

Ionic con Angular



Juan Carlos Pérez Rodríguez

Sumario

Introducción.....	3
Instalaciones necesarias.....	5
Comenzando: Pequeña introducción apps con Angular.....	6
Estructura de un proyecto Angular.....	8
App ejemplo TablaMultiplicar.....	9
Introducción Ionic.....	20
Creando app muy básica en Ionic.....	21
App sencilla: Hacer una calculadora de suma, resta, multiplicación, división.....	28
[(ngModel)].....	30
Accediendo a atributos con: [].....	31
Conocimientos iniciales rutas Ionic.....	32
Navegar entre páginas.....	34
back button.....	34
Creando aplicación cliente api monedas.....	35
Servicios angular-ionic.....	38
Manejo de formularios.....	44
Consideraciones finales aplicación.....	46
Anexo: Trabajando con Router en Angular.....	49
Módulo de páginas y enlazarlo al router.....	54
Anexo: Error y subsanación ngModel el campo de formulario.....	57

Introducción

Angular (comúnmente llamado "Angular 2+" o "Angular 2"), 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. 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.

La biblioteca lee el HTML que contiene atributos de las etiquetas personalizadas adicionales, entonces obedece a las directivas de los atributos personalizados, y une las piezas de entrada o salida de la página a un modelo representado por las variables estándar de JavaScript. Los valores de las variables de JavaScript se pueden configurar manualmente, o ser recuperados de recursos JSON estáticos o dinámicos.

Angular es la evolución de AngularJS aunque incompatible con su predecesor.

Desarrollo del lado del cliente:

Con Angular te llevas al navegador mucha programación que antes estaba del lado del servidor, comenzando por el renderizado de las vistas. Esto hace que surjan nuevos problemas y desafíos. Uno de ellos es la sobrecarga en el navegador, haciendo que algunas aplicaciones sean lentas usando AngularJS como motor.

Por otra parte tenemos un impacto negativo en la primera visita, ya que se tiene que descargar todo el código de la aplicación (todas las páginas, todas las vistas, todas las rutas, componentes, etc), que puede llegar a tener un peso de megas. Tal problema será en la descarga inicial, una vez descargada ya irá convenientemente

Angular promueve el uso de **TypeScript** a sus desarrolladores. El propio framework está desarrollado en TypeScript, un lenguaje que agrega las posibilidades de ES6 y ES7, **además de un**

tipado estático y ayudas durante la escritura del código, el refactoring, etc. pero sin alejarte del propio Javascript (ya que el código de Javascript es código perfectamente válido en TypeScript).

Así pues se puede considerar Typescript como un superset de Javascript

Juan Carlos Pérez Rodríguez

Instalaciones necesarias

Chrome/ Chromium. Visual Studio Code. Postman. Git (recomendable no imprescindible)

Node, Android Studio, AngularCLI, ionic framework

Extensiones vscode: ionic snippets, typescript importer, json to ts (esta última necesita también instalar xclip: `sudo apt install xclip`)

Una vez instalado los programas base (navegador, vscode, postman, node)

Ejecutamos:

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

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

Comenzando: Pequeña introducción apps con Angular

Nos vamos a saltar toda la parte de instalación de NPM, Angular y demás, hay múltiples tutoriales al respecto en la web

Crear proyecto angular:

`ng new miproyecto`

nos preguntará por routing (de momento respondemos no)

le decimos que usaremos css

cuando termine entramos en la carpeta y ejecutamos para lanzar webpack:

`ng serve`

Lanzado `ng new`: se creará una carpeta igual que el nombre del proyecto indicado y dentro de ella se generarán una serie de subcarpetas y archivos que quizás por su número despisten a un desarrollador que se inicia en Angular

También se instalarán y se configurarán en el proyecto una gran cantidad de herramientas útiles para la etapa del desarrollo front-end. De hecho, gran cantidad de los directorios y archivos generados al crear un nuevo proyecto son necesarios para que estas herramientas funcionen. Entre otras cosas tendremos:

- Un servidor para servir el proyecto por HTTP
- Un sistema de live-reload, para que cuando cambiamos archivos de la aplicación se refresque el navegador
- Herramientas para testing
- Herramientas para despliegue del proyecto

Una vez creado el proyecto debemos entrar en la carpeta generada:

```
cd miProyectoAngular
```

Ya podemos iniciar el servidor de desarrollo (nos activa un servidor por defecto en el puerto 4200 de nuestro ordenador):

```
ng serve
```

Accedemos con un navegador en la url: <http://localhost:4200>

Estructura de un proyecto Angular

Si bien si miramos desde consola podemos ver la estructura de la carpeta del proyecto, vamos a iniciarnos en el editor recomendado (VisualStudioCode) y ver desde allí la estructura

Abrimos visualstudiocode y abrimos la carpeta (File → Open folder):

Ahí seleccionamos la carpeta del proyecto que hemos creado

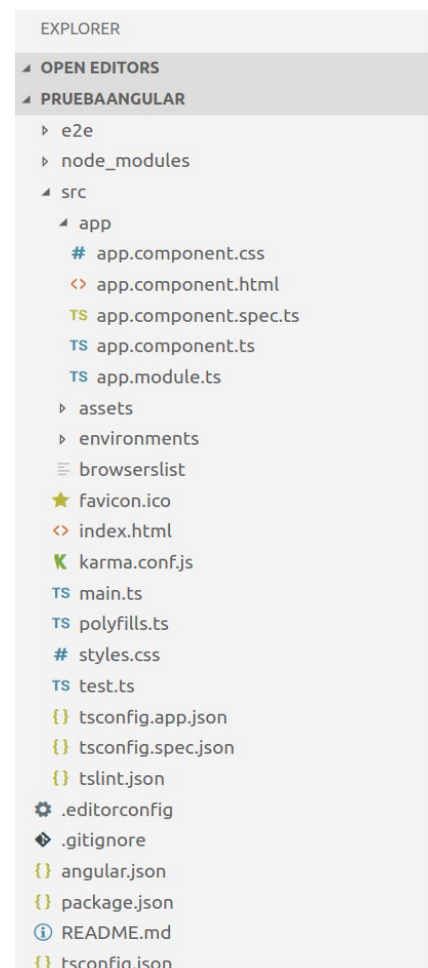
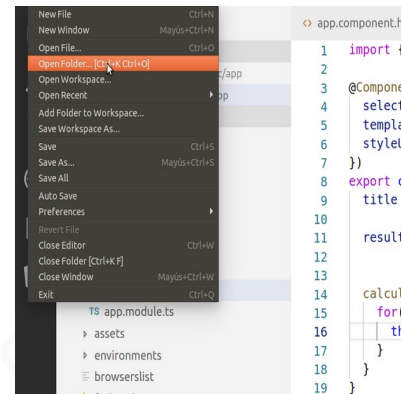
La estructura que nos mostrará visualstudio para un proyecto llamado “pruebaangular”:

Observar que ya mostramos desplegada la carpeta **src** y dentro de ella **app** ya que es la zona donde debemos concentrarnos a la hora de codificar

Observamos que dentro de la carpeta app tenemos ficheros de estilos (.css), páginas web (.html) y typescript (.ts)

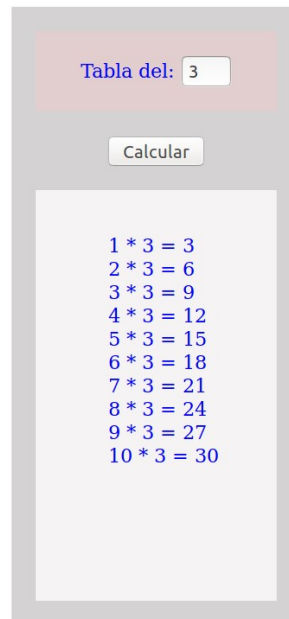
Las vistas de nuestra aplicación las obtendremos con los ficheros html y los css

Modelo y controlador los haremos con ficheros typescript



App ejemplo TablaMultiplicar

Como muchas veces la mejor forma de entender es por medio de un ejemplo vamos a realizar una aplicación muy sencilla:



Vamos a ir por pasos. Lo primero observar como ponemos información de nuestros typescript en las vistas.

Una forma para hacer eso es por medio de las llaves: `{{ }}`

Podemos acceder a los atributos creados en nuestro fichero typescript `app.component.ts`:

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

```
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css']  
})
```

```
export class AppComponent {  
  title = 'pruebaAngular';
```

```
  otroAtributo = "este texto se mostrará al llamar a {{otroAtributo}}";
```

En el fichero app.component.html podríamos poner:

```
<!--The content below is only a placeholder and can be replaced.-->
<p>
  {{otroAtributo}}
</p>
```

La salida que veremos en el navegador web:



Dicho eso en una primera versión podemos hacer un programa que vuelque en pantalla la tabla del 2. Para ello en lugar de: {{otroAtributo}} en el html pondremos: {{resultado }} y crearemos un método llamado: calcularTabla() en nuestro fichero typescript:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  resultado: String = "";

  calcularTabla():void{
    for(let i:number=1;i<11;i++){
      this.resultado += i + " * " + 2 + " = " + (i*2);
    }
  }
}
```

Aprovechamos para hablar un poco de typescript:

observar:

```
resultado: String = "";
```

con esa línea estamos creando un atributo llamado resultado que es de tipo String y se ha inicializado a comillas vacías. Como podemos ver la forma de definir un tipo de variable es mediante el símbolo dos puntos: ":" y poniendo el nombre del tipo después del nombre de la variable

observar:

```
for(let i:number=1;i<11;i++)
```

No difiere mucho de un for de C o Java. La diferencia principal es en la definición e inicialización de la variable: `i`. La instrucción: `let` nos permite definir una variable local con ámbito las llaves del bloque que estemos (en este caso el for)

Como antes usamos los dos puntos y luego ponemos el tipo de datos (en este caso number)

Observar que definimos el tipo de datos que devuelve el método de nuevo mediante el símbolo dos puntos. En este caso estamos diciendo que no devuelve nada:

```
calcularTabla():void{
```

Finalmente observar que cuando queremos usar un atributo dentro de un método tenemos que usar la palabra clave `this`:

```
this.resultado += i + " * " + 2 + " = " + (i*2);
```

Para que el código anterior haga algo en pantalla hace falta que le pongamos un nuevo evento que sea escuchable al botón del html. Pondremos el siguiente código en nuestra página app.component.html:

```
<button (click)="calcularTabla()">
  Calcular
</button>
```

la parte de Angular del código anterior es: **(click)** con esa instrucción le estamos diciendo que queremos ponerle un listener a la etiqueta html en la que estemos y encerrado entre comillas pondremos el nombre del método typescript que queremos llamar. En este caso calcularTabla()

Con lo anterior ya tendremos una aplicación que nos devuelve la tabla del 2 al pulsar en el botón. La siguiente fase será tomar la información que nos introduzca el usuario y así poder generar una tabla del número que nos soliciten.

El fichero app.component.html nos quedará:

```
<!--The content below is only a placeholder and can be replaced.-->
```

```
<div class="principal">
  <div class="introTabla">
    <label for="tabla">Tabla del: </label>
    <input type="text" #tabla value="0" />
  </div>

  <button (click)="calcularTabla(tabla.value)">
    Calcular
  </button>

  <div class="resultados">
    {{resultado}}
  </div>
</div>
```

Los div y las clases son únicamente por razones de CSS que nos saltaremos al no ser lo que corresponde estudiar ahora

Respecto a la versión anterior es especialmente importante fijarse en la almohadilla: “#”

```
<input type="text" #tabla value="0" />
```

El input anterior no difiere de un input cualquiera html salvo por: “#tabla” con eso lo que estamos haciendo es establecer una etiqueta angular para identificar ese input. Cuando queramos hacer referencia a ese input usaremos la etiqueta “tabla”

Como debemos saber ya en el atributo **value** se guarda el dato que el usuario haya introducido en el input Así pues si queremos tomar ese dato lo haremos con el nombre de la etiqueta que hemos puesto y accediendo a ese atributo: **tabla.value**

Eso es lo que podemos observar en la acción del click:

```
<button (click)="calcularTabla(tabla.value)">  
  Calcular  
</button>
```

Así pues le estamos pasando el dato que haya introducido el usuario al método calcularTabla()

La nueva versión del método typescript será:

```
calcularTabla(strtabla: string): void {  
  const tabla = Number(strtabla);  
  for(let i=1;i<11;i++){  
    this.resultado += i + ' * ' + tabla + ' = ' + (i*tabla);  
  }  
}
```

Observar que al igual que hemos hecho en las veces anteriores definimos el tipo de datos del parámetro mediante los dos puntos:

```
calcularTabla(strtabla: string): void{
```

Estamos diciendo que recibimos un atributo llamado tabla que es de tipo string. En las siguientes líneas tomamos una variable tabla que convierte el dato recibido como string en un número y luego en el bucle generamos la tabla para guardarlo en el atributo: this.resultado

Con lo anterior ya tenemos un código que nos hace lo que queremos. Ahora bien, no nos da la estética que quizás deseáramos al dejarnos la tabla toda en una única línea. ¿ cómo lo solucionamos ?

Al volcar directamente el resultado no saldría como queremos por no interpretar el “\n” como salto de línea en html. La alternativa de insertar
 tampoco nos funcionará por si sola.

InnerHTML permite en javascript introducir en el lugar del elemento actual. Con cierta semejanza la propiedad [innerHTML] en Angular nos permite introducir html que será interpretado:

```
<div class="principal">
  <div class="introTabla">
    <label for="tabla">Tabla del: </label>
    <input type="text" #tabla value="0" />
  </div>

  <button (click)="calcularTabla(tabla.value)">
    Calcular
  </button>

  <div class="resultados" [innerHTML]="resultado">

  </div>
</div>
```

Con la instrucción anterior ahora interpretará una salida html así que únicamente tenemos que poner en nuestro método la etiqueta


```

calcularTabla(tabla:number):void{
  for(let i:number=1;i<11;i++){
    this.resultado += i + " * " + tabla + " = " + (i*tabla) + "<br>";
  }
}

```

Pero vamos a usar otra de las opciones que nos da Angular para conseguir ese salto de línea. En este caso lo que haremos es rellenar una lista de forma dinámica con los datos que tengamos en un array de nuestro typescript

Vemos que se ha creado un atributo llamado array que es un array de string y como en el método vamos agregando elementos al array en cada iteración del formateada

```

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  resultado: String = "";
  array: Array<String>;

  calcularTabla(tabla:number):void{
    this.array = [];
    for(let i:number=1;i<11;i++){
      this.resultado += i + " * " + tabla + " = " + (i*tabla) + "<br>";
      this.array.push(i + " * " + tabla + " = " + (i*tabla));
    }
  }
}

```

En nuestro fichero html vamos a cambiar la zona de salida de resultados:

```

<div class="resultados">
  <ul>
    <li *ngFor="let item of array" >
      {{item}}
    </li>
  </ul>
</div>

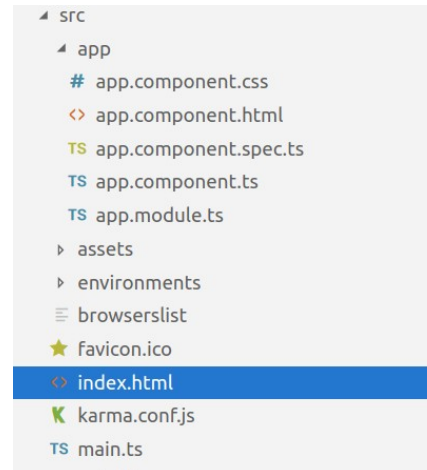
```

lo importante de lo anterior es el ngFor :

```
<li *ngFor="let item of array" >
```

vemos que hacemos un `foreach` y vamos guardando en cada iteración en la variable `item` el dato correspondiente del array. Lo bueno de `ngFor` es que nos irá generando un nuevo elemento `` en cada iteración.

Vamos a fijarnos en el fichero: `index.html` que está dentro de la carpeta `src`:



Estamos acostumbrados en desarrollo web que la página de inicio de un sitio sea cosas similares a: `index.html`, `index.php`,....

Pues bien, esta también sería nuestra página de inicio de la aplicación. Veamos su contenido:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>PruebaAngular</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```


Podemos observar que no tiene nada de especial respecto a una página web normal, salvo que al iniciar body tenemos dentro: `<app-root></app-root>` Esa etiqueta no es propia de html es una etiqueta que hemos generado mediante Angular.

Angular usa el concepto de los “componentes” imaginemos que creara un elemento nuevo (un componente) y que pudiéramos llamarlo e interpretarlo desde el navegador con una etiqueta que le hayamos puesto.

¿ Y qué es ese “componente” `<app-root>` ? Vamos a ver de nuevo nuestro fichero Typescript `app.component.ts`

Lo primero que nos fijaremos es que el nombre de nuestro fichero dice: `app . component` Podemos presuponer que ese fichero pretende ser el gestor del componente principal de la aplicación

El contenido del fichero:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  resultado: String = "";
  array: Array<String>;

  calcularTabla(tabla:number):void{
    this.array = [];
    for(let i:number=1;i<11;i++){
      this.resultado += i + " * " + tabla + " = " + (i*tabla) + "<br>";
    }
  }
}
```

Vemos que tenemos una **anotación `@Component`** Angular hace uso de las anotaciones para definir sus objetos. Esta anotación la ponemos a la Clase: `AppComponent` que tenemos declarada y desarrollada en el fichero `app.component.ts`

Así en el caso actual que estamos creando un componente le establecemos una etiqueta para usarlo:

```
selector: 'app-root'
```

una plantilla html (observar que apunta al fichero donde hemos la página de entrada-visionado de nuestra página):

```
templateUrl: './app.component.html'
```

y los ficheros de estilos:

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

Este potencial de crear nuevos componentes nos permite crear diferentes objetos-componentes y poder llamarlos con su selector asociado en nuestra página html. Teniendo una modularización bastante buena de nuestra aplicación.

Juan Carlos Pérez Rodríguez

Introducción Ionic

Ionic es un framework que da herramientas y servicios para desarrollar aplicaciones móviles híbridas usando tecnologías Web como: CSS, HTML5, Javascript. Las aplicaciones se construyen con esas tecnologías web y luego son distribuidas a través de aplicaciones nativas en las tiendas de aplicaciones. Ionic hace uso de Apache Cordova para obtener acceso a los recursos del sistema operativo tales como la cámara, gps, etc

¿ qué es Apache Cordova ?

Primero tenemos que entender que IOS, Android, etc incluyen un componente llamado: **WebView**. Este componente permite renderizar una web. Así si nosotros desarrollamos una aplicación web podemos “incrustarla” en el webview y el código javascript que hemos realizado podrá ejecutarse indistintamente en todos esos sistemas operativos

Apache Cordova permite encapsular CSS, HTML, y código de Javascript dependiendo de la plataforma del dispositivo. Extiende las características de HTML y JavaScript para trabajar con el dispositivo. Las aplicaciones resultantes son híbridas, lo que significa que no son ni una aplicación móvil nativa (porque toda la representación gráfica se realiza vía vistas de Web en vez del framework nativo) ni puramente basadas en web (porque no son solo aplicaciones web, sino que están empaquetadas como aplicaciones para su distribución y tienen acceso a las APIs nativas del dispositivo). **Apache Cordova incrusta el código HTML5 dentro de un WebView nativo** en el dispositivo, utilizando una interfaz de función foránea para acceder los recursos nativos de del dispositivo

Continuando con Ionic se ha construido usando Angular. Así si sabemos trabajar en Angular podremos hacer aplicaciones Ionic muy fácilmente

Creando app muy básica en Ionic

Desde la ubicación donde queramos crear la app:

```
ionic start nombreapp
```

En este ejemplo usamos angular como el motor de ionic al crear la aplicación: y que nos genere una aplicación blank (sin ningún template de renderizado)

```
carlos@cyc-principal:~/Escritorio/AprenderIonic$ ionic start tablamultiplicar
```

Pick a framework! 🤖

Please select the JavaScript framework to use for your new app. To bypass this prompt next time, supply a value for the `--type` option.

? Framework: Angular

Let's pick the perfect starter template! 🍌

Starter templates are ready-to-go Ionic apps that come packed with everything you need to build your app. To bypass this prompt next time, supply `template`, the second argument to `ionic start`.

? Starter template:

tabs	A starting project with a simple tabbed interface
sidemenu	A starting project with a side menu with navigation in the content area
> blank	A blank starter project
list	A starting project with a list
my-first-app	An example application that builds a camera with gallery
conference	A kitchen-sink application that shows off all Ionic has to offer

```
cd nombreapp
ionic serve
```

A partir de ahora cualquier cambio que hagamos tendrá reflejo en la página web que se nos abre

Abrimos con visual studio code la carpeta: nombreapp. En el ejemplo es: tablamultiplicar

La estructura de nuestra aplicación la podemos ver toda dentro de src así en: `nombreapp/src/app` tenemos las diferentes páginas de la aplicación

Así si se hubiera creado una app con tabs observamos que nos ha creado las pages: tab1, tab2, tab3

Veamos ahora una página, por ejemplo home. Cada página tiene ya creada la parte de vista:

`.html`

`.scss`

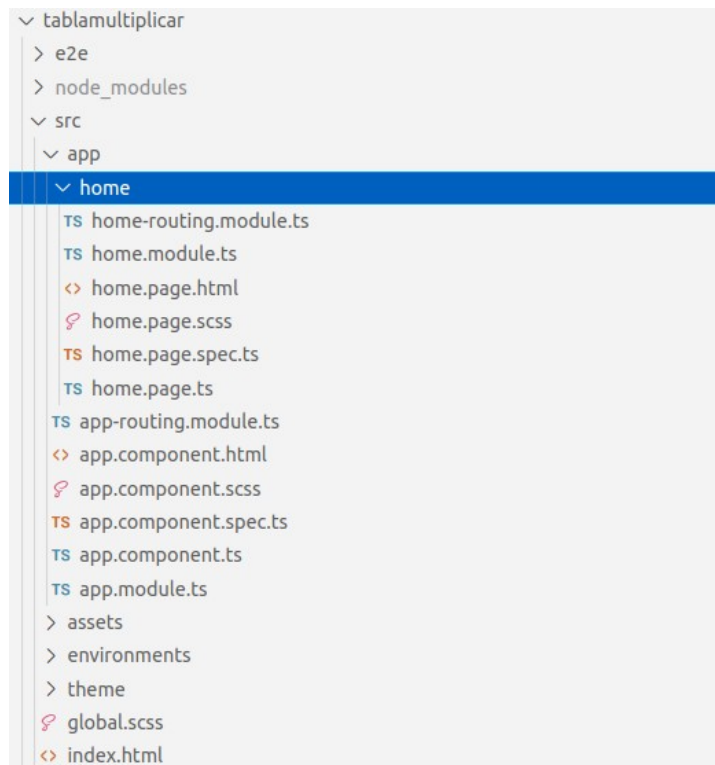
controlador: `.ts`

Observamos que aparece: `home.page.html`, `home.page.scss`, `home.page.ts`

Como antes dijimos la vista la conforman el html y los estilos css (en este caso se usa sass que es un preprocesador de css). Mientras que la parte de controlador está en `home.ts` La extensión: `ts` hace mención a TypeScript, que es un lenguaje que sirve de “superconjunto” a javascript. Esto significa que se puede codificar en javascript y funcionará. Sin embargo si se hiciera así se desaprovecharían las ventajas que permite (definir tipos de datos a las variables etc). El compilador de typescript se encargará de convertir a javascript y así poder ser interpretado sin problemas por cualquier navegador

Para arrancar el servicio hacemos:

ionic serve



Vamos a echar un vistazo a `home.page.html`:

```
<> home.page.html x
tablamultiplicar > src > app > home > <> home.page.html > ion-header
1 <ion-header [translucent]="true">
2   <ion-toolbar>
3     <ion-title>
4       Blank
5     </ion-title>
6   </ion-toolbar>
7 </ion-header>
8
9 <ion-content [fullscreen]="true">
10   <ion-header collapse="condense">
11     <ion-toolbar>
12       <ion-title size="large">Blank</ion-title>
13     </ion-toolbar>
14   </ion-header>
15
16   <div id="container">
17     <strong>Ready to create an app?</strong>
18     <p>Start with Ionic <a target="_blank" rel="noopener noreferrer" href="https://ionic
19   </div>
20 </ion-content>
21
```

Ionic se puede basar en Angular, ese framework permite crear tus propios componentes y llamarlos como a cualesquier otra etiqueta html. Así pues, podemos observar que aparece: `ion-header`, `ion-navbar`, `ion-title`, `ion-content`

Todas esas etiquetas han tenido que ser creadas en Angular y forman parte del conjunto de elementos que tenemos disponibles en Ionic

Por otro lado vemos que aparecen otras etiquetas html estandar que conviven con las anteriores. Ejemplo de ellas son: `h2`, `p`

Vamos a sustituir lo anterior de la siguiente forma:

```
<ion-header [translucent]="true">
  <ion-toolbar>
    <ion-title>
      Tabla de multiplicar
    </ion-title>
  </ion-toolbar>
</ion-header>

<ion-content [fullscreen]="true">
  <ion-item>
    <ion-label fixed>
      Tabla:
    </ion-label>
    <ion-input type="number" [(ngModel)]="inputtabla" ></ion-input>
    <button ion-button color="light" (click)="calcular()">Calcular </button>
  </ion-item>
  <ion-textarea autoGrow=true value="{{resultado}}"></ion-textarea>
</ion-content>
```

El fichero typescript: home.page.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss'],
})
export class HomePage {
  resultado: string;
  inputtabla: string;
  constructor() {}
  calcular() {
    this.resultado = "";
    let tabla = Number(this.inputtabla);
    for(let i:number = 1; i<11; i++){
      this.resultado += tabla + " * " + i + " = " + (tabla * i) + "\n";
    }
  }
}
```


Observaremos que una vez guardemos tiene efecto inmediato en la página web al estar arrancado ionic serve

Ahora queremos crear un espacio para que el usuario nos introduzca un número. Eso es algo que podemos conseguir simplemente introduciendo un `<input>` html pero vamos a apoyarnos en un componente Ionic. Cuando queramos ver/buscar/aprender a usar un componente ionic podemos dirigirnos a: ionicframework.com/docs iremos a components e ir viendo lo que podemos usar. Buscaremos por Inputs. Elegiremos de las opciones que allí tenemos. Aquí observamos fixed inline labels:

```
<ion-item>
<ion-label fixed>
Tabla:
</ion-label>
<ion-input type="number" [(ngModel)]="inputtabla" ></ion-input>
<button ion-button color="light" (click)="calcular()">Calcular </button>
</ion-item>
```

Observamos `ion-item`, en la documentación nos dice que un item puede contener texto, imagenes, y cualquier otra cosa. Que habitualmente suele encontrarse en una lista junto a otros item. Este elemento hace fácil la interacción, como arrastrar para editar, arrastrar, editar, eliminar,... No es obligatorio usarlo dentro de una lista. Y el ejemplo anterior es una muestra de ello

`ion-label` se ubica dentro de ion-item para dar una descripción del elemento que queremos usar. En este caso será para un input

`ion-input` nos servirá para introducir un elemento de texto

La opción: `fixed` dentro de `ion-label` nos permite poner la etiqueta a la izquierda del input.

Un elemento habitual para enviar información por parte del usuario son los botones. Buscamos en ionicframework.com/docs

```
<button ion-button color="light" (click)="calcular()">Calcular </button>
```

donde el atributo color nos permite establecer color al botón

Hemos ya podido observar en lo que hemos hecho hasta ahora cosas de Angular. Los componentes Angular creados por Ionic: ion-input, ion-item, etc y la forma de establecer un listener para eventos mediante: (click) como vimos antes

Para visualizar el resultado vamos a usar un ion-textarea:

```
<ion-textarea autoGrow=true value="{{resultado}}"></ion-textarea>
```

Ya tenemos algo similar a antes pero con etiquetas-componentes Ionic. Ponemos el código Typescript en el fichero: home.ts y ya la aplicación nos funcionará apropiadamente

Observamos el uso ngmodel para vincular un control con un atributo del typescript. De tal forma que las modificaciones en cualquiera de los lados repercute inmediatamente en el otro (cambios en el input html aparecen en el objeto typescript y viceversa)

```
<ion-input type="number" [(ngModel)]="inputtabla" ></ion-input>
```

El atributo: inputtabla que hemos declarado en typescript queda vinculado con el elemento ion-input y cualquier modificación tiene lugar

Así es como aparece el “*banana in a box*” que es una combinación de ambos símbolos:
[(ngModel)]

que nos permite hacer algo bidireccional (la misma información que registremos en el html se estará registrando en el typescript)

Podemos pues, mediante `ngModel` conseguir tener tantos atributos como queramos tener vinculados entre `typescript` y `html`

Juan Carlos Pérez Rodríguez

App sencilla: Hacer una calculadora de suma, resta, multiplicación, división

Crearemos 3 atributos en el typescript: operador, numero1, numero2

los tres los vincularemos respectivamente a 3 input

Finalmente el método calcular() toma los datos de los 3 y calcula la operación (puede usarse un switch con el atributo de tipo string: operador y distinguir:

```
switch(operacion){
```

```
  case "+": this.resultado = this.numero1 + this.numero2 ;
```

```
  ...
```

```
}
```

El html: home.page.html:

```
<ion-header [translucent]="true">
```

```
<ion-toolbar>
```

```
<ion-title>
```

```
Calculadora
```

```
</ion-title>
```

```
</ion-toolbar>
```

```
</ion-header>
```

```
<ion-content [fullscreen]="true">
```

```
<ion-item>
```

```
<ion-label fixed>
```

```
Número 1:
```

```
</ion-label>
```

```
<ion-input type="number" [(ngModel)]="numero1" ></ion-input>
```

```
</ion-item>
```

```
<ion-item>
```

```
<ion-label fixed>
```

```
Operador:
```

```
</ion-label>
```

```
<ion-input type="text" [(ngModel)]="operador" ></ion-input>
```

```
</ion-item>
```

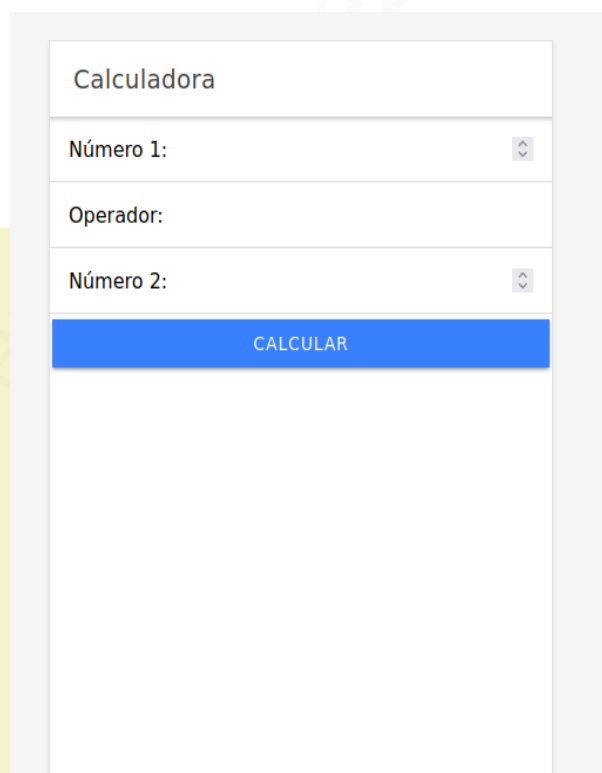
```
<ion-item>
```

```
<ion-label fixed>
```

```
Número 2:
```

```
</ion-label>
```

```
<ion-input type="number" [(ngModel)]="numero2" ></ion-input>
```



```

</ion-item>
<ion-button expand="full" (click)="calcular()">Calcular</ion-button>

<ion-textarea autoGrow=true value="{{resultado}}"></ion-textarea>
</ion-content>

```

ahora home.page.ts:

```

import { Component } from '@angular/core';
@Component({
  selector: 'app-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss'],
})
export class HomePage {
  resultado: string;
  numero1: number;
  numero2: number;
  operador: string;
  constructor() {}
  calcular() {
    this.resultado = "";
    switch(this.operador){
      case "+": this.resultado = "" + (this.numero1 + this.numero2);break;
      case "-": this.resultado = "" + (this.numero1 - this.numero2);break;
      case "*": this.resultado = "" + (this.numero1 * this.numero2);break;
      case "/": this.resultado = "" + (this.numero1 / this.numero2);break;
    }
  }
}

```

[(ngModel)]

Básicamente ngmodel se trata de un enlace, entre algo que tienes en la definición del componente con un campo de formulario del template (vista) del componente

Nota: "model" en ngModel viene de lo que se conoce como el modelo en el MVC. El modelo trabaja con los datos, así que podemos entender que el enlace creado con ngModel permite que desde la vista pueda usar un dato. Sin embargo, con expresiones ya se podía usar un dato, volcando su valor en la vista como {{dato}}. La diferencia es que al usar ese dato en campos de formulario, debes aplicar el valor mediante la directiva ngModel.

Por ejemplo, tenemos un componente llamado "cuadrado", que declara una propiedad llamada "lado". La class del componente podría quedar así:

```
export class CuadradoComponent {  
  lado = 4;  
}
```

Si queremos que ese valor "lado" se vuelque dentro de un campo INPUT de formulario, tendríamos que usar algo como esto.

```
<input type="number" [(ngModel)]="lado">
```

Estamos usando la directiva ngModel como si fuera una propiedad del campo INPUT, asignándole el valor que tenemos declarado en el componente.

Accediendo a atributos con: []

Otras propiedades de Angular/Ionic

ya vimos los eventos que disparábamos mediante los paréntesis: `(click)`. Y también vimos la interpolación: `{{resultado}}` que es útil para introducir información de texto del controlador en la vista. El otro elemento de Angular que utilizaremos es la propiedad: `[]`

Con la propiedad: `[]` podemos tocar propiedades de la vista y asignarles información que puede no ser string proveniente del controlador. En el ejemplo anterior hemos visto que el atributo resultado del controlador ahora no será tomado como texto sino será interpretado por la propiedad `innerHTML` obteniendo la salida deseada.

Otro ejemplo sería dar un color u otro a la letra para un párrafo en el html:

```
<p [style.color]="variableBoolean ? 'red' : 'green'">texto a veces verde, a veces rojo</p>
```

Ahora habría que crear `variableBoolean` como un atributo boolean de la clase `HomePage`

Como podemos ver a la propiedad `style.color` no se le pasa una string sino una expresión lógica que se tiene que interpretar-valorar

Conocimientos iniciales rutas Ionic

Creamos una nueva aplicación sin plantilla:

```
ionic start componentes blank
cd componentes && ionic serve
```

Para hacer las cosas desde cero, borramos la carpeta home dentro de src/app

Ahora creamos una nueva página inicio:

```
ionic g page pages/inicio
```

Lo anterior nos crea un montón de cosas... El módulo subrouting para esa carpeta pages, etc

Pero hay que eliminar los rastros del home que hemos borrado. Al ir a **src/app/app-routing.module.ts** vemos que nos marca como mal el path a home (ya no existe) y nos ha creado el path a inicio al haber creado la página inicio antes

borramos la ruta a home que ya no existe (las dos referencias dentro del const routes) quedando únicamente el path a inicio en esa constante

Ya nos lo ha dejado casi todo funcional. Falta verificar que funciona. Si no nos carga (se recomienda parar ionic serve y volver a lanzar)

Si aún no funcionara, poner en app-routing.module.ts que en la parte de imports de @NgModule se importe InicioPageModule:

```
import { NgModule } from '@angular/core';
import { PreloadAllModules, RouterModule, Routes } from '@angular/router';
import { InicioPageRoutingModule } from './pages/inicio/inicio-routing.module';

const routes: Routes = [
  {
    path: 'inicio',
    loadChildren: () => import('./pages/inicio/inicio.module').then( m => m.InicioPageModule)
  },
];
@NgModule({
  imports: [
    RouterModule.forRoot(routes, { preloadingStrategy: PreloadAllModules }),
    InicioPageRoutingModule
  ],
```



```
    exports: [RouterModule]  
  })  
  export class AppRoutingModule { }
```

Juan Carlos Pérez Rodríguez

Navegar entre páginas

Vamos a crear una nueva página para poder ver como se navega de una a otra:

ionic p page pages/alert

Para navegar lo único que precisamos es hacer uso del router de angular:

En inicio.page.html:

```
<ion-header> <ion-toolbar><ion-title>inicio</ion-title></ion-toolbar></ion-header>

<ion-content>
  <ion-button routerLink="/alert" expand="block" shape="round">
    Ir a Alert
  </ion-button>
</ion-content>
```

back button

Con los snippets de ionic si escribimos: i-backbutton aparece algo similar a:

```
<ion-buttons slot="start">
<ion-back-button defaultHref="/" text="Regresar"></ion-back-button>
</ion-buttons>
```

slot="start" indica la posición (al inicio) donde se va a ubicar.

Ya el ion-back-button se encarga de hacernos regresar. Ahora bien, si no tiene a donde regresar (por ejemplo si se carga la aplicación en una pantalla diferente del inicio) le podemos decir una posición por defecto a la que regresar: defaultHref="/" y el texto que queramos que se muestre lo ponemos con: "text='mensaje'"

Creando aplicación cliente api monedas

Vamos a reproducir lo que vimos antes del router pero ya con la app que queremos

Primero creamos una nueva aplicación ionic blank

borramos la carpeta home que ha hecho para comenzar desde cero

creamos nuestras páginas:

ionic generate page pages/inicio

ionic generate page pages/monedas

ionic generate page pages/historicos

en app-routing.module.ts hay que borrar las referencias a home y dejamos que por defecto nos vaya a inicio:

```
monedas / src / app / ts app-routing.module.ts / routes / loadChildren
import { NgModule } from '@angular/core';
import { PreloadAllModules, RouterModule, Routes } from '@angular/router';

const routes: Routes = [
  {
    path: '',
    redirectTo: 'inicio',
    pathMatch: 'full'
  },
  {
    path: 'inicio',
    loadChildren: () => import('./pages/inicio/inicio.module').then( m => m.InicioPageModule)
  },
  {
    path: 'monedas',
    loadChildren: () => import('./pages/monedas/monedas.module').then( m => m.MonedasPageModule)
  },
  {
    path: 'historicos',
    loadChildren: () => import('./pages/historicos/historicos.module').then( m => m.HistoricosPageModule)
  },
];

@NgModule({
```

en src/app/pages/inicio/inicio.page.html agregamos una lista que nos lleve a las otras dos páginas

```
<ion-header>
  <ion-toolbar>
    <ion-title>inicio</ion-title>
  </ion-toolbar>
</ion-header>

<ion-content>
<ion-list>
  <ion-item>
    <ion-label routerLink="/monedas" > Monedas
    </ion-label>
  </ion-item>
  <ion-item>
    <ion-label routerLink="/historicos">Historicos</ion-label>
  </ion-item>
</ion-list>
</ion-content>
```

Ahora en cada una de ellas agregamos la posibilidad de regresar

```
<ion-buttons slot="start">
  <ion-back-button defaultHref="/" text="Regresar"></ion-back-button>
</ion-buttons>
```

Veamos ejemplo en src/app/pages/monedas/monedas.page.html

```
<ion-header>
  <ion-toolbar>
    <ion-buttons slot="start">
      <ion-back-button defaultHref="/" text="Regresar"></ion-back-button>
    </ion-buttons>
    <ion-title>MONEDAS</ion-title>
  </ion-toolbar>
</ion-header>
```

Vamos a construir un formulario que nos ayude con el filtrado de monedas:

```
<ion-header>
  <ion-toolbar>
    <ion-buttons slot="start">
      <ion-back-button defaultHref="/" text="Regresar"></ion-back-button>
    </ion-buttons>
    <ion-title>MONEDAS</ion-title>
  </ion-toolbar>
</ion-header>

<ion-content>
  <form #form="ngForm" (ngSubmit)="filtrarFormulario(form)">
    <ion-header>
      <ion-title class="ion-padding">Filtrar por:</ion-title>

      <ion-item>
        <ion-label>ID moneda: </ion-label>
        <ion-input name="inputidmoneda" ngModel type="number"></ion-input>
      </ion-item>

      <ion-item>
        <ion-label>Nombre monedas: </ion-label>
        <ion-input name="inputnombremoneda" ngModel type="text"></ion-input>
      </ion-item>

      <ion-button expand="full" shape="round" color="primary" type="submit">
        Filtrar
      </ion-button>
    </ion-header>
  </form>

  <ion-list>
    <ion-item *ngFor="let moneda of monedasFiltradas">
      <ion-label>{{moneda.nombre}} de {{ moneda.pais }}</ion-label>
    </ion-item>
  </ion-list>
</ion-content>
```

A los campos que se les ha establecido: `ngModel` pasan al controlador typescript (el componente) con el `name` que haya sido establecido. La clave para el envío de esa información está en como hemos definido el formulario:

```
<form #form="ngForm" (ngSubmit)="filtrarFormulario(form)">
```

`#form="ngForm"` vemos que se le dice a angular-ionic que lo trate como un formulario (ngForm) y le ponemos una etiqueta para poderlo referenciar: `#form`

Así luego en el listener que se le ha puesto: `(ngSubmit)="filtrarFormulario(form)"` vemos que cuando se pulse el botón de submit se desencadenará la función: `filtrarFormulario` que está en el fichero typescript y que esa función recibirá todo el formulario (observar que se le envía: `form` que es la etiqueta que se le puso al formulario al declararlo: `#form`)

Veamos en un `console.log` los datos de los campos:

```
filtrarFormulario( formulario:NgForm){  
  let controles = formulario.form.controls;  
  console.log(controles.inputidmoneda.value);  
  console.log(controles.inputnombremoneda.value);  
}
```

Servicios angular-ionic

Vamos a generar un servicio para las monedas (es quién se comunicará con la api y los componentes tomarán la información del servicio). Este “store” de información se basará en que nuestros componentes se “suscriban” para que los cambios que tengan lugar (consultas http a la api) el renderizado se actualice y veamos los datos obtenidos de la api.

Vamos a la consola:

```
ionic g s services/monedas
```

Para establecer el servicio, como hace uso de **httpclient** (una librería que nos permite hacer get, post, etc a una url) , debemos importar el módulo correspondiente: **HttpClientModule** en app.module.ts

```
lemondeas / src / app / ts app.module.ts / AppModule
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { RouteReuseStrategy } from '@angular/router';

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

import { AppComponent } from './app.component';
import { AppRoutingModule } from './app-routing.module';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  declarations: [AppComponent],
  entryComponents: [],
  imports: [BrowserModule, IonicModule.forRoot(), AppRoutingModule, HttpClientModule],
  providers: [{ provide: RouteReuseStrategy, useClass: IonicRouteStrategy }],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Vamos a ver un trozo del fichero del servicio: **src/app/services/monedas.service.ts**

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

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

  constructor(private http: HttpClient) { }

  getMonedas(){
    return this.http.get("http://localhost:8080/api/v1/monedas");
  }

  getMonedaId(id: number){
    return this.http.get(`http://localhost:8080/api/v1/monedas/${id}`);
  }
}
```

Aquí aparece un nuevo concepto: inyectar las dependencias. El concepto de inyección consiste en angular-ionic en poner en el constructor la declaración de un atributo (en el ejemplo: **private http: HttpClient**) Al hacerlo, angular se encarga de crear por si solo un objeto de ese tipo (observar que en ninguna parte de nuestro código hemos escrito: `new HttpClient`) Esto es muy útil para que haya dinamismo en las actualizaciones. Nosotros no vemos el trabajo interno que hace angular pero ya se encargará el framework de ponernos la librería `HttpClient` más apropiada (normalmente la más actualizada estable) por nosotros y proveernos del objeto allí donde lo necesitemos mediante la inyección: Recordar para hacer una inyección declaramos el atributo con el modificador de visibilidad: `public`, `private` (ya sea privado o público) dentro de los parámetros que recibe el constructor) Ejemplo:

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

Por lo demás lo único que hay que hacer es declarar los métodos/funciones que vamos a utilizar para comunicarnos con la api. En el ejemplo se muestra como obtener todas las monedas y una única moneda. ES IMPORTANTE que sepamos que `http.get()` devuelve un observable (Estamos ante asincronía. Se ejecutará algo cuando el servidor nos de la respuesta. Es una especie de promesa pero para múltiples eventos y tiene más funcionalidades)

¿ cómo usamos el servicio en nuestra aplicación ?

Vale, vamos a hacer que cuando se cargue la página del componente monedas: `src/app/pages/monedas` aparezcan todas ellas listadas. Eso lo tenemos un poco más arriba donde estábamos viendo la página html. Veamos de nuevo el trozo de código relevante:

```
<ion-list>
  <ion-item *ngFor=" let moneda of monedasFiltradas" >
    <ion-label>{{moneda.nombre}} de {{ moneda.pais }}</ion-label>
  </ion-item>
</ion-list>
```

vemos un `*ngFor` que lo que hará es recorrer un atributo de tipo array llamado: **monedasFiltradas** que está en el typescript de nuestra página: **src/app/pages/monedas/monedas.page.ts** y va a ir poniendo en cada línea de la lista tanto el nombre como el país de la moneda

Vamos a ver ahora la parte de typescript: **src/app/pages/monedas/monedas.page.ts**

```
import { MonedasService } from '../services/monedas.service';
import { Component, OnInit } from '@angular/core';
import { NgForm } from '@angular/forms';

interface Moneda {
  idmoneda: number;
  nombre: string;
  pais: string;
  historicos: Historico[];
}

interface Historico {
  idhistoricocambioeuro: number;
  fecha: string;
  equivalenteeuro: number;
}

@Component({
  selector: 'app-monedas',
  templateUrl: './monedas.page.html',
```

```

        styleUrls: ['./monedas.page.scss'],
    })
    export class MonedasPage implements OnInit {

        constructor( private monedasService:MonedasService){ }

        ngOnInit( ) {
            this.monedasService
                .getMonedas()
                .subscribe( (datos: any) =>{
                    this.monedasFiltradas = datos;
                });
        }

        monedasFiltradas: Moneda[]; //este es el atributo que se usa en el html para *ngFor
    }

```

Hemos marcado un par de cosas relevantes:

interface Moneda, interface Historico

Los interfaces en typescript sirven para generar una estructura de datos que podamos luego validar que es correcta (recordar que el gran valor de typescript respecto a javascript es que nos puede informar cuando las cosas están mal con los tipos de datos)

En nuestro caso nosotros recibimos un JSON de la api (un array de monedas o una moneda individual) Así que habría que decirle a typescript como son esos datos recibidos. Como esa es una tarea ardua podemos aprovechar una extensión de visual studio code:

JSON to TS: coverter from clipboard

La extensión toma el texto copiado en formato JSON (por ejemplo de la salida que nos muestre postman al hacer la consulta GET) y nos lo convierte directamente en interfaces

La siguiente línea interesante es:

```

constructor( private monedasService:MonedasService){ }

```

En esa línea tenemos nuevamente una inyección de dependencias. NO escribimos en ningún momento: new MonedasService y sin embargo nos crea el servicio que antes realizamos.

Nos falta ver como se usa el servicio. Para hacerlo hay que “suscribirse” al observable que devuelven los HttpClient:

```
ngOnInit( ) {  
    this.monedasService  
    .getMonedas()  
    .subscribe( (datos: any) =>{  
        this.monedasFiltradas = datos;  
    });  
}
```

Vemos que estamos en: `ngOnInit()` este método se desencadena cuando se carga nuestro componente monedas así que es un momento apropiado para tomar la información de las monedas que queremos mostrar. Para poder usar esa funcionalidad nuestra clase typescript implementa el interfaz:

```
export class MonedasPage implements OnInit {
```

La parte crucial de la llamada a nuestro servicio está en el método: “suscribe()” que es quién se encarga de resolver la respuesta asíncrona del observable: Esto es, ejecuta lo que incluyamos en el método cuando hayamos tenido respuesta de nuestra api:

```
.subscribe( (datos: any) =>{  
    this.monedasFiltradas = datos;  
});
```

En el ejemplo lo que hace es almacenar los datos obtenidos de la api en un atributo de nuestra clase typescript llamado: **this.monedasFiltradas** Observar que precisamente ese es el array de información que usa la página html. Veamos de nuevo el `*ngFor` en `monedas.page.html`:

```
<ion-list>  
  <ion-item *ngFor="let moneda of monedasFiltradas" >  
    <ion-label>{{moneda.nombre}} de {{ moneda.pais }}</ion-label>  
  </ion-item>  
</ion-list>
```

Como se ve gracias a que tenemos un atributo en la clase typescript que recibe la información que hemos pedido a la api (llamando al servicio que creamos) podemos luego usarlo en el `*ngFor` de nuestro html

Manejo de formularios

Cuando vimos el formulario de ejemplo que teníamos en nuestra página html: `src/app/pages/monedas/monedas.page.html` hablamos de como íbamos a enviar nuestro formulario:

```
(ngSubmit)="filtrarFormulario(form)"
```

ese trozo de código hace que cuando pulsemos en el botón de submit se desencadene el método: `filtrarFormulario()` que está en nuestra clase typescript y que ese método reciba el formulario. Vale, ahora lo que falta es ver que es lo que debe hacer ese método

¿ qué queremos que ocurra cuando se envía el formulario de filtrar monedas ? : Queremos que nos muestre las monedas que cumplen con el filtro que el usuario haya puesto (ya se por id de moneda o nombre de moneda) Así que deberá modificar el atributo: `this.monedasFiltradas` que es el que almacena el array de monedas que se va a mostrar en la página html

Veamos un ejemplo de la función `filtrarFormulario()` en el trozo que sirve para filtrar por id:

```
filtrarFormulario( formulario:NgForm){  
  let controles = formulario.form.controls;  
  //console.log(controles);  
  console.log(controles.inputidmoneda.value);  
  console.log(controles.inputnombremoneda.value);  
  if( controles.inputidmoneda.value > 0){  
    this.monedasService  
      .getMonedaId(controles.inputidmoneda.value)  
      .subscribe( (datos: Moneda) =>{  
        this.monedasFiltradas = [ datos];  
      });  
  }  
}
```

Vemos que recibimos un objeto de tipo `NgForm`: `formulario` (es lo que nos envía el html cuando pulsamos en submit) y que los `NgForm` puedes acceder a sus controles (los input, checkbox etc) mediante el objeto: **`formulario.form.controls`** que contiene todos los objetos. **IMPORTANTE!!:** si en el html no le hemos puesto la palabra: `ngModel` a los objetos que queremos usar del formulario aquí en la función que gestiona el formulario no podemos acceder a ellos. Así que si queremos acceder al input donde el usuario pone el id de la moneda:

```
<ion-input name="inputidmoneda" ngModel type="number"></ion-input>
```

TENEMOS QUE PONER: **`ngModel`** y darle un atributo: **`name`**

En el ejemplo se ha puesto el name: inputidmoneda Así que ese es el nombre que podemos usar en nuestra función typescript:

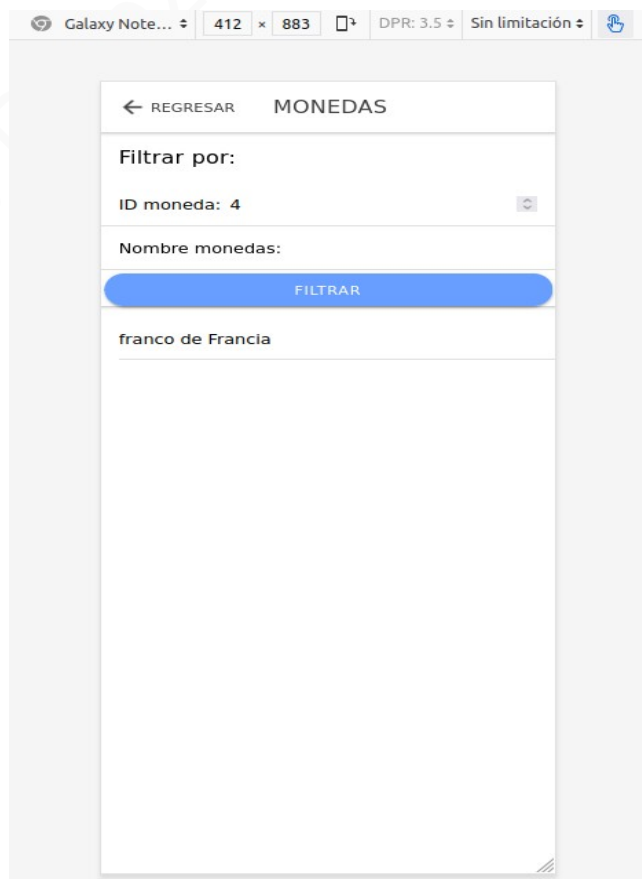
```
formulario.form.controls.inputidmoneda.value
```

Con lo anterior estamos tomando el valor que puso el usuario en el input del id de moneda

El resto del código es muy sencillo si el campo id del formulario tiene un número mayor que cero, se llama al servicio para que nos obtenga la moneda por el id dado y lo establecemos en nuestro atributo: `this.monedasFiltradas` de esa forma la página html va a aparecer renderizada con, únicamente la moneda puesta por el usuario:

```
if( controles.inputidmoneda.value > 0){  
  this.monedasService  
    .getMonedaId(controles.inputidmoneda.value)  
    .subscribe( (datos: Moneda) =>{  
      this.monedasFiltradas = [ datos];  
    });  
}
```

Veamos un ejemplo del renderizado que nos quedaría:



Consideraciones finales aplicación

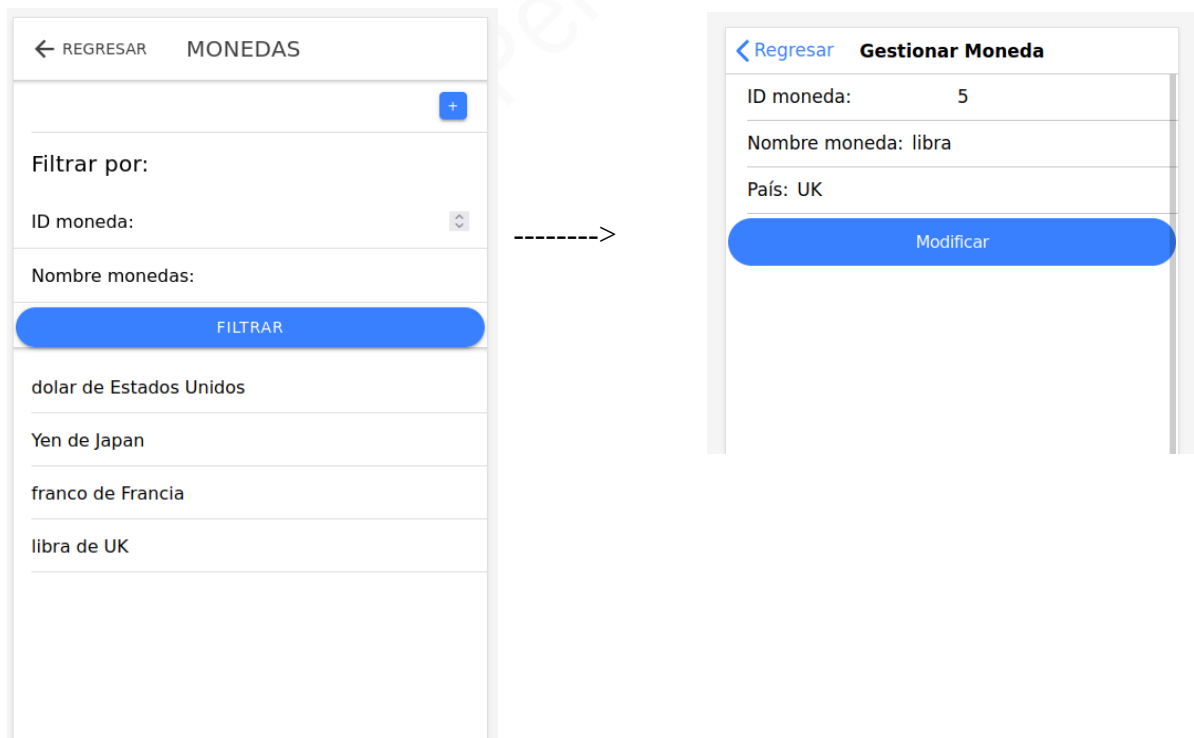
La aplicación precisará que se permita crear nuevas monedas:

Fichero: monedas.service.ts:

```
postMoneda(moneda: Moneda){  
  const headers = { 'content-type': 'application/json' };  
  const body=JSON.stringify(moneda);  
  console.log(body);  
  return this.http.post("http://localhost:8080/api/v1/monedas",body,{ 'headers':headers});  
}
```

y también borrarlas por id

Cuando pulsamos sobre una moneda de la lista nos la debe mostrar:



Para mostrar la moneda se propone usar los parámetros de router. Así la lista cuando nos muestra las monedas debe mostrarse ahora de una forma diferente:

```
<ion-list>
<ion-item *ngFor=" let moneda of monedasFiltradas" >

<ion-label routerLink="/gestionarmoneda/{ {{moneda.idmoneda}} }">{{moneda.nombre}} de
{{ moneda.pais }} </ion-label>
</ion-item>

</ion-list>
```

Observar que la ruta lleva un **id** como parámetro de **path**. Así que debemos recoger esa información en la página al inicio del renderizado:

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

ngOnInit(){
  this.route.paramMap.subscribe( params => {
    const id = Number(params.get('id'));
    this.monedasservice
      .getMonedaId(id)
      .subscribe( (m: Moneda) => {
        this.monedaActual = m;
      });
  });
}
```

Observar que angular-ionic ya recoge el uso de parámetros por id. Para usarlo, inyectamos: `ActivatedRoute` y luego lo recogemos con: `route.paramMap`

Luego tomamos el parámetro id solicitándolo específicamente: `params.get('id')`

Finalmente también haremos el agregado de una nueva moneda. Observar que en la página que mostramos las monedas nos aparece un botón con el símbolo: “+” La idea es que eso nos lleve a otra página que permita agregar una nueva moneda (método post en la api)



The screenshot shows a mobile application interface for adding a new coin. At the top, there is a navigation bar with a back arrow and the text "Regresar" in blue, followed by the title "Agregar Moneda" in bold black. Below the navigation bar, there are two text input fields: "Nombre moneda:" and "País:". Below these fields is a large blue button with the text "Agregar" in white. The entire form is enclosed in a light gray border. A large, faint watermark "Juan Carlos Pérez Rodríguez" is visible diagonally across the page.

Anexo: Trabajando con Router en Angular

Angular hace aplicaciones SPA (se emula la navegación por páginas mediante javascript)

Vamos a construir una aplicación desde cero para ver el router y entenderlo

Crear proyecto angular:

`ng new ejemplorouter`

nos preguntará por routing (de momento respondemos no)

le decimos que usaremos css

cuando termine entramos en la carpeta

Crear un Componente:

`ng g c pages/about`

`ng g c pages/contact`

Vamos a usar bootstrap. Así que en index.html ponemos en `<head>`:

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0-beta3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-e0JMYsd53ii+sc0/bJGFsiCZc+5NDVN2yr8+0RDqr0Ql0h+rP48ckxlpbZKgwr a6" crossorigin="anonymous">
```

Vamos a borrar el contenido de app.component.html y ponemos:

```
<h1>Componente principal</h1>
<app-about></app-about>
<app-contact></app-contact>
<app-home></app-home>
```

creamos el routing:

ng g m appRouting

Ahora en app-routing.module.ts debajo de los imports:

```
const routes: Routes = [
  {
    path: "home",
    component: HomeComponent
  },
  {
    path: "about",
    component: AboutComponent
  },
  {
    path: "contact",
    component: ContactComponent
  },
  {
    path: "**",
    component: HomeComponent
  }
];
```

en: @NgModule para las rutas tenemos:

RouterModule.forChild y RouterModule.forRoot

el forRoot es para las rutas principales → una única por aplicación habitualmente

y forChild es para las hijas

Luego hay que ir a app.module.ts y hay que importar el AppRoutingModule creado

```
@NgModule({
  declarations: [
    AppComponent,
    AboutComponent,
    ContactComponent,
    HomeComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Ahora en la página principal html: app.component.html debemos poner:

```
<h1>Componente principal</h1>

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

Observar que en app.component.html estamos dando una “estructura general” ahora a la página y cuando accedemos al <router-outlet> de una forma embebida nos pone el componente que corresponda (si vamos a: /about nos incorpora el contenido del componente about en el lugar dando hemos puesto router-outlet y así con las otras páginas)

Vamos a crear un componente que nos será un pequeño menú de navegación:

ng g c components/menu

Ahora en menu.component.html:

```
<ul class="list-group">
```

```
<a routerLink="/home" class="list-group-item">  
ir a la página inicial: home  
  
</a>  
</ul>
```

Observar routerLink Al haber creado nuestro módulo Router ahora los enlaces en nuestra app SPA ya funcionan y son llamados con routerLink

en el menu.component.ts:

```
constructor() { }  
  
ngOnInit(): void {  
}  
  
rutas = [  
  {  
    name: "Home",  
    path: "/home"  
  },  
  {  
    name: "About",  
    path: "/about"  
  },  
  {  
    name: "Contact",  
    path: "/contact"  
  },  
];  
}
```

Ahora en menu.component.html:

```
<ul class="list-group">
```

```
<li class="list-group-item" *ngFor="let ruta of rutas">
  <a routerLink="{{ruta.path}}" >
    {{ ruta.name }}
  </a>
</li>
</ul>
```

Juan Carlos Pérez Rodríguez

Módulo de páginas y enlazarlo al router

Creando un módulo páginas:

```
ng g m pages/pages --flat
```

Vamos a crear un módulo de rutas que sea específico para esta parte (dependerá del router padre y será cargado únicamente si es necesario: lazy load) por lo que se gana en eficiencia y tenemos compartimentación

```
ng g m pages/posts --routing
```

con `--routing` también crear un archivo de configuración de rutas

La ventaja es que hace el fichero de forma automática

ahora creamos un componente:

```
ng g c pages/posts
```

Al haber creado el componente lo lógico es que le asignemos una ruta. Pero en esta ocasión en lugar de irnos la módulo general nos vamos al que hemos creado ahora (`src/app/pages/posts/posts-routing.module.ts`) y lo ponemos:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { PostsComponent } from './posts.component';

const routes: Routes = [
  {
    path: 'posts', |
    component: PostsComponent
  }
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class PostsRoutingModule { }
```

Con lo anterior no es suficiente. Hay que informar al router principal que tiene que “delegar” en este otro router cuando corresponda:

```
const routes: Routes = [
  {
    path: "home",
    component: HomeComponent
  },
  {
    path: "about",
    component: AboutComponent
  },
  {
    path: "contact",
    component: ContactComponent
  },
  {
    path: "posts",

    loadChildren: () => import("../pages/posts/posts.module").then( m => m.PostsModule)
  },
  {
    path: "**",
    component: HomeComponent
  }
];

@NgModule({
  declarations: [],
  imports: [
    RouterModule.forRoot( routes),
    PostsModule
  ],
  exports: [
    RouterModule
  ]
})
export class AppRoutingModule { }
```

Nota: Observar que, como casi siempre, si hay un módulo hay que ponerlo en los Imports. En este caso adicionalmente a la carga perezosa (loadChildren) también ponemos el modulo en la parte de imports

Observar que aquí entra la “carga perezosa” Nosotros vemos que hay una promesa (el .then() nos ayuda a verlo) únicamente se cargará el import cuando se requiera acceder a posts.

El código nuevo en el módulo sería entonces:

```
{
  path: "posts",
  loadChildren: () => import("../pages/posts/posts.module").then( m => m.PostsModule)
```

}

Juan Carlos Pérez Rodríguez

Anexo: Error y subsanación ngModel el campo de formulario

Si se dan problemas (con las últimas versiones no se suele dar) con el uso de ngModel, esto es porque Angular a partir de la versión 4 en principio no reconoce ngModel como propiedad de un campo INPUT de formulario. Por ello, si ejecutas tu aplicación con el código tal como hemos hecho hasta ahora, obtendrás un mensaje de error como este: "Can't bind to 'ngModel' since it isn't a known property of 'input'."

La solución pasa por traernos esa propiedad, que está en el módulo "FormsModule".

Para ello tenemos que hacer la operativa tradicional de importar aquello que necesitamos. En este caso tendremos que importar FormsModule en el módulo donde vamos a crear aquel componente donde se quiera usar la directiva ngModel.

Por ejemplo, si nuestro componente "CuadradoComponent" está en el módulo "FigurasModule", este sería el código necesario para declarar el import de "FormsModule".

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

@NgModule({
  imports: [
    CommonModule,
    FormsModule
  ],
  declarations: [CuadradoComponent],
  exports: [CuadradoComponent]
})
export class FigurasModule { }
```

Observar que si no hemos puesto ningún módulo adicional el existente es el fichero: app.module.ts y es ahí donde trabajaremos

Vamos a ver un ejemplo reproducible:

en la parte de html:

```

<p>Input bidireccional:
    <input type="text" [(ngModel)]="enlaceDoble" />

</p>
<p>
    Vemos el contenido de la variable enlaceDoble:
    {{enlaceDoble}}
</p>

```

en la parte de app.component.ts:

```

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

@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
})
export class AppComponent {
    enlaceDoble: String = "";
}

```

En la parte de app.module.ts:

```

import { FormsModule } from '@angular/forms';
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

@NgModule({
    declarations: [
        AppComponent
    ],
    imports: [
        BrowserModule,
        FormsModule
    ],
    providers: [],
    bootstrap: [AppComponent]
})
export class AppModule { }

```

Observar que tabs está dentro de pages.

crear una nueva página:

ionic generate page pruebapagina

En tabs.html agregamos:

```
<ion-tab [root]="tab4Root" tabTitle="Prueba" tabIcon="contacts"></ion-tab>
```

En tabs.ts agregamos el import y el export:

```
import { PruebapaginaPage } from '../pruebapagina/pruebapagina';
```

```
tab4Root = PruebapaginaPage;
```

en src → app → app.module.ts:

```
import { PruebapaginaPage } from '../pages/pruebapagina/pruebapagina';
```

```
@NgModule({
  declarations: [
    MyApp,
    AboutPage,
    ContactPage,
    HomePage,
    TabsPage,
    PruebapaginaPage
  ],
  imports: [
    BrowserModule,
    IonicModule.forRoot(MyApp)
  ],
  bootstrap: [IonicApp],
  entryComponents: [
    MyApp,
    AboutPage,
    ContactPage,
    HomePage,
    TabsPage,
    PruebapaginaPage
  ]
})
```

```
});
```

Juan Carlos Pérez Rodríguez