PMDM02.- PROGRAMACIÓN DE APLICACIONES PARA DISPOSITIVOS MÓVILES.

1- CLI IONIC.

En nuestro caso vamos a utilizar Visual Studio Code para el desarrollo de las aplicaciones con Ionic. A diferencia de otros IDEs que incluyen la posibilidad de: Archivo->Nuevo->Proyecto, con este entorno no disponemos de esta opción, ni de otras como publicar. Por lo tanto, necesitamos una herramienta que nos permita crear proyectos, añadir "elementos" a la aplicación o publicar nuestra aplicación. Para ello Ionic cuenta con una CLI que nos permite realizar todas las acciones necesarias para el desarrollo de las aplicaciones. La documentación la podemos encontrar en el siguiente enlace: https://ionicframework.com/docs/cli

Comandos para realizar la instalación

Aunque la instalación ya está realizada se incluyen estos comandos para tener los contenidos ordenados y completos.

Antes de realizar la instalación es necesario tener instalado Node.js.

Instalación CLI

```
npm install -g @ionic/cli
```

Listado de comandos

La lista completa de comandos se pueden consultar en la documentación oficial.

A continuación vamos a ver los comandos básicos para poder crear proyectos y generar diferentes elementos:

Start

Documentación comando

Comando para crear un nuevo proyecto. Antes de lanzarlo debemos estar en la carpeta que queramos crear el proyecto. El proyecto se crea en una carpeta nueva, por lo que podríamos crearla en un repositorio o carpeta en la que tengamos todos los proyectos.

```
ionic start nombreProyecto nombrePlantilla [options]
```

Generate

Documentación comando

Comando para crear páginas, pipes o proveedores. Se pueden crear más elementos, aunque durante el curso no utilizaremos todas las opciones. Con versiones anteriores la generación de nuevos elementos no era tan sencilla. Pero, con la nueva versión actual el trabajo es más sencillo.

```
ionic generate tipoElemento nombreElemento [options]
```

Serve

Documentación comando

Comando para inciar un servidor de pruebas local. Gracias al uso de Capacitor en el servidor local se puede utilizar funcionalidad que con Cordova no era posible utilizar. De esta forma, no será necesario utilizar Android Studio en algunos casos.

```
ionic serve [options]
```

Capacitor add

Documentación comando

Comando para añadir una plataforma. En versiones anteriores se utilizaba el comando cordova add. Pero, en nuestro caso utilizaremos Capacitor.

```
ionic capactior add nombrePlataforma [options]
```

Capacitor build

Documentación comando

Comando para construir un proyecto para una plataforma. En versiones anteriores se utilizaba el comando cordova build. Pero, en nuestro caso utilizaremos Capacitor.

```
ionic capactior build nombrePlataforma [options]
```

Capacitor run

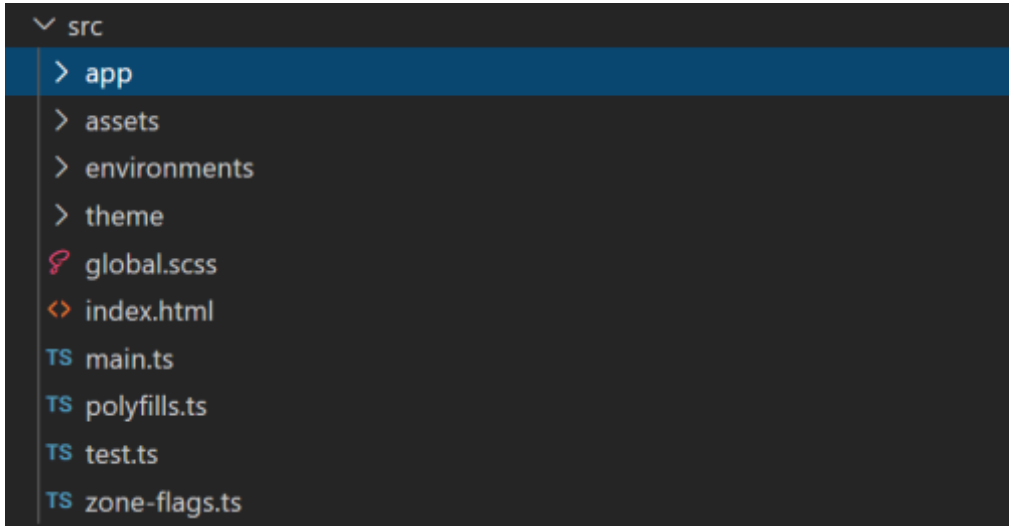
Documentación comando

Comando para lanzar un proyecto en un dispositivo conectado. En versiones anteriores se utilizaba el comando cordova run. Pero, en nuestro caso utilizaremos Capacitor. Desde Android Studio también se puede emular proyectos.

```
ionic capactior run nombrePlataforma [options]
```

2- ESTRUCTURA DE UN PROYECTO IONIC.

Dentro de la estructura de un nuevo proyecto de Ionic nos encontramos con los siguientes elementos:



BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

La caperta src cuenta con el código fuente de la aplicación. Dentro de esta carpeta se encuentran las carpetas app, assets, environments y theme. Aunque, a medida que incluyamos pipes y providers se crearán nuevas carpetas. Además, en recomendable crear una carpeta para incluir las clases del modelo. Si sólo contamos con una página o vista no hara falta incluir una carpeta para las vistas. En caso contrario, es recomendable crear una carpeta para todas las vistas.

2.1-INDEX.HTML.

Este fichero es el punto de entrada a la aplicación. Dentro del código se encuentra la etiqueta que va a servir para incluir la app que estamos desarrollando:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8" />
```

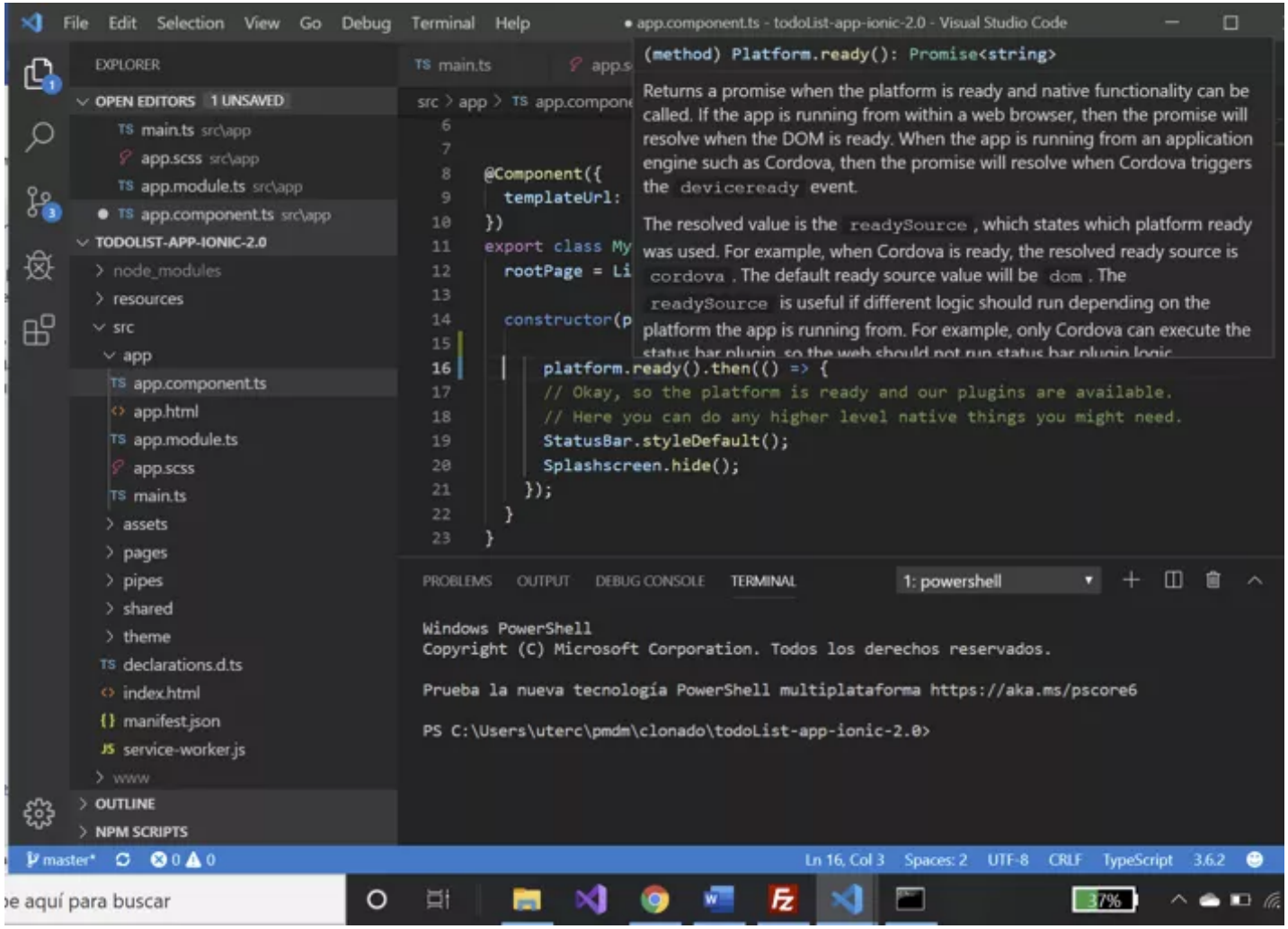
```
  <title>Ionic App</title>

  <base href="/" />

  <meta name="color-scheme" content="light dark" />
  <meta name="viewport" content="viewport-fit=cover, width=device-width, initial-scale=
  <meta name="format-detection" content="telephone=no" />
  <meta name="msapplication-tap-highlight" content="no" />

  <link rel="icon" type="image/png" href="assets/icon/favicon.png" />

  <!-- add to homescreen for ios -->
  <meta name="apple-mobile-web-app-capable" content="yes" />
  <meta name="apple-mobile-web-app-status-bar-style" content="black" />
</head>

<body>
  <app-root></app-root>
</body>

</html>
```

BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

3- PROGRAMACIÓN ASÍNCRONA.

La programación de dispositivos móviles implica el uso de programación asíncrona para evitar bloqueos en el hilo en ejecución. Para entender este concepto sirve una petición o a una base de datos local. Si la petición tardará 2 segundos (algo poco probable) el hilo se quedaría bloqueado y el interfaz de usuario se quedaría “tostado”.

Los métodos o funciones asíncronas devuelven tienen un retorno de tipo Promise<tipo> tal y como se observa en la siguiente imagen:



BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

Cuando hagamos uso de estas funciones deberemos esperar a que terminen y para ello utilizaremos la cláusula then o await. Si se quisiera captura una excepción ocurrida dentro de la función a la que se ha llamado se utilizará una cláusula catch, la cual funciona de la misma forma que un bloque catch de un bloque try/catch.

Con el siguiente ejemplo vamos aclarar lo explicado:

El método getNombre() retorna una Promise<string>. El método hace una petición a un servidor y desde el servidor se recibe un string.

```
let nombre = getNombre();
let sms = "Hola " + nombre;
console.log(sms);
```

Si probamos en código anterior veremos que en la consola no aparece nada más que Hola. Esto se debe a que no hemos esperado a que la respuesta que hemos realizado al servidor. Las llamadas a métodos asíncronos no detienen la ejecución, por lo tanto, no se espera a tener la respuesta del servidor antes de establecer el valor de la variable sms.

```
let nombre = "Variable sin inicializar";
getNombre().then((datos)=>{
        nombre = datos;
});
let sms = "Hola " + nombre;
console.log(sms);
```

Si probamos el código anterior tampoco funcionará ya que no hemos esperado a que el servidor nos responda para asignar el valor a la variable sms. Para que el código funcione será necesario incluir las tres líneas dentro del bloque then.

```
let nombre = "Variable sin inicializar";
getNombre().then((datos)=>{
        nombre = datos;
        let sms = "Hola " + nombre;
        console.log(sms);
});
```

Otra opción disponible es await. Si queremos utilizar await en vez de then, hay que declarar el método como async. Aunque, por ahora basta con entender los conceptos. Utilizando await estamos indicando que queremos esperar a que se resuelva la promise.

```
let nombre = await getNombre();
let sms = "Hola " + nombre;
console.log(sms);
```

3.1- FUNCIONES LAMBDA O ARROW FUNCTIONS.

Dentro de los bloques then utilizaremos funciones lambda para indicar qué se debe hacer una vez haya terminado la ejecución de la función asíncrona. La forma más sencilla de entender las funciones lambda es comparándolas con la firma de una función normal:

```
public void miFuncion (){...}
()=>{...}
public void miFuncion (param1){}
(param1)=>{...}
```

Viendo el ejemplo, lo que haremos será mantener el cuerpo de la función y los parámetros de entrada, incluir un símbolo igual y mayor y quitar el retorno y nombre de la función.

4- PAGES.

La carpeta pages contiene todas las vistas juntos con sus archivos de estilos y sus controladores. Es conveniente que los controladores únicamente se encarguen de inicializar las vistas y gestionar los eventos en las vistas. Por lo tanto, una vez se produzca un evento se recogerá en una función y dentro de la función se llamará a un proveedor externo para que realice el trabajo, a menos que las tareas a realizar sean mínimas. Más adelante veremos como generar proveedores y

como inyectarlos utilizando el patrón de diseño inyección por dependencias. Dejando de lado los estilos de cada una de las vistas, por cada page generada habrá una vista y controlador. Es obvio que la vista y su controlador deben estar enlazados de alguna manera. A partir de la versión 4 de Ionic este trabajo puede realizarse con Angular, Vue, React o JavaScript. Pero en veriones anteriores, como la nuestra, únicamente puede realizarse con Angular. No es necesario tener conocimientos previos de Angular, ya que únicamente utilizaremos bindings y directerivas estructurales.

- Binding o enlazado bidireccional: gracias este binding todos los cambios que se producen en controlador se ven en la vista y viceversa. Se debe tener cuidado cuando se editan instancias del modelo, ya que en caso de no estar editando una copia no habrá forma de descartar los cambios. En el siguiente ejemplo se enlazan la prodiedad description de la variable model del contralador con el componente ion-input.

```
<ion-input placeholder="Task description" clearInput
[(ngModel)]="model.description" name="description" required
#descriptionState="ngModel"></ion-input>
```

BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

- Binding o enlazado a una función: gracias a este binding se puede especificar que función del controlador se debe especificar al ocurrir un evento en la vista.

```
<button ion-fab (click)="showAddList()">
```

BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

- Representación de expresiones: si se desea mostrar el contenido de una variable del controlador en la vista se puede utilizar la presentación de expresiones con interpolaciones.

```
{{list.name}}
```

BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

- Directivas estructurales: gracias a estas directivas se puede modificar los templates o plantillas. Con la directiva *ngIf permite eliminar de la plantilla una sección

```
<ion-buttons end *ngIf="selectedList != null">
```

BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

La directiva *ngFor permite iterar sobre un array, por ejemplo para generar los elementos de una lista:

```
<ion-list>
  <ion-item *ngFor="let list of listsService.lists" (tap)="goToList(list)"
  (press)="selectList(list)" [ngClass]="{'selected-list': selectedList === list}">
    {{list.name}}
  </ion-item>
</ion-list>
```

BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

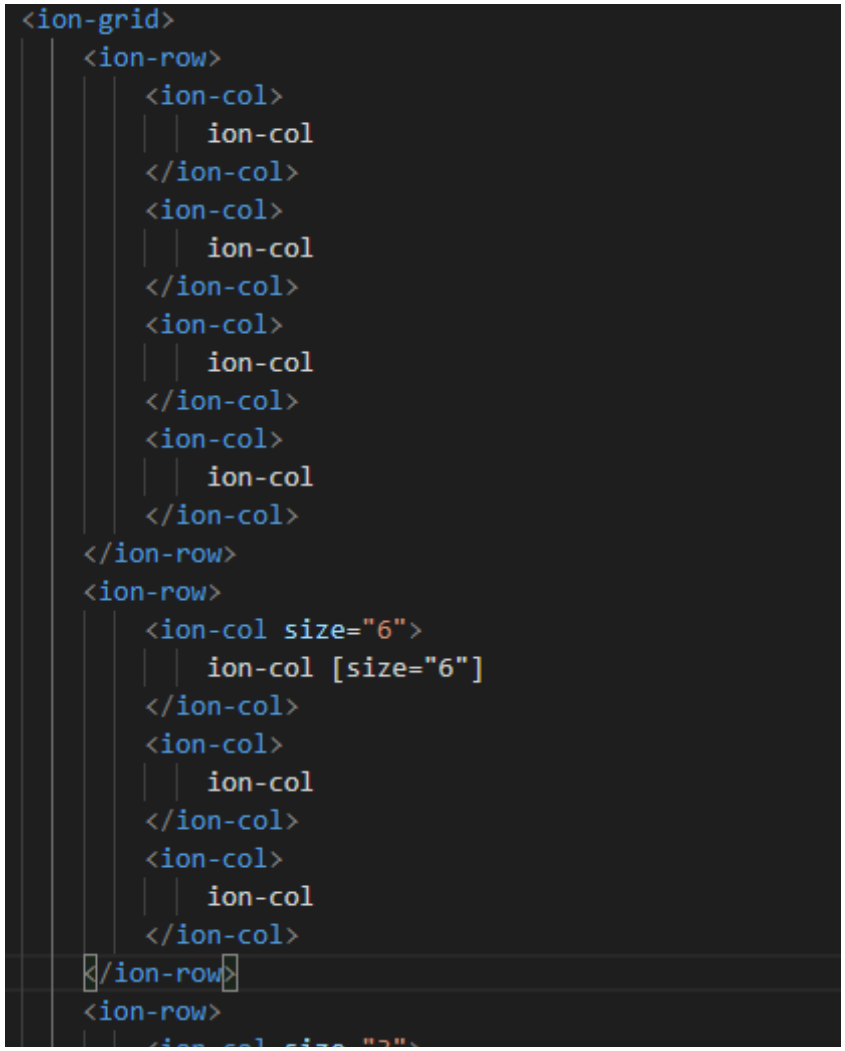
4.1- VISTAS.

Las vistas o páginas son documentos html. En ellos se pueden incluir tanto elementos html como component Ionic. Los elementos html se pueden consultar en el siguiente enlace: https://developer.mozilla.org/es/docs/HTML/HTML5/HTML5_lista_elementos. Para consultar la lista completa de componentes Ionic consultar el siguiente enlace: https://ionicframework.com/docs/components/. A continuación, veremos alguno de los componentes Ionic:

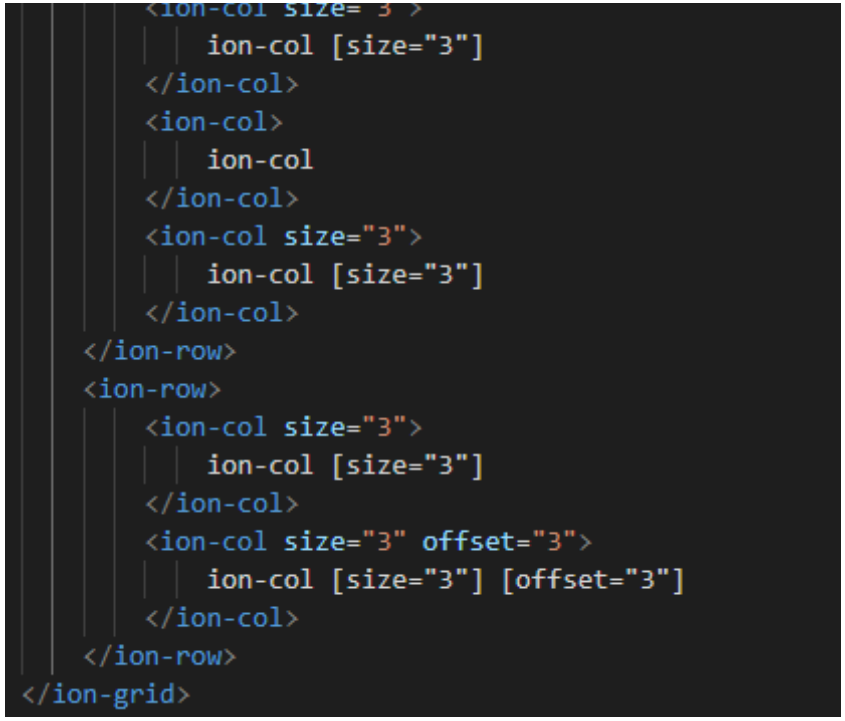
4.1.1- GRID.

Gracias a este componente se va a poner dar formato a las vistas de forma sencilla, pudiendo dividir el interior del componente en filas y columnas de forma parecida a como se hace en otros frameworks como Bootstrap. Al igual que Bootstrap, Grid está basado en flex. Cada una de las filas se divide en 12 unidades que se pueden dividir entre las diferentes columnas.

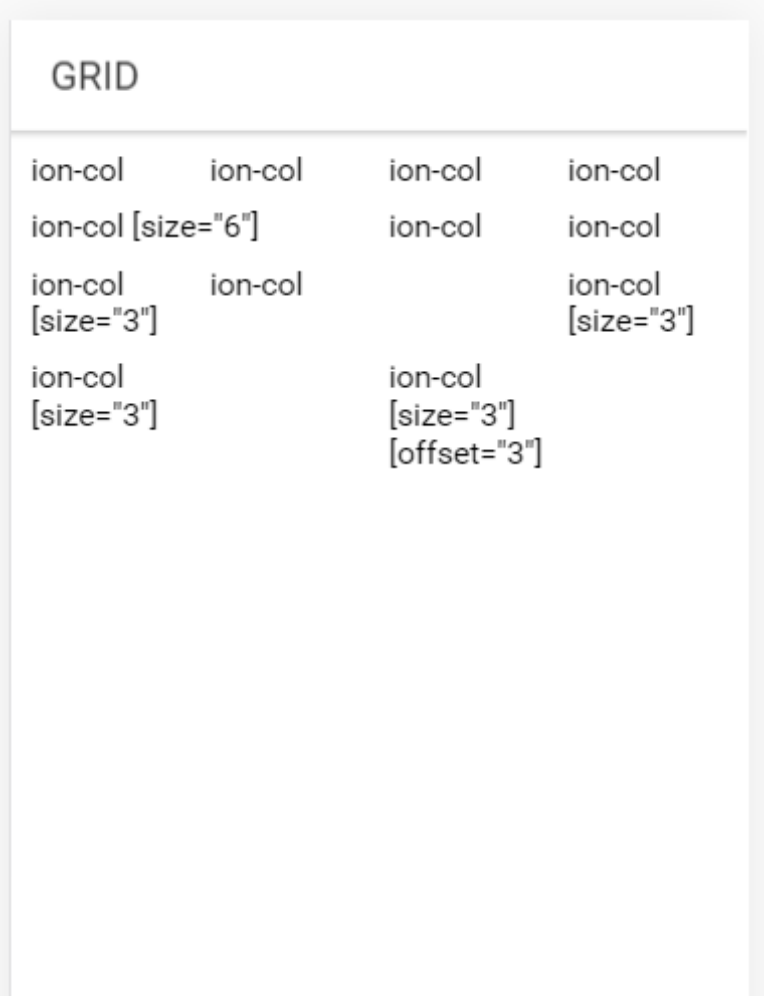
Las etiquetas necesarias para utilizar el componente son ion-grid, ion-row y ion-col y para especificar el ancho que debe tener cada columna ion-col cuenta con un atributo, en caso de no indicar el ancho deseado se divide la fila entre las columnas. También existe la posibilidad de indicar el offset de las columnas. A continuación, se muestra un ejemplo:



```
<ion-grid>
    <ion-row>
        <ion-col>
            ion-col
        </ion-col>
        <ion-col>
            ion-col
        </ion-col>
        <ion-col>
            ion-col
        </ion-col>
        <ion-col>
            ion-col
        </ion-col>
    </ion-row>
    <ion-row>
        <ion-col size="6">
            ion-col [size="6"]
        </ion-col>
        <ion-col>
            ion-col
        </ion-col>
        <ion-col>
            ion-col
        </ion-col>
    </ion-row>
    <ion-row>
```



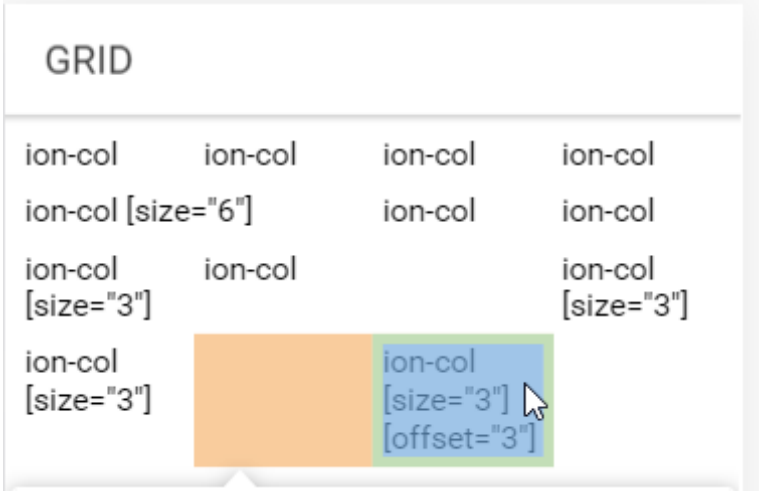
BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation



BIRT LH (Copyright (cita))

Obra derivada de captura de pantalla del programa Google Chrome, propiedad de Google

En la sigiuente imagen se puede observar el offset equivalente a 3/12. El offset corresponde con el margen.



BIRT LH (Copyright (cita))

Obra derivada de captura de pantalla del programa Google Chrome, propiedad de Google

4.1.2- ALERTS.

Las alertas permiten mostrar pequeños avisos a los usuarios, no se deben confundir con vistas modales, ya que ni ocupan toda la pantalla ni han sido diseñadas para realizar tareas “largas”. Entre los usos más habituales de estos componentes se encuentran los siguientes:

- Informar de un evento
- Hacer una consulta corta como los datos de login
- Pedir confirmación para una acción

Las alertas se crean directamente en los controladores. Para poder crear y mostrar una alerta es necesario disponer de un AlertControler, para ello lo inyectaremos en el contructor:

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

@Component({
  selector: 'app-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss'],
})
export class HomePage {

  constructor(public alertController: AlertController) {}

  async mostrarAlerta(){
      const alert = await this.alertController.create({
        cssClass: 'my-custom-class',
        header: 'Titulo alerta',
        subHeader: 'Subtitulo alerta',
        message: 'Texto de la alerta.',
        buttons: ['Confirmar']
      });

      await alert.present();
    }


}
```

BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

Una vez disponemos del AlertController crearemos la alerta, especificaremos el título, añadiremos las entradas necearías y los botones. Si fuera necesario obtener los datos introducidos por el usuario en la alerta se indicará la gestión al añadir los botones.

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

@Component({
  selector: 'app-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss'],
})
export class HomePage {

  constructor(public alertController: AlertController) {}

  async mostrarAlerta(){
      const alert = await this.alertController.create({
        cssClass: 'my-custom-class',
        header: 'Titulo alerta',
        subHeader: 'Subtitulo alerta',
        message: 'Texto de la alerta.',
        buttons: [
          {
            text:'Confirmar',
            handler: () =>{
              //realizo las tareas oportunas
              console.log("He confirmado la peticion");
            }
          },
          {
            text:'Cancelar',
            handler: () =>{
              //realizo las tareas oportunas
              console.log("He cancelado la peticion");
            }
          }
          ]
```

```
      });

      await alert.present();
    }


}
```

BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

4.1.3- BUTTONS.

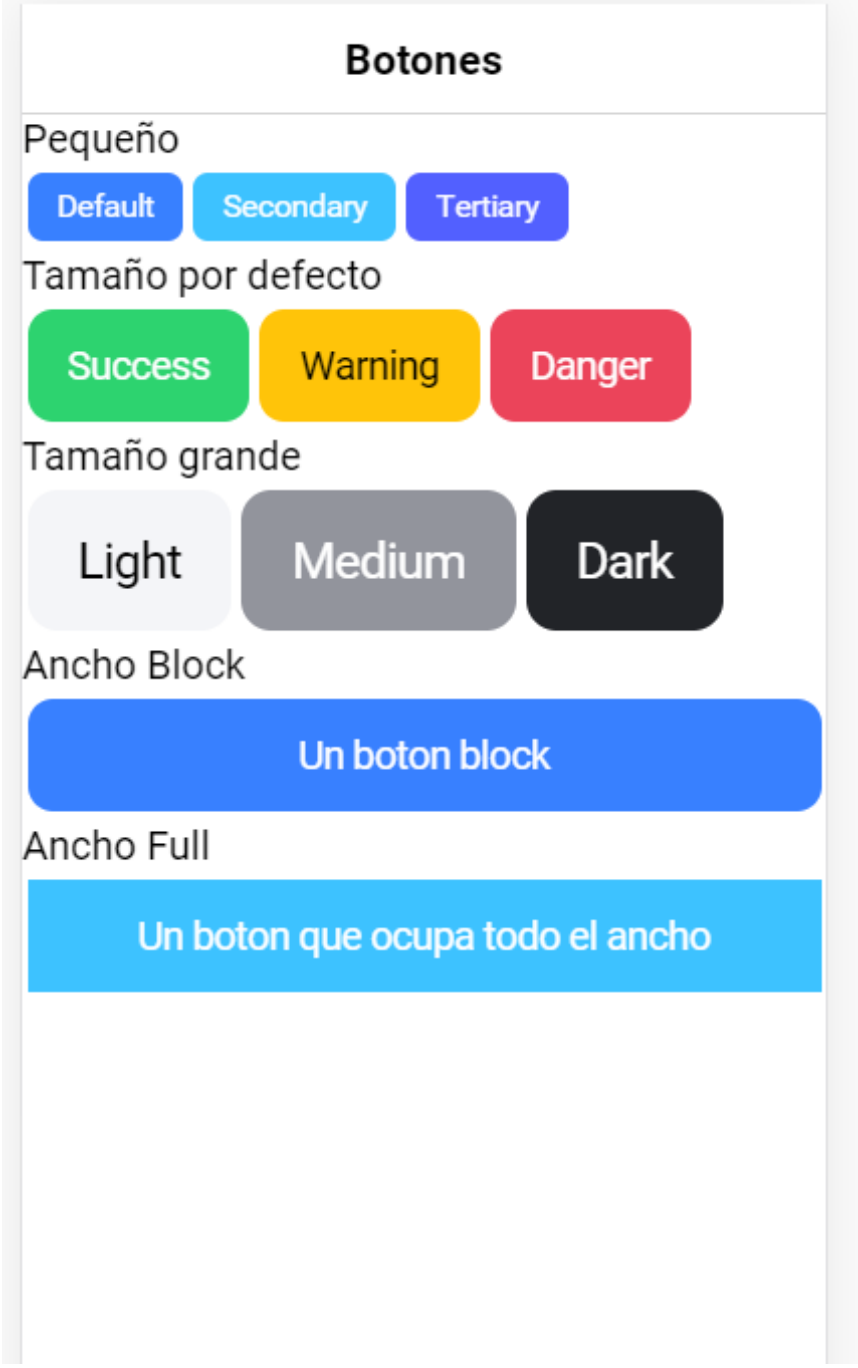
Ionic utilizar botones HTML. Aunque a través de la directiva ion-button se mejoran. Los botones nos a permitir interactuar con los usuarios. Los botones pueden consistir en texto y / o un icono, y pueden mejorarse con una amplia variedad de atributos que veremos a continuación:

- Color: permite indicar un color para el botón haciendo uso de los colores indicados en el archivo de estilos. Valores: primary, secondary, danger, light y dark. Si hicieran falta más colores se pueden añadir.
- Full: indica que el botón debe ocupar todo el ancho disponible
- Block: indica que el botón debe ocupar todo el ancho disponible, pero tendrá los bordes laterales del color del botón
- Clear: indica que el botón tendrá un fondo transparente
- Outline: indica que el botón tendrá el borde del color indicado en vez de tener el fondo del color indicado.

```
<ion-header [translucent]="true">
    <ion-toolbar>
        <ion-title>
            Botones
        </ion-title>
    </ion-toolbar>
```

```
</ion-header>

<ion-content [fullscreen]="true">
    <section>
        <header>Pequeño</header>
        <ion-button size="small">Default</ion-button>
        <ion-button size="small" color="secondary">Secondary</ion-button>
        <ion-button size="small" color="tertiary">Tertiary</ion-button>
    </section>
    <section>
        <header>Tamaño por defecto</header>
        <ion-button color="success">Success</ion-button>
        <ion-button color="warning">Warning</ion-button>
        <ion-button color="danger">Danger</ion-button>
    </section>
    <section>
        <header>Tamaño grande</header>
        <ion-button size="large" color="light">Light</ion-button>
        <ion-button size="large" color="medium">Medium</ion-button>
        <ion-button size="large" color="dark">Dark</ion-button>
    </section>
    <section>
        <header>Ancho Block</header>
        <ion-button expand="block">Un boton block</ion-button>
    </section>
    <section class="full-width">
        <header>Ancho Full</header>
        <ion-button expand="full" color="secondary">Un boton que ocupa todo el ancho</ion-button>
    </section>
</ion-content>
```

BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation



BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

4.1.4- INPUTS Y FORMULARIOS.

Los inputs son los componentes que utilizaremos para construir formularios. Los formularios son elementos necesarios para solicitar datos a los usuarios. En la mayoría de los casos los datos introducidos por los usuarios deben ser validados con expresiones regular para asegurarnos que los datos de entrada cumplen con las restricciones del dominio. Si un usuario debe introducir su nombre, no podrá introducir caracteres numéricos. A continuación se muestra un formulario con input:

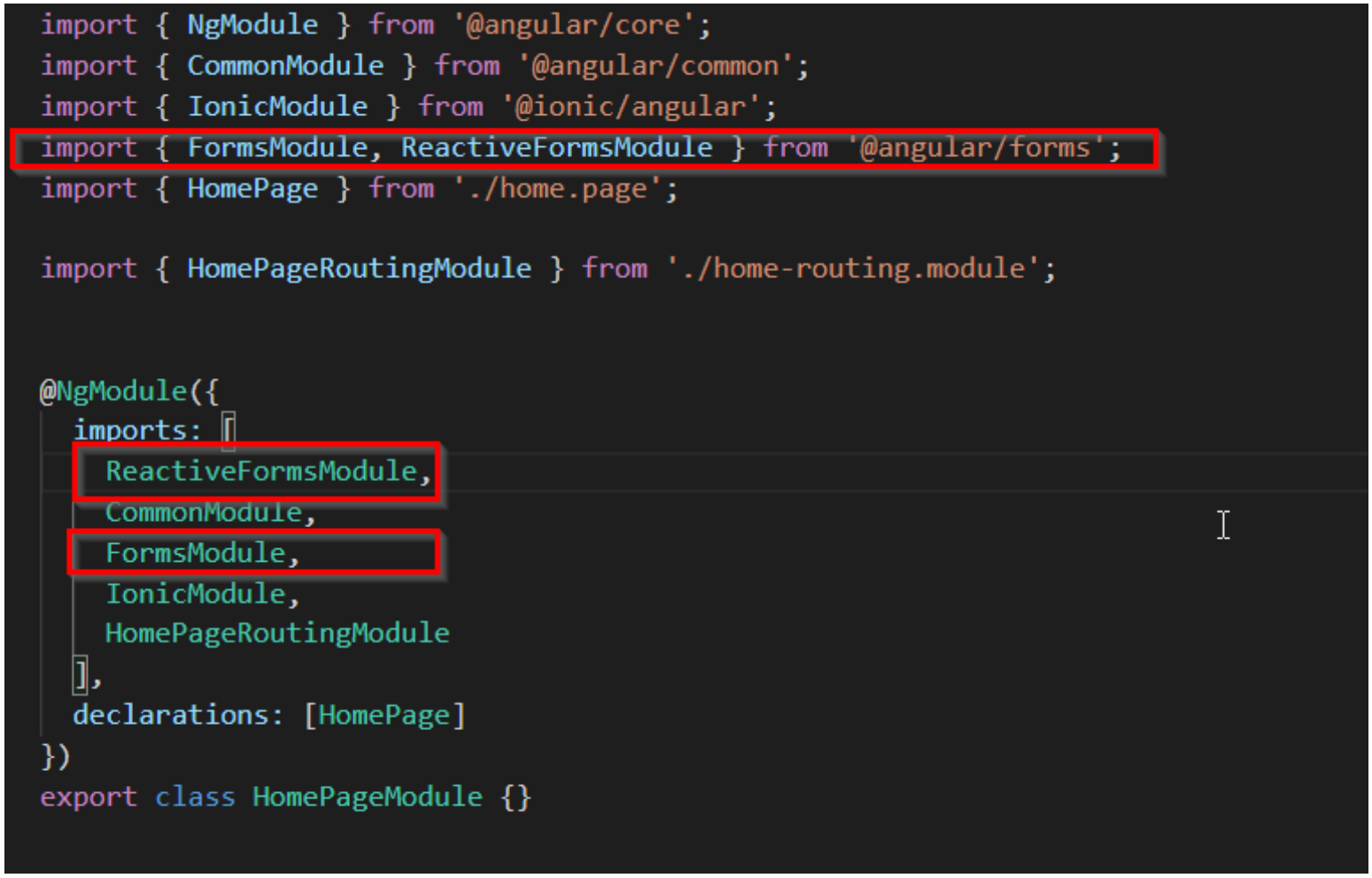
```
<ion-header [translucent]="true">
    <ion-toolbar>
        <ion-title>Formularios</ion-title>
    </ion-toolbar>
</ion-header>

<ion-content [fullscreen]="true">
    <form [formGroup]="formulario" (ngSubmit)="onSubmit(formulario.value)">
        <ion-item>
            <ion-label position="floating"> Nombre </ion-label>
            <ion-input formControlName="nombre" type="text"> </ion-input>
        </ion-item>
        <ion-item>
            <ion-label position="floating"> Apellido </ion-label>
            <ion-input formControlName="apellido" type="text"> </ion-input>
        </ion-item>
        <ion-item>
            <ion-label position="floating"> Edad </ion-label>
            <ion-input formControlName="edad" type="number"> </ion-input>
        </ion-item>
        <ion-button color="primary" class="submit-btn" expand="block" type="submit" [disabled]="!formulari
    </form>
</ion-content>
```

Como se puede ver en formulario los inputs se agrupan dentro ion-item y a cada input le acompaña una etiqueta, aunque se podría utilizar un placeholder en vez de la etiqueta. Es importante indicar el tipo del input para las validaciones. Los inputs pueden ser de alguno de los siguientes tipos: text, password, email, number, search, tel, y url. En caso de necesitar otro tipo de entrada como un chexkbox Ionic dispone de componentes o se puede utilizar elementos HTML. Para la validación se dispone del siguiente código en controlador de la vista:

```
import { Component } from '@angular/core';
import { FormBuilder, Validators } from '@angular/forms';


@Component({
  selector: 'app-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss'],
})
export class HomePage {

  formulario;
  constructor(public formBuiler:FormBuilder) {
    this.formulario = formBuiler.group({
      nombre:['', Validators.compose([Validators.maxLength(30),Validators.pattern(""),Validators.minLengt
      apellido:['', Validators.compose([Validators.maxLength(30),Validators.pattern(""),Validators.minLe
      edad:['']
    });
  }
  onSubmit(values){

    console.log(values);

  }


}
```

BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

Tambien será necesario modificar el archivo module.ts e incluir los siguientes imports:

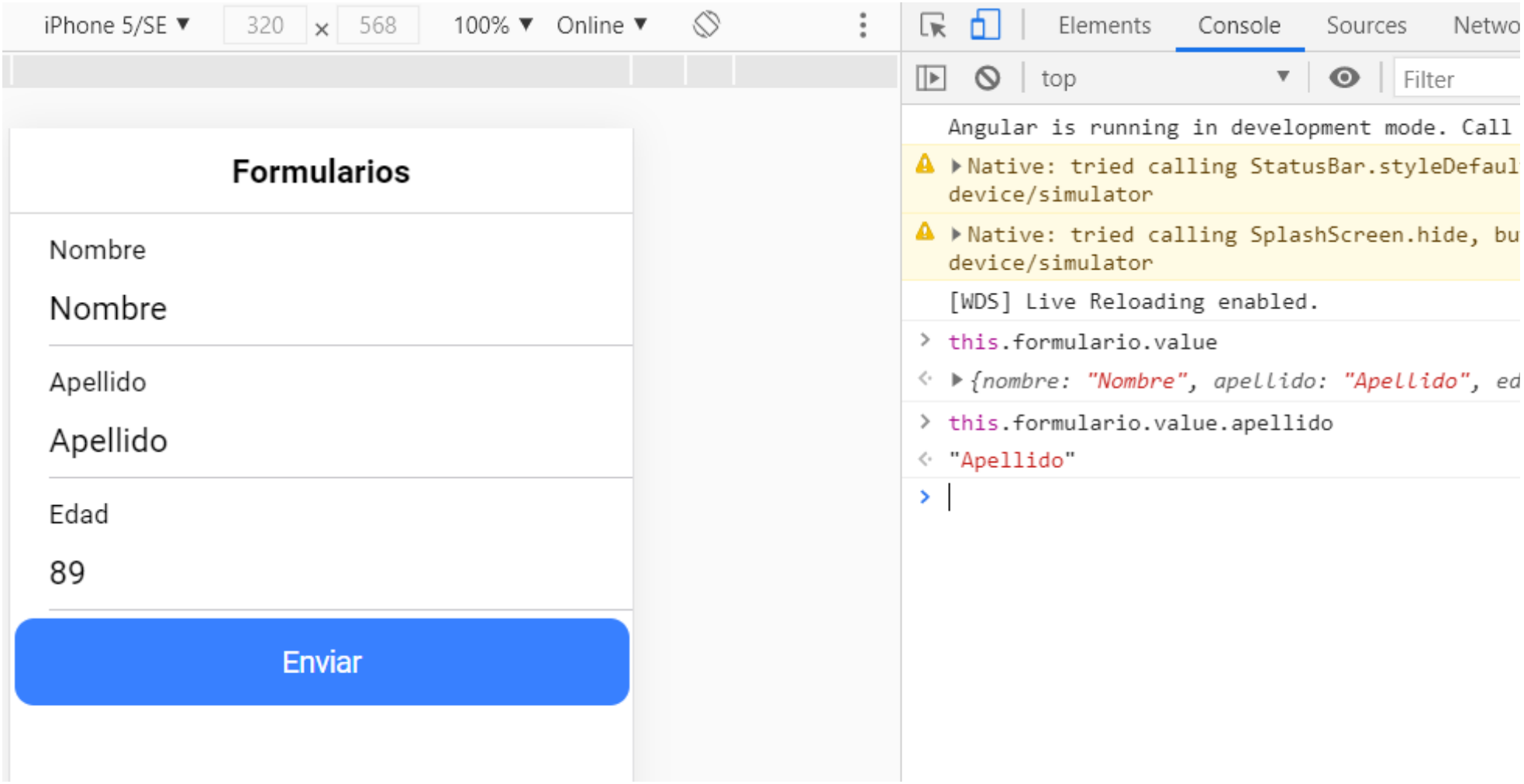


BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

Fijaros que es necesario inyectar un FormBuilder en el constructor y que se ha importado el paquete Validators en el controlador de la vista. Para poder acceder a los valores que el usuario a introducido en formulario debemos utilizar la variable formulario de alguna de las siguientes maneras:

- ```
  // toda la información del formulario
  this.formulario.value
  ```

- ```
  // acceso a un input concreto
  this.formulario.value.nombre
  ```



BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

Por lo que se refiere a los validadores, utilizaremos la función Validators.compose cuando queremos que se cumpla más de una condición para que las expresiones regulares sean más sencillas y fáciles de leer. Además, utilizaremos las siguientes funciones de Validators:

- pattern: para indicar una expresión regular

- maxLenght: para indicar longitud máxima
- minLenght: para indicar longitud mínima

4.1.5- LISTS.

Las listas son un elemento de interfaz ampliamente utilizado en casi cualquier aplicación móvil, y puede incluir contenido que va desde texto básico hasta botones, íconos y miniaturas. Tanto la lista, que contiene elementos, como los elementos de la lista pueden ser cualquier elemento HTML.

Existe la posibilidad de definir en la vista cada uno de los elementos de la lista, aunque lo más común será utilizar una lista para mostrar los elementos de un array.

Dentro de una lista podemos utilizar diferentes elementos como divisores, cabeceras, elementos con desplazamiento... En la siguiente imagen se muestra un ejemplo de una lista que hace uso de directica *ngFor para mostrar cada uno de los items que contiene la variable items del controlador de la vista.

```
<ion-list inset>
  <button ion-item *ngFor="let item of items" (click)="itemSelected(item)">
    {{ item }}
  </button>
</ion-list>
```

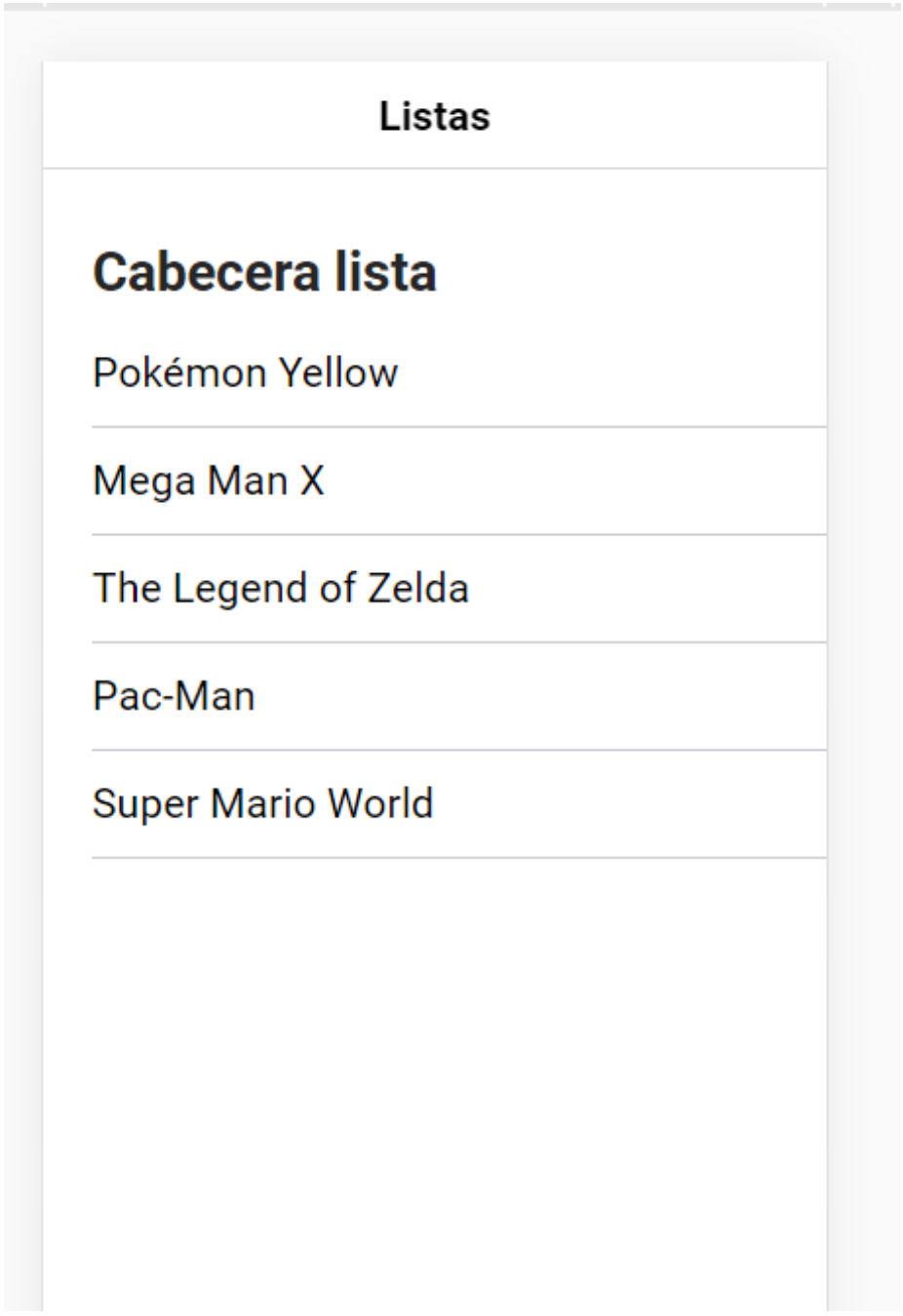
BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

En las siguientes imagenes se pueden observar una lista con una cabecera.

```
<ion-header [translucent]="true">
    <ion-toolbar>
        <ion-title>Listas</ion-title>
    </ion-toolbar>
</ion-header>
```

```
<ion-content [fullscreen]="true">
    <ion-list>
        <ion-list-header>
            <ion-label>Cabecera lista</ion-label>
        </ion-list-header>

        <ion-item>
            <ion-label>Pokémon Yellow</ion-label>
        </ion-item>
        <ion-item>
            <ion-label>Mega Man X</ion-label>
        </ion-item>
        <ion-item>
            <ion-label>The Legend of Zelda</ion-label>
        </ion-item>
        <ion-item>
            <ion-label>Pac-Man</ion-label>
        </ion-item>
        <ion-item>
            <ion-label>Super Mario World</ion-label>
        </ion-item>
    </ion-list>
</ion-content>
```

BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation



BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

4.1.6- FABS.

Los botones flotantes nos van a servir para posicionar botones sobre el contenido en una posición fija. La forma de estos botones es redonda y suele incluir un icono en su interior. Los elementos de este tipo tienen la siguiente estructura:

```
<ion-fab vertical="top" horizontal="end" slot="fixed">
    <ion-fab-button>
        <ion-icon name="add"></ion-icon>
    </ion-fab-button>
</ion-fab>
```

BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

Tal y como se puede observar, dentro del componente ion-fab se incluye un botón. Si se desea incluir un icono en vez de un texto dentro del botón flotante únicamente hay que cambiar el texto Button por un ion-icon como se muestra en la siguiente imagen:

```
<ion-fab vertical="bottom" horizontal="end" slot="fixed">
    <ion-fab-button>
        <ion-icon name="arrow-forward-circle"></ion-icon>
    </ion-fab-button>
</ion-fab>
```

BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

El listado completo de iconos disponibles puede consultarse en el siguiente enlace: https://ionicframework.com/docs/v3/ionicons/

4.1.7- LOADING.

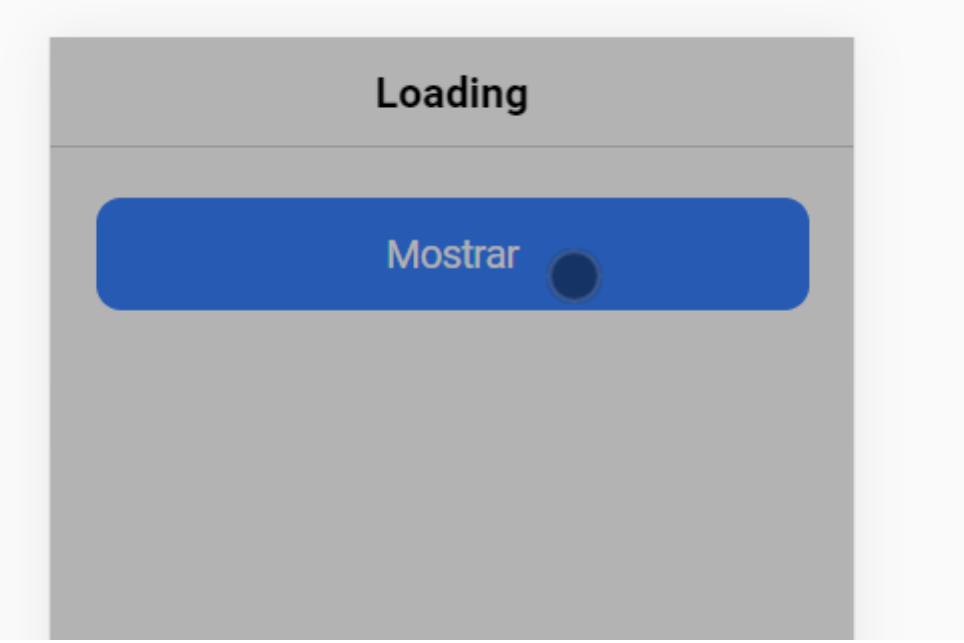
El componente Loading es una superposición que impide la interacción del usuario con el interfaz mientras indica que se está produciendo una actividad. De forma predeterminada, muestra una ruleta. La ruleta se puede ocultar o personalizar para usar varias opciones predefinidas. El indicador de carga se presenta sobre otro contenido incluso durante la navegación. Es necesario disponer de un LoadingController para poder crear, mostrar y eliminar estos componentes. En el ejemplo que se muestra a continuación se presenta un Loading, se espera 3 segundos y se elimina.

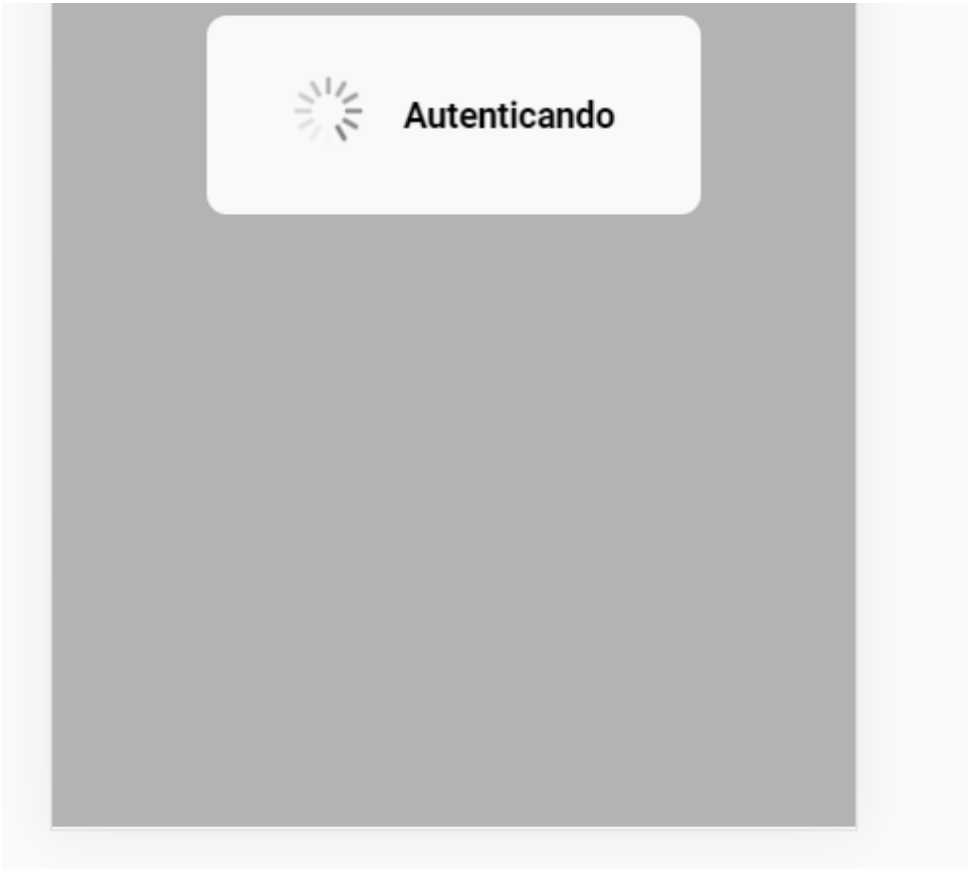
```
import { LoadingController } from '@ionic/angular';

@Component({
  selector: "app-home",
  templateUrl: "home.page.html",
  styleUrls: ["home.page.scss"],
})
export class HomePage {
  constructor(public loadingController:LoadingController) {}

  public async mostrarLoading(){
    let loading = await this.loadingController.create({
      message: 'Autenticando',
      duration: 3000
    })

    await loading.present();
  }
}
```

BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

```
<ion-header [translucent]="true">
    <ion-toolbar>
        <ion-title>Loading</ion-title>
    </ion-toolbar>
</ion-header>

<ion-content [fullscreen]="true">
    <ion-content fullscreen class="ion-padding">
        <ion-button expand="block" (click)="mostrarLoading()">Mostrar</ion-button>
    </ion-content>
</ion-content>
```

BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation





BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

4.1.8- MODALS.

Las modales sirven para mostrar interfaces temporales. A menudo se utilizan para iniciar sesión o modificar los valores de una variable que se quiere editar. Al igual que ocurre con las Alerts no permiten al usuario acceder al interfaz desde el que se generó la modal. Pero, al contrario de lo que ocurre con las Alerts, las modales ocupan toda la pantalla. Otra diferencia es que las modales sirven para mostrar vistas, por lo tanto, la vista debe estar creada antes de poder presentarla. Para trabajar con modales seguiremos los siguientes pasos:

Crear pagina que deseamos mostrar en la modal.

```
PS C:\Users\ulhi\pmdm2021\proyectoVacio> ionic g page modal
```

BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

En la vista de la modal añadiremos un botón para cerrar la modal

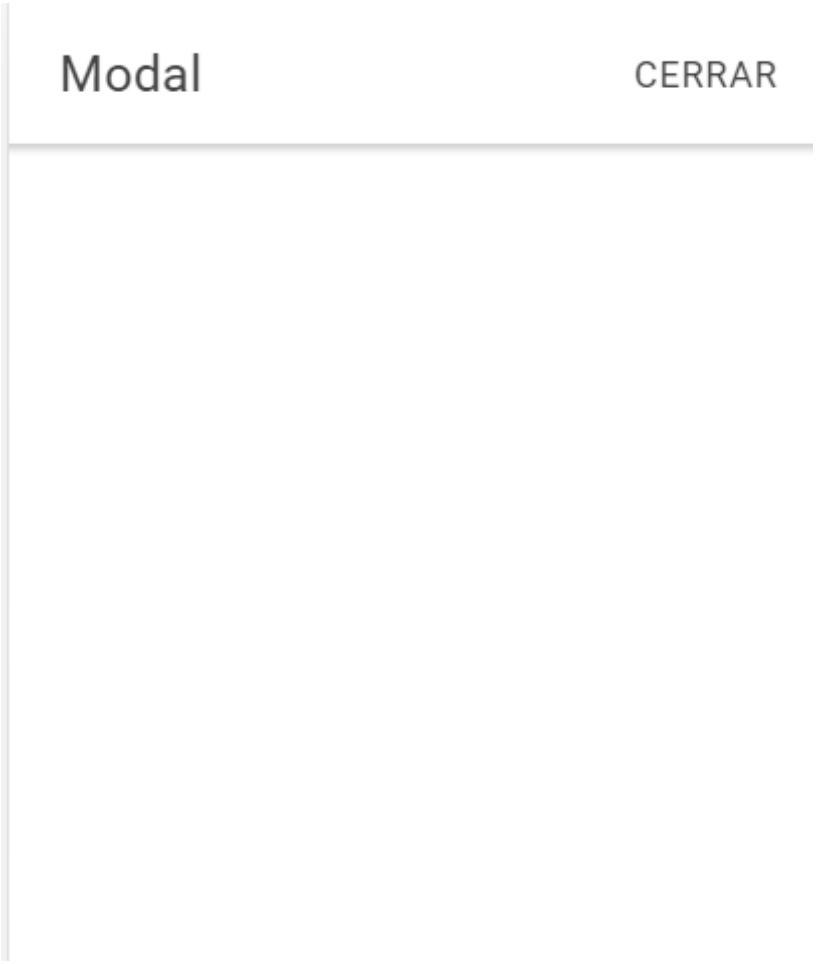
```
<ion-header>
  <ion-toolbar>
    <ion-title>Modal</ion-title>
    <ion-buttons slot="end">
      <ion-button (click)="dismiss()">Cerrar</ion-button>
    </ion-buttons>
  </ion-toolbar>
</ion-header>

<ion-content> </ion-content>
```

BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

En el controlador de la vista hay que añadir el método dismiss para cerrar la modal. Para acceder a los datos que nos han pasado hay que utilizar @Input(). Es necesario importar Input de @angular/core. Por último, si se desea retornar datos a la hora de llamar al método dismiss se pueden indicar.

```
import { Component, OnInit, Input } from "@angular/core";
import { ModalController } from "@ionic/angular";

@Component({
  selector: "app-modal",
  templateUrl: "./modal.page.html",
  styleUrls: ["./modal.page.scss"],
})
export class ModalPage implements OnInit {
  @Input() nombre: string;
  @Input() param: string;

  constructor(public modalCtrl: ModalController) {}

  ngOnInit() {}

  dismiss() {
    // using the injected ModalController this page
    // can "dismiss" itself and optionally pass back data
    this.modalCtrl.dismiss({
      out1: "datos devueltos",
      out2: "dato devuelto 2",
      dismissed: true,
    });
  }
}
```

BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation



BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

En controlador de la vista tenemos que inyectar en el constructor un ModalController, importar la vista que utilizaremos y crear el método que lanzará la modal. Si es necesario pasar algún parámetro a la modal hay que utilizar el atributo componentProps. Si es necesario obtener datos desde la modal lo haremos haciendo uso del método onWillDismiss().

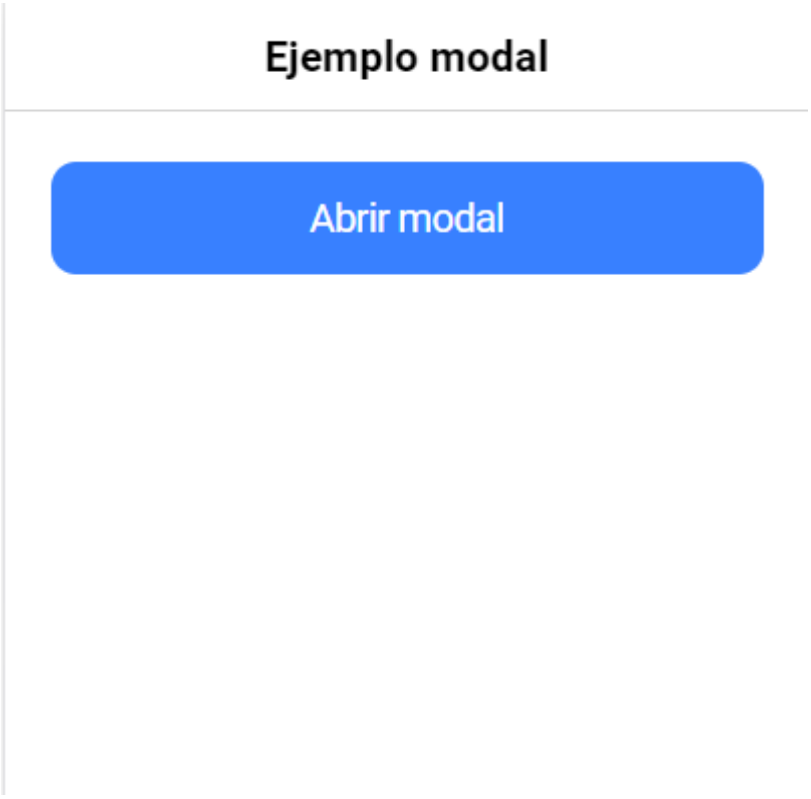
```
import { Component } from "@angular/core";
import { ModalController } from '@ionic/angular';
import { ModalPage } from "../modal/modal.page";
```

```
@Component({
  selector: "app-home",
  templateUrl: "home.page.html",
  styleUrls: ["home.page.scss"],
})
export class HomePage {

  param1:string;
  constructor(public modalCtrl: ModalController) {
    this.param1 = "Parametro";
  }

  public async mostrarModal(){
    const modal = await this.modalCtrl.create({
      component: ModalPage,
      componentProps: {
        'nombre': 'Nombre',
        'param': this.param1
      }
    });
    await modal.present();
    const {data} = await modal.onWillDismiss();
    console.log(data);

  }
  }
```

BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

```
<ion-header [translucent]="true">
    <ion-toolbar>
        <ion-title>Ejemplo modal</ion-title>
    </ion-toolbar>
</ion-header>

<ion-content [fullscreen]="true">
    <ion-content fullscreen class="ion-padding">
        <ion-button expand="block" (click)="mostrarModal()">Abrir modal</ion-button>
    </ion-content>
</ion-content>
```

BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation



BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

5- PROVIDERS O SERVICIOS.

Los servicios nos van a permitir desplazar cierto código que no es recomendable que esté dentro del controlador de las vistas. Pero antes de explicar el código que debería estar dentro de los proveedores, vamos a ver dos conceptos importantes para entender cómo funcionan los servicios:

1. Inyección de dependencias: los servicios se "inyectan por dependencias" en los constructores de los controladores que los van a utilizar. Por lo tanto, si se quiere utilizar un servicio únicamente se debe incluir en el constructor como si fuera un parámetro de entrada.
2. Singleton: los servicios solo se instancian la primera vez que se inyectan. Una vez que han sido instanciados, ya no se vuelven a instancias y todos los controladores utilizan la misma instancia. Esto hace que en los servicios se puedan almacenar datos para que sean compartidos por diferentes

controladores, pero también implica que el código que se encuentra en el constructor únicamente se va a ejecutar una vez.

Una vez que hemo visto las características de los servicios, vamos a ver que código debe estar en los servicios y no debe estar en los controladores. Como norma, todo código que no esté relacionado con la vista debería estar en un servicio. Es decir, la vista no tiene que saber cómo se realizan las acciones, solo debe saber que acciones se deben realizar. A continuación, se listan algunos ejemplos en los que el controlador no debería tener código de este tipo, ya que si en un futuro cambia la implementación habría que cambiar todos los controladores en vez de un único servicio:

- Peticiones http/https:
- Conexiones a Storage
- Cuando el mismo código se va a utilizar en diferentes páginas.
- Recuperación de datos que han sido recibidos con un id

Para definir y utilizar en servicio seguiremos los siguientes pasos:

1. Generar el servicio con el comando ionic generate provider nombreServicio
2. Importar el servicio e inyectarlos en el constructor del controlador que necesitu usar el servicio.

A continuación se muestra el controlador y la vista que hacen uso del servicio. En el ejemplo se enlaza el valor de la variable nombre del servicio:

```
import { Component } from "@angular/core";
import { ModalController } from "@ionic/angular";
import { ModalPage } from "../modal/modal.page";
import { ServicioService } from "../servicios/servicio.service";

@Component({
  selector: "app-home",
  templateUrl: "home.page.html",
  styleUrls: ["home.page.scss"],
})
export class HomePage {
  param1: string;
  constructor(
```

```
  constructor(
    public modalCtrl: ModalController,
    public servicio: ServicioService
  ) {
    this.param1 = "Parametro";
  }

  public async mostrarModal() {
    const modal = await this.modalCtrl.create({
      component: ModalPage,
      componentProps: {
        nombre: "Nombre",
        param: this.param1,
      },
    });
    await modal.present();
    const { data } = await modal.onWillDismiss();
    console.log(data);
  }
}
```

BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

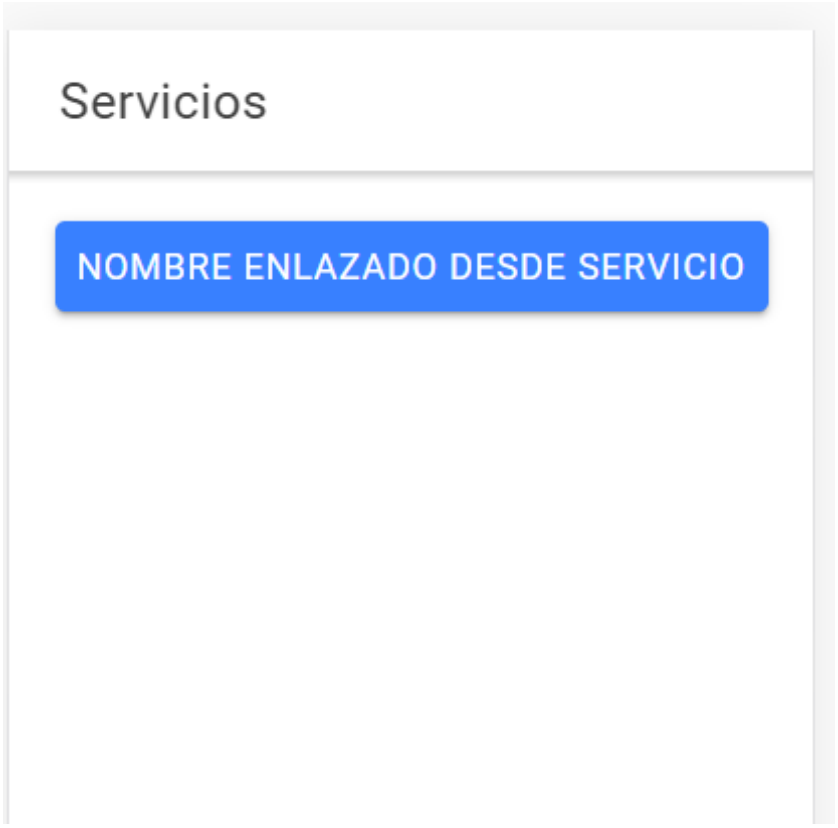
```
<ion-header [translucent]="true">
    <ion-toolbar>
        <ion-title>Servicios</ion-title>
    </ion-toolbar>
</ion-header>

<ion-content [fullscreen]="true">
    <ion-content fullscreen class="ion-padding">
        <ion-button expand="block">{{servicio.nombre}}</ion-button>
    </ion-content>
</ion-content>
```

BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

A continuación se muestra el controlador. Aunque en el ejemplo sólo se hace hace uso de la variable nombre. Desde el controlador de la vista se puede acceder a los métodos publicos del controlador.

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

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

  public nombre="Nombre enlazado desde servicio";
  constructor() {

  }
}
```

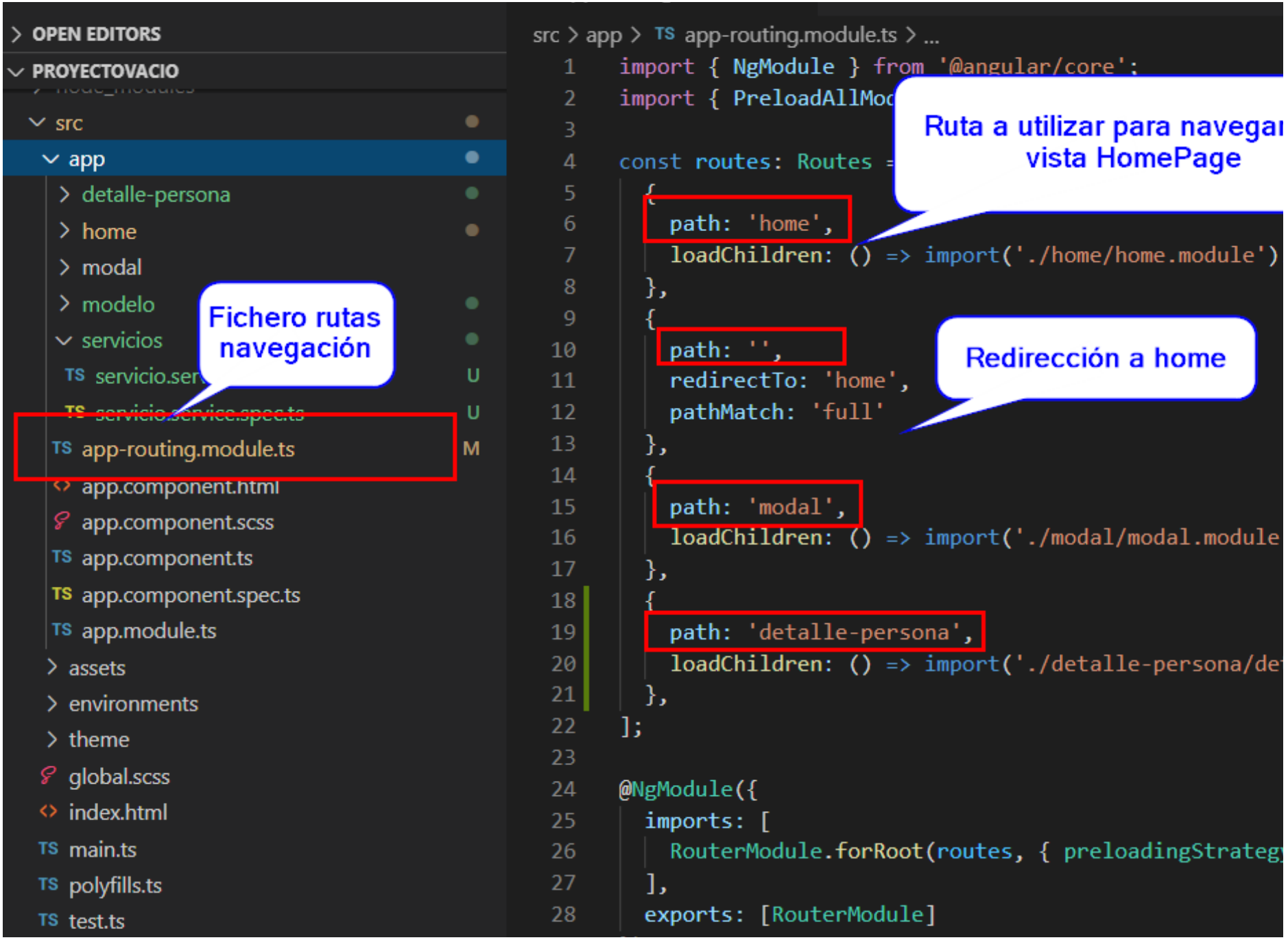
BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

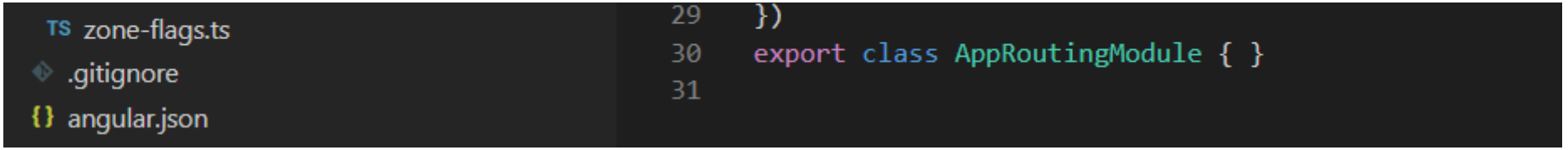


BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

5- NAVEGACIÓN ENTRE PÁGINAS.

Lo siguiente en la unidad será ver cómo realizar la navegación entre diferentes páginas de la aplicación. La navegación entre páginas a cambiado respecto a ve
rutas. Estas rutas se pueden consultar en el fichero app-routing.module.ts:

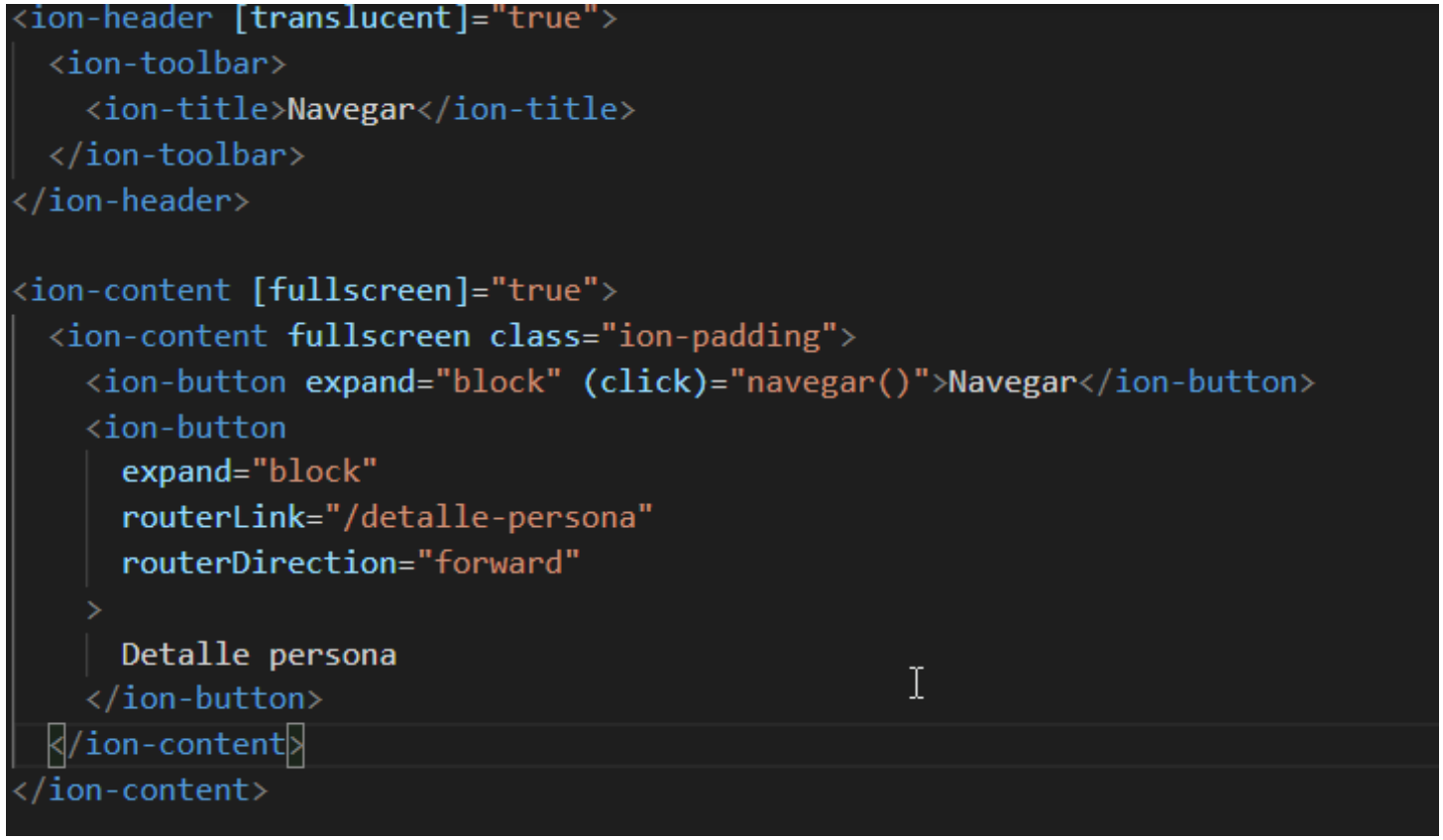




BIRT LH (Copyright (cita))

Obra derivada de captura de pantalla del programa Visual Studio Cod

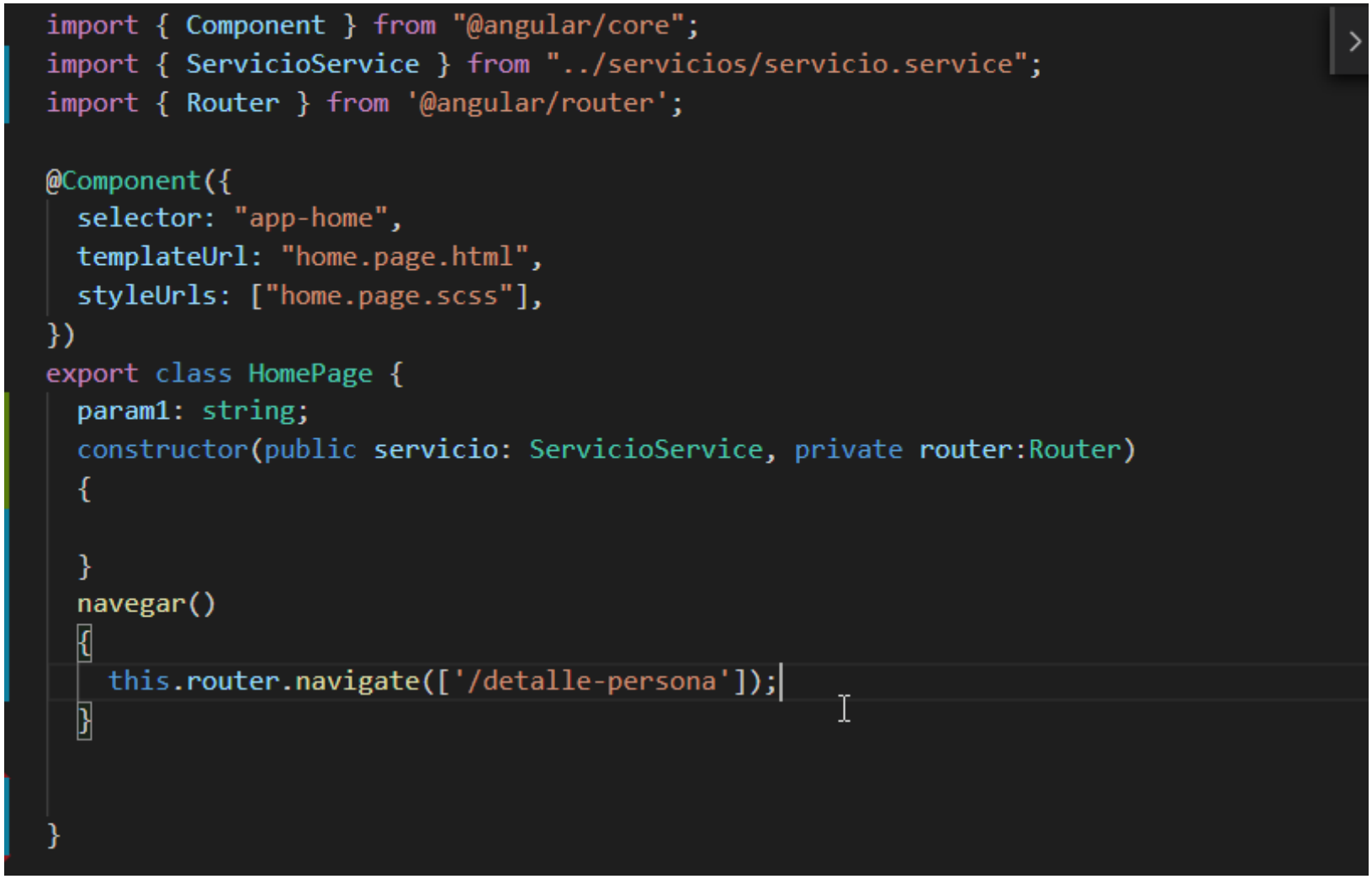
Tal y como se puede observar en el fichero existe un array con cada una de las rutas y una redirección a la ruta home. Este fichero no va a hacer falta modificarlo si las páginas se crean haciendo uso de la CLI y no se modifica su ruta. Por lo tanto, conviene no realizar modificaciones en las rutas de páginas para evitar problemas. La navegación puede llevarse a cabo en las vistas o en los controladores. En el segundo botón de la vista que se muestra a continuación se puede ver un ejemplo:



BIRT LH (Copyright (cita))

Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

En los controladores se hace haciendo uso de Router. Para utilizarlo hay que importarlo e inyectarlo en el contructor. Una vez realizado lo indicado, hay que hacer uso del método navigate:



```
import { Component } from "@angular/core";
import { ServicioService } from "../servicios/servicio.service";
import { Router } from '@angular/router';

@Component({
  selector: "app-home",
  templateUrl: "home.page.html",
  styleUrls: ["home.page.scss"],
})
export class HomePage {
  param1: string;
  constructor(public servicio: ServicioService, private router:Router)
  {

  }
  navegar()
  {
    this.router.navigate(['/detalle-persona']);
  }


}
```

BIRT LH (Copyright (cita))

Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

En versiones anteriores haciendo uso del controlador de navegación se pasaban como parámetros objetos. En la versión actual al utilizar rutas para la navegación l indicar el identificador de los objetos y a partir del identificador realizar una petición HTTP o realizar una búsqueda en un array. El array tendrá que ser una variable utilizaremos desde las diferentes páginas. En el fichero app-routing.module.ts indicaremos que se quiere hacer uso de un parámetro en la ruta de la siguiente man

```
import { NgModule } from '@angular/core';
import { PreloadAllModules, RouterModule, Routes } from '@angular/router';

const routes: Routes = [
  {
    path: 'home',
    loadChildren: () => import('./home/home.module').then( m => m.HomePageModule)
  },
  {
    path: '',
    redirectTo: 'home',
    pathMatch: 'full'
  },
  {
    path: 'modal',
    loadChildren: () => import('./modal/modal.module').then( m => m.ModalPageModule)
  },
  {
    path: 'detalle-persona/:myId',
    loadChildren: () => import('./detalle-persona/detalle-persona.module').then( m => m.DetallePerso
  },
];

@NgModule({
  imports: [
```

```
    RouterModule.forRoot(routes, { preloadingStrategy: PreloadAllModules })
  ],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

BIRT LH (Copyright (cita))

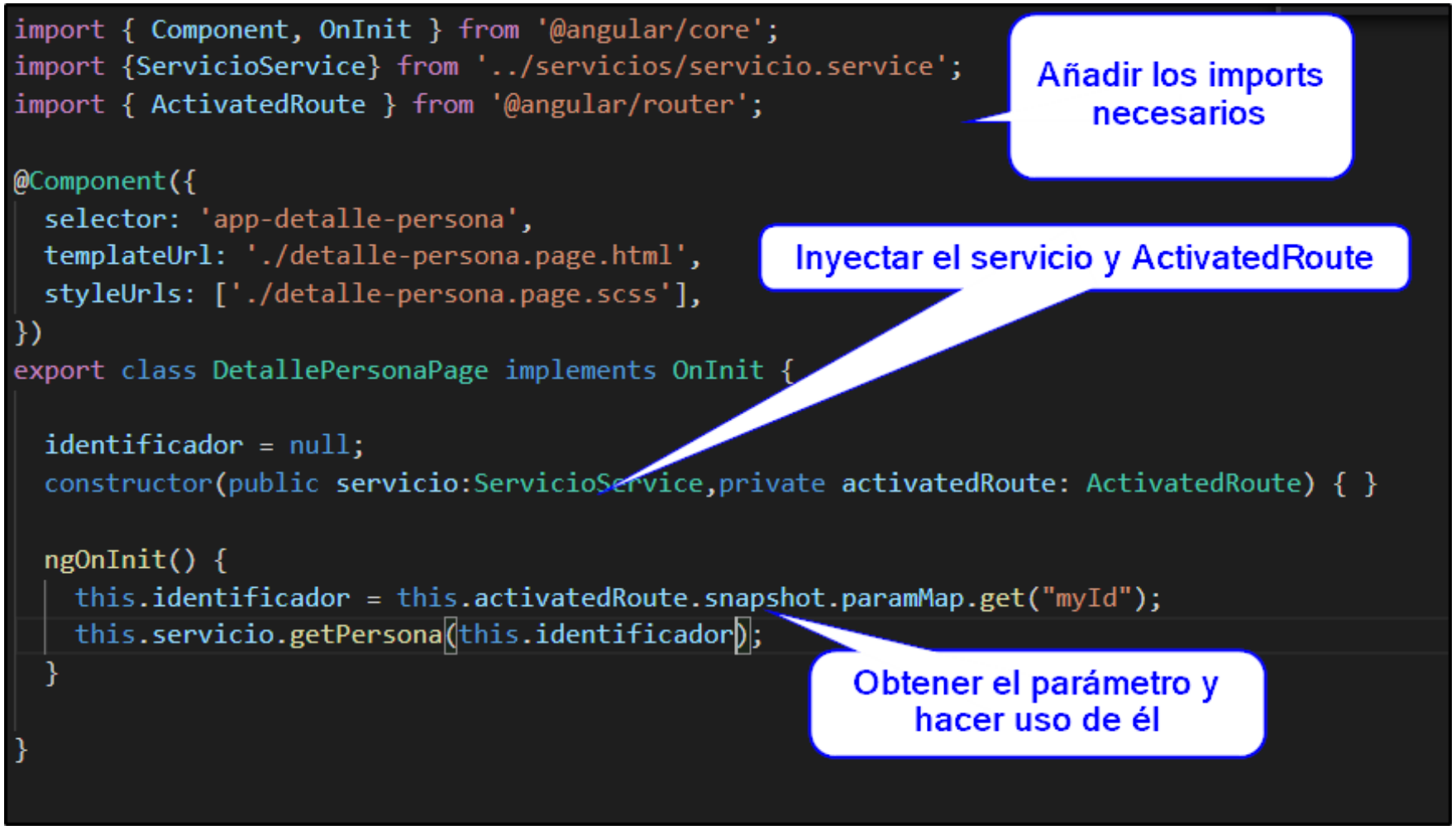
Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

En la vista o el controlador sólo tendremos que añadir el parámetro a la ruta:

BIRT LH (Copyright (cita))

Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

En el controlador de la página en la que queremos hacer uso del parámetro tendromos que acceder a el de la siguiente manera:



BIRT LH (Copyright (cita))

Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

Tal y como se puede ver en la captura los pasos a seguir son:

1. Importar el servicio que vamos a utilizar para obtener, buscar o estrablecer el objeto y ActivatedRoute

2. Inyectar en el constructor lo necesario
3. Obtener el parámetro y hacer uso de él

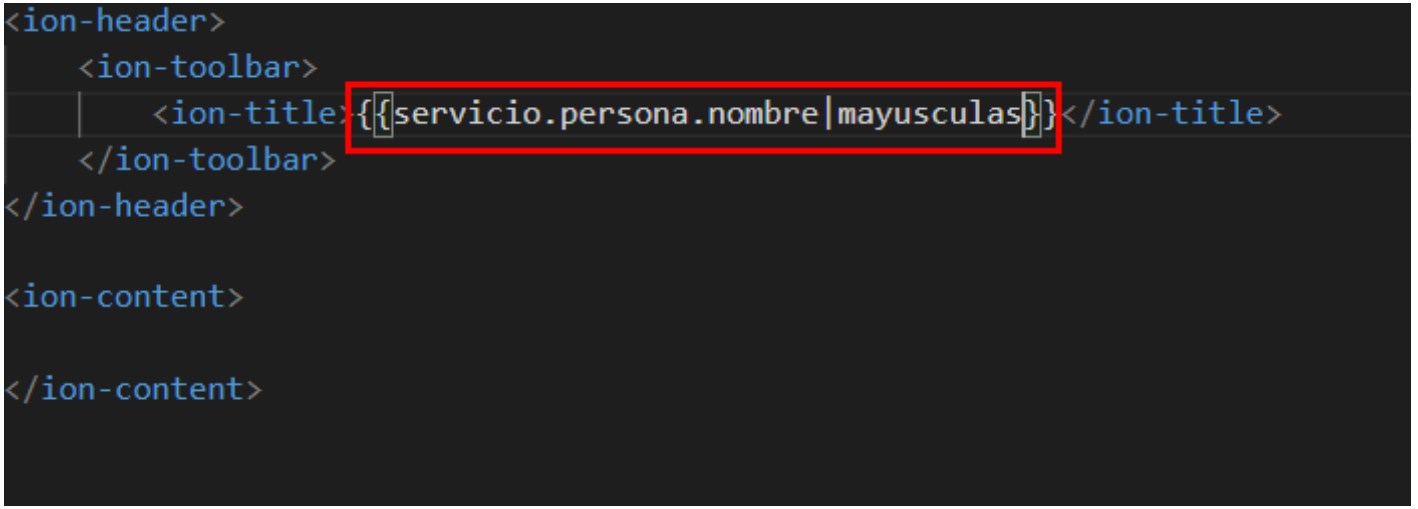
7- PIPES.

Los pipes son transformaciones que podemos hacer a cualquier dato desde nuestros templates, por ejemplo, mostrar un texto en mayúscula, mostrar valores de moneda, formatos de fecha etc. Un tipo de dato que es obvio que necesita ser transformado son los booleanos. Una vez que se ha generado el pipe, únicamente habrá que sobrescribir la función transform. Para utilizar un pipe dentro de una vista seguiremos los siguientes pasos:

1. Generar el pipe haciendo uso del comando ionic generate pipe nombrePipe
2. Incluir el Pipe en el fichero module.ts de la vista que se quiere utilizar
3. Hacer uso del Pipe en el vista

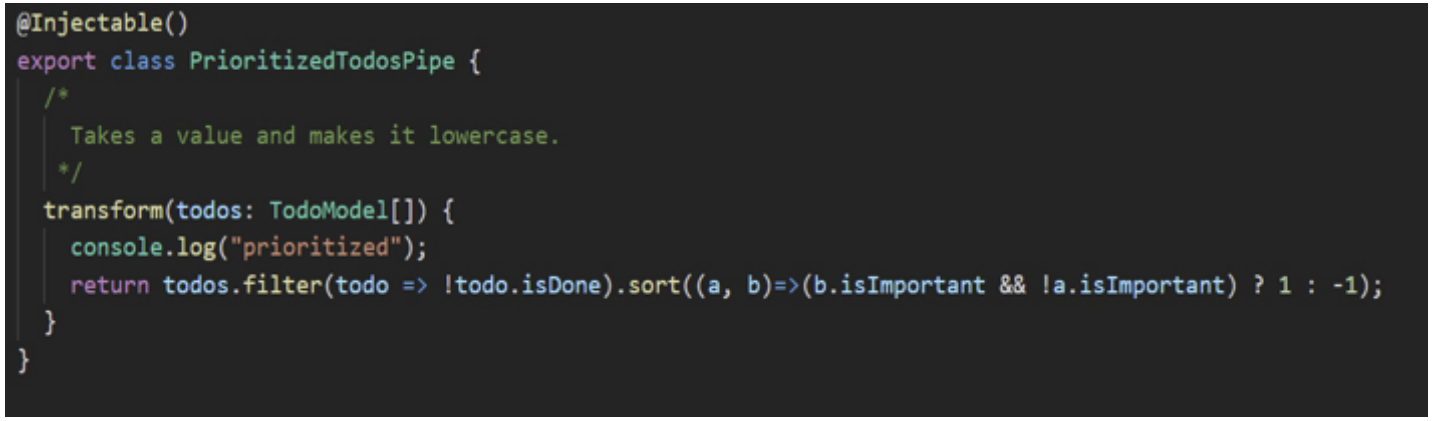
```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';

import { IonicModule } from '@ionic/angular';

import { DetallePersonaPageRoutingModule } from './detalle-persona-routing.module';

import { DetallePersonaPage } from './detalle-persona.page';

import {MayusculasPipe} from '../pipes/mayusculas.pipe';

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    IonicModule,
    DetallePersonaPageRoutingModule
  ],
  declarations: [DetallePersonaPage,MayusculasPipe]
})
export class DetallePersonaPageModule {}
```

BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation



BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation

El código de la imagen muestra un pipe que recibe un array y filtra los elementos que no han sido realizados y los ordena por prioridad.



BIRT LH (Copyright (cita)) Obra derivada de captura de pantalla del programa Visual Studio Code, propiedad de Microsoft Corporation