

PROGRAMACIÓN WEB EN
ENTORNO CLIENTE

ANGULAR

ÍNDICE

INTRODUCCIÓN	3
ESTRUCTURA DE UN PROYECTO ANGULAR.....	3
A. INSTALANDO LAS DEPENDENCIAS NECESARIAS Y LANZANDO LA APP	4
B. CONTENIDO DE LA APLICACIÓN.....	5
C. CAMBIANDO EL TÍTULO DE LA PÁGINA Y DEL CONTENIDO DE <app-root>	7
COMPONENTES	8
A. AÑADIR EL COMPONENTE AL LAYOUT DE LA APP	9
B. MODIFICAR EL CONTENIDO DEL TEMPLATE DE app-root	9
C. AÑADIENDO CAMBIOS EN EL COMPONENTE Home	10
D. AÑADIENDO UN NUEVO COMPONENTE HousingLocation	11
INTERFACES.....	12
A. PROBANDO LA INTERFAZ EN EL COMPONENTE Home.....	13
@INPUT	14
VINCULACIÓN DE PROPIEDADES	15
INTERPOLACIÓN	15
DIRECTIVAS	17
A. USANDO UNA DIRECTIVA ngFor EN NUESTRA APLICACIÓN	17
SERVICIOS	18
A. CREACIÓN DE UN SERVICIO	19
B. AÑADIENDO DATOS ESTÁTICOS AL SERVICIO.....	20
C. INYECTAR EL SERVICIO DENTRO DEL COMPONENTE Home.....	20
ENRUTADO.....	21
A. CREAR UN COMPONENTE DE DETALLES POR DEFECTO	21
B. AÑADIENDO ENRUTADO A NUESTRA APLICACIÓN	22
C. CONFIGURANDO EL FICHERO DE RUTAS	23
D. EDITANDO LAS CARDS PARA QUE AÑADAN EL LINK A DETAILS.....	24
E. EDITANDO EL COMPONENTE DETAILS	25
F. ÚLTIMAS CONFIGURACIONES DEL COMPONENTE DETAILS.....	26
FORMULARIOS	28
FUNCION DE BÚSQUEDA	30
COMUNICACIÓN HTTP	32

INTRODUCCIÓN

Angular es un marco de desarrollo de código abierto utilizado para construir aplicaciones web de una sola página (SPA) y aplicaciones web dinámicas. Fue desarrollado y es mantenido por Google. Angular proporciona una estructura y un conjunto de herramientas para simplificar el desarrollo de aplicaciones web complejas al facilitar la manipulación del DOM, la gestión del estado de la aplicación, la manipulación de eventos y la gestión de la comunicación con el servidor.

Algunas características clave de Angular incluyen:

1. **Binding bidireccional:** Angular facilita la sincronización automática entre el modelo y la vista, lo que significa que los cambios en el modelo se reflejan automáticamente en la vista y viceversa.
2. **Inyección de dependencias:** Angular utiliza un sistema de inyección de dependencias que facilita la gestión y la organización de los componentes de la aplicación.
3. **Directivas:** Las directivas en Angular permiten extender el HTML con nuevas funcionalidades y comportamientos personalizados.
4. **Módulos:** Angular organiza la aplicación en módulos, que son conjuntos de componentes, servicios y otros elementos relacionados.
5. **Routing:** Angular proporciona un sistema de enrutamiento que permite la navegación entre diferentes vistas o componentes sin necesidad de recargar toda la página.
6. **Servicios:** Los servicios en Angular son singleton y se utilizan para encapsular la lógica de negocio, compartir datos entre componentes y gestionar la comunicación con el servidor.
7. **TypeScript:** Angular está construido con TypeScript, un superset de JavaScript que agrega tipado estático al lenguaje.

El uso de Angular puede facilitar el desarrollo de aplicaciones web escalables y mantenibles, especialmente aquellas que requieren una estructura organizada y funcionalidades avanzadas.

ESTRUCTURA DE UN PROYECTO ANGULAR

Para iniciarnos en Angular vamos a empezar utilizando una aplicación ya creada por Angular, esta app se puede descargar desde la [web oficial](#). Haciendo clic en “download example”.

Starting code: [live example](#) / [download example](#)

Completed code: [live example](#) / [download example](#)

Una vez descarga descomprimos y renombramos el directorio, por ejemplo, lo llamaremos first-app, una vez hecho esto abrimos la app en nuestro IDE.

A. INSTALANDO LAS DEPENDENCIAS NECESARIAS Y LANZANDO LA APP

Una vez inicializado el proyecto, debemos instalar las dependencias necesarias para que nuestra app funcione, para ello debemos utilizar el comando:

`npm install --force`

```
pwsh D:\DAW\2DAW\02 PRÁCTICAS Y EJERCICIOS\07 ANGULAR\first-app
> npm install --force
npm WARN using --force Recommended protections disabled.
npm WARN ERESOLVE overriding peer dependency
npm WARN ERESOLVE overriding peer dependency
npm WARN While resolving: @angular-devkit/build-angular@17.0.9
npm WARN Found: typescript@4.9.5
npm WARN node_modules/typescript
npm WARN   peer typescript@>=5.2 <5.3 from @ngtools/webpack@17.0.9
npm WARN   node_modules/@angular-devkit/build-angular/node_modules/@ngtools/webpack
npm WARN   @ngtools/webpack@17.0.9 from @angular-devkit/build-angular@17.0.9
npm WARN   node_modules/@angular-devkit/build-angular
npm WARN Could not resolve dependency:
npm WARN   peer typescript@>=5.2 <5.3 from @angular-devkit/build-angular@17.0.9
npm WARN   node_modules/@angular-devkit/build-angular
npm WARN   dev @angular-devkit/build-angular@17.0.9 from the root project
npm WARN Conflicting peer dependency: typescript@5.2.2
npm WARN   node_modules/typescript
npm WARN   peer typescript@>=5.2 <5.3 from @angular-devkit/build-angular@17.0.9
npm WARN   node_modules/@angular-devkit/build-angular
npm WARN   dev @angular-devkit/build-angular@17.0.9 from the root project
[ ] | idealTree:webpack-dev-server: still placeDep ROOT webpack-dev-
```

Una vez instaladas estas dependencias podemos lanzar nuestra aplicación, para ello debemos utilizar el siguiente comando:

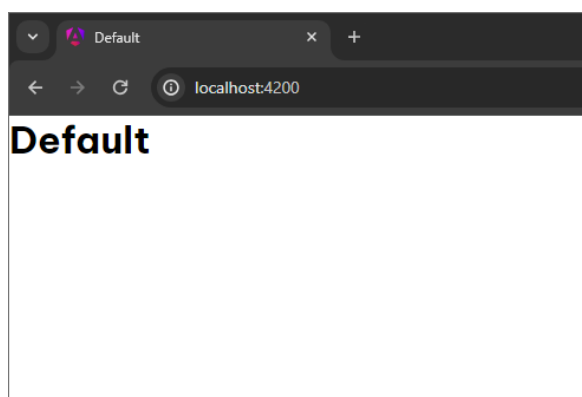
`ng serve`

```
pwsh D:\DAW\2DAW\02 PRÁCTICAS Y EJERCICIOS\07 ANGULAR\first-app
> ng serve

Initial Chunk Files | Names          | Raw Size
polyfills.js        | polyfills      | 83.60 kB
main.js             | main           | 2.15 kB
styles.css           | styles         | 457 bytes
                    | Initial Total  | 86.19 kB

Application bundle generation complete. [2.822 seconds]
Watch mode enabled. Watching for file changes...
→ Local: http://localhost:4200/
```

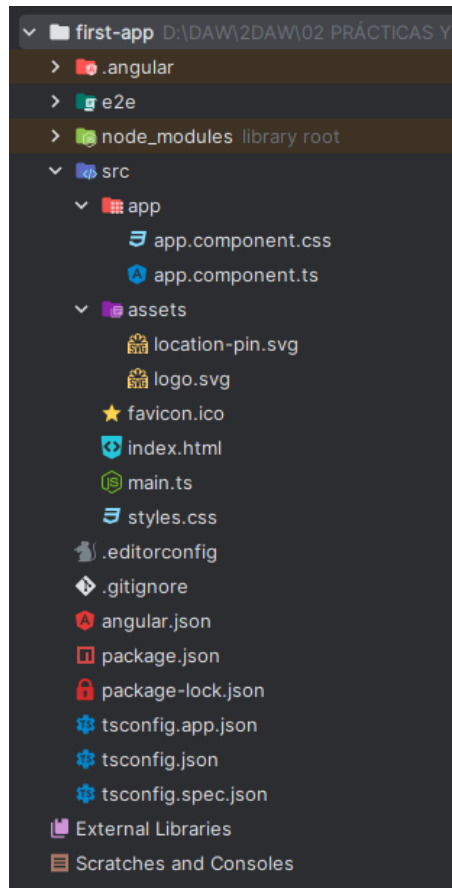
Esto lanzará un servidor en localhost en el puerto 4200:



Se trata de una especie de Live Server que nos permite ver los cambios que realizamos sobre el código de Angular en tiempo real.

B. CONTENIDO DE LA APLICACIÓN

La aplicación se compone de una serie de directorios:



En el directorio /src es donde se alojarán los ficheros que darán forma a nuestro proyecto.

- **index.html.** Es el principal documento HTML de nuestra aplicación. Como vemos su contenido se basa únicamente en una etiqueta `<app-root>`.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Default</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link rel="preconnect" href="https://fonts.googleapis.com">
  <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
  <link href="https://fonts.googleapis.com/css2?family=Be+Vietnam+Pro:ital,wght@0,400;0,700;1,400;1,700&display=swap"
        rel="stylesheet">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

- **style.css.** Es la hoja de estilos principal. En ella añadimos los estilos globales.

```

1  /* You can add global styles to this file, and
2  *
3  * margin: 0;
4  * padding: 0;
5  * }
6  *
7  * body {
8  *   font-family: 'Be Vietnam Pro', sans-serif;
9  * }
10 *
11 * :root {
12 *   --primary-color: #605DC8;
13 *   --secondary-color: #8889E6;
14 *   --accent-color: #e8e7fa;
15 *   --shadow-color: #E8E8E8;
16 * }
17 *
18 * button.primary {
19 *   padding: 10px;
20 *   border: solid 1px var(--primary-color);
21 *   background: var(--primary-color);
22 *   color: white;
23 *   border-radius: 8px;

```

- **main.ts.** Es el archivo donde se inicia la aplicación Angular.

```

import { bootstrapApplication, provideProtractorTestingSupport } from '@angular/platform-browser';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent,
  options: {providers: [provideProtractorTestingSupport()]})
  .catch(err => console.error(err));

```

- **favicon.ico.** Se trata del icono principal del sitio.

Dentro del directorio /src encontramos un segundo directorio /app que contiene los ficheros de los componentes:

```

v app
  app.component.css
  app.component.ts

```

- **app.component.ts.** Es el fichero de fuente, donde describimos el componente app-root que vemos en el index. Se trata del componente de máximo nivel de Angular y es la base de cualquier bloque de Angular.

```

1  import { Component } from '@angular/core';
2
3  1+ usages
4  @Component({
5    selector: 'app-root',
6    standalone: true,
7    imports: [],
8    template: '<h1>Default</h1>',
9    styleUrls: ['./app.component.css'],
10  })
11  export class AppComponent {
12    title: string = 'default';

```

- **app.component.css.** Es la hoja de estilos propia de este componente.

```

1  :host {
2    --content-padding: 10px;
3  }
4  header {
5    display: block;
6    height: 60px;
7    padding: var(--content-padding);
8    box-shadow: 0px 5px 25px var(--shadow-color);
9  }
10 .content {
11   padding: var(--content-padding);
12 }

```

Por último, dentro de /src tenemos un tercer directorio llamado /assets que, en este caso, contiene las imágenes usadas por la aplicación.

Otros directorios y ficheros que podemos ver en nuestro proyecto son:

- **.angular.** Contiene los ficheros necesarios para construir la aplicación de Angular.
- **.e2e.** Tiene los ficheros para testear la aplicación.
- **.node_modules.** Contiene los paquetes de node.js que utiliza la aplicación.
- **angular.json.** Describe la aplicación a las herramientas de generación de la aplicación.
- **package.json.** Lo utiliza npm para correr la aplicación finalizada.
- **tsconfig.*.** Son los ficheros que describen la configuración de la app al compilador de TS.

C. CAMBIANDO EL TÍTULO DE LA PÁGINA Y DEL CONTENIDO DE <app-root>

Ahora vamos a cambiar el título de nuestra página, para ello vamos a modificar el contenido de <title> en el fichero index.html para que sea homes.

```

<head>
  <meta charset="utf-8">
  <title>Homes</title>
  <base href="/">

```

Además vamos a modificar el fichero app.component.ts para modificar el contenido del template para que imprima "Hello World!".

```

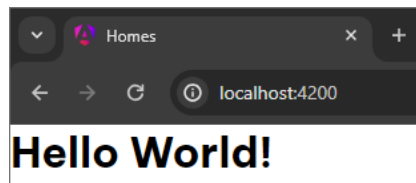
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  template: '<h1>Hello World!</h1>',
  styleUrls: ['./app.component.css'],
})

```

Y el valor del atributo title pasando de 'default' a 'homes':

```
export class AppComponent {  
  title: string = 'homes';  
}
```

Una vez hechos estos cambios si comprobamos el estado de nuestra aplicación en el navegador veremos que los cambios se han realizado con éxito:



COMPONENTES

Las aplicaciones de Angular se construyen utilizando lo que conocemos como componentes. Estos contienen el código que aporta funcionalidad al componente, el template de HTML que le da la estructura y la hoja de estilos CSS que le aporta el estilado propio.

Es importante tener en cuenta que un componente Angular puede no sólo albergar contenido HTML en su estructura sino otros componentes, es decir, un componente puede ser padre de otros componentes.

Además, estos componentes pueden visualizarse o no en función de ciertas situaciones, por ejemplo, situaciones condicionales, iteracionales, etc.

En Angular los componentes tienen una serie de metadatos que definen sus propiedades, esto se conoce como decorador.

```
@Component({  
  selector: 'app-root',  
  standalone: true,  
  imports: [],  
  template: '<h1>Hello World!</h1>',  
  styleUrls: ['./app.component.css'],  
})
```

- **selector.** Describe como Angular se referirá a este componente en los templates.
- **standalone.** Capacidad de un componente para funcionar de manera independiente sin depender de la aplicación Angular completa.
- **imports.** Define las dependencias del componente.
- **template.** Describe la estructura de HTML del componente. También podríamos tener hojas de código HTML, en cuyo caso el atributo se llamaría templateUrl.
- **styleUrls.** Conjunto de hojas de estilos que usa el componente.

Para crear un componente tenemos varias opciones, por ejemplo, podemos usar la línea de comandos o crear el componente a mano dentro del directorio /app.

En nuestro caso lo crearemos a través de los comandos, para ello en una terminal escribimos el siguiente comando:

```
ng generate component nombre_componente --inline-template --skip-tests1
```

```
powershell D:\DAW\2DAW\02 PRÁCTICAS Y EJERCICIOS\07 ANGULAR\first-app
ng generate component home --inline-template --skip-tests
CREATE src/app/home/home.component.ts (256 bytes)
CREATE src/app/home/home.component.css (0 bytes)
```

Esto habrá creado un directorio “home” dentro del directorio “app” que contiene dos ficheros, un home.component.css y un home.component.ts.

```
home
├── home.component.css
└── home.component.ts
```

A. AÑADIR EL COMPONENTE AL LAYOUT DE LA APP

Ahora vamos a añadir el componente home al app-root para que pueda ser utilizado por la aplicación.

Para ello debemos de editar el fichero app.component.ts e importar el componente:

```
import { HomeComponent } from "../home/home.component";
```

Y añadir el HomeComponent al array de imports del decorador @Component:

```
imports: [
  HomeComponent,
],
```

De esta forma, el componente ya puede ser utilizado por app-root.

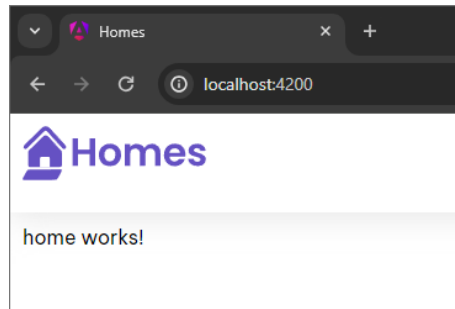
B. MODIFICAR EL CONTENIDO DEL TEMPLATE DE app-root

Vamos a modificar el contenido de nuestro componente app-root para que utilice el componente home:

```
template: `
  <main>
    <header class="brand-name">
      
    </header>
    <section class="content">
      <app-home></app-home>
    </section>
  </main>
`;
```

¹ Las opciones --inline-template y --skip-test nos sirven para crear el template HTML en el propio componente y no en un HTML externo y evitar la creación de un fichero de pruebas.

Si ahora accedemos al navegador, veremos que se han producido cambios en nuestra aplicación:



C. AÑADIENDO CAMBIOS EN EL COMPONENTE Home

Vamos a realizar una serie de cambios en el componente, por ejemplo vamos a hacer que no tenga el texto por defecto “home Works!” sino un template propio basado en un formulario y un estilado.

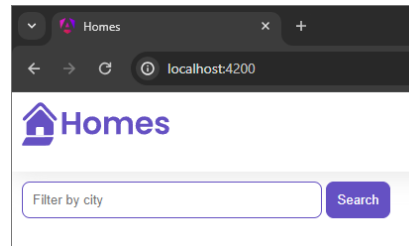
Para ello vamos a modificar el fichero home.component.ts para modificar el atributo template:

```
template: `
  <section>
    <form>
      <input type="text" placeholder="Filter by city">
      <button class="primary" type="button">Search</button>
    </form>
  </section>
`;
```

Y añadimos un estilado en la hoja de estilos home.component.css:

```
1  .results {
2    display: grid;
3    column-gap: 14px;
4    row-gap: 14px;
5    grid-template-columns: repeat(auto-fill, minmax(400px, 400px));
6    margin-top: 50px;
7    justify-content: space-around;
8  }
9
10 input[type="text"] {
11   border: solid 1px var(--primary-color);
12   padding: 10px;
13   border-radius: 8px;
14   margin-right: 4px;
15   display: inline-block;
16   width: 30%;
17 }
18
19 button {
20   padding: 10px;
21   border: solid 1px var(--primary-color);
22   background: var(--primary-color);
23   color: white;
24   border-radius: 8px;
25 }
```

Si ahora visualizamos en el navegador veremos que ha cambiado la forma en la que vemos el componente home:

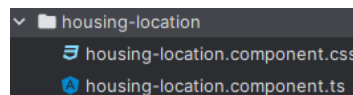


D. AÑADIENDO UN NUEVO COMPONENTE HousingLocation

Vamos a añadir un nuevo componte en este caso de nombre HousingLocation, para ello lo primero será crearlo a través de la línea de comandos:

```
pwsh D:\DAW\2DAW\02 PRÁCTICAS Y EJERCICIOS\07 ANGULAR\first-app
ng generate component housingLocation --inline-template --skip-tests
CREATE src/app/housing-location/housing-location.component.ts (303 bytes)
CREATE src/app/housing-location/housing-location.component.css (0 bytes)
```

Esto nos habrá creado un nuevo componente en nuestro directorio /app:



Este componente será un componente hijo de home y no de app-root, por tanto, los imports que hicimos en anteriores apartados en el archivo app.component.ts habrá que hacerlos en el fichero home.component.ts ya que este es el componente que debe reconocer a houseLocation.

```
import { HousingLocationComponent } from "../housing-location/housing-location.component";
```

Además, añadimos el componente en el array de imports del decorador:

```
imports: [
  CommonModule,
  HousingLocationComponent,
],
```

Hemos importado además el CommonModule que sirve para importar todas las directivas básicas de Angular como NgIf, NgForOf...

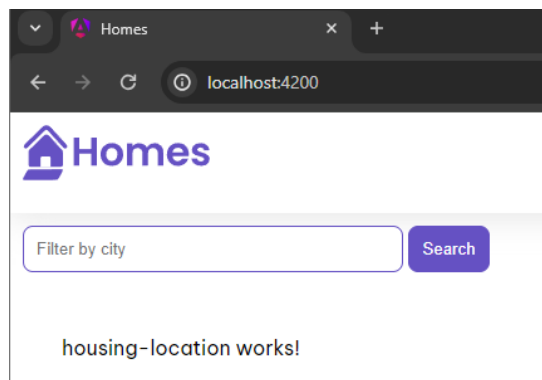
Ahora modificamos el template de home para que incluya el componente housing-location:

```
template: `
  <section>
    <form>
      <input type="text" placeholder="Filter by city">
      <button class="primary" type="button">Search</button>
    </form>
  </section>
  <section class="results">
    <app-housing-location></app-housing-location>
  </section>
`
```

Y añadimos un estilado en la hoja de estilos del componente housingLocation:

```
1 .listing {
2   background: var(--accent-color);
3   border-radius: 30px;
4   padding-bottom: 30px;
5 }
6 .listing-heading {
7   color: var(--primary-color);
8   padding: 10px 20px 0 20px;
9 }
10 .listing-photo {
11   height: 250px;
12   width: 100%;
13   object-fit: cover;
14   border-radius: 30px 30px 0 0;
15 }
16 .listing-location {
17   padding: 10px 20px 20px 20px;
18 }
19 .listing-location::before {
20   content: url("/assets/location-pin.svg") / "";
21 }
22
23 section.listing a {
24   padding-left: 20px;
25   text-decoration: none;
26   color: var(--primary-color);
27 }
28 section.listing a::after {
29   content: "\203A";
30   margin-left: 5px;
31 }
32 }
```

Si abrimos la aplicación en el navegador veremos lo siguiente:



INTERFACES

Angular usa un elemento que conocemos como interfaz y que cumplen la función de definir la estructura de un objeto en TypeScript. Es decir, se utilizan para definir la forma que debe seguir un objeto, especificando qué propiedades debe tener y qué datos deben tener esas propiedades.

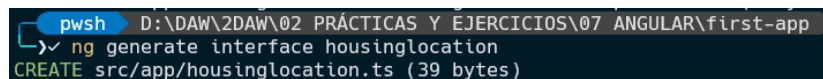
Son especialmente útiles al trabajar con un tipado fuerte en TS, ya que permiten definir y adherirse a un conjunto específico de reglas.

Es común su uso en situaciones como la definición de modelos de datos para garantizar una estructura de código más limpia y segura.

En nuestro caso vamos a crear una interfaz `housingLocation` que definirá los atributos que puede tener un objeto de este tipo.

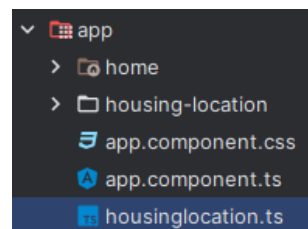
Al igual que ocurre con los componentes, las interfaces pueden ser creadas tanto por línea de comandos como de forma manual, en este caso de nuevo usaremos los comandos. La estructura del comando es muy similar a la de la creación de componentes:

```
ng generate interface nombre_de_la_interfaz
```

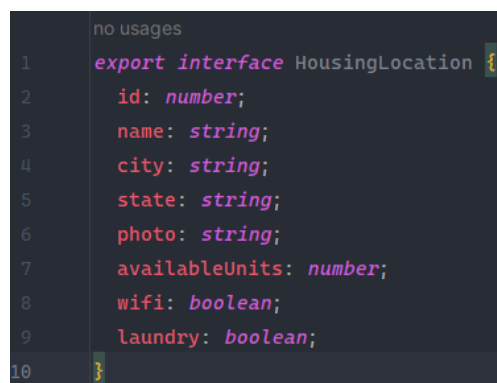


```
pwsh D:\DAW\2DAW\02 PRÁCTICAS Y EJERCICIOS\07 ANGULAR\first-app
> ng generate interface housinglocation
CREATE src/app/housinglocation.ts (39 bytes)
```

Las interfaces se generan como archivos independientes dentro de nuestro directorio `/app`:



Dentro de esta interfaz vamos a definir al hombre `housingLocation` de la siguiente forma:




```
no usages
1  export interface HousingLocation {
2      id: number;
3      name: string;
4      city: string;
5      state: string;
6      photo: string;
7      availableUnits: number;
8      wifi: boolean;
9      laundry: boolean;
10 }
```

A. PROBANDO LA INTERFAZ EN EL COMPONENTE Home

Vamos a comprobar el uso de esta interfaz `housingLocation` dentro del componente `home`.

Para ello debemos modificar el fichero `home.component.ts` y, tal y como hicimos con los componentes, debemos importar la interfaz en la cabecera de nuestro fichero `.ts`:



```
import {HousingLocation} from "../housinglocation";
```

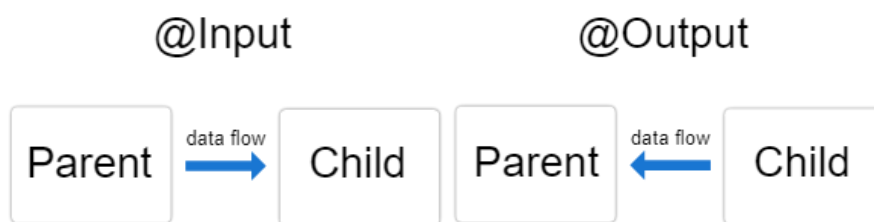
Ahora en la clase HomeComponent, instanciamos un objeto de tipo HousingLocation que será utilizado para mostrar información:

```
export class HomeComponent {
  readonly baseUrl: string = 'https://angular.io/assets/images/tutorials/faa';

  housingLocation: HousingLocation = {
    id: 9999,
    name: 'Test Home',
    city: 'Test city',
    state: 'ST',
    photo: `${this.baseUrl}/example-house.jpg`,
    availableUnits: 99,
    wifi: true,
    laundry: false,
  };
}
```

@INPUT

Angular tiene dos formas de pasar información padre-hijo en función de la dirección, si la información viaja desde un componente padre a un componente hijo, se utiliza un @Input, si por el contrario la información viaja desde un componente hijo hacia su padre, se utiliza un @Output.



En esta sección vamos a utilizar el @Input sobre el componente HousingLocation.

Para ello debemos importar Input desde @angular/core, así que tenemos dos opciones, o creamos una nueva línea import o añadimos "Input" al import de Component:

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

Y añadimos la propiedad @Input dentro de la clase HousingLocationComponent:

```
export class HousingLocationComponent {
  @Input() housingLocation!: HousingLocation;
}
```

Hemos añadido el signo de exclamación (!) porque se espera que se pase un valor. En este caso, no habrá un valor predeterminado. Es el operador de afirmación de no nulidad e informa al compilador de TS que el valor de esta propiedad no será null o undefined.

Lo que hemos hecho es preparar a este componente HousingLocation para recibir el objeto housingLocation que creamos en el padre en el punto anterior. Este objeto lo recibirá por @Input a través de una vinculación de propiedades.

VINCULACIÓN DE PROPIEDADES

En Angular, la vinculación de propiedades (*property binding*) es una forma de vincular una propiedad de un componente a un valor o expresión. Esto permite pasar datos desde la clase del componente al template.

Cuando se añade una vinculación de propiedades a una etiqueta de componente, usamos la estructura [atributo] = "valor" para informar a Angular de que el valor asignado debe ser tratado como una propiedad de la clase del componente y no como un valor de tipo cadena.

En nuestro caso vamos a añadir una vinculación de propiedades en la estructura HTML del componente home.

```
<section class="results">
  <app-housing-location [housingLocation]="housingLocation"></app-housing-location>
</section>
```

En este caso estamos pasándole al componente hijo app-housing-location el objeto housingLocation que creamos en apartados anteriores, de forma que sea consciente de este objeto.

INTERPOLACIÓN

La interpolación en Angular es una técnica que permite incorporar valores de variables o expresiones directamente en un template de HTML.

Para crear una interpolación usamos la siguiente estructura {{ valor }} en la plantilla. Los valores dentro de las llaves son evaluados y su resultado se inserta en el lugar correspondiente de la plantilla.

Por tanto, es una forma sencilla de mostrar dinámicamente datos en la interfaz de usuario en Angular.

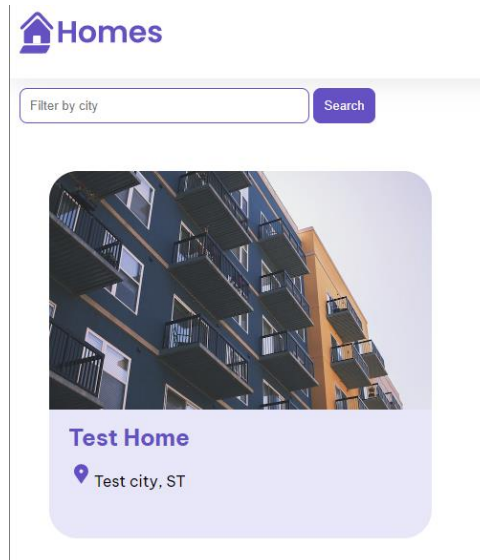
En nuestro caso vamos a modificar el template del componente HousingLocation para crear una especie de "cards" de HTML que contengan los valores de los atributos del objeto housingLocation que creamos en la clase HomeComponent:

```
template: `
  <section class="listing">
    <img class="listing-photo" [src]="housingLocation.photo" alt="Exterior photo of {{housingLocation.name}}">
    <h2 class="listing-heading">{{ housingLocation.name }}</h2>
    <p class="listing-location">{{ housingLocation.city}}, {{housingLocation.state }}</p>
  </section>
`
```

En esta actualización de código hemos creado una vinculación de propiedad en el atributo src de la etiqueta img para vincularlo al enlace generado en el atributo photo del objeto housingLocation.

Además, las propiedades name, city y state del objeto se han incluido en las etiquetas h2 y p (y en el atributo alt de la imagen) con interpolación.

El resultado final lo podemos ver en la web:



Si modificamos cualquiera de los atributos del objeto housingLocation que tenemos en la clase HomeComponent, se verán reflejados en la web:

```
housingLocation: HousingLocation = {  
  id: 9999,  
  name: 'Mi casa',  
  city: 'Coslada',  
  state: 'Comunidad de Madrid',  
  photo: `${this.baseUrl}/example-house.jpg`,  
  availableUnits: 99,  
  wifi: true,  
  laundry: false,  
};
```



DIRECTIVAS

Otro elemento importante de Angular son las directivas, estas son instrucciones en el template HTML que indican a Angular cómo manipular el DOM al momento de renderizar la aplicación.

Principalmente tenemos dos tipos de directivas:

- a) **Las directivas estructurales.** Son todas aquellas que van a alterar la estructura del DOM agregando, eliminando o reemplazando elementos. Por ejemplo, tenemos las directivas `ngIf` (condicional simple), `ngFor` (iteración) y `ngSwitch` (condicional múltiple).
- b) **Las directivas de atributos.** Son las directivas que van a alterar la apariencia o el comportamiento de un elemento sin cambiar su estructura interna. Dentro de este tipo de directivas tenemos `ngStyle` (afecta al estilo de un elemento), `ngClass` (afecta a la clase de un elemento) y `ngModel` (permite la vinculación bidireccional de datos).

Las directivas son fundamentales para la construcción de aplicaciones dinámicas y reactivas, ya que permiten la manipulación del DOM de manera declarativa según el estado y los datos de la aplicación.

A. USANDO UNA DIRECTIVA `ngFor` EN NUESTRA APLICACIÓN

Vamos a añadir una directiva `ngFor` en la aplicación que estamos creando, esta directiva nos va a servir para iterar sobre una estructura de datos (por ejemplo, un array) y realizar una determinada acción, en nuestro caso vamos a crear múltiples objetos de tipo `HousingLocation` y vamos a generar cards de cada uno de estos objetos.

Para ello debemos modificar el fichero `home.component.ts`, vamos a eliminar el objeto `housingLocation` y en su lugar vamos a crear un `housingLocationList` que será un array de objetos `housingLocation`:

```
housingLocationList: HousingLocation[] = [
  {
    id: 0,
    name: 'Acme Fresh Start Housing',
    city: 'Chicago',
    state: 'IL',
    photo: `${this.baseUrl}/bernard-hermant-CLKGGwIBTaY-unsplash.jpg`,
    availableUnits: 4,
    wifi: true,
    laundry: true
  },
  {
    id: 1,
    name: 'A113 Transitional Housing',
    city: 'Santa Monica',
    state: 'CA',
    photo: `${this.baseUrl}/brandon-griggs-wR11KBaB86U-unsplash.jpg`,
    availableUnits: 0,
    wifi: false,
    laundry: true
  },
]
```

Ahora debemos añadir la directiva al elemento `<app-housing-location>` del template:

```
<app-housing-location
  *ngFor="let housingLocation of housingLocationList"
  [housingLocation]="housingLocation">
</app-housing-location>
```

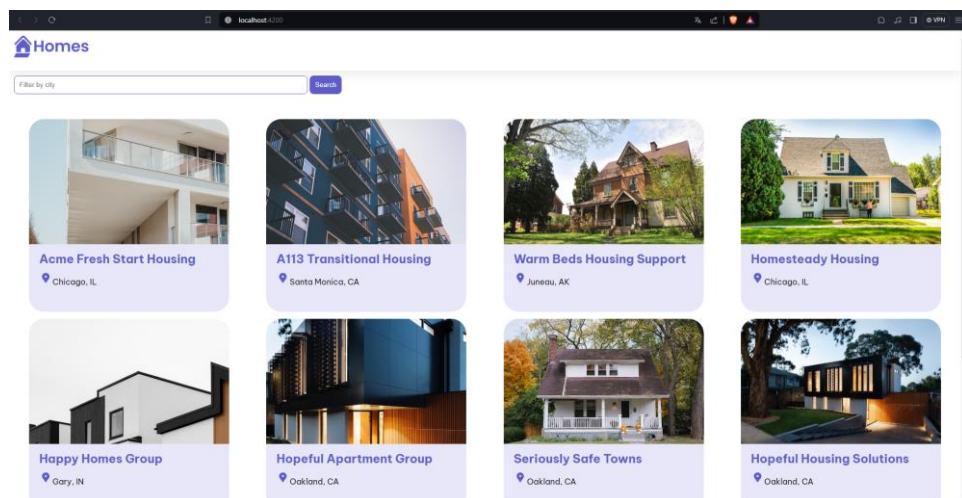
Lo que quiere decir este código es que mientras existan elementos `housingLocation` en el array `housingLocationList` se generará un elemento `app-hosing-location` que tendrá una vinculación de propiedad con el objeto `housingLocation` correspondiente.

A partir de la versión 17 de Angular, existe una forma alternativa de crear la sintaxis para una directiva `ngFor`:

```
@for (housingLocation of housingLocationList; track housingLocation.id) {
  <app-housing-location [housingLocation]="housingLocation"></app-housing-location>
}
```

En este caso se necesita un parámetro *track* para hacer un seguimiento de los elementos que se están eligiendo del array.

Si ahora accedemos a la web veremos el siguiente resultado:



SERVICIOS

En Angular, un servicio es una clase con un propósito específico que ofrece funcionalidades y características reutilizables a través de la inyección de dependencias (*dependency injection*). Permiten compartir lógica y datos entre diferentes componentes de una aplicación Angular.

Las características principales de los servicios son las siguientes:

1. **Inyección de dependencias.** Los servicios se “inyectan” en los componentes, directivas u otros servicios que los necesiten. Angular se encarga de proporcionar instancias únicas de los servicios en toda la aplicación.
2. **Reutilización de Lógica.** Los servicios son ideales para contener lógica de negocio, operaciones de red, manipulación de datos, etc., que pueden ser compartidas entre varios componentes.
3. **Separación de Responsabilidades.** Utilizar servicios ayuda a mantener una arquitectura limpia y modular, donde cada componente se enfoca en su propia responsabilidad y delega tareas específicas a los servicios.

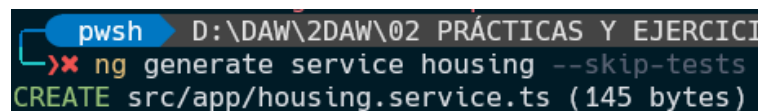
Por tanto, los servicios van a proporcionar una forma de separar los datos de la aplicación Angular y las funciones que pueden ser usadas en varios componentes de la aplicación. Para que esto sea posible, el servicio debe ser creado como inyectable (*injectable*) en su decorador. El componente depende de esos servicios y no puede funcionar sin ellos.

A. CREACIÓN DE UN SERVICIO

Como hemos visto en anteriores casos los elementos de Angular se pueden crear tanto de forma manual como por consola, de nuevo utilizaremos esta última opción ya que es la que nos proporciona el esqueleto base del elemento que creamos.

En este caso el comando será:

```
ng generate service nombre_servicio --skip-tests
```



```
pwsh D:\DAW\2DAW\02 PRÁCTICAS Y EJERCICIOS> ng generate service housing --skip-tests
CREATE src/app/housing.service.ts (145 bytes)
```

Esto nos habrá creado un archivo `housing.service.ts` que se encuentra dentro del directorio `/app` y que al igual que ocurre con las interfaces no requiere de un directorio propio.

El aspecto que tiene el fichero es el siguiente:



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

no usages
@Injectable({
  providedIn: 'root'
})
export class HousingService {

  no usages
  constructor() { }
}
```

B. AÑADIENDO DATOS ESTÁTICOS AL SERVICIO

Una vez creado debemos añadir funcionalidad a nuestro servicio, en este caso vamos a añadir dos funciones get que utilicen los objetos housingLocation creados en apartados anteriores.

Para ello debemos copiar el array del fichero home.component.ts a nuestro servicio.

```
export class HousingService {
  readonly baseUrl = 'https://angular.io/assets/images/tutorials/faa';

  housingLocationList: HousingLocation[] = [
    {
      id: 0,
      name: 'Acme Fresh Start Housing',
      city: 'Chicago',
      state: 'IL',
      photo: `${this.baseUrl}/bernard-hermant-CLKGGwIBTaY-unsplash.jpg`,
      availableUnits: 4,
      wifi: true,
      laundry: true
    },
    {
      id: 1,
      name: 'A113 Transitional Housing',
      city: 'Santa Monica',
      state: 'CA',
      photo: `${this.baseUrl}/brandon-griggs-wR1lKBaB86U-unsplash.jpg`,
      availableUnits: 0,
      wifi: false,
      laundry: true
    }
  ],
}
```

Y creamos las siguientes funciones:

```
no usages
getAllHousingLocations(): HousingLocation[] {
  return this.housingLocationList;
}

no usages
getHousingLocationById(id: number): HousingLocation | undefined {
  return this.housingLocationList.find(housingLocation : HousingLocation => housingLocation.id === id);
}
```

Por último, si no se ha realizado de forma automática, debemos importar la interfaz HousingLocation:

```
import { HousingLocation } from './housinglocation';
```

C. INYECTAR EL SERVICIO DENTRO DEL COMPONENTE Home

Ahora que hemos creado nuestro servicio, vamos a añadirlo a un componente, en este caso el componente Home.

Lo primero que debemos añadir es importar inject desde @angular/core y nuestro servicio:

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

import { HousingService } from "../housing.service";
```

A continuación, borramos el contenido que hemos copiado en el servicio ya que no necesitamos que esté aquí y en lugar creamos el siguiente código:

```
export class HomeComponent {
  housingLocationList: HousingLocation[] = [];
  housingService: HousingService = inject(HousingService);

  no usages new *
  constructor() {
    this.housingLocationList = this.housingService.getAllHousingLocations();
  }
}
```

Con esto inyectamos el servicio HousingService e inicializamos los datos de la aplicación. El constructor será la primera función que se ejecutará cuando el componente se cree.

El código en el interior del constructor asignará a housingLocationList el valor que devuelva la función getAllHousingLocations que hemos creado en nuestro servicio.

Con esto habremos inyectado el servicio y la aplicación mantendrá la funcionalidad creada anteriormente, pero los datos en lugar de estar encerrados en un componente (HomeComponent) están en un servicio por lo que podrán ser accedidos por múltiples componentes en caso de ser necesario.

ENRUTADO

El enrutado en Angular se refiere a la capacidad de la aplicación para navegar entre diferentes componentes y vistas de manera estructurada. El enrutado es gestionado por el módulo RouterModule, que proporciona una forma de definir las rutas y las reglas de navegación en una aplicación Angular.

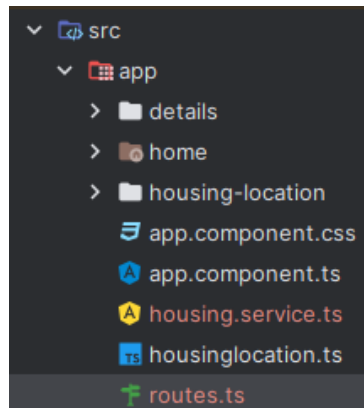
A. CREAR UN COMPONENTE DE DETALLES POR DEFECTO

Vamos a crear un nuevo componente, para ello usamos el comando de ng generate correspondiente:

```
powershell D:\DAW\2DAW\02 PRÁCTICAS Y EJERCICIOS\07 ANGULAR\first-
ng generate component details --inline-template --skip-tests
CREATE src/app/details/details.component.ts (268 bytes)
CREATE src/app/details/details.component.css (0 bytes)
```

B. AÑADIENDO ENRUTADO A NUESTRA APLICACIÓN

En el directorio `/src/app` vamos a crear un fichero `routes.ts` que será un fichero que definirá las rutas de nuestra aplicación:



Para que el enrutado de nuestra aplicación funcione, debemos modificar dos archivos, el primero el archivo `main.ts` debemos importar la variable `routeConfig` del fichero de rutas que hemos creado en el paso anterior (esta variable se creará en el paso siguiente) y la función `provideRouter` de `@angular/router`.

```
import { provideRouter } from '@angular/router';  
import routeConfig from './app/routes';
```

Además, debemos modificar la llamada a `bootstrapApplication` para incluir la configuración de enrutado:

```
bootstrapApplication(AppComponent,  
  options: {  
    providers: [  
      provideProtractorTestingSupport(),  
      provideRouter(routeConfig)  
    ]  
  })  
).catch(err => console.error(err));
```

También debemos modificar el fichero `app.component.ts` añadiendo un `import` del `RouterModule` e incluyendo este módulo dentro del array `imports`.

```
import { RouterModule } from '@angular/router';  
  
imports: [  
  HomeComponent,  
  RouterModule,  
],
```

Modificamos por último la propiedad `template` reemplazando el componente `app-home` por `router-outlet` y añadimos un link que devuelva a la página principal.

```
template: `
  <main>
    <a [routerLink]="['/']">
      <header class="brand-name">
        
      </header>
    </a>
    <section class="content">
      <router-outlet></router-outlet>
    </section>
  </main>
`
```

C. CONFIGURANDO EL FICHERO DE RUTAS

En el fichero `routes.ts` debemos realizar los siguientes imports:

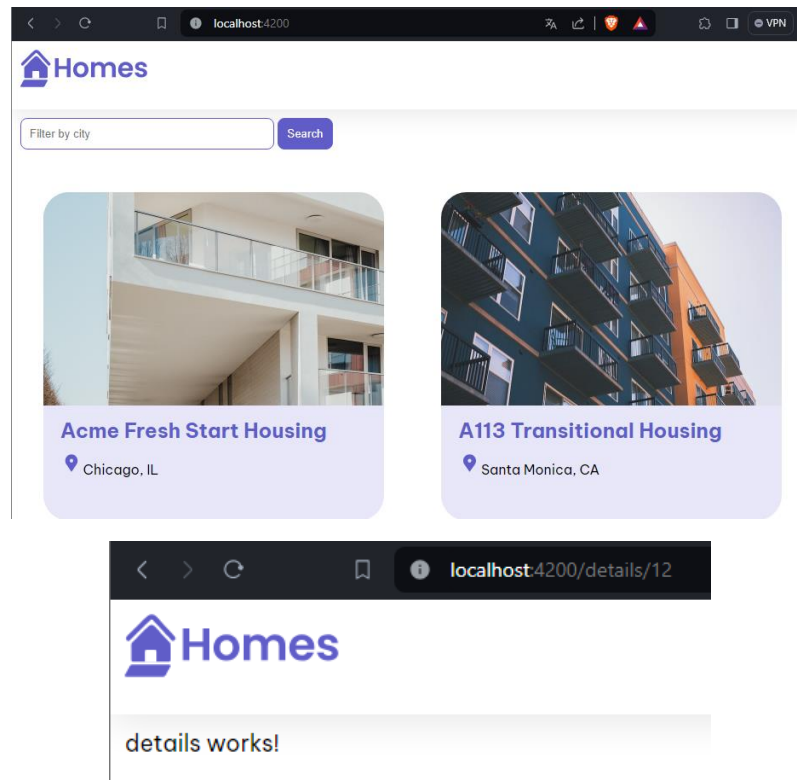
```
import { Routes } from '@angular/router';
import { HomeComponent } from '../home/home.component';
import { DetailsComponent } from '../details/details.component';
```

Y añadimos una variable `routeConfig` que se trata de un array de objetos que contienen un `path`, el componente a utilizar y un título, por último, exportamos esta variable para que pueda ser usada por otras clases.

```
const routeConfig: Routes = [
  {
    path: '',
    component: HomeComponent,
    title: 'Home page'
  },
  {
    path: 'details/:id',
    component: DetailsComponent,
    title: 'Home details'
  }
];

1+ usages
export default routeConfig;
```

Con esto quedará configurado el enrutado de nuestra web:



D. EDITANDO LAS CARDS PARA QUE AÑADAN EL LINK A DETAILS

Como hemos visto en el punto anterior hemos creado una ruta `details/:id` donde `:id` es un valor dinámico que cambiará en función de cómo se solicite la ruta por el código.

Lo primero que debemos de hacer es modificar el fichero `housing-location.component.ts` para que en su template tenga un elemento `<a>` que lleve a esta vista de detalles. Además, añadiremos la directiva `routerLink`.

Esta directiva permite al enrutador de Angular crear enlaces dinámicos en la aplicación. El valor asignado a `routerLink` es un array con dos entradas: la parte estática de la ruta y los datos dinámicos.

```
template: `
  <section class="listing">
    <img class="listing-photo" [src]="housingLocation.photo" alt="Exterior photo of {{housingLocation.name}}">
    <h2 class="listing-heading">{{ housingLocation.name }}</h2>
    <p class="listing-location">{{ housingLocation.city}}, {{housingLocation.state }}</p>
    <a [routerLink]="['/details', housingLocation.id]">Learn More</a>
  </section>
`
```

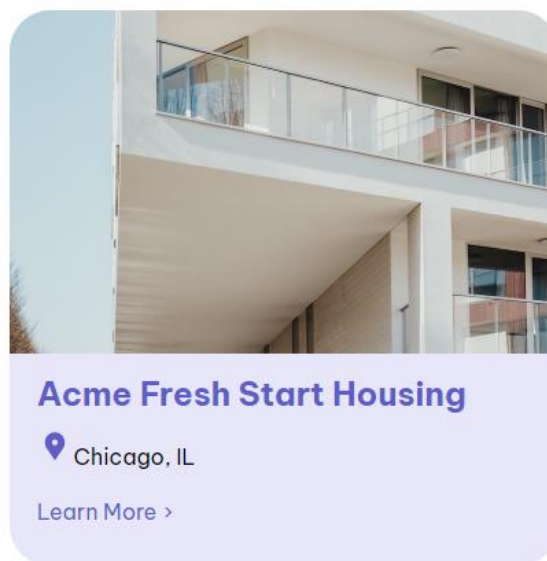
Para que esto funcione es necesario importar `RouterLink` y `RouterOutlet` de `@angular/router`.

```
import { RouterLink, RouterOutlet } from "@angular/router";
```


También tendremos que añadirlo al array imports del decorador @Component para que puedan ser utilizados:

```
imports: [  
  CommonModule,  
  RouterLink,  
  RouterOutlet,  
],
```

Ahora si accedemos a la web veremos lo siguiente:



Si hacemos clic en “Learn More” nos llevará al details/:id correspondiente.

E. EDITANDO EL COMPONENTE DETAILS

Ahora vamos a editar el componente details para que muestre una primera información, en concreto el id de cada una de las casas (atributo id de housingLocation).

Lo primero será importar dentro del fichero details.component.ts las funciones, clases y servicios que utilizaremos:

```
import { Component, inject } from '@angular/core';  
import { CommonModule } from '@angular/common';  
import { ActivatedRoute } from '@angular/router';  
import { HousingService } from '../housing.service';  
import { HousingLocation } from '../housinglocation';
```

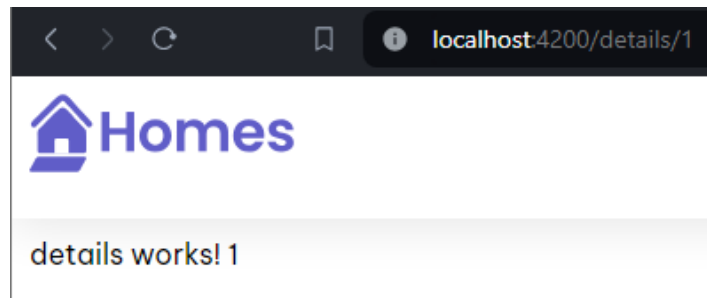
Luego daremos una forma al template, en este caso añadiremos el id:

```
template: `<p>details works! {{ housingLocationId }}</p>`,
```

Este id lo obtendremos de la siguiente manera:

```
export class DetailsComponent {
  route: ActivatedRoute = inject(ActivatedRoute);
  housingLocationId: number = -1;
  no usages
  constructor() {
    this.housingLocationId = Number(this.route.snapshot.params['id']);
  }
}
```

Ahora si hacemos clic en “Learn More” de alguno de los edificios veremos lo siguiente:



F. ÚLTIMAS CONFIGURACIONES DEL COMPONENTE DETAILS

Para finalizar con la configuración del componente details vamos a actualizar su template para que muestre una información específica de cada uno de los elementos housingLocation.

```
template: `
<article>
  <img class="listing-photo" [src]="housingLocation?.photo"
    alt="Exterior photo of {{housingLocation?.name}}"/>
  <section class="listing-description">
    <h2 class="listing-heading">{{housingLocation?.name}}</h2>
    <p class="listing-location">{{housingLocation?.city}}, {{housingLocation?.state}}</p>
  </section>
  <section class="listing-features">
    <h2 class="section-heading">About this housing location</h2>
    <ul>
      <li>Units available: {{housingLocation?.availableUnits}}</li>
      <li>Does this location have wifi: {{housingLocation?.wifi}}</li>
      <li>Does this location have laundry: {{housingLocation?.laundry}}</li>
    </ul>
  </section>
</article>
`
```

Hemos colocado el símbolo ? que es el operador opcional, esto asegura que si el valor housingLocation que se pasa es null o undefined no se produzca un crasheo de la aplicación.

También debemos modificar la clase de DetailsComponent con el código siguiente:

```
export class DetailsComponent {  
  
  route: ActivatedRoute = inject(ActivatedRoute);  
  housingService: HousingService = inject(HousingService);  
  housingLocation: HousingLocation | undefined;  
  
  no usages  
  constructor() {  
    const housingLocationId: number = Number(this.route.snapshot.params['id']);  
    this.housingLocation = this.housingService.getHousingLocationById(housingLocationId);  
  }  
}
```

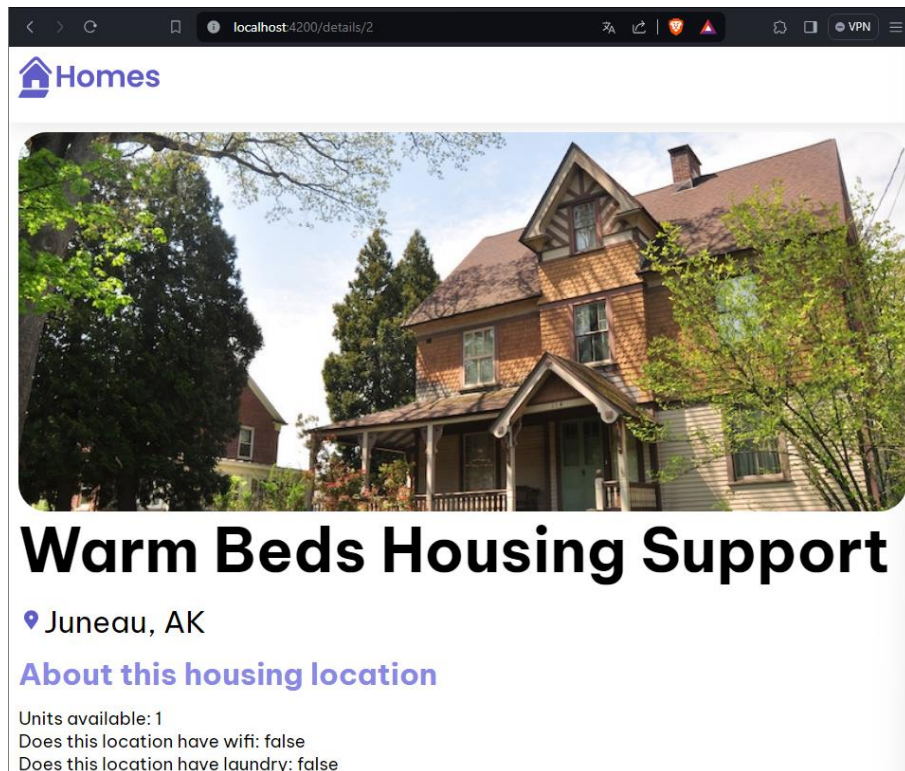
Ahora el componente tiene el código necesario para permitir la visualización de la información en base al elemento houseLocation seleccionado.

El constructor ahora incluye una llamada al servicio HousingService para pasar el parámetro de ruta como argumento de la función getHousingLocationById del servicio.

Por último, vamos a modificar el fichero CSS para aportar cierto estilo a nuestra vista details:

```
.listing-photo {  
  height: 600px;  
  width: 50%;  
  object-fit: cover;  
  border-radius: 30px;  
  float: right;  
}  
  
.listing-heading {  
  font-size: 48pt;  
  font-weight: bold;  
  margin-bottom: 15px;  
}  
  
.listing-location::before {  
  content: url('/assets/location-pin.svg') / '';  
}  
  
.listing-location {  
  font-size: 24pt;  
  margin-bottom: 15px;  
}
```

Si accedemos a la web el aspecto será el siguiente:



FORMULARIOS

En este apartado veremos como crear un formulario para enviar datos.

En primer lugar, debemos añadir un método para enviar datos, esto se consigue modificando el servicio housing. Dentro de la clase HousingService debemos añadir el siguiente código:

```
submitApplication(firstName: string, lastName: string, email: string): void {  
  console.log('Homes application received: firstName: ${firstName}, lastName: ${lastName}, email: ${email}.');  
}
```

Ahora debemos añadir el formulario a la vista de detalles. Para ello necesitamos importar las clases FormControl, FormGroup y ReactiveFormsModule.

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

Además, la última de estas clases debe ser añadida en el array de imports del decorador.

```
imports: [  
  CommonModule,  
  ReactiveFormsModule  
],
```

En la clase DetailsComponent, antes del constructor, añadiremos el siguiente código para crear el objeto formulario.

```
applyForm : FormGroup<{firstName: FormCont... = new FormGroup( controls: {  
  firstName: new FormControl( value: '' ),  
  lastName: new FormControl( value: '' ),  
  email: new FormControl( value: '' )  
});
```

En Angular, FormGroup y FormControl son tipos que permiten crear formularios. El tipo FormControl puede proporcionar un valor predeterminado y dar forma a los datos del formulario.

Ahora, después del constructor, añadimos este código:

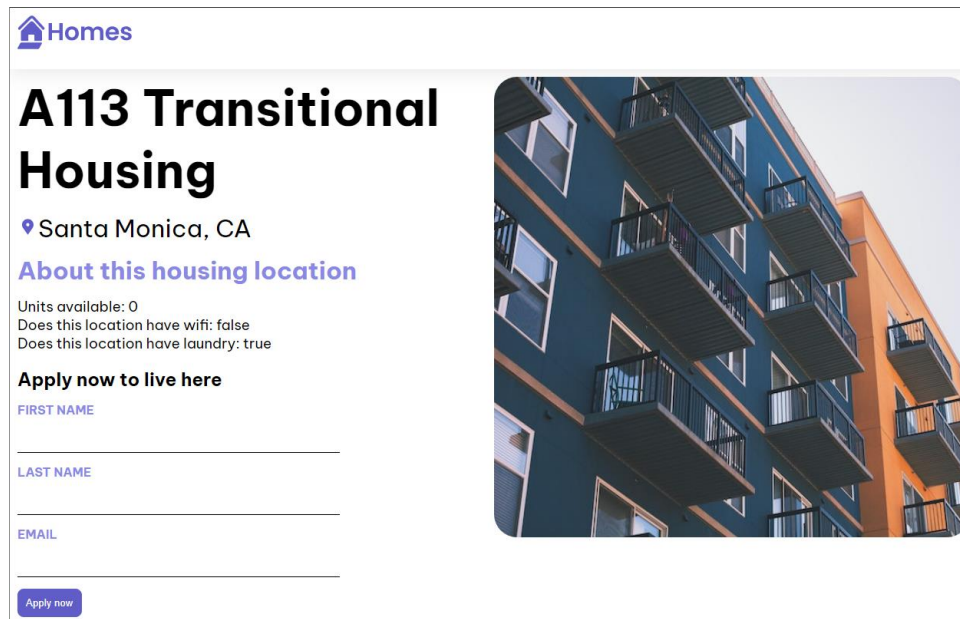
```
no usages  
submitApplication() : void {  
  this.housingService.submitApplication(  
    firstName: this.applyForm.value.firstName ?? '',  
    lastName: this.applyForm.value.lastName ?? '',  
    email: this.applyForm.value.email ?? ''  
  );  
}
```

Por último, modificamos el HTML del componente:

```
template: `<article>  
  <img class="listing-photo" [src]="housingLocation?.photo"  
    alt="Exterior photo of {{housingLocation?.name}}"/>  
  <section class="listing-description">  
    <h2 class="listing-heading">{{housingLocation?.name}}</h2>  
    <p class="listing-location">{{housingLocation?.city}}, {{housingLocation?.state}}</p>  
  </section>  
  <section class="listing-features">  
    <h2 class="section-heading">About this housing location</h2>  
    <ul>  
      <li>Units available: {{housingLocation?.availableUnits}}</li>  
      <li>Does this location have wifi: {{housingLocation?.wifi}}</li>  
      <li>Does this location have laundry: {{housingLocation?.laundry}}</li>  
    </ul>  
  </section>  
  <section class="listing-apply">  
    <h2 class="section-heading">Apply now to live here</h2>  
    <form [formGroup]="applyForm" (submit)="submitApplication()">  
      <label for="first-name">First Name</label>  
      <input id="first-name" type="text" formControlName="firstName">  
  
      <label for="last-name">Last Name</label>  
      <input id="last-name" type="text" formControlName="lastName">  
  
      <label for="email">Email</label>  
      <input id="email" type="email" formControlName="email">  
      <button type="submit" class="primary">Apply now</button>  
    </form>  
  </section>  
</article>
```

El template ahora incluye un manejador de eventos “(submit)=submitApplication()”. Angular utiliza la sintaxis de paréntesis alrededor del nombre del evento para definir eventos en el código de la plantilla. El código del lado derecho del signo igual es el código que se ejecutará cuando se desencadene este evento. Puedes vincularte a eventos del navegador y eventos personalizados.

Si ahora visualizamos la aplicación veremos lo siguiente dentro de details:



Homes

A113 Transitional Housing

📍 Santa Monica, CA

About this housing location

Units available: 0
Does this location have wifi: false
Does this location have laundry: true

Apply now to live here

FIRST NAME

LAST NAME

EMAIL

Apply now

FUNCIÓN DE BÚSQUEDA

En este apartado vamos a añadir una función de búsqueda a la aplicación.

Lo primero que haremos será añadir una nueva propiedad a la clase HomeComponent.

```
filteredLocationList: HousingLocation[] = [];
```

Este listado deberá contener un set de housingLocations por defecto cuando cargue la página, por lo que debemos modificar el constructor:

```
constructor() {  
  this.housingLocationList = this.housingService.getAllHousingLocations();  
  this.filteredLocationList = this.housingLocationList;  
}
```

Ahora debemos modificar el input del template para añadir el elemento #filter.

```
<input type="text" placeholder="Filter by city" #filter>
```

Este ejemplo utiliza una variable de referencia de plantilla para acceder al elemento de entrada y obtener su valor.

Ahora actualizamos el botón para añadir un evento de clic:

```
<button class="primary" type="button" (click)="filterResults(filter.value)">Search</button>
```

Al vincular al evento de clic en el elemento del botón, puedes llamar a la función `filterResults`. El argumento de la función es la propiedad `value` de la variable de plantilla `filter`. Específicamente, la propiedad `.value` del elemento HTML de entrada.

La última actualización del template de HTML será sobre la directiva `ngFor`, debemos hacer que el `for` itere sobre la lista de localizaciones filtradas que acabamos de crear:

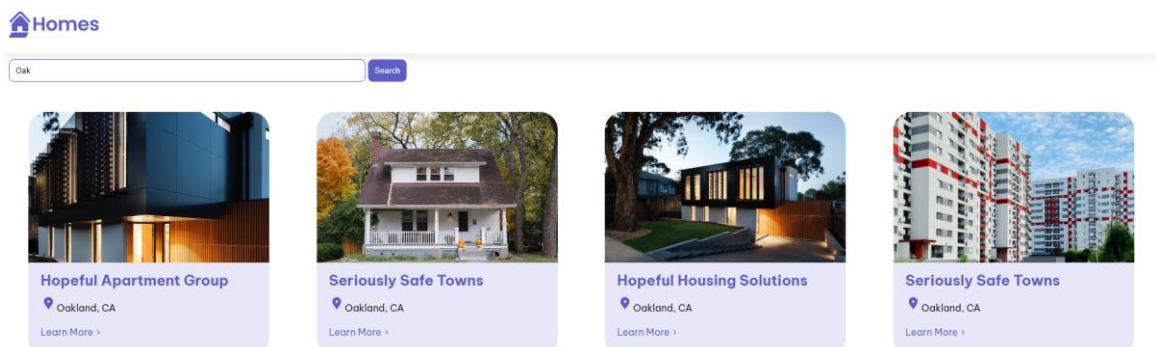
```
@for (housingLocation of filteredLocationList; track housingLocation.id) {  
  <app-housing-location [housingLocation]="housingLocation"></app-housing-location>  
}
```

Por último, debemos añadir funcionalidad al evento clic que hemos creado, para ello modificamos el la clase `HomeComponent` añadiendo las siguientes funciones:

```
filterResults(text: string) : void {  
  if (!text) {  
    this.filteredLocationList = this.housingLocationList;  
    return;  
  }  
  
  this.filteredLocationList = this.housingLocationList.filter(  
    housingLocation : HousingLocation => housingLocation?.city.toLowerCase().includes(text.toLowerCase())  
  );  
}
```

Esta función utiliza la función `filter` de JavaScript para comparar el valor del parámetro `text` con la propiedad `housingLocation.city`. Puedes actualizar esta función para comparar contra cualquier propiedad o múltiples propiedades como un ejercicio divertido.

Si ahora probamos la funcionalidad veremos lo siguiente:



COMUNICACIÓN HTTP

Hasta este punto, nuestra aplicación ha leído datos de un array estático en un servicio de Angular. El siguiente paso es utilizar un servidor JSON con el que tu aplicación se comunicará a través de HTTP. La solicitud HTTP simulará la experiencia de trabajar con datos de un servidor.

Para este paso necesitamos instalar un nuevo módulo, el módulo json-server:

```
pwsh D:\DAW\2DAW\02 PRÁCTICAS Y EJEMPLOS
> npm install -g json-server

added 54 packages in 10s
14 packages are looking for funding
run `npm fund` for details
```

En el directorio raíz de nuestro proyecto, vamos a crear un fichero db.json con el siguiente contenido:

```
db.json
{
  "locations": [
    {
      "id": 0,
      "name": "Acme Fresh Start Housing",
      "city": "Chicago",
      "state": "IL",
      "photo": "https://angular.io/assets/images/tutorials/faa/bernard-hermant-CLKGGwIBTaY-unsplash.jpg",
      "availableUnits": 4,
      "wifi": true,
      "laundry": true
    },
    {
      "id": 1,
      "name": "A113 Transitional Housing",
      "city": "Santa Monica",
      "state": "CA",
      "photo": "https://angular.io/assets/images/tutorials/faa/brandon-griggs-wR1lKBaB86U-unsplash.jpg",
      "availableUnits": 0,
      "wifi": false,
      "laundry": true
    }
  ]
}
```

Ahora vamos a testear la configuración lanzando el siguiente comando:

```
pwsh D:\DAW\2DAW\02 PRÁCTICAS Y EJEMPLOS
> json-server --watch db.json
--watch/-w can be omitted, JSON Server 1
JSON Server started on PORT :3000
Press CTRL-C to stop
Watching db.json...

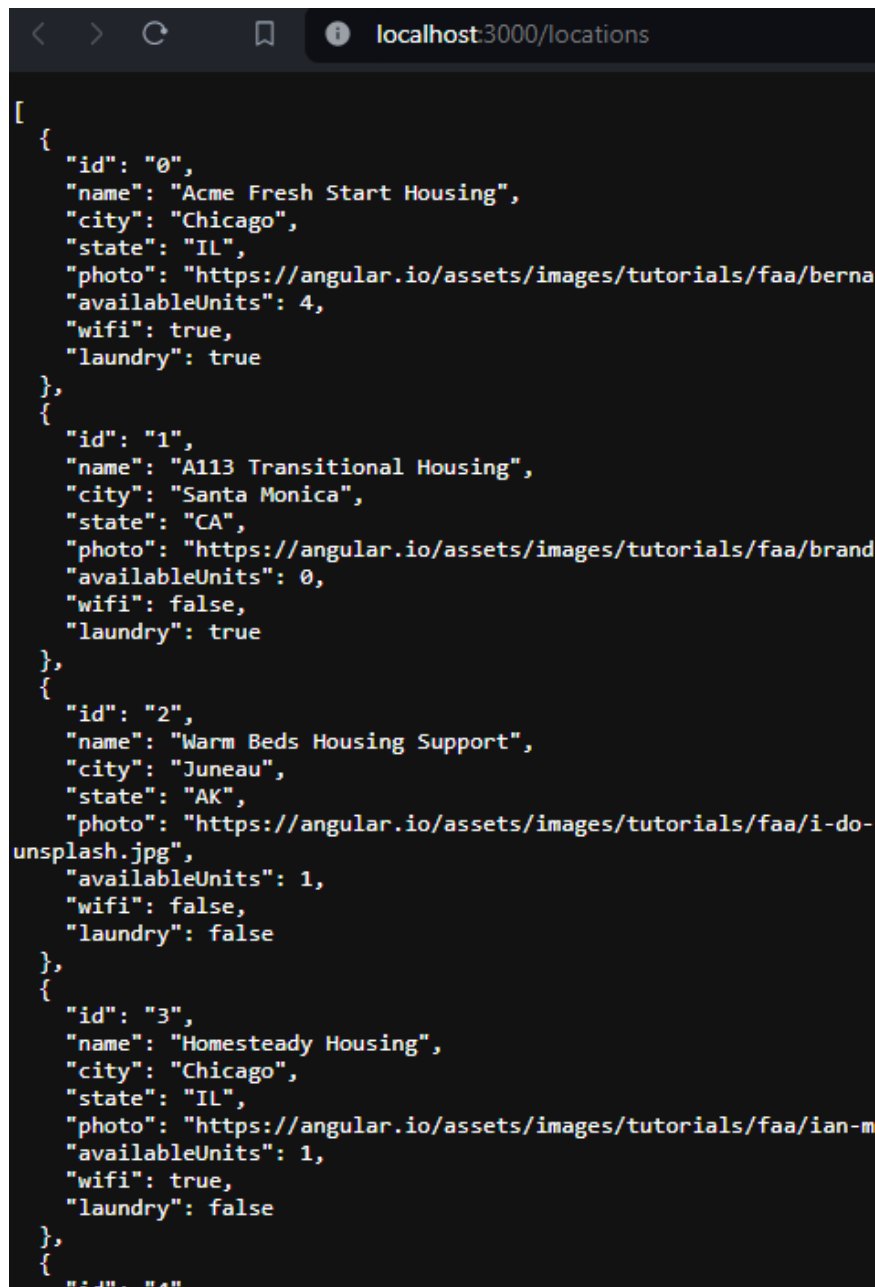
( ^ _ ^ )

Index:
http://localhost:3000/

Static files:
Serving ./public directory if it exists

Endpoints:
http://localhost:3000/locations
```


Y accedemos a la URL <http://localhost:3000/locations>. Si todo ha funcionado correctamente veremos algo similar a lo siguiente:



```
[
  {
    "id": "0",
    "name": "Acme Fresh Start Housing",
    "city": "Chicago",
    "state": "IL",
    "photo": "https://angular.io/assets/images/tutorials/faa/berna",
    "availableUnits": 4,
    "wifi": true,
    "laundry": true
  },
  {
    "id": "1",
    "name": "A113 Transitional Housing",
    "city": "Santa Monica",
    "state": "CA",
    "photo": "https://angular.io/assets/images/tutorials/faa/brand",
    "availableUnits": 0,
    "wifi": false,
    "laundry": true
  },
  {
    "id": "2",
    "name": "Warm Beds Housing Support",
    "city": "Juneau",
    "state": "AK",
    "photo": "https://angular.io/assets/images/tutorials/faa/i-do-unsplash.jpg",
    "availableUnits": 1,
    "wifi": false,
    "laundry": false
  },
  {
    "id": "3",
    "name": "Homesteady Housing",
    "city": "Chicago",
    "state": "IL",
    "photo": "https://angular.io/assets/images/tutorials/faa/ian-m",
    "availableUnits": 1,
    "wifi": true,
    "laundry": false
  },
  {
    "id": "4"
```

Ahora que la fuente de datos está configurada, lo siguiente es actualizar la aplicación para que utilice estos datos.

Para ello vamos a actualizar el servicio housing.

- Eliminaremos el array `housingLocationList`.
- Añadiremos una propiedad `url` con el valor `'http://localhost:3000/locations'`.
- Actualizaremos la función `getAllHousingLocation` para convertirla en una función asíncrona.
- Haremos lo mismo con la función de `getHousingLocationById`.

Una vez realizados estos cambios, el aspecto de nuestro fichero debería ser el siguiente:

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

1+ usages
@Injectable({
  providedIn: 'root'
})
export class HousingService {

  url : string = 'http://localhost:3000/locations';

  1+ usages
  async getAllHousingLocations(): Promise<HousingLocation[]> {
    const data : Response = await fetch(this.url);
    return await data.json() ?? [];
  }

  1+ usages
  async getHousingLocationById(id: number): Promise<HousingLocation | undefined> {
    const data : Response = await fetch(input: `${this.url}/${id}`);
    return await data.json() ?? {};
  }

  1+ usages
  submitApplication(firstName: string, lastName: string, email: string): void {
    console.log(firstName, lastName, email);
  }
}
```

Ahora debemos modificar los constructores de los componentes para que usen la nueva versión de getAllHousingLocation.

a) En el componente home:

```
constructor() {
  this.housingService.getAllHousingLocations().then((housingLocationList: HousingLocation[]) : Promise<HousingLocation[]> => {
    this.housingLocationList = housingLocationList;
    this.filteredLocationList = housingLocationList;
  });
}
```

b) En el componente details:

```
constructor() {
  const housingLocationId : number = parseInt(this.route.snapshot.params['id'], radix: 10);
  this.housingService.getHousingLocationById(housingLocationId).then(housingLocation : HousingLocation | undefined => {
    this.housingLocation = housingLocation;
  });
}
```

Con esto habrá terminado la configuración y si ahora hacemos solicitudes a nuestra web, los datos vendrán desde el JSON y no desde el array.

