

Módulo 4: Angular (Aplicaciones SPA)

AUTORAS: ROJAS CÓRSICO, Ivana Soledad, ALVARADO, Mónica Yamil

Índice

Índice	1
Introducción.	2
Objetivos	3
Introducción a las aplicaciones SPA	3
¿Qué es una aplicación SPA?	3
Varias vistas, no varias páginas	4
Plantillas y vistas	6
Arquitectura	7
Módulos	7
Crear módulos	7
Anatomía de un módulo	9
Componentes	10
Crear componentes	11
Anatomía de un componente	12
Crear un componentes dentro de un módulo	14
Manipulación de componentes	17
Exportar Componentes del módulo hacia afuera	17
Importar componentes de otros módulos	17
Sistema de Routing	21
Crear el módulo de rutas	22
Configurar las rutas de la aplicación	24
Crear rutas por defecto	27
Crear rutas a Página 404	28
Crear rutas con partes dinámicas en Angular	29
Crear rutas hijas	33
Data Binding	35
Tipos de Binding	35
Interpolation	37
Property Binding	38
Event Binding	40
Two way binding	42
Expresiones	43

Directivas	44
Directivas de Componente	44
Directiva: ngClass	45
Directiva ngModel	46
Directivas de estructura	46
Directiva ngFor	47
Directiva ngIf	48
Directiva ngSwitch	50
Pipes	52
Formularios en angular	54
Formularios reactivos	55
FormControl	56
Form Builder	57
Validaciones de Formularios	59
Tipo de validaciones	60
Validaciones con Form Builder	61
Validar previo enviar el formulario	63
Estilos Bootstrap en formularios reactivos	65
Servicios	66
Crear servicios	67
Anatomía de la clase de un servicio	67
¿Qué es Inyección de dependencias?	67
¿Cómo funciona?	68
Usando un servicio en un template	71
Referencias	71

Introducción.

A fin de aplicar todos estos conceptos abordaremos el prototipado en “Figma” de la aplicación y su desarrollo con html, bootstrap y css en Angular ahora, desde el punto de vista de componentes tal como se espera en Angular.

En esta sección profundizaremos Angular desde los conceptos básicos tales como ¿qué es una SPA?, módulos, componentes, directivas, pipes, rutas, entre otras.

Nota: Todo el fuente desarrollado en este documento puedes descargarlo del repositorio <https://github.com/pgClips/proyecto-pil-money-angular>

Objetivos

1. Crear módulos y componentes.
2. Crear diferentes tipos de rutas haciendo uso del sistema de rutas con Angular.
3. Crear formularios reactivos con sus respectivas validaciones.
4. Crear y consumir servicios.

Introducción a las aplicaciones SPA

Como expresamos en el módulo 2, Angular es una plataforma de desarrollo de código abierto mantenido en gran parte por Google y construido sobre Typescript para crear aplicaciones de una sola página Single Page Application (SPA).

¿Qué es una aplicación SPA?

A diferencia de las tradicionales MPA (múltiple page application), las aplicaciones SPA (single page application) consisten en aplicaciones de una sola página por lo que todas vistas de la misma se generan dinámicamente gracias a la capacidad de javascript para manipular el DOM. De esta manera, no recarga el navegador cada vez que el usuario hace una petición lo que permite que ésta sea óptima en rendimiento, mantenimiento y escalabilidad.

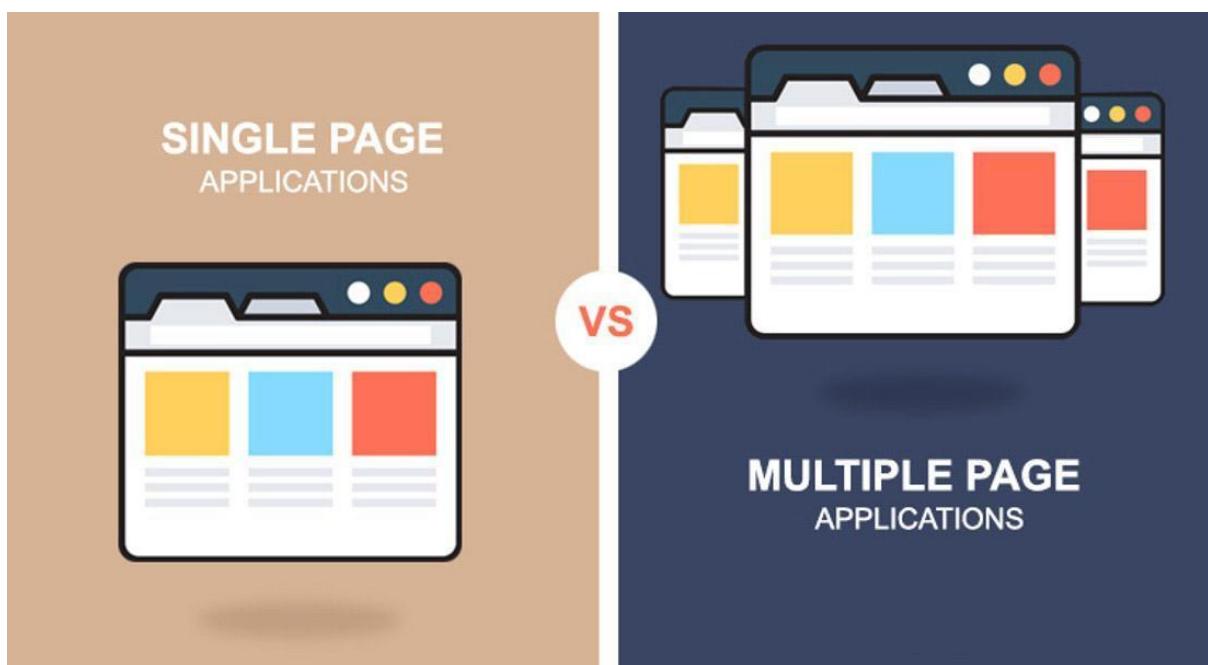


Figura 1: SPA vs MPA

Fuente: <https://livity.com/wp-content/uploads/2020/08/spavsmpa.jpg>

Es decir que en las aplicaciones SPA, encontramos toda la funcionalidad que hace a una

aplicación completa y en una única página web (index.html). Esto es posible, gracias al sistema de ruteo que propone Angular, el cual permite la generación dinámica de la página web.

A continuación un ejemplo gráfico de cómo sucede:

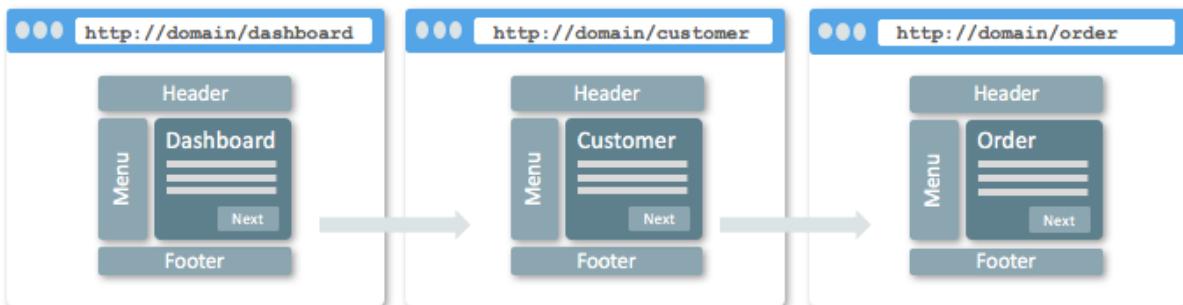


Figura 2: Vistas de una SPA

Fuente: <https://livity.com/wp-content/uploads/2020/08/spa-content-update.png>

En dicha imagen, podemos observar la generación dinámica de la página index.html dónde dashboard, customer y orders son generados dinámicamente según la petición del usuario.

Características funcionales de una SPA:

- **Punto de entrada central:** Un punto de entrada único que se genera dinámicamente según la petición del usuario.
- **Página fija, Vistas cambiantes:** Como en el caso de una aplicación de escritorio, nos mantenemos en un “marco único” y fijo, mientras que “vistas dinámicas” van ofreciéndonos las distintas posibilidades del uso y navegación.
- **Página fija, no URL fija:** Es posible que la dirección URL sufra cambios en base a las actividades de uso de la plataforma y vaya modificándose aunque ese “marco único” se mantenga fijado. Esto es un tanto reduccionista (existen SPA que no transforman sus direcciones), pero es útil para comprender su mecánica.
- **Viajar ligera de equipaje:** Las peticiones cliente — servidor tienden a ser más laxas y livianas que en las aplicaciones tradicionales MPA dado que sólo consisten en la transmisión de datos, sólo datos. Es importante agregar en este punto, que muchos procesos quedan del lado del cliente (navegador web) gracias a herramientas que provee Angular tales como el LocalStorage.” (<https://davidjguru.medium.com/single-page-application-un-viaje-a-las-spa-a-trav%C3%A9s-de-angular-y-javascript-337a2d18532>)

Varias vistas, no varias páginas

De lo expresado anteriormente podemos pensar que entonces las aplicaciones SPA

consisten de un único documento HTML y muchas vistas.

Existen entonces distintas partes (componentes) que controlan y definen una parte en pantalla (vista). Por ejemplo en la figura 2, los componentes individuales definen y controlan cada una de las vistas como sigue:

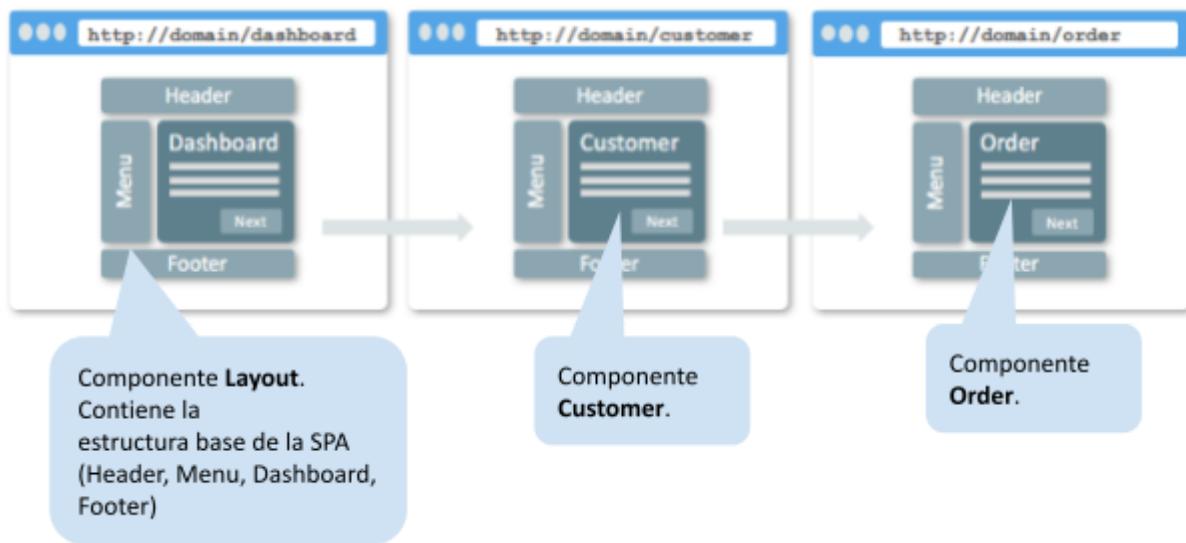


Figura 3: Vistas y componentes de una SPA

Fuente: <https://livity.com/wp-content/uploads/2020/08/spa-content-update.png>

Del ejemplo anterior podemos inferir que existe entonces una jerarquía de componentes tal y como se muestra en la siguiente imagen:

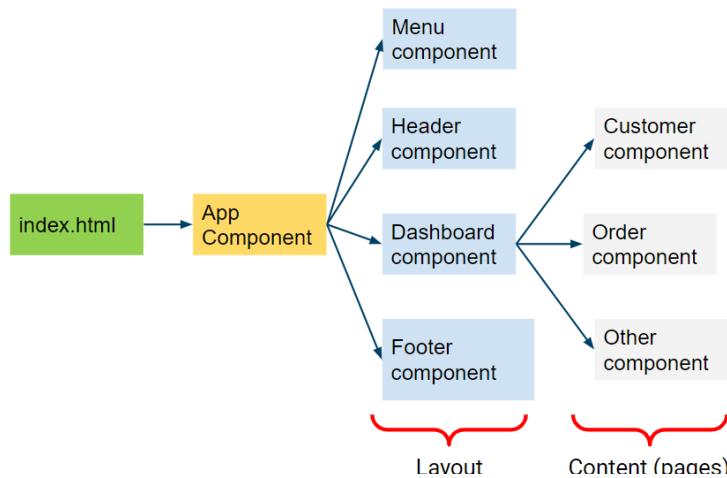


Figura 4: Jerarquía de componentes

Plantillas y vistas

De acuerdo a lo expresado anteriormente, es importante agregar que los componentes requieren de un template (o plantilla html) para que las vistas puedan ser interpretadas por el navegador web.

Un template no es más que un bloque HTML que le dice Angular cómo renderizar el componente definiendo así la vista.

El template es un HTML normal, excepto que también agrega sintaxis propia de Angular como por ejemplo la inclusión de lógica y bucles (ifs y fors), el sistema de binding que permite enlazar las variables al html (que eventualmente abordaremos más adelante).



Figura 5: Template y Vista

Fuente de la imagen: <https://angular.io/guide/architecture-components>

Entonces, dada la jerarquía de componentes, también podemos inferir que existe una jerarquía de vistas, las cuales contienen vistas embebidas, hosteadas incluso por otros componentes.

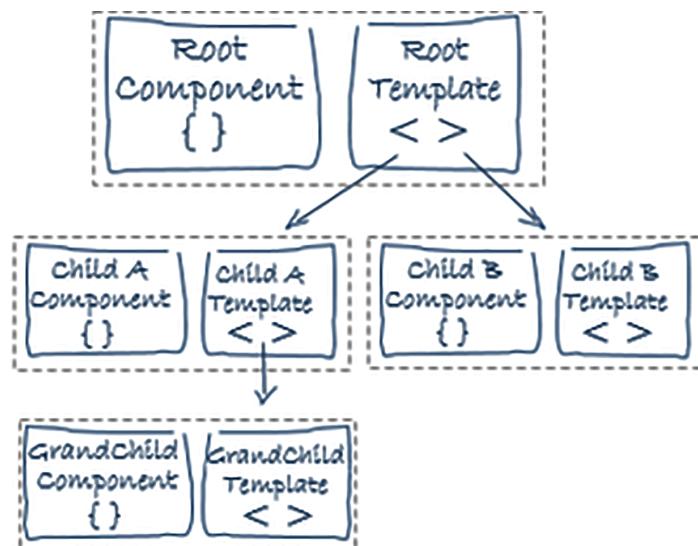


Figura 6: Jerarquía de vistas

Fuente de la imagen: <https://angular.io/guide/architecture-components>

Las vistas generalmente se organizan jerárquicamente, permiten mostrar u ocultar bloques HTML de la interfaz de usuario.

Arquitectura

La arquitectura de una aplicación en angular se basa en bloques de construcción, llamados Módulos o Ng Modules y proporcionan un contexto de compilación para cada uno de los componentes que hacen a la página. Cualquier sitio web creado con angular constituye un conjunto de módulos y componentes.

Módulos

De más está decir que las aplicaciones de Angular son modulares puesto que Angular tiene su propio sistema de modularidad llamado NgModules.

“Los NgModules son contenedores para un bloque cohesivo de código dedicado a un dominio de aplicación, un flujo de trabajo o un conjunto de capacidades estrechamente relacionadas. Pueden contener componentes, proveedores de servicios y otros archivos de código cuyo alcance está definido por el NgModule que los contiene. Pueden importar la funcionalidad que se exporta desde otros NgModules y exportar la funcionalidad seleccionada para que la utilicen otros NgModules.”
(<https://docs.angular.lat/guide/architecture-modules>)

En otras palabras, es una forma de crear y manipular los componentes agrupándolos de manera prolífica, cómoda y versátil. Es decir, en lugar de colocar todos los componentes, directivas o pipes en el mismo módulo principal, lo agrupamos en diferentes módulos de una manera lógica en base a las funcionalidades de la aplicación. Organizar el código de esta manera ayuda en el diseño, la reutilización, el mantenimiento y la escalabilidad.

Entonces, una aplicación de Angular debe contener al menos un módulo que convencionalmente se llama **AppModule** (el módulo raíz) y reside en el archivo **app.module.ts**. Si una aplicación es pequeña puede tener un solo módulo pero en aplicaciones más robustas lo ideal es tener más de un módulo.

Ejemplo:

Podríamos crear un módulo que contenga los componentes semánticos de nuestra aplicación web inicialmente a fin de crear la estructura básica de nuestra aplicación web.

Crear módulos

Para crear un módulo propio (aparte de app.module.ts), seguir los siguientes pasos:

1. Ir a la consola DOS o “Símbolo del Sistema” del sistema operativo o Terminal de VSCode.
2. Ejecutar el comando: ***ng generate module <<module-name>>***
o su abreviado: ***ng g m <<module-name>>***

Una vez ejecutado el comando, AngularCLI crea un subdirectorio con el nombre del módulo dentro de la carpeta "src/app" y un archivo: ***nombre.module.ts***.

como se muestra a continuación:

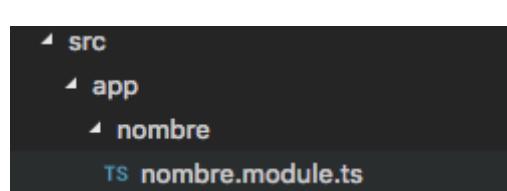


Figura 7: subdirectorio y archivo ts generado por Angular.

Nota: Angular CLI aplica las convenciones de nombres más adecuadas y como los módulos son clases, internamente les coloca en el código la primera letra siempre en mayúscula.

Ejemplo:

De acuerdo a lo explicado hasta el momento, nuestra primera tarea es definir los módulos que contendrá nuestra aplicación. Para ello, tomaremos como base la aplicación web “Mi aplicación” desarrollada en el módulo 2 la cual recordemos, contiene un archivo ***index.html*** con todo el fuente html (por el momento) y un único módulo ***app.module.ts***.

Los módulos tentativos podrían ser:

- **Layout**, incluye la estructura base de nuestra aplicación (etiquetas semánticas en HTML)
- **Pages**, incluye los contenidos dinámicos de la página web.

Nota: Denominamos pages dado que ayuda al entendimiento, pero recuerda que son componentes, no páginas web.

Para ello, ejecutamos en nuestra terminal de VSCode los siguientes comandos:

- `ng g m layout`
- `ng g m pages`

A continuación, podemos ver los directorios y archivos generados por el AngularCLI:

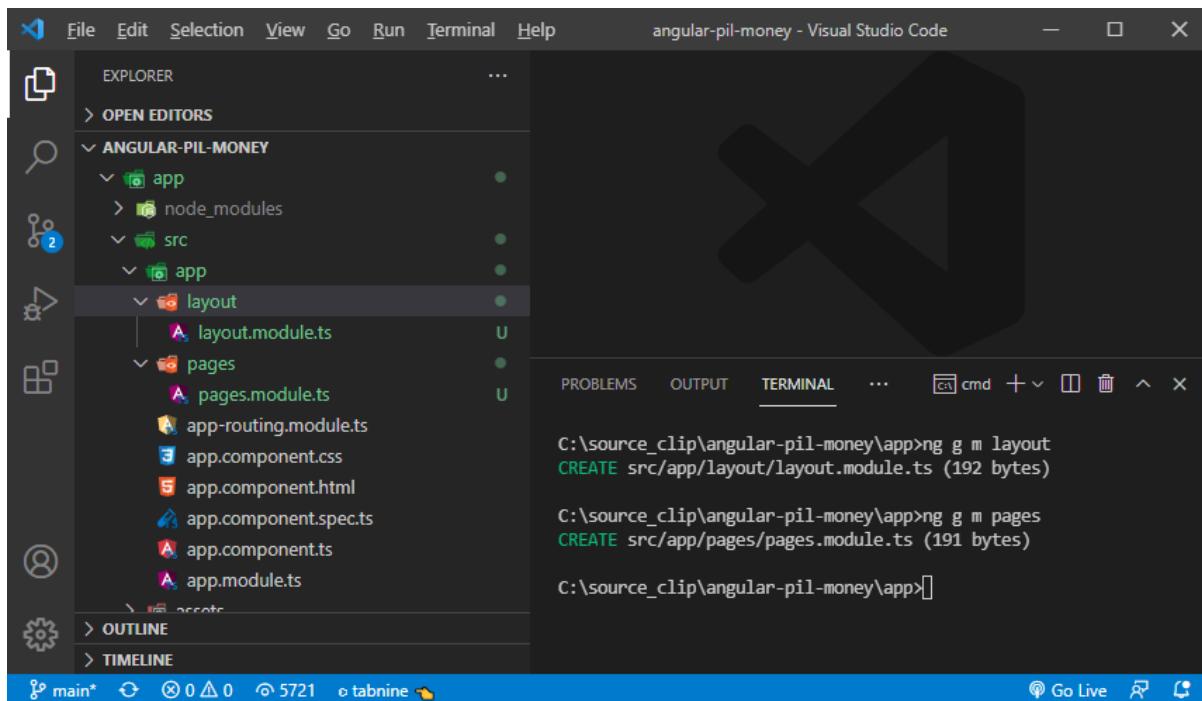


Figura 8: Generación de módulos

Anatomía de un módulo

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
@NgModule({
  declarations: [],
  imports: [
    CommonModule
  ]
})
export class LayoutModule { }
```

Como se puede observar, Angular define los módulos (que no son otra cosa que clases) a través del decorador `@NgModule`.

En el mismo, contiene dos arrays bien definidos:

- **imports:** clases exportadas importaciones necesarias. Le dice a Angular los sobre otros NgModules que este módulo en particular necesita para funcionar correctamente (<https://docs.angular.lat/guide/bootstrapping>)
- **declarations:** aquí se listan los componentes u otros artefactos que incluye este módulo.

Pudiendo además, agregarse lo siguientes:

- **providers**, enumera los proveedores de servicios necesarios.
- **bootstrap**, el componente raíz que Angular crea e inserta en la página web de host

index.html (<https://docs.angular.lat/guide/bootstrapping>)

- **exports:** aquí se listan los componentes exportados hacia afuera del módulo.

Componentes

Un componente es un bloque de código reutilizable que cuenta con 4 partes que definen la estructura del componente siendo obligatorios para la definición de las vistas el archivo .html (template) y .ts (lógica). “El mismo define una clase que contiene datos y lógica de la aplicación, y está asociado con una plantilla HTML que define una vista que se mostrará en un entorno de destino” (<https://docs.angular.lat/guide/architecture>).

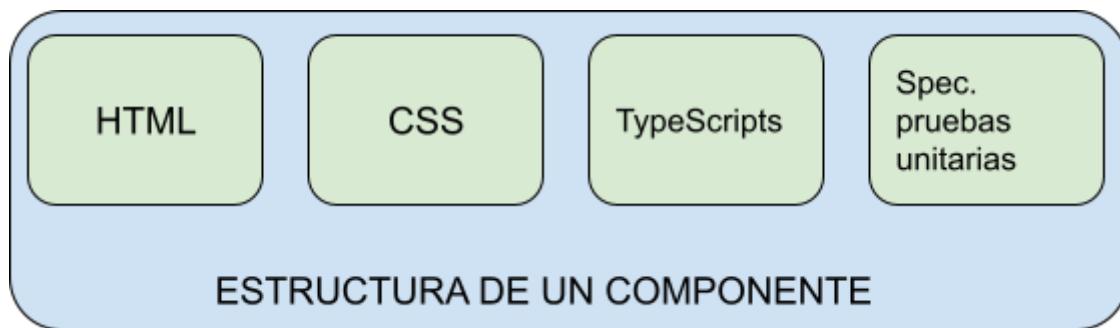


Figura 9: Archivos que constituyen un componente.

Cada aplicación de Angular tiene al menos un componente, el componente raíz: **app.component** que conecta una jerarquía de componentes con el modelo de objetos del documento de la página (DOM).

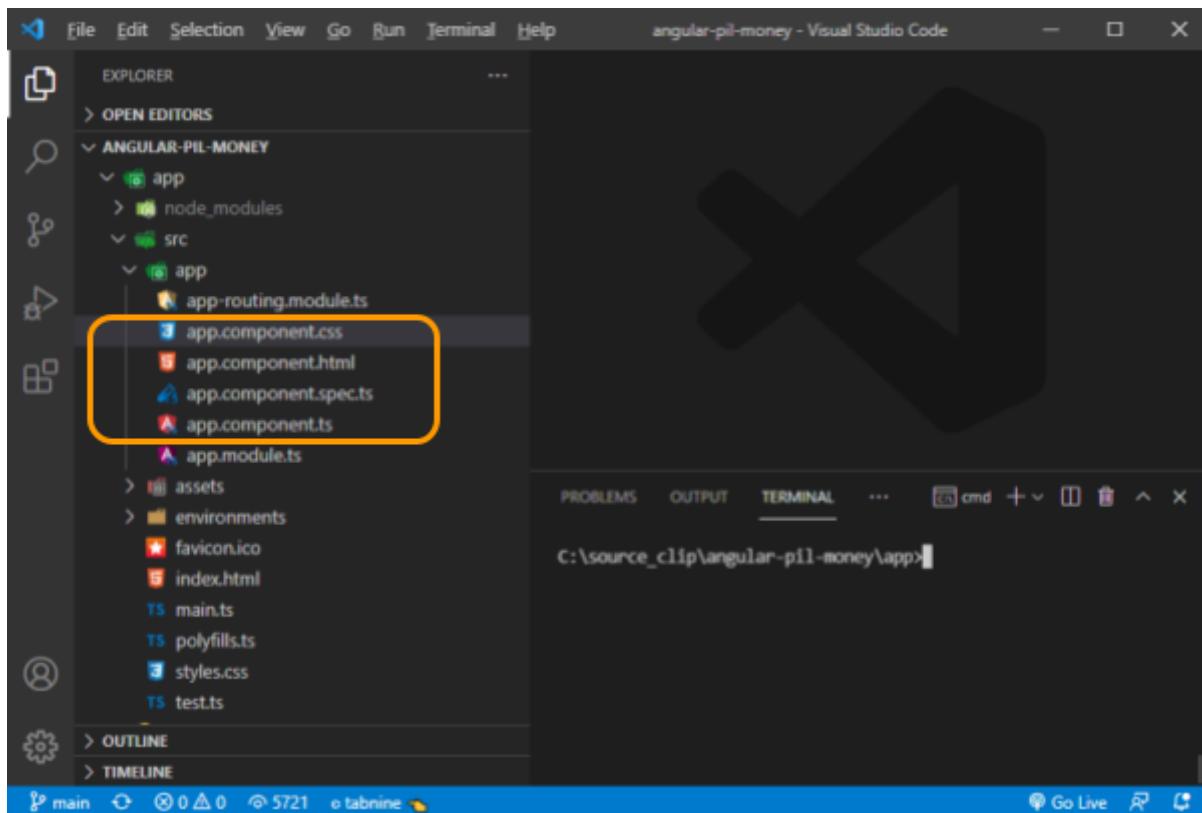


Figura 10: Componente raíz.

Como puede observarse arriba, el mismo cuenta de 4 archivos que son:

- **app.component.html**, el template html.
- **app.component.ts**, define la lógica.
- **app.component.css**, los estilos propios del componente.
- **app.component.spec.ts**, destinado para las pruebas unitarias.

Crear componentes

Para crear un componente, ejecutar los siguientes pasos:

1. Ir a la consola DOS o “Símbolo del Sistema” del sistema operativo o Terminal de VSCode.
2. Ejecutar el comando: ***ng generate component <<component-name>>***

o su abreviado: ***ng g c <<component-name>>***

Una vez ejecutado el comando, AngularCLI crea 4 archivos los siguientes 4 archivos

- <<nombre>>.component.ts
- <<nombre>>.component.html
- <<nombre>>.component.css

- <>component.spec.ts

y actualiza el módulo que lo contiene a fin de declarar dichos componentes.

Anatomía de un componente

Los componentes, como el resto de artefactos en Angular, son clases TypeScript decoradas con funciones específicas como podemos observar en el archivo **app.component.ts**:

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

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

En este caso el decorador **@Component()** define el componente identificando la clase y su metadata.

La metadata de un componente le dice a Angular dónde obtener los bloques de construcción que necesita para crear y presentar la vista que definen los componentes.



El diagrama muestra el código de app.component.ts con flechas rojas que apuntan a diferentes partes del código y describen su función:

- Importamos el componente: `import { Component } from '@angular/core';`
- Le damos nombre al "selector" con este llamaremos al componente: `@Component({ selector: 'app-root', ... })`
- Llamamos al template "html" del componente: `templateUrl: './app.component.html'`
- Llamamos a los estilos del componente: `styleUrls: ['./app.component.css']`
- Exportamos la clase para que sea utilizada al momento de cargar la página: `export class AppComponent { ... }`

Figura 11: Archivo app.component.ts

De la imagen anterior podemos inferir en 3 partes importantes que hacen a un componente

en Angular:

- **selector**, le dice a Angular que cree e inserte una instancia de este componente siempre que encuentre la etiqueta en el html. Por ejemplo, si el HTML de una aplicación contiene <app-header></app-header>, entonces Angular inserta una instancia de la vista HeaderComponent entre esas etiquetas.
- **templateUrl**, la dirección relativa al template HTML del componente. Alternativamente, se puede escribir código html.
- **stylesUrl**, la dirección relativa al archivo CSS del componente.

En pocas palabras, el selector es el nombre con el cual luego invocamos al componente en el html como se puede observar a continuación:

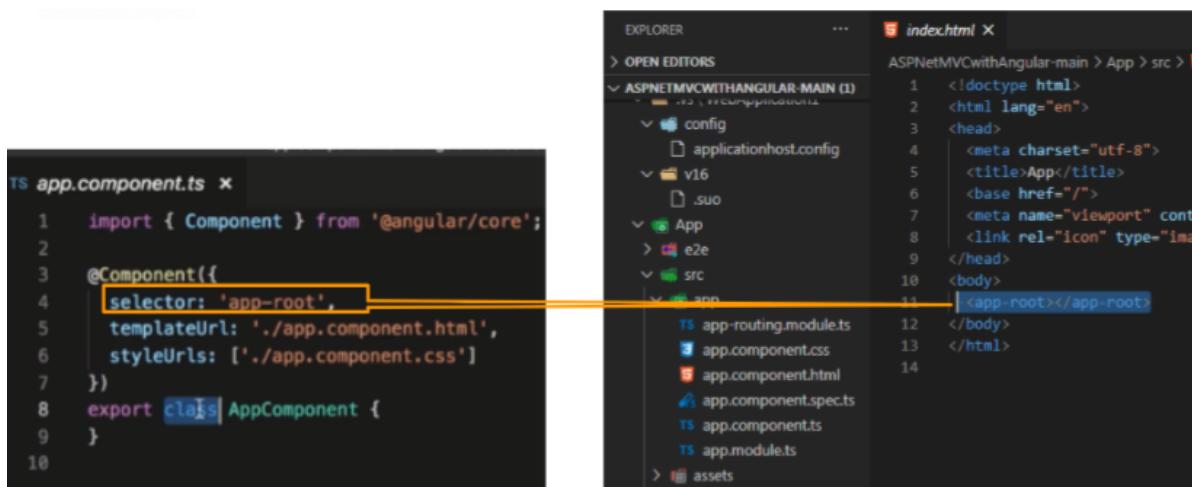


Figura 12: Selector del componente raíz (`app-root`) invocado en el archivo `index.html`.

Nota: Excepcionalmente en este caso, por ser el componente raíz, el mismo se invoca desde el `index.html` pero, es importante agregar que los mismos pueden ser invocados desde el template de otros componentes.

En este punto, es importante agregar que el componente **AppComponent** además de ser referenciado en el html a través de la etiqueta `<app-root>`, se debe declarar en el módulo que lo contiene. En este caso en el módulo raíz `app.module.ts`) como se muestra a continuación:

```

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

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

```

Declaración del componente AppComponent en el módulo.

```
declarations: [
  AppComponent
],
imports: [
  BrowserModule,
  AppRoutingModule
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }
```

Crear un componentes dentro de un módulo

Para crear un componentes de un módulo, ejecutar los siguientes pasos:

1. Ir a la consola DOS o “Símbolo del Sistema” del sistema operativo o Terminal de VSCode.
2. Ejecutar el siguiente comando: **ng generate component <>**
o su abreviado:
ng g c <>

Una vez que angularCLI no sólo genera el componente (archivos .html, .ts, .css y spec.ts) sino que también actualiza el módulo que lo contendrá a fin de hacer las declaraciones pertinentes.

Ejemplo:

Continuando con nuestro ejemplo, el siguiente paso es crear los componentes. Para ello, podemos valernos de la siguiente imagen explicativa:

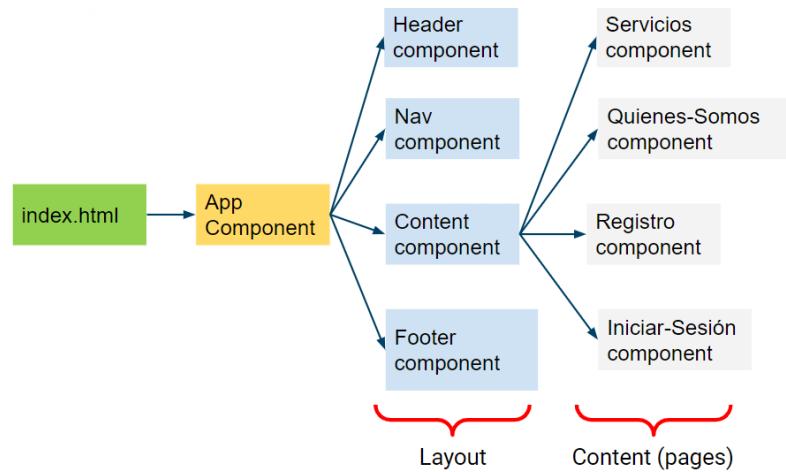


Figura 13: Jerarquía de componentes

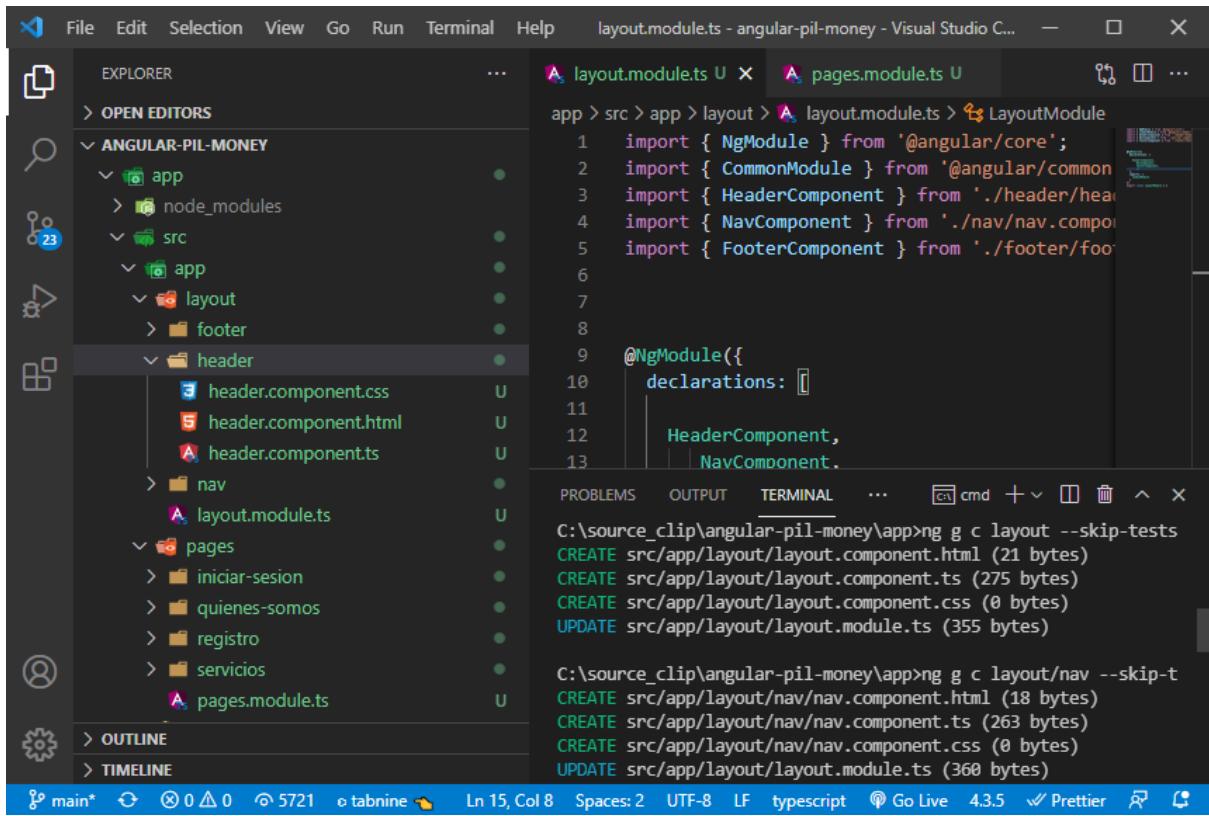
En la misma podemos apreciar que el módulo **Layout** está compuesto por los componentes: header, nav, content y footer mientras que los otros componentes pertenecen al módulo **Pages**.

A continuación, ejecutamos en nuestra terminal de VSCode los siguientes comandos para generar los componentes del layout como sigue:

- `ng g c layout/header --skip-tests`
- `ng g c layout/nav --skip-tests`
- `ng g c layout/footer --skip-tests`

Nota: Podemos utilizar el comando **--skip** para evitar crear todos los elementos o archivos del componente.

--skip -tests, nos permite evitar crear el archivo de pruebas unitarias.



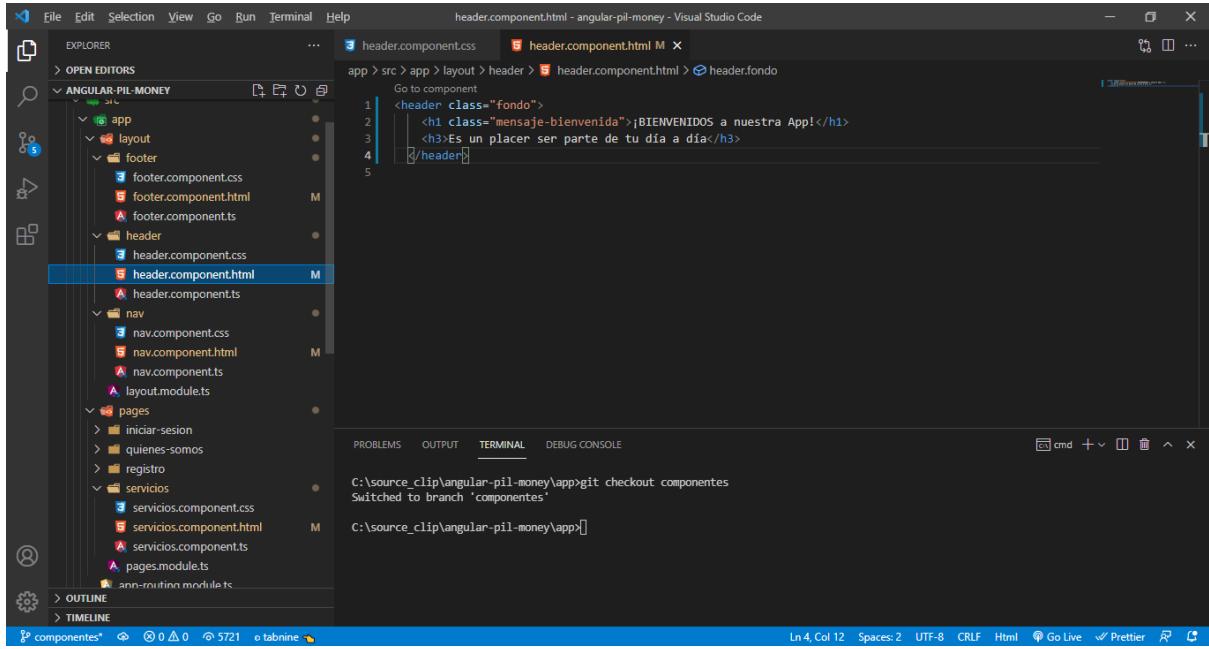
The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure under "ANGULAR-PIL-MONEY". The "header" folder is selected.
- Editor:** Displays the content of `layout.module.ts`, which imports various Angular modules and components.
- Terminal:** Shows command-line output from running Angular commands like `ng g c layout --skip-tests` and `ng g c layout/nav --skip-tests`.
- Bottom Status Bar:** Provides information such as the current file (`main*`), line and column numbers (`Ln 15, Col 8`), and other development settings.

Figura 11: Directories y archivos generados para cada componente.

De la misma manera, creamos el resto de los componentes correspondientes al módulo pages.

A continuación, distribuimos el código del `index.html` a los templates de los componentes que corresponden.



The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure under "ANGULAR-PIL-MONEY". The `header` folder is selected.
- Editor:** Displays the content of `header.component.html`, which contains HTML code for a header component.
- Terminal:** Shows command-line output from running `git checkout componentes`.
- Bottom Status Bar:** Provides information such as the current file (`componentes*`), line and column numbers (`Ln 4, Col 12`), and other development settings.

Figura 12: header.component.html

Manipulación de componentes

Sin lugar a dudas, al crear una aplicación en angular, trabajaremos con una jerarquía de módulos y componentes por lo que es importante comprender cómo manipularlos.

Exportar Componentes del módulo hacia afuera

Por defecto los componentes definidos dentro de un módulo sólo son accesibles por éste. Si deseamos dejar visibles componentes, para que luego sean utilizados desde otros componentes, simplemente deberemos invocarlos en el array **exports** del módulo como sigue:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { HeaderComponent } from './header/header.component';
import { NavComponent } from './nav/nav.component';
import { FooterComponent } from './footer/footer.component';

@NgModule({
  declarations: [
    HeaderComponent,
    NavComponent,
    FooterComponent,
  ],
  imports: [
    CommonModule
  ]
  exports: [
    HeaderComponent,NavComponent, FooterComponent,
  ]
})
export class LayoutModule { }
```

Exportamos los componentes hacia afuera

Importar componentes de otros módulos

Para hacer uso de los componentes declarados en otro módulo, se debe importar el módulo completo que contiene dichos componentes al módulo destino. Para ello, editar el módulo

destino como sigue:

1. Agregar la referencia relativa del módulo.
2. Referenciar en el array ***imports***

```

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

import { LayoutModule } from './layout/layout.module'; ←

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

Para comprenderlo mejor, continuemos con el ejemplo:

En el apartado anterior, dividimos el fuente html en componentes, ahora intentaremos hacerlo funcional dado que si lo dejamos así, nada podremos observar en el navegador web dado que, justamente nos falta manipular dichos componentes.

En primer lugar, tal como se muestra en el jerarquía de componentes de nuestra aplicación (figura 13) partimos del archivo **index.html**, el cual invoca al componente raíz **AppComponent**.

Para lograrlo, simplemente debemos editar el body de nuestro index.html agregando la etiqueta **<app-root></app-root>** como se muestra a continuación:

```

<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>App</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>

  A través de esta etiqueta
  instanciamos el componente
  AppComponent

<app-roo></app-roo>
<script src="https://use.fontawesome.com/releases/v5.15.3/js/all.js"
crossorigin="anonymous"></script>
</body>
</html>

```

A continuación, exportamos hacia afuera todos los componentes que luego, serán invocados desde el template del AppComponent. Para ello, editamos los módulos de Layout como sigue:

layout.module.ts

```

import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { HeaderComponent } from './header/header.component';
import { NavComponent } from './nav/nav.component';
import { FooterComponent } from './footer/footer.component';

@NgModule({
  declarations: [
    HeaderComponent,
    NavComponent,
    FooterComponent,
  ],
  imports: [
    CommonModule
  ],
  exports: [
    HeaderComponent, NavComponent, FooterComponent,
  ]
})
export class LayoutModule { }

```

Exportamos hacia afuera los componentes que vamos a compartir.

Finalmente editamos el módulo **AppComponent** a fin importar los componentes que necesitamos consumir.

Para ello, ejecutar los siguientes pasos

1. Editar el módulo que contiene al componente **AppComponent**, en este caso el módulo raíz **app.module.ts** a fin de importar los módulos de **Layout** y **Pages** como sigue:

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { LayoutModule } from './layout/layout.module';
import { PagesModule } from './pages/pages.module';

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

```

Importamos los módulos que vamos a consumir

2. Luego, simplemente modificar el template del componente **app.component.html** a fin de instanciar los componentes como sigue:

```

<app-nav></app-nav>
<app-header></app-header>
<app-servicios></app-servicios>
<app-footer></app-footer>

```

De esta manera, queda distribuido nuestro html en componentes y si visualizamos en el navegador web veremos que, en relación a tener todo el fuente en el index.html no hay cambios visibles.



Figura 12: Aplicación corriendo en un navegador web

Nota: Para visualizar en un navegador web, recuerda que puedes ejecutar el comando **ng serve -o**

Hasta aquí hemos distribuido el código html en componentes pero estas son estáticas. ¿Cómo logramos que sea dinámica? Es decir, que se construya ese html en función de las peticiones del usuario. Abordamos eso a continuación.

Sistema de Routing

Así como en las aplicaciones web tradicionales, las aplicaciones SPA también tienen rutas, pero estas son “virtuales” dado que el servidor no entrega una página web completa como en las aplicaciones tradicionales sino que, entrega datos (como expresamos anteriormente). Es decir, no habrá una página web blog.html, contacto.html, etc.

En Angular lo común es que el index.html sólo tiene un componente raíz AppComponent en su body y toda la acción se desarrollará en dicho componente. Todas las “páginas” (o vistas) se mostrarán sobre ese archivo, intercambiando el componente que se esté visualizando en cada momento.

Para facilitar esto, Angular provee el sistema de routing AppRoutingModule, el cual permite definir rutas “virtuales” tal como las que existen en los sitios tradicionales.

En el sistema de Routin, las *Rutas* “virtuales” le indican al enrutador qué vista mostrar

cuando un usuario hace clic en un enlace o pega una URL en la barra de direcciones del navegador. (Angular, <https://docs.angular.lat/tutorial/toh-pt5>).

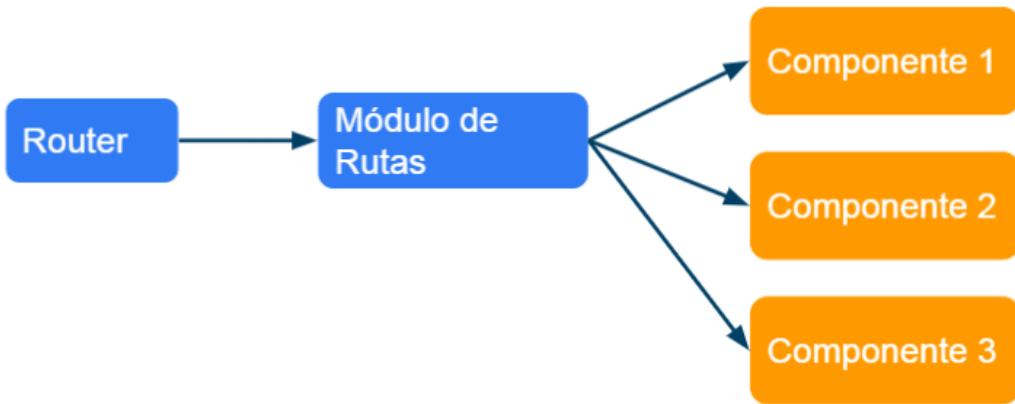


Figura 13: Estructuras de Rutas.

En resumen, podemos decir que el Sistema de Routing de Angular tiene por objeto mostrar las vistas de la aplicación en función de una ruta “virtual”.

“Los elementos básicos que forman parte del Sistema de Rutas son:

- **El módulo del sistema de rutas:** llamado RouterModule.
- **Rutas de la aplicación:** es un array con un listado de rutas que nuestra aplicación soportará.
- **Enlaces de navegación:** son enlaces HTML en los que incluiremos una directiva para indicar que deben funcionar usando el sistema de routing.
- **Contenedor:** donde colocar las páginas (o vistas) de cada ruta. Cada página (o vista) será representada por un componente.”
<https://desarrolloweb.com/articulos/introduccion-sistema-routing-angular.html>)

Crear el módulo de rutas

Para crear un módulo de rutas manualmente, ejecutar los siguientes pasos:

1. Ir a la consola DOS o “Símbolo del Sistema” del sistema operativo o Terminal de VSCode.
2. Ejecutar el comando: ***ng g m app-routing.module.ts***
3. Editar el módulo creado anteriormente a fin de:
 - a. Importar el módulo RouterModule y la clase Routes.

- b. Crear un array routes que contendrá luego las rutas virtuales.
- c. Importar y exportar el RouterModule.

A continuación un ejemplo de cómo debería quedar finalmente el archivo app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
const routes: Routes = [
];

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

4. En el módulo que contendrá los componentes dinámicos (app.module.ts), importar el módulo recién creado (AppRoutingModule) como sigue:

Módulo app.module.ts

```
import { AppRoutingModule } from './app-routing.module'; ←
import { AppComponent } from './app.component';
import { LayoutModule } from './layout/layout.module';
import { PagesModule } from './pages/pages.module';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    LayoutModule,
    PagesModule,
    AppRoutingModule ←
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Nota: Puedes obviar este paso si, al momento de crear el proyecto de angular, configuraste "Y" como respuesta a la pregunta **Would you like to add Angular routing?** ya que, se crearon de manera automática todos los archivos relacionados al routing.

Configurar las rutas de la aplicación

Para ello, ejecutar los siguientes pasos:

1. Editar el archivo **app-routing.module.app** a fin de importar: **RouterModule** y **Routes**.
2. Definir las rutas “virtuales” en el array **routes** como sigue:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { IniciarSesionComponent } from
'./pages/iniciar-sesion/iniciar-sesion.component';
import { QuienesSomosComponent } from
'./pages/quienes-somos/quienes-somos.component';
import { RegistroComponent } from './pages/registro/registro.component';
import { ServiciosComponent } from './pages/servicios/servicios.component';

const routes: Routes = [
  {path: 'servicios', component:ServiciosComponent},
  {path: 'quienes-somos', component: QuienesSomosComponent},
  {path: 'registro', component:RegistroComponent},
  {path: 'iniciar-sesion', component: IniciarSesionComponent}
];

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

dónde cada ruta está configurada en base a las propiedades:

- **path:** define la ruta virtual de nuestra aplicación.
- **component:** define el componente que le dice al enrutador que componente corresponde al seleccionar dicha ruta.

Nota: Existen propiedades complementarias que son usadas por las rutas al momento del enrutamiento como por ej. **pathMatch** para configurar que la ruta coincida parcial o total.

3. Agregar las rutas a la aplicación. Para ello, enlazamos la ruta creada anteriormente al atributo **routerLink** del enlace.

A continuación, y siguiendo el ejemplo que venimos desarrollando se muestra la configuración de rutas en el componente **nav.component.html** de nuestra **app**:

```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark fixed-top">
  <div class="container-fluid">
    <button class="navbar-toggler" type="button" data-bs-toggle="collapse"
aria-expanded="false" aria-label="Toggle navigation">
      <span class="navbar-toggler-icon"></span>
    </button>
    <p class="navbar-brand mb-2">Mi aplicación</p>
    <div class="collapse navbar-collapse">
      <ul class="navbar-nav ms-auto mb-2 mb-lg-0">
        <li class="nav-item">
          <a class="nav-link active" routerLink="/servicios"
aria-current="page">SERVICIOS</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" routerLink="/quienes-somos">¿QUIENES SOMOS?</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" routerLink="/registro">REGISTRARSE</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" routerLink="/iniciar-sesion">INICIAR SESIÓN</a>
        </li>
      </ul>
    </div>
  </div>
</nav>
```

Atributo que permite
 linkear la ruta virtual con
 el enlace.

El roterLink es el selector para la directiva RouterLink que convierte los clics del usuario en navegaciones del enrutador.

4. Finalmente, agregamos la etiqueta **<router-outlet></router-outlet>** en el html donde deseo que aparezca el ruteo.

router-outlet es una etiqueta especial de Angular que sirve para mostrar los componentes hijos de un componente.

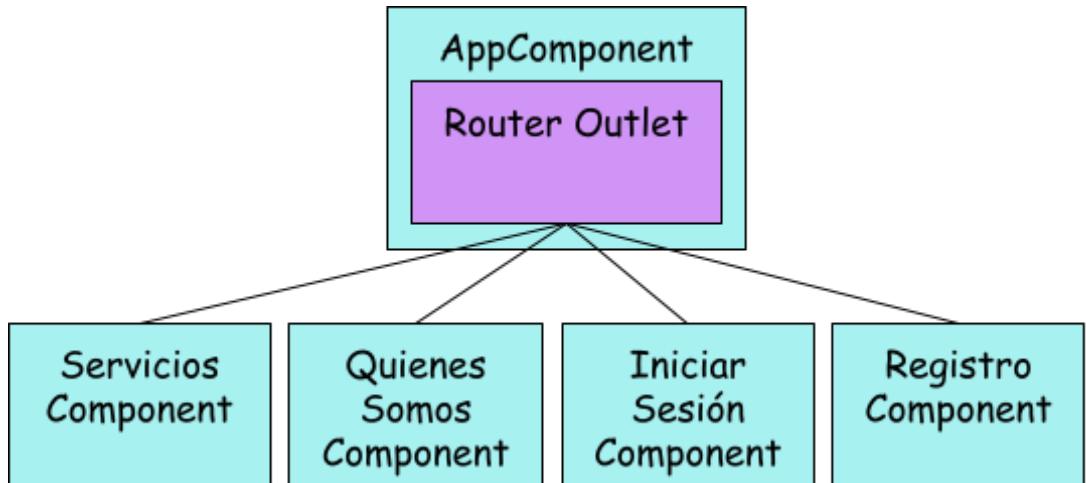


Figura 13: Gráfico del sistema de rutas de nuestra app

Nota: Por defecto todos los componentes son hijos del componente AppComponent, por lo que si incluimos esta etiqueta dentro de la vista de AppComponent, se renderizará cada uno de los componentes del routing dependiendo de la página en la que nos encontremos.

Además, es posible que los componentes hijos también tengan sus propios sistemas de rutas.

En nuestro ejemplo, reemplazamos la etiqueta `<app-services></app-services>` por `<router-outlet></router-outlet>` como sigue:

```

<app-nav></app-nav>
<app-header></app-header>
<router-outlet></router-outlet> ←
<app-footer></app-footer>
  
```

De esta manera, y dependiendo de la petición del usuario, se mostrará uno u otro componente.

Si ejecutamos `ng serve -o` podremos contemplar el sistema de routing funcionando.

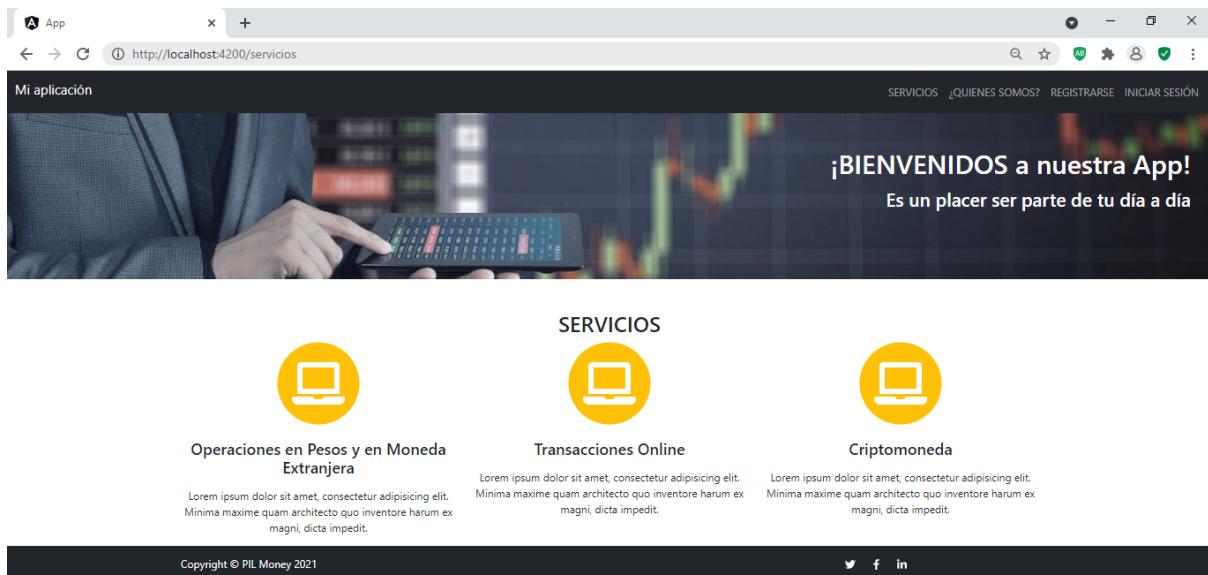


Figura 14: ruta servicios

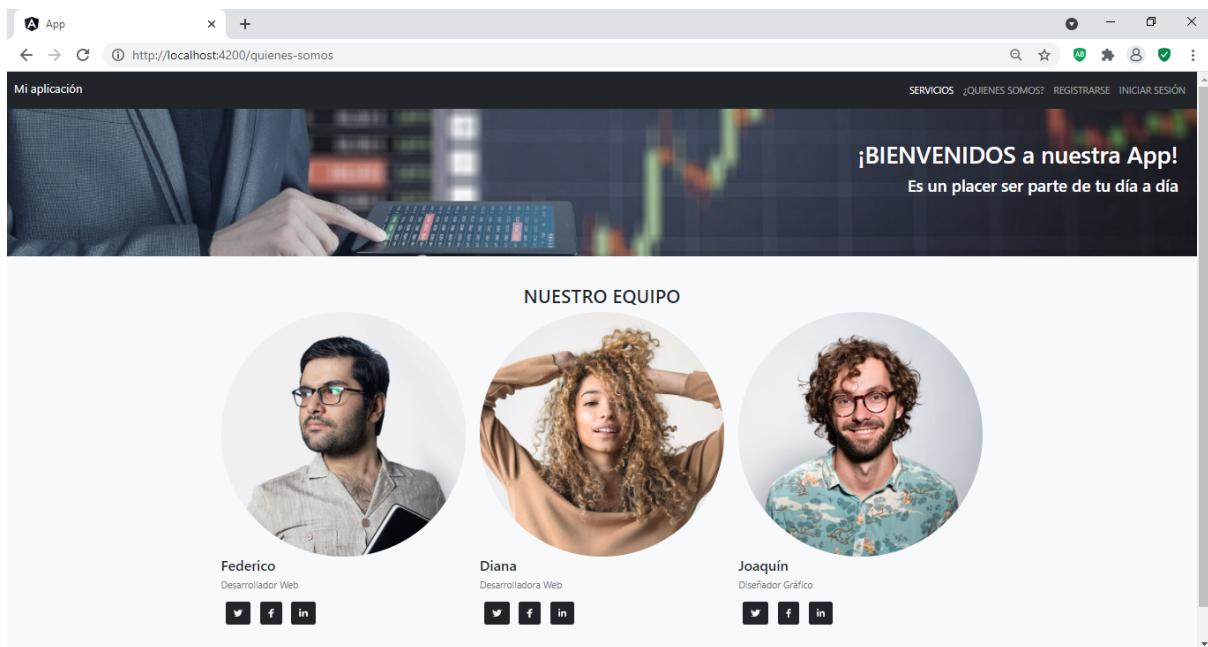


Figura 15: ruta quienes-somos

Observa que es posible navegar escribiendo las rutas en el navegador o, a través de la barra de navegación.

Crear rutas por defecto

Sin dudas, siempre que tengamos una aplicación web nos interesará configurar rutas por defecto.

Para configurar el redireccionamiento a un componente por defecto, como por ej. al home, editar el array routes del archivo **app-routing.module.ts** a fin de agregar la siguiente ruta por

defecto:

```
{path: '', redirectTo: '/inicio', pathMatch: 'full'}
```

Para definir una ruta por defecto, debemos utilizar 2 propiedades más de las rutas que son:

- **redirectTo**, permite redireccionar a la ruta “/inicio” si la ruta es una cadena vacía dado que,
- **patchMatch**, especifica que la coincidencia sea exacta. Es decir, que si introducimos otra cosa como ruta (que no sea la cadena vacía) no redireccionará al inicio.

Luego, si el usuario especifica la ruta <http://localhost:4200/> , el sistema de routing nos llevará automáticamente a la ruta <http://localhost:4200/inicio>.

Crear rutas a Página 404

Sin dudas, nos interesará también configurar una página 404 en el sistema de rutas para cuando el usuario intente escribir una ruta que no exista.

Para ello, ejecutar los siguientes pasos:

1- Crear el componente Pagina404 (not found)

2- Editar el array routes de nuestro app-routing.module.ts a fin de agregar la ruta a la página 404 como sigue:

```
{path: '**', component: Pagina404Component}
```

Luego, si introducimos una ruta cualquiera el sistema de rutas nos redireccionará a la página 404.



Figura 16: Página 404

Crear rutas con partes dinámicas en Angular

Angular también nos permite configurar cierta parte de la ruta dinámica.

Para ello, ejecutar los siguientes pasos:

1. Editar el archivo **app-routing.module.ts** a fin de introducir la ruta parametrizada como sigue:

```
{path: 'quienes-somos', component: QuienesSomosComponent},  
  
{path: 'quienes-somos/:id', component: IntegranteComponent},
```

Donde **:id** es el parámetro

Con esto, se crean todas las rutas posibles que empiecen con /quienes-somos/. El nombre **id** es un parámetro que sirve para manipular el componente en función del valor entregado por parámetros en la ruta. Es decir, podemos recoger el valor de la ruta parametrizada y en función de eso mostrar una u otra información.

Por tanto, esa ruta parametrizada será siempre servida por el componente Integrante en este caso, y aceptará rutas diversas. Algunas pueden ser:

- <http://localhost:4200/quienes-somos/1>
- <http://localhost:4200/quienes-somos/2>
- <http://localhost:4200/quienes-somos/3>

Siendo 1, 2 y 3 los identificadores de registro de cada integrante del equipo.

Nota: Es importante agregar que podemos tener más de un parámetro en la ruta dinámica que luego, podemos capturar desde nuestra aplicación.

2. Recuperar los valores de los parámetros. Para ello, y en base al ejemplo que estamos desarrollando, editamos el componente **integrante.component.ts** a fin de:

- a. importar las clases ActivatedRoute y Param:

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

- b. inyectar en el constructor el objeto ActiveRoute:

```
constructor(private rutaActiva: ActivatedRoute) {  
}  
}
```

- c. acceder a los parámetros a través de la suscripción al observable **params** que nos provee el objeto **rutaActiva**:

```
ngOnInit(): void {  
    this.rutaActiva.params.subscribe(  
        (params: Params) => {  
            this.integrante = params.id;  
        }  
    );  
};
```

El objeto Params nos permite acceder a los parámetros por medio del nombre que hemos definido en la ruta parametrizada.

Nota: los observables nos permiten suscribirnos a eventos que se recibirán de manera asíncrona, mediante programación reactiva y manteniendo un alto rendimiento

(<https://desarrolloweb.com/articulos/parametros-rutas-angular.html>)

“Básicamente, en el objeto de tipo ActivatedRoute tenemos una propiedad llamada "params" que es un observable y que nos sirve para suscribirnos a cambios en los parámetros enviados al componente.

Las suscripciones a los cambios en los observadores se realizan mediante el método `subscribe()`. Ese método recibe varios parámetros y el que nos interesa de momento es solo el primero, que consiste en una función callback que se

ejecutará con cada cambio de aquello que se está observando.

En resumen, "this.rutaActiva.params" es el observable y "this.rutaActiva.params.subscribe()" es el método para suscribirnos a los cambios. A subscribe() le enviaremos una función callback, la cual recibirá el nuevo valor, y se ejecutará cada vez que cambie.

La función callback recibirá un objeto de clase Params. Ese objeto también lo vas a tener que importar desde "@angular/route". (<https://desarrolloweb.com/articulos/parametros-rutas-angular.html>)

- d. generar enlaces a rutas enviando parámetros. Para ello, nos valemos del atributo **routerLink**.

```
<a class="btn btn-dark" routerLink="/quienes-somos/1">Ver Perfil</a>
```

En nuestro ejemplo, dicho enlace fue agregado en el componente **QuienesSomosComponent** como se muestra a continuación:

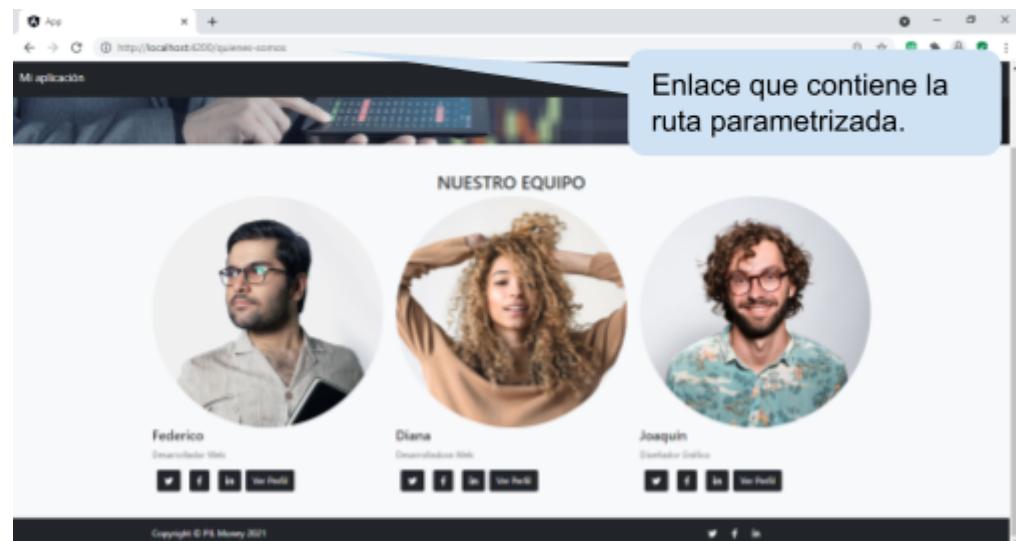


Figura 17: Enlace a ruta parametrizada

De esta manera, podemos agregar parámetros a las rutas, acceder a ellos desde los componentes para realizar alguna acción y/o mostrar las vistas en función de los parámetros.

En nuestra aplicación, podemos observar el comportamiento del sistema de rutas en función de las rutas dinámicas:



Figura 17: Ruta parametrizada <http://localhost:4200/quienes-somos/1>

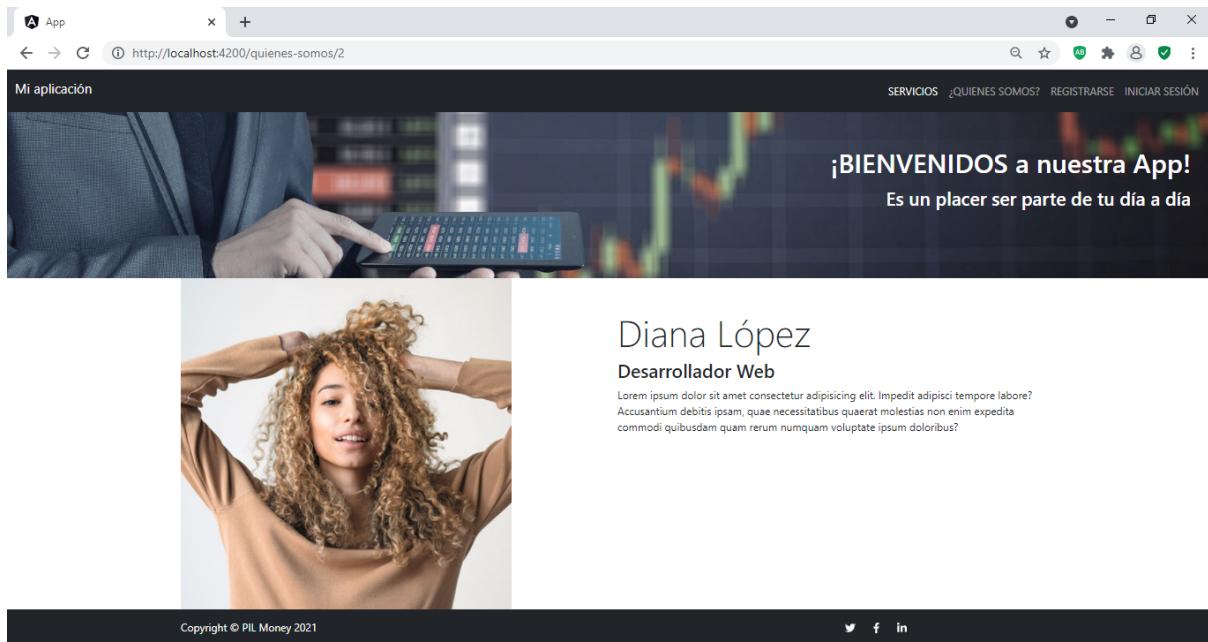


Figura 18: Ruta parametrizada <http://localhost:4200/quienes-somos/2>

Nota: Las rutas dinámicas son muy útiles para mostrar elementos individuales de una lista, o el detalle de un registro. Ej. Usuarios y UsuarioDetalle.

Puedes descargar el fuente del repositorio:

Crear rutas hijas

Las rutas hijas (o anidadas) nos permiten mostrar un componente dentro de la plantilla del componente padre.

En el ejemplo que venimos desarrollando, podríamos configurar una ruta home, la cual nos permite acceder a rutas hijas con las funciones que brinda la billetera: operaciones, transacciones y demás operaciones. Dicha ruta, será accedida sólo luego de que el usuario se autentique (pero esto es parte de otra historia).

Para definir rutas con una o más rutas hijas, ejecutar los siguientes pasos::

1. Editar el archivo **app-routing.module.ts** a fin de introducir la nueva ruta home y sus correspondientes rutas hijas como sigue:

```
{path: 'home', component: HomeComponent,
  children: [
    {path: 'operaciones', component: OperacionesComponent},
    {path: 'transacciones', component: TransaccionesComponent},
    {path: 'criptomoneda', component: CriptomonedaComponent},
  ]}
```

Observa la propiedad **children**, es dónde se debe configurar las rutas hijas.

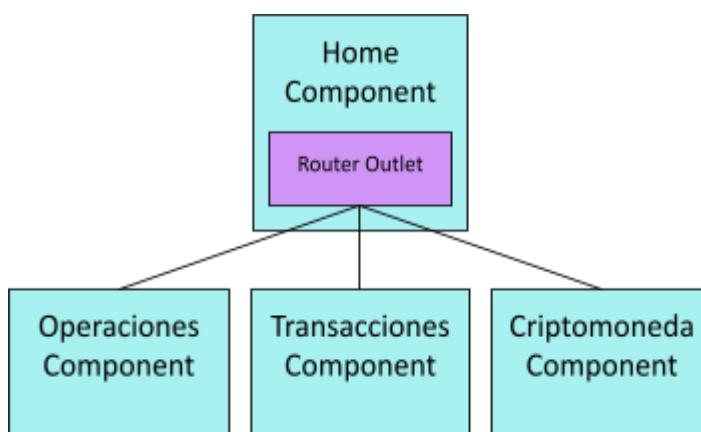


Figura 19: Gráfico del sistema de rutas padre e hijas.

Nota: Previamente debes crear los componentes Operaciones, Transacciones y Criptomoneda (para el ejemplo) y home.

2. Agregar el **<router-outlet></router-outlet>** en el componente padre home y configurar el atributo **routerLink** con las rutas hijas como sigue:

```
<div class="container-fluid">
  <div class="row">
```

```

<div class="col-md-2">
  <nav class="nav flex-column">
    <a class="nav-link active fs-3" routerLink="/home/operaciones">Operaciones</a>
      <a class="nav-link " routerLink="/home/transacciones">Transacciones</a>
      <a class="nav-link " routerLink="/home/criptomoneda">Criptomonedas</a>
  </nav>
</div>

<div class="col-md-10">
  <router-outlet></router-outlet>
</div>
</div>
</div>

```

Si todo salió bien, y accedemos a la ruta <http://localhost:4200/home/operaciones> deberíamos poder observar el anidamiento de componentes (tal como puede observarse en la ruta):

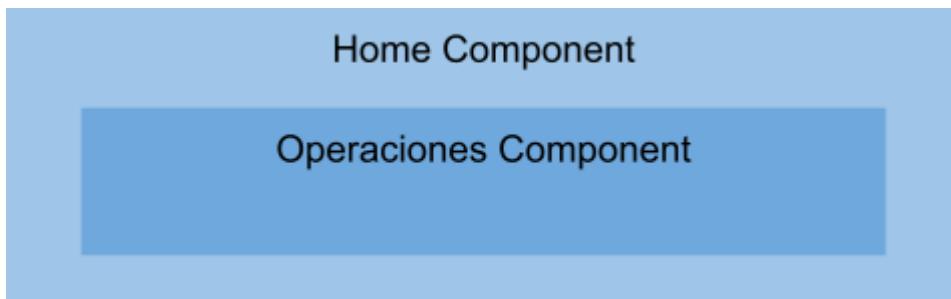


Figura 20: Anidamiento de componentes a partir de las clases hijas

En nuestra aplicación:

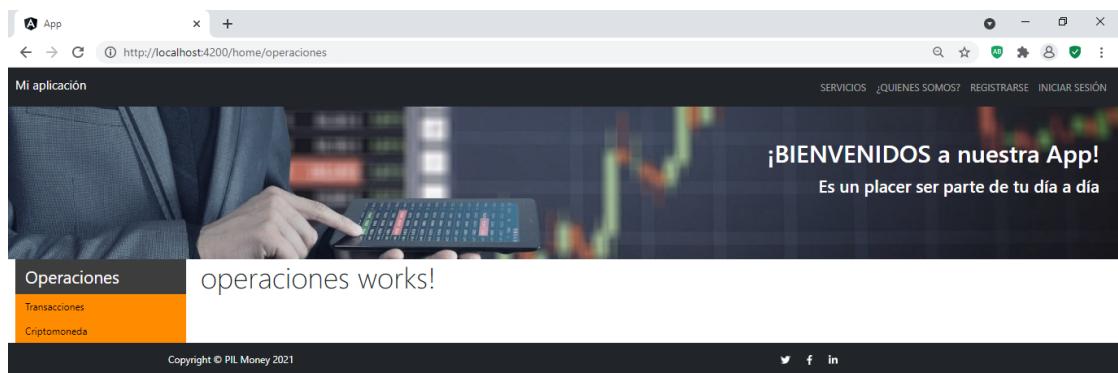


Figura 21: Gráfico del sistema de rutas padre e hijas.

En este punto es importante mencionar, que lo ideal (para este ejemplo) es que tanto la ruta home que a su vez habilita las rutas de operaciones, transacciones y criptomoneda no puedan ser

accedidas por un usuario no esté autenticado. Para ello, Angular nos provee un artefacto llamado **Guard** que se ejecuta antes de cargar la ruta y determina si se puede visualizar o no. Es decir, si la misma es privada (Abordaremos este concepto y más en el módulo 6).

Data Binding

Data Binding, nos abstractea de la lógica get/set asociada a insertar y actualizar valores en el HTML y, de convertir las respuestas de usuario (inputs, clicks, etc) en acciones concretas. Escribir toda esa lógica antes, era tedioso y propenso a errores dado que debíamos trabajar con javascript (o alguna librería como por ej. jquery).

En otras palabras, Angular introduce el concepto de *Data Binding*, el cual nos permite insertar y actualizar los valores en el HTML de una manera muy sencilla.

Tipos de Binding

A continuación se enumeran los tipos de binding, su correspondiente sintaxis y categoría.

Tipo	Sintaxis	Categoría
Interpolation	<code>{{expression}}</code>	One-way
Property		Desde el componente hacia el DOM
Attribute	<code>[target]="expression"</code> <code>bind-target="expression"</code>	
Class		
Style		
Event	<code>(target)="statement"</code> <code>on-target="statement"</code>	One-way Desde el DOM hacia el componente
Two-way	<code>[(target)]="expression"</code> <code>bindon-target="expression"</code>	Two-way En ambos sentidos

Tabla 1: Tipos de Data Binding

Fuente: <https://angular.io/guide/binding-syntax>

Las categorías especifican el flujo de datos tal como se muestra a continuación:

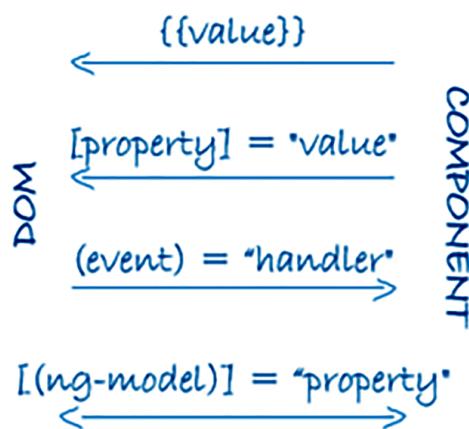


Figura 21: Formas de Data Binding

Fuente: <https://docs.angular.lat/generated/images/guide/architecture/databinding.png>

Observa que los binding types (a diferencia de los de interpolación) poseen un target (nombre) y están entre [], () o ambos [()].

Usa:

- () para enlazar del DOM al componente.
- [] para enlazar desde el componente al DOM.
- [()] para enlazar en ambos sentidos.

El target de un databinding puede ser una propiedad, un atributo, un evento, etc. Cada elemento público de una directiva está disponible para el binding en un template como se puede observar en el siguiente ejemplo:

Tipo	Target	Ejemplo
Property	src (element property) hero (component property) ngClass (directive property)	<pre> <app-hero-detail [hero]="currentHero"></app-hero- detail> <div [ngClass]="'special': isSpecial}"></div></pre>
Event	click (element event) deleteRequest (component event)	<pre><button (click)="onSave()">Save</button> <app-hero-detail (deleteRequest)="deleteHero()"><</pre>

	myClick (directiva event)	<pre>/app-hero-detail> <div (myClick)="clicked=\$event" clickable>click me</div></pre>
Two-way	Event y Property	<pre><input [(ngModel)]="name"></pre>
Attribute	Attribute	<pre><button [attr.aria-label]="help">help</button></pre>
Class	Class property	<pre><div [class.special]="isSpecial">Special</div></pre>
Style	Style property	<pre><button [style.color]="isSpecial ? 'red' : 'green'"></pre>

Fuente: <https://angular.io/guide/binding-syntax>

Interpolation

La interpolación es un mecanismo que nos provee Angular a fin de sustituir una expresión por un valor de cadena en el template (o vista). Es decir que, permite el flujo de datos desde el componente hacia el DOM.

Se utiliza generalmente, cuando recibimos datos del backend, los cuales son visualizados en la vista (de lectura) pero no deben ser modificados dado que existen otros medios para ello disponibles en la aplicación. Ej. el saldo de una cuenta.

Es decir, cuando Angular detecta la interpolación , la evalúa y trata de convertirla en una cadena, para luego renderizar en el template (desde el componente hacia el DOM)

Ejemplo:

```
<p>Esto es una {{interpolacion}}</p>
```

Para entenderlo mejor, modifiquemos la vista del headerComponent de nuestra aplicación a fin de que el mensaje de bienvenida sea enviado del componente a la vista:

Actual vista del headerComponent

```
<header class="fondo">
  <h1 class="mensaje-bienvenida">¡BIENVENIDOS a nuestra App!</h1>
  <h3>Es un placer ser parte de tu día a día</h3>
```

```
</header>
```

Podríamos usar interpolación para modificar el mensaje de bienvenida desde el componente.

Para ello, ejecutar los siguientes pasos:

1. Editar el archivo header.component.ts a fin de agregar una nueva variable que contendrá el mensaje de bienvenida como sigue:

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

@Component({
  selector: 'app-header',
  templateUrl: './header.component.html',
  styleUrls: ['./header.component.css']
})
export class HeaderComponent implements OnInit {
  mensajeBienvenida="¡BIENVENIDOS a nuestra App!";
  constructor() { }
  ngOnInit(): void {
  }
}
```

2. Editar el archivo header.component.ts respetando la sintaxis de la interpolación como sigue:

Interpolación

```
<header class="fondo">
  <h1 class="mensaje-bienvenida">{{mensajeBienvenida}}</h1>
  <h3>Es un placer ser parte de tu día a día</h3>
</header>
```

Si ejecutas el comando ng-serve podrás ver que nada parece haber cambiado. Sin embargo, el texto del mensaje no es aportado por la vista sino por el componente.

Property Binding

Property Binding es un mecanismo que nos permite asignar valores a las propiedades de los elementos HTML y/o directivas presentes en el template.

Se utiliza generalmente, cuando deseamos modificar de manera dinámica los atributos, clases, estilos y demás propiedades de un elemento html. Ej. la url de una imagen.

Para enlazar una propiedad de un elemento HTML debemos encerrar entre [] la propiedad

del elemento HTML que deseamos configurar.

Ejemplos:

```
<img [src]="itemImageUrl">
<span [innerHTML]="propertyTitle"></span>
<p [class.border-danger] ="!propertyValid"></p>
```

Veamos un ejemplo, intentemos modificar la vista del Componente Servicio a fin de que el el texto Servicios tenga un background amarillo.

Para ello, ejecutar los siguientes pasos:

1. Editar el archivo services.component.ts a fin de agregar la variable que deseamos enlazar con el template como sigue:

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

@Component({
  selector: 'app-servicios',
  templateUrl: './servicios.component.html',
  styleUrls: ['./servicios.component.css']
})
export class ServiciosComponent implements OnInit {

  Variable a enlazar

  bgPropertyValid=true;
  constructor() { }

  ngOnInit(): void {
  }
}
```

Nota: Observa que la variable para las clases debe ser booleana dado que, dependiendo de su estado: true o false, se aplicará el estilo.

2. Editar el archivo services.component.html a fin de incluir el enlace de propiedad.

```
<section class="mt-5 id="services">
  <div class="container">
    <div class="text-center">
```

Property Binding

```

<h2 [class.bg-warning]="bgPropertyValid" class="text-uppercase">Servicios</h2>
</div>
<div class="row text-center">
  <div class="col-md-4">
    <span class="fa-stack fa-4x">
      <i class="fas fa-circle fa-stack-2x text-warning"></i>
      <i class="fas fa-laptop fa-stack-1x fa-inverse"></i>
    </span>
    ...
  </div>
</div>
  
```

Si ejecutamos ng-serve, observamos el cambio en el ServiciosComponent

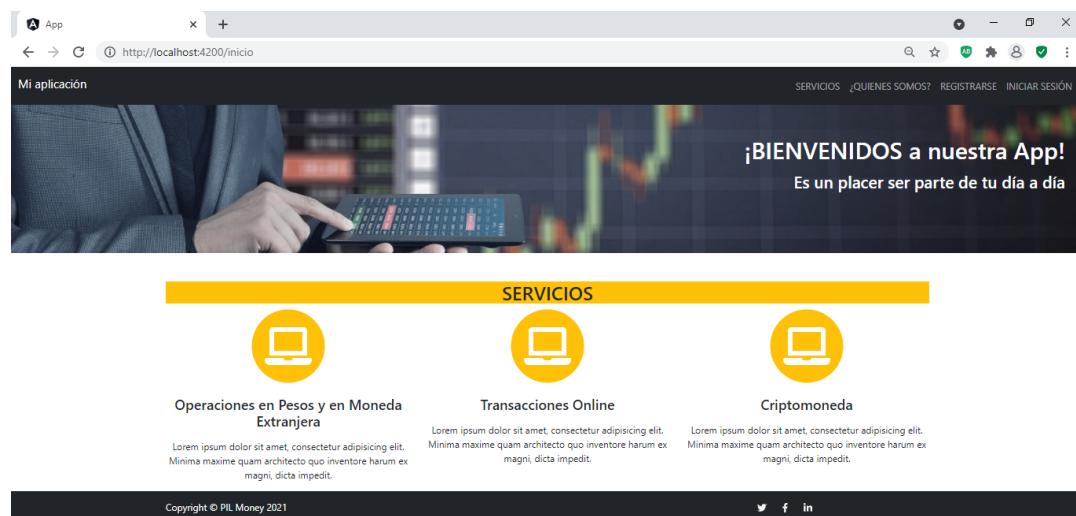


Figura 22: Background modificado en base al binding de tipo propiedad (property binding)

Event Binding

Es el mecanismo de data binding que nos permite trabajar con los eventos del DOM. El mismo permite además el flujo de datos del DOM hacia el componente.

Se usan generalmente en los formularios dado que permiten recuperar los datos que ingresó el usuario para así luego manipularlos y/o enviarlos al backend.

La sintaxis de un event binding es relativamente simple, basta con decorar el nombre del evento entre **()**. De esta manera, nos suscribimos al mismo. Además, debemos proporcionar

el método a ejecutar cuando el evento suceda.

Ejemplo:

```
<button (click)="onSaludar()">Click aquí!</button>
```

Nota: es posible también subscribirnos al evento usando esta otra sintaxis:

```
<button on-click="onSaludar()">Pulse aquí</button>
```

De esta manera, el componente sabe que cuando el evento (click) se lance, se deberá ejecutar método onSave().

Nota: No te olvides de crear el método onSaludar()

¿Qué eventos podemos utilizar? Puedes econtrarlos en
https://www.w3schools.com/jsref/dom_obj_event.asp

Veamos el ejemplo que simplemente muestre un alerta saludar. Para ello, ejecutar los siguientes pasos:

1. Editar el archivo servicios.component.html a fin de agregar un botón y subscribirnos al evento click como sigue:

```
<button class="btn btn-warning" (click)="onSaludar()">Clic  
Aquí!</button>
```

2. Editar el archivo servicios.component.ts a fin de crear el método onSaludar como sigue:

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

@Component({
  selector: 'app-servicios',
  templateUrl: './servicios.component.html',
  styleUrls: ['./servicios.component.css']
})
export class ServiciosComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }

}
```

```
{
}

Método que responde al
evento

onSaludar(): void {
    alert("Hola mundo!!");
}
}
```

- Ejecutar el comando ng serve, para ver y evaluar la aplicación en tiempo de ejecución.

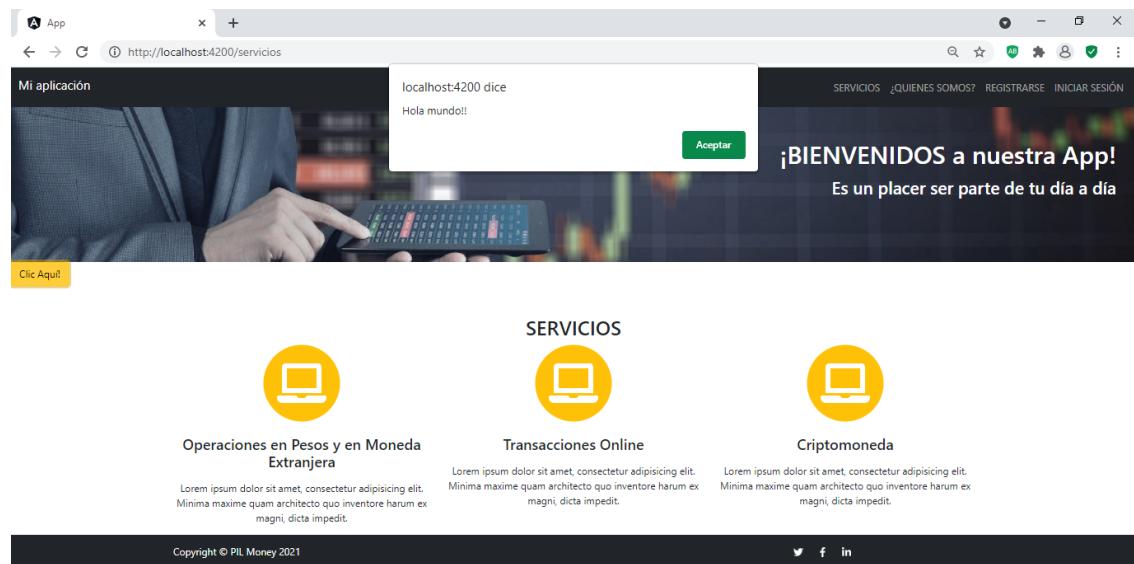


Figura 23: Event Binding

Two way binding

A menudo, se suele utilizar en conjunto Property Binding y Event Binding como se muestra en la siguiente imagen:

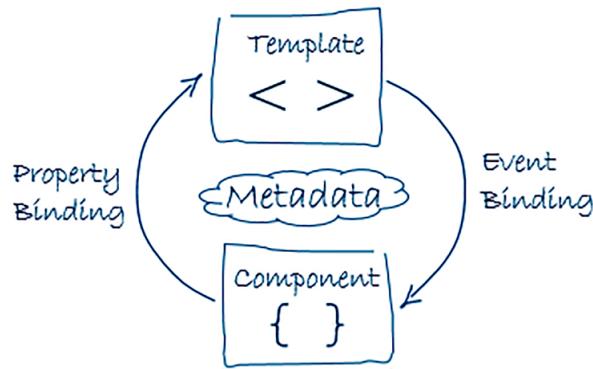


Figura 24: Two Way Binding

Fuente: <https://docs.angular.lat/generated/images/guide/architecture/component-databinding.png>

Sintaxis:

[(event)**]=<<expresión>>**

Ejemplo:

```
<p [(size)]="fontSizePx"></p>
```

En este tipo de binding, cualquier cambio relacionado con los datos en el modelo siempre afecta a las vistas y los cambios relacionados en las vistas también se ven reflejados en el modelo. Es decir que los cambios en cualquier lado de la aplicación (modelo o vista) se ven reflejados en el extremo contrario de manera inmediata y automática.

Expresiones

Angular nos permite trabajar con expresiones. Éstas se resolverán previo a devolver el resultado exactamente dónde la expresión se invocó.

Las mismas pueden:

- Escribirse entre llaves dobles: {{ expression }},
- En el interior de una directiva: ng-bind=" expression " .

Nota: las expresiones de angular son muy similares a las expresiones de JavaScript. Las mismas pueden contener literales, operadores y variables. (http://www.w3bai.com/es/angular/angular_expressions.html)

Ejemplo de expresiones:

`{{propiedad_del_componente}}`

`{{ método_del_componente}}`

`{{ 2020 + 1 }}`

`{{ ! valorBoleano}}}`

Las expresiones son utilizadas al momento de realizar modificaciones en los elementos HTML presentes en los componentes de la aplicación y lo hacen en algunos casos mediante directivas. Concepto que abordaremos a continuación.

Puedes ver y ejecutar el código de un ejemplo sencillo en

http://www.w3bai.com/es/angular/tryit.php?filename=try_ng_expression_2

Directivas

Son un mecanismo angular que nos permite manipular el DOM por lo que, podemos modificar el comportamiento de los elementos html, atributos, demás propiedades.

En otras palabras, las directivas son simplemente instrucciones interpretadas por el compilador, quién se encarga de recorrer el documento, localizarlas y ejecutar los comportamientos que se especifican en las mismas.

Directivas más comunes en Angular:



Figura 25: Tipos comunes de directivas.

Directivas de Componente

Como vimos previamente, son las más utilizadas en Angular y son capaces de modificar:

- La vista HTML

- Un archivo .ts (TypeScript?) que define el comportamiento.
- Un selector CSS que define cómo es utilizado en el template.
- Opcionalmente un archivo .css que define el estilo del componente y un archivo .ts para las pruebas unitarias.

(<https://docs.angular.lat/guide/built-in-directives>).

Las directivas de atributos más comunes son:

- **ngClass**. Permite agregar o remover clases CSS en función de un estado o expresión de manera dinámica.
- **ngModel**. Permite Two-way binding (Desde/Hacia el DOM) de manera dinámica.
- **ngStyle**. Permite agregar o eliminar estilos de componente.

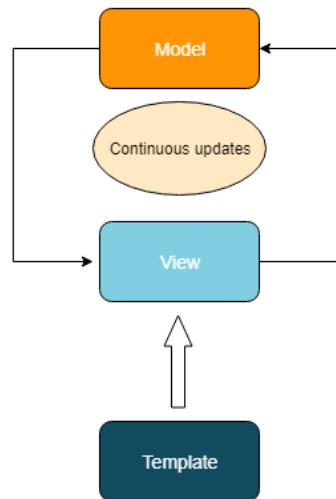


Figura 26: Representación gráfica de Binding en dos sentidos

Fuente: <https://www.altexsoft.com/media/2018/10/One-and-two-way-data-bind.png>

A continuación se describen las directivas ngClass y ngModel:

Directiva: ngClass

Permite establecer dinámicamente clases CSS en los elementos HTML por medio de las expresiones.

Ejemplo:

Supongamos que queremos cambiar el estado de un botón según esté apagado o encendido: OFF/ON:

En la vista:

```
<button class="btn" [ngClass]="{on: estadoPositivo, off:  
!estadoPositivo}" (click)="cambiarEstado('{{texto}}')</button>
```

En el archivo ts:

```
estadoPositivo: boolean = true;  
texto:string="si";  
  
cambiarEstado()  
{  
    this.estadoPositivo = !this.estadoPositivo;  
    if (this.estadoPositivo)  
        {this.texto="si";}  
    else  
        {this.texto="no";}  
}
```

Dónde on y off son clases CSS y, de acuerdo al estado de la variable estadoPositivo, el elemento button implementará una u otra clase.

Figura 27: ejemplo de ngClass

Directiva ngModel

Es un enlace que permite el flujo de datos entre la variable (componente) y el elemento de la vista (DOM) en ambos sentidos.

Ejemplo:

Si deseamos que el valor de la variable se muestre en un input type (one way binding):

```
<input type="text" [ngModel]="nombre">
```

Si deseamos que el valor de la variable se muestre en un input type pero además deseamos acceder luego al dato que el usuario ingresó (two way binding):

```
<input type="text" [(ngModel)]="nombre">
```

Nota: Se requiere implementar **FormsModule** y declarar la variable nombre en el archivo .ts

Directivas de estructura

Permiten manipular la estructura de la vista agregando o eliminando elementos de una manera muy sencilla.

A continuación se describen las directivas ngIf, ngFor y ngSwitch:

Directiva ngFor

***ngFor:** es una directiva repetidora que permite recorrer un array y para cada uno de sus elementos replicar una cantidad de elementos en el DOM.

Sintaxis:

***ngFor="expresión"**

Ejemplo:

Continuando con nuestro ejemplo y para observar cómo funciona el ngFor, ahora vamos a crear una tabla de movimientos en nuestra aplicación.

Para ello, ejecutar los siguientes pasos:

1. Crear el componente movimientos. Para ello, ejecutar: **ng g c pages/movimientos --skip-tests**
2. En el archivo **movimientos.component.ts** crear una variable movimientos como sigue (por ahora hardcodeada, luego deberemos reemplazar con los datos que vienen del backend).

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

@Component({
  selector: 'app-movimientos',
  templateUrl: './movimientos.component.html',
  styleUrls: ['./movimientos.component.css']
})
export class MovimientosComponent implements OnInit {

  Variable movimientos

  movimientos=[{operacion:"Extracción",monto:1500}, {operacion:"Depósito", monto:1520}];

  constructor() { }

  ngOnInit(): void {
  }
}
```

3. En el archivo **movimientos.component.html** completar las filas de la tabla haciendo uso del ngFor:

```
<h1 class="display-5">Últimos Movimientos</h1>
<table class="table">
  <thead>
    <th>Operación</th>
    <th>Monto</th>
  </thead>
  <tbody>
```

```
directiva ngFor

<tr *ngFor="let element of movimientos" >
    <td>{{element.operacion}}</td>
    <td>{{element.monto|currency}}</td>
</tr>
</tbody>
</table>
```

- Ejecutar el comando `ng serve` (si no está corriendo el servidor), y luego ir a <http://localhost:4200/home/movimientos> para ver y evaluar la aplicación en tiempo de ejecución.

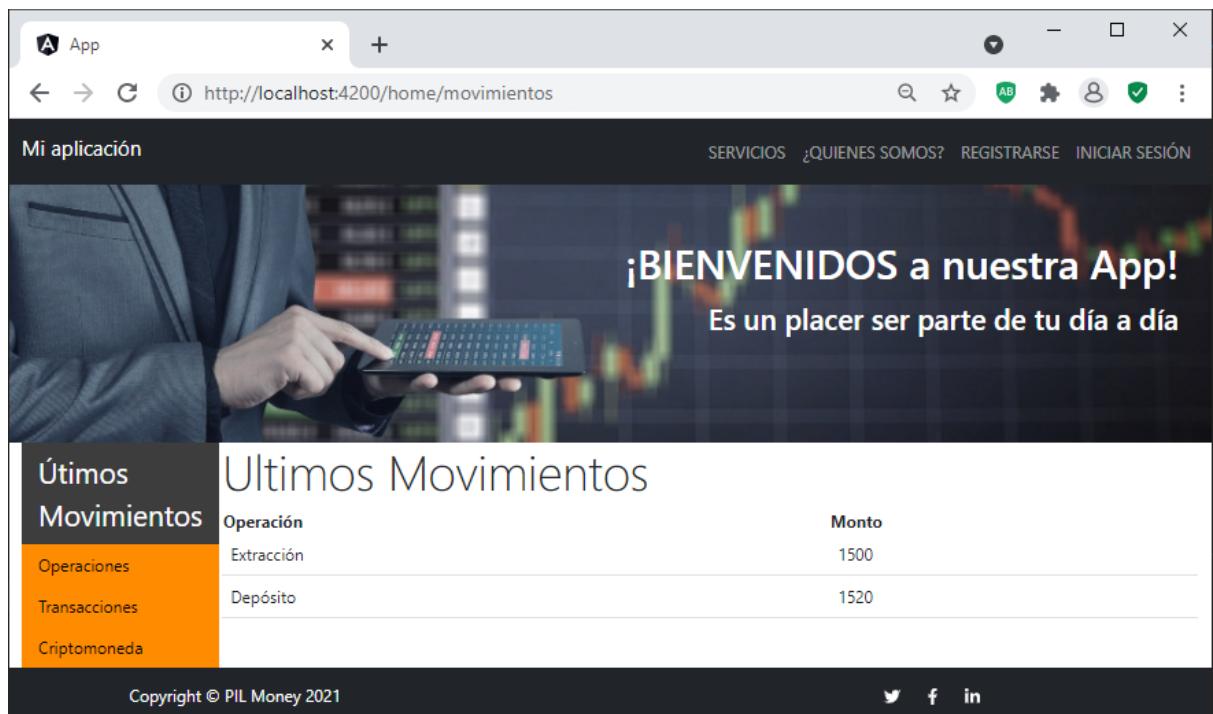


Figura 28: Uso del `ngFor` - Tabla movimientos

Nota: No te olvides que para que funcione hay que configurar las rutas y el enlace a Últimos Movimientos en la barra de navegación del home.

Directiva `nglf`

***ngIf:** permite ocultar o mostrar un elemento html a partir de una expresión.

Sintaxis:

`*ngIf="expresion"`

Ejemplo:

Continuando con nuestro ejemplo y para observar cómo funciona el `ngIf`, ahora vamos a crear variable que nos permita ver esa tabla de movimientos.

Para ello, ejecutar los siguientes pasos:

1. En el archivo `movimientos.component.ts` crear una variable booleana que utilizaremos luego para mostrar o no la tabla movimientos:

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

@Component({
  selector: 'app-movimientos',
  templateUrl: './movimientos.component.html',
  styleUrls: ['./movimientos.component.css']
})
export class MovimientosComponent implements OnInit {

  Variable booleana

  mostrarMovimientos: boolean=true;
  movimientos=[{operacion:"Extracción",monto:1500}, {operacion:"Depósito", monto:1520}];
  constructor() { }

  ngOnInit(): void {
  }
}
```

2. En el archivo `movimientos.component.html` agregar la directiva `ngIf` para mostrar o no la tabla:

```
<h1 class="display-5">Últimos Movimientos</h1>

  Directiva ngIf

<table *ngIf="mostrarMovimientos" class="table">
  <thead>
    <th>Operación</th>
    <th>Monto</th>
  </thead>
  <tbody>
    <tr *ngFor="let element of movimientos" >
      <td>{{element.operacion}}</td>
      <td>{{element.monto}}</td>
    </tr>
  </tbody>
</table>
```

Luego, dependiendo de la variable se mostrará o no la tabla.

Directiva ngSwitch

***ngSwitch:** Muestra u oculta un elemento entre varios dependiendo de una condición específica.

Sintaxis:

[ngSwitch] = "expresión" //se define en el contenedor

*ngSwitchCase="valor numérico" //se define en el elemento a mostrar

***ngSwitchDefault** //se define en el elemento por defecto (si ninguna de las anteriores se cumple)

Ejemplo:

Continuando con nuestro ejemplo, podemos observar el componente integrante creado previamente.

El mismo cuenta con la directiva ngSwitch para mostrar el perfil de uno u otro integrante dependiendo del parámetro recibido en la ruta:

```

Directiva  
ngSwitch

<div [ngSwitch]="id">
  <div *ngSwitchCase="1" class="container">
    <div class="row">
      <div class="col">
        
      </div>
      <div class="col">
        <h1 class="mt-5 display-4 ">Federico Gonzales</h1>
        <h3>Desarrollador Web</h3>
        <p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Impedit adipisci tempore labore? Accusantium debitis ipsam, quae necessitatibus quaerat molestias non enim expedita commodi quibusdam quam rerum numquam voluptate ipsum doloribus?</p>
      </div>
    </div>
  </div>
</div>

Directiva ngSwitchCase permite  
mostrar o no el componente  
dependiendo de la expresión.

<div *ngSwitchCase="2" class="container">
  <div class="row">
    <div class="col">
      
    </div>
    <div class="col">
      <h1 class="mt-5 display-4 ">Diana López</h1>
      <h3>Desarrollador Web</h3>
    </div>
  </div>
</div>
```

```
<p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Impedit adipisci tempore labore? Accusantium debitibus ipsam, quae necessitatibus quaerat molestias non enim expedita commodi quibusdam quam rerum numquam voluptate ipsum doloribus?</p>
</div>
</div>
</div>
<div *ngSwitchCase="3" class="container">
<div class="row">
<div class="col">

</div>
<div class="col">
<h1 class="mt-5 display-4 ">Joaquín Aguirre</h1>
<h3 class="display-5">Diseñador</h3>
<p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Impedit adipisci tempore labore? Accusantium debitibus ipsam, quae necessitatibus quaerat molestias non enim expedita commodi quibusdam quam rerum numquam voluptate ipsum doloribus?</p>
</div>
</div>
</div>
<p *ngSwitchDefault>...</p>
</div>
```

Para verlo funcionando en nuestra aplicación, ejecutar el comando ng serve (si no está corriendo el servidor) e ir a <http://localhost:4200/quienes-somos/1> (el último valor es el que determinará qué integrante del equipo mostrar.

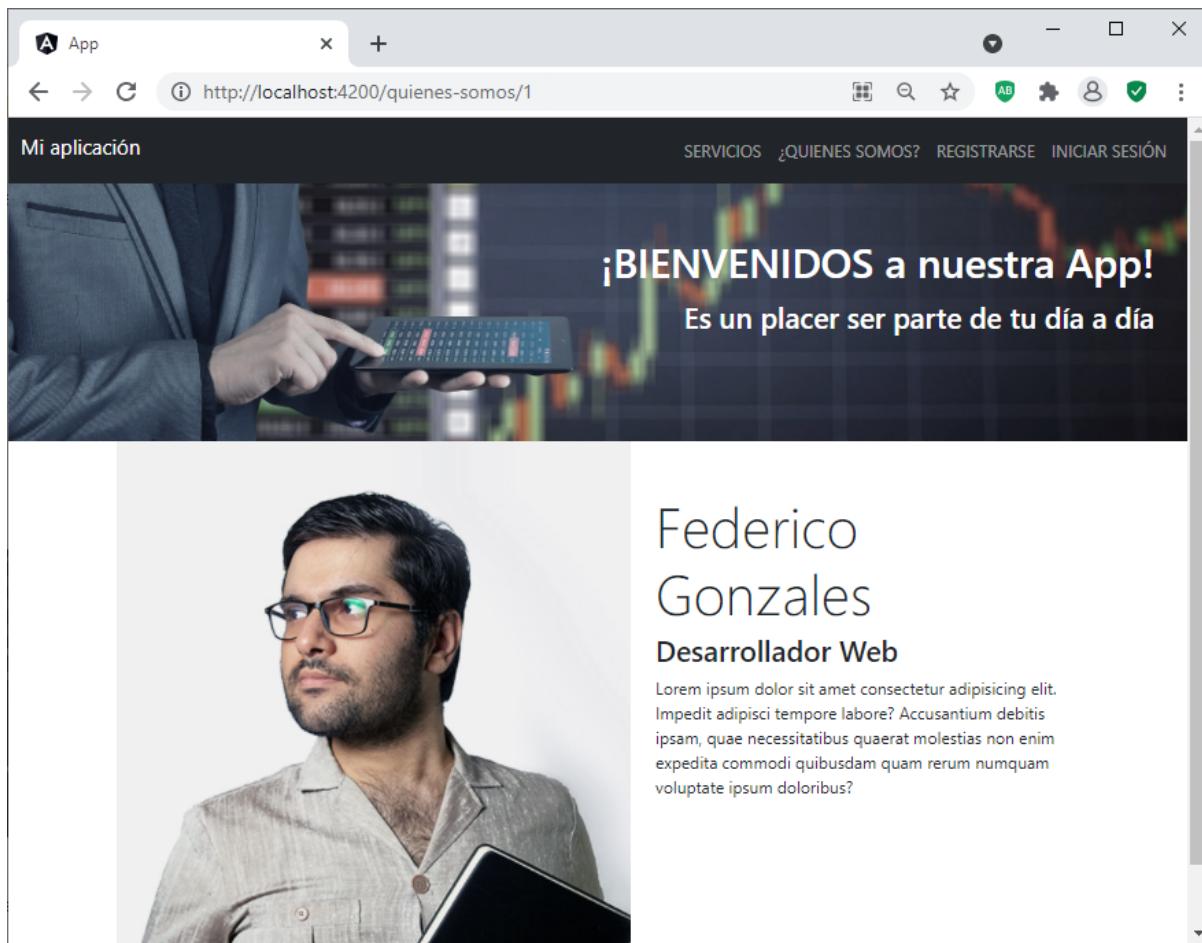


Figura 29: Uso del ngSwitch - Mostrar Perfil

Pipes

Es el mecanismo que nos permite transformar los valores de entrada en valores de salida. Es decir, nos permite especificar el formato.

Los pipes se declaran una sola vez y pueden ser utilizados en toda la aplicación.

Angular provee pipes integrados para las transformaciones de datos, las siguientes son pipes integrados:

- **DatePipe**, cambia el valor de fecha.
- **UpperCasePipe**, transforma texto en Mayúscula.
- **LowerCasePipe**, transforma el texto en minúscula.
- **CurrencyPipe**, transforma un número en una cadena de moneda de acuerdo a ciertas reglas.
- **DecimalPipe**, transforma un número en una cadena con punto decimal.

- **PercentPipe**, Transforma un número en una cadena de porcentaje.

Nota: Puedes obtener la lista completa de pipes en <https://angular.io/api/common#pipes>

Ejemplo:

Continuando con nuestro ejemplo, vamos a modificar la tabla de movimientos a fin de que los montos se muestren en formato moneda. Además agregaremos la fecha con su formato en español arriba de la tabla.

Para ello, ejecutar los siguientes pasos:

1. En el archivo movimientos.component.ts crear una nueva variable para la fecha:

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

@Component({
  selector: 'app-movimientos',
  templateUrl: './movimientos.component.html',
  styleUrls: ['./movimientos.component.css']
})
export class MovimientosComponent implements OnInit {

  hoy= new Date();
  mostrarMovimientos: boolean=true;
  movimientos=[{operacion:"Extracción",monto:1500}, {operacion:"Depósito", monto:1520}];
  constructor() { }
  ngOnInit(): void {
  }
}
```

Variable de fecha

2. En el archivo movimientos.component.html agregar los pipes:

```
<h1 class="display-5">Últimos Movimientos</h1>



Pipe para la fecha,  
establece el formato  
día/mes/año



<h2>{{hoy|date: "d/M/yy"}}</h2>
<table *ngIf="mostrarMovimientos" class="table">
  <thead>
    <th>Operación</th>
    <th>Monto</th>
  </thead>
```

```

<tbody>
  <tr *ngFor="let element of movimientos" >
    <td>{{element.operacion}}</td>

    Pipe para los montos de
    dinero, establece el $ y
    dos decimales

    <td>{{element.monto|currency}}</td>
  </tr>
</tbody>
</table>

```

- Ejecutar el comando `ng serve` (si no está corriendo el servidor), y luego ir a <http://localhost:4200/home/movimientos> para ver el funcionamiento de los pipes.

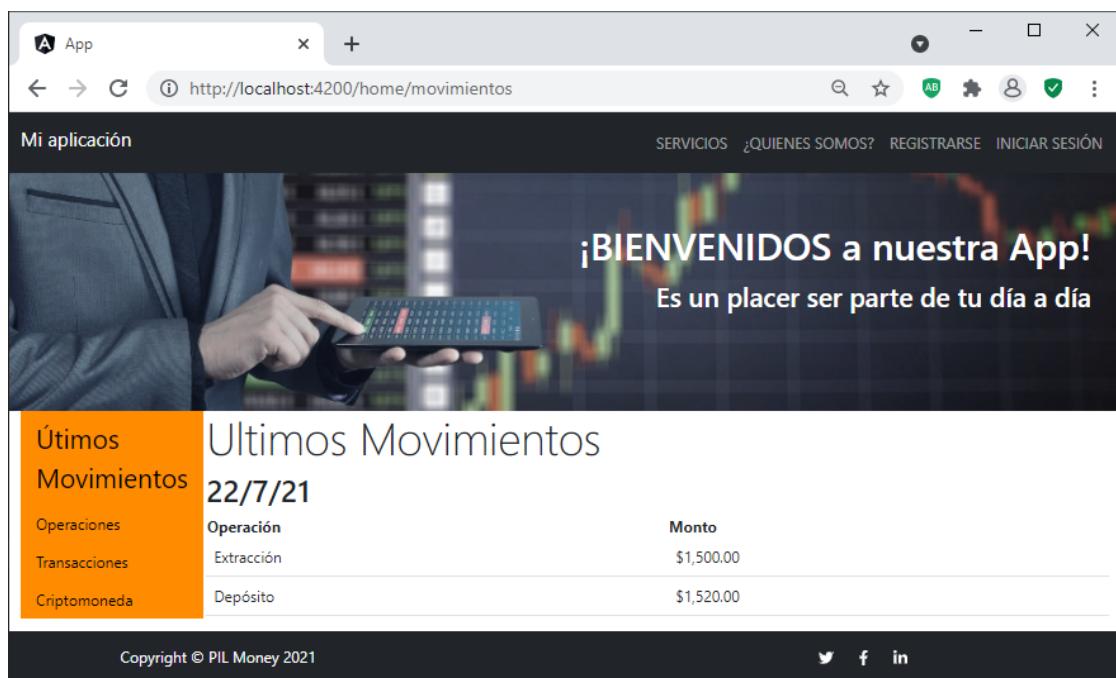


Figura 30: Uso de Pipes

Formularios en angular

La utilización de formularios en aplicaciones es muy común, permiten al usuario ingresar datos a la base de datos realizar el login, registrarse, entre otras acciones muy necesarias.

Angular nos provee dos tipos de formularios:

- Basados en plantilla (Template driven):** proporcionan un enfoque basados en directivas. Los mismos, hacen foco en escenarios simples y no son tan reusables.
- Reactivos (Model Driven):** proporcionan un enfoque basado en modelos por lo que

son más robustos, escalables, reusables y testeables.

	Reactivos	Basados en plantillas
Configuración	Explícita. Se crea en la clase del componente.	Implícita. Se crea a través de directivas.
Modelo de Datos	Estructurado e inmutable	No estructurado y mutable
Flujo de datos (entre vista y el modelo)	Síncrono	Asíncrono
Validación de Formularios	Por medio de funciones	Por medio de directivas

Tabla: Comparación formularios reactivos y formularios de plantilla

Fuente: <https://docs.angular.lat/guide/forms-overview>

Es importante agregar, que en este documento se abordarán los formularios reactivos dado que son mucho más robustos, escalables, mantenibles y testeables.

Formularios reactivos

Proveen un approach explícito e inmutable para administrar el estado del formulario cuyos valores cambian con el tiempo. Cada cambio de estado en el formulario retorna un nuevo valor permitiendo mantener la integridad con el modelo.

Además, cada elemento de la vista está directamente enlazado al modelo mediante una instancia de FormControl. Las actualizaciones de la vista al modelo y del modelo a la vista son síncronas y no dependen de la representación en la Interfaz de Usuario.

Dichos formularios están basados en flujos de datos del tipo **Observable**, donde cada entrada/valor puede ser accedido de manera asíncrona.

Nota: el concepto de **Observable** se estará abordando más adelante en esta sección.

Para que Angular, intérprete nuestros formularios como reactivos, debemos importar el módulo de Formularios Reactivos en el archivo **app.module.ts** o en los módulos dónde necesites trabajar dichos formularios.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
```

```
import { ReactiveFormsModule } from '@angular/forms'; ←  
  
@NgModule({  
  declarations: [  
    AppComponent  
  ],  
  imports: [  
    BrowserModule,  
    AppRoutingModule,  
    ReactiveFormsModule ←  
  ],  
  providers: [],  
  bootstrap: [AppComponent]  
})  
export class AppModule {}
```

Nota: Tanto los formularios reactivos como los basados en plantillas realizan un seguimiento de los cambios de valor entre los elementos de entrada del formulario con los que interactúan los usuarios y los datos del formulario en su modelo de componente. Los dos enfoques comparten bloques de construcción subyacentes, pero difieren en cómo se crean y administran las instancias comunes de control de formularios.

(<https://docs.angular.lat/guide/forms-overview>)

FormControl

Es la unidad más pequeña de un formulario reactivo.

Ejemplo: un cuadro de texto, un calendario, una lista desplegable, etc.

Para configurar un FormControl reactivo, ejecutar los siguientes pasos:

1. Importar la clase FormControl:

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

2. Declarar el nuevo form control en el archivo .ts:

```

@Component({
  selector: 'app-iniciar-sesion',
  templateUrl: './iniciar-sesion.component.html',
  styleUrls: ['./iniciar-sesion.component.css']
})
export class IniciarSesionComponent implements OnInit {
  mail= new FormControl('',[],[]);
  constructor() { }
  ngOnInit(): void {
  }
}
  
```



3. En el archivo .html, enlazar el form control al template utilizando el property binding [formControl]:

Sintaxis:

[formControl] = "variable Form Control"

```

<form class="m-2">
  <div class="form-group">
    <label for="email">Email address:</label>
    <input type="email" [formControl]="mail"
class="form-control" placeholder="Enter email"
id="email">
  </div>
  ...
  
```

De esta manera, el control de formulario “mail” y el input type se comunican entre sí.
 La vista refleja los cambios en el modelo y el modelo refleja los cambios en la vista.

Form Builder

Dado que crear instancias de FormControl manualmente puede llegar a ser repetitivo, Angular nos provee el servicio FormBuilder.

Para usar este servicio, ejecutar los siguientes pasos:

1. Importar el servicio FormBuilder
2. Inyectar el servicio FormBuilder
3. Generar el contenido del formulario, creando el form group con sus respectivos form

controls y enlazando al template.

Ejemplo:

Continuando con nuestro ejemplo, vamos a crear el formulario de inicio de sesión.

Para ello, ejecutar los siguientes pasos:

1. En el archivo **iniciar-sesion.component.ts** importar la clase:

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

2. Inyectar en el constructor el formBuilder:

```
constructor(private formBuilder: FormBuilder){ }
```

3. En el constructor crear el grupo de controles para el formulario

```
this.form= this.formBuilder.group(
{
  password:[ '', []],
  mail:[ '', []]
})
```

4. En el archivo **iniciar-sesion.component.html** enlazar el form builder utilizando el property binding **[formGroup]** y los atributos **formControlName** con sus respectivos form control:

Enlace al servicioFormBuilder

```
<form [formGroup]="form" >
```

Enlace al formControl

```

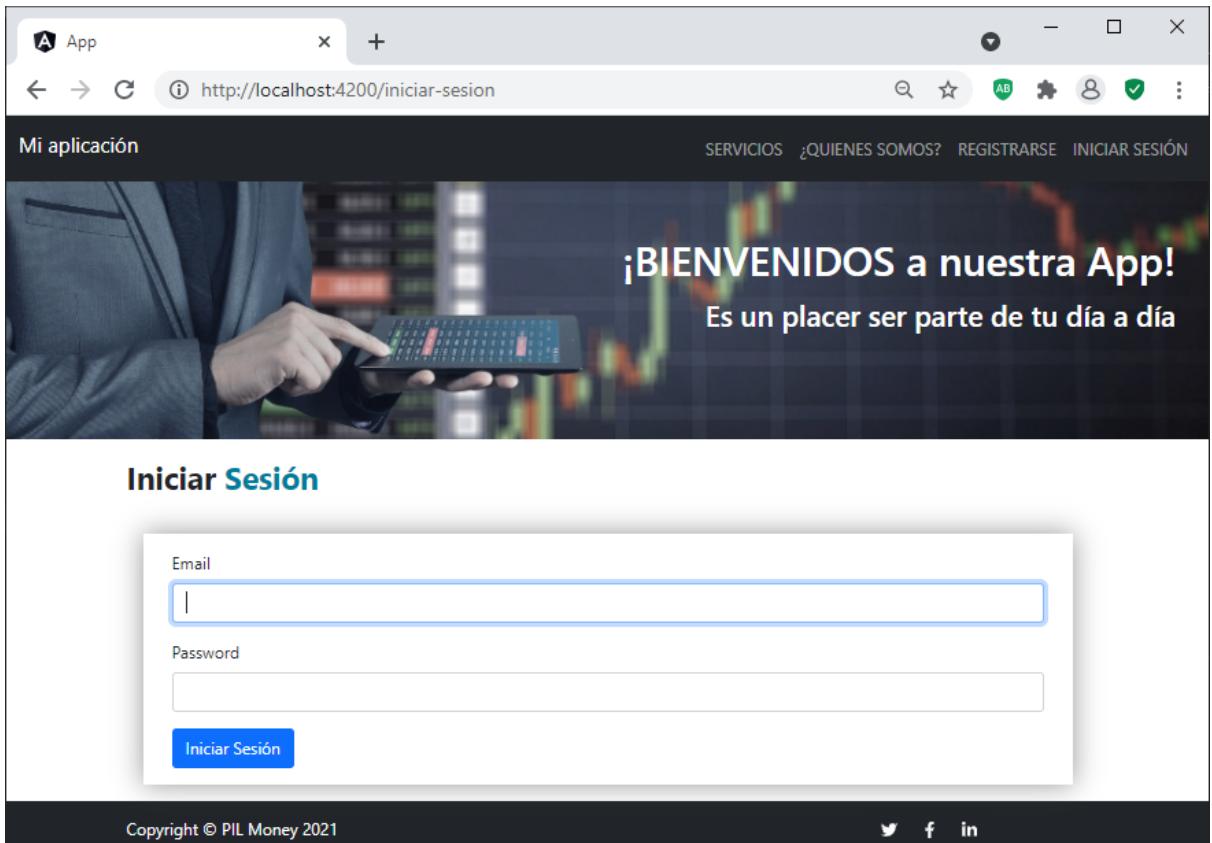
<div class="m-3">
  <label for="exampleInputEmail1" class="form-label">Email</label>
  <input type="email" class="form-control" formControlName="mail">
</div>

<div class="m-3">
  <label for="exampleInputPassword1" class="form-label">Password</label>
  <input type="password" class="form-control" formControlName="password">
</div>

<div class="m-3">
  <button type="submit" class="btn btn-primary">Iniciar Sesión</button>
</div>
```

```
</form>
```

- Ejecutar el comando ng serve (si no está corriendo el servidor), y luego ir a <http://localhost:4200/iniciar-sesion/> para ver el formulario.



The screenshot shows a web browser window with the URL <http://localhost:4200/iniciar-sesion>. The page has a dark header with the text 'Mi aplicación', 'SERVICIOS', '¿QUIENES SOMOS?', 'REGISTRARSE', and 'INICIAR SESIÓN'. Below the header is a large image of a person in a suit holding a smartphone. Overlaid on the image is the text '¡BIENVENIDOS a nuestra App!' and 'Es un placer ser parte de tu día a día'. The main content area is titled 'Iniciar Sesión' and contains two input fields: 'Email' and 'Password', followed by a blue 'Iniciar Sesión' button. At the bottom of the page, there is a copyright notice 'Copyright © PIL Money 2021' and social media icons for Twitter, Facebook, and LinkedIn.

Figura 31: Formulario de Inicio de Sesión

Puedes descargar el fuente de:

Validaciones de Formularios

Angular nos provee la clase **Validators**, la cual contiene una serie de métodos estáticos que nos permiten validar las entradas de datos comunes tales como el formato del email, valores numéricos, máximos y mínimos, cantidad mínima y máxima de caracteres, entre otros.

Métodos estáticos que provee angular:

- min()**: realiza el control de que un número sea mayor o igual al número ingresado.
- max()**: realiza el control de que un número sea menor o igual al número ingresado.
- required()**: valor requerido.
- required True()**, valor requerido como verdadero. Utilizado para la casilla de

verificación.

- **email()**: valor requerido pase la verificación de valor de mail.
- **minlength()**: requiere que el valor ingresado sea de una longitud mínima según la cantidad de caracteres requeridos.
- **maxlength()**: requiere que el valor ingresado sea de una longitud máxima según la cantidad de caracteres requeridos.
- **pattern()**: requiere que el valor ingresado coincida con un patrón de expresiones regulares.
- **nullValidator()**: no realizará ninguna validación.
- **compose()**: composición de varias validaciones en una sola función.
- **composeAsync()**: composición de varias validaciones asincrónicas en una sola función.

(<https://docs.angular.lat/api/forms/Validators>)

Tipo de validaciones

Angular nos provee dos tipos de validaciones:

- **Validadores sincrónicas**: consisten en funciones síncronas que toman una instancia de control y devuelven inmediatamente un conjunto de errores de validación o un valor nulo.
- **Validadores asíncronas**: consisten en funciones asíncronas que toman una instancia de control y devuelven una Promesa u Observable que luego emite un conjunto de errores de validación o nulos.



Figura 32: Nomenclatura de las validaciones

Validaciones con Form Builder

Siguiendo nuestro ejemplo de inicio de sesión, veamos los pasos para configurar las validaciones de nuestro formulario.

Para ello, ejecutar los siguientes pasos:

1. Importar la clase **Validators**

```
import { Validators } from '@angular/forms';
```

2. En el archivo **iniciar-sesion.component.ts** escribir las validaciones.

En nuestro caso, vamos a trabajar con validaciones síncronas por lo que configuraremos entre comas las requeridas para nuestros form controls sabiendo que:

- a. **mail**, debe ser requerido para el inicio de sesión así como también debe respetar el formato propio del mail.
- b. **password**, debe ser también requerido para el inicio de sesión y debe tener al menos 8 dígitos.

En el código:

```
this.form= this.formBuilder.group({
  password:[ '',[Validators.required, Validators.minLength(8)]],
  mail:[ '', [Validators.required, Validators.email]]
})
```

Luego, crear las propiedades Password y Mail (getter) para acceder luego desde la vista:

```
get Password()
```

```
{
    return this.form.get("password");
}

get Mail()
{
    return this.form.get("mail");
}
```

3. A fin de lograr una mejor experiencia de usuario, en el archivo **iniciar-sesion.component.html** crear un div inmediatamente después del input type que contendrá los mensajes al usuario. Los mismos, se mostrarán al usuario dependiendo de la validación. Para ello, utilizaremos las directivas ***ngIf**.

```
<form [formGroup]="form" >
  <div class="m-3">
    <label for="exampleInputEmail1" class="form-label">Email</label>
    <input type="email" class="form-control" formControlName="mail">

      Mail, propiedad (getter) declarada en
      el componente. Nos permite acceder
      al formControl.

    <div *ngIf="Mail?.errors && Mail?.touched">
      hasError (). Método que
      permite evaluar de qué
      error se trata.

      <p *ngIf="Mail?.hasError('required')" class="text-danger">
        El email es requerido.
      </p>
      <p *ngIf="Mail?.hasError('email')" class="text-danger">
        El formato del email debe ser válido.
      </p>
    </div>
  </div>

  <div class="m-3">
    <label for="exampleInputPassword1" class="form-label">Password</label>
    <input type="password" class="form-control" formControlName="password">

      error, propiedad que
      especifica si el form control
      tiene errores de validación.

      touched, propiedad que
      especifica si el form control
      fue tocado.

    <div *ngIf="Password?.errors && Password?.touched">
      <p *ngIf="Password?.hasError('required')" class="text-danger">
        El password es requerido.
      </p>
      <p *ngIf="Password?.errors?.minlength
      " class="text-danger">
        El password debe ser de 8 o más caracteres.
      </p>
    </div>
```

```

        </div>
    <div class="m-3">
        <button type="submit" class="btn btn-primary">Iniciar Sesión</button>
    </div>
</form>

```

Observa las propiedades:

- **.touched.** Propiedad booleana que especifica si el form control fue tocado por el usuario.
 - **.errors.** Propiedad booleana que especifica si el formulario tiene errores (falla una o más validaciones).
4. Ejecutar el comando `ng serve` (si no está corriendo el servidor), y luego ir a <http://localhost:4200/iniciar-sesion/> para ver el formulario. Luego, comprueba las validaciones.

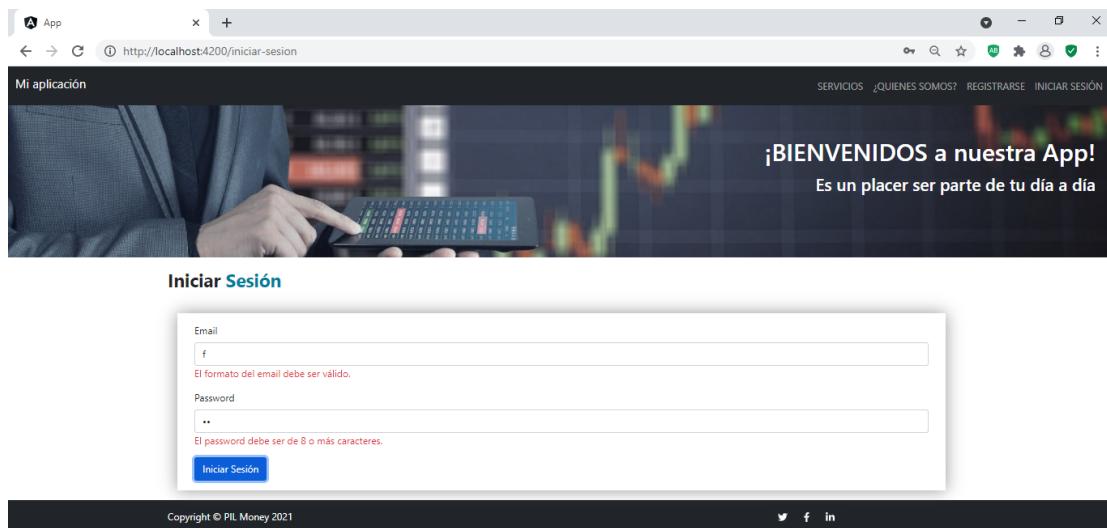


Figura 33: Validaciones

Nota: observa que el color rojo de los mensajes está dado porque el párrafo implementa la clase `text-danger` de bootstrap.

Validar previo enviar el formulario

Para hacer validaciones previo a enviar los datos al servidor, es buena práctica configurar el evento `onSubmit` como sigue:

1. En la etiqueta `<form>` agregar (`ngSumit`):

<form [FormGroup]="form" (ngSubmit)="onEnviar(\$event)">

Método que se ejecutará cuando suceda el evento.

2. En el archivo .ts configurar el método *onEnviar*:

```
onEnviar(event: Event)
{
    event.preventDefault();

    if (this.form.valid)
    {
        alert ("Enviar al servidor...")
    }
    else
    {
        this.form.markAllAsTouched();
    }
}
```

Cancela la funcionalidad por defecto.

Activa todas las validaciones.

De esta manera, si presionas el botón enviar sin ingresar ningún valor, se ejecutarán todas las validaciones.

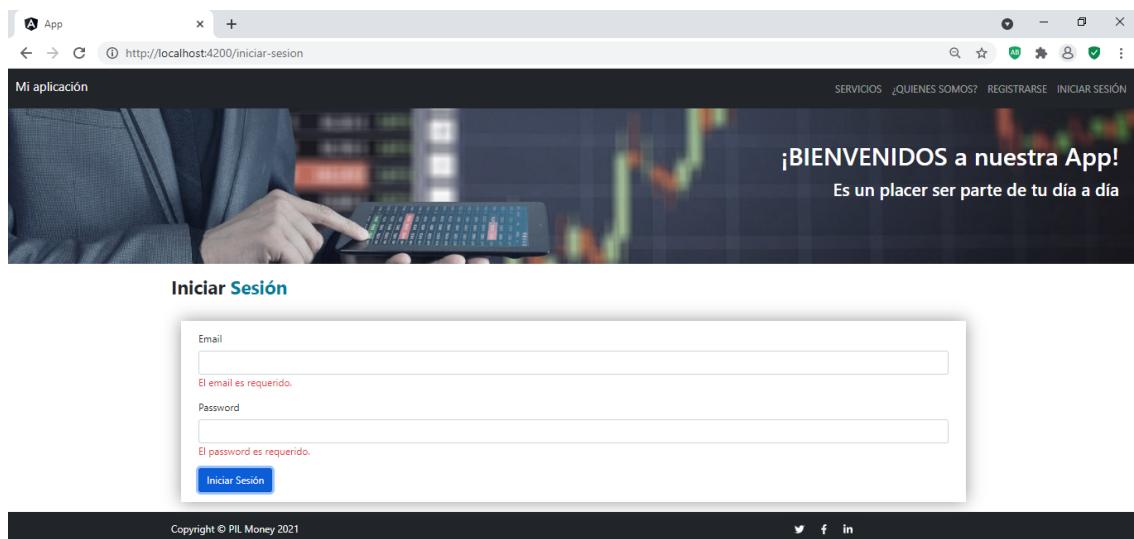


Figura 34: Validaciones

Estilos Bootstrap en formularios reactivos

¿Y si deseamos mejorar la experiencia de usuario haciendo que el borde del input type también esté en rojo?

Para ello, completar los siguientes pasos:

1. En el archivo **iniciar-sesion.component.ts** agregar dos propiedades (getter): PasswordValid y MailValid. Las mismas nos van a servir luego, para mostrar o no el borde rojo:

```
get PasswordValid()
{
    return this.Password?.touched && !this.Password?.valid;
}

get MailValid()
{
    return this.Mail?.touched && !this.Mail?.valid;
}
```

2. En el archivo **iniciar-sesion.component.html** agregar el property binding class en los input type:

```
<input type="email" class="form-control" formControlName="mail"
```

border-danger es una clase de bootstrap.

MailValid es la propiedad creada previamente.

```
[class.border-danger] = "MailValid">
...
<input type="password" class="form-control" formControlName="password"
[class.border-danger] = "PasswordValid">
```

Luego, en nuestra aplicación si existen errores, se mostrarán los bordes en rojo como sigue:

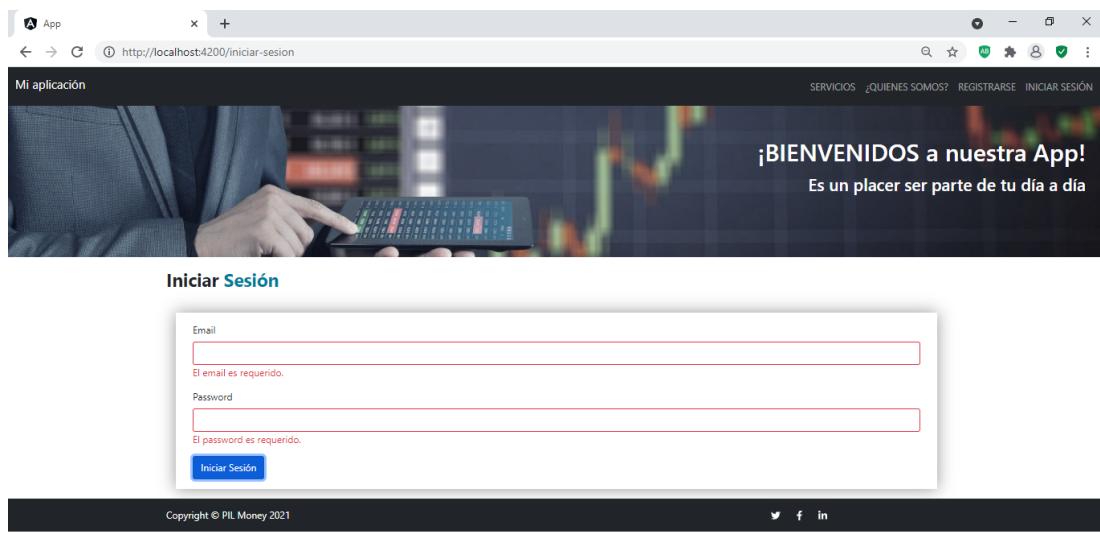


Figura 35: Bordes según la validación

Servicios

Se puede llegar a presentar la situación de que dos o más componentes hagan uso de los mismos datos y para poder ser utilizados son referenciados en cada uno de los componentes. Pero a la hora de mantener la simplicidad Angular incorpora el concepto de Servicios permitiendo así que estos datos serán referenciados por los componentes desde los servicios ayudando a la simplicidad en el componente.

Los componentes delegan actividades a los servicios como ser: obtener datos necesarios, validar los mismos o registrar la información. Es decir que los servicios:

- Son proveedores de datos.
- Ayudan a mantener la lógica de acceso a los mismos.
- Proveen la operatoria del negocio.
- Manipulan de datos en la aplicación.
- Invocar a un servidor HTTP para consumir una API

Podemos pensar entonces que un “servicio es una categoría amplia que abarca cualquier valor, función o característica que necesite una aplicación. Un servicio es típicamente una clase con un propósito limitado y bien definido. Debe hacer algo específico y hacerlo bien.

Angular distingue los componentes de los servicios para aumentar la modularidad y la reutilización. Al separar la funcionalidad relacionada con la vista de un componente de otros tipos de procesamiento, puedes hacer que tus componentes sean ágiles y eficientes.

Idealmente, el trabajo de un componente es permitir la experiencia del usuario y nada más. Un componente debe presentar propiedades y métodos para el enlace de datos, para mediar entre la vista (representada por la plantilla) y la lógica de la aplicación (que a menudo incluye alguna noción de modelo).”(<https://docs.angular.lat/guide/architecture-services>)

Crear servicios

1. Ir a la consola DOS o “Símbolo del Sistema” del sistema operativo o Terminal de VSCode.

2. Ejecutar el comando: **ng generate service <>service-name>>**

o su abreviado: **ng g s <>service-name>>**

A continuación, AngularCLI generará un archivo **<>nombre>.servicio.ts** el cual contendrá las siguiente líneas de código:

Anatomía de la clase de un servicio

Los servicios, como el resto de artefactos en Angular, son clases TypeScript decoradas con funciones específicas como podemos observar en el archivo **<>nombre del servicio>.services.ts**:

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

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

  constructor() { }
}
```

En este caso el decorador **@Injectable()** define el servicio identificando la clase y su metadata que permite a Angular injectarlo a un componente como dependencia.

¿Qué es Inyección de dependencias?

Los componentes consumen servicios; es decir, que podemos injectar un servicio en un componente, dándole acceso al componente a ese servicio.



Figura 36: Inyección de dependencias

Fuente: <https://docs.angular.lat/guide/architecture-services>

De esta manera, el inyector crea dependencias y mantiene un contenedor de instancias de dependencia que reutilizará si es posible.

Es importante mencionar además que, se requiere de un proveedor, dado que éste le dice a un inyector cómo obtener o crear una dependencia.

Para cualquier dependencia que necesites en tu aplicación, debes registrar un proveedor con el inyector de la aplicación, con el fin de que el inyector pueda utilizar el proveedor para crear nuevas instancias. Para un servicio, el proveedor suele ser la propia clase de servicio.” (<https://docs.angular.lat/guide/architecture-services>)

La Inyección de dependencias permite mantener las clases componentes ligeras y eficientes. No obtienen datos del servidor, validan la entrada del usuario o registra directamente en la consola; tales tareas son delegadas a los servicios. (<https://docs.angular.lat/guide/architecture>).

Es un patrón utilizado en la programación orientada a objetos que ayuda en la necesidad de creación de objetos de una forma práctica y con versatilidad en el código.

Los servicios deben tener idealmente un objetivo bien definido y un propósito.

Nota: Angular permite la inyección de servicios de terceros, todo ello a través de la inyección de dependencias.

¿Cómo funciona?

Cuando Angular crea una nueva instancia de un componente, determina qué servicios u otras dependencias necesita ese componente al observar los tipos de parámetros del constructor.

Si Angular descubre que un componente depende de un servicio, primero verifica si el inyector tiene instancias existentes de ese servicio. Si una instancia de servicio solicitada aún no existe, el inyector crea una utilizando el proveedor registrado y la agrega al inyector antes de devolver el servicio a Angular.

Cuando todos los servicios solicitados se han resuelto y devuelto, Angular puede llamar al constructor del componente con esos servicios como argumentos (<https://docs.angular.lat/guide/architecture-services>)

Finalmente el componente puede hacer uso del mismo.

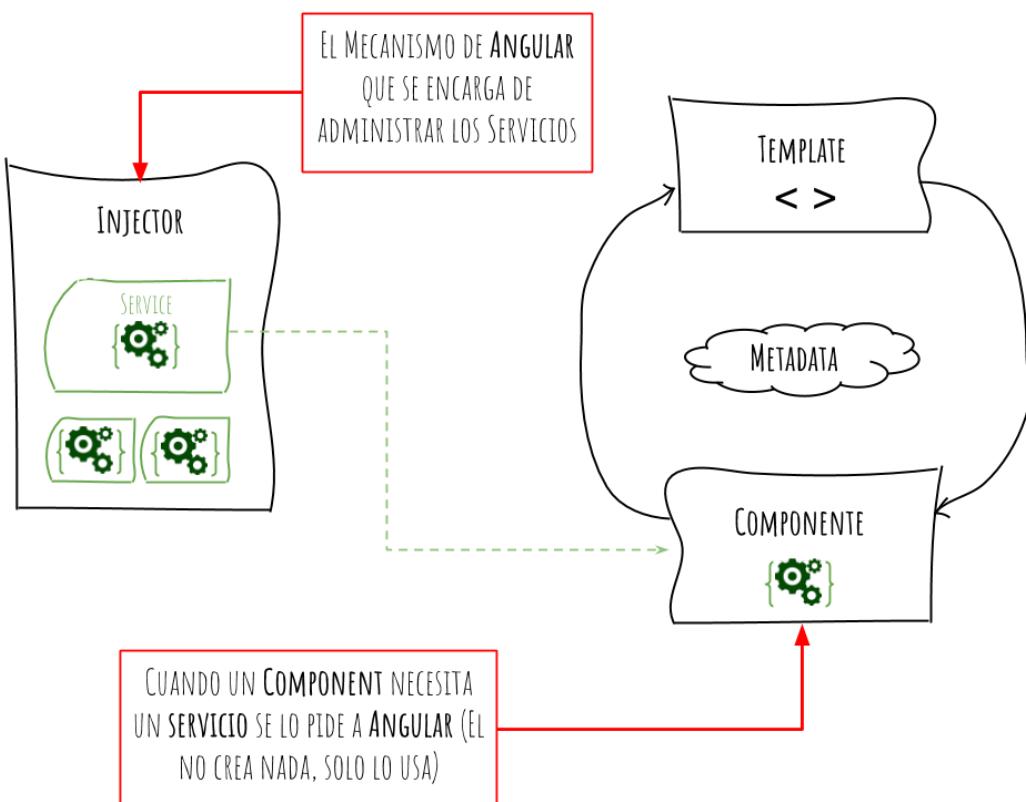


Figura 37: Servicios en Angular

Fuente: <https://gustavodohara.com/blogangular/agregar-servicios-angular-no-componentes-modulos-la-vida/>

Ejemplo:

Continuando con nuestro ejemplo y, a fines de comprender como crear y consumir servicios vamos a crear un servicio que nos provea de los últimos movimientos.

Nota: a fines demostrativos los datos estarán en duro en el servicio pero estos deberían ser obtenidos de una API que se conecte con la base de datos.

Para ello, ejecutar los siguientes pasos:

1. Ir a la consola DOS o “Símbolo del Sistema” del sistema operativo o Terminal de VSCode.
2. Ejecutar el comando: **ng g s services/cuenta**

Nota: services es el nombre de la carpeta dónde deseo se cree el archivo.

A continuación, AngularCLI crea un archivo **cuenta.service.ts**

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

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

  constructor() { }
}
```

3. En el archivo **cuenta.service.ts**, crear un método que devuelva los últimos movimientos.

```
ObtenerUltimosMovimientos()
{
  return [{operacion:"Extracción", monto:1500}, {operacion:"Depósito", monto:1520}];
}
```

4. Importar el servicio cuentas

```
import { CuentaService } from 'src/app/services/cuenta.service';
```

5. Inyectar el servicio en el constructor del componente movimientos (archivo **movimientos.component.ts**)



```
constructor( cuenta: CuentaService )
{
  ...
}
```

6. Consumir el servicio.

```
constructor( cuenta: CuentaService )
{
  this.movimientos=cuenta.ObtenerUltimosMovimientos();
}

ngOnInit(): void {
}
}
```

7. Ejecutar el comando ng serve (si no está corriendo el servidor), y luego ir a <http://localhost:4200/home/movimientos> para ver el formulario. Si bien no se observan cambios, ahora estamos consumiendo un servicio para traer los datos de

los últimos movimientos.

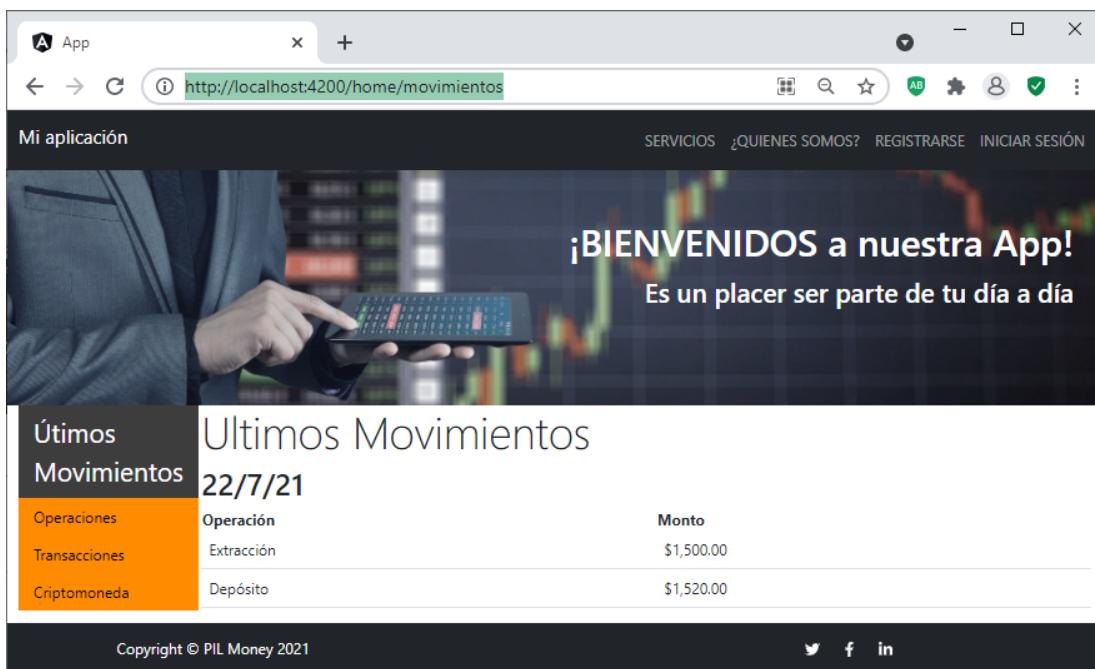


Figura 36: Datos obtenidos del servicio Cuenta

En el ejemplo planteado podemos observar que realizamos las validación desde un servicio y ellas son igualmente efectivas como si estuvieran integradas en el archivo “.ts” del componente.

Usando un servicio en un template

Desde el HTML, también se puede acceder a servicios creados para mostrar propiedades o invocar sus métodos a través de los eventos.

```
<p>
  Nro. Cuenta: {{Cuenta.Nro}}
</p>
```

Referencias

- <https://docs.angular.lat/>
- <https://desarrolloweb.com/articulos/introduccion-teorica-observables-angular.html>
- <https://angular.io/>
- <https://davidguru.medium.com/single-page-application-un-viaje-a-las-spa-a-trav%C3%A9s-de-angular-y-javascript-337a2d18532>
- <https://startbootstrap.com/theme/agency>