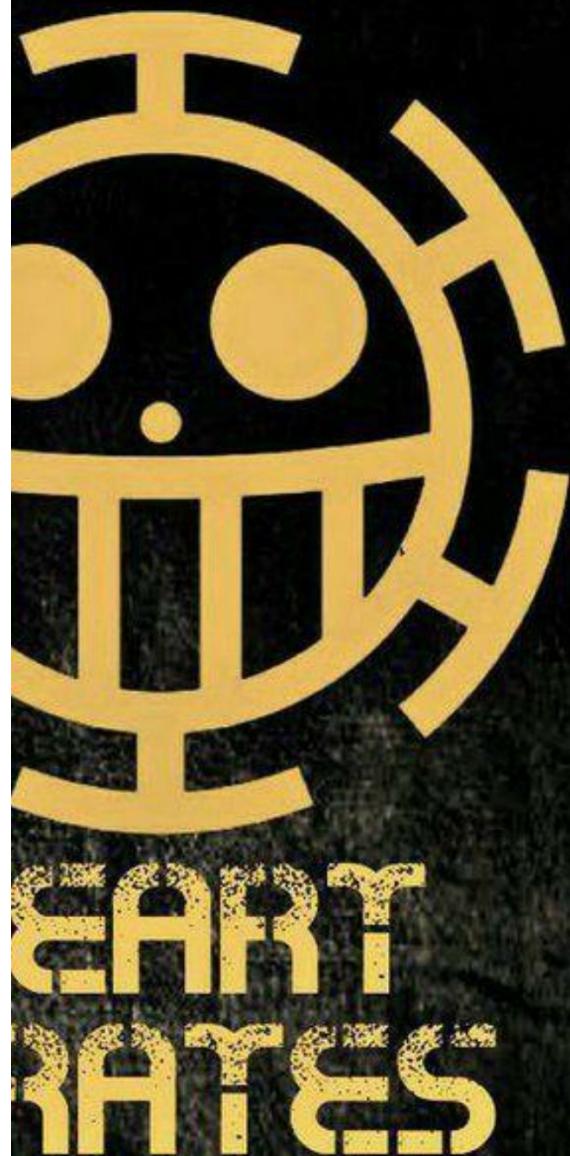
[Instalar](#)



Una vez termina de instalarse, ya la podremos ver en nuestro **HomeScreen**.

15:47

34 %



jobs



Al abrirla, saldrá primero la pantalla de splash con el icono, nombre y colores que habíamos definido en el archivo de **manifest**.

15:47

– 🔔 ✈ 34 % 🔋



Jobs, tu futuro en la palma de tu mano



Pues ya tenemos algunas de las características de las aplicaciones nativas solo añadiendo el paquete de las PWAs en nuestro proyecto:

- Aplicación instalable
- Pantalla de splash

Vamos a añadir una pequeña mejora en nuestro archivo **package.json**. En la sección de scripts, vamos a crear un nuevo script **start:pwa** que se va a encargar de lanzar los 3 comandos que hemos lanzado anteriormente para levantar la aplicación.

/angular-18-pwa-lab/package.json

```
{
  "name": "angular-18-pwa-lab",
  "version": "0.0.0",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "watch": "ng build --watch --configuration development",
    "test": "ng test",
    "start:pwa": "ng build && cd dist/angular-18-pwa-lab/browser && npx http-serve"
  },
  "private": true,
  "dependencies": {
    "@angular/animations": "^18.2.0",
    "@angular/common": "^18.2.0",
    "@angular/compiler": "^18.2.0",
    "@angular/core": "^18.2.0",
    "@angular/forms": "^18.2.0",
    "@angular/platform-browser": "^18.2.0",
    "@angular/platform-browser-dynamic": "^18.2.0",
    "@angular/router": "^18.2.0",
    "@angular/service-worker": "^18.2.0",
    "rxjs": "~7.8.0",
    "tslib": "^2.3.0",
    "zone.js": "~0.14.10"
  },
  "devDependencies": {
    "@angular-devkit/build-angular": "^18.2.21",
    "@angular/cli": "^18.2.21",
    "@angular/compiler-cli": "^18.2.0",
    "@types/jasmine": "~5.1.0",
    "jasmine-core": "~5.2.0",
    "karma": "~6.4.0",
    "karma-chrome-launcher": "~3.2.0",
    "karma-coverage": "~2.2.0",
    "karma-jasmine": "~5.1.0",
    "karma-jasmine-html-reporter": "~2.1.0",
    "typescript": "~5.5.2"
  }
}
```

```
}
```

A partir de ahora, cuando queramos construir la aplicación y servirla lanzaremos el siguiente comando, en lugar de lanzar los 3 comandos que habíamos lanzado antes:

```
$ npm run start:pwa
```

Vamos a ver ahora la parte de como cachear los distintos elementos de la aplicación como los estilos, los scripts, las imágenes y los datos.

Vamos a empezar por añadir por ejemplo la librería de bootstrap directamente en el **index.html**, por lo que lo vamos a abrir y añadiremos el enlace `<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3" crossorigin="anonymous">`.

/angular-18-pwa-lab/src/index.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Angular18PwaLab</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3" crossorigin="anonymous">
  <link rel="manifest" href="manifest.webmanifest">
  <meta name="theme-color" content="#1976d2">
</head>
<body>
  <app-root></app-root>
  <noscript>Please enable JavaScript to continue using this application.</noscript>
</body>
</html>
```

La idea es ver que los archivos principales de la aplicación ya se están cacheando según está definido en el archivo de **ngsw-config.json**, pero los estilos de Bootstrap no.

Vamos al navegador y vamos a cargar de nuevo la página. Una vez hecho esto, deberíamos de ver la aplicación con los estilos de Bootstrap aplicados.

Ahora vamos a deshabilitar la conexión a internet desde las herramientas del desarrollador, en la pestaña de **Application > Service Workers > Offline**.

Al volver a refrescar la página, veremos que no se han cargado los datos de las ofertas de trabajo.

También veremos que los estilos de Bootstrap tampoco los ha cacheado el service worker, sino que este solo ha dejado en la caché aquellos archivos que tenemos en **ngsw-config.json**, es decir, la página de HTML, el manifest, los scripts y css de la aplicación, los archivos de las carpetas de

assets...

Vamos a añadir los estilos de Bootstrap en la caché, y como estos estilos se descargan desde un CDN, entonces tendremos que meterlo dentro de **resources.urls**. Al ser parte de los estilos de la aplicación, lo meteremos en el assetGroup que tiene como nombre **app**.

/angular-18-pwa-lab/ngsw-config.json

```
{
  "$schema": "./node_modules/@angular/service-worker/config/schema.json",
  "index": "/index.html",
  "assetGroups": [
    {
      "name": "app",
      "installMode": "prefetch",
      "resources": {
        "files": [
          "/favicon.ico",
          "/index.csr.html",
          "/index.html",
          "/manifest.webmanifest",
          "/*.css",
          "/*.js"
        ],
        "urls": [
          "https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css"
        ]
      }
    },
    {
      "name": "assets",
      "installMode": "lazy",
      "updateMode": "prefetch",
      "resources": {
        "files": [
          "**/*.(svg|cur|jpg|jpeg|png|apng|webp|avif|gif|otf|ttf|woff|woff2)"
        ]
      }
    }
  ]
}
```

Ahora, al realizar los mismos pasos que antes, deberíamos de ver que los estilos de Bootstrap si que se aplican, ya que los hemos dejado cacheados con el **service worker**.

Ahora nos toca cachear los datos de las ofertas de trabajo que pedimos a nuestro backend (en este caso a Firebase). Es decir, la petición GET.

En este caso, al ser datos dinámicos que van a cambiar, tendremos que añadirlos en la sección de **dataGroups**.

Vamos a añadir un objeto con el nombre de **datos-ofertas** y la url <https://fluentjobs-fa22e.firebaseio.com/ejemplo-pwa/ofertas>.

La estrategia a utilizar esta vez es **freshness**, ya que queremos obtener siempre la versión más reciente de estos datos. Además, le vamos a poner que mantenga estos datos cacheados durante 2 minutos añadiéndole **1m** como valor de **maxAge**, de esta forma podemos probarlo sin esperar una eternidad a que se eliminen.

/angular-18-pwa-lab/ngsw-config.json

```
{
  "$schema": "./node_modules/@angular/service-worker/config/schema.json",
  "index": "/index.html",
  "assetGroups": [
    {
      "name": "app",
      "installMode": "prefetch",
      "resources": {
        "files": [
          "/favicon.ico",
          "/index.html",
          "/manifest.webmanifest",
          "/*.css",
          "/*.js"
        ],
        "urls": [
          "https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css"
        ]
      }
    },
    {
      "name": "assets",
      "installMode": "lazy",
      "updateMode": "prefetch",
      "resources": {
        "files": [
          "/assets/**",
          "/*.svg|cur|jpg|jpeg|png|apng|webp|avif|gif|otf|ttf|woff|woff2"
        ]
      }
    }
  ],
  "dataGroups": [
    {
      "name": "datos-ofertas",
      "urls": [
        "https://fluentjobs-fa22e.firebaseio.com/ejemplo-pwa/ofertas"
      ],
      "cacheConfig": {
        "strategy": "freshness",
        "maxAge": "2m",
      }
    }
  ]
}
```

```
        "maxSize": 5
    }
}
]
}
```

Si volvemos a seguir los pasos anteriores, veremos que esta vez tras refrescar la página sin internet, ya se carga el listado de ofertas.

4.5. Notificaciones Push

Las **Notificaciones Push** son una de las novedades más importantes de las PWAs ya que es una de las principales características que tienen las aplicaciones nativas, y que no tenían las aplicaciones web hasta que se han implementado los service workers.

Estas notificaciones son aquellas que aparecen en las aplicaciones nativas, navegadores, aplicaciones de escritorio y que avisan a los usuarios de que algo interesante ha ocurrido en la aplicación, como por ejemplo, un aviso de que te ha llegado un email nuevo, alguien te ha mencionado en un tweet...

Para recibir las alertas no es necesario tener la aplicación abierta, ni siquiera el navegador, y esto es gracias a los service workers, que se quedan en segundo plano esperando a recibir estos mensajes.

Las notificaciones push se componen de dos estándares de los navegadores:

- **Notification API:** es una API que se encarga de mostrar las notificaciones en los dispositivos. Estas notificaciones son las nativas de cada dispositivo.
- **Push API:** es una API que permite enviar mensajes desde un servidor a los navegadores.

A la hora de configurar la apariencia que va a tener la notificación, podemos usar las siguientes opciones:

- **body:** es el cuerpo de la notificación.
- **image:** la ruta hasta una imagen que se mostrará en el cuerpo de la notificación.
- **icon:** la ruta hasta un ícono que aparecerá en la notificación.
- **badge:** aquí se le pasa la ruta al ícono que queremos usar para que aparezca en la barra de notificaciones del dispositivo. No hace falta que sea un ícono en blanco y negro porque en caso de no serlo, Android se ocupa de modificarlo.
- **tag:** etiqueta que le ponemos a la notificación para que pueda agrupar por tags las notificaciones que tenemos.
- **renotify:** con esto le indicamos si queremos que el dispositivo suene cada vez que nos llegue una notificación o si por el contrario solo queremos que suene la primera vez.
- **actions:** una lista de acciones que nos saldrán junto a la notificación y nos permiten hacer algo al pulsar sobre ellas. Esta lista contendrá un objeto por acción con las siguientes propiedades:
 - **action:** nombre que se le da a la acción para poder identificarla y hacer algo con ella.
 - **title:** el texto que aparece sobre la acción.

Para poder utilizar las notificaciones Push necesitamos usar un sistema de llaves públicas/privadas llamadas Vapid Keys. Estas claves las podemos generar con alguna librería de NPM como **web-push**.

Una vez generadas estas claves, tendremos que guardar en el cliente la clave pública y en el servidor tanto la pública como la privada, ya que estas claves son las que permitirán que el navegador reciba los mensajes para levantar las notificaciones.

En el **servidor** usaremos **setVapidDetails** de la librería **web-push** para setear las claves Vapid pública y privada, junto a un email o url de contacto.

A la hora de enviar una notificación, con esta misma librería vamos a utilizar el método **sendNotification** al que le vamos a pasar la suscripción del cliente al que enviar la notificación push, y un payload con los datos que queremos mostrar en la notificación.

El **cliente** tiene que generar un objeto con una serie de datos que representan una suscripción a los mensajes push de uno de los servidores de notificaciones push que usan los navegadores, como por ejemplo Google Chrome que usa el servicio de mensajería de Firebase.

En la aplicación de Angular usaremos el servicio de **SwPush** para pedir los permisos de la aplicación con el método **requestSubscription** que recibe como parámetro un objeto con **serverPublicKey: <la-clave-vapid-publica>**.

Esta función devuelve una promesa en la que obtendremos los datos de la suscripción que tenemos que guardar en el servidor.



Es posible que las notificaciones no estén soportadas en todos los navegadores. Por ejemplo, Brave parece ser que todavía no las soporta correctamente.

4.6. Lab: Notificaciones Push en PWAs

En este laboratorio vamos a ver como enviar notificaciones Push a los dispositivos que están usando nuestra aplicación al crear una nueva oferta de trabajo.

Para este laboratorio vamos a utilizar el proyecto de Angular que hemos creado en el laboratorio **Lab: Progressive Web Apps (PWAs)**.



Vamos a empezar creando una carpeta donde crearemos la parte del backend y donde vamos a copiar el proyecto de angular.

```
$ mkdir angular-18-pwa-notificaciones-push-lab
$ cd angular-18-pwa-notificaciones-push-lab
$ mkdir server-notificaciones-push
```

Dentro de la carpeta **angular-18-pwa-notificaciones-push-lab** vamos a copiar el proyecto **angular-18-pwa-lab** del laboratorio anterior.

Después de esto, tenemos que tener una estructura como la siguiente:

```
|- angular-18-pwa-notificaciones-push-lab
  |- server-notificaciones-push
  |- angular-18-pwa-lab
```

Una vez tenemos la estructura, vamos a inicializar el proyecto del servidor.

```
$ cd server-notificaciones-push
$ npm init -y
```

Ahora vamos a instalar las dependencias que vamos a necesitar, con los siguientes comandos:

- express: framework para crear el servidor
- cors: habilita el cors
- axios: realizar peticiones http
- web-push: se encarga de las notificaciones push
- dotenv: carga variables de entorno de un archivo .env
- nodemon: watcher para levantar el servidor con cada cambio

```
$ npm install express cors web-push dotenv axios
$ npm install -D nodemon
```

Con esto, ya deberíamos de tener un archivo **package.json** con la información y dependencias del proyecto.

```
{  
  "name": "server-notificaciones-push",  
  "version": "1.0.0",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \\"Error: no test specified\\" && exit 1"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "description": "",  
  "dependencies": {  
    "axios": "^1.12.2",  
    "cors": "^2.8.5",  
    "dotenv": "^17.2.3",  
    "express": "^5.1.0",  
    "web-push": "^3.6.7"  
  },  
  "devDependencies": {  
    "nodemon": "^3.1.10"  
  }  
}
```

Vamos a sustituir el script **test** por uno de **start** que va a arrancar la aplicación con **nodemon**.

```
{  
  "name": "server-notificaciones-push",  
  "version": "1.0.0",  
  "main": "index.js",  
  "scripts": {  
    "start": "nodemon src/app.js"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "description": "",  
  "dependencies": {  
    "axios": "^1.12.2",  
    "cors": "^2.8.5",  
    "dotenv": "^17.2.3",  
    "express": "^5.1.0",  
    "web-push": "^3.6.7"  
  },  
  "devDependencies": {  
    "nodemon": "^3.1.10"  
  }  
}
```

```
}
```

Ahora vamos a crear el archivo **app.js** dentro de una carpeta **src** en este proyecto.

Dentro de este archivo vamos a crear un servidor con **express** y lo pondremos a escuchar peticiones en el puerto **3000**.

```
/angular-18-pwa-notificaciones-push-lab/server-notificaciones-push/src/app.js
```

```
const express = require('express')

const app = express()

app.listen(3000, () => {
  console.log('Listening on http://localhost:3000')
})
```

Ahora vamos a utilizar los middlewares de **cors** y **express.json**, el primero para poder recibir peticiones desde otros dominios y el segundo para guardar el cuerpo de las peticiones POST, PUT... en una propiedad **body** de la petición.

```
/angular-18-pwa-notificaciones-push-lab/server-notificaciones-push/src/app.js
```

```
const express = require('express')
const cors = require('cors')

const app = express()

app.use(cors())
app.use(express.json())

app.listen(3000, () => {
  console.log('Listening on http://localhost:3000')
})
```

Al principio del todo, vamos a hacer que se carguen las variables de entorno con la dependencia de **dotenv**.

```
/angular-18-pwa-notificaciones-push-lab/server-notificaciones-push/src/app.js
```

```
require('dotenv').config()
const express = require('express')
const cors = require('cors')

const app = express()

app.use(cors())
app.use(express.json())
```

```
app.listen(3000, () => {
  console.log('Listening on http://localhost:3000')
})
```

Ahora vamos a generar nuestras Vapid Keys para poder enviar notificaciones push. Para generarlas necesitamos instalar de forma global la dependencia **web-push** y con ella generarlas.

```
$ npm install -g web-push
$ web-push generate-vapid-keys --json

{"publicKey":"BA1MZ0EcfvkMVhiKOqG6K1Z5W1XeS5kDk-
SA37HT37nRPr2NeKtKA_te7eSAqcwwaSC8sIkV14b0NdcApSCrVf3","privateKey":"89DfSIcqj5R7ngBkJV11FB0jdNliFZLptmEqz1X0mik"}
```

Vamos a crear un archivo **.env** dentro de la carpeta **server-notificaciones-push** y vamos a añadir dos variables con esos dos valores que hemos obtenido al generar las claves.

/angular-18-pwa-notificaciones-push-lab/server-notificaciones-push/.env

```
VAPID_PUBLIC_KEY=BA1MZ0EcfvkMVhiKOqG6K1Z5W1XeS5kDk-SA37HT37nRPr2NeKtKA_te7eSAqcwwaSC8sIkV14b0NdcApSCrVf3
VAPID_PRIVATE_KEY=89DfSIcqj5R7ngBkJV11FB0jdNliFZLptmEqz1X0mik
```

Ahora vamos a setear para la librería webpush estas claves Vapid, para ello, al ejecutar nuestro **app.js** vamos a llamar a la función **webpush.setVapidDetails** pasandole como parámetros una url o email de contacto, la clave pública y la privada.

/angular-18-pwa-notificaciones-push-lab/server-notificaciones-push/src/app.js

```
require('dotenv').config()
const express = require('express')
const cors = require('cors')
const webpush = require('web-push')

const app = express()

webpush.setVapidDetails('mailto:contact@jobs.com', process.env.VAPID_PUBLIC_KEY, process.env.VAPID_PRIVATE_KEY)

app.use(cors())
app.use(express.json())

app.listen(3000, () => {
  console.log('Listening on http://localhost:3000')
})
```

Ahora vamos a crear nuestro modelo de ofertas para poder crear este tipo de objetos y guardarlas en Firebase, donde las estabamos guardando desde el Front en el anterior laboratorio.

Aquí usaremos **axios** para realizar la petición POST.

/angular-18-pwa-notificaciones-push-lab/server-notificaciones-push/src/models/oferta.model.js

```
const axios = require('axios')
```

```

class Oferta {
  constructor(id, titulo, descripcion, empresa, salario, ciudad, urlImagen) {
    this.id = id
    this.titulo = titulo
    this.descripcion = descripcion
    this.empresa = empresa
    this.salario = salario
    this.ciudad = ciudad
    this.urlImagen = urlImagen
  }

  save() {
    return axios.post('https://fluentjobs-fa22e.firebaseio.com/ejemplo-pwa/ofertas.json', this)
  }
}

module.exports = Oferta

```

Ahora vamos a crear un controlador para guardar las suscripciones a las notificaciones push de los clientes que aceptan los permisos para recibirlas.

Dentro del controlador vamos a crear una función **saveSuscripcionPush** en la que vamos a obtener los datos de la suscripción del cuerpo de la petición. Estos datos tendriamos que guardarlos en alguna BBDD, pero nosotros lo vamos a hacer en un archivo JSON en **src/suscripciones/suscripciones-db.json**.

/angular-18-pwa-notificaciones-push-lab/server-notificaciones-push/src/controllers/push.controller.js

```

const path = require('path')
const fs = require('fs').promises

exports.saveSuscripcionPush = (req, res) => {
  const suscripcion = req.body
  const pathSuscripciones = path.join(__dirname, '..', 'suscripciones', 'suscripciones-db.json')

}

```

Ahora vamos a leer el archivo donde se guardan las suscripciones para no perder las que ya teníamos guardadas. Una vez obtenidas, vamos a añadir al final esta nueva y las volveremos a guardar en el mismo archivo.

/angular-18-pwa-notificaciones-push-lab/server-notificaciones-push/src/controllers/push.controller.js

```

const path = require('path')
const fs = require('fs').promises

exports.saveSuscripcionPush = (req, res) => {
  const suscripcion = req.body
  const pathSuscripciones = path.join(__dirname, '..', 'suscripciones', 'suscripciones-db.json')

  fs.readFile(pathSuscripciones)

```

```

    .then(data => {
      const suscripciones = JSON.parse(data)
      suscripciones.push(suscripcion)
      return fs.writeFile(pathSuscripciones, JSON.stringify(suscripciones, null, 2))
    })
    .then(() => {
      return res.json({msg: 'suscripción guardada'})
    })
  }
}

```

Ahora vamos a crear otro controlador para las ofertas, y en el vamos a crear un método **createOfertaTrabajo**.

Dentro de este método, primero vamos a obtener los datos que llegan en la petición POST y vamos a crear un objeto Oferta desde el que vamos a llamar al método **save** del modelo que hemos creado antes.

/angular-18-pwa-notificaciones-push-lab/server-notificaciones-push/src/controllers/ofertas.controller.js

```

const path = require('path')
const fs = require('fs').promises
const webpush = require('web-push')
const Oferta = require("../models/oferta.model")

exports.createOfertaTrabajo = (req, res) => {
  const { titulo, empresa, ciudad, salario, urlImagen, descripcion } = req.body
  const oferta = new Oferta(null, titulo, descripcion, empresa, salario, ciudad, urlImagen)

  oferta.save()
    .then(resp => {
      })
}

```

Una vez obtenemos la respuesta de la petición POST vamos a definir el objeto de payload con el que vamos a indicar que como se tiene que mostrar la notificación en los dispositivos.

/angular-18-pwa-notificaciones-push-lab/server-notificaciones-push/src/controllers/ofertas.controller.js

```

const path = require('path')
const fs = require('fs').promises
const webpush = require('web-push')
const Oferta = require("../models/oferta.model")

exports.createOfertaTrabajo = (req, res) => {
  const { titulo, empresa, ciudad, salario, urlImagen, descripcion } = req.body
  const oferta = new Oferta(null, titulo, descripcion, empresa, salario, ciudad, urlImagen)

  oferta.save()
    .then(resp => {
      const payload = {
        title: 'Oferta de trabajo',
        body: 'Has recibido una nueva oferta de trabajo',
        icon: 'https://angular-18-pwa-notificaciones-push-lab.s3.amazonaws.com/imagenes/icono-notificacion.png',
        url: 'https://angular-18-pwa-notificaciones-push-lab.s3.amazonaws.com/notificaciones-push.html'
      }
      webpush.sendNotification(res.locals.token, payload)
    })
}

```

```

        notification: {
            title: 'Nueva oferta de trabajo',
            body: `${titulo} - ${salario}€`,
            image: urlImagen,
            vibrate: [100, 50, 150],
        }
    }

})
}

```

Ahora vamos a leer nuestro archivo de suscripciones para recorrerlas e ir enviando una notificación push a cada una de ellas con el método **webpush.sendNotification** al que le vamos a pasar la suscripción (objeto que indica a qué dispositivo hay que enviar la notificación y a través de qué servicio de mensajería hacerlo), y un JSON con el payload.

/angular-18-pwa-notificaciones-push-lab/server-notificaciones-push/src/controllers/ofertas.controller.js

```

const path = require('path')
const fs = require('fs').promises
const webpush = require('web-push')
const Oferta = require("../models/oferta.model")

exports.createOfertaTrabajo = (req, res) => {
    const { titulo, empresa, ciudad, salario, urlImagen, descripcion } = req.body
    const oferta = new Oferta(null, titulo, descripcion, empresa, salario, ciudad, urlImagen)

    oferta.save()
        .then(resp => {

            const payload = {
                notification: {
                    title: 'Nueva oferta de trabajo',
                    body: `${titulo} - ${salario}€`,
                    image: urlImagen,
                    vibrate: [100, 50, 150],
                }
            }

            fs.readFile(path.join(__dirname, '..', 'suscripciones', 'suscripciones-db.json'))
                .then(data => {
                    const suscripciones = JSON.parse(data)
                    suscripciones.forEach((suscripcion) => {
                        webpush.sendNotification(suscripcion, JSON.stringify(payload))
                            .then(() => {
                                console.log('Notificación PUSH enviada')
                            })
                            .catch(err => {
                                console.log(err)
                            })
                    })
                })
                .return res.json({name: resp.data.name})
        })
}

```

```
    })
  }
}
```

Ahora vamos a crear un archivo de rutas para poner las rutas y que método de los controladores tienen que ejecutar.

/angular-18-pwa-notificaciones-push-lab/server-notificaciones-push/src/routes/app.routes.js

```
const express = require('express')
const OfertasController = require('../controllers/ofertas.controller')
const PushController = require('../controllers/push.controller')

const router = express.Router()

router.post('/ofertas', OfertasController.createOfertaTrabajo)
router.post('/suscribir-push', PushController.saveSuscripcionPush)

module.exports = router
```

Ahora vamos a añadir estas rutas en un **app.use** dentro del archivo principal.

/angular-18-pwa-notificaciones-push-lab/server-notificaciones-push/src/app.js

```
require('dotenv').config()
const express = require('express')
const cors = require('cors')
const webpush = require('web-push')
const appRoutes = require('./routes/app.routes')

const app = express()

webpush.setVapidDetails('mailto:contact@jobs.com', process.env.VAPID_PUBLIC_KEY, process.env.VAPID_PRIVATE_KEY)

app.use(cors())
app.use(express.json())
app.use(appRoutes)

app.listen(3000, () => {
  console.log('Listening on http://localhost:3000')
})
```

Con esto ya tenemos la parte del backend. Podemos levantar el servidor con el siguiente comando:

```
$ npm start
```

Ahora nos toca modificar nuestra aplicación de Angular para poder pedir el permiso para recibir notificaciones y así enviar los datos de la suscripción al servidor y que este los guarde en el archivo JSON que habíamos generado.

Esto lo vamos a hacer en el componente App, en el que tendremos que inyectar el servicio **SwPush** con el que llamaremos al método **requestSubscription** que recibe como parámetro un objeto con la Vapid Key pública que habíamos generado para el servidor.

/angular-18-pwa-notificaciones-push-lab/angular-18-pwa-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { RouterLink, RouterOutlet } from '@angular/router';
import { SwPush } from '@angular/service-worker';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    RouterOutlet,
    RouterLink,
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor(private swPush: SwPush) {
    this.pedirPermisoNotificaciones()
  }

  private pedirPermisoNotificaciones() {
    this.swPush.requestSubscription({
      serverPublicKey: 'BA1MZ0EcfvkMWhiK0qG6K1Z5W1XeS5kDk-SA37HT37nRPr2NeKtKA_te7eSAqcwwaSC8sIkV14b0NdcApSCrVf3'
    })
    .then((suscripcion: PushSubscription) => {
      })
    }
  }
}
```

Esta suscripción es la que tenemos que guardar en el servidor para que luego nos pueda enviar notificaciones Push, por lo que vamos a crear un servicio nuevo que se va a encargar de realizar la petición POST a nuestro servidor.

Navegamos dentro del proyecto de Angular desde un terminal y lanzamos el siguiente comando:

```
$ cd angular-18-pwa-notificaciones-push-lab/angular-18-pwa-lab
$ ng g s notificaciones-push
```

Ahora dentro del servicio vamos a inyectar el servicio **HttpClient** y vamos a crear un método para realizar la petición POST.

/angular-18-pwa-notificaciones-push-lab/angular-18-pwa-lab/src/app/notificaciones-push.service.ts

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

@Injectable({
```

```

    providedIn: 'root'
})
export class NotificacionesPushService {

  constructor(private http: HttpClient) { }

  saveSuscripcion(suscripcion: PushSubscription): Observable<any> {
    return this.http.post('http://localhost:3000/suscribir-push', suscripcion)
  }
}

```

Ahora que tenemos el servicio, vamos de nuevo al componente App, donde vamos a inyectar este servicio y llamar al método que hemos creado pasandole como parámetro los datos de la suscripción.

/angular-18-pwa-notificaciones-push-lab/angular-18-pwa-lab/src/app/app.component.ts

```

import { Component } from '@angular/core';
import { RouterLink, RouterOutlet } from '@angular/router';
import { SwPush } from '@angular/service-worker';
import { NotificacionesPushService } from './notificaciones-push.service';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    RouterOutlet,
    RouterLink,
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor(
    private swPush: SwPush,
    private notificacionesPush: NotificacionesPushService,
  ) {
    this.pedirPermisoNotificaciones()
  }

  private pedirPermisoNotificaciones() {
    this.swPush.requestSubscription({
      serverPublicKey: 'BA1MZ0EcfvkMVhiK0qG6K1Z5W1XeS5kDk-SA37HT37nRPr2NeKtKA_te7eSAqcwwaSC8sIkV14b0NdcApSCrVf3'
    })
      .then((suscripcion: PushSubscription) => {
        this.notificacionesPush.saveSuscripcion(suscripcion)
          .subscribe({
            next: (resp: any) => {
              console.log(resp)
            },
            error: (err: any) => {
              console.log(err)
            }
          })
      })
  }
}

```

Con esto ya podemos pedir el permiso en el navegador para poder recibir notificaciones push.

Ahora tenemos que realizar otro cambio, y es que cuando guardamos una nueva oferta de trabajo, estamos haciendo la petición a Firebase, pero ahora tenemos que hacerla a nuestro servidor con express para que este se encargue de enviar la notificación Push.

Por tanto en el servicio de Ofertas, tenemos que cambiar la URL a la que se está haciendo la petición POST.

/angular-18-pwa-notificaciones-push-lab/angular-18-pwa-lab/src/app/ofertas.service.ts

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

@Injectable({
  providedIn: 'root'
})
export class OfertasService {
  URL: string = 'https://fluentjobs-fa22e.firebaseio.com/ejemplo-pwa/ofertas'

  constructor(private http: HttpClient) { }

  getOfertas(): Observable<any> {
    return this.http.get(`${this.URL}.json`)
      .pipe(
        map((objOfertas: any) => {
          const arrOfertas = []

          for (let id in objOfertas) {
            arrOfertas.push({...objOfertas[id], id})
          }

          return arrOfertas
        })
      )
  }

  getOferta(id: string): Observable<any> {
    return this.http.get(`${this.URL}/${id}.json`)
  }

  crearOferta(nuevaOferta: any): Observable<any> {
    return this.http.post('http://localhost:3000/ofertas', nuevaOferta)
  }
}
```

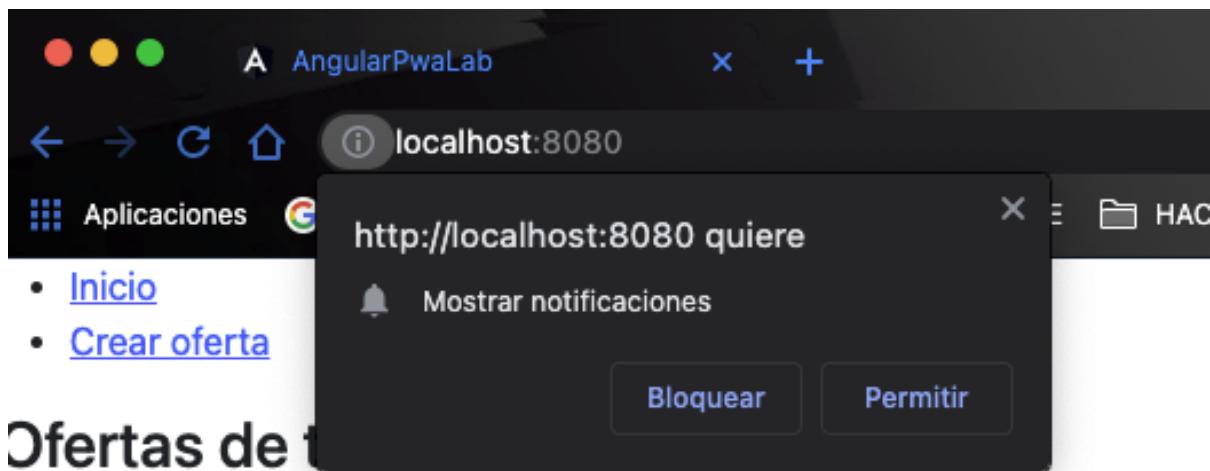
Y con esto ya tenemos nuestra aplicación de angular lista para recibir notificaciones push.

Vamos a comprobar que todo esto funciona.

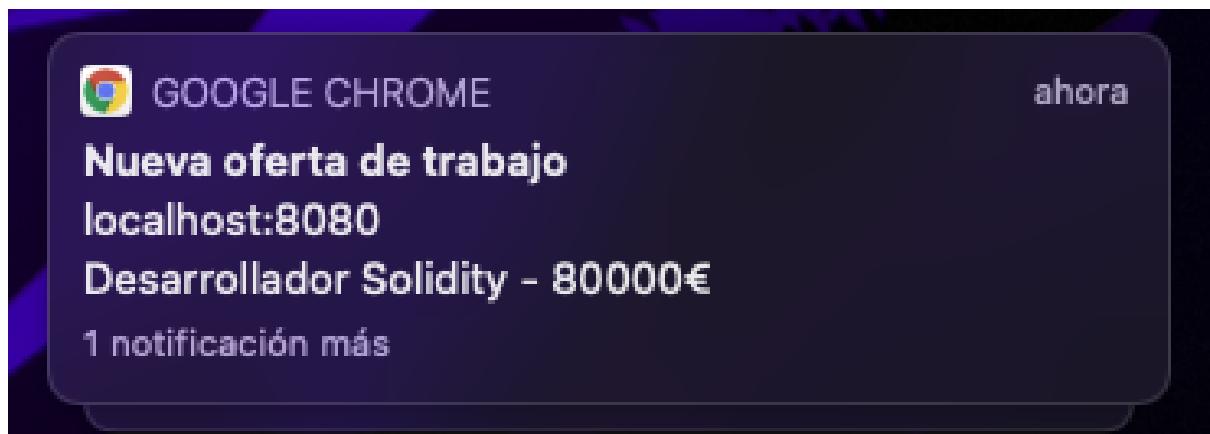
Ya teníamos levantado el servidor de express, y ahora vamos a lanzar el siguiente comando dentro del proyecto de angular para levantar esta aplicación.

```
$ npm run start:pwa
```

Una vez levantados los dos servidores, vamos a entrar en el navegador en <http://localhost:8080> y debería de salirnos un popup pidiéndonos permiso para recibir las notificaciones.

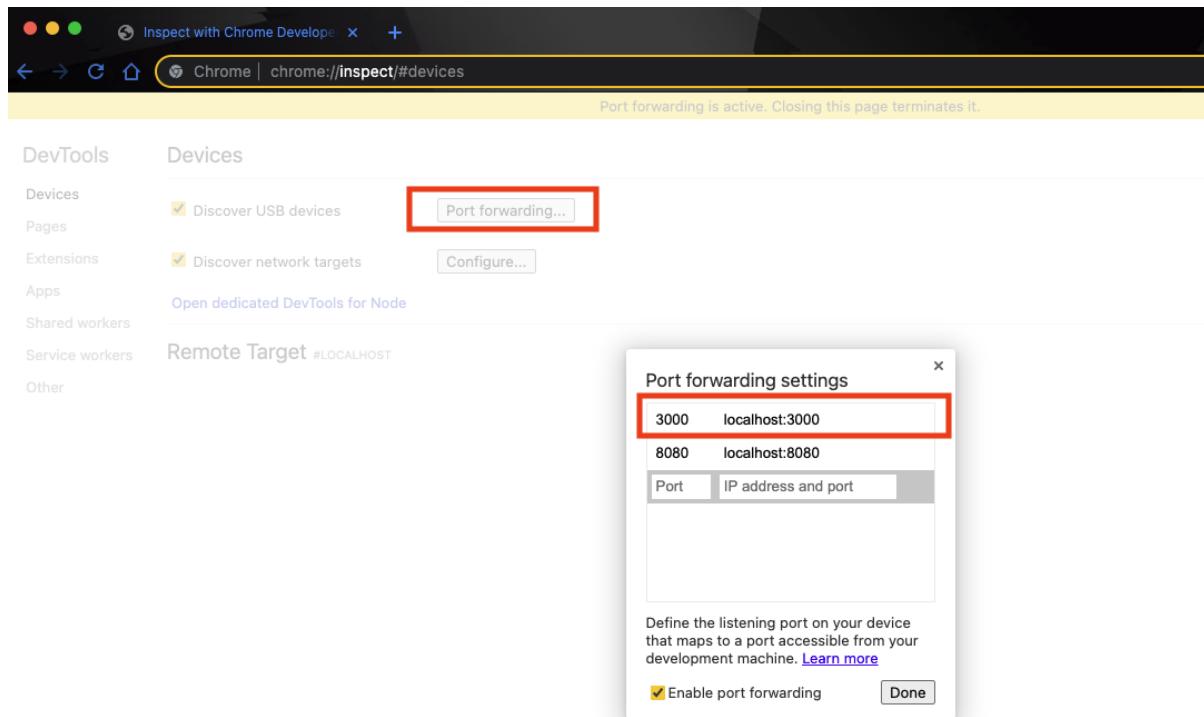


Ahora ya podemos crear una nueva oferta de trabajo y debería de salirnos una notificación en nuestro dispositivo.

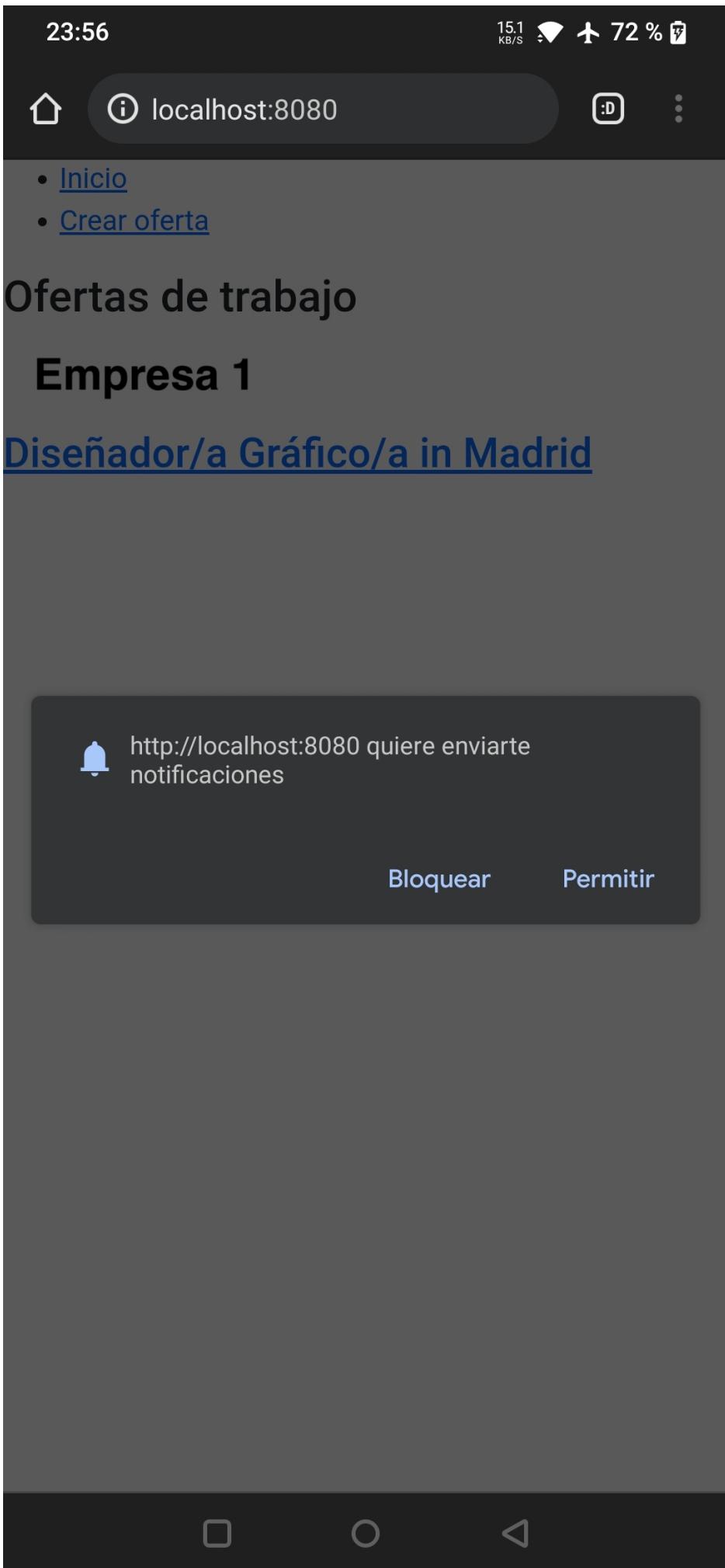


Esto funciona de la misma forma en los dispositivos móviles, hay que conectar el móvil (con la depuración USB activada) al ordenador.

En el ordenador entramos en <chrome://inspect/#devices> para comprobar que aparece el dispositivo conectado. Dentro de esta pestaña tenemos que habilitar el **port forwarding** para el puerto **3000** a parte del **8080**, ya que haremos peticiones al servidor de express desde el móvil.



Una vez activado, al entrar en <http://localhost:8080/> dentro del dispositivo móvil, debería de salirnos también la petición de permisos para recibir notificaciones.



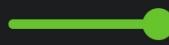
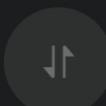
Al guardar una oferta nueva también nos tiene que salir la notificación en el dispositivo, y al igual que en las aplicaciones nativas se añade un circulito sobre el icono de la aplicación.

23:57

72 % 

Jue., 3 feb.

0.06
KB/S



Modo avión



Notificaciones

 Sistema Android 

Depuración USB habilitada

Toca aquí para desactivar la depuración USB



jobs • localhost:8080 • ahora 



Nueva oferta de trabajo

Desarrollador Solidity - 80000€



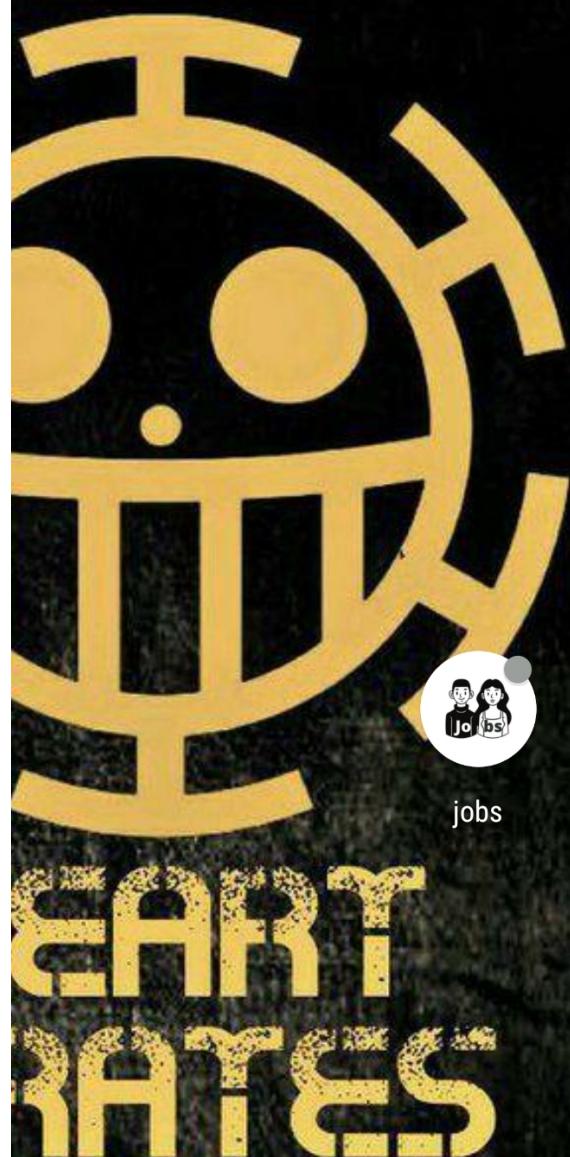
Gestionar

Borrar todo



23:57

2.84 KB/S 72 %



Pues con esto ya hemos visto como añadir notificaciones push en nuestra aplicación.

Chapter 5. Angular Universal

Angular Universal es una herramienta que permite hacer el **Server Side Rendering** (SSR) y **pre-rendering** o **Static Site Generation** (SSG) de las aplicaciones de Angular.

5.1. Server Side Rendering

El **SSR** consiste en realizar el primer renderizado de la aplicación en el servidor, con lo que conseguimos que la primera vez que llega la página al cliente no venga vacía como ocurre de forma normal con las SPAs.

Gracias al SSR conseguimos que la primera página de la aplicación se muestre muy rápido al cliente, algo que es muy beneficioso, ya que según la la regla de los 3 segundos de la UX, si una página tarda en cargar más de 3 segundos estas perdiendo el interés de los usuarios por tu aplicación.

Otro de los beneficios del SSR es que ayuda a que nuestras aplicaciones también puedan funcionar en dispositivos con menos recursos, sobre todo si estamos realizando un trabajo intenso con los scripts, ya que es este lo tendrán que hacer en estos dispositivos. Al menos con esto, le hemos quitado parte del trabajo inicial de pintar la primera página mediante scripts en el cliente.

Y si nos interesa que los buscadores nos tengan en cuenta a la hora de posicionarnos, esto es un gran paso, ya que no es lo mismo que estos cuando van indexando las páginas se encuentren con una página en blanco, a que se la encuentren con contenido.

5.2. Lab: Server Side Rendering

En este laboratorio vamos a ver la diferencia entre cargar una aplicación de Angular con SSR y sin el.

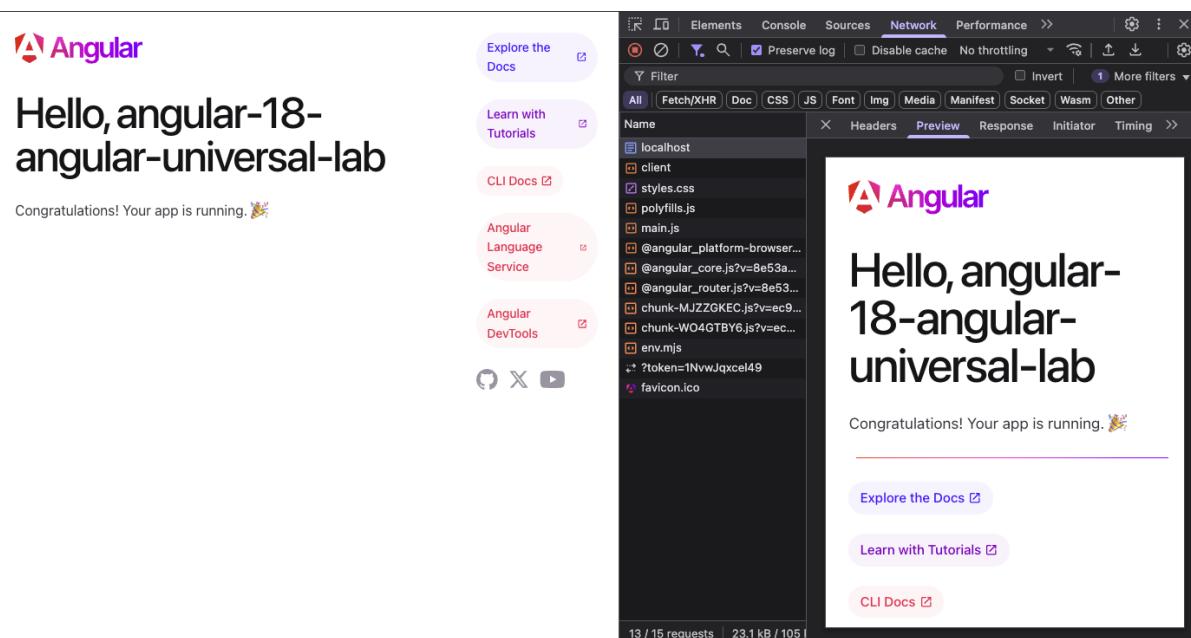
Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-18-angular-universal-ssr-lab  
? Which stylesheet format would you like to use? CSS  
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? y
```

Una vez creado el proyecto, vamos a levantar el servidor, con el siguiente comando:

```
$ ng s
```

Se levanta en <http://localhost:4200/>, con SSR, y si vamos a las herramientas del desarrollador, en la pestaña de **Network** buscamos la petición de **localhost** y entramos en la **preview** veremos que la página tiene contenido porque estamos usando el SSR.



5.3. Static Site Generation (SSG o Prerendering)

El **Static Site Generation (SSG)** o **prerendering** consiste en la generación de páginas HTML estáticas a partir de las páginas dinámicas.

Esto puede sernos útil si tenemos aplicaciones como un blog en el que el contenido es estático y no cambia muy a menudo.

Esto se debe a que con cada cambio que queramos realizar sobre el proyecto, tendremos que volver a prerenderizar las páginas para que se generen de nuevo con los últimos cambios.

Dentro del archivo de **angular.json** se genera una propiedad (por el final) **prerender** donde podemos configurar que rutas de la aplicación queremos prerenderizar.

Indicaremos que rutas son con **options.routes**, al que le pasamos un array de strings con las rutas. Aunque también podemos utilizar **options.routesFile** al que se le asigna como valor la ruta a un archivo en el que se han definido las rutas.

5.4. Lab: Prerendering

En este laboratorio vamos a prerenderizar una wiki con los datos de los digimon que vamos a extraer de la API <https://digimon-api.vercel.app/>.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-18-angular-prerendering-lab
? Which stylesheet format would you like to use? CSS
Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? y
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado, generaremos unos componentes y un servicio:

```
$ cd angular-18-angular-prerendering-lab
$ ng g c lista-digimon
$ ng g c digimon
$ ng g s digimon-api
```

Ahora vamos a levantar el servidor de desarrollo para ir construyendo la aplicación:

```
$ ng s
```

Vamos a crear una clase **Digimon** en una carpeta **models** para crear el modelo.

La API nos devuelve por cada digimon solo 3 propiedades:

- El nombre: name
- La imagen: img
- El nivel: level

/angular-18-angular-prerendering-lab/src/app/models/digimon.model.ts

```
export class Digimon {
  constructor(
    public name: string,
    public img: string,
    public level: string
  ) {}
}
```

Ahora vamos al archivo de rutas **app.routes.ts** en la carpeta **app** para definir 3 rutas:

- / → Lista el nombre de todos los digimons
- /digimon/:name → Muestra los datos de un digimon dado su nombre

- /digimon/level/:level → Lista el nombre de todos los digimons filtrados por el nivel

/angular-18-angular-prerendering-lab/src/app/app.routes.ts

```
import { Routes } from '@angular/router';
import { ListaDigimonComponent } from './lista-digimon/lista-digimon.component';
import { DigimonComponent } from './digimon/digimon.component';

export const routes: Routes = [
  { path: '', component: ListaDigimonComponent },
  { path: 'digimon/:name', component: DigimonComponent },
  { path: 'digimon/level/:level', component: ListaDigimonComponent },
];

```

En el archivo de configuración, vamos a añadir en los providers el **provideHttpClient(withFetch())** para poder hacer las peticiones a la API. El **withFetch()** es para poder hacer las peticiones usando la API de **fetch** del navegador y de Node, ya que es común a ambos entornos.

/angular-18-angular-prerendering-lab/src/app/app.config.ts

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

import { routes } from './app.routes';
import { provideClientHydration } from '@angular/platform-browser';
import { provideHttpClient, withFetch } from '@angular/common/http';

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

Dentro del TS del componente App vamos a declarar un array con los niveles de los digimon.

Y vamos a aprovechar a importar el pipe **TitleCasePipe** para poder capitalizar los niveles, y la directiva de **RouterLink** para poner unos enlaces.

/angular-18-angular-prerendering-lab/src/app/app.component.ts

```
import { TitleCasePipe } from '@angular/common';
import { Component } from '@angular/core';
import { RouterLink, RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
```

```

  standalone: true,
  imports: [
    RouterOutlet,
    RouterLink,
    TitleCasePipe,
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  levels: Array<string> = [
    'in training',
    'fresh',
    'training',
    'rookie',
    'champion',
    'ultimate',
    'mega'
  ]
}

```

En la plantilla vamos a poner un enlace para poder navegar hasta el inicio, y después crearemos un enlace por cada nivel del array.

Además de poner los enlaces, vamos a añadir el componente **router-outlet** para mostrar ahí los componentes asociados a las rutas.

`/angular-18-angular-prerendering-lab/src/app/app.component.html`

```

<h1>Digimon Wiki</h1>

<ul>
  <li>
    <a [routerLink]="/">Inicio</a>
  </li>
  @for (level of levels; track $index) {
    <li>
      <a [routerLink]="/digimon', 'level', level]">{{level | titlecase}}</a>
    </li>
  }
</ul>

<router-outlet></router-outlet>

```

El siguiente paso es añadir las 3 peticiones GET en el servicio para poder obtener los datos de la API:

- `getDigimons`: obtiene todos los digimons
- `getDigimonByName`: obtiene el digimon dado su nombre

- `getDigimonByLevel`: obtiene todos los digimons del nivel dado

/angular-18-angular-prerendering-lab/src/app/digimon-api.service.ts

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { map, Observable } from 'rxjs';
import { Digimon } from './models/digimon.model';

@Injectable({
  providedIn: 'root'
})
export class Digimon ApiService {
  URL: string = 'https://digimon-api.vercel.app/api/digimon'

  constructor(
    private http: HttpClient,
  ) { }

  getDigimons(): Observable<Array<Digimon>> {
    return this.http.get<Array<Digimon>>(this.URL)
  }

  getDigimonByName(name: string): Observable<Digimon> {
    return this.http.get<Array<Digimon>>(` ${this.URL}/name/${name}`)
      .pipe(
        map((digimons: Array<Digimon>) => digimons[0])
      )
  }

  getDigimonByLevel(level: string): Observable<Array<Digimon>> {
    return this.http.get<Array<Digimon>>(` ${this.URL}/level/${level}`)
  }
}
```

Pues ya que tenemos las peticiones, vamos a añadir el código del componente que muestra los datos de un digimon.

En el componente **DigimonComponent** vamos a inyectar el servicio que hemos creado además del **ActivatedRoute** para poder obtener el nombre de la URL.

/angular-18-angular-prerendering-lab/src/app/digimon/digimon.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Digimon ApiService } from '../digimon-api.service';
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-digimon',
  standalone: true,
  imports: [],
})
```

```

        templateUrl: './digimon.component.html',
        styleUrls: ['./digimon.component.css']
    })
export class DigimonComponent implements OnInit {

    constructor(
        private route: ActivatedRoute,
        private digimon ApiService: Digimon ApiService,
    ) { }

    ngOnInit(): void{
    }

}

```

En el método **ngOnInit** vamos a pedir el parámetro de la ruta con el **route.snapshot.paramMap** ya que esta vez cada vez que cambiemos de URL se elimina y crea el componente.

Y este nombre se lo vamos a pasar a la función **getDigimonByName** de nuestro servicio.

/angular-18-angular-prerendering-lab/src/app/digimon/digimon.component.ts

```

import { Component, OnInit } from '@angular/core';
import { Digimon ApiService } from '../digimon-api.service';
import { ActivatedRoute } from '@angular/router';

@Component({
    selector: 'app-digimon',
    standalone: true,
    imports: [],
    templateUrl: './digimon.component.html',
    styleUrls: ['./digimon.component.css'
})
export class DigimonComponent implements OnInit {

    constructor(
        private route: ActivatedRoute,
        private digimon ApiService: Digimon ApiService,
    ) { }

    ngOnInit(): void{
        const name = this.route.snapshot.paramMap.get('name');
        this.digimon ApiService.getDigimonByName(name!)
    }

}

```

Por último, nos suscribimos al observable y vamos a guardar los datos del digimon en una propiedad del componente.

/angular-18-angular-prerendering-lab/src/app/digimon/digimon.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Digimon ApiService } from '../digimon-api.service';
import { ActivatedRoute } from '@angular/router';
import { Digimon } from '../models/digimon.model';

@Component({
  selector: 'app-digimon',
  standalone: true,
  imports: [],
  templateUrl: './digimon.component.html',
  styleUrls: ['./digimon.component.css']
})
export class DigimonComponent implements OnInit {
  digimon: Digimon = new Digimon('', '', '')

  constructor(
    private route: ActivatedRoute,
    private digimon ApiService: Digimon ApiService,
  ) { }

  ngOnInit(): void{
    const name = this.route.snapshot.paramMap.get('name');
    this.digimon ApiService.getDigimonByName(name!)
      .subscribe((digimon: Digimon) => {
        this.digimon = digimon
      })
  }
}
```

Vamos a importar el **RouterLink** para poner un enlace en la plantilla.

/angular-18-angular-prerendering-lab/src/app/digimon/digimon.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Digimon ApiService } from '../digimon-api.service';
import { ActivatedRoute, RouterLink } from '@angular/router';
import { Digimon } from '../models/digimon.model';

@Component({
  selector: 'app-digimon',
  standalone: true,
  imports: [
    RouterLink,
  ],
  templateUrl: './digimon.component.html',
  styleUrls: ['./digimon.component.css']
})
```

```

export class DigimonComponent implements OnInit {
  digimon: Digimon = new Digimon('', '', '')

  constructor(
    private route: ActivatedRoute,
    private digimon ApiService: Digimon ApiService,
  ) { }

  ngOnInit(): void{
    const name = this.route.snapshot.paramMap.get('name');
    this.digimon ApiService.getDigimonByName(name!)
      .subscribe((digimon: Digimon) => {
        this.digimon = digimon
      })
  }
}

```

En la plantilla vamos a mostrar la imagen, el nombre y pondremos un enlace que nos va a mandar a la página que lista todos los digimon del mismo nivel.

/angular-18-angular-prerendering-lab/src/app/digimon/digimon.component.html

```

<div>
  <img [src]="digimon.img" alt="Imagen de {{digimon.name}}" width="150">
  <h2>{{digimon.name}}</h2>
  <p>Level: <a [routerLink]=["/digimon", 'level', digimon.level]">{{digimon.level}}</a></p>
</div>

```

En el componente **ListaDigimonComponent** vamos a pintar una lista de digimons, y para ello vamos a crearnos una propiedad para guardarlos.

Inyectaremos en el constructor nuestro servicio y el **ActivatedRoute**, al igual que en el componente anterior.

Y también vamos a crear un método **getList** con el que vamos a devolver un observable.

/angular-18-angular-prerendering-lab/src/app/lista-digimon/lista-digimon.component.ts

```

import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { Digimon ApiService } from '../digimon-api.service';
import { Observable } from 'rxjs';
import { Digimon } from '../models/digimon.model';

@Component({
  selector: 'app-lista-digimon',
  standalone: true,
  imports: [],
  templateUrl: './lista-digimon.component.html',
}

```

```

    styleUrls: './lista-digimon.component.css'
  })
export class ListaDigimonComponent implements OnInit {
  digimons: Array<Digimon> = []

  constructor(
    private route: ActivatedRoute,
    private digimon ApiService: Digimon ApiService
  ) { }

  ngOnInit(): void {
  }

  getLista(): Observable<Array<Digimon>> {
  }
}

```

Dentro de **getLista**, vamos a comprobar si la ruta tiene un parámetro **level** o no.

Si tiene el parámetro, vamos a llamar al método **getDigimonByLevel** y si no llamaremos a **getDigimons**. Vamos a devolver el observable que devuelven estos dos métodos.

/angular-18-angular-prerendering-lab/src/app/lista-digimon/lista-digimon.component.ts

```

import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, ParamMap } from '@angular/router';
import { Digimon ApiService } from '../digimon-api.service';
import { Observable } from 'rxjs';
import { Digimon } from '../models/digimon.model';

@Component({
  selector: 'app-lista-digimon',
  standalone: true,
  imports: [],
  templateUrl: './lista-digimon.component.html',
  styleUrls: './lista-digimon.component.css'
})
export class ListaDigimonComponent implements OnInit {
  digimons: Array<Digimon> = []

  constructor(
    private route: ActivatedRoute,
    private digimon ApiService: Digimon ApiService
  ) { }

  ngOnInit(): void {
  }
}

```

```

getLista(): Observable<Array<Digimon>> {
  const params: ParamMap = this.route.snapshot.paramMap
  if (params.has('level')) {
    const level = params.get('level')!
    return this.digimon ApiService.getDigimonByLevel(level)
  }

  return this.digimon ApiService.getDigimons()
}
}

```

En el **ngOnInit** nos vamos a suscribir al observable que retorne la función **getLista** y vamos a asignarle la lista de digimons a la propiedad que habíamos declarado antes.

/angular-18-angular-prerendering-lab/src/app/lista-digimon/lista-digimon.component.ts

```

import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, ParamMap } from '@angular/router';
import { Digimon ApiService } from '../digimon-api.service';
import { Observable } from 'rxjs';
import { Digimon } from '../models/digimon.model';

@Component({
  selector: 'app-lista-digimon',
  standalone: true,
  imports: [],
  templateUrl: './lista-digimon.component.html',
  styleUrls: ['./lista-digimon.component.css']
})
export class ListaDigimonComponent implements OnInit {
  digimons: Array<Digimon> = []

  constructor(
    private route: ActivatedRoute,
    private digimon ApiService: Digimon ApiService
  ) { }

  ngOnInit(): void {
    this.getLista()
      .subscribe((digimons: Array<Digimon>) => {
        this.digimons = digimons
      })
  }

  getLista(): Observable<Array<Digimon>> {
    const params: ParamMap = this.route.snapshot.paramMap
    if (params.has('level')) {
      const level = params.get('level')!
      return this.digimon ApiService.getDigimonByLevel(level)
    }
  }
}

```

```

        return this.digimon ApiService.getDigimons()
    }
}

```

Y por último vamos a añadir el **RouterLink** en las importaciones para añadir unos enlaces en la plantilla del componente.

/angular-18-angular-prerendering-lab/src/app/lista-digimon/lista-digimon.component.ts

```

import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, ParamMap, RouterLink } from '@angular/router';
import { Digimon ApiService } from '../digimon-api.service';
import { Observable } from 'rxjs';
import { Digimon } from '../models/digimon.model';

@Component({
  selector: 'app-lista-digimon',
  standalone: true,
  imports: [
    RouterLink,
  ],
  templateUrl: './lista-digimon.component.html',
  styleUrls: ['./lista-digimon.component.css'
})
export class ListaDigimonComponent implements OnInit {
  digimons: Array<Digimon> = []

  constructor(
    private route: ActivatedRoute,
    private digimon ApiService: Digimon ApiService
  ) { }

  ngOnInit(): void {
    this.getLista()
      .subscribe((digimons: Array<Digimon>) => {
        this.digimons = digimons
      })
  }

  getLista(): Observable<Array<Digimon>> {
    const params: ParamMap = this.route.snapshot.paramMap
    if (params.has('level')) {
      const level = params.get('level')!
      return this.digimon ApiService.getDigimonByLevel(level)
    }

    return this.digimon ApiService.getDigimons()
  }
}

```

Una vez obtenida la lista de digimons, vamos a pintar los nombres de estos dentro de un enlace que nos llevará a su información.

`/angular-18-angular-prerendering-lab/src/app/lista-digimon/lista-digimon.component.html`

```
<ul>
  @for (digimon of digimons; track $index) {
    <li>
      {{$index}}: <a [routerLink]=[['/digimon', digimon.name]]>{{digimon.name}}</a>
    </li>
  }
</ul>
```

Ya tenemos la aplicación, vamos a configurar el prerender para indicarle que rutas tenemos y de las cuales tiene que generar una página de HTML estática.

Vamos a ir al archivo **angular.json** y buscaremos por el final una propiedad **prerender.options**, en la que tenemos que añadir la propiedad **routesFile** con el valor a un archivo donde vamos a poner las rutas que hay que generar.

`/angular-18-angular-prerendering-lab/angular.json`

```
{
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
  "version": 1,
  "newProjectRoot": "projects",
  "projects": {
    "angular-18-angular-prerendering-lab": {
      "projectType": "application",
      "schematics": {},
      "root": "",
      "sourceRoot": "src",
      "prefix": "app",
      "architect": {
        "build": {
          "builder": "@angular-devkit/build-angular:application",
          "options": {
            "outputPath": "dist/angular-18-angular-prerendering-lab",
            "index": "src/index.html",
            "browser": "src/main.ts",
            "polyfills": [
              "zone.js"
            ],
            "tsConfig": "tsconfig.app.json",
            "assets": [
              {
                "glob": "**/*",
                "input": "public"
              }
            ],
          }
        }
      }
    }
  }
}
```

```

"styles": [
  "src/styles.css"
],
"scripts": [],
"server": "src/main.server.ts",
"prerender": {
  "routesFile": "digimon-routes.txt"
},
"ssr": {
  "entry": "server.ts"
}
},
"configurations": {
  "production": {
    "budgets": [
      {
        "type": "initial",
        "maximumWarning": "500kB",
        "maximumError": "1MB"
      },
      {
        "type": "anyComponentStyle",
        "maximumWarning": "2kB",
        "maximumError": "4kB"
      }
    ],
    "outputHashing": "all"
  },
  "development": {
    "optimization": false,
    "extractLicenses": false,
    "sourceMap": true
  }
},
"defaultConfiguration": "production"
},
"serve": {
  "builder": "@angular-devkit/build-angular:dev-server",
  "configurations": {
    "production": {
      "buildTarget": "angular-18-angular-prerendering-lab:build:production"
    },
    "development": {
      "buildTarget": "angular-18-angular-prerendering-lab:build:development"
    }
  },
  "defaultConfiguration": "development"
},
"extract-i18n": {
  "builder": "@angular-devkit/build-angular:extract-i18n"
}
}

```

```
  "test": {
    "builder": "@angular-devkit/build-angular:karma",
    "options": {
      "polyfills": [
        "zone.js",
        "zone.js/testing"
      ],
      "tsConfig": "tsconfig.spec.json",
      "assets": [
        {
          "glob": "**/*",
          "input": "public"
        }
      ],
      "styles": [
        "src/styles.css"
      ],
      "scripts": []
    }
  }
}
```

Como queremos generar las distintas páginas de HTML de cada uno de los digimons, pero son tantos que escribirlas todas a mano nos llevaría bastante tiempo, vamos a generar un script con Node que lo hará por nosotros.

Crearemos en la raíz del proyecto un nuevo archivo que vamos a llamar **generate-routes-file.js**.

Necesitaremos instalar la dependencia de axios para obtener todos los digímos y de estos sacar los nombres para generar las rutas. Por lo que vamos a lanzar el siguiente comando dentro de la carpeta del proyecto:

```
$ npm install axios
```

Ahora dentro del archivo **generate-routes-file.js**, vamos a empezar por importar las dependencias que usaremos:

- axios: para hacer una petición GET a la API
 - fs: para crear el archivo digimon-routes.txt con las rutas
 - path: para crear la ruta donde generar el archivo TXT

<https://github.com/NetanelBasile/angular-18-angular-prerendering-lab> /generate-routes-file.js

```
const path = require('path')
const fs = require('fs').promises
```

```
const axios = require('axios')
```

Vamos a empezar haciendo una petición GET con axios a la api para traer todos los digimons.

Una vez los tengamos, vamos a generar a partir de la respuesta un string con las rutas `/digimon/<nombre-digimon>` separadas por saltos de línea.

/angular-18-angular-prerendering-lab/generate-routes-file.js

```
const path = require('path')
const fs = require('fs').promises
const axios = require('axios')

axios.get('https://digimon-api.vercel.app/api/digimon')
  .then((resp) => {
    const rutas = resp.data.map(d => `/digimon/${d.name}`).join('\n')
  })
```

Ahora tenemos que escribir estas rutas en el archivo `digimon-routes.txt`, y para ello usaremos el módulo de fs (con promesas) y el de path.

/angular-18-angular-prerendering-lab/generate-routes-file.js

```
const path = require('path')
const fs = require('fs').promises
const axios = require('axios')

axios.get('https://digimon-api.vercel.app/api/digimon')
  .then((resp) => {
    const rutas = resp.data.map(d => `/digimon/${d.name}`).join('\n')
    return fs.writeFile(path.join(__dirname, 'digimon-routes.txt'), rutas)
  })
  .then(() => {
    console.log('Archivo de rutas generado correctamente :)')
  })
  .catch(err => {
    console.log('Error al generar el archivo de rutas', err)
  })
})
```

Ya tenemos el script, ahora solo falta comprobar que todo esto funciona y nos genera los archivos estáticos.

Vamos a lanzar desde la carpeta raíz del proyecto los siguientes comandos:

- El primero para generar el archivo de rutas.
- El segundo para generar los archivos estáticos.

```
$ node generate-routes-file.js
```

```
$ ng build
```

Cuando termina la ejecución del segundo comando, deberíamos de tener todos las páginas HTML dentro de **dist/angular-18-angular-prerendering-lab/browser**, entonces podemos servirlas lanzando el siguiente comando:

```
$ npx http-serve dist/angular-18-angular-prerendering-lab/browser
```

Al entrar en <http://localhost:8080/> ya podemos ver la página web.

Para mejorar el proceso de prerenderizar las páginas podemos crear un script que lance primero el script para generar el archivo de rutas, y después haga el prerenderizado.

Dentro del **package.json**, vamos a añadir un nuevo script **genfile:prerender**.

/angular-18-angular-prerendering-lab/package.json

```
{
  "name": "angular-18-angular-prerendering-lab",
  "version": "0.0.0",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "watch": "ng build --watch --configuration development",
    "test": "ng test",
    "genfile:prerender": "node generate-routes-file.js && npm run build"
  },
  "private": true,
  "dependencies": {
    "@angular/animations": "^18.2.0",
    "@angular/common": "^18.2.0",
    "@angular/compiler": "^18.2.0",
    "@angular/core": "^18.2.0",
    "@angular/forms": "^18.2.0",
    "@angular/platform-browser": "^18.2.0",
    "@angular/platform-browser-dynamic": "^18.2.0",
    "@angular/platform-server": "^18.2.0",
    "@angular/router": "^18.2.0",
    "@angular/ssr": "^18.2.21",
    "axios": "^1.12.2",
    "express": "^4.18.2",
    "rxjs": "~7.8.0",
    "tslib": "^2.3.0",
    "zone.js": "~0.14.10"
  },
  "devDependencies": {
    "@angular-devkit/build-angular": "^18.2.21",
    "@angular/cli": "^18.2.21",
  }
}
```

```
  "@angular/compiler-cli": "^18.2.0",
  "@types/express": "^4.17.17",
  "@types/jasmine": "~5.1.0",
  "@types/node": "^18.18.0",
  "jasmine-core": "~5.2.0",
  "karma": "~6.4.0",
  "karma-chrome-launcher": "~3.2.0",
  "karma-coverage": "~2.2.0",
  "karma-jasmine": "~5.1.0",
  "karma-jasmine-html-reporter": "~2.1.0",
  "typescript": "~5.5.2"
}
}
```

Ahora en lugar de lanzar dos comandos para realizar el prerenderizado, podemos lanzar el siguiente:

```
$ npm run genfile:prerender
```



Parece ser que en Angular 18 hay un bug por el que al ejecutar el script para realizar el SSG o prerender de las páginas se queda colgado en "Building..." y no termina de ejecutarlo. Habrá que esperar a ver si lo corrigen en las próximas versiones.

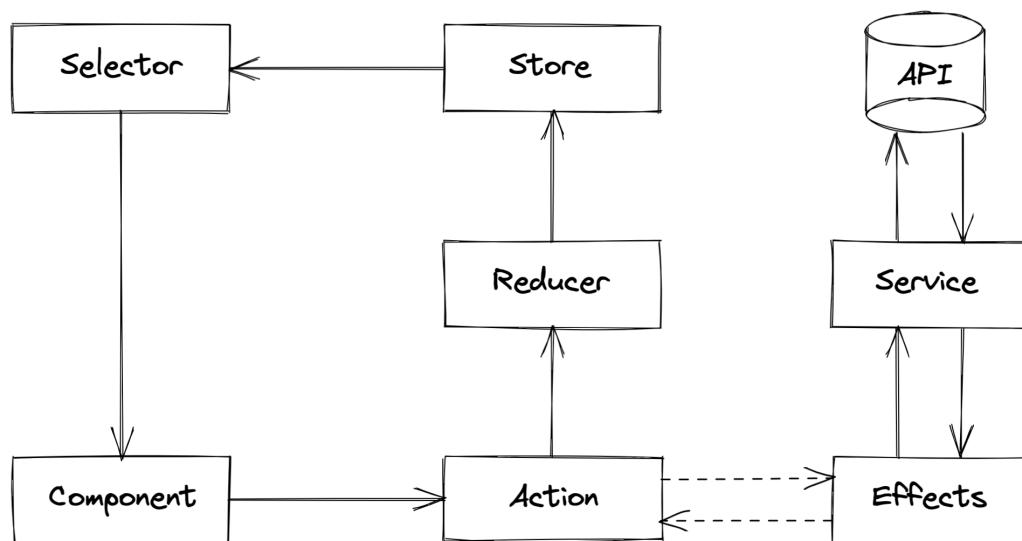
Chapter 6. Ngrx

Ngrx es la implementación del patrón **Flux** para las aplicaciones de Angular.

Con Ngrx podemos **gestionar el estado** de las aplicaciones de Angular, dejando así los componentes mucho más limpios ya que vamos a poder quitar toda la lógica que haya en ellos. De lo único que se van a encargar es pintar cosas e indicar que cambios hay que realizar en el estado.

Dentro de Ngrx nos encontramos con varios elementos que tendremos que utilizar:

- Actions
- Reducers
- Selectors
- Store
- Effects



Ngrx sigue 3 principios:

- Los cambios en el estado se definen a través de las **actions** que son eventos que se despachan desde los componentes y servicios.
- Los **reducers** son los encargados de realizar los cambios en el estado.
- Los **selectors** son las funciones puras con las que pedimos una parte del estado.

6.1. Actions

Las **actions** son eventos únicos que vamos a emitir desde los componentes, servicios... y que van a pasar a través de la aplicación para indicar que cambios hay que realizar sobre el estado.

Un **action** es un objeto que tiene una propiedad **type** con la que indicamos que cambio se va a realizar en el estado, y opcionalmente se le puede pasar un **payload** con datos extra necesarios para realizar dicho cambio.

Las acciones se crean con la función **createAction** a la que le pasamos como primer parámetro el tipo de la acción, y se le puede pasar un segundo parámetro que sería una llamada a **props** en la que se le indica como tipo genérico el tipo de datos que vamos a enviar en el **payload**.

Las acciones se despachan desde el servicio del **Store** con el método **dispatch** al que le pasamos como parámetro la acción que devuelve la función **createAction** cuando se ejecuta.

6.2. Reducers

Los **reducers** son funciones puras que reciben el **estado actual** y un **action** y los usan para crear el **nuevo estado** de la aplicación. Este nuevo estado se inyecta en los componentes para refrescar la interfaz de usuario con los nuevos datos.

Los reducers los creamos con la función **createReducer** que recibe como parámetros el estado inicial y una lista de handlers (función **on**) para gestionar el estado de distintas formas en función de la acción despachada.

La función **on** recibe como parámetros el tipo de la acción como primer parámetro y una función de callback con el estado y la acción despachada. Estos handlers tienen que retornar el nuevo estado.

6.3. Selectors

Los selectores son funciones puras que se usan para obtener una parte del estado del store de Ngrx.

Creamos los selectores con la función **createSelector**. Esta función recibe varias funciones de callback que reciben el estado global (o anterior si ya hay una función previa) y nos permite devolver la parte del estado que necesitamos obtener (normalmente en la vista).

Estos datos se piden desde la función **select** del servicio del Store, pasándole como parámetro la referencia a la función selectora que nos va a devolver lo que queremos.

6.4. Lab: Contador con Ngrx

En este laboratorio vamos a ver como usar las acciones, reducers, selectores y store de Ngrx para gestionar el estado de un contador.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-ngrx-contador-lab  
? Would you like to add Angular routing? N  
? Which stylesheet format would you like to use? CSS
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado y dentro de ella vamos a empezar creando el pipe filtro y después levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-ngrx-contador-lab  
$ ng g c contador
```

Además de crear el componente, vamos a instalar la librería de **@ngrx/store** con el siguiente comando:

```
$ ng add @ngrx/store  
The package @ngrx/store@13.0.2 will be installed and executed.  
Would you like to proceed? Yes
```

Y ahora levantaremos el servidor de desarrollo:

```
$ ng s
```

Antes de ponernos manos a la obra a escribir código, vamos a crear la estructura de carpetas y archivos pertenecientes a Ngrx. Dentro de la carpeta **app**, vamos a crear el siguiente árbol de carpetas y archivos:

```
|- app  
| |- actions  
| | |- contador.actions.ts  
| | |- action.types.ts  
| |- reducers  
| | |- contador.reducer.ts  
| |- selectors  
| | |- contador.selectors.ts  
| |- app.state.ts
```

Una vez creados todos esos archivos y carpetas, vamos a empezar por declarar unas constantes con el tipo de las acciones que vamos a enviar desde nuestro componente. Tendremos 3 acciones:

- Incrementar la cuenta
- Decrementar la cuenta
- Cambiar la cuenta

Por tanto en el archivo de **action.types.ts** vamos a declarar 3 constantes que vamos a exportar para luego usarlas al definir las acciones.

/angular-ngrx-contador-lab/src/app/state/actions/action.types.ts

```
export const INCREMENTAR = 'Incrementar'  
export const DECREMENTAR = 'Decrementar'  
export const CAMBIAR CUENTA = 'Cambiar cuenta'
```

Ahora que tenemos los tipos, vamos a crear las funciones que van a generar las acciones.

Dentro del archivo de **contador.actions.ts**, vamos a crear una acción **incrementar** usando la función **createAction** que viene de Ngrx, y le vamos a pasar el tipo de la acción, es decir la constante que hemos añadido en el archivo de antes.

/angular-ngrx-contador-lab/src/app/state/actions/contador.actions.ts

```
import { createAction, props } from "@ngrx/store";  
import * as ActionTypes from './action.types'  
  
export const incrementar = createAction(  
  ActionTypes.INCREMENTAR  
)
```

Vamos a hacer lo mismo para la acción de **decrementar**.

/angular-ngrx-contador-lab/src/app/state/actions/contador.actions.ts

```
import { createAction, props } from "@ngrx/store";  
import * as ActionTypes from './action.types'  
  
export const incrementar = createAction(  
  ActionTypes.INCREMENTAR  
)  
  
export const decrementar = createAction(  
  ActionTypes.DECREMENTAR  
)
```

Y por último, la acción de **cambiarCuenta** es un poco distinta, ya que a parte del tipo de la acción, vamos a necesitar pasarle una propiedad (un payload) **cuenta** para poder cambiar el valor del

estado con unos datos que se le van a dar. Para ello, le pasamos como segundo parámetro la llamada a la función **props** en la que vamos a indicar el tipo de dato del payload.

/angular-ngrx-contador-lab/src/app/state/actions/contador.actions.ts

```
import { createAction, props } from "@ngrx/store";
import * as ActionTypes from './action.types'

export const incrementar = createAction(
  ActionTypes.INCREMENTAR
)

export const decrementar = createAction(
  ActionTypes.DECREMENTAR
)

export const cambiarCuenta = createAction(
  ActionTypes.CAMBIAR CUENTA,
  props<{cuenta: number}>()
)
```

Con esto ya tenemos las acciones, ahora toca llenar el reducer al que le van a llegar estas acciones y en el que cambiaremos el estado de la aplicación.

Dentro del archivo de **contador.reducer.ts** vamos a empezar declarando un estado inicial.

/angular-ngrx-contador-lab/src/app/state/reducers/contador.reducer.ts

```
export interface IContadorState {
  cuenta: number
}

const initialState: IContadorState = {
  cuenta: 0
}
```

Ahora vamos a crear el reducer con la función **createReducer** a la que le vamos a pasar como primer parámetro el estado inicial que hemos creado antes.

/angular-ngrx-contador-lab/src/app/state/reducers/contador.reducer.ts

```
import { createReducer } from "@ngrx/store"

export interface IContadorState {
  cuenta: number
}

const initialState: IContadorState = {
  cuenta: 0
}
```

```
export const contadorReducer = createReducer(  
  initialState  
)
```

También le vamos a pasar otros 3 parámetros más para indicarle como tiene que cambiar el estado en función de cada acción. Esto lo haremos con la función **on** en la que le pasamos como primer parámetro la acción, y como segundo parámetro una función de callback en la que recibimos el estado y la acción y en la que tenemos que retornar el nuevo estado.

/angular-ngrx-contador-lab/src/app/state/reducers/contador.reducer.ts

```
import { createReducer, on } from "@ngrx/store"  
import { decrementar, incrementar, cambiarCuenta } from '../actions/contador.actions';  
  
export interface IContadorState {  
  cuenta: number  
}  
  
const initialState: IContadorState = {  
  cuenta: 0  
}  
  
export const contadorReducer = createReducer(  
  initialState,  
  on(incrementar, (state) => {  
    return { cuenta: state.cuenta + 1 }  
  }),  
  on(decrementar, (state) => {  
    return { cuenta: state.cuenta - 1 }  
  }),  
  on(cambiarCuenta, (state, action) => {  
    return { cuenta: action.cuenta }  
  }),  
)
```

Ahora vamos a registrar nuestro reducer, y para ello, en el archivo **app.state.ts** vamos a exportar un objeto con todos los reducers de nuestra aplicación.

/angular-ngrx-contador-lab/src/app/state/app.state.ts

```
import { ActionReducerMap } from "@ngrx/store";  
import { contadorReducer, IContadorState } from "./reducers/contador.reducer";  
  
export interface IAppState {  
  contadorState: IContadorState  
}  
  
export const AppReducers: ActionReducerMap<IAppState> = {  
  contadorState: contadorReducer
```

```
}
```

Y este objeto **AppReducers** lo tenemos que pasar como primer parámetro a la función **provideStore** de la configuración de la aplicación.

/angular-ngrx-contador-lab/src/app/app.config.ts

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

import { routes } from './app.routes';
import { provideStore } from '@ngrx/store';
import { AppReducers } from './app.state';

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

Ahora vamos a ir a mostrar el componente del contador en App, por tanto tenemos que añadirlo en el imports del componente standalone.

/angular-ngrx-contador-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { ContadorComponent } from './contador/contador.component';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, ContadorComponent],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'angular-18-ngrx-contador-lab';
}
```

/angular-ngrx-contador-lab/src/app/app.component.html

```
<app-contador></app-contador>
```

Dentro del contador, vamos a añadir dos botones (para incrementar y decrementar), la cuenta y un input para cambiar la cuenta.

/angular-ngrx-contador-lab/src/app/contador/contador.component.html

```
<p>Cuenta: {{cuenta}}</p>
<button type="button" (click)="decrementar()">-</button>
<button type="button" (click)="incrementar()">+</button>
<input type="number" (change)="cambiarCuenta($event)">
```

En el archivo de TypeScript vamos a añadir estos método y la cuenta.

/angular-ngrx-contador-lab/src/app/contador/contador.component.ts

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

@Component({
  selector: 'app-contador',
  standalone: true,
  imports: [],
  templateUrl: './contador.component.html',
  styleUrls: ['./contador.component.css'
})
export class ContadorComponent implements OnInit {
  cuenta: number = 0

  constructor() { }

  ngOnInit(): void {
  }

  incrementar() {

  }

  decrementar() {

  }

  cambiarCuenta(event: any) {
  }
}
```

Desde estas 3 funciones vamos a despachar las acciones que ya habíamos creado anteriormente, para poder hacer esto necesitamos inyectar el **store** en el componente.

/angular-ngrx-contador-lab/src/app/contador/contador.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Store } from '@ngrx/store';
import { incrementar, cambiarCuenta } from '../state/actions/contador.actions';
import { IAppState } from '../state/app.state';
```

```

@Component({
  selector: 'app-contador',
  standalone: true,
  imports: [],
  templateUrl: './contador.component.html',
  styleUrls: ['./contador.component.css']
})
export class ContadorComponent implements OnInit {
  cuenta: number = 0

  constructor(private store: Store<IAppState>) { }

  ngOnInit(): void {
  }

  incrementar() {
  }

  decrementar() {
  }

  cambiarCuenta(event: any) {
  }
}

```

Ahora podemos usar el método **dispatch** al cual le vamos a pasar la llamada a las acciones, y en el caso de la función de **cambiarCuenta** también le tendremos que pasar la nueva cuenta como parámetro de la función creadora de la acción.

/angular-ngrx-contador-lab/src/app/contador/contador.component.ts

```

import { Component, OnInit } from '@angular/core';
import { Store } from '@ngrx/store';
import { incrementar, cambiarCuenta, decrementar } from '../state/actions/contador.actions';
import { IAppState } from '../state/app.state';

@Component({
  selector: 'app-contador',
  standalone: true,
  imports: [],
  templateUrl: './contador.component.html',
  styleUrls: ['./contador.component.css']
})
export class ContadorComponent implements OnInit {
  cuenta: number = 0

  constructor(private store: Store<IAppState>) { }

```

```

ngOnInit(): void {
}

incrementar() {
  this.store.dispatch(incrementar())
}

decrementar() {
  this.store.dispatch(decrementar())
}

cambiarCuenta(event: any) {
  this.store.dispatch(cambiarCuenta({ cuenta: Number(event.target.value) }))
}
}

```

Ahora nos falta obtener la cuenta del Store para mostrarla, por lo que primero vamos a crear el selector que nos la va a retornar.

Dentro del archivo de **contador.selectors.ts** vamos a empezar creando una función que va a recibir el estado global y del cual vamos a sacar el estado perteneciente al contador.

/angular-ngrx-contador-lab/src/app/state/selectors/contador.selectors.ts

```

import { IAppState } from "../app.state";

const selectContador = (state: IAppState) => state.contadorState

```

Ahora vamos a crear el selector con la función **createSelector** al que le vamos a pasar el **selectContador** que acabamos de añadir como primera parámetro y después una función similar en la que recibiremos el estado del **contadorState** y sacaremos los datos que vamos a usar en el componente. En este caso, la cuenta.

/angular-ngrx-contador-lab/src/app/state/selectors/contador.selectors.ts

```

import { createSelector } from "@ngrx/store";
import { IAppState } from "../app.state";
import { IContadorState } from "../reducers/contador.reducer";

const selectContador = (state: IAppState) => state.contadorState

export const selectCuenta = createSelector(
  selectContador,
  (state: IContadorState) => state.cuenta
)

```

Ahora podemos volver al componente del Contador en el que vamos a pedir con la función **store.select** el selector que nos devuelve la cuenta.

Esto nos devuelve un observable por lo que nos vamos a suscribir para obtener la cuenta y asignarla a la propiedad que tenemos en el componente.

/angular-ngrx-contador-lab/src/app/contador/contador.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Store } from '@ngrx/store';
import { incrementar, cambiarCuenta, decrementar } from '../state/actions/contador.actions';
import { IAppState } from '../state/app.state';
import { selectCuenta } from '../state/selectors/contador.selectors';

@Component({
  selector: 'app-contador',
  templateUrl: './contador.component.html',
  standalone: true,
  imports: [],
  styleUrls: ['./contador.component.css']
})
export class ContadorComponent implements OnInit {
  cuenta: number = 0

  constructor(private store: Store<IAppState>) { }

  ngOnInit(): void {
    this.store.select(selectCuenta)
      .subscribe((cuenta: number) => {
        this.cuenta = cuenta
      })
  }

  incrementar() {
    this.store.dispatch(incrementar())
  }

  decrementar() {
    this.store.dispatch(decrementar())
  }

  cambiarCuenta(event: any) {
    this.store.dispatch(cambiarCuenta({ cuenta: Number(event.target.value) }))
  }
}
```

Ahora ya deberíamos de poder cambiar la cuenta pulsando sobre los botones y escribiendo un número sobre el campo numérico.

6.5. Effects

Los **effects** son una forma de poder separar la comunicación con los servicios de los componentes, dejando que sean los efectos los que interactúen con estos servicios cuando ciertas acciones se van despachando en la aplicación.

Los efectos se suelen utilizar para gestionar tareas como pedir datos, interactuar con otros elementos externos a los componentes...

Chapter 7. Internacionalización

La **internacionalización** consiste en preparar nuestra aplicación para que pueda utilizarse en cualquier parte del mundo, traduciendo el contenido de esta, incluso adaptando la presentación de ciertos datos a los formatos que se aplican en los distintos países.

Por ejemplo, los precios no se muestran igual en España que en EEUU. En España ponemos el símbolo de la moneda a la derecha del precio, mientras que en EEUU lo suelen poner a la izquierda.

Angular ya cuenta con su propio módulo de internacionalización, `@angular/localize`.

Para ir sacando las traducciones de las distintas etiquetas de la aplicación vamos a utilizar la directiva **i18n** sobre ellas. Con ello, el módulo de Angular conseguirá extraer a un archivo de traducciones todos estos textos, archivo en el que tendremos que poner cual es el lenguaje destino y cuales son las traducciones en dicho lenguaje.

Por otro lado, tendremos que configurar nuestra aplicación de Angular para que al lanzar el servidor de desarrollo, o al generar la aplicación para producción, esta pueda ser traducida.

7.1. Lab: internacionalización

En este laboratorio vamos a ver como añadir la parte de internacionalización a una aplicación para poder tenerla en distintos idiomas y dependiendo del idioma seleccionado que esta utilice unos formatos u otros por ejemplo para el pipe currency.

Lo primero que vamos a hacer es crearnos un proyecto nuevo de Angular lanzando el siguiente comando y seleccionaremos las opciones que mostramos a continuación:

```
$ ng new angular-18-internacionalizacion-lab
? Which stylesheet format would you like to use? CSS
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? N
```

Una vez creado el proyecto entramos desde el terminal a la carpeta que se ha creado ya que tendremos que lanzar una serie de comandos en ella:

```
$ cd angular-18-internacionalizacion-lab
```

Ahora vamos a eliminar el contenido del html del componente raíz (app.component.html) para rellenarlo con el siguiente:

/angular-18-internacionalizacion-lab/src/app/app.component.html

```
<h1>Internationalization</h1>
<p>Hello world</p>
<p>{{329.9 | currency}}</p>
```

Como vamos a utilizar el pipe currency, necesitamos importarlo en el componente de App.

/angular-18-internacionalizacion-lab/src/app/app.component.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    CurrencyPipe,
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
```

```
}
```

Estos elementos que acabamos de añadir son los que vamos a traducir a distintos idiomas.

El primer paso que hay que dar es añadir la librería de angular encargada de la internacionalización con el siguiente comando:

```
$ ng add @angular/localize
```

```
The package @angular/localize@18.2.14 will be installed and executed.  
Would you like to proceed? (Y/n) Y
```

Una vez instalada la librería de **@angular/localize**, vamos a añadir las directivas **i18n** en aquellas etiquetas sobre las que haya que realizar las traducciones. A esta directiva vamos a añadirle un identificador único para fijarlo en las traducciones.

```
/angular-18-internacionalizacion-lab/src/app/app.component.html
```

```
<h1 i18n="@@tituloApp">Internationalization</h1>  
<p i18n="@@saludoApp">Hello world</p>  
<p>{{329.9 | currency}}</p>
```

Ahora toca extraer los archivos de traducciones de nuestros componentes, y para ello tenemos que lanzar el siguiente comando:

```
$ ng extract-i18n --format=xlf --output-path src/i18n
```

Ahí le estamos indicando que nos genere los archivos de traducciones dentro de la carpeta **src/i18n**. Después de que el comando lanzado haya terminado de ejecutarse, podremos ver dicha carpeta.

En esta carpeta se ha debido de generar un archivo **messages.xlf**, el cual vamos a copiar por cada idioma al cual queramos traducir nuestra página. A todos ellos les pondremos el código de idioma en el nombre.

Nosotros realizaremos traducciones a inglés (por defecto), español y francés, por lo que tendremos los siguientes archivos de traducciones:

- src
 - i18n
 - messages.es.xlf
 - messages.fr.xlf

En estos archivos tendremos que añadir una etiqueta **target** por cada bloque de traducción con el texto traducido, además de añadir un atributo **target-language** con el código de idioma del archivo de traducciones.

Nuestros archivos quedarán como podemos ver a continuación:

/angular-18-internacionalizacion-lab/src/i18n/messages.es.xlf

```
<?xml version="1.0" encoding="UTF-8" ?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
  <file source-language="en-US" target-language="es" datatype="plaintext" original="ng2.template">
    <body>
      <trans-unit id="tituloApp" datatype="html">
        <source>Internationalization</source>
        <target>Internacionalización</target>
        <context-group purpose="location">
          <context context-type="sourcefile">src/app/app.component.html</context>
          <context context-type="linenumber">1,2</context>
        </context-group>
      </trans-unit>
      <trans-unit id="saludoApp" datatype="html">
        <source>Hello world</source>
        <target>Hola mundo</target>
        <context-group purpose="location">
          <context context-type="sourcefile">src/app/app.component.html</context>
          <context context-type="linenumber">2,3</context>
        </context-group>
      </trans-unit>
    </body>
  </file>
</xliff>
```

/angular-18-internacionalizacion-lab/src/i18n/messages.fr.xlf

```
<?xml version="1.0" encoding="UTF-8" ?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
  <file source-language="en-US" target-language="fr" datatype="plaintext" original="ng2.template">
    <body>
      <trans-unit id="tituloApp" datatype="html">
        <source>Internationalization</source>
        <target>Internationalisation</target>
        <context-group purpose="location">
          <context context-type="sourcefile">src/app/app.component.html</context>
          <context context-type="linenumber">1,2</context>
        </context-group>
      </trans-unit>
      <trans-unit id="saludoApp" datatype="html">
        <source>Hello world</source>
        <target>Salut monde</target>
        <context-group purpose="location">
          <context context-type="sourcefile">src/app/app.component.html</context>
          <context context-type="linenumber">2,3</context>
        </context-group>
      </trans-unit>
    </body>
  </file>
</xliff>
```

Ahora tenemos que ya tenemos nuestro componente y sus traducciones, toca modificar el archivo de **angular.json** para añadir estos idiomas en la configuración del proyecto.

El primer cambio que vamos a realizar en este archivo consiste en añadir la propiedad **i18n.locales** para indicar que idiomas vamos a permitir, y donde se encuentran sus archivos de traducciones.

/angular-18-internacionalizacion-lab/angular.json

```
{
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
  "version": 1,
  "newProjectRoot": "projects",
  "projects": {
    "angular-18-internacionalizacion-lab": {
      "projectType": "application",
      "schematics": {},
      "root": "",
      "sourceRoot": "src",
      "prefix": "app",
      "i18n": {
        "sourceLocale": "en-US",
        "locales": {
          "es": {
            "translation": "src/i18n/messages.es.xlf"
          },
          "fr": {
            "translation": "src/i18n/messages.fr.xlf"
          }
        }
      },
      "architect": { ... }
    }
  },
  "defaultProject": "angular-18-internacionalizacion-lab"
}
```

Ahora dentro de **architect.build.configurations** tenemos que añadir la configuración del **locale** de Angular para cada uno de estos lenguajes que hemos añadido en el paso anterior. De esta forma cuando lancemos comandos de construcción del proyecto, podremos seleccionar para que idioma queremos traducirlo pasandole la opción de **--configuration=es**.

/angular-18-internacionalizacion-lab/angular.json

```
{
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
  "version": 1,
  "newProjectRoot": "projects",
  "projects": {
    "angular-18-internacionalizacion-lab": {
      "projectType": "application",
      "architect": {
        "build": {
          "configurations": [
            {
              "name": "es",
              "locale": "es"
            },
            {
              "name": "fr",
              "locale": "fr"
            }
          ]
        }
      }
    }
  }
}
```

```

"schematics": {},
"root": "",
"sourceRoot": "src",
"prefix": "app",
"i18n": {
  "sourceLocale": "en-US",
  "locales": {
    "es": {
      "translation": "src/i18n/messages.es.xlf"
    },
    "fr": {
      "translation": "src/i18n/messages.fr.xlf"
    }
  }
},
"architect": {
  "build": {
    "builder": "@angular-devkit/build-angular:application",
    "options": { ... },
    "configurations": {
      "es": {
        "localize": ["es"]
      },
      "fr": {
        "localize": ["fr"]
      },
      "production": { ... },
      "development": { ... }
    },
    "defaultConfiguration": "production"
  },
  "serve": { ... },
  "extract-i18n": { ... },
  "test": { ... }
}
},
"defaultProject": "angular-18-internacionalizacion-lab"
}

```

Vamos a hacer exactamente lo mismo para poder levantar el proyecto con el servidor de desarrollo pero con la configuración de los distintos idiomas. Esta vez añadiremos los cambios en **architect.serve.configurations** y haremos que coja la configuración añadida en el anterior paso.

/angular-18-internacionalizacion-lab/angular.json

```
{
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
  "version": 1,
  "newProjectRoot": "projects",
}
```

```
"projects": {  
  "angular-18-internacionalizacion-lab": {  
    "projectType": "application",  
    "schematics": {},  
    "root": "",  
    "sourceRoot": "src",  
    "prefix": "app",  
    "i18n": {  
      "sourceLocale": "en-US",  
      "locales": {  
        "es": {  
          "translation": "src/i18n/messages.es.xlf"  
        },  
        "fr": {  
          "translation": "src/i18n/messages.fr.xlf"  
        }  
      }  
    },  
    "architect": {  
      "build": {  
        "builder": "@angular-devkit/build-angular:application",  
        "options": {  
          "outputPath": "dist/angular-18-internacionalizacion-lab",  
          "index": "src/index.html",  
          "browser": "src/main.ts",  
          "polyfills": [  
            "zone.js",  
            "@angular/localize/init"  
          ],  
          "tsConfig": "tsconfig.app.json",  
          "assets": [  
            {  
              "glob": "**/*",  
              "input": "public"  
            }  
          ],  
          "styles": [  
            "src/styles.css"  
          ],  
          "scripts": []  
        },  
        "configurations": {  
          "es": {  
            "localize": ["es"]  
          },  
          "fr": {  
            "localize": ["fr"]  
          },  
          "production": { ... },  
          "development": { ... }  
        },  
        "options": {}  
      }  
    }  
  }  
}
```

```
  "defaultConfiguration": "production"
},
"serve": {
  "builder": "@angular-devkit/build-angular:dev-server",
  "configurations": {
    "es": {
      "buildTarget": "angular-18-internacionalizacion-lab:build:es"
    },
    "fr": {
      "buildTarget": "angular-18-internacionalizacion-lab:build:fr"
    },
    "production": {
      "buildTarget": "angular-18-internacionalizacion-lab:build:production"
    },
    "development": {
      "buildTarget": "angular-18-internacionalizacion-lab:build:development"
    }
  },
  "defaultConfiguration": "development"
},
"extract-i18n": { ... },
"test": { ... }
}
},
},
"defaultProject": "angular-18-internacionalizacion-lab"
}
```

Ahora ya podemos probar a levantar la aplicación en modo desarrollo con cualquiera de los 3 idiomas lanzando alguno de los siguientes comandos:

```
$ ng s
$ ng s --configuration=es
$ ng s --configuration=fr
```

Internationalization

Hello world

\$329.90

Internacionalización

Hola mundo

329,90 US\$

Internationalisation

Salut monde

329,90 \$US

Como podemos observar en las anteriores imágenes, las traducciones han salido bien, además de que el formato para los pipes se está cogiendo de forma correcta según el locale.

Por ejemplo, para español y francés podemos ver que el símbolo de las monedas aparece a la derecha en lugar de hacerlo a la izquierda como ocurre con el inglés (el que se usa por defecto).

Por último vamos a generar los archivos estáticos de la aplicación junto a las traducciones, pero antes, necesitamos un menú con el que poder navegar desde la página en un idioma a otro cualquiera de los que hemos puesto.

Esto vamos a añadirlo en el componente App.

Empezamos añadiendo, en el archivo de typescript, una lista con los distintos código de los idiomas y el texto que queremos que aparezca en los enlaces.

/angular-18-internacionalizacion-lab/src/app/app.component.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    CurrencyPipe,
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  localesList = [
    { code: 'en-US', label: 'English' },
    { code: 'es', label: 'Spanish' },
    { code: 'fr', label: 'French' }
  ]
}
```

Ahora vamos a añadir la lista de enlaces al final del html que ya teníamos.

/angular-18-internacionalizacion-lab/src/app/app.component.html

```
<h1 i18n="@@tituloApp">Internationalization</h1>
<p i18n="@@saludoApp">Hello world</p>
<p>{{329.9 | currency}}</p>

<ul>
  @for (locale of localesList; track locale.code) {
    <li>
      <a href="/{{locale.code}}/">
        {{locale.label}}
      </a>
    </li>
  }

```

```
</ul>
```

Ahora vamos a generar los archivos estáticos finales para todos los idiomas configurados con el siguiente comando:

```
$ ng build --localize
```

Una vez que termina de ejecutarse este comando, veremos una carpeta **dist/angular-18-internacionalizacion-lab** dentro de la cual tendremos una carpeta **browser** con los códigos de cada idioma, y a su vez dentro de estas carpetas tendremos la aplicación con sus distintas traducciones.

Podemos levantar esta aplicación con un servidor http para ver que todo esté bien.

Vamos a navegar hasta esta carpeta desde el terminal y vamos a lanzar los siguientes comandos:

```
$ npx http-serve dist/angular-18-internacionalizacion-lab/browser
```

Entramos en <http://localhost:8080/en-US/>, y al ir pulsando sobre los distintos enlaces podremos ver como vamos navegando entre las 3 aplicaciones que hemos creado, cada una en un idioma distinto.

7.2. Lab: Internacionalización con ngx-translate

En este laboratorio vamos a ver como internacionalizar una aplicación de Angular usando la librería de **ngx-translate**.

Lo primero que vamos a hacer es crearnos un proyecto nuevo de Angular lanzando el siguiente comando y seleccionaremos las opciones que mostramos a continuación:

```
$ ng new angular-18-internacionalizacion-ngx-translate-lab
? Which stylesheet format would you like to use? CSS
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation
(SSG/Prerendering)? N
```

Una vez creado el proyecto entramos desde el terminal a la carpeta que se ha creado ya que tendremos que lanzar una serie de comandos en ella:

```
$ cd angular-18-internacionalizacion-ngx-translate-lab
```

Tenemos que instalar las dependencias de **ngx-translate** y levantar el servidor con los siguientes comandos:

```
$ npm install --save @ngx-translate/core @ngx-translate/http-loader
$ ng s
```

Ahora vamos a crear la página a la que después le añadiremos las traducciones necesarias.

Empezaremos por declarar una serie de propiedades en el componente App.

/angular-18-internacionalizacion-ngx-translate-lab/src/app/app.component.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  lenguajes: Array<string> = ['es', 'en']
  lenguajeSeleccionado: string = 'es'

  nombre: string = 'Charly'
  fechaActual: Date = new Date()
  hamburguesas: Array<any> = [
```

```

        { nombre: 'Burger Barney', precio: 9.95 },
        { nombre: 'Burger Mando', precio: 10.95 },
        { nombre: 'Burger Groot', precio: 11.95 },
        { nombre: 'Burger Doble Smash', precio: 12.95 },
        { nombre: 'Burger Michelangelo Zayken', precio: 12.95 },
    ]
}

```

Como vamos a utilizar un desplegable para seleccionar el lenguaje a utilizar, vamos a añadir una función para actualizar el valor de la propiedad **lenguajeSeleccionado** al elegido.

/angular-18-internacionalizacion-ngx-translate-lab/src/app/app.component.ts

```

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  lenguajes: Array<string> = ['es', 'en']
  lenguajeSeleccionado: string = 'es'

  nombre: string = 'Charly'
  fechaActual: Date = new Date()
  hamburguesas: Array<any> = [
    { nombre: 'Burger Barney', precio: 9.95 },
    { nombre: 'Burger Mando', precio: 10.95 },
    { nombre: 'Burger Groot', precio: 11.95 },
    { nombre: 'Burger Doble Smash', precio: 12.95 },
    { nombre: 'Burger Michelangelo Zayken', precio: 12.95 },
  ]

  cambiarLenguaje(event: any) {
    this.lenguajeSeleccionado = event.target.value
  }
}

```

Ahora vamos a añadir el siguiente contenido en la plantilla del componente App para mostrar la bienvenida al usuario, la fecha actual, el desplegable de idiomas y el listado de hamburguesas.

/angular-18-internacionalizacion-ngx-translate-lab/src/app/app.component.html

```

<h1>Hamburguesas La Nave</h1>
<p>{{fechaActual}}</p>

```

```

<select (change)="cambiarLenguaje($event)">
  @for (lenguaje of lenguajes; track $index) {
    <option [value]="lenguaje" [selected]="lenguaje === lenguajeSeleccionado">{{lenguaje}}</option>
  }
</select>

<h2>Bienvenid@ {{nombre}}</h2>

@for (hamburguesa of hamburguesas; track $index) {
  <div>
    <h3>{{hamburguesa.nombre}}</h3>
    <p>Precio: {{hamburguesa.precio}}</p>
  </div>
}

```

Una vez tenemos la estructura de la página, vamos a añadir la configuración inicial para las traducciones.

Dentro del archivo de configuración de la aplicación, vamos a añadir tanto el HttpClient como el TranslateService para las traducciones.

Para el servicio de traducciones, vamos a utilizar **provideTranslateService** pasándole un objeto de configuración con el lenguaje por defecto **fallbackLang** y el loader para cargar las traducciones, donde le indicamos que lo haremos con Http y donde se encuentran los archivos de traducciones indicando la ruta con el **prefix** y la extensión de los archivos que hay que cargar con el **suffix**.

/angular-18-internacionalizacion-ngx-translate-lab/src/app/app.config.ts

```

import { ApplicationConfig, importProvidersFrom, provideZoneChangeDetection } from '@angular/core';
import { provideRouter } from '@angular/router';
import { provideTranslateService } from '@ngx-translate/core';
import { provideTranslateHttpLoader } from '@ngx-translate/http-loader';

import { routes } from './app.routes';
import { provideHttpClient } from '@angular/common/http';

export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes),
    provideHttpClient(),
    provideTranslateService({
      fallbackLang: 'es',
      loader: provideTranslateHttpLoader({
        prefix: './i18n/',
        suffix: '.json'
      })
    })
  ]
};

```

Una vez tenemos el módulo, vamos a crear los archivos de traducciones. Por defecto se buscan en la carpeta **/public** y nosotros le hemos indicado que estarán en **i18n/[lang].json** donde el **[lang]** sería

el código del lenguaje que vamos a usar en la aplicación para guardar el lenguaje seleccionado en el que queremos las traducciones.

Por tanto, vamos a crearnos dos archivos:

- `/public/i18n/es.json`
- `/public/i18n/en.json`

En estos archivos tenemos que poner las traducciones asociándolas a unas claves JSON que luego usaremos en nuestras plantillas con un pipe **translate** que nos ofrece esta librería.

`/angular-18-internacionalizacion-ngx-translate-lab/src/public/i18n/en.json`

```
{  
  "HOME": {  
    "title": "La Nave's Hamburgers",  
    "welcome": "Welcome {{name}}",  
    "LANGUAGE_DROPDOWN": {  
      "en": "English",  
      "es": "Spanish"  
    },  
    "price": "Price"  
  }  
}
```

`/angular-18-internacionalizacion-ngx-translate-lab/src/public/i18n/es.json`

```
{  
  "HOME": {  
    "title": "Hamburguesas La Nave",  
    "welcome": "Bienvenid@ {{name}}",  
    "LANGUAGE_DROPDOWN": {  
      "en": "Inglés",  
      "es": "Español"  
    },  
    "price": "Precio"  
  }  
}
```

Una vez tenemos las traducciones, nos vamos al componente App, en el que vamos a añadir el constructor y le vamos a inyectar el servicio de traducciones, **TranslateService** para añadir los lenguajes (con la función **addLangs**) e inicializar el **lenguajeSeleccionado** con el valor que habíamos puesto como valor por defecto en la configuración del módulo (con la función **getFallbackLang**).

`/angular-18-internacionalizacion-ngx-translate-lab/src/app/app.component.ts`

```
import { Component } from '@angular/core';  
import { TranslateService } from '@ngx-translate/core';
```

```

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'
})
export class AppComponent {
  lenguajes: Array<string> = ['es', 'en']
  lenguajeSeleccionado: string = 'es'

  nombre: string = 'Charly'
  fechaActual: Date = new Date()
  hamburguesas: Array<any> = [
    { nombre: 'Burger Barney', precio: 9.95 },
    { nombre: 'Burger Mando', precio: 10.95 },
    { nombre: 'Burger Groot', precio: 11.95 },
    { nombre: 'Burger Doble Smash', precio: 12.95 },
    { nombre: 'Burger Michelangelo Zayken', precio: 12.95 },
  ]
}

constructor(
  private translateService: TranslateService,
) {
  this.translateService.addLangs(this.lenguajes)
  this.lenguajeSeleccionado = this.translateService.getFallbackLang() || 'es'
}

cambiarLenguaje(event: any) {
  this.lenguajeSeleccionado = event.target.value
}
}

```

Ahora al cambiar de lenguaje, le vamos a indicar a la librería que tiene que usar las traducciones pertenecientes al lenguaje seleccionado. Usaremos la función de **use** del servicio de traducciones.

/angular-18-internacionalizacion-ngx-translate-lab/src/app/app.component.ts

```

import { Component } from '@angular/core';
import { TranslateService } from '@ngx-translate/core';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'
})
export class AppComponent {

```

```

lenguajes: Array<string> = ['es', 'en']
lenguajeSeleccionado: string = 'es'

nombre: string = 'Charly'
fechaActual: Date = new Date()
hamburguesas: Array<any> = [
  { nombre: 'Burger Barney', precio: 9.95 },
  { nombre: 'Burger Mando', precio: 10.95 },
  { nombre: 'Burger Groot', precio: 11.95 },
  { nombre: 'Burger Doble Smash', precio: 12.95 },
  { nombre: 'Burger Michelangelo Zayken', precio: 12.95 },
]

constructor(
  private translateService: TranslateService,
) {
  this.translateService.addLangs(this.lenguajes)
  this.lenguajeSeleccionado = this.translateService.getFallbackLang() || 'es'
}

cambiarLenguaje(event: any) {
  this.lenguajeSeleccionado = event.target.value
  this.translateService.use(this.lenguajeSeleccionado)
}
}

```

Como vamos a necesitar utilizar el pipe **translate** para traducir textos en la plantilla, vamos a importarlo en el componente.

/angular-18-internacionalizacion-ngx-translate-lab/src/app/app.component.ts

```

import { Component } from '@angular/core';
import { TranslatePipe, TranslateService } from '@ngx-translate/core';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    TranslatePipe,
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'
})
export class AppComponent {
  lenguajes: Array<string> = ['es', 'en']
  lenguajeSeleccionado: string = 'es'

  nombre: string = 'Charly'
  fechaActual: Date = new Date()
  hamburguesas: Array<any> = [

```

```

        { nombre: 'Burger Barney', precio: 9.95 },
        { nombre: 'Burger Mando', precio: 10.95 },
        { nombre: 'Burger Groot', precio: 11.95 },
        { nombre: 'Burger Doble Smash', precio: 12.95 },
        { nombre: 'Burger Michelangelo Zayken', precio: 12.95 },
    ]

constructor(
    private translateService: TranslateService,
) {
    this.translateService.addLangs(this.lenguajes)
    this.lenguajeSeleccionado = this.translateService.getFallbackLang() || 'es'
}

cambiarLenguaje(event: any) {
    this.lenguajeSeleccionado = event.target.value
    this.translateService.use(this.lenguajeSeleccionado)
}
}

```

Ahora vamos a utilizar el pipe **translate** en nuestra plantilla, allí donde queremos realizar las traducciones. Este pipe se usa sobre un string con la ruta del documento JSON hasta la traducción que queremos.

Vamos a empezar por cambiar el título, por lo que usaremos como clave **HOME.title** como se muestra a continuación.

/angular-18-internacionalizacion-ngx-translate-lab/src/app/app.component.html

```

<h1>{{ 'HOME.title' | translate }}</h1>

<p>{{fechaActual}}</p>

<select (change)="cambiarLenguaje($event)">
    @for (lenguaje of lenguajes; track $index) {
        <option [value]="lenguaje" [selected]="lenguaje === lenguajeSeleccionado">{{lenguaje}}</option>
    }
</select>

<h2>Bienvenid@ {{nombre}}</h2>

@for (hamburguesa of hamburguesas; track $index) {
    <div>
        <h3>{{hamburguesa.nombre}}</h3>
        <p>Precio: {{hamburguesa.precio}}</p>
    </div>
}

```

Ahora si cambiamos el lenguaje desde el desplegable, deberíamos de ver como cambia el título de la aplicación.

Vamos a añadir las traducciones para el precio y el desplegable de la misma forma, pero cada una

con su clave.

/angular-18-internacionalizacion-ngx-translate-lab/src/app/app.component.html

```
<h1>{{ 'HOME.title' | translate }}</h1>

<p>{{fechaActual}}</p>

<select (change)="cambiarLenguaje($event)">
  @for (lenguaje of lenguajes; track $index) {
    <option [value]="lenguaje" [selected]="lenguaje === lenguajeSeleccionado">{{ 'HOME.LANGUAGE_DROPDOWN.' + lenguaje | translate }}</option>
  }
</select>

<h2>Bienvenid&#64; {{nombre}}</h2>

@for (hamburguesa of hamburguesas; track $index) {
  <div>
    <h3>{{hamburguesa.nombre}}</h3>
    <p>{{ 'HOME.price' | translate }}: {{hamburguesa.precio}}</p>
  </div>
}
```

Ahora queremos añadir la traducción para el título de bienvenida, pero en este caso, vamos a hacerlo de otra forma, ya que en la propia traducción le habíamos puesto una interpolación de string con la clave **name**.

Para pasarle el nombre y que se devuelva junto a la traducción, le tenemos que pasar al pipe un parámetro que es un objeto de JS, donde la clave será **name** y el valor la propiedad **nombre**.

/angular-18-internacionalizacion-ngx-translate-lab/src/app/app.component.html

```
<h1>{{ 'HOME.title' | translate }}</h1>

<p>{{fechaActual}}</p>

<select (change)="cambiarLenguaje($event)">
  @for (lenguaje of lenguajes; track $index) {
    <option [value]="lenguaje" [selected]="lenguaje === lenguajeSeleccionado">{{ 'HOME.LANGUAGE_DROPDOWN.' + lenguaje | translate }}</option>
  }
</select>

<h2>{{ 'HOME.welcome' | translate:{name: nombre} }}</h2>

@for (hamburguesa of hamburguesas; track $index) {
  <div>
    <h3>{{hamburguesa.nombre}}</h3>
    <p>{{ 'HOME.price' | translate }}: {{hamburguesa.precio}}</p>
  </div>
}
```

Lo último que nos queda por hacer con lo que tenemos es el poder cambiar tanto el formato de la fecha y el del precio con los pipes de **date** y **currency** teniendo en cuenta el idioma seleccionado.

Para ello, vamos a empezar por registrar los distintos locales que vamos a usar en el constructor del módulo de App. Usaremos la función de **registerLocaleData** pasándole como parámetros el **locale**

que hay que importar, y un identificador para dicho locale. Esto lo haremos en el archivo **main.ts** antes de arrancar la aplicación.

/angular-18-internacionalizacion-ngx-translate-lab/src/main.ts

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

import { registerLocaleData } from '@angular/common'
import localeEn from '@angular/common/locales/en'
import localeEs from '@angular/common/locales/es'

registerLocaleData(localeEn, 'en')
registerLocaleData(localeEs, 'es')

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

Como vamos a utilizar los pipes de las fechas y las monedas, tenemos que importarlos en el componente primero.

/angular-18-internacionalizacion-ngx-translate-lab/src/app/app.component.ts

```
import { CurrencyPipe, DatePipe } from '@angular/common';
import { Component } from '@angular/core';
import { TranslatePipe, TranslateService } from '@ngx-translate/core';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    TranslatePipe,
    DatePipe,
    CurrencyPipe,
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'
})

export class AppComponent {
  lenguajes: Array<string> = ['es', 'en']
  lenguajeSeleccionado: string = 'es'

  nombre: string = 'Charly'
  fechaActual: Date = new Date()
  hamburguesas: Array<any> = [
    { nombre: 'Burger Barney', precio: 9.95 },
    { nombre: 'Burger Mando', precio: 10.95 },
    { nombre: 'Burger Groot', precio: 11.95 },
  ]
```

```

        { nombre: 'Burger Doble Smash', precio: 12.95 },
        { nombre: 'Burger Michelangelo Zayken', precio: 12.95 },
    ]

constructor(
    private translateService: TranslateService,
) {
    this.translateService.addLangs(this.lenguajes)
    this.lenguajeSeleccionado = this.translateService.getFallbackLang() || 'es'
}

cambiarLenguaje(event: any) {
    this.lenguajeSeleccionado = event.target.value
    this.translateService.use(this.lenguajeSeleccionado)
}
}

```

Una vez hecho esto, ya podemos ir a aplicar los dos pipes comentados antes a los que les pasaremos como parámetro el lenguaje seleccionado. En el caso del **currency** es el cuarto, mientras que para el de **date** es el tercero.



Para poder utilizar estos parámetros de locale en los pipes de date y currency, es necesario registrarlos como hemos hecho anteriormente con la función de **registerLocaleData**.

/angular-18-internacionalizacion-ngx-translate-lab/src/app/app.component.html

```

<h1>{{ 'HOME.title' | translate }}</h1>

<p>{{fechaActual | date:'medium':undefined:lenguajeSeleccionado}}</p>

<select (change)="cambiarLenguaje($event)">
    @for (lenguaje of lenguajes; track $index) {
        <option [value]="lenguaje" [selected]="lenguaje === lenguajeSeleccionado">{{ 'HOME.LANGUAGE_DROPDOWN.' + lenguaje | translate }}</option>
    }
</select>

<h2>{{ 'HOME.welcome' | translate:{name: nombre} }}</h2>

@for (hamburguesa of hamburguesas; track $index) {
    <div>
        <h3>{{hamburguesa.nombre}}</h3>
        <p>{{ 'HOME.price' | translate }}: {{hamburguesa.precio | currency:'EUR':undefined:undefined:lenguajeSeleccionado}}</p>
    </div>
}

```

Para ir actualizando la fecha automáticamente, vamos a usar el observable **interval** para actualizar su valor cada segundo que pasa.

/angular-18-internacionalizacion-ngx-translate-lab/src/app/app.component.ts

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

```

import { Component } from '@angular/core';
import { TranslatePipe, TranslateService } from '@ngx-translate/core';
import { interval } from 'rxjs';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    TranslatePipe,
    DatePipe,
    CurrencyPipe,
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  lenguajes: Array<string> = ['es', 'en']
  lenguajeSeleccionado: string = 'es'

  nombre: string = 'Charly'
  fechaActual: Date = new Date()
  hamburguesas: Array<any> = [
    { nombre: 'Burger Barney', precio: 9.95 },
    { nombre: 'Burger Mando', precio: 10.95 },
    { nombre: 'Burger Groot', precio: 11.95 },
    { nombre: 'Burger Doble Smash', precio: 12.95 },
    { nombre: 'Burger Michelangelo Zayken', precio: 12.95 },
  ]
}

constructor(
  private translateService: TranslateService,
) {
  this.translateService.addLangs(this.lenguajes)
  this.lenguajeSeleccionado = this.translateService.getFallbackLang() || 'es'

  interval(1000)
    .subscribe(() => {
      this.fechaActual = new Date()
    })
}

cambiarLenguaje(event: any) {
  this.lenguajeSeleccionado = event.target.value
  this.translateService.use(this.lenguajeSeleccionado)
}
}

```

Por último, vamos a añadir una traducción de fallback para cuando se intente traducir algo que no tiene configurada su traducción aparezca un mensaje por defecto y podamos darnos cuenta de ello para agregarla.

Para ello, vamos a añadir una clave más en los archivos JSON de traducciones.

/angular-18-internacionalizacion-ngx-translate-lab/src/public/i18n/en.json

```
{  
  "HOME": {  
    "title": "La Nave's Hamburgers",  
    "welcome": "Welcome {{name}}",  
    "LANGUAGE_DROPDOWN": {  
      "en": "English",  
      "es": "Spanish"  
    },  
    "price": "Price"  
  },  
  "NO_TRANSLATION": "We don't have the translation"  
}
```

/angular-18-internacionalizacion-ngx-translate-lab/src/public/i18n/es.json

```
{  
  "HOME": {  
    "title": "Hamburguesas La Nave",  
    "welcome": "Bienvenid@ {{name}}",  
    "LANGUAGE_DROPDOWN": {  
      "en": "Inglés",  
      "es": "Español"  
    },  
    "price": "Precio"  
  },  
  "NO_TRANSLATION": "No tenemos la traducción"  
}
```

Ahora en el archivo de configuración vamos a añadir la clave **missingTranslationHandler** dentro del provider de traducciones donde le vamos a indicar que cuando se pida el token **MissingTranslationHandler** se use la clase **MyMissingTranslationHandler** que vamos a crear en un archivo **my-missing-translation-handler.ts**.

/angular-18-internacionalizacion-ngx-translate-lab/src/app/app.config.ts

```
import { ApplicationConfig, importProvidersFrom, provideZoneChangeDetection } from '@angular/core';  
import { provideRouter } from '@angular/router';  
import { MissingTranslationHandler, provideTranslateService } from '@ngx-translate/core';  
import { provideTranslateHttpLoader } from '@ngx-translate/http-loader';  
  
import { routes } from './app.routes';  
import { provideHttpClient } from '@angular/common/http';  
import { MyMissingTranslationHandler } from './my-missing-translation-handler';  
  
export const appConfig: ApplicationConfig = {  
  providers: [  
    {  
      provide: MissingTranslationHandler,  
      useValue: MyMissingTranslationHandler  
    }  
  ]  
}
```

```

provideZoneChangeDetection({ eventCoalescing: true }),
provideRouter(routes),
provideHttpClient(),
provideTranslateService({
  fallbackLang: 'es',
  loader: provideTranslateHttpLoader({
    prefix: './i18n/',
    suffix: '.json'
  }),
  missingTranslationHandler: {
    provide: MissingTranslationHandler,
    useClass: MyMissingTranslationHandler,
  },
}),
];
);

```

Ahora creamos el archivo con la clase **MyMissingTranslationHandler** que va a extender de **MissingTranslationHandler**. Tenemos que implementar un método **handle** y aquello que devuelva dicho método será lo que se va a mostrar como traducción.

Aquí lo que haremos es acceder al servicio de traducciones que se recibe en lo **params** y vamos a pedir el texto de **NO_TRANSLATION** que hemos añadido en los archiso JSON.

/angular-18-internacionalizacion-ngx-translate-lab/src/app/my-missing-translation-handler.ts

```

import { MissingTranslationHandler, MissingTranslationHandlerParams } from "@ngx-translate/core";

export class MyMissingTranslationHandler extends MissingTranslationHandler {
  handle(params: MissingTranslationHandlerParams) {
    return params.translateService.get('NO_TRANSLATION')
  }
}

```

Ahora vamos a la plantilla y vamos a añadir un párrafo con una clave que no existe para ver que se muestra el mensaje que hemos puesto antes.

/angular-18-internacionalizacion-ngx-translate-lab/src/app/app.component.html

```

<h1>{{ 'HOME.title' | translate }}</h1>

<p>{{fechaActual | date:'medium':undefined:lenguajeSeleccionado}}</p>

<select (change)="cambiarLenguaje($event)">
  @for (lenguaje of lenguajes; track $index) {
    <option [value]="lenguaje" [selected]="lenguaje === lenguajeSeleccionado">{{ 'HOME.LANGUAGE_DROPDOWN.' + lenguaje | translate }}</option>
  }
</select>

<h2>{{ 'HOME.welcome' | translate:{name: nombre} }}</h2>

@for (hamburguesa of hamburguesas; track $index) {
  <div>
    <h3>{{hamburguesa.nombre}}</h3>
    <p>{{ 'HOME.price' | translate }}: {{hamburguesa.precio | translate }}</p>
  </div>
}

```

```
currency: 'EUR':undefined:lenguajeSeleccionado}}}</p>
</div>
}

<p>{{ 'HOME.no-existe' | translate }}</p>
```