**Angular**

**Referencias:**

* <https://angular.io/cli>

**Comandos:**

* **ng new <ProjectName>** 🡪 Para generar una nueva aplicación con Angular

NOTA:

En versiones más recientes (Angular 17+) ya no se agrega por default el archivo app.module ya que estás versiones trabajan con la versión standalone, para agregar el archivo app.module se debe ejecutar: **ng new <ProjectName> --standalone=false**

* **ng g**

Par ver el detalle completo del comando, consultar <https://angular.io/cli/generate>

**Por ejemplo:**

* + **ng g c <componentName> --module <moduleName>** 🡪 Se genera un componente en el módulo indicado.
  + **ng g m <moduleName> 🡪** Se genera un nuevo módulo

**ng g m <moduleName> --flat 🡪** Para indicar que el module se debe generar en la raíz del directorio donde se está generando el módulo.

* + **ng g m <moduleName> --routing 🡪** Se genera un módulo normal y routing module.
  + **ng g m <newModuleName> --module = <addToModuleName> 🡪** Se genera un módulo (newModuleName) y se agrega a otro módulo (addToModuleName)

* **ng serve** 🡪 Para levantar el proyecto (Ejecutar dentro del directorio del proyecto)

**Características de Angular:**

* **Single Page Application**

Angular es un framework para el diseño/desarrollo de aplicaciones DE UNA SOLA PÁGINA.

* **POO** (Programación Orientada a Objetos)
* **Herencia** (extends)
* **Clases** **Abstract**
* **Interfaces (implements)**

Funcionan como en Java, con la diferencia de que en las interfaces de TypeScript se deben definir los atributos y las firmas de los métodos.

* **Inyección de Dependencias (Como patrón y como framework)**

Angular se basa en la Inyección de Dependencias para la gestión de los recursos (objetos) que vaya requiriendo.

Mediante la Inyección de Dependencias se le da flexibilidad a nuestro código (Se evita un fuerte acoplamiento entre los objetos relacionados)

Angular crea y gestiona las instancias de los objetos requeridos (Mediante Inyección de Dependencias)

**La Inyección de Dependencias se realiza mediante el constructor de las clases y para configurar las clases que requieren manejar mediante Inyección de Dependencias, se utiliza Injector (@angular/core)**

**Texto

Descripción generada automáticamente**

**Jerarquía de Inyección de Dependencias**

Si un componentes hijo no tiene una dependencia, trata de obtenerlos de su padre, si su padre tampoco tiene la dependencia requerida, entonces, se va a buscar un nivel más arriba, y así sucesivamente.

Interfaz de usuario gráfica

Descripción generada automáticamente

* **readonly**

Atributos de clase de solo lectura

* + readonly nombre: string; 🡪 Atributo de clase de solo lectura.
  + private nombre: string; 🡪 Atributo de clase que solo se puede acceder desde la clase
  + protected nombre: string; 🡪 Atributo de clase que se puede leer en la clase donde fue declarada y en las clases que la extiendan.
  + nombre: string; 🡪 Son atributos públicos (Se puede omitir la palabra reservada public)

**NOTA:**

Existe otra forma de declarar un atributo de clase de solo lectura (readonly), esto se realiza mediante lo métodos getter/setter, si únicamente se definen los métodos getter, TypeScript infiere que se trata de un atributo de clase de solo lectura al no tener definido los métodos setter.

* **Decoradores / Anotaciones**

Mediante los decoradores/anotaciones se agregan metadatos y/o funcionalidad a las clases.

Un decorador/anotación es una característica del lenguaje JavaScript que se implementa en TypeScript.

**Las clases en Angular se decoran/anotan de acuerdo con la funcionalidad que van a tener.** Por ejemplo:

**@ngmodule** 🡪decorador/anotación para indicar que una clase está fungiendo como un módulo.

**@Injectable 🡪** decorador/anotación para indicar que una clase está fungiendo como una clase de tipo Servicio (Los servicios se incluyen en el módulo principal de la aplicación – app.module.ts).

**@Pipe 🡪** decorador/anotación para indicar que una clase está fungiendo como una clase de tipo pipe.

**@providers 🡪** decorador/anotación para indicar los servicios que se requieren en un componente.

* **Manejo de Observables (RxJS )**

Mediante los observables, angular realiza el manejo de las respuestas. Los observables son la alternativa a las Promesas.

Mediante los Observables se implementan las peticiones asynchronize.

Características de los Observables:

* + Son declarativos, es decir, su ejecución inicia hasta que se realiza la suscripción al observable (Las Promesas se ejecutan en cuanto son creadas).
  + Proporcionan múltiples valores (Las promesas proporcionan un solo tipo de valor).
  + Diferencian entre encadenamiento (de métodos) y la suscripción, lo que permite que los observables puedan generar métodos de transformación complejos sin hacer que la operación se ejecute (Las Promesas solo tienen clausulas then),
  + La suscripción al Observable es el responsable del manejo de errores
  + Tiene la capacidad de ejecutarse n veces (Las promesas se ejecutan una sola vez)
  + La suscripción a las promesas son cancelables – Se evita que los suscriptores reciban más notificaciones – (Las Promesas no son cancelables)

**Arquitectura de una aplicación Angular:**

**Interfaz de usuario gráfica, Diagrama

Descripción generada automáticamente**

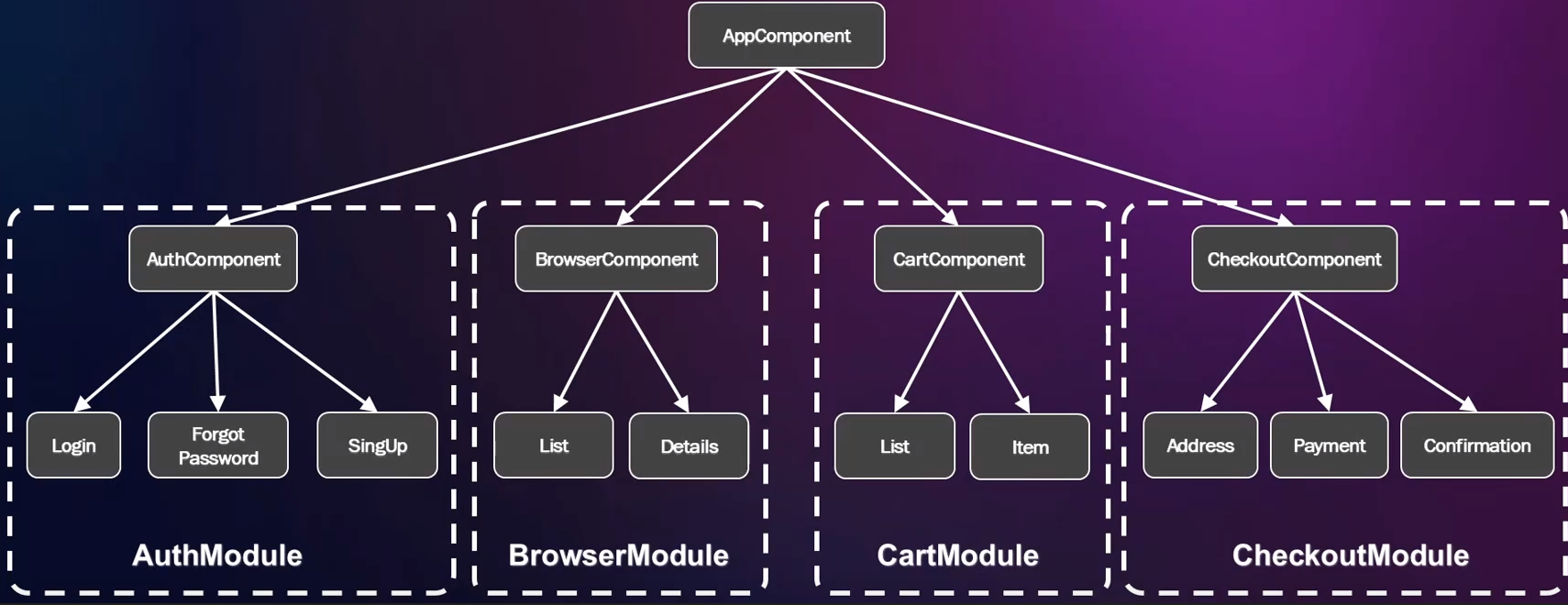
* **Componentes de Angular**

**Diagrama

Descripción generada automáticamente**

* **Módulos**

Mediante los módulos se realiza la agrupación de componentes (El equivalente a los package de Java), por consiguiente, todos los componentes que se generen, se deben dar de alta en al modulo correspondiente (Por ejemplo, app.module.ts)



NOTA:

**export/import** para consumir componentes de un mismo módulo

Para que los elementos (clases, variables, atributos de clase, métodos, etc…) de un componente se puedan consumir en otros componentes del **mismo módulo** se debe utilizar **export/import**

Mediante **export** se expone el recurso a compartir.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Mediante **import** se incluye el componente a utilizar

Captura de pantalla de un celular

Descripción generada automáticamente

**export/import** para consumir componentes de diferentes módulos

Para que los elementos (clases, variables, atributos de clase, métodos, etc…) de un componente se puedan consumir en otros componentes de **diferentes módulo**s se debe utilizar **export/import** en las clases (ver el escenario anterior) y para el caso de los módulos de la siguiente manera:

Mediante **export** se exponen los componentes del módulo a compartir (En el export, se indica cada uno de los componentes a exponer)

Texto

Descripción generada automáticamente

Mediante **import** se incluyen los componentes requeridos de otros módulos (En el import, se indica el module que contiene los componentes a incluir).

Texto

Descripción generada automáticamente

Las Clases, variables, atributos de clase, métodos definidos dentro de un módulo tienen como ámbito su propio módulo, la comunicación de componentes (clases, variables, atributos de clase, métodos, …) entre módulos, se realiza mediante las palabras **export/import**

**@NgModule(…, providers[lista de servicios], …)**

En el módulo principal (app.module.ts) se deben incluir los servicios de la aplicación

* **Aspectos importantes de la Arquitectura de Angular**
  + **Flujo de datos unidireccional**

La información entre componentes viaja únicamente en una sola dirección.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

* + **Inyección de Dependencias.**

La inyección de Dependencias se realiza a través de los constructores de las clases.

* + **Directivas** (Parámetros de las etiquetas HTML)

Mediante las directicas se extiende/agrega la funcionalidad del HTML usando una nueva sintaxis.

Texto

Descripción generada automáticamente

**Tipos de Directivas:**

Existen 3 tipo de directivas:

* + - Directivas de Componente
      * Las que proporciona Angular (Son los decoradores)
        + @Component
        + @Module
      * Las personalizadas (custom)
    - Directivas de Estructura
      * ngIf
      * ngFor
      * ngSwitch
      * ngPlural
      * ngTemplate
      * ngComponentOutlet
    - Directivas de Atributo
      * ngClass
      * ngStyle
      * ngModel
      * …

NOTA:

Los selectores de los componentes (clases de angular anotadas con @Component) son una especie de directivas, por ejemplo:

Texto

Descripción generada automáticamente

* + **Template / Plantillas** (Instrucciones HTML asociado al componente)

En angular siempre se mantiene separada la plantilla del componente.

Texto

Descripción generada automáticamente

* **Encapsulación de la vista (Ámbitos de estilos CSS)**

Se definen dentro de @Component

Texto

Descripción generada automáticamente

Existen 3 tipos de encapsulación:

* + **Emulated**

Encapsulamiento por default, el código CSS del componente únicamente afecta al componente y si se permite heredar estilos globales. Para ello, se modifican los selectores CSS del componente para que solo se apliquen a la vista del componente y no se afecten a otros elementos de la aplicación. Se emula el comportamiento del ShadowDom

(Los estilos del componente no se desbordan y si permite heredar estilos globales)

* + **ShadowDom (Native)**

El código CSS del componente se aplica únicamente para el componente (ignora los estilos globales). Para ello, se utiliza el ShadowDom del navegador (estilos globales) para agregar de los estilos del componente en un ShadowRoot

(Los estilos del componente no se desbordan y además se ignoran los estilos globales)

* + **None**

No se aplica encapsulamiento del código CSS del componente, lo que significa que, los estilos del componente se aplican globalmente y afectan a otros componentes de la aplicación. (Los estilos del componente se desbordan a otros componentes)

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

* **Data Binding (Enlace de datos)**

**Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente**

**NOTA: Esto también se puede aplicar para directivas de estilos, por ejemplo: tablas dinámicas, estilos dinámicos.**

**Tipos de data binding:**

* + **Interpolación:**

{{value}}

No permite modificar valores

* + **Binding de propiedades:**

[property] = “value”

Permite asignarle valores a la propiedad

* + **Binding de eventos:**

(evento) = “handler”

Permite asignar el eventos a ejecutar, por ejemplo: (click) = “saludar()”

* + **Binding bidireccional:**

[(ng-model)] = “property”

Permite las acciones del “binding de propiedad” y el “binding de evento”

Ejemplo:

En el componente padre (.ts) se debe definir una variable de tipo:

**dataParent: dataParent: EventEmitter<string> = new EventEmitter<string>();**

En el componente padre (.html) se debe definir:

**<app-child [dataParent]="data"></app-child> 🡪** Para invocar el componente hijo

**<input type="text" name="test" id="test" placeholder="test" [(ngModel)]="dataParent" />**

Y en el componente hijo (.ts) se define:

**@Input() dataParent: string='';**

* **Decoradores @Input, @Output**

Modificadores mediante los cuales se comparten datos entre un componente padre y un componente hijo.

Diagrama

Descripción generada automáticamente

* **Ciclo de vida / lifecycle hook (Ganchos de ciclo de vida)**

Los **Componentes** y **Directivas** de Angular tienen un **ciclo de vida** (Se crean, se actualizan y se destruyen), a continuación, se mencionan las **interfaces** mediante los cuales se permite **acceder** al ciclo de vida de Directivas y Componentes de Angular.

Cada interface tiene un único método de alcance cuyo nombre en el nombre de la interface con el prefijo ng, por ejemplo, Interface: OnInit 🡪 método: ngOnInit()

**Importante:**

Angular realiza la gestión de los ciclos de vida de los Componentes y Directivas.

**Secuencia de ejecución:**

Captura de pantalla con letras

Descripción generada automáticamente con confianza media

**ngOnChange:**

Identifica la **asignación/reasignación** de propiedades de **entrada** vinculados a datos. (Como se puede observar en la imagen de arriba, también se ejecuta después de que se ejecuta el constructor)

Se llama antes de ngOnInit() (si el componente tiene entradas vinculadas a datos) y cada vez que una o más propiedades de entrada vinculadas a datos cambian.

**ngOnInit:**

Inicializa las propiedades de un Componente/Directiva.

Se ejecuta una sola vez (Si existen propiedades de entrada, se ejecuta después del primer ngOnChange())

**ngDoCheck:**

Maneja cambios que Angular no puede o no detectará por sí mismo.

Se ejecuta cada vez que se detectan cambios y después de ejecutar ngOnInit()

**ngAfterContentInit:**

Responde después de proyectar contenido externo en la vista del componente o en la vista en la que se encuentre una directiva.

Se llama una sola vez después del primer ngDoCheck().

**ngAfterContentChecked:**

Responde después de que Angular verifica el contenido proyectado en el componente o directiva.

Se llama después de ngAfterContentInit() y en cada ngDoCheck() subsiguiente.

**ngAfterViewInit:**   
Responde después de que Angular inicialice las vistas del componente y las vistas secundarias, o la vista que contiene la directiva.

Se llama solo una vez después del primer ngAfterContentChecked().

**ngAfterViewChecked:**

Responde después de que Angular verifique las vistas del componente y las vistas secundarias, o la vista que contiene la directiva.

Se llama después de ngAfterViewInit() y en cada ngAfterContentChecked() subsiguiente.

**ngOnDestroy:**

Realiza limpieza justo antes de que Angular destruya la directiva o componente. Cancela la suscripción a Observables y desvincula los manejadores de eventos para evitar fugas de memoria.

Se llama inmediatamente antes de que Angular destruya la directiva o componente

* **View Queries (Técnicas de interacción entre componentes)**

Mediante los cuales se aplican las técnicas de interacción entre componentes.

**View Child**

Se especifica mediante el decorador @ViewChild

**View Children**

Se especifica mediante el decorador @ViewChildren

**Content Child**

Se especifica mediante el decorador @ContentChild

**Content Children**

Se especifica mediante el decorador @ContentChildren

* **Routing**

Mediante el Routing se realiza el manejo de los enrutamientos en las aplicaciones desarrolladas con angular. Una parte importante del Routing se maneja mediante los Guards.

**Guards (Guardias):**

Los Guards son simples clases (Permiten Inyección de Dependencias) que contiene funciones cuyo valor de retorno es un boolean, un Observable o una Promesa que resuelven booleans, mediante los Guards se realizan:

* + Comprobaciones antes de navegar hacia/desde diferentes rutas de la aplicación.
  + Restringe el acceso a determinadas rutas de la aplicación a determinados usuarios.
  + Valida/confirma antes de navegar fuera de las rutas.
  + Una sola ruta puede tener múltiples protecciones, se verifican en el orden en el que son inyectas.

**Tipos de Guards:**

* + **CanActivate:** Comprueba si un usuario puede visitar una ruta.
  + **CanActivateChild:** Comprueba si un usuario puede visitar una ruta hija.
  + **CanDeactivate:** Comprueba si un usuario puede abandonar una ruta.
  + **CanLoad:** Comprueba si los módulos de carga diferida (Lazy-loaded) deben cargarse.
  + **Resolve:** Realiza la recuperación de datos de ruta antes de la activación de la ruta.

**Configuración de Routers:**

Para el tema de la configuración de los routing, existe la forma normal y Lazyload.

**Configuración normal de routers:**

Para la configuración de las rutas se realizan las siguientes configuraciones:

* <name>-**routing.module.ts**

**Definición de arreglo de rutas**

A manera de ejemplo, el arreglo de las rutas se realiza en **PaisesRoutingModule**

Texto

Descripción generada automáticamente

**Importación de las rutas**

**Texto

Descripción generada automáticamente**

**Exportación de las rutas**

**Texto

Descripción generada automáticamente**

* **App.module.ts**

Se importa el módulo de routing (ver **PaisesRoutingModule**)

Texto

Descripción generada automáticamente

* **app.component.htmp**

**routerLink** para definir las rutas de los componentes

**<router-outlet></router-outlet>** para renderizar el contenido de las rutas

Texto

Descripción generada automáticamente

**Lazyload**:

<https://www.youtube.com/watch?v=K6XsI0svtwE> 🡪 Referencias

Mediante esta opción, se realiza la configuración de rutas por módulos y posteriormente se dan de alta en el routing principal de la aplicación. El proceso de configuración de las rutas se realiza de la siguiente manera:

**Configuraciones a nivel módulo**

* **auth-routing.module.ts (module-routing.module.ts)**

A manera de ejemplo, se utiliza el módulo auth. En este archivo se realizan las siguientes configuraciones:

* + **Crear arreglo de rutas hijas**

**Texto

Descripción generada automáticamente**

* + **Importar las rutas del módulo**

**Texto

Descripción generada automáticamente**

* + **Exportar las rutas del módulo**

**Texto

Descripción generada automáticamente**

* **auth.module.ts**

En este archivo se realizan las siguientes configuraciones:

* + **Importar el archivo auth-routing (module-routing)**

**Texto, Chat o mensaje de texto

Descripción generada automáticamente**

**Configuraciones a nivel aplicación**

* **app-routing.module.ts**

En este archivo se realizan las siguientes configuraciones:

* + **Crear arreglo de rutas**

**Texto

Descripción generada automáticamente**

En la imagen de arriba se realiza la configuración para consumir los servicios del módulo auth.

* + **Importar las rutas**

**Texto

Descripción generada automáticamente**

* + **Exportar las rutas**

**Texto

Descripción generada automáticamente**

* **app.module.ts**

En este archivo se realizan las siguientes configuraciones:

* + **Importar el archivo routing**

@NgModule(…, imports: […, appRoutingModule, …], …)

Se le indica a Angular de la existencia del archivo de rutas.

* **app.component.html**

En este archivo se configuran los links de y se renderiza el contenido de las páginas asociadas a los routing configurados.

**<a routerLink=”path”>**<linkName>**</a>** para configurar los link de los componentes.

**<router-outlet></router-outlet>** para renderizar el contenido de las plantillas (html) de los componentes.

**Texto

Descripción generada automáticamente**

* **Finalmente, para consumir (poder visualizar las rutas configuradas)**

Consultar la url: <http://localhost:4200/auth>

**NOTA:**

En la url se debe incluir el nombre de alguno de los módulos donde se configuraron los routing.

**Tipos de rutas:**

* + **Path Absoluto:** Denotado con **‘/’**
  + **Path Relativo:** Denotado con ‘**./’**
  + **Path Principal:** denotado con **‘../’**
* **navigate**

Nos permite navegar entre componentes desde las clases, para ello, se deben realizar las siguientes configuraciones:

Mediante la **Inyección de Dependencias**, en el constructor realizar la inyección de las dependencias requeridas:



En el método que se invoca desde algún evento (click, …), realizar la navegación hacia el componente al que se quiere redirigir:

Texto

Descripción generada automáticamente

Y en el componente al que se redirige la petición, se realiza lo siguiente, para obtener el parámetro que se pasa en la url:

**Location**

Mediante esta librería podemos navegar entre los componentes de la aplicación.

Por ejemplo:



Texto

Descripción generada automáticamente

Instrucción para navegar al componente anterior (**this.location.back()**)

Texto

Descripción generada automáticamente

* **JSON Server (Servidor de pruebas)**

En el que se pueden realizar pruebas de los servicios, para ello, se debe utilizar **HttpClient**

JSON Server es un paquete npm que nos proporciona una API Rest con datos de prueba en formato JSON.

**¿Cómo conectar angular con JSON Server?**

* + **app.module.ts**

Se importa HttpClientModule

Texto

Descripción generada automáticamente

* + **servise.ts**

Se importa HttpClient, Observable; Se define la ruta (JSON Server) de donde debemos obtener los datos; Mediante Inyección de Dependencias, se inyecta una dependencia de HttpClient en el constructor.

Texto

Descripción generada automáticamente

* + **component.ts**

En los componentes donde se consuman los servicios, extraer la información de la siguiente manera:

Texto

Descripción generada automáticamente

* **Observables**

Similares a los Promise, con la diferencia de que los Observables nos proporcionan

Para el manejo de Observables se requiere tener instalado rxjs (Angular ya lo tiene preinstalado, por lo que ya no se requieren realizar instalaciones)

**Consultar referencia:** <http://reactivex.io/> se pueden consultar las funciones con las que disponen los Observer

**ReactiveX** nos permite el manejo de flujos y eventos a partir de patrón Observer, Iterator y la programación funcional

**NOTA:**

Para obtener la información de un Observer, antes nos tenemos que suscribir al Observer. RECORDEMOS QUE LOS OBSERVER SE BASAN EN EL PATRON DE SIDEÑO OBSERVER y la programación funcional.

Los elementos de un Observer llegan en tiempos diferentes, por Ejemplo:

let dataObservers = of(1,2,3,4,5,6,7,8,9,10);

Primero llega el valor 1, luego el 2, …, al final llega el valor 10

La información de un Observer **no** se puede consumir hasta que el flujo (Carga de datos) se termine.

Por consiguiente, los elementos de un Observer se consumen con forme van llegando, es decir, no se puede obtener el elemento 3 sin antes haber obtenido/consumido los elementos 1, 2.

Imagen que contiene Diagrama

Descripción generada automáticamente

* **Subject**
  + Es un observable especial, funciona como Observador y como Observable.
  + Al ser un Observable, nos podemos suscribir.
  + Un Subject se puede suscribir a otros Observers.
  + Un Subject puede actuar como un proxy entre la fuente Observable y varios Observers, lo que permite que varios Observers compartan la misma ejecución Observable.
  + **Pueden usarse para compartir datos entre componentes**

Su funcionamiento se base en el patrón de diseño Observer, en cuanto se realiza un cambio en el Subject, se notifica a todos los suscriptores.

**Configuraciones requeridas:**

Para el siguiente ejemplo, se realiza la configuración de una variable definida en un service y que se comparte en varios componentes (hijo-uno.component.ts, hijo-dos.component.ts)

* + **pais.service.ts**

Definir las siguientes variables:





Texto

Descripción generada automáticamente

* + **hijo-uno.component.ts, hijo-uno.component.ts**

|  |  |
| --- | --- |
| Hijo uno | Hijo dos |
| Texto  Descripción generada automáticamente |  |

* + **hijo-uno.component.html, hijo-uno.component.html**

|  |  |
| --- | --- |
| Hijo uno | Hijo dos |
|  |  |

* + **padre.component.html**

****

* **Servicios**

Los servicios son clases de typescript que se decoran/anotan con @Injectable

Para el manejo de servicios se realizan las siguientes configuraciones:

* + **Creación del servicio**

**Texto

Descripción generada automáticamente**

* + **Importar el servicio**

En el módulo correspondiente, importar el servicio. Por ejemplo, en la siguiente imagen se importa el servicio al **app.module.ts** mediante **providers: [PaisService]**

Texto

Descripción generada automáticamente

* + **Implementar las funciones de los servicios**

Mediante la Inyección de Dependencias, en el componente requerido se inyecta el servicio requerido (la inyección se realiza a través del constructor del componente). Por ejemplo, en la siguiente imagen se inyecta una dependencia del servicio **PaisService** en el componente **buscar**

Texto

Descripción generada automáticamente

* **HttpClient, HttpHeader, HttpParams**

Para el consumo de servicios externos (Servicios API Rest), para implementar HttpHeader y HttpParams se realizan las siguientes configuraciones:



Captura de pantalla de computadora

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

* **s**