BEST PRACTICES - UI

Guide to general UI Best Practices

This document provides the best practices that every UI developer should follow during development. Following these results in clean, error free, easy to maintain deliverables.

Contents

[1. Use Correct Data Type 3](#_Toc82610046)

[2. Use ‘let’ instead of ‘var’ 3](#_Toc82610047)

[3. Use ‘const’ for constants 3](#_Toc82610048)

[4. Use access modifiers for class members 3](#_Toc82610049)

[5. Avoid unnecessary comments 4](#_Toc82610050)

[6. Use a Linter code / code beautifier 4](#_Toc82610051)

[7. Use trackBy along with ngFor 4](#_Toc82610052)

[8. Use ES6 features 5](#_Toc82610053)

[9. Break down Larger complex components into small reusable components 5](#_Toc82610054)

[10. Use Lazy Loading when developing new features 6](#_Toc82610055)

[11. Use async pipe in templates 6](#_Toc82610056)

[12. Use state management 6](#_Toc82610057)

[13. Naming Styles 6](#_Toc82610058)

[14. Use strict equality comparison operator 7](#_Toc82610059)

[15. Use switchcase for nested if case 8](#_Toc82610060)

[16. Use ternary operator wherever possible 8](#_Toc82610061)

[17. Use RxJS operators 8](#_Toc82610062)

[18. Don't use resolver to avoid unwanted loading time and white screen 8](#_Toc82610063)

[19. Don’t implement business logics inside components 8](#_Toc82610064)

[20. Move larger, complex operations to utilities 8](#_Toc82610065)

[21. Avoid possible memory leakages 8](#_Toc82610066)

[22. Don’t add services to common shared modules 8](#_Toc82610067)

[23. Use existing components 9](#_Toc82610068)

[24. Use simplified Truthy check 9](#_Toc82610069)

[25. Don’t use jQuery 9](#_Toc82610070)

### Use Correct Data Type

When initializing a variable use the exact data type. Don’t use “any”

Example:

public name :string;

public age : number;

public isVaccinated : boolean

### Use ‘let’ instead of ‘var’

When declaring variables use **let** keyword instead of **var**

Example:

function foo(){

~~var i=0;~~

    let j=0;

}

### Use ‘const’ for constants

Use **const** keyword for variable that are constant and cannot be reassigned.

Example:

const PI = 3.141592653589793;

### Use access modifiers for class members

Use required access modifiers for the classes as per requirement

* private: can be accessed only within the class.
* protected: can be accessed only within the class and by the instance of its child class.
* public: accessible everywhere without restrictions

Example:

public name :string;

private age : number;

protected isVaccinated : boolean

### Avoid unnecessary comments

Comment only when needed, do not comment unwanted code instead remove them.

Example:

*// This comment is invalid*

*// I'll remove this code later, keep it for reference*

*// x= x[y].abc;*

*/\**

*\* This is a valid comment*

*\* Created: 09/15/2021*

*\* This class is created by Balamurugan for Pet Service.*

*\* Do not edit the class unless required.*

*\*/*

export class PetService {

*/\*\*\*\*\*\*/*

}

### Use a Linter code / code beautifier

Use any code linter or beautifier to clean up extra spaces and make code look prettier and find unused variables or imports.

Example:

Before Linting

if ('this\_is'==/an\_example/){of\_beautifier();}else{var a=b?(c%d):e[f];}

After Linting

if ('this\_is' == /an\_example/) {

    of\_beautifier();

} else {

    var a = b ? (c % d) : e[f];

}

### Use trackBy along with ngFor

When using ngFor use trackBy function to avoid any unwanted DOM re-rendering.

Example:

@Component({

    selector:'my-app',

    template:`<li \*ngFor="let item of list; trackBy:identify">{{item.name}}</li>`

  })

  export class App {

    list:[];

      identify(*index*,*item*){

*//do what ever logic you need to come up with the unique identifier of*

*your item in loop, for example I've just returned the object name.*

        return *item*.name;

       }

    constructor() {

      this.list = [{name: 'Bruce'}, {name: 'Wayne'}]

*// simulating request*

      setInterval( () => {

        this.list = [{name: 'Bruce'}, {name: 'Wayne'}];

      }, 2000);

    }

  }

### Use ES6 features

Use ES6 features like arrow functions, template literals spread operator etc whenever possible as they are shorter and cleaner.

Example:

*//spread operator*

var arr = [1,2,3];

var arr2 = [...arr];

*//template literals*

let myName = 'John';

let myRole = 'Software Developer';

console.log(`My name is ${myName} and I am a ${myRole}.`);

*//arrow functions*

let getName = ((*x*,*y*)=> console.log(*x*+*y*));

### Break down Larger complex components into small reusable components

When you encounter any larger complex components try to find out small reusable functional units which can be turned to new components.

### Use Lazy Loading when developing new features

When you are developing a new feature and it is not needed during app initialization, use lazy loading to load them to DOM on demand.

Example:

const routes: Routes = [

    {

        path: 'users',

        loadChildren: ()=>import('./users/users.module').then(*m* => *m*.UsersModule)

    }

];

### Use async pipe in templates

When displaying values from observables or promise in templates use async pipe. It subscribes to an observable or promise and returns the latest value it has emitted.

Example:

<div><code>observable|async</code>: Time: {{ time | async }}</div>

### Use state management

Avoid using local and session storages for data transfer instead use state management and maintain single source of truth for the data throughout your application.

### Naming Styles

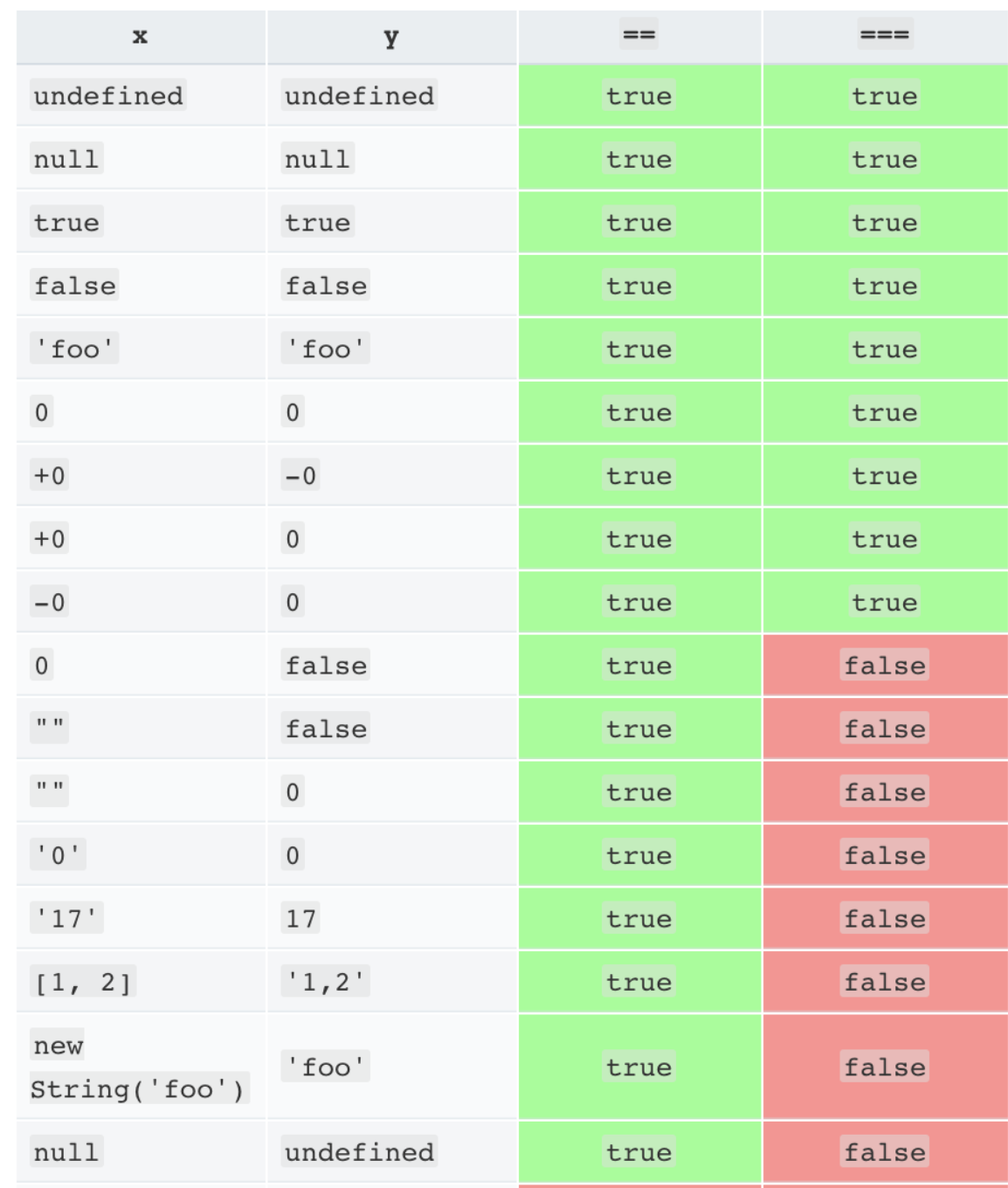
Follow below naming standards throughout the application.

1. Kebab case (kebab-case) =>ex: customer-list.service.ts.
   1. The kebab-case is used in naming folders, component selectors, files, and the Angular application itself. Typical files in an Angular project include component files, service files, template files, module files, etc.
2. Pascal case (PascalCase) =>ex: Export class CustomerList
   1. The Pascal case is mainly used for naming classes in an Angular Project.
3. Camel case (camelCase) =>public getCustomerList();
   1. The camelCase is used for naming methods or function, properties, fields, directive selectors, and pipe selectors as highlighted below.

### Use strict equality comparison operator

When comparing use === instead of == to avoid type coercion.

Example:

Refer below image for possible type coercions

### Use switchcase for nested if case

If you encounter complex and nested if else conditions use switchCase instead.

### Use ternary operator wherever possible

For small assignments with if else conditions use ternary operators.

Example:

let person = isVaccinated? 'get a reward' : 'get a penality';

### Use RxJS operators

Use RxJS operators wherever possible for better performance and error handling.

### Don't use resolver to avoid unwanted loading time and white screen

Avoid using resolver when routing as it creates significant lag in navigation.

### Don’t implement business logics inside components

Move business logics to a services or common service files for code maintainability and re-usability.

### Move larger, complex operations to utilities

Move or implement any large/complex arithmetic, logical or business operations to utility file

Example: customer-utils.ts

### Avoid possible memory leakages

Use takeUntil or unsubscribe any observables to avoid memory leakages.

### Don’t add services to common shared modules

Avoid adding services to shared modules because it will create circular dependency.

### Use existing components

Use shared components like popup, slider, alert, fpolayout which are already available in our application.

### Use simplified Truthy check

Don’t use multiple conditions to check null, undefined or empty.

Example:

if(a){

    ...

    ...

    ...

}

The above will evaluate following conditions

* null
* undefined
* NaN
* empty string ("")
* 0
* false

### Don’t use jQuery

Avoid using jQuery in the application.