Manual de Buenas Prácticas al hacer aplicaciones en Angular

1) Uso de const, let y var

Al declarar una variable en JS tenemos tres opciones: const, var y let

Cada una de estas tiene un significado y un uso específico, y el correcto manejo de las mismas permiten una lectura del código más clara, evitan errores y permiten generar un código más robusto.
Antes de poder detallar las diferencias entre las mismas hace falta denotar que en JS existen varios scope(digamos el alcance de un bloque de código), esto quiere decir que si declaro una variable por ejemplo en un scope global, esta variable va a ser conocida por todo el script, si fuera un scope de función solo será conocida dentro de la función, etc.

En criollo:

-**const:** Este es un valor constante es decir **NO** puede cambiar, una vez asignado la variable no puede ser modificado.

-**var:** Esta forma de declarar una variable es la que menos se debe utilizar, declara una variable dinámica a diferencia de const, sin embargo lo hace en un scope de función, esto quiere decir que si declaro una variable al comienzo de una función va a existir dentro de toda la ejecución de la misma y dentro de las funciones que están dentro de la misma.

-**let:** A diferencia de var, let tiene scope de bloque, es decir solo existe dentro de su bloque de código, es decir todo lo que este dentro de las mismas llaves ({})

Un ejemplo claro se ve en la declaración de una variable en un for:

```
var i
for( i = 0; i < n; i++){}

let i
for( i = 0; i < n; i++){}
```

En el primer caso el for correra normalmente, sin embargo en el segundo el compilador devolverá un error: “*i no esta indefinida”*.
cabe aclarar que si se intenta usar una variable i en el primer caso mas adelante en el código el último valor del for estara habilitado.

2) **Uso pipes de RxJs**

El uso de RxJs es algo necesario al trabajar con observables, en vez de tener que importar cada operador de la libreria que queremos para trabajar se puede importar **pipe** este operador permite usar operadores de RxJs como map en secuencia, reduciendo los operadores a ejecutar a solo los necesarios, además de permitir identificar casos de tener operadores "*sueltos*". En caso que la ejecución corte a la mitad de la función solo se habrán importado y utilizado los operadores RxJs que se solicitaron entre medio, no todos los que estén en el código. El caso más claro se ve en el uso de un catch, el catch de rxjs solo se utilizará en caso de error sino, el compilador lo verá como si no estuviera ahí.
EJEMPLO:

```
import { map, take } from 'rxjs/operators';

iAmAnObservable
    .pipe(
      map(value => value.item),
      take(1)
     );
```

3) **Utilizar servicios**

En angular los injectables son elementos que se ejecutan de forma volátil, es decir existen durante la vida del componente que los llamo, por lo tanto las llamadas a APIs internas o externas a nuestro backend deberían realizarse por este medio, no debería haber llamadas HTTP dentro de ningún componente.(debido a la persistencia de los mismos)
De la misma manera para comunicarse entre componentes o incluso para reutilizar código los componentes deben poder comunicarse con servicios propios o con los servicios del padre.

En caso de tener un módulo con varios componentes hermanos, supongamos en el módulo 1 se encuentran los componentes Ca y Cb, cada uno con sus respectivos servicios Sa y Sb, en caso de que Ca necesite de utilizar un bloque de código de Sb, sería prudente que llame al servicio del padre S1 o en su defecto que realize su propia implementación en Sa.

Lo que nos lleva a el número 5.

4) **Parsear las respuestas de las API en los servicios**

Dado que estamos trabajando en Frontend, debemos tener en cuenta que no siempre las respuestas de una API van a estar **bien** diseñadas sino que estas pueden estar mal formadas o incluso pueden no ser específicas para la funcionalidad que queremos usar.

Supongamos un API que devuelve todas las personas de una organización para cargar un combo box, en el caso que este esté mal formado(en vez de ser un array de objetos es un objeto con un array de objetos dentro por ejemplo), o que no nos interesan todos los datos de las personas para el combo, suponiendo que solo necesitamos el id y el nombre, sería prudente solo pasarle dichos datos a el componente y no todos los datos para que cada componente deba parsearlos a su manera, así reutilizamos código y los servicios no proveen solo servicios sino que proveen **buenos** servicios.

5) **Subscribirse en las vistas, no en los componentes**

Los componentes pueden perdurar incluso una vez terminado su uso, las vistas no, un componente no se destruye a menos que asi se le indique o que se refresque el browser, por ende siempre es una buena idea subscribirse en el HTML mediante el pipe **async**, evitando así manejar la desubscripción del mismo.
EJEMPLO:

```
// template

<p>{{ textToDisplay$ | async }}</p>

// component

this.textToDisplay$ = iAmAnObservable
    .pipe(
      map(value => value.item)
    );
```

6) **Cerrar las subscripciones**

Así como se mencionó en el caso 6, al subscribirse a un observable es necesario desubscribirse correctamente, para no perder memoria; mantener las subscripciones abiertas consume memoria y esto puede resultar en la aplicación ralentizandose o incluso el cierre inesperado de la misma.
Para ello RxJs provee operadores especificos, como take o takeUntil. Siempre es una buena idea desubscribirse una vez terminado el uso de un observable o incluso en la muerte del componente(ngOnDestroy).

EJEMPLO:

```
private _destroyed$ = new Subject();

public ngOnInit (): void {
    iAmAnObservable
    .pipe(
       map(value => value.item)
      // We want to listen to iAmAnObservable until the component is destroyed,
       takeUntil(this._destroyed$)
     )
    .subscribe(item => this.textToDisplay = item);
}

public ngOnDestroy (): void {
    this._destroyed$.next();
    this._destroyed$.complete();
}
```

7) **Usar el operador apropiado para cada situación**

Así como la declaración de variables, cada operador tiene una única función o al menos una en la que destaque; esto permite que a simple vista uno pueda interpretar que está realizando el código.

Supongamos el caso de usar un for o un map para iterar sobre un array, si bien los dos funcionan de manera similar, el map al ser un operador específico de colecciones, no hace falta saber que se está haciendo para saber que se está iterando una colección a diferencia del for. Al igual que este caso existen casos mucho más específicos, como el switchMap o el exhaustMap. Si bien puede ser engorroso aprender tantos operadores y todo se puede resolver con un for o un map y algunos arreglos, la especialización y el uso del operador correcto permite mayor lectura y menos comportamientos no esperados.

8) **¿Lazy load, Eager load o ambos?**

A la hora de cargar una aplicación debemos ser cautelosos, cargar todos los módulos de una aplicación muy grande podría provocar un gasto innecesario de memoria y puede resultar en una aplicación lenta, sobre todo en el arranque, mientras que la implementación de lazy load o carga perezosa permite cargar los módulos a necesidad, “en caliente” si se puede decir.

Entonces: ¿Siempre se debería usar Lazy Load en vez de Eager load?

En criollo: No.

El uso de lazy load puede resultar en una mala experiencia de usuario, ya que cada módulo debe buscarlo del servidor a disposición, pero funciona bien para módulos pequeños o unos que no se utilizen en todos los casos, más que nada en usuarios con diferentes perfiles.

lo que nos lleva al siguiente punto.

9) **Utilizar estrategias de precarga**

Como se dijo en el punto anterior no es factible elegir una forma única para cargar los módulos de una aplicación en angular sino que debería analizarse según cada proyecto.

Para esto angular nos provee de una prediseñada llamada PreloadAllModules, que permite realizar una carga perezosa de todos los módulos a medida que van terminando de cargar, una vez cargado el padre carga a los hijos. Pero a veces no nos alcanza con esta estrategia, entonces debemos crear una estrategia propia de precarga, parte perezosa, parte ansiosa.
Ejemplo:
App.route:

```
@NgModule({
  imports: [RouterModule.forRoot(ROUTES, { useHash: true,
  preloadingStrategy: EstrategiaDeCargaPersonalizada })],
  exports: [RouterModule]
})
export class AppRoutingModule {
}
```

Estrategia:

```
@Injectable({
  providedIn: 'root'
})
export class EstrategiaDeCargaPersonalizada implements PreloadingStrategy {

  constructor() { }

  preload(route: Route, load: Function): Observable<any> {
    return route.data && route.data.preload ? load(): of(null);
  }

}
```

Hijos.route:

```
const routes: Routes = [
  {
    path: '',
    component: PagesComponent,
    children: [
      { path: 'inicio', loadChildren: './inicio/inicio.module#InicioModule', data: { preload: true }},
      { path: '', redirectTo:'inicio', pathMatch:'full' },
      { path: '**', redirectTo:'not-found', pathMatch:'full'}
    ]
  }
];
```

10) Evitar suscripciones anidadas

El uso de los operadores de cadena de RxJs como combineLatest o withLatestFrom, seran nuestros aliados a la hora de encadenar subscripciones, y así evitar anidarlas. Esto permite una mejor lectura del código y un uso correcto de RxJs ademas de trabajar con el asincronismo con mayor facilidad evitando cadenas de *“async y await”*

EJEMPLO:

```
firstObservable$.pipe(
    withLatestFrom(secondObservable$),
    first()
)
.subscribe(([firstValue, secondValue]) => {
    console.log(`Combined values are: ${firstValue} & ${secondValue}`);
});
```

11) Tipificar los datos

Si bien TS es un lenguaje de tipado dinámico, es decir permite la creación de variables de tipo *any* que pueden ser cualquier tipo de dato, es una buena práctica tipificar correctamente cada dato. En una variable que es un número declararla como un número, una que es una cadena de caracteres declararla como un string y una que es un array de estudiantes declararlos como un array de Estudiantes; esto permite mejor visualización del código, es más robusto y legible, y también elimina problemas inesperados.

En este ejemplo se puede ver fácilmente al intentar operar dos números.

```
let a = "8"
let b = 1000
let c = b*a
console.log(c);

let a = 8
let b = 1000
let c = b*a
console.log(c);
```

En el caso de b*a el resultado correcto debería ser (8)*(1000) = 8000.
sin embargo el primer bloque de código va a arrojar el siguiente resultado:
(“8”)*(1000) = 88888888888888888888…… (mil ochos)

En el caso de haber tenido tipificadas las variables, esto no habria sido posible, sino que saldrá un error de compilación: *tipo string no es asignable a una variable del tipo integer*

Este principio se puede aplicar a variables, argumentos de función, retornos, etc.
Básicamente tipificar **TODO** lo tipificable.

12) Usar reglas Lint

Lint son un conjunto de reglas que aplican a la aplicación en general, por ejemplo el uso de lint no-console, no permite que en la consola del browser se imprima nada, errores o incluso logs del programador a la mitad del debugueo de un componente.
Para ver más sobre Lint: https://palantir.github.io/tslint/rules/

13) Los componentes deben ser atómicos

El uso de componentes grandes conlleva a errores, y mientras mayor tamaño tenga este y la aplicación más funcionalidades van a estar afectadas.
La práctica ideal consiste en reducir todos los componentes a su mínima expresión y así reducir código y reutilizar componentes. Mientras más atómicos menos posibilidades habrá de que fallen, levantando menos errores en su camino.
Un componente para una única funcionalidad.

14) Los componentes deben lidiar solo con lógicas de visualización

Los componentes son los elementos con los que el usuario va a interactuar, estos no deben calcular precios ni promediar variables, para eso estan las demas capas de la aplicación, el componente debe ser provisto de datos para visualizar, no entender lógicas de negocio, así como encargarse de mostrar esos datos.

15) Usar el cache a tu favor

El cache del browser al hacer aplicaciones web puede llegar a ser un dolor de cabeza, sobre todo en los desarrollos iniciales de la aplicación. Sin embargo el uso del mismo permite una mejora importante de performance de la aplicación. Hay que saber decidir que es necesario que maneje cache y que no… **NO** hay que dejar que el browser decida por nosotros, esto termina implicando en cambios que no se visualizan a menos que se borre el mismo.

El buen uso del cache puede permitir ahorrar llamadas al backend, y reducir tiempos de espera por tener una mala conexión.

16) Versionar la aplicación

Una vez manejemos el cache de la app cada release que se haga un versionado de la misma.
Esto incluye un bonito numerito que indica en qué versión estamos, y qué cambios hubo en la última versión. Esta es una buena forma de rastrear de donde vino un bug que salió hoy de un cambio de hace 4 semanas.
A su vez mediante el versionado le podemos indicar al cliente que debe actualizar su cache, si se encuentra en una versión anterior, permitiendo que este esté siempre actualizado con la última versión.

17) No utilizar lógica en la vista de un componente

Con angular es posible utilizar lógica directamente en el HTML, sin embargo el uso de la misma conlleva a una peor escalabilidad y no puede realizarse unitesting sobre ella.
En cada *ngIf deberia de haber una función, no deberia utilizarse la lógica de la misma en el mismo.
EJEMPLO:
No:

```
<app-navbar></app-navbar>
  <div *ngIf="soyUnaVariable == 1" class="container-fluid
     <app-sidebar class="d-none d-sm-block"></app-sidebar>

       <div class="main-panel">
          <div class="content-wrapper">
             <router-outlet></router-outlet>
          </div>
       </div>


   </div>
```

Si:

```
<app-navbar></app-navbar>
  <div *ngIf="retornoUnBooleano()" class="container-fluid
     <app-sidebar class="d-none d-sm-block"></app-sidebar>

       <div class="main-panel">
          <div class="content-wrapper">
             <router-outlet></router-outlet>
          </div>
       </div>


   </div>
```

En el caso que no se desee usar una función puede usar una variable manejada en el componente, pero esta debe ser un booleano, y no se deben usar comparaciones en la misma.

18) Utilizar strings “seguras”

En caso de tener una string que solo puede tener valores fijos, supongamos tipos de perfiles, en vez de declararla como tipo string se la puede declarar como los valores posibles que puede llegar a utilizar, reduciendo la posibilidad de tener bugs en la aplicación.
EJEMPLO:

```
private myStringValue: 'First' | 'Second';

if (itShouldHaveFirstValue) {
   myStringValue = 'First';
} else {
   myStringValue = 'Other'
}

// This will give the below error
Type '"Other"' is not assignable to type '"First" | "Second"'
(property) AppComponent.myValue: "First" | "Second"
```

19) Alias de importación

Hay algunos elementos de angular que van a ser utilizados por muchos lugares de la aplicación por ejemplo un servicio.
En caso que se quiera realizar esto, a veces resulta engorroso tipear toda la ruta hacia dicho elemento e incluso si realizamos un movimiento de un elemento de una carpeta a otra se desrreferenciará de todos lados, para ello angular provee de la opción de guardar rutas absolutas en su configuración utilizando un alias, seguido de un @.
Suponiendo que queremos usar una variable de entorno podemos configurar un alias para eso escribiendo en el **tsconfig.json** bajo paths.

```
"paths": {
  "@angular/*": ["node_modules/@angular/*"],
  "@enviroment/*":["/soyUnaRuta/*"]

},
```

permitiendo asi, importar los elementos mediante la linea:

import{ nombre1, nombre2 } from ‘@enviroment’

20) Comentarios y nombres en las variables

Al igual que en cualquier lenguaje de programación TS y JS proveen herramientas para hacer comentarios, si bien esto no es puntual de estos lenguajes o del framework siempre es bueno recordar que el que lee nuestro código puede o no saber que estamos intentando hacer, y una ayuda de nuestra parte, mediante comentarios o variables claras puede facilitarle la interpretación del mismo.
La nomenclación de una variable debería cumplir con el objetivo de mostrar de forma lo más clara posible su contenido y/o su tipo de datos sin necesidad de entender su contexto.
Ejemplo: indP => indicadorProductividad(esto puede ser un numero o un booleano)
personaL => personasList(indica una lista de personas)