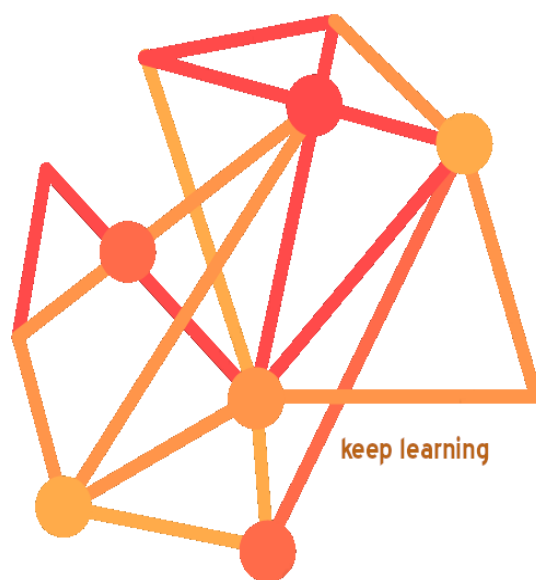


# Angular



Juan Carlos Pérez Rodríguez

## Sumario

Introducción.....	3
Instalación.....	3
Primeros pasos.....	4
Componentes.....	6
Bucles en templates: *ngFor.....	7
Pasar información a un componente hijo.....	8
Eventos sencillos: (click).....	9
Formularios.....	9
Atributos y Condiciones en templates.....	12
[(ngModel)].....	13
Comunicación entre componentes: Servicios.....	15
Router.....	17
Instalación, uso del router.....	18
Parámetros en rutas.....	21
Api.....	23
Preparación de la app para httpclient.....	23
Uso de Post.....	25

# Introducción

Según wikipedia:

“Angular es un **framework** para aplicaciones web desarrollado en **TypeScript**, de código abierto, mantenido por Google, que se utiliza para crear y mantener **aplicaciones web de una sola página** (SPA [https://es.wikipedia.org/wiki/Single-page\\_application](https://es.wikipedia.org/wiki/Single-page_application) ). Su objetivo es aumentar las aplicaciones basadas en navegador con capacidad de **Modelo Vista Controlador (MVC)**, en un esfuerzo para hacer que el desarrollo y las pruebas sean más fáciles. “

## Instalación

Debemos tener instalado npm ( es conveniente tener la última lts de node: **nvm install --lts**)

Con npm instalado, ejecutaremos:

```
npm install @angular/cli
```

Lo anterior nos deja instalado la terminal de comandos de angular. Así, un nuevo proyecto se instalará mediante:

```
ng new primer-angular-app
```

Una vez creado, podemos ejecutar la app mediante:

```
ng serve -o
```

**Nota:** El parámetro: -o es innecesario. Es únicamente para que nos abra el navegador una vez arrancada la app

Extensiones vscode recomendables:

**Tabnine:** es una ia que ayuda a la generación de código

**Prettier**

**angular snippets**

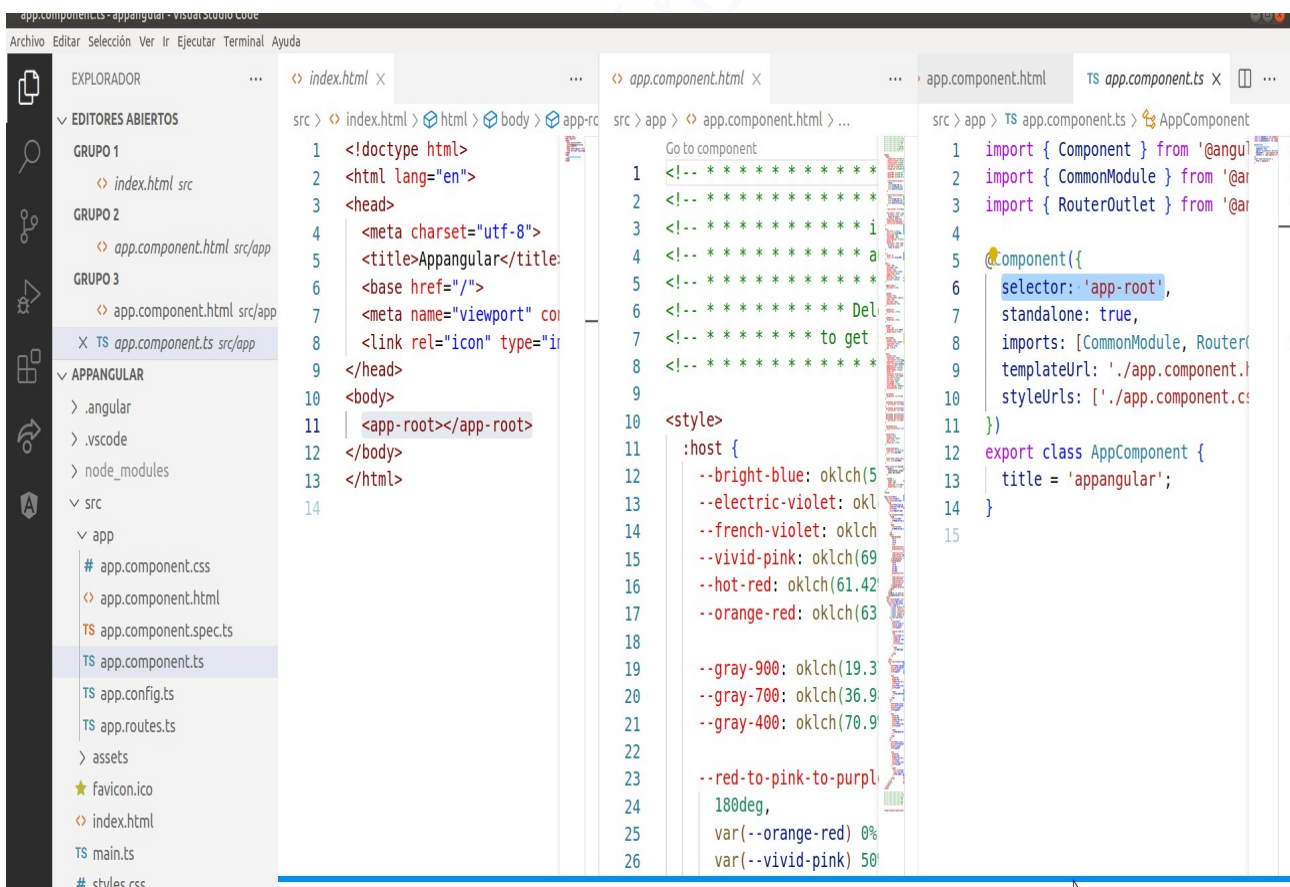
**eslint**

**json to ts**

**material icon theme**

## Primeros pasos

Para ver la estructura de un proyecto vamos a comenzar por crear uno nuevo:



En la anterior imagen se muestra: index.html Observar en ese fichero ( es el comienzo de la aplicación ) la etiqueta: `<app-root></app-root>` que indica la aplicación angular que será cargada

Esa no es una etiqueta html . La declaración de esa etiqueta la podemos observar en otro de los ficheros que se muestran: app.component.ts En ese fichero, vemos que se está declarando esa etiqueta como un identificador de componente Angular. Dice:

selector: 'app-root'

Ahí también se observa que se declaran dos ficheros: Uno para el html del componente, otro css:

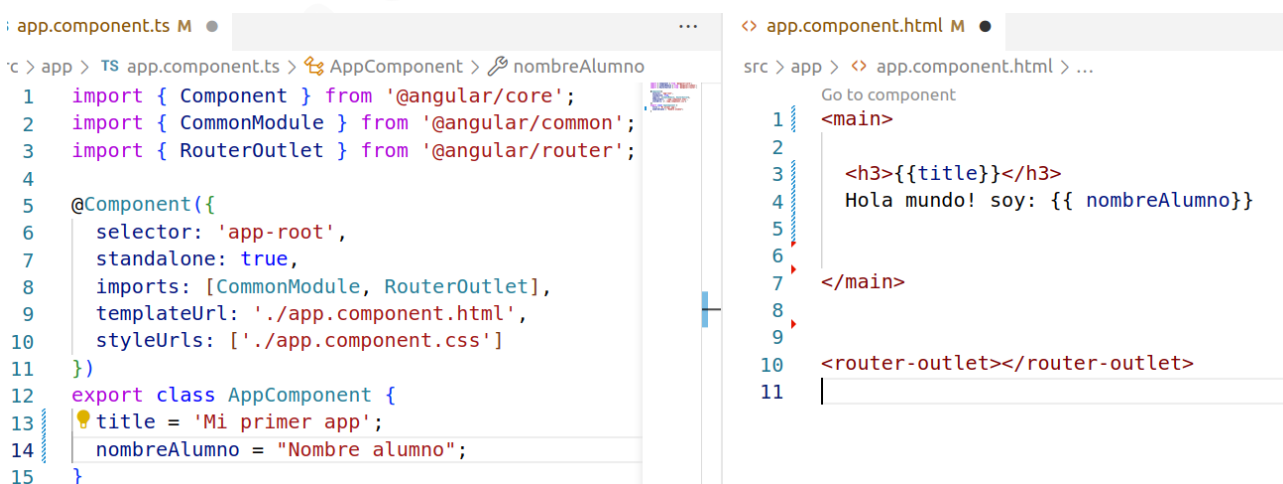
templateUrl: './app.component.html'

styleUrls: ['./app.component.css']

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

● **Práctica 1:** Crear un proyecto llamado: nombrealumno-practicass-angular  
Ir al fichero: app.component.html borrar el contenido y escribir el clásico hola mundo:  
hola mundo!.

La forma en la que se vincula la vista ( app.component.html en este caso ) con el controlador ( app.component.ts en este caso ) lo vamos a ver declarando una variable allí y visualizarla en el html usando: `{{nombrevARIABLE}}` Esto es, usaremos la doble llave: `{{ }}` para pasar información al html



```
app.component.ts
1 import { Component } from '@angular/core';
2 import { CommonModule } from '@angular/common';
3 import { RouterOutlet } from '@angular/router';
4
5 @Component({
6   selector: 'app-root',
7   standalone: true,
8   imports: [CommonModule, RouterOutlet],
9   templateUrl: './app.component.html',
10  styleUrls: ['./app.component.css']
11 })
12 export class AppComponent {
13   title = 'Mi primer app';
14   nombreAlumno = "Nombre alumno";
15 }

app.component.html
1 <main>
2
3   <h3>{{title}}</h3>
4   Hola mundo! soy: {{ nombreAlumno}}
5
6
7 </main>
8
9
10 <router-outlet></router-outlet>
11
```

● **Práctica 2:** Modificar los dos ficheros de la forma descrita en la imagen para que aparezca el nombre y el título en el html

## Componentes

La forma de trabajar de Angular es mediante componentes. Los componente son objetos completos, que renderizan un html y css dado coordinado por un controlador.

Hay una forma con angular cli para crear un nuevo componente:

```
ng generate component prueba
```

El comando anterior creará una carpeta dentro de: **app**, con los ficheros que usa el componente

Para que un componente sea reconocido y usado lo debemos importar en: app.component.ts

En la imagen siguiente se muestra como se ha importado ( observar que lo que se hace es declarar PruebaComponent en el array: imports ) Una vez importado podemos usar el selector en el html. Observar que aparece en: app.component.html el selector del componente: <app-prueba> Finalmente se muestra el html del componente: prueba.component.html donde se ha escrito: “Mi primer componente!!”

```
app.component.ts M ●  app.component.ts M ●  <> app.component.html M ×  ...  tml M  <> prueba.component.html U ●
src > app > TS app.component.ts > AppComponent  src > app > <> app.component.html > main  src > app > prueba > <> prueba.component.html
1 import { Component } from '@angular/core';
2 import { CommonModule } from '@angular/common';
3 import { RouterOutlet } from '@angular/router';
4 import { PruebaComponent } from './prueba.component';
5
6 @Component({
7   selector: 'app-root',
8   standalone: true,
9   imports: [
10    CommonModule,
11    RouterOutlet,
12    PruebaComponent
13  ],
14   templateUrl: './app.component.html',
15   styleUrls: ['./app.component.css']
16 })
17 export class AppComponent {
18   title = 'app';
19 }
20
21 src > app > <> app.component.html > main  src > app > prueba > <> prueba.component.html
Go to component  Go to component  Go to component
1 <main>
2
3
4 <h3>{{title}}</h3>
5 Hola mundo! soy: {{ nombreAlumno }}
6
7 <app-prueba></app-prueba>
8
9
10 </main>
11
12
13 <router-outlet></router-outlet>
14
15
16 src > app > prueba > <> prueba.component.html
Go to component
1 <p>
2 Mi primer componente!!
3
4 </p>
5
```

● **Práctica 3:** Crear el componente descrito y cargarlo con el selector en app.component.html

## Bucles en templates: \*ngFor

Mediante: \*ngFor podemos realizar un bucle. Lo pondremos en la etiqueta que queremos repetir. Ejemplo

```
<ul>
<li *ngFor="let item of array" >
  {{item.id}}: {{item.nombre}}
</li>
</ul>
```

El código anterior repetirá la línea <li> por cada elemento del array. Ese dato lo toma del fichero .ts del componente ( el controlador )

```
export class EjemploTemplatesComponent {
  array = [
    {
      id: 1,
      nombre: 'Ana',
    },
    {
      id: 2,
      nombre: 'Mario',
    },
    {
      id: 3,
      nombre: 'Marta',
    },
  ];
}
```

- **Práctica 4:** Crear el componente gente y cargarlo con el selector en app.component.html. Ese componente tendrá creado un array con datos de personas ( nombre y edad ). Se debe mostrar el listado de personas, una persona por fila de la tabla

## Pasar información a un componente hijo

Sea un componente que tiene un selector: `<app-info>` podemos pasarle información enviando un atributo entre corchetes:

```
<app-info [dato]="5" />
```

Para recibir ese número 5 en el componente usaremos: `@Input`

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

export class InfoComponent {
  @Input() dato: number = -1;
}
```

- **Práctica 5:** Crear el componente: Tabla de tal forma que se le pase un número y genere una tabla html con una fila por cada una de las operaciones de la tabla de multiplicar. Así la tabla del 4 la generaríamos con:  
`<app-tabla [numero]=4 />`



## Eventos sencillos: (click)

Podemos poner en la template a un objeto un listener de eventos.

Se pone entre paréntesis el evento y se asocia a alguna acción:

```
<button (click)="aumentarContador()" >Aumentar contador </button>
```

En este ejemplo podemos tener en el controlador un atributo: contador y un método: aumentarContador() que lo que hace es aumentar el contador

● **Práctica 6:** Crear dos botones, uno para incrementar un contador y otro para decrementarlo. Se debe mostrar el contenido del contador en el template

## Formularios

Los formularios en angular precisan varios pasos:

- Crear una estructura ( es un JSON que se pasará a un objeto de angular: FormGroup ) que representa los datos que nos envía el formulario. Por ejemplo para recoger los datos de una persona:

```
export class PersonaCardComponent {  
  personaFormData = new FormGroup({  
    nombre: new FormControl(""),  
    peso: new FormControl(""),  
    sexo: new FormControl(""),  
    adulto: new FormControl(""),  
  });  
}
```

- La anterior línea fallará si no se hacen los imports correspondientes en el componente:

```
import { FormControl, FormGroup, ReactiveFormsModule } from '@angular/forms';
@Component({
  selector: 'app-persona-card',
  standalone: true,
  imports: [
    CommonModule,
    ReactiveFormsModule
  ],
  templateUrl: './persona-card.component.html',
  styleUrls: ['./persona-card.component.css'],
})
```

Se ha destacado en amarillo lo que tenemos que incorporar.

- En el template html creamos el formulario y lo vinculamos a la estructura que hemos creado en el controlador ( en el ejemplo lo hemos llamado: personaFormData )

```
<form
[formGroup]="personaFormData"
(submit)="guardarPersona()"
>

<input type="text" id="nombre" formControlName="nombre" placeholder="nombre" /><br>
<input type="text" id="peso" formControlName="peso" placeholder="peso" /><br>
<select name="sexo" formControlName="sexo" >
<option value="Hombre">Hombre</option>
<option value="Mujer">Mujer</option>
</select>
<br>
<label for="adultotru">Adulto</label>
<input type="radio" id="adultotru" formControlName="adulto" name="adulto" value="true">
<br>
<label for="adultofalse">Menor</label>
<input type="radio" id="adultofalse" name="adulto" formControlName="adulto" value="false">
```

```
<br>

<button type="submit" >Guardar</button>

</form>
```

Se ha destacado lo relevante: Tenemos que vincular a la estructura de datos creada en el controlador. Eso lo hacemos mediante: `[formGroup]="personaFormData"`

Luego cada campo del formulario le tenemos que poner un nombre para poderlo identificar ( **cuidado!** No vale con el id o el name hay que usar: `formControlName` )

Finalmente se le pone al formulario un evento submit: `(submit)` informando del método en el controlador que va a gestionar el formulario

- Tenemos que crear en el controlador el método que gestiona el formulario. En ese método podemos ya acceder a los datos enviados, mediante la estructura que hemos creado ( en el ejemplo `personaFormData` )

```
guardarPersona() {
    let persona = new Persona();
    persona.nombre = this.personaFormData.value.nombre??"";
    persona.peso = parseInt(this.personaFormData.value.peso?? '0');
    persona.sexo = this.personaFormData.value.sexo?"Mujer";
    persona.adulto = this.personaFormData.value.adulto == "true";

    this.personas.push(persona);
}

personas: Persona[] = [];
```

Observar que accedemos a la información mediante: `personaFormData.value`

● **Práctica 7:** Crear una app para almacenar datos de gatos: nombre, peso, edad, equivalenciaedadhumano, Se usará un formulario para enviar la información y se calculará la equivalencia de edad ( el formulario envía únicamente la edad real del gato ) mediante la tabla de equivalencia: (Edad gatuna / Edad humana)  
1 mes / 1 año, 2-3 meses / 3 años , 4 meses / 7 años, 5 meses / 8 años ,6 meses / 10 años  
Los datos almacenados se muestran en una tabla html

## Atributos y Condiciones en templates

Vamos a realizar el juego de piedra papel tijera.

Si vemos las dos imágenes, observamos que el botón de: “apostar” está disabled una vez finalizada la partida. Y el botón: “reiniciar” aparece ( no aparece hasta que finaliza la partida )

Para conseguir esos efectos acudimos a los atributos: []

Ya los hemos usado para: [formGroup] en los formularios y como atributos personalizados que enviamos a los componentes hijos ( en el componente hijo usamos @Input() )

Angular soporta múltiples atributos en los objetos de los templates. Como:

[disabled] → establece si está o no disabled. Ej:

```
<input type="text" [disabled]="atributobooleano" />
```

[ngStyle] → se le pasa una string con un estilo css entre llaves. Ejemplo:

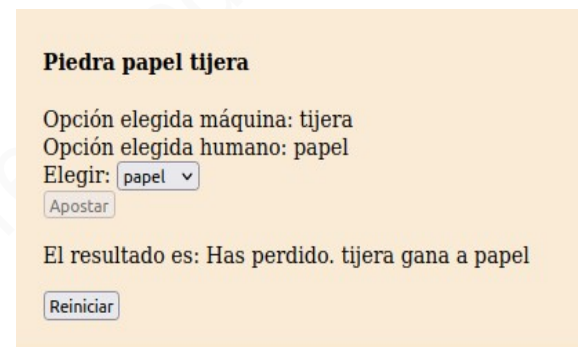
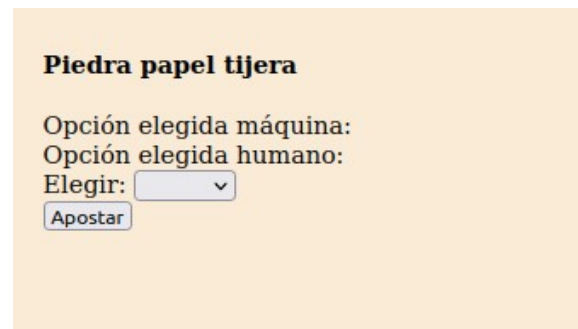
```
<h4 [ngStyle]="{'background-color': estilo}">Piedra papel tijera</h4>
```

Donde **estilo** es un atributo string declarado en el controlador

[innerHTML] → Si queremos pasarle una string con el html que queremos introducir. Ej:

```
<div [innerHTML]="textoprueba"></div>
```

Y así muchos más.



La directiva `*ngIf` nos permite actuar de forma condicional en las templates:

Por ejemplo, si queremos que **no** se muestre el resultado de una partida hasta que finalice:

```
<p *ngIf="partida.finalizada">
```

El resultado es: {{partida.resultado}}

```
</p>
```

No mostrará el contenido del párrafo hasta que `partida.finalizada` sea `true`

 **Práctica 8:** Crear el juego de piedra, papel, tijera usando: `*ngIf`, `[disabled]`, y `[ngStyle]`

## [(ngModel)]

Ya habremos ido observando que cuando usamos los corchetes: `[]` estamos tomando información del controlador. Ejemplo: `[innerHTML] = "atributodelcontrolador"` asignaba el contenido. También hemos visto que podemos enviar información al controlador por medio de los paréntesis: `()` ¿ cómo ? Pues enviando un evento ya que los establecemos dentro de paréntesis Ej (click):

```
<button (click)="agregarAlArray( 8 )" >Agregar 8 </button>
```

En el ejemplo anterior tendríamos un método en el controlador llamado: `agregarAlArray()` que guarda en un array el dato recibido

Hablamos del: “banana in a box”: `[(ngModel)]` cuando estamos haciendo el doble enlace: lo que está en el controlador viene a la template y viceversa. Usaremos `[(ngModel)]` para conseguir ese efecto:


- Necesitamos importar en el Componente FormsModule:

```
@Component({
  selector: 'app-persona-card',
  standalone: true,
  imports: [CommonModule, FormsModule ],
  templateUrl: './persona-card.component.html',
  styleUrls: ['./persona-card.component.css'],
})
export class PersonaCardComponent {
  dobleEnlazado: string = 'hola';
}
```

Y luego en la template escribimos un input con [(ngModel)]:

```
<input type="text" [(ngModel)]="dobleEnlazado" />
{{dobleEnlazado}}
```

Al escribir en el input vemos que nos escribe justo debajo lo mismo

 **Práctica 9:** reproducir el ejemplo anterior

## Comunicación entre componentes: Servicios

Para componentes que tienen una relación jerárquica padres-hijos nos podemos comunicar mediante el paso de atributos del padre al hijo: [atributo]= “información” y con eventos del hijo al padre: (notificación) = “onNotify()” ( lanza un evento pasando información a una función , luego el padre escucha el evento y recibe la información de la función ) Pero hay una opción mucha más general que son los servicios

La idea es que hay un almacén de información compartida con unos métodos asociados que inyectamos en los componentes al inicio. Entonces se puede leer y guardar información en ese almacén todos los componentes que han inyectado el servicios

- Primer paso: creamos el servicio:

```
ng generate service gatos
```

Lo anterior crear un servicio llamado: GatosService . En el servicio declaramos las estructuras y métodos que precisemos

Observar que angular genera las cosas necesarias para poder inyectar el servicio ( @Injectable )

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

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

  gatos: Gato[] = [];
  constructor() { }

  add(gato: Gato){
    this.gatos.push(gato);
  }

  getAll(){
    return this.gatos;
  }
}
```

- Segundo paso: Inyectar el servicio en los componentes

Una forma habitual de hacer la inyección es poner en el constructor la llamada al servicio:

```
constructor(private gatosService: GatosService ){  
}
```

Con lo anterior quedaría un servicio: `gatosService` ya inyectado. Si se quiere inicializar un array local con los datos del servicio habría que implementar: `onInit` en el componente ( te generará un método `onInit()` donde haríamos los pasos de inicialización )

Una alternativa más reciente y con más posibilidades es usar la función: **`inject()`** de **`@angular/core`**:

```
export class GatosComponent {  
  gatosService : GatosService = inject(GatosService);  
}
```

Veamos ejemplo:

```
//  
export class VisorComponent {  
  gatosService: GatosService = inject(GatosService);  
  arrayGatos: Gato[] = [];  
  constructor(){  
    this.arrayGatos = this.gatosService.getAll();  
  }  
}
```

```
1 <div>  
2   array gatos:  
3   {{arrayGatos | json}}  
4 </div>  
5
```

Supongamos que tenemos un formulario para crear gatos en un componente llamado: `GatosComponent` Entonces enviamos un nuevo gato al servicio como se ve en la imagen.

Ahora desde otro componente ( por ejemplo llamado `VisorComponent` ) tomamos el servicio y mostramos la información

```
gatosService : GatosService = inject(GatosService);  
  
gatoForm: FormGroup = new FormGroup({  
  nombre: new FormControl(""),  
  raza: new FormControl(""),  
  edad: new FormControl("")  
});  
gato: Gato|null = null;  
  
agregarGato(){  
  this.gato = new Gato(  
    this.gatoForm.value.nombre,  
    this.gatoForm.value.raza,  
    parseInt(this.gatoForm.value.edad)  
  );  
  this.gatosService.add(this.gato);  
}
```



● **Práctica 10:** Crear un servicio y un componente: VisorPartidasComponent que muestre el resultado de las partidas de piedra, papel, tijera. Cada vez que termina una partida se guarda la información en el servicio y el componente Visor muestra en una lista html el resultado de todas las partidas

## Router

Cuando estamos en una aplicación, por ejemplo de Android nativo ( java/kotlin con Android studio ), sabemos que tenemos una pantalla ( una Activity ) y pasamos a otra pantalla ( otra Activity ) De igual forma, en las aplicaciones web de servidor pasar de una pantalla a otra es tan sencillo como cambiar mediante un enlace de una página web a otra. En definitiva, según el uso que tengamos en nuestra aplicación, puede ser interesante pasar a otro renderizado completamente distinto con controles completamente distintos.

En el mundo de las aplicaciones web del lado cliente empezó a hablarse de las aplicaciones SPA:

SPA son las siglas de Single Page Application. Es un tipo de aplicación web donde **todas las pantallas las muestra en la misma página, sin recargar el navegador**

Pero veamos una definición más amplia que da wikipedia:

Una **single-page application (SPA)**, o aplicación de página única, es una aplicación web o es un sitio web que cabe en una sola página con el propósito de dar una experiencia más fluida a los usuarios, como si fuera una aplicación de escritorio. En un SPA todos los códigos de HTML, JavaScript, y CSS se cargan una sola vez<sup>1</sup> o los recursos necesarios se cargan dinámicamente cuando lo requiera la página, normalmente como respuesta a las acciones del usuario. La página no tiene que cargarse de nuevo en ningún punto del proceso y **tampoco es necesario transferir a otra página**, aunque las tecnologías modernas (como el pushState() API del HTML5) **permiten la navegabilidad en páginas lógicas dentro de la aplicación**.

En definitiva una SPA permite imitar el comportamiento de “cambio de página” ( y por tanto de pantalla )

## Instalación, uso del router

En angular precisamos poner un componente específico: `<router-outlet >` que permite visualizar el componente al que hemos mapeado una ruta. Y un fichero donde establecemos ese mapa de rutas/componentes ( típicamente: `routes.ts` )

**Nota:** en versiones anteriores a **Angular 15** ( antes de los standalone components ) había que establecer el router en **@NgModule** Para esas configuraciones buscar en la documentación oficial

Pero para hacer lo anterior hay que hacer algunos pasos de configuración en toda la aplicación:



En la imagen se muestran tres ficheros en la carpeta raíz: **main.ts**, **app.config.ts** y **app.routes.ts** Se debe comprobar que dentro de: `bootstrapApplication()` hemos dado de alta en `providers` a `provideRouter`. Es posible que ya angular nos haya hecho ese paso y lo haya dejado

separado en los tres ficheros señalados: en main.ts es donde está bootstrapApplication() y allí se carga un fichero llamado: appConfig. Ese fichero realmente lo que hace es poner un JSON con una etiqueta: providers y que llama a un fichero: routes que es el que realmente tiene las rutas.

Si angular no los ha generado, podemos en un primer acercamiento hacerlo nosotros así:

```
main.ts > ...
import { bootstrapApplication } from '@angular/platform-browser';
import { appConfig } from './app/app.config';
import { AppComponent } from './app/app.component';
import { routeConfig } from './app/routes/routeConfig';
import { provideRouter } from '@angular/router';
import { routes } from './app/app.routes';

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

TS app.routes.ts M X
src > app > TS app.routes.ts > ...
1 import { Routes } from '@angular/router';
2
3 export const routes: Routes = [
4
5
6
7 ];
8
```

Aquí vemos que **main.ts** incorpora el `provideRouter(routes)` que nos habilita el enrutado tomando el mapa de rutas establecido en: **app.routes.ts**

Ahora en: app.routes.ts

podemos poner la ruta que queramos:

¡ Observar que el **path** lo estamos poniendo **sin la barra inicial: “/”** !

También vemos que podemos hacer redirecciones: **redirectTo**. Se especifica otra ruta que hayamos declarado en el fichero.

Así la ruta por defecto de la app nos lleva al componente: GatosComponent

```
import { Routes } from '@angular/router';
import { GatosComponent } from './gatos/gatos.component';
import { VisorComponent } from './visor/visor.component';

export const routes: Routes = [
  {
    path: '',
    redirectTo: 'agregargato',
    pathMatch: 'full',
  },
  {
    path: 'vergatos',
    component: VisorComponent,
    title: 'lista de gatos',
  },
  {
    path: 'agregargato',
    component: GatosComponent,
    title: 'Agregar gato',
  },
];
```

Lo único que falta es decidir donde queremos que se visualicen los componentes cargados con las rutas. Para eso usamos: `<router-outlet>` Basta ponerlo en el componente principal , cargando en el componente: `app.component.ts` `RouterModule` y `RouterOutlet`:


`app.component.ts`:

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    CommonModule,
    RouterOutlet,
    GatosComponent,
    VisorComponent,
    RouterModule
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

ahora en la template `app.component.html`:

```
<main>
  <h3>App de gatos</h3>
  <router-outlet></router-outlet>
</main>
```

Observar que la etiqueta: App de gatos se mostrará en todas las páginas de la aplicación y justo debajo se cargará el componente que se acceda por el enrutado

 **Práctica 11:** Crear un router y poner el componente del juego piedra papel tijera en la ruta: `/jugar` y el componente para ver los resultados de las partidas en: `/verpartidas` Por defecto mostrará la nueva partida . Ejecutar esto como rutas en el navegador

Ejecutar como rutas en el navegador, tiene el problema de que nos vuelve a renderizar toda la aplicación ( se reinicia toda al app ) . Lo suyo es usar un <nav> interno. Para eso vamos a utilizar router-link

Para ello lo importamos en el controlador del componente

Luego lo que haremos es usar en la template unos enlaces: <a> específicos

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

  imports: [
    CommonModule,
    GatosComponent,
    VisorComponent,
    RouterOutlet,
    RouterModule,
    RouterLink,
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
```

```
<ul>

<li><a [routerLink]="['/vergatos']">ver gatos</a></li>
<li><a [routerLink]="['/nuevogato']">intro gato</a></li>

</ul>
```

Observar que las rutas se ponen con la barra inicial ( a diferencia de lo que vimos en el fichero de rutas )

## Parámetros en rutas

Para usar rutas paramétricas Ej: /vergatos/23 -> mostrar el gato con id 23 necesitamos **ActivatedRoute** que contiene información sobre la ruta y los parámetros. Veamos todos los pasos:

- Especificar en el fichero de rutas que es una ruta paramétrica (se hace poniendo dos puntos en la etiqueta): **path: 'vergatos/:id'**

- En la etiqueta routerLink pasamos el parámetro como una segunda opción del array:

```
<li><a [routerLink]="['/vergatos', 25]">ver gatos</a></li>
```

-- Habrá que inyectar ActivatedRoute en el controlador del componente que recibe el parámetro

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

@Component({ ...
})
export class VisorComponent {
  route: ActivatedRoute = inject(ActivatedRoute);
  idGato: number;

  gatosService: GatosService = inject(GatosService);
  arrayGatos: Gato[] = [];
  constructor() {
    this.idGato = Number(this.route.snapshot.paramMap.get("id"));
    this.arrayGatos = this.gatosService.getAll();
  }
}
```

En la imagen vemos como recuperamos el dato enviado en la ruta:

```
this.route.snapshot.paramMap.get("id")
```

Como se ve, utilizamos **ActivatedRoute.snapshot.paramMap.get()** y le pasamos la etiqueta que hemos definido en el fichero de rutas:

```
{
  path: 'vergatos/:id',
  component: VisorComponent,
  title: 'lista de gatos',
},
```

● **Práctica 12:** Crear un componente adicional en el del juego piedra papel tijera con una ruta paramétrica: /partida/:idpartida que muestre la partida con el id correspondiente

## Api

Podemos hacer uso de fetch o incluso instalar axios. El trabajo con la api sería pues, el habitual con promesas. Incluso se podría decir que es recomendable en casos en los que no haga falta nada más. Pero angular tiene potencial para nativamente con: **HttpClient** que se maneja con **observables**.

Los observables llegan más allá que las promesas. Pueden actuar como si fuera streaming (enviar todos los datos de una secuencia a un nuevo cliente, en lugar del último dato)

Los observables debemos tratarlos con cuidado, ya que usan suscripciones y hay que cancelar la suscripción una vez no lo precisemos

## Preparación de la app para httpClient

Al trabajar con standalone components se simplificó mucho el uso de @NgModule. La forma de preparar la app ahora consiste en:

- modificar **main.ts** ( en la carpeta raíz )

```
import { provideRouter } from '@angular/router';
import { routes } from './app/app.routes';
import { HttpClientModule, provideHttpClient } from '@angular/common/http';

bootstrapApplication(AppComponent, {
  providers: [
    provideRouter(routes),
    provideHttpClient()
  ],
}).catch((err) => console.error(err));
//providers: [provideHttpClient(withNoXsrfProtection())];
```

Observar que usamos: **provideHttpClient()** y lo ponemos en los **providers**. De esta forma habilitamos HttpClient en la app

- En el servicio que vayamos a hacer consulta a la api inyectar HttpClient:

```
import { Injectable, inject } from '@angular/core';
import { HttpClient } from '@angular/common/http';
export interface GatosApiArray {
  id: string;
  url: string;
  width: number;
  height: number;
  breeds: any[];
  favourite: Favourite;
}
export interface Favourite {}

@Injectable({
  providedIn: 'root',
})
export class GatosService {
  http = inject(HttpClient);
  getGatosApi() {
    return this.http.get<GatosApiArray[]>(
      'https://api.thecatapi.com/v1/images/search?limit=10'
    );
  }
}
```

Tener en cuenta que **HttpClient.get()** devuelve un **observable**

- En el componente que se vaya a usar el servicio hay que hacer una suscripción al observable

```
export class VisorComponent {
  gatosApiArray: GatosApiArray[] = [];

  gatosService: GatosService = inject(GatosService);
  arrayGatos: Gato[] = [];
  constructor() {
    this.gatosService.getGatosApi().subscribe((data) => {
      this.gatosApiArray = data;
    });
  }
}
```

En el código vemos que, al montar el

componente estamos haciendo la suscripción. Esta suscripción lanzará el evento cada vez que haya un cambio y en este caso hará que: `this.gatosApiArray = data`

● **Práctica 13:** Reproducir el ejemplo anterior Crear un router y una página específica para ver la imagen del gato por id. Crear una página en la que se generen tantos enlaces como elementos del array de gatos obtenido por la consulta http anterior. Al pulsar sobre alguno de los enlaces se abrirá el componente-página que muestra la foto del gato



## Uso de Post


Es similar a get, pero pasando el objeto, por supuesto, en ocasiones tendremos que enviar cabeceras específicas ( para un token por ejemplo )

Un ejemplo sencillo:

```
crearLibro(): Observable<Libro> {  
  return this.http.post<Libro>('http://miapi/libros'),  
  {  
    id: 100,  
    title: 'Javascript para todos',  
    category: 'Programación',  
    author: 'Desconocido'  
  }  
};  
}
```

Vemos que primero se le pasa la url y luego el objeto ( el cuerpo del mensaje )

Lo interesante de post está en lo que devuelve, que es un observable. El manejo es el mismo que antes con get, ejecutamos un: subscribe() y dentro de subscribe le pasamos el callback

 **Práctica 14:** Buscar información respecto a put y delete con HttpClient. Crear una fake api para usarla en una app que maneje tareas ( el clásico todo app ) y que permita un CRUD completo usando HttpClient