# **No se encuentran entradas de índice.**

# Temario para Angular

# 1 Introducción a Angular 5

¿Angular5 o angular.io?

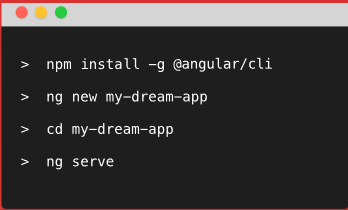
**Angular** (comúnmente llamado "**Angular** 2+" o "**Angular** 2"), es un framework para aplicaciones web desarrollado en TypeScript, de código abierto, mantenido por Google, que se utiliza para crear y mantener aplicaciones web de una sola página.

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. Te puede dar una idea inicial de qué es lo que encontrarás en estas carpetas y cómo usar el proyecto. Complementará sin duda las explicaciones de este [manual de Angular](https://www.desarrolloweb.com/manuales/manual-angular-2.html). Encontrarás algunos comandos explicados del CLI, como "ng serve" el cuál ya hemos tratado, para servir el proyecto. También comentará que tienes que hacer para realizar el build, comando "ng build" y llevar a producción una aplicación.

**.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.

Definiciones comunes

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 caracteristicas 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

Programación orientada a componentes

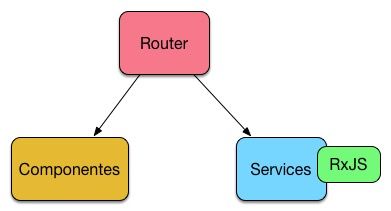
La **programación orientada a componentes** (que también es llamada *basada en componentes*) es una rama de la [ingeniería del software](https://es.wikipedia.org/wiki/Ingenier%C3%ADa_del_software), con énfasis en la descomposición de sistemas ya conformados en componentes funcionales o lógicos con [interfaces](https://es.wikipedia.org/wiki/Interfaces) bien definidas usadas para la comunicación entre componentes.

Se considera que el nivel de [abstracción](https://es.wikipedia.org/wiki/Capa_de_abstracci%C3%B3n) de los componentes es más alto que el de los [objetos](https://es.wikipedia.org/wiki/Objeto) y por lo tanto no comparten un estado y se comunican intercambiando mensajes que contienen datos.

# 3 Componentes

## Arquitectura de una aplicación

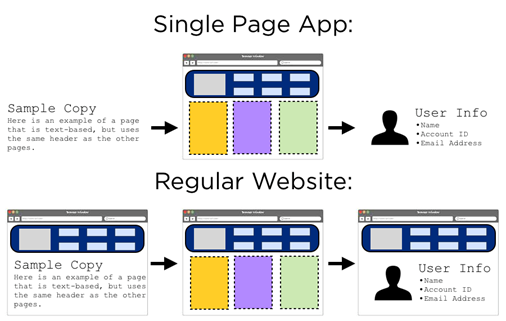
**Arquitectura:** Podemos decir que hay muchos cambios, **pero la arquitectura MVC se mantiene**. Disponemos de Componentes para la vista , Enrutador para la capa de control y servicios para la capa de backend.

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

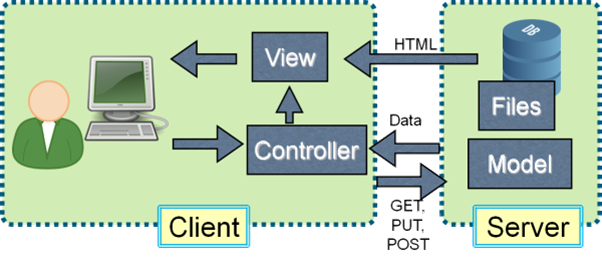
**Inyección de dependencias**: Soportan de forma muy natural las inyecciones de dependencias apoyándose **en TypeScript.**

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

Con Angular 4 podemos crear Single Page Applications (SPA):



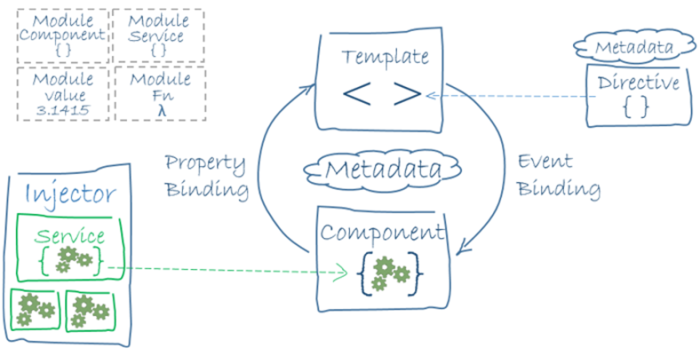
Las SPA’s gestionan toda la interactividad con el usuario desde el lado cliente, cargando recursos del servidor (vistas, datos… etc.) vía AJAX



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**.

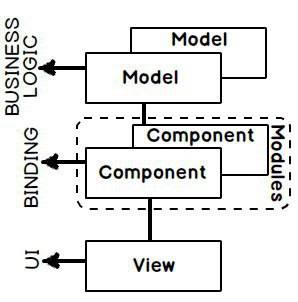
En el siguiente esquema se puede observar gráficamente la Arquitectura de Angular 4:



<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: - Esto tendrá 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. No se preocupe por las configuraciones que hay en él, lo explicaremos a medida que avancemos. Recuerde, NO APRENDE MÁS. 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. Vincula la interfaz de usuario HTML con el código JavaScript (modelo).

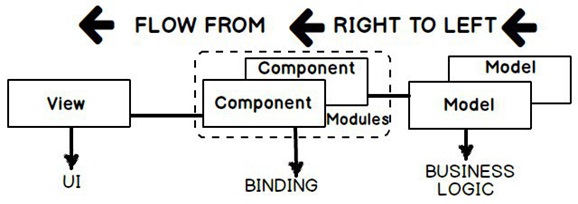
Entonces, si visualizamos, se verá algo como se muestra en la imagen a continuación. Así que vayamos de derecha a izquierda. Así que hagamos la codificación en la siguiente secuencia:

    Crea el modelo.

    Crea el Componente.

    Crea el módulo.

    Crea la IU HTML.



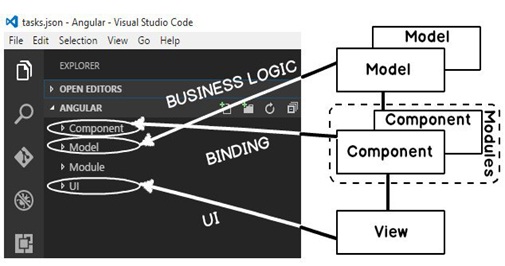
Primero, creemos cuatro carpetas en nuestro proyecto:

    Ver carpeta: esta carpeta contendrá la IU HTML.

    Carpeta modelo: esta carpeta tendrá las principales clases de mecanografía comerciales.

    Carpeta de componentes: esta carpeta tendrá el código de enlace que vincula la interfaz de usuario HTML y el modelo.

    Módulo: - 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. Así que vamos a crear primero el modelo de cliente con tres propiedades "CustomerName", "CustomerCode" y "CustomerAmount".

Así que haga clic derecho en la carpeta "Modelo" y 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".

En "Customer.ts", creemos una clase "Cliente" con tres propiedades. En este libro no repasaremos los aspectos básicos de la mecanografía, por favor revise este video de 1 hora de mecanografía que explica la mecanografía con más detalle.

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.

Así que haga clic derecho en la carpeta del componente y agregue el archivo "CustomerComponent.ts" como se muestra en la figura a la izquierda.

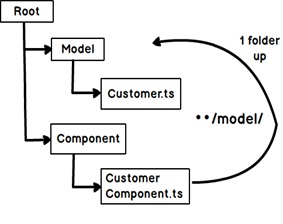
En el componente, necesitamos importar dos cosas, el núcleo angular y nuestro modelo de cliente. Tenga en cuenta que "importar" es una sintaxis de mecanografía y no de JavaScript. Entonces, en caso de que no sigas el código, mira este Learn Typescript en 1 hora de video antes de seguir adelante.

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 "exportar" en el archivo "Customer.ts". La importación y la exportación generan código que sigue las especificaciones CommonJs, AMD o UMD. En caso de que usted sea nuevo en estas especificaciones, consulte este video de CommonJs que explica el protocolo en más detalle.

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



Tratemos de entender cómo se ubica el componente "Cliente". Si ves que está usando una ruta relativa. En la importación, dice "../Model/Customer".

El "../" dice ir una carpeta arriba. Por lo tanto, actualmente estamos en la carpeta "Componente", por lo que con "../" viaja a la raíz y desde ese punto ingresa a la carpeta "Modelo".

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();

}

## Creating the 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 { }

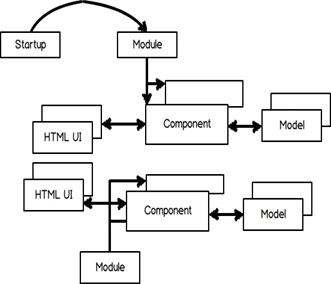
import { NgModule }      from '@angular/core';

## 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);

## Invoking “Startup.ts” file using main angular page

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 the typescript compiler

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 the http-server

Entonces, si todo está bien en el Paso 11, ahora debemos ejecutar el servidor http. Para ejecutar el servidor, debemos escribir "http-server" en el terminal integrado del código VS como se muestra en la figura.

## Implementing SPA using 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.

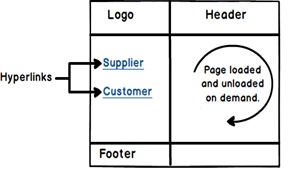
En el lado derecho hay un sitio web sencillo donde tenemos el logotipo y el encabezado en la parte superior, los enlaces de menú a la izquierda y el pie de página en la parte inferior.

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".



### Creating the Master Page

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

amos 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.

A continuación se muestra el texto de las páginas del proveedor.

Esta es la página de Proveedores

A continuación se muestra el texto de las páginas de bienvenida.

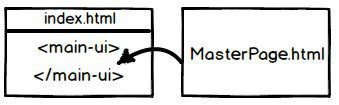
Bienvenido al sitio web

### Renaming placeholder in Index.html

Como se explica en la Parte 1, "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".



### Removing selector from 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.

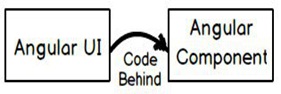
### Step 5 :- Creating Components for Master , Supplier and Welcome page

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 {

}

### Step 6 :- Creating the routing constant collection

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 = [

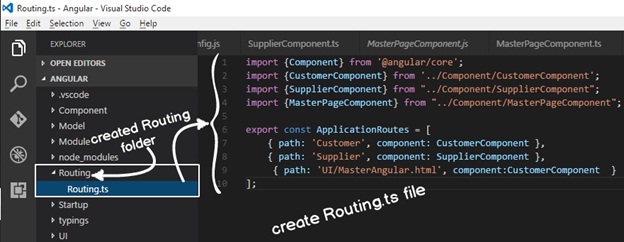
    { path: 'Customer', component: CustomerComponent },

    { 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".



### Defining routerLink and router-outlet

La navegación (rutas) definida en "Paso 6" en el collectin 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]"

Ocultar código de copia

<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.

Ocultar código de copia

<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.

Ocultar código de copia

<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>

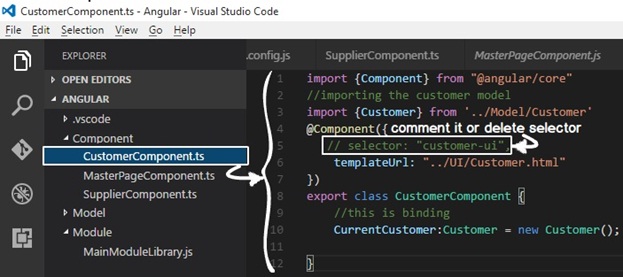
</tr>

<tr>

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

</tr>

</table>



El código final de "CustomerComponent.ts" se vería como se muestra a continuación.

import {Component} from "@angular/core"

//importing the customer model

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

@Component({

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

})

export class CustomerComponent {

    //this is binding

    CurrentCustomer:Customer = new Customer();

}

# 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>