Angular

ng new nombre-proyecto -> Crear nuevo proyecto Angular.

ng serve -o -> Arrancar servidor. La opción -o permite abrir el navegador directamente al arrancarlo. = **npm start**

npm i -> Instalar todas las dependencias.

ng new nombre-proyecto --skip-git -> Crea un proyecto sin inicializar un repositorio, de forma que podamos tener nuestros proyectos angular en nuestro repositorio personal sin provocar conflictos. **IMPORTANTE:** añadir a .gitignore el directorio **/node_modules**, para evitar subir esta carpeta al repositorio, ya que tiene un tamaño muy grande.

ng new nombre-proyecto --routing -> Genera el proyecto con el enrutado de Angular activo.

Extensiones útiles para VSCode: *vs icons, angular snippets v12, angular language service*

Ctrl + c -> Parar servidor

app.compnent.css + app.component.html + app.component.ts conforman un componente de Angular

ng generate (g) schematic nombre --skip-tests -> Crear schematic sin crear fichero de pruebas

Todo componente debe pertenecer a un módulo (solo uno).

Un componente está formado por un fichero html, un css y un ts.

Pasos para incluir un componente en el proyecto:

1. En el fichero ts del componente, hay que ver el nombre que aparece en la etiqueta 'selector.'
2. Luego hay que incluir ese componente en el app.component.html, el cual se cargará con el html por defecto que se genera en su html.
3. Para modificar el componente hay que modificar su fichero html.

Selector tipo "moustache" → Se le denomina template literals. Se escribe con llaves dobles {{ }} . Se utiliza para traer el valor de una variable directamente al html de un componente.

Selector [ ] -> Similar al selector moustache pero para usar dentro de etiquetas html.

Dentro del fichero ts del componente, encontramos un constructor y un init. Dentro del init se instancian las variables, al contrario que en java. El constructor lo utilizaremos para otra cosa más adelante.

console.log muestra por la consola del navegador lo que queramos.

En Angular existen diferentes formas de comunicación entre el controlador y el componente:

- Controlador: fichero .ts del componente.
- Vista: fichero .html del componente.

1. **One way:**
    a. **Controlador ⇒ Vista** (Template literal, sintaxis moustache *{{ }}*)
    b. **Vista ⇒ Controlador** (event binding, por ejemplo: (click)="metodo()", de manera que metodo() debe ser definido en el fichero .ts)
    c. **Vista ⇐⇒ Controlador** (Property binding, por ejemplo: <img [src]="expresion">, en el fichero .ts existe una variable "expresion" donde esta almacenada la URL de la imagen. También podemos invocar a un método: <img [src]="metodo('parametro')">, el cual debe estar definido en el fichero .ts)

2. **Two-way data binding:** Se realiza a través de un módulo que hay que importar en el app.module.ts → *FormsModule*
   La comunicación se mantiene entre vista y controlador en todo momento.
   Se utiliza la sintaxis Banana in a box → [( )]

```
<p>
    VALOR ACTUAL DEL NOMBRE: {{nombre}}<br/>
    <label for="name">Nombre</label>
    <input id="name" type="text" [(ngModel)]="nombre" />
    <input type="button" (click)="saveForm()" value="Guardar" />
</p>
```

ng g s ruta_servicio (--skip-test) → Generar un servicio

Para inyectar dependencias tenemos que declarar una instancia de la clase como argumento del constructor de tipo private.

Declarar tipo de la variable → **nombre_variable: tipo_variable**

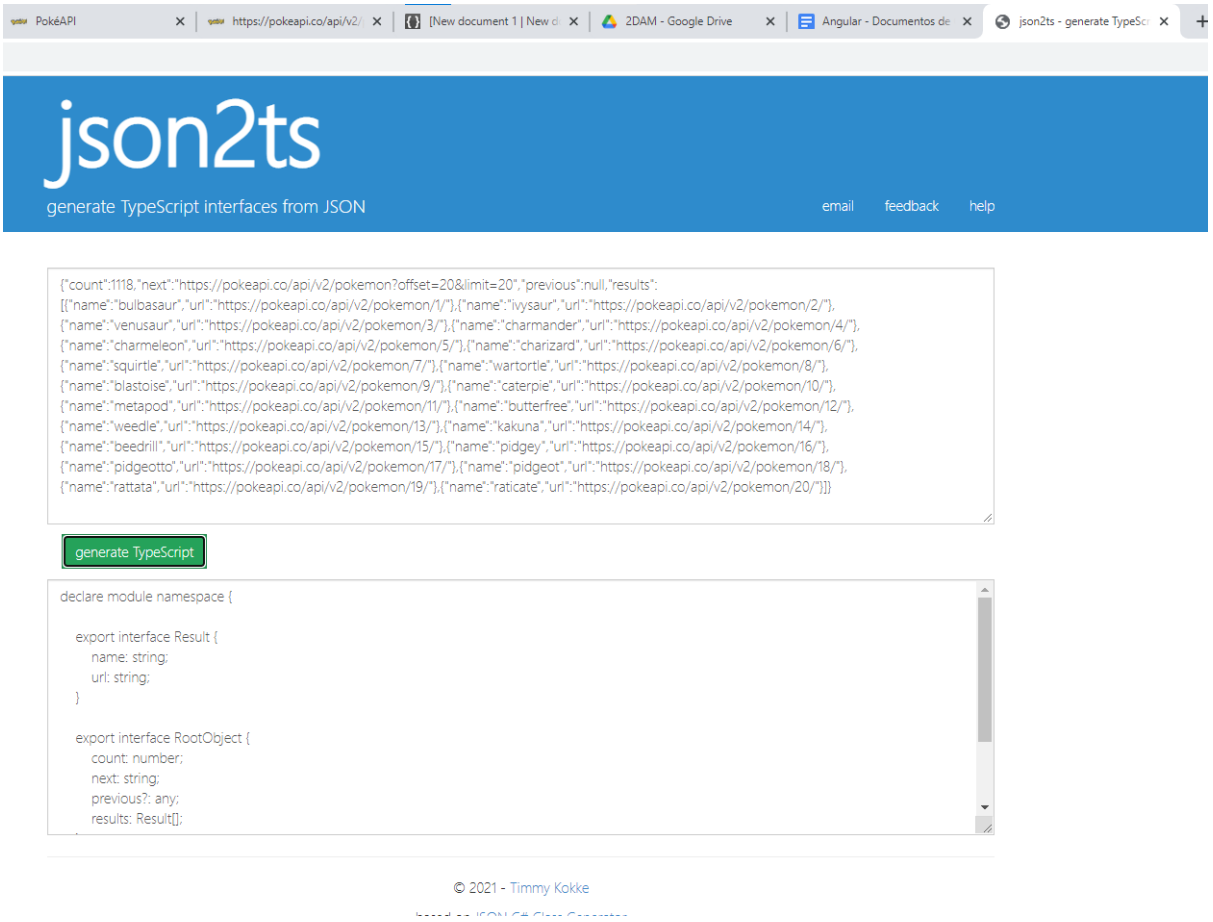
https://angular.io/guide/http

Hay que importar **HttpClientModule**, y luego declarar una variable de ese tipo en la clase del servicio, justo dentro de los parámetros del constructor.

```
const API_BASE_URL = 'https://pokeapi.co/api/v2/';
```

Esto sería como el @RequestMapping en Java.

http://json2ts.com → Para convertir JSON en interfaces de TypeScript.




```
previous?: any;
```

La interrogación significa que este parámetro es opcional, es decir, puede venir a null.

```
getPokemonList(){
```

```
  this.http.get<PokemonListResponse>(`${API_BASE_URL}/pokemon`);
}
```

Forma de una petición tipo GET dentro de un servicio. Las comillas francesas más la sintaxis ***${ }*** indican que se trata de una variable y no un string, de forma que podemos concatenar la ruta de la petición a continuación.

```
getPokemonList(): Observable<PokemonListResponse>
```

Encapsulamos la respuesta de la petición en un objeto tipo Observable, de forma que la petición devuelve el resultado una vez que éste ha sido obtenido.

```
pokemonList: PokemonListResponse | undefined;
```

Especificamos que pokemonList puede ser de tipo PokemonListResponse o de tipo indefinido.

```
ngOnInit(): void {
  this.pokemonService.getPokemonList().subscribe(resultado => {
    console.log(resultado);
  });
}
```

Llamamos desde el controlador de pokemonList al método GET. El método subscribe() de la clase Observable nos permite suscribirnos al resultado de la petición GET, de forma que quedará pendiente hasta que devuelva un resultado. Como argumento recibe una función que imprimirá el resultado por pantalla.

Importar todos los módulos

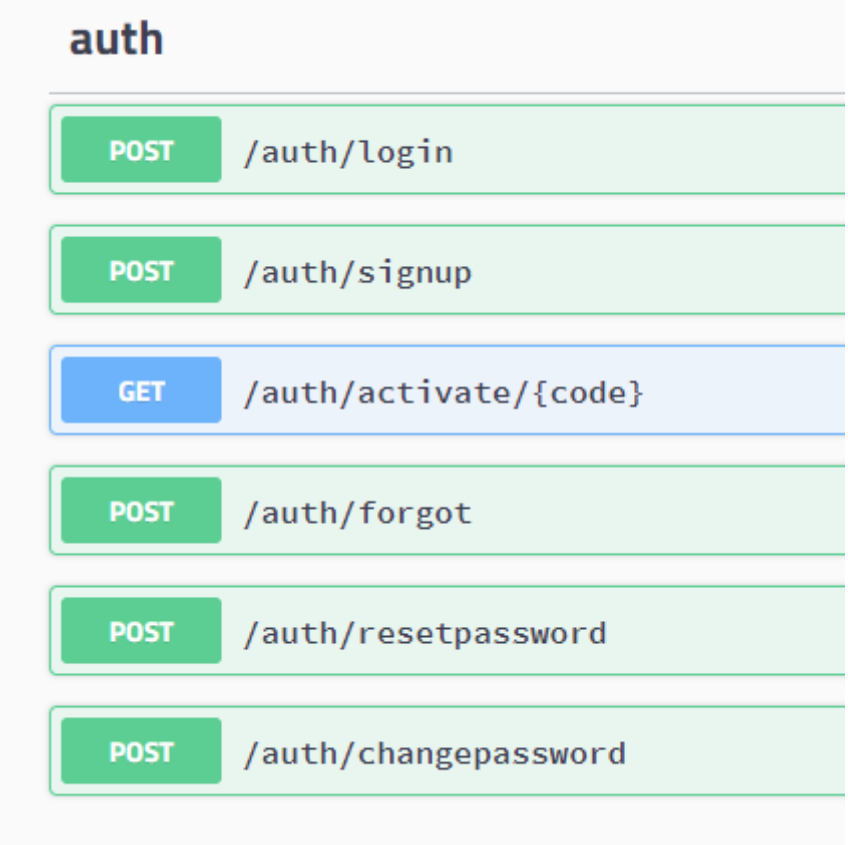
ng g m material-imports

Sacar la carpeta y eliminarla, excepto el único archivo que tiene. Dentro de este, copiamos y pegamos, tanto en imports como en exports.

Luego en app.module importamos ese archivo.

API Servicios

Endpoint → Punto de la lógica de la API donde se aglutinan diferentes procesos propios de la API.



ng g s services/auth --skip-tests para crear el servicio del login con el nombre que queramos.

Una vez creado, como en ejemplos anteriores, en el constructor de la clase tenemos que pasarle un parámetro privado HttpClient.

Todo servicio debe tener:

```
const API_BASE_URL = 'https://www.minitwitter.com:3001/apiv1';
```

```
constructor(private http: HttpClient) { }
```

En fichero environment.ts

```
export const environment = {
  production: false,
  apiBaseUrl: 'http://localhost:4200/api'
};
```

En fichero environment.production.ts

```
export const environment = {
  production: true,
  apiBaseUrl: 'https://www.minitwitter.com:3001/apiv1'
};
```

En fichero auth.service.ts

```
login(){
  let requestUrl = `${environment.apiBaseUrl}/${AUTH_BASE_URL}/login`;
}
```

forgot() {

let requestUrl = `${environment.apiBaseUrl}/${AUTH_BASE_URL}/forgot`;

Para la petición de login, copiamos la respuesta que este método da en el Swagger y lo ponemos en el json2ts para que nos lo pase como interfaz. Luego creamos la carpeta **model / interfaces**, y dentro creamos una interfaz para auth.

A la interfaz la llamamos **AuthListResponse**.

Dentro de esta carpeta model, creamos un apartado dto y un documento auth.dto.ts donde ponemos la clase **AuthLoginDto**. Allí creamos un constructor con las variables email y password vacías para que no haya errores.

Importante diferenciar entre la clase que envía la petición y la que responde.

evento “change( )” ha cambiado a “selectionChange( )”

Rutas

ng new angular-rutas --skip-git

const routes: Routes = [ ] es lo mismo que ***const routes: Route[ ] = [ ]***

Dentro de Routes = [
{ path: 'login', component: loginComponent },
{ path: 'dashboard', component: DashboardComponent }
];

DashboardComponent

Tenemos dos formas de acceder al Login, con routerLink = '/login', o con un método y un evento.

En el enrutado se tiene en cuenta el orden en el que se han declarado las rutas, por lo que hay que tener especial cuidado con esto. Por ejemplo, podemos tener dos rutas 'user/add' y 'user/list', donde el enrutador por defecto tomará la primera coincidencia que encuentre, por lo que puede equivocarse. Para ello podemos definir la propiedad **pathMatch: 'full'** al definir la ruta para que la coincidencia sea obligatoriamente completa y no parcial.

AppRoutingModule (excerpt)

```
const routes: Routes = [
  { path: 'first-component', component: FirstComponent },
  { path: 'second-component', component: SecondComponent },
];
```

Para definir el comportamiento de la aplicación cuando la acabamos de iniciar, podemos definir la siguiente ruta:

{ path: ' ', redirectTo: '/first-component', pathMatch: 'full' }

Donde el path en blanco indica que aún no se ha cargado nada (es decir, al iniciar) y redirectTo nos indica una ruta a cargar en vez de un componente. El pathMatch a full previene que tome como un path en blanco una ruta como 'user/ ', ya que no es lo que queremos definir.

*{path: '**', component: PageNotFoundComponent }*

Cuando no se encuentra la página a la que el usuario quiere acceder. Esto se suele poner lo último en el array. ** significa "cualquier otra cosa", es decir, aquello que no hayamos definido en nuestras rutas.

path: 'edit/:id'

Los dos puntos tras una barra indica que vamos a declarar un parámetro y a continuación su nombre, en este caso, "id". En el html, le pasaríamos esa ruta con el id correspondiente.

***ngFor = "let student of students" [routerLink] = "['/student', student]"**

<router-outlet></router-outlet> → Esta etiqueta se declara en app.component.html en sustitución a declarar componentes por separado. Esta etiqueta carga lo que el array de rutas le indique, con los parámetros y rutas que se hayan definido.

Para poder utilizar esta etiqueta y sus métodos necesitaremos importar el RouterModule en el app.module.ts.

1. Import `ActivatedRoute` and `ParamMap` to your component.

In the component class (excerpt)

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

These `import` statements add several important elements that your component needs. To learn more about each, see the following API pages:

- `Router`
- `ActivatedRoute`
- `ParamMap`

2. Inject an instance of `ActivatedRoute` by adding it to your application's constructor:

In the component class (excerpt)

```
constructor(
  private route: ActivatedRoute,
) {}
```

3. Update the `ngOnInit()` method to access the `ActivatedRoute` and track the `id` parameter:

In the component (excerpt)

```
ngOnInit() {
  this.route.queryParams.subscribe(params => {
    this.name = params['name'];
  });
}
```

queryParams son los parámetros que se pasan por la URL.

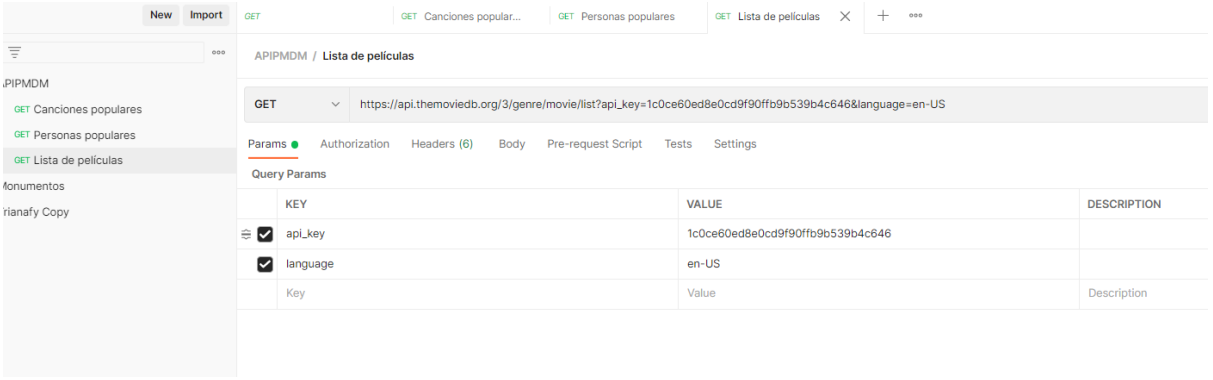
ENTRADAS

Podemos iterar un componente que hayamos creado declarando el bucle dentro de la etiqueta del componente. Para ello tendremos que haber implementado previamente lo que queremos iterar dentro del html del componente.

@Input → pa el año que viene :)

API EN TMDB

Ejemplos de cómo se harían las peticiones.



Ejemplo Movies

Generamos los servicios, y ahí hacemos un método para conseguir las películas populares. Antes ponemos el HttpClient y lo importamos.

Ahora, creamos una interfaz con un archivo que se llame movies-popular y le ponemos el json de movie/popular como código con Paste JSON as Code (Ctrl + Shift + P) Dentro de este código, cambiamos el Result [ ] por Movie [ ], y el nombre de la interfaz por Movie.

const API_BASE_URL = `{environment.apiBaseUrl}/movie`

(Todo lo que hay de la api en un environment se pega en el otro)

```
export interface MoviesListResponse {
    page:          number;
    results:       Movie[];
    total_pages:   number;
    total_results: number;
}

export interface Movie {
    adult:             boolean;
    backdrop_path:     string;
    genre_ids:         number[];
    id:                number;
    original_language: string;
    original_title:    string;
    overview:          string;
    popularity:        number;
    poster_path:       string;
    release_date:      Date;
    title:             string;
    video:             boolean;
    vote_average:      number;
    vote_count:        number;
}
```

```
const API_BASE_URL = `${environment.apiBaseUrl}/movie`;

You, seconds ago | 2 authors (María Inmaculada Domínguez Vargas and others)
@Injectable({
  providedIn: 'root'
})
export class MoviesService {

  constructor(private http: HttpClient) { }

  getPopularMovies(): Observable<MoviesPopularResponse> {
    return this.http.get<MoviesPopularResponse>(`${API_BASE_URL}/popular?api_key=${environment.apiKey}`);
  }

}
```

Creamos el componente moviesPopularList. En su ts, ponemos el moviesServices en el constructor. Lo mismo con el movie-item.

Volvemos al ts de MoviesPopularComponent:

```
export class MoviesPopularListComponent implements OnInit {

  popularMovies: Movie[] = [];      You, seconds ago • Uncommitted changes

  constructor(private moviesService: MoviesService) { }

  ngOnInit():void {
    this.moviesService.getPopularMovies().subscribe(popularMoviesResponse => {
      this.popularMovies = popularMoviesResponse.results;
    });
}

}
```

En el html del MoviesPopular, invocamos al app-movie-item. Dentro de la etiqueta, ponemos *ngFor =”let movie of popularMovies”

```
movies-popular-list.component.html M ×   movies-popular-list.component.spec.ts   TS movies-popular-list.component.t
src > app > movies-popular-list > movies-popular-list.component.html > app-movie-item
      You, a minute ago | 1 author (You)
1   <app-movie-item *ngFor ="let movie of popularMovies" [movieInput]="movie"></app-movie-item>
```

En el MovieItem.ts, ponemos un movieInput : Movie en el input, para luego en la etiqueta <app-movie-item> poner un [movieInput]=”movie”.

En el html del movie-item algo como:

```
pp > movie-item > movie-item.compo
<p>{{movieInput.title}}</p>
```

Dentro de Routes = [
        { path: 'movie-item', component:MovieItemComponent },
];

<router-outlet> en el app.component.html

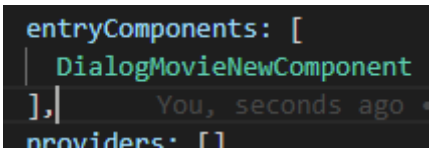
MENÚ LATERAL

```
<mat-sidenav-container class="example-container">
    <mat-sidenav #sidenav mode="push">
        <p><button mat-button (click)="sidenav.toggle()">Toggle</button></p>
    </mat-sidenav>

    <mat-sidenav-content>
        <p><button mat-button (click)="sidenav.toggle()">Toggle</button></p>
    </mat-sidenav-content>
</mat-sidenav-container>
```

#sidenav identifica al componente mat-sidenav.

DIÁLOGOS

Hay que añadir en el app.module.ts:



Guardar datos en el navegador:

sessionStorage.setItem('idVivienda', id);

donde idVivienda es el nombre e id la variable que pasamos

Rescatar datos del navegador (previamente guardados):

let idV = sessionStorage.getItem('idVivienda');

Flex layout

https://github.com/angular/flex-layout

Para utilizarlo necesitamos instalar una librería:

```
npm install @angular/flex-layout @angular/cdk
```

Luego tendremos que importarlo en el archivo app.module.ts

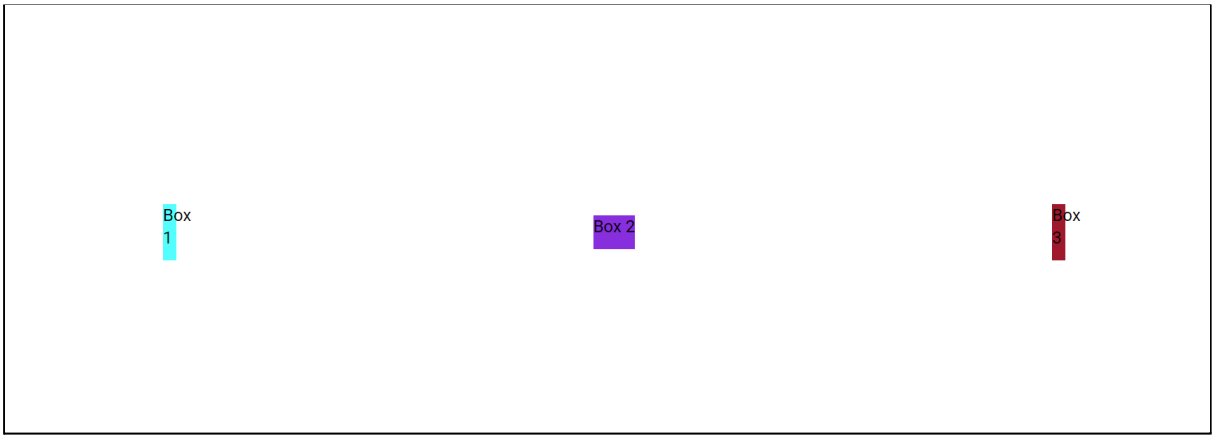
```
<div fxFlexFill fxLayout="row">
    <div fxFlex="1">

    </div>
</div>
```

A través de la propiedad **fxFlexFill**, le diremos al contenedor que ocupe todo el ancho. **fxLayout** indica el sentido de colocación de los elementos en su interior (row o column). Por último, la propiedad **fxFlex** en cada elemento hijo indica la relación de tamaño respecto a los demás elementos.

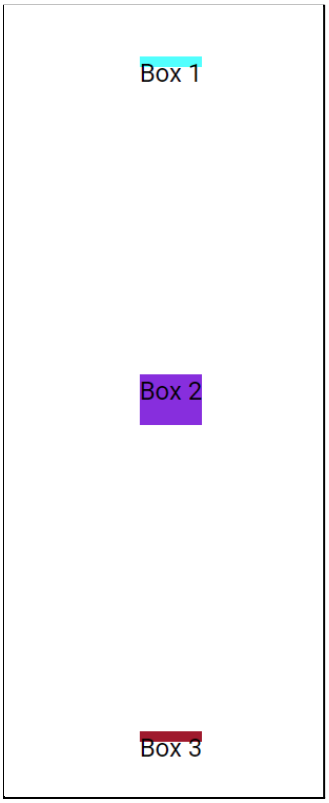
Guía de uso con ejemplos

Utilizando un ejemplo sencillo con un contenedor padre y 3 contenedores hijos, vamos a probar las siguientes propiedades:

- **Layout direction - fxLayout**→ row / column
    - row

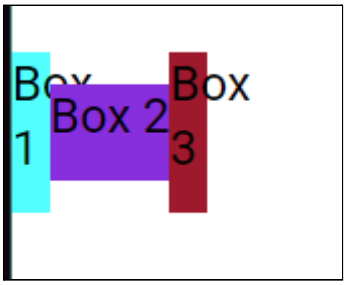


- column

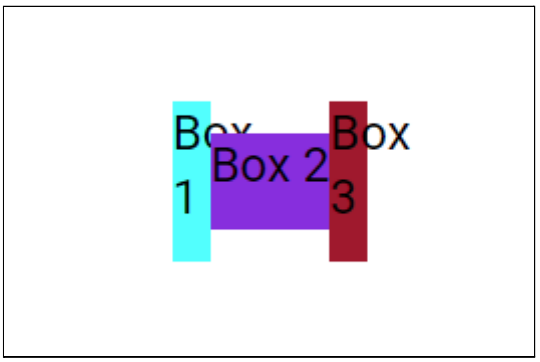


- **Layout align (main axis) fxLayoutAlign** → none / start (default) / center / end / space-around / space-between / space-evenly

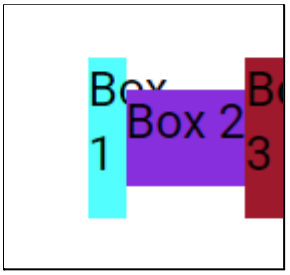
    - none → start by default



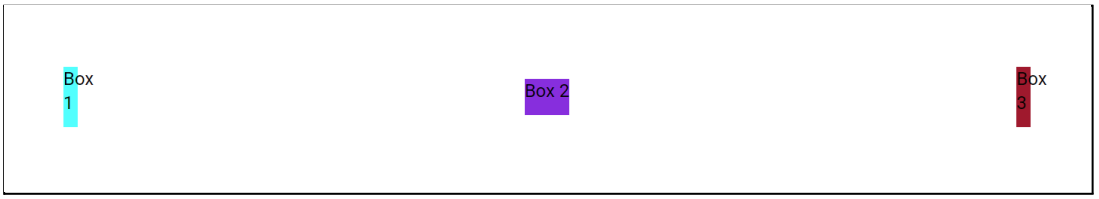
- center



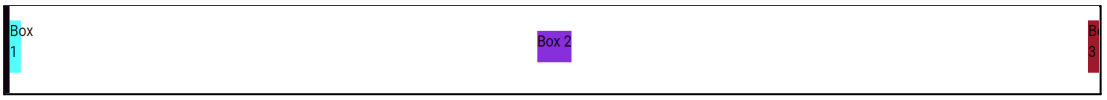
- end



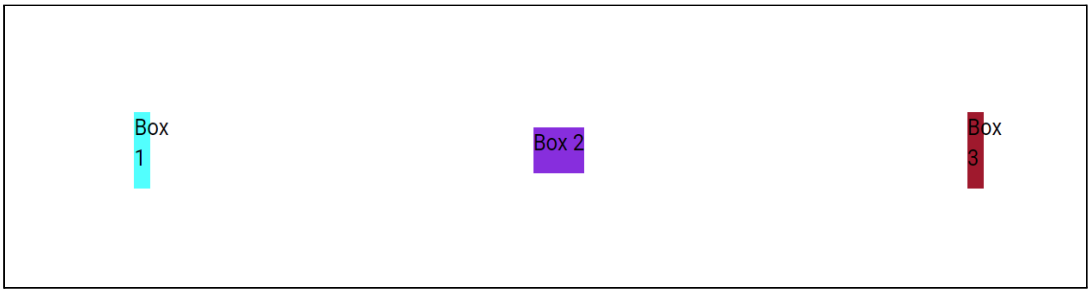
- space-around



- space-between

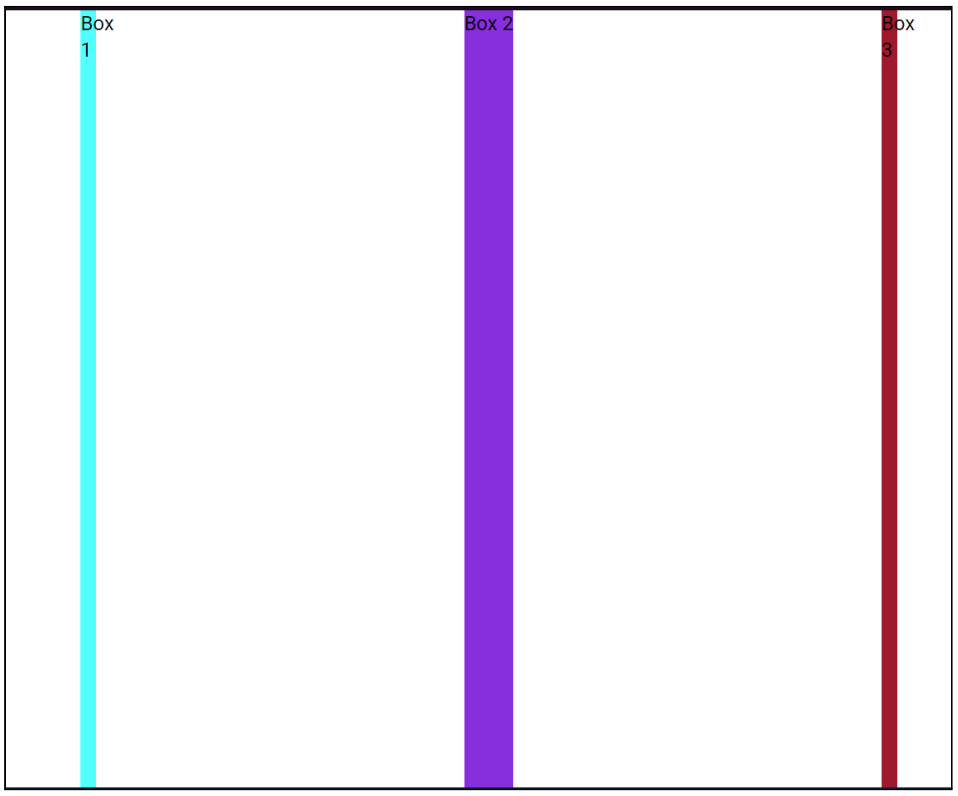


- space-evenly

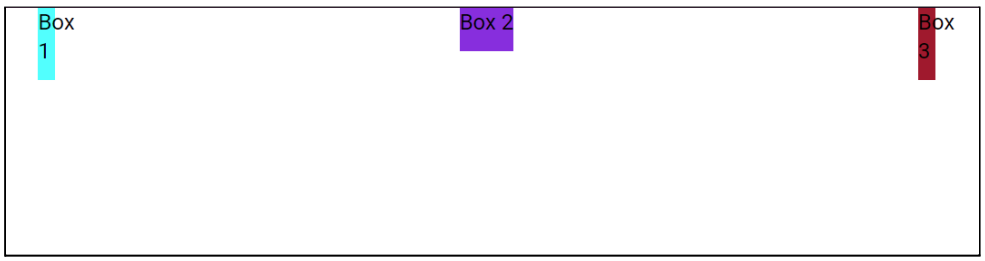


- **Layout align (cross axis) - fxLayoutAlign** → none / start / center / end / stretch (default)

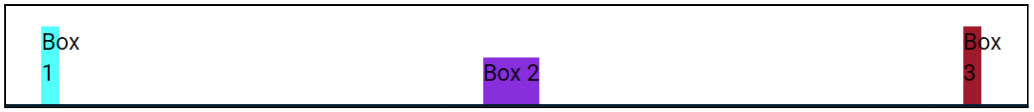
    - none → stretch by default



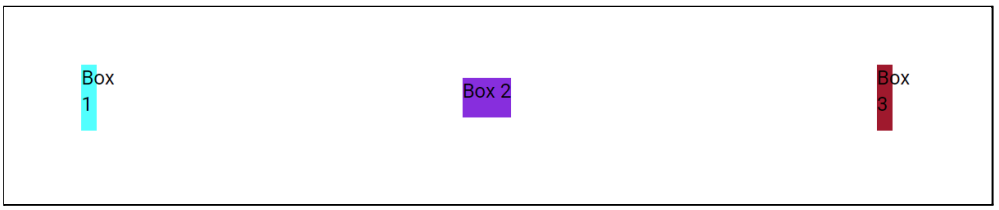
    - start



    - end



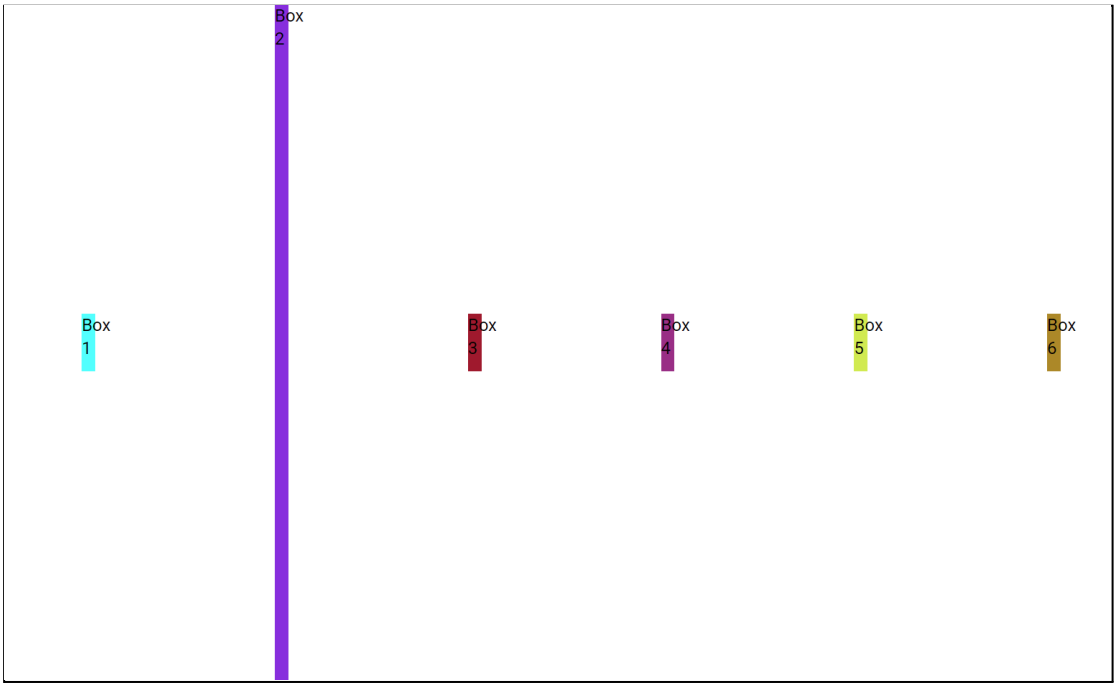
- center



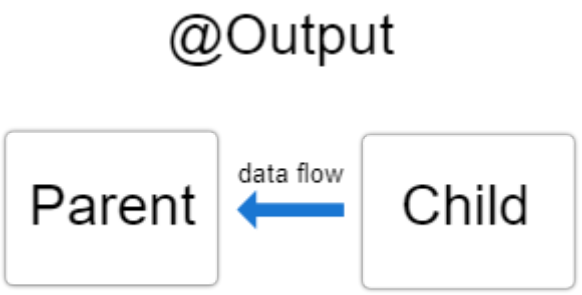
- **Layout fill - fxFill**

- **Layout Gap - fxLayoutGap**

- **Flex with Align Self - fxFlexAlign**



Mandar datos a un componente padre



Lo realizaremos a través de un *event binding,* desde el componente hijo.

```
<app-item-output (newItemEvent)="addItem($event)"></app-item-output>
```

- **Cargar en la misma página el componente que seleccionemos**

  https://github.com/miguelcamposedu/starwars-webapp/tree/output-event