Contenido

[Temario para Angular 3](#_Toc508353250)

[1 Introducción a Angular 5 3](#_Toc508353251)

[¿Angular5 o angular.io? 3](#_Toc508353252)

[Instalando Angular 3](#_Toc508353253)

[Instalando Angular CLI 3](#_Toc508353254)

[Carpeta src 4](#_Toc508353255)

[Carpeta node\_modules 4](#_Toc508353256)

[Carpeta e2e 4](#_Toc508353257)

[Otras carpetas 4](#_Toc508353258)

[Conceptos básicos 4](#_Toc508353259)

[Componente 4](#_Toc508353260)

[Decorador 4](#_Toc508353261)

[Plantilla 4](#_Toc508353262)

[Metadatos 5](#_Toc508353263)

[2 TypeScript 5](#_Toc508353264)

[Introducción a TS 5](#_Toc508353265)

[3 Componentes 5](#_Toc508353266)

[Programación orientada a componentes 5](#_Toc508353267)

[Arquitectura de una aplicación en Angular5 5](#_Toc508353268)

[Creando un modelo 10](#_Toc508353269)

[Creando un componente 10](#_Toc508353270)

[Creando el Module 12](#_Toc508353271)

[Creating the “Startup.ts” file 14](#_Toc508353272)

[Invocando el archivo “Startup.ts” usando la página principal de angular 15](#_Toc508353273)

[Setting up el compilador typescript 16](#_Toc508353274)

[Running el http-server 17](#_Toc508353275)

[Implementando SPA usando Angular routing 17](#_Toc508353276)

[Crear la página del proveedor y la página de bienvenida 18](#_Toc508353277)

[Cambiar el nombre del marcador de posición in Index.html 18](#_Toc508353278)

[Removiendo el selector de CustomerComponent 19](#_Toc508353279)

[Creando componentes para la página maestra, Supplier y Welcome. 19](#_Toc508353280)

[Creando la colección constante del routing 20](#_Toc508353281)

[Definiendo el routerLink y router-outlet 21](#_Toc508353282)

[Cargando el enrutamiento en los módulos principales 22](#_Toc508353283)

[Definir APP BASE HREF 23](#_Toc508353284)

[Viendo la salida 24](#_Toc508353285)

[Corrección No se puede encontrar ningún error de ruta 24](#_Toc508353286)

[Implementando la validación usando la forma angular 25](#_Toc508353287)

[Importar componentes necesarios para validadores angulares 27](#_Toc508353288)

[Agregar una validación simple 27](#_Toc508353289)

[Agregar una una validación compuesta 28](#_Toc508353290)

[Código de modelo completo con validación 28](#_Toc508353291)

[Referencia "ReactiveFormsModule" en CustomerModule. 29](#_Toc508353292)

[Aplicar formGroup a formulario HTML 29](#_Toc508353293)

[Aplicar validaciones al control de HTML 29](#_Toc508353294)

[Elementos standalone 30](#_Toc508353295)

[Complete code of Customer UI with validations applied 32](#_Toc508353296)

[Run and see your validation in action 32](#_Toc508353297)

[Dirty , pristine , touched and untouched 33](#_Toc508353298)

[Bibliografía 33](#_Toc508353299)

# Temario para Angular

# 1 Introducción a Angular 5

¿Angular5 o angular.io?

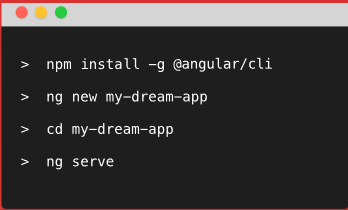
**Angular** (comúnmente llamado "**Angular** 2+" o "**Angular** 2"), es un framework para aplicaciones web desarrollado en TypeScript, de código abierto, mantenido por Google, que se utiliza para crear y mantener aplicaciones web de una sola página.

Instalando Angular

1.- Abrimos una consola nos colocamos en la carpeta deseada e instalamos un proyecto de node.

npm install

## Instalando Angular CLI



Encontrarás varios archivos sueltos en la carpeta raíz de tu proyecto. Te señalamos algunos importantes que debes conocer:

**index.html:** Este archivo es información básica del proyecto recién creado. Encontrarás algunos comandos explicados del CLI, como "ng serve" el cuál ya hemos tratado, para servir el proyecto.

**.angular-cli.json:** Es un archivo oculto (en Linux o Mac, pues comienza por ".") en el que se almacenan configuraciones del CLI de Angular.

**package.json:** Este es el archivo que resume las dependencias del proyecto, las librerías sobre las que nos apoyamos, que se gestionan por medio de npm.

**tslint.json:** Este archivo sirve para configurar el linter, el programa que nos alertará cuando tengamos problemas de estilo en el código.

**.editorconfig:** Este es un archivo que sirve para definir la configuración para el editor de código que estemos utilizando. Permite centralizar la configuración, de modo que sea común para todos los desarrolladores que vayan a trabajar en el proyecto.

### Carpeta src

Es la carpeta donde están las fuentes del proyecto. Dentro de "src" se encuentran archivos como el index.html, que hace de raíz del proyecto.

Otro de los detalles que encontrarás en "src" son varios archivos con extensión ".ts". Son archivos con código TypeScript. Dentro de "src" encontrarás también una carpeta llamada "app", que contiene el código del componente principal, que está dividido en varios archivos. Si abres el archivo ".html" verás el código de presentación del componente, que es el que se muestra cuando se visualiza la aplicación recién creada.

### Carpeta node\_modules

Es la carpeta donde npm va colocando todas las dependencias del proyecto, es decir, el código de todas las librerías o componentes que estemos usando para basarnos en el desarrollo de una aplicación. Por ejemplo, el propio Angular es una dependencia.

### Carpeta e2e

En esta carpeta se colocan los archivos para la realización de las pruebas "end to end".

### Otras carpetas

A medida que trabajes podrás encontrar en el proyecto carpetas como "test" o "dist", para realizar el test o para almacenar los archivos listos para producción. Sigue leyendo las carpetas disponibles en el proyecto Angular 2.0 para más información.

## Conceptos básicos

### Componente

Un componente al final va a controlar un trozo de pantalla o de la vista, todo lo que se puede ver en la pantalla es controlado y gestionado por este tipo de elementos

La lógica de un componente dentro de una clase en angula es que se da soporte a una vista interactuando con ella a través de un api con propiedades y métodos

El componente hace de mediador entre la lista a través de la plantilla y la lógica de la app donde se incluirá el modelo de datos, es decir una especie de controlador

### Decorador

Los decoradores aportan características extra a las clases y condicionan el comportamiento de las mismas dependiendo de los metadatos o propiedades que tengan

### Plantilla

Las plantillas can a definir la vista de los componentes Generalmente son archivos HTML y tiene l sintaxis especial entro de Angular, permitiéndonos trabajar con directivas, data-binding, pipies, etc

### Metadatos

Los metadatos en los decoradores son propiedades que apostan características extra a las clase y escriben relacione en ellas, ejemplo si tenemos un componente y una plantilla el metadato se encargar de decirle Angular que ese componente y esa platilla van juntos.

# 2 TypeScript

## Introducción a TS

TypeScript convierte su código en Javascript común. Es llamado también Superset de Javascript, lo que significa que si el navegador está basado en Javascript, este nunca llegará a saber que el código original fue realizado con TypeScript y ejecutará el Javascript como lenguaje original.

¿Qué es un superset?

Se trata de un lenguaje escrito sobre otro lenguaje. En este caso Typescript es eso, un lenguaje basado en el original

# 3 Componentes

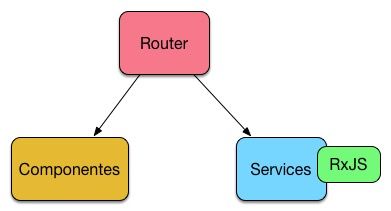
## Programación orientada a componentes

La **programación orientada a componentes** (que también es llamada *basada en componentes*) es una rama de la ingeniería del software, con énfasis en la descomposición de sistemas ya conformados en componentes funcionales o lógicos con interfaces bien definidas usadas para la comunicación entre componentes.

Se considera que el nivel de abstracción de los componentes es más alto que el de los objetos y por lo tanto no comparten un estado y se comunican intercambiando mensajes que contienen datos.

## Arquitectura de una aplicación en Angular5

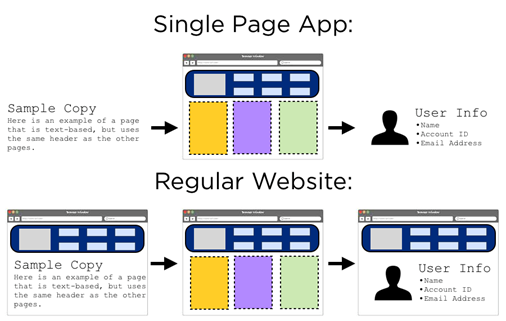
Podemos decir que hay muchos cambios desde Angular 2, **pero la arquitectura MVC se mantiene**. Disponemos de Componentes para la vista, Enrutador para la capa de control y servicios para la capa de backend.

[](https://www.arquitecturajava.com/wp-content/uploads/Angular2Diagram.jpg)

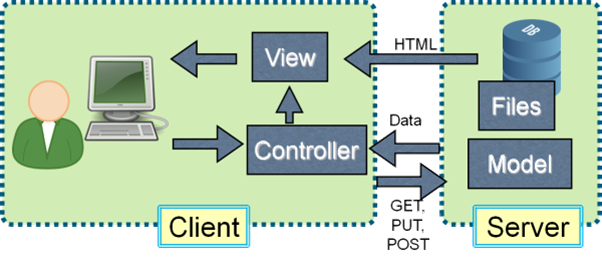
**Inyección de dependencias**: Soportan de forma muy natural las inyecciones de dependencias apoyándose **en TypeScript.**

[](https://www.arquitecturajava.com/wp-content/uploads/Angular2Proxies.jpg)

Con Angular podemos crear Single Page Applications (SPA):



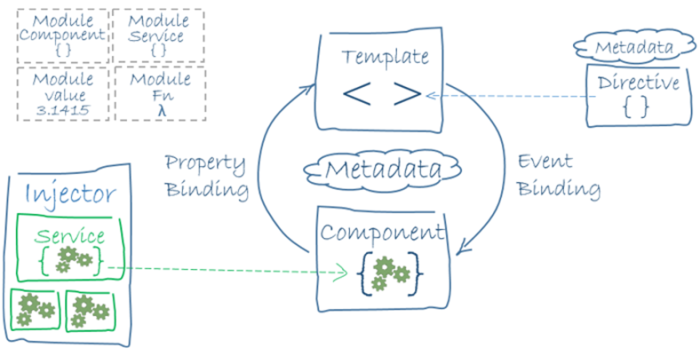
Las SPA’s gestionan toda la interactividad con el usuario desde el lado cliente, cargando recursos del servidor (vistas, datos… etc.) vía AJAX



Estas son algunas características de la estructura Angular 4:

* **Framework** para construir SPA’s
* El código se escribe en **TypeScript** que transpila a JavaScript
* Angular consiste de un conjunto de librerías multipropósito, varias son opcionales
* El desarrollador crea **vistas** (HTML Templates) con atributos propios de Angular
* El desarrollador escribe **Componentes** (clases TypeScript) para gestionar el contenido de la vista
* La lógica de la aplicación a nivel cliente se escribe con clases denominadas **Services**.
* Todos los componentes, y services se crean dentro de **módulos**
* Se lanza la aplicación ejecutando el **Root Module**.

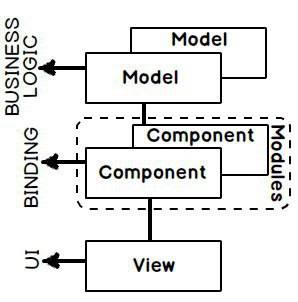
En el siguiente esquema se puede observar gráficamente la Arquitectura de Angular 4:



<https://angular.io/guide/architecture>

Como mencionamos anteriormente, Angular 4 se compone de módulos que internamente pueden tener un conjunto de componentes, servicios, valores y funciones.

En proyectos empresariales, puede tener muchos componentes. Con muchos componentes, puede ser muy difícil manejar el proyecto. De modo que puede agrupar los componentes lógicamente en módulos.



**Componentes:** - Tiene la lógica de enlace para enlazar la interfaz de usuario y el modelo.

**Módulos:** - Esto lógicamente agrupará los componentes.

También en la misma carpeta necesitamos crear tsconfig.json que define cómo se compilará el texto mecanografiado. Por ahora, solo tenga en cuenta que el siguiente archivo define cómo se compilará TSC.

|  |
| --- |
| {  "compilerOptions": {  "target": "es5",//Define el tipo de codigo a generar.  "module": "commonjs",  "moduleResolution": "node",  "sourceMap": true,  "emitDecoratorMetadata": true,// para usar los metadatos.  "experimentalDecorators": true,//ocupan decorators como @Component..  "removeComments": false,  "noImplicitAny": false,  "noStrictGenericChecks": true,  "lib": ["es2016", "dom"]  }  } |

Antes de comenzar a codificar, visualicemos los pasos de la codificación. Como dijimos, Angular es un marco vinculante. Sigue la arquitectura MVW (Model-View-Whatever). Vincula la interfaz de usuario HTML con el código JavaScript (modelo).

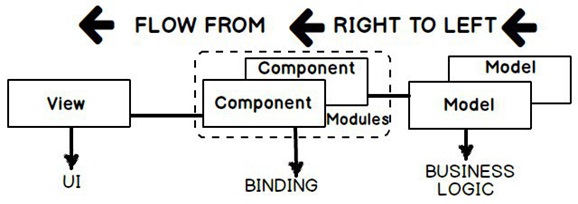
Entonces, si visualizamos, se verá algo como se muestra en la imagen a continuación. Así que hagamos la codificación en la siguiente secuencia:

    1.- Crea el modelo.

    2.- Crea el Componente.

    3.- Crea el módulo.

    4.- Crea la IU HTML.



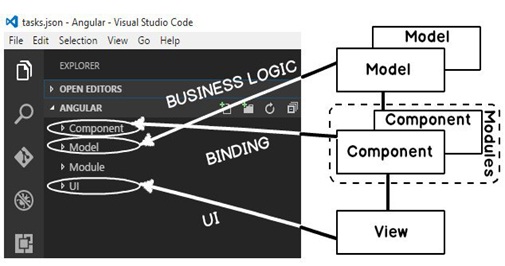
Primero, creemos cuatro carpetas en nuestro proyecto:

**UI:** Contiene la IU HTML.

**Model:** Contiene las principales clases de mecanografía comerciales.

**Component**: Contiene el código de enlace que vincula la UI, HTML y el modelo.

**Module:** Esta carpeta tendrá un código que lógicamente agrupará los componentes.



## Creando un modelo

Un modelo no es más que una clase con propiedades y comportamiento.

1.- Crea un modelo de cliente con tres propiedades "CustomerName", "CustomerCode" y "CustomerAmount".

2.- Agregue un nuevo archivo "Customer.ts". Mantenga la extensión de este archivo como ".ts", ya que es un archivo de texto mecanografiado.

Al compilar el comando de mecanografía, identifica solo los archivos con la extensión ".ts".

3.- En "Customer.ts", creemos una clase "Cliente" con tres propiedades.

export class Customer {

CustomerName: string = "";

CustomerCode: string = "";

CustomerAmount: number = 0;

}

## Creando un componente

Lo siguiente que debemos codificar es el código de enlace. El código de enlace en Angular está representado por algo denominado "COMPONENTES". Los componentes angulares tienen la lógica que ayuda a unir la interfaz de usuario con el modelo.

1. Agregue el archivo "CustomerComponent.ts"

En el componente, necesitamos importar dos cosas, el núcleo angular y nuestro modelo de cliente.

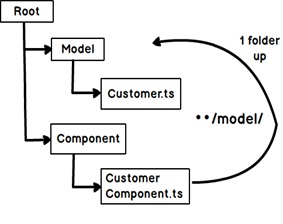
2.-

import {Customer} from '../Model/Customer'

import {Component} from "@angular/core"

La primera línea importa la clase "Cliente" en el "ClienteComponente.ts". Esta importación solo es posible porque hemos escrito "export" en el archivo "Customer.ts". La importación y la exportación generan código que sigue las especificaciones CommonJs, AMD o UMD.

import {Customer} from '../Model/Customer'



El siguiente comando de importación importa componentes centrales angulares. En esto, no hemos dado ninguna ruta relativa utilizando "../", etc. Entonces, ¿cómo puede el texto mecanografiado ubicar los componentes del núcleo angular?

*import {Component} from "@angular/core"*

Si recuerda, hemos utilizado un nodo para cargar Angular y el nodo carga los archivos JS en la carpeta "node\_modules". Entonces, ¿cómo sabe el compilador de mecanografía que tiene que cargar los componentes angulares de la carpeta "node\_modules"?

El compilador de tipos utiliza la configuración del archivo "tsconfig.json". En la configuración, tenemos una propiedad denominada "moduleResolution". Tiene dos valores:

**Clásico**: - En este modo, el mecanografiado se basa en "./" y "../" para localizar carpetas.

**Nodo:** - En este modo, el texto mecanografiado primero trata de ubicar los componentes en la carpeta "node\_modules" y si no se encuentra, sigue la convención "../" para recorrer las carpetas.

En nuestro tsconfig.json hemos definido el modo como "nodo" esto hace que los módulos de búsqueda mecanografiados automáticamente en la carpeta "node\_modules". Eso hace que la "importación" de componentes angulares funcione.

|  |
| --- |
| { |
| { |
| .... |
| .... |
| "moduleResolution": "node", |
| .... |
| .... |
| } |
| } |

Entonces, ahora que se aplican ambas declaraciones de importación, creemos el "ClienteComponente" y, a partir de eso, expongamos el objeto "Cliente" a la interfaz de usuario con el nombre de objeto "Cliente actual".

|  |
| --- |
| export class CustomerComponent { |
| CurrentCustomer:Customer = new Customer(); |
| } |

Como dijimos anteriormente, ese componente conecta/une el modelo a la IU HTML. Entonces debería haber algún código que diga que "ClienteComponente" está limitado con la IU HTML. Esto se hace mediante algo denominado "Atributo de metadatos de componente". Un atributo de metadato de componente comienza con "@Component" que tiene una propiedad "templateUrl" que especifica la UI HTML con la que está asociada la clase de componente.

|  |
| --- |
| @Component({ |
| selector: "customer-ui", |
| templateUrl: "../UI/Customer.html" |
| }) |

Este atributo se decora en la parte superior del componente. A continuación aparece el código completo.

|  |
| --- |
| @Component({ |
| selector: "customer-ui", |
| templateUrl: "../UI/Customer.html" |
| }) |
| export class CustomerComponent { |
| CurrentCustomer:Customer = new Customer(); |
| } |

Al poner en palabras simples el código de enlace no es más que una clase de mecanografía simple decorada por el atributo "@Component" que dicta que esta clase de texto mecanografiado está vinculada con qué UI



A continuación se muestra el código completo del componente angular.

|  |
| --- |
| // Import statements |
| import {Component} from "@angular/core" |
| import {Customer} from '../Model/Customer' |
|  |
| // Attribute metadata |
| @Component({ |
| selector: "customer-ui", |
| templateUrl: "../UI/Customer.html" |
| }) |
| // Customer component class exposing the customer model |
| export class CustomerComponent { |
| CurrentCustomer:Customer = new Customer(); |
| } |

## Creando el Module

El módulo es un contenedor o puede decir que es una agrupación lógica de componentes y otros servicios.

Entonces, la primera importación en este módulo es el componente "ClienteComponente".

|  |
| --- |
| import { CustomerComponent }   from '../Component/CustomerComponent'; |

También necesitamos importar "BrowserModule" y "FormsModule" del núcleo angular. "BrowserModule" tiene componentes mediante los cuales podemos escribir las condiciones IF y el ciclo FOR. "FormsModule" proporciona funcionalidad directiva como "ngModel", expresiones, etc.

|  |
| --- |
| import { BrowserModule } from '@angular/platform-browser'; |
| import {FormsModule} from "@angular/forms" |

También necesitamos crear una clase de mecanografía "MainModuleLibrary". En este momento, esta clase no tiene ningún código, pero puede tener un código que proporcione lógica de nivel de componente como el almacenamiento en caché, el código de inicialización para ese grupo de componentes, etc.

|  |
| --- |
| export class MainModuleLibrary { } |

Para crear un módulo, necesitamos usar la importación "NgModule" del núcleo angular. Esto nos ayuda a definir directivas de módulos.

|  |  |
| --- | --- |
| NgModule "tiene tres propiedades: |  |

**Importaciones:** - Si este módulo está utilizando otros módulos definimos los módulos en esta sección.

**Declaraciones:** - En esta sección definimos los componentes de los módulos. Por ahora solo tenemos un componente 'CustomerComponent'.

**Bootstrap:** - Esta sección define el primer componente que se ejecutará. Por ejemplo, podemos tener "HomeComponent", "CustomerComponent", etc. Pero el primer componente que se ejecutará es el "HomeComponent", por lo que debemos definirlo en esta sección.

|  |
| --- |
| @NgModule({ |
| imports: [BrowserModule, |
| FormsModule], |
| declarations: [CustomerComponent], |
| bootstrap: [CustomerComponent] |
| }) |

A continuación se muestra el código completo del módulo angular que discutimos en esta sección.

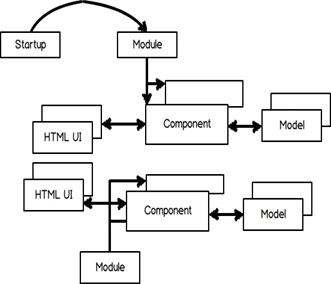
|  |
| --- |
| import { NgModule }      from '@angular/core'; |
| import { BrowserModule } from '@angular/platform-browser'; |
| import {FormsModule} from "@angular/forms" |
| import { CustomerComponent }   from '../Component/CustomerComponent'; |
|  |
| @NgModule({ |
| imports: [BrowserModule, |
| FormsModule], |
| declarations: [CustomerComponent], |
| bootstrap: [CustomerComponent] |
| }) |
| export class MainModuleLibrary { } |
|  |

## Creating the “Startup.ts” file

Así que hemos creado la interfaz de usuario, hemos creado los modelos, hemos creado componentes y estos componentes están agrupados en módulos. En una aplicación angular, puede tener muchos módulos.

Entonces necesitas definir el módulo de inicio. Así que creemos un archivo "Startup.ts" que definirá el módulo de inicio.

A continuación, se muestra el archivo "Startup.ts" en el que hemos definido qué módulo se iniciará.



|  |
| --- |
| import { platformBrowserDynamic } from '@angular/platform-browser-dynamic'; |
| import { MainModuleLibrary } from '../Module/MainModuleLibrary'; |
| const platform = platformBrowserDynamic(); |
| platform.bootstrapModule(MainModuleLibrary); |

## Invocando el archivo “Startup.ts” usando la página principal de angular

Así que vamos a crear una página HTML de inicio que invocará los "Startup.ts". Ahora en esta página necesitaremos importar cuatro archivos de marcos de JavaScript Shim, Zone, Meta-data y System JS como se muestra en el siguiente código.

<script src="../../node\_modules/core-js/client/shim.min.js"></script>

<script src="../../node\_modules/zone.js/dist/zone.js"></script>

<script src="../../node\_modules/reflect-metadata/Reflect.js"></script>

<script src="../../node\_modules/systemjs/dist/system.src.js"></script>

A continuación se muestra el uso de archivos JS:

**Shim.min.js:** Este marco garantiza que ES 6 javascript pueda ejecutarse en navegadores antiguos.

**Zone.js:** Este marco nos garantiza que tratemos el grupo de actividades Async como una sola zona.

**Reflect.js:** nos ayuda a aplicar metadatos en las clases de Javascript.

Actualmente estamos usando @NgModule y @NgComponent como atributos.

**System.js:**  Este módulo ayudará a cargar archivos JS usando protocolos de módulos como commonjs, AMD o UMD.

En esta página HTML llamaremos al archivo "systemjs.config.js". Este archivo le indicará al sistema JS qué archivos se cargarán en el navegador.

<script src="../systemjs.config.js"></script>

<script>

System.config({

"defaultJSExtensions": true

});

System.import('startup').catch(function (err) { console.error(err); });

</script>

En la "importación" necesitamos especificar "inicio" que invocará el archivo "startup.js".

System.import('startup').catch(function (err) { console.error(err); });

Nuestra pantalla de cliente entra con el nombre "Customer.html". Entonces, para cargar en esta pantalla, necesitamos definir un marcador de posición. Por lo tanto, en este lugar, se cargará nuestra página HTML de cliente.

<customer-ui></customer-ui>

Si recuerda cuando creamos la clase de componente, habíamos dicho que carguemos la página HTML en un selector. Entonces ese selector no es más que una etiqueta (marcador de posición) para cargar nuestra página de Cliente.

@Component({

    selector: "customer-ui",

    templateUrl: "../UI/Customer.html"

})

A continuación, se muestra la página HTML completa con todos los scripts y la etiqueta del marcador de posición.

<!DOCTYPE html>

<html>

<head>

<title></title>

<meta charset="utf-8" />

</head>

<!-- 1. Load libraries -->

<!-- Polyfill(s) for older browsers -->

<script src="../../node\_modules/core-js/client/shim.min.js"></script>

<script src="../../node\_modules/zone.js/dist/zone.js"></script>

<script src="../../node\_modules/reflect-metadata/Reflect.js"></script>

<script src="../../node\_modules/systemjs/dist/system.src.js"></script>

<!-- 2. Configure SystemJS -->

<script src="../systemjs.config.js"></script>

<script>

System.config({

"defaultJSExtensions": true

});

System.import('startup').catch(function (err) { console.error(err); });

</script>

<body>

<customer-ui></customer-ui>

</body>

</html>

## Setting up el compilador typescript

También necesitamos ejecutar mecanografiado para que los archivos TS se compilen en los archivos JS. Abra el terminal integrado y escriba "tsc -w". Esto hará que el texto mecanografiado se ejecute continuamente en el fondo. Así que a medida que escribe mecanografiado se seguirá compilando cuando el archivo cambie.

## Running el http-server

Entonces, si todo está bien, ahora debemos ejecutar el servidor http. Para ejecutar el servidor, debemos escribir "http-server" en el terminal.(En caso de que el puerto 80 esteocupado se puede especifiar el puerto con ”http-server –p 99”)

## Implementando SPA usando Angular routing

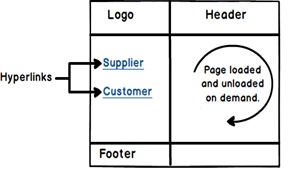
Hoy en día, la aplicación de una sola página (SPA) se ha convertido en el estilo de creación de sitios web. En SPA solo cargamos cosas que necesitamos y nada más que eso.

Por lo tanto, la primera vez que el usuario ingrese al sitio, se cargarán todas las secciones de la página maestra. Pero cuando el usuario hace clic en el enlace Proveedor, solo se cargará la página del Proveedor y no el logotipo, el encabezado y el pie de página nuevamente. Cuando el usuario hace clic en el enlace del Cliente, solo se cargará la página del Cliente y no todas las demás.

El enrutamiento angular nos ayuda a lograr lo mismo.

Paso 1: - Creando la Página Maestra

Como todo gira alrededor de la Página Maestra, el primer paso lógico sería crear la "Página maestra".



En esta página maestra crearemos marcadores de posición para logotipo, encabezado, menú, pie de página, derechos de autor, etc. Estas secciones se cargarán solo una vez cuando el usuario navegue por el sitio web la primera vez. Y en los últimos tiempos solo se cargarán las páginas que se necesiten a pedido.

A continuación se muestra el código HTML de muestra que tiene todas las secciones de marcador de posición. También puede ver en este código que hemos mantenido una sección de etiqueta "DIV" en la que cargaríamos las páginas a pedido.

A continuación se muestran las principales secciones generales de "MasterPage". Tenga en cuenta la sección "DIV" con el nombre "dynamicscreen" donde tenemos la intención de cargar pantallas dinámicamente. Llenaremos estas secciones más tarde.

<table border="1" width="448">

<tr>

<td>Logo</td>

<td width="257">Header</td>

</tr>

<tr>

<td>Left Menu</td>

<td width="257">

<div id="dynamicscreen">

Dynamic screen will be loaded here

</div>

</td>

</tr>

<tr>

<td>Footer</td>

<td width="257">Copyright</td>

</tr>

</table>

# Crear la página del proveedor y la página de bienvenida

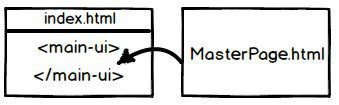
Vamos a crear dos UI HTML más, una página de "Proveedor" y una de "Bienvenida". En ambas páginas HTML no estamos haciendo mucho, solo tenemos mensajes de saludo.

## Cambiar el nombre del marcador de posición in Index.html

"Index.html" es la página de inicio y carga todas las demás páginas usando systemjs. En la lección anterior dentro de "Index.html", se estaba cargando la página "Customer.html". Pero ahora que tenemos una página maestra, se cargará la página índice "MasterPage.html".

Para que sea más significativo, cambiemos el nombre de la etiqueta "customer-ui" por "main-ui". En esta sección "main-ui" cargaremos la página maestra y cuando el usuario final haga clic en la página maestra del menú izquierdo, el proveedor de enlaces se cargará.

Entonces, si observamos el flujo, primero se cargará index.html y luego dentro del "main-ui", se cargará "masterpage.html".



### Removiendo el selector de CustomerComponent

Ahora la primera página para cargar en el index.html será Masterpage y no la página del Cliente. Entonces, debemos eliminar el selector de "CustomerComponent.ts". Este selector se moverá al componente de página maestra en las secciones posteriores.

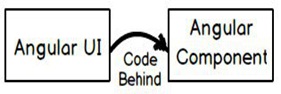
### Creando componentes para la página maestra, Supplier y Welcome.

Cada UI que tiene Angular habilitado debe tener un archivo de código de componente. Hemos creado 3 interfaces de usuario, por lo que necesitamos tres archivos de código de componente para el mismo.

En la carpeta de componentes, crearemos tres archivos TS de componentes "MasterPageComponent.ts", "SupplierComponent.ts" y "WelcomeComponent.ts".

Puede visualizar archivos de códigos de componentes como código detrás de la interfaz de usuario angular.

Así que primero comencemos con el componente "MasterPage.html" que hemos nombrado como "MasterPageComponent.ts". Esta página maestra se cargará en "Index.html" en el proceso inicial de arranque. Puede ver en este componente que hemos puesto el selector y este será el único componente que tendrá el selector.



import {Component} from "@angular/core"

@Component({

    selector: "main-ui",

    templateUrl: "../UI/MasterPage.html"

})

export class MasterPageComponent {

}

A continuación se muestra el código del componente para "Supplier.html".

import {Component} from "@angular/core"

@Component({

    templateUrl: "../UI/Supplier.html"

})

export class SupplierComponent {

}

A continuación se muestra el código del componente para "Welcome.html". Tanto el proveedor como el componente de bienvenida no tienen el selector, solo el componente de página maestra lo tiene, ya que será la interfaz de usuario de inicio que se cargará en la página de índice.

import {Component} from "@angular/core"

@Component({

    templateUrl: "../UI/Welcome.html"

})

export class WelcomeComponent {

}

### Creando la colección constante del routing

Una vez que la página maestra se carga en la página de índice, el usuario final hará clic en los enlaces de la página maestra para navegar a la página del proveedor, a la página del cliente, etc. Ahora, para que el usuario pueda navegar correctamente, necesitamos definir las rutas de navegación. Estas rutas se especificarán en las etiquetas "href" en los pasos posteriores.

Cuando se naveguen estas rutas, invocarán los componentes y los componentes cargarán la IU. A continuación se muestra una tabla simple con tres columnas. La primera columna especifica el patrón de ruta, segundo qué componente invocará cuando se naveguen estas rutas y la columna final especifica la UI que se cargará.

|  |  |  |
| --- | --- | --- |
| **Path/URL** | **Component** | **UI which will be loaded** |
| / | WelcomeComponent.ts | Welcome.html |
| /Customer | CustomerComponent.ts | Customer.html |
| /Supplier | SupplierComponent.ts | Supplier.html |

Las rutas y las entradas de componentes se deben definir en una colección literal simple como se muestra en el siguiente código. Puede ver que "ApplicationRoutes" es una colección simple donde hemos definido la ruta y el componente que se invocará. Estas entradas se realizan según la tabla especificada en la parte superior.

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

import {CustomerComponent} from '../Component/CustomerComponent';

import {SupplierComponent} from "../Component/SupplierComponent";

import {WelcomeComponent} from "../Component/WelcomeComponent";

export const ApplicationRoutes = [

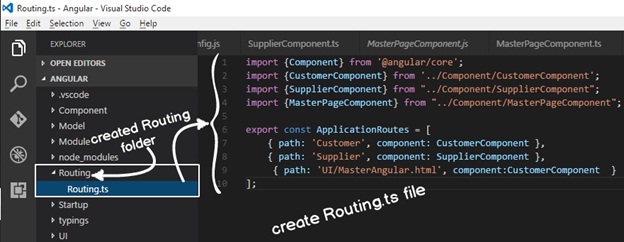
    { path: 'Customer', component: CustomerComponent },

    { path: 'Supplier', component: SupplierComponent },

     { path: '', component:WelcomeComponent  }

];

Como una buena práctica, todos los códigos anteriores los hemos definido en una carpeta separada "routing" y en un archivo separado "routing.ts".



### Definiendo el routerLink y router-outlet

La navegación (rutas) definida en "Paso 6" en el collection debe referirse cuando intentamos navegar dentro del sitio web. Por ejemplo, en la página maestra, hemos definido los hipervínculos del menú de la izquierda.

Entonces, en lugar de usar la etiqueta "href" de HTML, necesitamos usar "[routerLink]"

<a href="Supplier.html"> Proveedor </a>

Necesitamos usar "[routerLink]" y el valor de "[routerLink]" será la ruta especificada en la colección de rutas que define el paso anterior. Por ejemplo, en la colección "ApplicationRoutes" hemos realizado una entrada para la ruta del Proveedor, necesitamos especificar la ruta en la etiqueta de anclaje como se muestra en el siguiente código.

<a [routerLink]="['Supplier']"> Proveedor </a>

Cuando el usuario final hace clic en los enlaces de la página maestra izquierda, las páginas (página del proveedor, página del cliente y página de bienvenida) se cargarán dentro de la etiqueta "div". Para eso tenemos que definir el marcador de posición "enrutador de salida". Dentro de este marcador de posición, las páginas se cargarán y descargarán dinámicamente.

<div id = "dynamicscreen">

<router-outlet> </ router-outlet>

</ div>

Por lo tanto, si actualizamos la página maestra definida en "Paso 1" con "router-link" y "router-outlet", obtendremos un código como se muestra a continuación.

<table border="1">

<tr>

<td><img src="http://www.questpond.com/img/logo.jpg" alt="Alternate Text" />

</td>

<td>Header</td></tr><tr>

<td>Left Menu<br /><br /><br />

<a [routerLink]="['Supplier']">Supplier</a> <br /><br />

<a [routerLink]="['Customer']">Customer</a></td><td>

<div id="dynamicscreen">

<router-outlet></router-outlet>

</div>

</td>

</tr>

<tr>

<td>Footer</td><td></td>

</tr>

</table>

## Cargando el enrutamiento en los módulos principales

Para habilitar las rutas de recolección de enrutamiento definidas en "Rutas de aplicación", debemos cargar eso en la "Biblioteca de módulo principal" como se muestra en el código siguiente. "RouterModule.forRoot" ayuda a cargar las rutas de la aplicación a nivel de módulo.

Una vez cargados en el nivel del módulo, estarán disponibles para todos los componentes para fines de navegación que se cargan dentro de este módulo.

@NgModule({

    imports: [RouterModule.forRoot(ApplicationRoutes),

             BrowserModule,

             FormsModule],

    declarations: [CustomerComponent,MasterPageComponent,SupplierComponent],

    bootstrap: [MasterPageComponent]

})

export class MainModuleLibrary { }

El código completo con rutas se vería como se muestra a continuación.

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

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

import {FormsModule} from "@angular/forms"

import { CustomerComponent }   from '../Component/CustomerComponent';

import { SupplierComponent }   from '../Component/SupplierComponent';

import { MasterPageComponent }   from '../Component/MasterPageComponent';

import { RouterModule }   from '@angular/router';

import { ApplicationRoutes }   from '../Routing/Routing';

@NgModule({

    imports: [RouterModule.forRoot(ApplicationRoutes),

             BrowserModule,

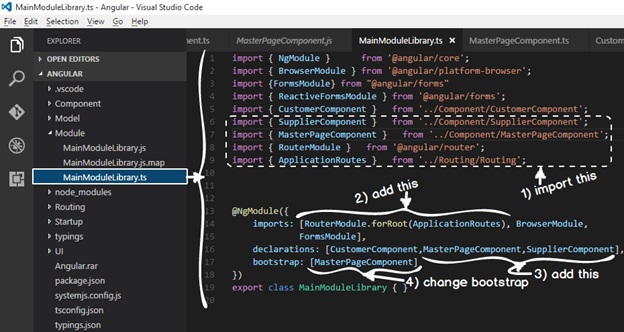
             FormsModule],

    declarations: [CustomerComponent,MasterPageComponent,SupplierComponent],

    bootstrap: [MasterPageComponent]

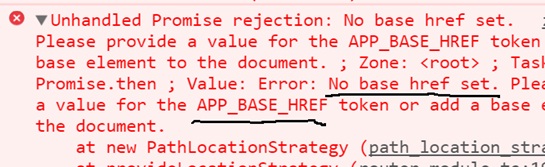
})

export class MainModuleLibrary { }



## Definir APP BASE HREF

El módulo de enrutador requiere una ruta raíz. En otras palabras, la ruta del "proveedor" se convertiría en "nombre de la compañía / proveedor", la ruta del "cliente" se convertiría en "nombre de la compañía / cliente" y así sucesivamente. Si no proporciona una ruta, terminaría con un error como se muestra a continuación.



Por lo tanto, en su "Index.html" necesitamos agregar la etiqueta HTML BASE HREF como se muestra en el código resaltado a continuación. En este momento, no proporcionamos ningún directorio raíz.

<!DOCTYPE html>

<html>

<head>

<title></title>

<meta charset="utf-8" />

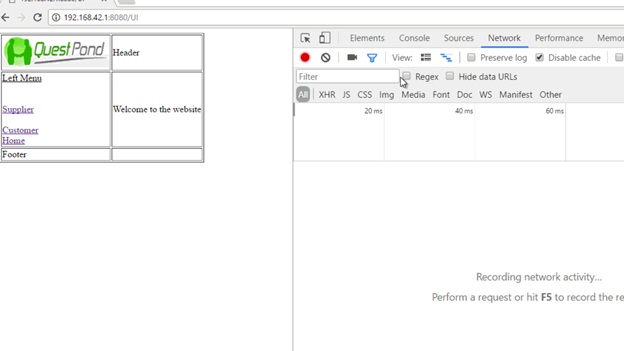
</head>

<base href="./">

<!—Other code has been removed for clarity -->

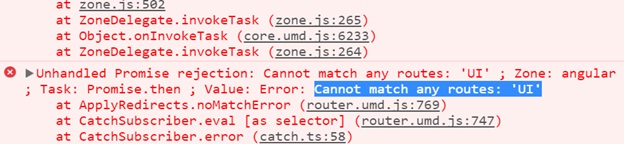
## Viendo la salida

Ahora ejecuta el sitio web y prueba el navegador a la carpeta UI. Puede ver que el logotipo se carga solo una vez y más adelante cuando el usuario hace clic en los enlaces del proveedor, la imagen de los enlaces del cliente no se carga una y otra vez.



## Corrección No se puede encontrar ningún error de ruta

Si haces un "f12" y compruebas en la parte de la consola del navegador chrome, verás el siguiente error. ¿Puedes adivinar cuál es el error?



Su aplicación angular actual está habilitada para la ruta. Por lo tanto, cada URL que se examina se busca en la recopilación de rutas.

Entonces, la primera URL que navegas es "/ UI" e intenta buscar en tu colección de rutas y no encuentra ninguna.

Para arreglar lo mismo, realice una entrada más para la ruta "UI" y apúntela a "WelcomeComponent".

export const ApplicationRoutes = [

    { path: 'Customer', component: CustomerComponent },

    { path: 'Supplier', component: SupplierComponent },

     { path: '', component:WelcomeComponent  },

    { path: 'UI', component:WelcomeComponent  }

];

Comprender el flujo

1.- El usuario final carga la página index.html.

2.-  Index.html desencadena systemjs y carga masterpage.html.

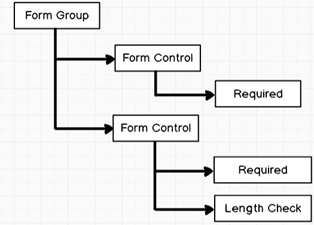
3.-  Cuando los usuarios finales hacen clic en enlaces de páginas maestras, otras páginas se cargan en marcadores de posición "enrutador de salida".

## Implementando la validación usando la forma angular

Las validaciones se aplican normalmente a las interfaces de usuario (Formularios) y las interfaces de usuario tienen controles y en los controles, aplicamos validaciones.

En la arquitectura de validación de forma angular, la estructura es la siguiente.

En la parte superior tenemos FormGroup, FormGroup tiene FormControls y FormControls tiene una o muchas validaciones.



Hay 3 pasos más para implementar la validación angular

    1.- Crear FormGroup.

    2.- Crea FormControl y con las validaciones adecuadas.

    3.- Asigne esas validaciones al Formulario HTML

¿Qué tipo de validación implementaremos ?

Implementaremos la siguiente validación en nuestra pantalla de Cliente:

    El nombre del cliente debe ser obligatorio.

    El código del cliente debe ser obligatorio.

    El código del cliente debe estar en el formato de A1001, B4004 y así sucesivamente.

El primer carácter debe ser un alfabeto de mayúscula seguido de un número de 4 letras.

Nota: - El código del cliente tiene validaciones compuestas.

Dónde poner Validaciones?

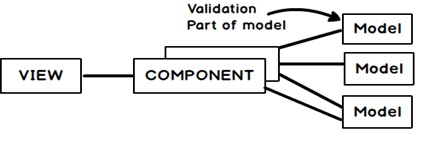
Antes de que empecemos con las validaciones, debemos decidir cuál es el lugar correcto para poner las validaciones. Si ve en Angular tenemos tres UI de sección más amplia, Componente y Modelo. Entonces pensemos cuál es la capa correcta para poner validaciones.

**IU:** - La IU se trata de la apariencia y el posicionamiento de los controles. Entonces poner validación en esta capa es una mala idea. Sí, en esta capa aplicaremos validaciones pero el código de validación debería estar en alguna otra capa.

**Componente**: - Componente es el código detrás (código de enlace) que vincula la interfaz de usuario y el modelo. En esta capa, se debe poner más actividad relacionada con el enlace. Un componente puede usar muchos modelos, de modo que si ponemos la lógica de validación aquí, también debemos duplicar eso en otros componentes.

**Modelo:** - Un modelo representa una entidad del mundo real como persona, usuario, administrador, producto, etc. Comportamiento Las validaciones de a.k.a son parte del modelo. Un modelo tiene Datos y Comportamiento. Entonces, el lugar correcto donde las validaciones deberían estar presentes es esta Capa.

Entonces, validemos la validación como parte del modelo.



## Importar componentes necesarios para validadores angulares

Entonces, el primer paso es importar los componentes necesarios para los validadores angulares en el modelo del cliente. Todos los componentes del validador angular están presentes en la carpeta "@ angular / forms". Necesitamos importar cinco componentes NgForm, FormGroup, FormControl y validadores.

**NgForm:** Etiquetas angulares para validación.

**FormGroup:** Nos ayuda a crear una colección de validación.

**FormControl y validadores**: Nos ayuda a crear validación individual dentro de FormGroup.

**FormBuilder:** Nos ayuda a crear la estructura del grupo de formulario y los controles de formulario. Tenga en cuenta que un grupo de Formularios puede tener muchos FormControls.

import {NgForm,

    FormGroup,

    FormControl,

    Validators,

    FormBuilder } from '@angular/forms'

Crear FormGroup usando FormBuilder

El primer paso es crear un objeto de FormGroup en el que tendremos una colección de validación. El objeto FormGroup se construirá utilizando la clase "FormBuilder".

formGroup: FormGroup = null; // Create object of FormGroup

var \_builder = new FormBuilder();

this.formGroup = \_builder.group({}); // Use the builder to create object

## Agregar una validación simple

Una vez que se crea el objeto FormGroup, el siguiente paso es agregar controles en la colección FormGroup. Para agregar un control, necesitamos usar el método "addControl". El primer parámetro en el método "addControl" es el nombre de la validación y el segundo es el tipo de validador angular que se agregará.

A continuación se muestra un código simple en el que estamos agregando un "CustomerNameControl" utilizando el formulario "Validators.required" FormControl. Tenga en cuenta que "CustomerNameControl" no es una palabra clave reservada. Puede ser cualquier nombre como "CustControl".

this.formGroup.addControl('CustomerNameControl', new

            FormControl('',Validators.required));

## Agregar una una validación compuesta

Si desea crear una validación compuesta, entonces necesita crear una colección y agregarla usando el método "redactar" como se muestra en el siguiente código.

var validationcollection = [];

validationcollection.push(Validators.required);

validationcollection.push(Validators.pattern("^[A-Z]{1,1}[0-9]{4,4}$"));

this.formGroup.addControl('CustomerCodeControl', new FormControl('', Validators.compose(validationcollection)));

## Código de modelo completo con validación

A continuación se muestra el código completo del Modelo de Cliente con las tres validaciones, tal como se discutió en la sección anterior. También hemos comentado el código para que pueda seguirlo.

// import components from angular/form

import {NgForm,

    FormGroup,

    FormControl,

    Validators,

    FormBuilder } from '@angular/forms'

export class Customer {

    CustomerName: string = "";

    CustomerCode: string = "";

    CustomerAmount: number = 0;

    // create object of form group

    formGroup: FormGroup = null;

    constructor(){

        // use the builder to create the

        // the form object

        var \_builder = new FormBuilder();

        this.formGroup = \_builder.group({});

        // Adding a simple validation

        this.formGroup.addControl('CustomerNameControl', new

            FormControl('',Validators.required));

        // Adding a composite validation

        var validationcollection = [];

        validationcollection.push(Validators.required);

        validationcollection.push(Validators.pattern("^[A-Z]{1,1}[0-9]{4,4}$"));

        this.formGroup.addControl('CustomerCodeControl', new

            FormControl('', Validators.compose(validationcollection)));

    }

}

## Referencia "ReactiveFormsModule" en CustomerModule.

// code has been removed for clarity.

import { FormsModule, ReactiveFormsModule } from "@angular/forms"

@NgModule({

    imports: [RouterModule.forChild(CustomerRoute),

                CommonModule,

                FormsModule,

                ReactiveFormsModule,

                HttpModule,

                InMemoryWebApiModule.forRoot(CustomerService)],

    declarations: [CustomerComponent , GridComponent],

    bootstrap: [CustomerComponent]

})

export class CustomerModule {

}

## Aplicar formGroup a formulario HTML

Lo siguiente es aplicar el objeto 'formGroup' al formulario HTML. Para eso tenemos que usar la etiqueta angular "[formGroup]" porque necesitamos especificar el objeto "formGroup" expuesto a través del objeto del cliente.

<form [formGroup]="CurrentCustomer.formGroup">

</form>

### Aplicar validaciones al control de HTML

Lo siguiente es aplicar el objeto 'formGroup' al formulario HTML. Para eso tenemos que usar la etiqueta angular "[formGroup]" porque necesitamos especificar el objeto "formGroup" expuesto a través del objeto del cliente.

<input type="text" formControlName="CustomerNameControl"

[(ngModel)]="CurrentCustomer.CustomerName"><br /><br />

Compruebe si las validaciones son correctas

Cuando el usuario comienza a completar los datos y cumple las validaciones, nos gustaría verificar si todas las validaciones son correctas y, en consecuencia, mostrar un mensaje de error o activar / desactivar los controles de la interfaz de usuario.

Para verificar si todas las validaciones son correctas, necesitamos usar la propiedad "válida" de "formGroup". A continuación se muestra un ejemplo simple donde el botón se desactivará dependiendo de si las validaciones son válidas o no. "[disabled]" es un atributo angular que habilita y deshabilita los controles HTML.

<input type="button"

value="Click" [disabled]="!(CurrentCustomer.formGroup.valid)"/>

Checando validaciones individuales

"CurrentCustomer.formGroup.valid" verifica todas las validaciones de "FormGroup", pero ¿qué ocurre si queremos verificar la validación individual de un control?

Para eso necesitamos usar la función "hasError".

"CurrentCustomer.formGroup.controls ['CustomerNameControl']. HasError ('required')" comprueba que para "CustomerNameControl" se ha cumplido la regla de validación "requerida". A continuación se muestra un código simple en el que se muestra un mensaje de error en una etiqueta "div" que es visible y no visible según si la función "hasError" devuelve verdadero o falso.

También tenga en cuenta el "!" (NO) antes de "hasError", que dice que si "hasError" es verdadero, entonces hidden debería ser falso y viceversa.

<div [hidden]="!(CurrentCustomer.formGroup.controls['CustomerNameControl'].hasError('required'))">Customer name is required </div>

### Elementos standalone

En nuestros formularios, tenemos tres cuadros de texto "CustomerName", "CustomerCode" y "CustomerAmount". En estos tres cuadros de texto, solo "CustomerName" y "CustomerCode" tienen validaciones, mientras que "CustomerAmount" no tiene validaciones.

Ahora, esto es un poco gracioso, pero si no especificamos las validaciones para un control de usuario que está dentro de una etiqueta "form" que tiene "formGroup" especificado, terminaría con una excepción larga como se muestra a continuación.

Error: Uncaught (in promise): Error: Error in ../UI/Customer.html:15:0 caused by:

ngModel cannot be used to register form controls with a parent formGroup directive. Try using

formGroup's partner directive "formControlName" instead. Example:

<div [formGroup]="myGroup">

<input formControlName="firstName">

</div>

In your class:

this.myGroup = new FormGroup({

firstName: new FormControl()

});

Or, if you'd like to avoid registering this form control, indicate that it's standalone in ngModelOptions:

Example:

<div [formGroup]="myGroup">

<input formControlName="firstName">

<input [(ngModel)]="showMoreControls" [ngModelOptions]="{standalone: true}">

</div>

Error:

ngModel cannot be used to register form controls with a parent formGroup directive. Try using formGroup's partner directive "formControlName" instead. Example:

<div [formGroup]="myGroup">

<input formControlName="firstName">

</div>

In your class:

this.myGroup = new FormGroup({

firstName: new FormControl()

});

Or, if you'd like to avoid registering this form control, indicate that it's standalone in ngModelOptions:

Example:

<div [formGroup]="myGroup">

<input formControlName="firstName">

<input [(ngModel)]="showMoreControls" [ngModelOptions]="{standalone: true}">

</div>

El error anterior se puede simplificar en tres puntos simples:

Dice que ha incluido un control HTML dentro de una etiqueta HTML FORM que tiene validaciones de forma angular.

Todos los controles especificados dentro de la etiqueta HTML FORM que tienen validación angular aplicada DEBERÍAN TENER VALIDACIONES.

Si un control HTML dentro de la validación de formulario angular no tiene validación, puede hacer una de las siguientes acciones para eliminar la excepción:

Debe especificar que se trata de un control independiente.

Mueva el control fuera de la etiqueta HTML FORM.

A continuación se muestra el código de cómo especificar "independiente" para validaciones angulares.

<input type="text" [ngModelOptions]="{standalone:true}"

También hablamos de que podemos eliminar del control de formularios y lo que sucede

### Complete code of Customer UI with validations applied

A continuación se muestra la interfaz de usuario completa del cliente con las tres validaciones aplicadas a los controles "CustomerName" y "CustomerCode".

<form [formGroup]="CurrentCustomer.formGroup">

<div>

Name:

<input type="text" formControlName="CustomerNameControl"

[(ngModel)]="CurrentCustomer.CustomerName"><br /><br />

<div [hidden]="!(CurrentCustomer.formGroup.controls['CustomerNameControl'].hasError('required'))">Customer name is required </div>

Code:

<input type="text" formControlName="CustomerCodeControl"

[(ngModel)]="CurrentCustomer.CustomerCode"><br /><br />

<div [hidden]="!(CurrentCustomer.formGroup.controls['CustomerCodeControl'].hasError('required'))">Customer code is required </div>

<div [hidden]="!(CurrentCustomer.formGroup.controls['CustomerCodeControl'].hasError('pattern'))">Pattern not proper </div>

Amount:

<input type="text"

[(ngModel)]="CurrentCustomer.CustomerAmount"><br /><br />

</div>

{{CurrentCustomer.CustomerName}}<br /><br />

{{CurrentCustomer.CustomerCode}}<br /><br />

{{CurrentCustomer.CustomerAmount}}<br /><br />

<input type="button"

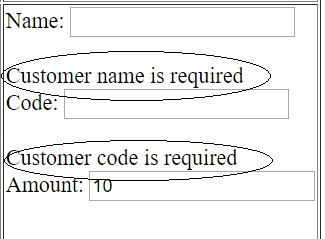
value="Click" [disabled]="!(CurrentCustomer.formGroup.valid)"/>

</form>

Write reactive forms

### Run and see your validation in action

Una vez que haya terminado, debería poder ver la validación en acción, como se muestra en la figura siguiente.



### Dirty , pristine , touched and untouched

Explicación de la propiedad

**Dirty:** Esta propiedad indica si el valor ha sido modificado.

**Pristine:** Esta propiedad dice si el campo ha cambiado o no.

**Touched:** cuando ocurre el enfoque perdido para ese control.

**Untouched:** El campo no se toca.

# Bibliografía

<https://www.codeproject.com/Articles/1207118/Learn-Angular-Tutorial#How_does_this_article_series_teach_you_Angular_>

<https://www.infoworld.com/article/3213244/javascript/whats-new-in-angular-5-easier-progressive-web-apps.html>

<https://www.udemy.com/the-complete-guide-to-angular-2/>

<https://angular.io/guide/quickstart>