**ANGULAR DE CERO A EXPERTO**

**Inyección de dependencias es algo muy importante de Angular.**

**Aprender Angular Con Redux??? Para el manejo de estados**

**JAVASCRIPT**

- Ha ido creciendo y siendo mas demandado con el tiempo ( Ahora hasta backend )

- No fue diseñado para crear aplicaciones de mediana a gran escala

- Surgieron frameworks -> Angular, react, view

- Js es de tipado débil y en apps muy grandes puede ser algo complejo

- Antes de ES6 carecía de clases y módulos

- ES5 está siendo aceptado mundialmente

**LET Y CONST**

LET y const son del estándar ES6

Un estándar mas atrás (ES5)

var mensaje = 'Hola';

console.log(mensaje));

Estándar mas actual puedo usar let donde usaba var

let -> Estas variables existen dentro del scope {} --> Es mas eficiente que usar var --> Con let tengo un scope mas específico {}

const -> Si nunca va a cambiar en el programa, pesan menos porque no tienen funciones para modificarlas.

Función autoinvocada es la base para los módulos en javascript, para que no quede en el global y no se choque o colapse el código

--> Esto de abajo se llama el patrón módulo

(function(){

})();

**TYPESCRIPT**

Es similar a programar en un lenguaje más común de programación

- Lo mantiene Microsoft

- Extensión .ts

- TYPESCRIPT es un superconjunto de javascript (Superset)

- Código más ordenado y comprensivo

- Ofrece tipado estricto y flexible

**TODO LO QUE SIGUE ACÁ ANGULAR YA LO HACE PERO VALE APRENDERLO**

En consola: tsc --init

Esto crea el archivo de configuración de typescript --> tsconfig.json --> Aquí decimos como queremos que compile o como queremos que trabaje typescript

Ya con el archivo se puede usar tsc --watch y lo compila a javascript

**TIPOS DE DATOS EN TYPESCRIPT**

let cualquiercosa; --> Queda de tipo any que es un tipo de dato que soporta cualquier cosa. --> Deberíamos evitarlo.

*let* nombre: *string* = 'Strider';

*let* hp: *number* | *string* = 95;

*let* estaVivo: *boolean* = true;

hp = 'FULL';

OBJETOS EN TYPESCRIPT

*interface* Personaje {

    nombre: *string*;

    hp: *number*;

    habilidades: *string*[];

    puebloNatal?: *string*;

}

*const* personaje: Personaje = {

    nombre: 'Strider',

    hp: 100,

    habilidades: ['Bash','Counter','Healing']

}

personaje.puebloNatal = 'Pueblo Paleta';

Una interface es una serie de llaves para asegurarnos que un objeto luzca como queramos.

**TEMPLATE LITERALES ES6**

Puedo llamar una función o lo que quiera dentro de {}

const salida = `

${ nombre }

${ apellido }

( ${ getEdad() } )`;

// Fernando Herrera (Edad: 33)

**FUNCIONES DE FLECHA ES6**

La funciones de flecha no modifican a lo que apunta THIS -> ES MUY IMPORTANTE.

**DESESTRUCTURACION DE OBJETOS Y ARREGLOS**

PUEDO EXTRAER LAS PROPIEDADES QUE ME INTERESAN DE UN OBJETO Y ARREGLO Y CREAR VARIABLES INMEDIATAMENTE.

**PROMESAS**

Ejecutar un código sin bloquear el código de mi aplicación.

Son propias del ES6 por lo cual para ser pasadas el ES5 requerirían de otras librerías como angular polyfills, entre otras.

Dentro de la promesa debo definir una función de flecha que recibe dos argumentos (dos funciones) -> resolver, reject.

--> El no manejar errores en las promesas puede llevar a que se detenga el programa.

--> Con el .then se inicia la ejecución de la promesa que se puede pensar como una función asincrona.

**INTERFACES TYPESCRIT**

Las uso con el fin de tipar objetos -> En js es como un tipo de dato.

No tienen equivalencia en javascript --> Si tengo mil interfaces cuando lleve a Js voy a tener cero lineas

Son reglas o contratos si quiero tipar datos.

**CLASES DE LA POO**

Es una abstracción de un objeto de la vida real.

En Js el prototype es como la herencia o el ADN del objeto

En TS con solo el constructor y ponerle public ya no necesito crear la propiedad a nivel de clase.

**IMPORTACIONES**

A la clase ponerle export class

**DECORADORES DE CLASES**

Me permite añadirle o ejecutar cosas de la clase --> Añadirle funcionalidades a la clase

**ENCADENAMIENTO OPCIONAL**

Sirve para cuando no estoy seguro si la propiedad hijos está presente o no, sino está presente sería un undefined y hacer undefined.length daría error. Por eso al poner el ? le estoy diciendo que haga la operación sólo si es seguro hacerlo, sino retorne el undefined.

*const* cuantosHijos = pasajero.hijos?.length || 0;

**ANGULAR**

ANGULAR🡪 Web, mobile web, native mobile, etc. 🡪 Intervienen factores como para la web, para mobile native script o mobile web, ionic, etc etc etc.

**ANGULAR es el framework 🡪** Marco de trabajo estandarizado. Es modular. Es mantenido por Google.

Bloques fundamentales de Angular:

1. Componentes: Bloque de código con su segmento HTML y su typescript (generalmente una clase)
2. Rutas: Mostrar diferentes componentes basados en las urls del navegador web
3. Directivas: Existen directivas de componentes, directivas estructurales y directivas de atributos.
4. Servicios: Son singletons bastante fuertes y son los lugares que centralizan la información, generalmente un componente llama a los servicios ---- (pueden evitar trabajar con redux pero no quiere decir que no sea necesario redux) 🡪
5. Módulos. Ejemplo, módulo de productos, módulo de proveedores, módulo de autenticación. Permiten agrupar todo lo que hay todo lo que se mencionó atrás (Componentes, services, rutas, etc). -> Es una funcionalidad completa.

-> En el inicio de los tiempos Angular JS 1.7

-> Desde la 2 el cambio fue mucho y sigue evolucionando hasta la 8, 9, 10

-> Versionamiento semántico 7.0.2 --> (2 parches o corrección de error, retro compatibles) (0 actualización menor, agrega funciones, retrocompatible) (7 nueva version y puede ser o no retrocompatible)

-> Cada 6 meses el equipo de Angular libera una versión mayor.

**INICIANDO**

ng new my-app

ng serve -o

**DIRECTORIOS**

Usualmente un .json es un archivo de configuración

\*tslint.json -> Linter basado para typescript -> Reglas de como codificar en ts

\*tscongif.json -> Archivo de configuración de typescript -> A cual javascript traspilar, etc.

\*tsconfig.spec.json -> .spec es relacionado a pruebas --> unitarias o de integración

\*package-lock.json -> Dice como se construyeron los módulos del proyecto

\*karma.conf.js -> Pruebas unitarias y de integración basadas en karma

\*\*tsconfig.app.json -> Enfocado propiamente en la app -> Está el output directory

**\*\*package.json -> MUY IMPORTANTE, están las dependencias del proyecto -> No debería ser modificado manualmente** -> Generalmente añadir un nuevo paquete se hace por consola

\*\*\*angular.json -> Configuraciones importante para nuestra app -> Assets, styles

\*\*\*src -> Carpeta muy importante

\*\*\*node\_modules --> módulos de node, unos irán a pdn y otros no, etc.

\*\*\*e2e -> Carpeta de end to end testing

**CONTENIDO DE EL DIRECTORIO SRC**

**1. app** -> Adentro estan los archivos del componente app de la aplicación, usualmente punto inicial de la misma

- app.component.html -> Contenido estático

- app.component.ts -> Clase de typescript

- app.component.css -> Estilos

- app.component.spec.ts -> Pruebas

**app.module.ts** -> \***IMPORTANTE**\* -> Es una clase con un decorador especial. Es la aplicación en general y los grupos que contiene la misma. Agrupador de los módulos de mi aplicación

**2. assets** -> Recursos estáticos de la app

**3. environments** -> Separar dllo de pdn por ej.

4. **index.html** -> Página principal de la app, se agregan módulos

**5 main.ts** -> Dice el target de la app, usualmente no se toca.

**6. pollyfils** -> Compatibilidad con otros navegadores web

**7. styles.css** -> Hoja de estilos global

**COMPONENTES**

Crear componente:

**nombre.component.ts**

**Ej: contador-increible.component.ts**

No basta sólo con crearlo y usar el selector -> También hay que agregarlo al app.module en declarations para que pueda ser renderizado.

**UN COMPONENTE TIENE QUE HACER PARTE DE ALGÚN MÓDULO.**

**MODULOS**

Es recomendable hacer separaciones por módulos, ejemplo crear un directorio llamado contador y generalmente dentro de este habrá un archivo que se llame contador.module.ts

\*\*\*\*Cuando se crea un componente este debe ser importado dentro de un módulo\*\*\*

**ONE WAY DATABINDING – ANGULAR HACE EL REDIBUJADO MUY EFICIENTE**

Si una propiedad cambia en el .ts se va a redibujar en el HTML --> ES EL RECOMENDABLE ( Se puede hacer two way también con input )

**GENERAR COMPONENTE DE FORMA AUTOMÁTICA**

ng g component heroes/listado

**DIRECTIVAS**

**DIRECTIVAS ESTRUCTURALES: PERMITEN CAMBIAR EL DOM**

NGFOR

<li \*ngFor="let heroe of heroes; let i = index">

        {{ i + 1 }} {{heroe}}

</li>

NGIF

 <h3 \*ngIf="heroeBorrado.length > 0">Héroe borrado</h3>

NGTEMPLATE y NGIF-ELSE

Template -🡪 Referencia local

<div \*ngIf="heroeBorrado; else noBorrado">

    <h3>Héroe borrado: <small>{{ heroeBorrado }}</small> </h3>

</div>

<ng-template #noBorrado>

    <h3> No ha borrado nada. </h3>

</ng-template>

**MODULOS**

Inicialmente tenemos un módulo principal llamado app.mdoule.ts y es cargado desde el main.ts

@NgModule({

  declarations: [

    AppComponent,

    ContadorComponent,

    HeroeComponent,

    ListadoComponent

  ],

  imports: [

    BrowserModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

**El objetivo de los módulos es ayudarnos a agrupar componentes relacionados entre si, sin necesidad de que este crezca desbordadamente.**

**Objetivos de los módulos:**

1. Encapsular las cosas
2. Ayudarnos con el lazy loading -> Carga perezosa -> No es necesario cargar los módulos que no se estén usando cuando arranque la aplicación. Que sean cargados bajo demanda.

**Creando nuevo módulo**

Carpeta héroes para que agrupe todo lo que allí va, y también crear un héroe.modoule.ts.

**declarations:** Es un arreglo, dicen que cosas contiene el módulo. Que componentes, que pipes, etc.

**exports**: Todo lo que está dentro de las declaraciones está invisible, entonces aquí debo poner lo que quiero que sea visible por fuera del módulo.

**imports**: Van módulos, usualmente sólo módulos es lo que se pone en los imports

**PARA USAR EL MÓDULO QUE CREE, ENTONCES DEBO LLEVARLO A LOS IMPORTS DEL APP.MODULE.TS**

Sirven mucho los módulos porque implica que ya cada equipo de trabajo puede enfocarse en trabajar en un módulo sin afectar a lo otro y al final lo único es importarlo en el app.module.ts

**GENERAR MODULO CON EL CLI**

***ng generate module dbz***

**Y CREAR COMPONENTE ADENTRO**

***ng generate component dbz/mainPage --skipTests***

**FORMULARIOS**

Por defecto recargan toda la página, pero si capturo el evento submit del formulario y le hago en el .ts event.preventDefault entonces ya no haría eso:

<form (submit)="agregar( $event )">

            <input type="text" placeholder="Nombre" />

            <input type="number" placeholder="Poder" />

            <button type="submit">

                Agregar

            </button>

        </form>

**.ts**

  agregar( *event*: *any* ){

    event.preventDefault();

    console.log('Hey!!!')

  }

**PERO ANGULAR AYUDA Y YA NO HAY QUE HACER ESO – FORMS MODULE se debe importar**

<form (ngSubmit)="agregar()">

**ngModel**

**Two way data binding – Emite y recibe eventos**

<input

                type ="text"

                placeholder ="Nombre"

                name="nombre"

                [(ngModel)] = "nuevo.nombre"

            />

**COMPONENTES HIJOS**

**@Input:** Se le agrega a una propiedad del componente hijo para decirle que una propiedad va a venir desde el componente padre.

@Input() personajes: Personaje[] = [];

En el html del componente padre se agrega la etiqueta del hijo y se pone entre [] el nombre de la propiedad del hijo al que se quiere mapear y entre “” la propiedad en el padre.

<app-personajes [personajes]="personajes">

        </app-personajes>

    </div>

@Output: Emitir valor del hijo al padre.

@Output() onNuevoPersonaje: EventEmitter<Personaje> = new EventEmitter();

this.onNuevoPersonaje.emit( this.nuevo );

En el padre para recibir

<app-agregar [nuevo] = "nuevo" (onNuevoPersonaje)="agregarNuevoPersonaje( $event )"></app-agregar>

**SERVICIOS**

Son muy parecidos a trabajar con un singleton y hacen que no sea tan necesario el uso de patrones como REDUX, etc.

@Injectable

Permite centralizar la información.

Inyección de dependencias

*constructor*(private *dbzService*: DbzService){

  }

Significa que es un servicio global

@Injectable({

  providedIn: 'root'

})

**DESPLIEGUES RÁPIDOS A PDN**

Build de producción crea una carpeta que se llama **dist** que es donde están los archivos js de la aplicación.

**ng build –prod**

**@ViewChild**

Relaciona la variable con un elemento del HTML.

@ViewChild('txtBuscar') txtBuscar!: ElementRef<HTMLInputElement>;

  buscar() {

*const* valor = this.txtBuscar.nativeElement.value;

    console.log(valor);

    this.txtBuscar.nativeElement.value = '';

**LOCALSTORAGE**

Persiste la info luego de cerrar el navegador –No guardar información sensible, solo cosas que el usuario pueda ver y manipular.