ANGULAR CODING STANDARDS

BY

ADEEL FAKHAR – SENIOR MANAGER IT

Check	Severity	1st Level	2nd Level
Coding Conventions & Guidelines	**Low**		

Classes

● Do use upper camel case when naming classes. Classes can be instantiated and construct an instance. By convention, upper camel case indicates a constructible asset.

Constants

Do declare variables with const if their values should not change during the application lifetime.

● Consider spelling const variables in lowercamelcase. The tradition of naming constants in UPPER_SNAKE_CASE reflects an era before the modern IDEs that quickly reveal the const declaration. TypeScript prevents accidental reassignment.

● Do tolerate existing const variables that are spelled in UPPER_SNAKE_CASE. The tradition of UPPER_SNAKE_CASE remains popular and pervasive, especially in third party modules

Interfaces

● Do name an interface using upper camel case.

Properties and methods

● Do use lower camel case to name properties and methods.

● Avoid prefixing private properties and methods with an underscore.

Import line spacing

● Consider leaving one empty line between third party imports and application imports.

● Consider listing import lines alphabetized by the module.

● Consider listing destructured imported symbols alphabetically.

Components

● Do use dashed-case or kebab-case for naming the element selectors of components because it will Keeps the element names consistent with the specification for Custom Elements.

● Do give components an element selector, as opposed to attribute or class selectors because components have templates containing HTML and optional Angular template syntax. They display content. Developers place components on the page as they would native HTML elements and web components. It is easier to recognize that a symbol is a component by looking at the template's html.

● Do extract templates and styles into a separate file, when more than 3 lines.

● Do name the template file [component-name].component.html, where [component-name] is the component name.

● Do name the style file [component-name].component.css, where [component-name] is the component name.

● Do specify component-relative URLs, prefixed with ./. Large, inline templates and styles obscure the component's purpose and implementation, reducing readability and maintainability. In most editors, syntax hints and code snippets aren't available when developing inline templates and styles. The Angular TypeScript Language Service (forthcoming) promises to overcome this deficiency for HTML templates in those editors that support it; it won't help with CSS styles. The ./ prefix is standard syntax for relative URLs; don't depend on Angular's current ability to do without that prefix.

● Do use the @Input() and @Output() class decorators instead of the inputs and outputs properties of the@Directive and @Component metadata.

● Consider placing @Input() or @Output() on the same line as the property it decorates because It is easier and more readable to identify which properties in a class are inputs or outputs. If you ever need to rename the property or event name associated with @Input or @Output, you can modify it in a single place.

● The metadata declaration attached to the directive is shorter and thus more readable.

● Placing the decorator on the same line usually makes for shorter code and still easily identifies the property as an input or output. Put it on the line above when doing so is clearly more readable.

● Avoid input and output aliases except when it serves an important purpose because Two names for the same property (one private, one public) is inherently confusing.

● You should use an alias when the directive name is also an input property, and the directive name doesn't describe the property.

Identify

● Do name the file such that you instantly know what it contains and represents.

● Do be descriptive with file names and keep the contents of the file to exactly one component.

● Avoid files with multiple components, multiple services, or a mixture.

> **Why?**
>
> Spend less time hunting and pecking for code, and become more efficient. Longer file names are far better than short-but-obscure abbreviated names

Flat

● Do keep a flat folder structure as long as possible.

● Consider creating sub-folders when a folder reaches seven or more files.

● Consider configuring the IDE to hide distracting, irrelevant files such as generated .js and .js.map files.

> **Why?**
>
> No one wants to search for a file through seven levels of folders. A flat structure is easy to scan.

Base your decision on your comfort level. Use a flatter structure until there is an obvious value to creating a new folder.

trackBy

● When using ngFor to loop over an array in templates, use it with a trackBy function which will return an unique identifier for each item.

> **Why?**
>
> When an array changes, Angular re-renders the whole DOM tree. But if you use trackBy, Angular will know which element has changed and will only make DOM changes for that particular element.

Before:

```
<li *ngFor="let item of items;">{{ item }}</li>
```

After:

```
// in the template
```

```
<li *ngFor="let item of items; trackBy: trackByFn">{{ item }}</li>

// in the component

trackByFn(index, item) {
  return item.id; // unique id corresponding to the item
}
```

Services

● Do provide services to the Angular injector at the top-most component where they will be shared. When providing the service to a top level component, that instance is shared and available to all child components of that top level component. This is ideal when a service is sharing methods or state and this is not ideal when two different components need different instances of a service. In this scenario it would be better to provide the service at the component level that needs the new and separate instance.

● Do use the `@Injectable()` class decorator instead of the @Inject parameter decorator when using types as tokens for the dependencies of a service. The Angular Dependency Injection (DI) mechanism resolves a service's own dependencies based on the declared types of that service's constructor parameters. When a service accepts only dependencies associated with type tokens, the @Injectable() syntax is much less verbose compared to using @Inject() on each individual constructor parameter.

***T-DRY* (Try to be *DRY*)**

● Do be DRY (Don't Repeat Yourself).

● Avoid being so DRY that you sacrifice readability.

● Why? Being DRY is important, but not crucial if it sacrifices the other elements of LIFT. That's why it's called *T-DRY*.

For example, it's redundant to name a template hero-view.component.html because with the .html extension, it is obviously a view. But if something is not obvious or departs from a convention, then spell it out.

Lifecycle hooks

● Do implement the lifecycle hook interface because lifecycle interfaces prescribe typed method signatures. use those signatures to flag spelling and syntax mistakes.

Check	Severity	1st Level	2nd Level
Project Structure Guidelines	**Low**		

Evolve Angular apps in a modular style using core, shared and feature modules

```
|-- app
   |-- modules
     |-- home
         |-- [+] components
         |-- [+] pages
         |-- home-routing.module.ts
         |-- home.module.ts
   |-- core
     |-- [+] authentication
     |-- [+] footer
     |-- [+] guards
     |-- [+] http
     |-- [+] interceptors
     |-- [+] mocks
     |-- [+] services
     |-- [+] header
     |-- core.module.ts
     |-- ensureModuleLoadedOnceGuard.ts
     |-- logger.service.ts
   |
   |-- shared
        |-- [+] components
        |-- [+] directives
        |-- [+] pipes
        |-- [+] models
   |
|-- assets
   |-- scss
        |-- [+] partials
        |-- _base.scss
        |-- styles.scss
```

Note! The [+] means that the folder has extra files.

CoreModule

● Create a CoreModule with providers for the singleton services you load when the application starts.

● Import CoreModule in the root AppModule only. Never import CoreModule in any other module.

● Consider making CoreModule a pure services module with no declarations.

```
|-- core
      |-- [+] authentication
      |-- [+] footer
      |-- [+] guards
      |-- [+] http
      |-- [+] interceptors
      |-- [+] mocks
      |-- [+] services
      |-- [+] header
      |-- core.module.ts
      |-- ensureModuleLoadedOnceGuard.ts
      |-- logger.service.ts
```

Shared Module

● Build a SharedModule with the components, directives, and pipes that you use throughout your app. This module should consist completely of declarations, most of them exported.

● The SharedModule may re-export other widget modules, such as CommonModule, FormsModule, and modules with the UI controls that you use most widely.

● The SharedModule should not have providers for reasons explained previously. Nor should any of its imported or re-exported modules have providers. If you deviate from this guideline, know what you're doing and why.

● Import the SharedModule in your feature modules, both those loaded when the app starts and those you lazy load later.

```
|-- shared
    |-- [+] components
    |-- [+] directives
    |-- [+] pipes
```

Feature Module

● To create multiple feature modules for every independent feature of our application. Feature modules should only import services from CoreModule. If feature module A needs to import service from feature module B consider moving that service into core.To attempt to build features which don't depend on any other features just on services provided by CoreModule and components provided by SharedModule

FEATURE MODULE	CAN BE SOME IMPORTED BY	EXAMPLES
Domain	Feature, AppModule	ContactModule (before routing)
Routed	Nobody	ContactModule, DashboardModule,
Routing	Feature (for routing)	AppRoutingModule, ContactRoutingModule
Service	AppModule	HttpModule, CoreModule
Widget	Feature	CommonModule, SharedModule

Lazy loading a feature module

The feature module should be placed synchronously when the app startup to show initial content. Each other feature module should be loaded lazily after user-triggered navigation. That way we will be able to make our Angular app faster. In other words, a feature module won't be loaded initially, but when you decide to initiate it. Therefore, making an initial load of the Angular app faster too

Here is an example on how to initiate a lazy loaded feature module via app-routing.module.ts file.

```
const routes: Routes = [

{
path: 'dashboard', loadChildren: 'app/dashboard/dashboard.module#DashboardModule', component: CoreComponent

}

];
```

Aliases for imports

Aliasing our app and environments folders will enable us to implement clean imports which will be consistent throughout our application. Consider hypothetical, but the usual situation. We are working on

a component which is located three folders deep in a feature A and we want to import service from the core which is located two folders deep. This would lead to import statement looking something like

```
import { SomeService } from '../../../core/subpackage1/subpackage2/some.service'.
```

And what is even worse, any time we want to change the location of any of those two files our import statement will break. Compare that to much shorter `import { SomeService } from "@app/core"`

Using Sass

Sass is a styles preprocessor which brings support for fancy things like variables (even though css will get variables soon too), functions, mixins etc The global styles for the project are placed in a scss folder under assets.

```
|-- scss
    |-- partials
        |-- _layout.vars.scss
        |-- _responsive.partial.scss
    |-- _base.scss
|-- styles.scss
```

The scss folder does only contain one folder — partials. Partial-files can be imported by other scss files. In my case, styles.scss imports all the partials to apply their styling.

Use of smart vs. dummy components

Most typical use case of developing Angular's components is a division of smart and dummy components. The estimate of a dummy component as a component used for presentation purposes only, meaning that the component doesn't know where the data came from. For that purpose, we can use one or more smart components that will inherit the dummy's component presentation logic.

Check	Severity	1st Level	2nd Level
Typescript Guidelines	**Low**		

Names

- Use PascalCase for type names.
- Do not use "I" as a prefix for interface names.
- Use PascalCase for enum values.
- Use camelCase for function names.

Components

- 1 file per logical component (e.g. parser, scanner, emitter, checker).
- files with ".generated.*" suffix are auto-generated, do not hand-edit them.

Component should only deal with display logic

Avoid having any logic other than display logic in your component whenever you can and make the component deal only with the display logic.

Why?

Components are designed for presentational purposes and control what the view should do. Any business logic should be extracted into its own methods/services where appropriate, separating business logic from view logic.

Business logic is usually easier to unit test when extracted out to a service, and can be reused by any other components that need the same business logic applied.

Types

- Do not export types/functions unless you need to share it across multiple components.
- Do not introduce new types/values to the global namespace.
- Shared types should be defined in 'types.ts'.
- Within a file, type definitions should come first

Null and undefined

- Use undefined. Do not use null.

General Assumptions

● Consider objects like Nodes, Symbols, etc. as immutable outside the component that created them. Do not change them.
● Consider arrays as immutable by default after creation.

Classes

● For consistency, do not use classes in the core compiler pipeline. Use function closures instead

Flags

More than 2 related Boolean properties on a type should be turned into a flag.

Comments

● Use JSDoc style comments for functions, interfaces, enums, and classes. 2.9 Strings
● Use double quotes for strings.
● All strings visible to the user need to be localized (make an entry in diagnosticMessages.json).

Bootstrapping

● Do put bootstrapping and platform logic for the app in a file named main.ts.

● Do include error handling in the bootstrapping logic.

● Avoid putting app logic in main.ts. Instead, consider placing it in a component or service.

● Follows a consistent convention for the startup logic of an app.

● Follows a familiar convention from other technology platforms.

Small Functions

Define small functions, no more than 75 LOC (less is better).

Small functions are **easier to test**, especially when they do one thing and serve one purpose.

- Small functions promote **reuse**.
- Small functions are **easier to read**.
- Small functions are **easier to maintain**.

- Small functions help **avoid hidden bugs** that come with large functions that share variables with external scope, create unwanted closures, or unwanted coupling with dependencies.

const vs let

● When declaring variables, use const when the value is not going to be reassigned.

Why?

Using let and const where appropriate makes the intention of the declarations clearer. It will also help in identifying issues when a value is reassigned to a constant accidentally by throwing a compile time error. It also helps improve the readability of the code.

Before:

```
let car = 'ludicrous car';

let myCar = `My ${car}`;
let yourCar = `Your ${car};

if (iHaveMoreThanOneCar) {
  myCar = `${myCar}s`;
}

if (youHaveMoreThanOneCar) {
  yourCar = `${youCar}s`;
}
```

After:

```
// the value of car is not reassigned, so we can make it a const
const car = 'ludicrous car';

let myCar = `My ${car}`;
let yourCar = `Your ${car};

if (iHaveMoreThanOneCar) {
  myCar = `${myCar}s`;
}
```

```
if (youHaveMoreThanOneCar) {
  yourCar = `${youCar}s`;
}
```

Avoid any; type everything;

Always declare variables or constants with a type other than any.
Why?
When declaring variables or constants in Typescript without a typing, the typing of the variable/constant will be deduced by the value that gets assigned to it. This will cause unintended problems. One classic example is:

```
const x = 1;
const y = 'a';
const z = x + y;

console.log(`Value of z is: ${z}`

// Output
Value of z is 1a
```

This can cause unwanted problems when you expect y to be a number too. These problems can be avoided by typing the variables appropriately.

```
const x: number = 1;
const y: number = 'a';
const z: number = x + y;

// This will give a compile error saying:

Type '"a"' is not assignable to type 'number'.

const y:number
```

This way, we can avoid bugs caused by missing types.

Single Responsibility Principle

Component should use SRS, which is

Define 1 component per file, recommended being less than 400 lines of code.

One component per file promotes **easier unit testing and mocking**.

o One component per file makes it far **easier to read, maintain, and avoid collisions** with teams in source control.

One component per file **avoids hidden bugs** that often arise when combining components in a file where they may share variables, create unwanted closures, or unwanted coupling with dependencies.

Small reusable components

Extract the pieces that can be reused in a component and make it a new one. Make the component as dumb as possible, as this will make it work in more scenarios. Making a component dumb means that the component does not have any special logic in it and operates purely based on the inputs and outputs provided to it.

As a general rule, the last child in the component tree will be the dumbest of all.

Why?

Reusable components reduce duplication of code therefore making it easier to maintain and make changes.

Dumb components are simpler, so they are less likely to have bugs. Dumb components make you think harder about the public component API, and help sniff out mixed concerns.

Add caching mechanisms

● When making API calls, responses from some of them do not change often. In those cases, you can add a caching mechanism and store the value from the API. When another request to the same API is made, check if there is a value for it in the cache and if so, use it. Otherwise, make the API call and cache the result.

● If the values change but not frequently, you can introduce a cache time where you can check when it was last cached and decide whether or not to call the API.

Why?

Having a caching mechanism means avoiding unwanted API calls. By only making the API calls when required and avoiding duplication, the speed of the application improves as we do not have to wait for the network. It also means we do not download the same information over and over again.

Avoid logic in templates

● If you have any sort of logic in your templates, even if it is a simple && clause, it is good to extract it out into its component.

Why?

Having logic in the template means that it is not possible to unit test it and therefore it is more prone to bugs when changing template code.

Before

```
// template

<p *ngIf="role==='developer'"> Status: Developer </p>

// component
public ngOnInit (): void {
   this.role = 'developer';
}
```

After

```
// template
<p *ngIf="showDeveloperStatus"> Status: Developer </p>

// component
public ngOnInit (): void {
   this.role = 'developer';
   this.showDeveloperStatus = true;
}
```

Strings should be safe

● If you have a variable of type string that can have only a set of values, instead of declaring it as a string type, you can declare the list of possible values as the type.

Why?

By declaring the type of the variable appropriately, we can avoid bugs while writing the code during compile time rather than during runtime.

Before

```
private myStringValue: string;

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

After

```
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"
```

Avoid having subscriptions inside subscriptions

Sometimes you may want values from more than one observable to perform an action. In this case, avoid subscribing to one observable in the subscribe block of another observable. Instead, use appropriate chaining operators. Chaining operators run on observables from the operator before them. Some chaining operators are: withLatestFrom, combineLatest, etc.

Before

```
firstObservable$.pipe(
  take(1)
)
.subscribe(firstValue => {
  secondObservable$.pipe(
    take(1)
  )
  .subscribe(secondValue => {
    console.log(`Combined values are: ${firstValue} & ${secondValue}`);
  });
});
```

After

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

Why?

Code smell/readability/complexity: Not using RxJs to its full extent, suggests developer is not familiar with the RxJs API surface area.

Performance: If the observables are cold, it will subscribe to firstObservable, wait for it to complete, THEN start the second observable's work. If these were network requests it would show as synchronous/waterfall.

Pipeable operators

● Use pipeable operators when using RxJs operators.

Why?

Pipeable operators are tree-shakeable meaning only the code we need to execute will be included when they are imported.

This also makes it easy to identify unused operators in the files.

Note: This needs Angular version 5.5+.

Before

```
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/take';

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

After

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

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

Use appropriate operators

When using flattening operators with your observables, use the appropriate operator for the situation.

● *switchMap:* when you want to ignore the previous emissions when there is a new emission
● *mergeMap:* when you want to concurrently handle all the emissions
● *concatMap:* when you want to handle the emissions one after the other as they are emitted
● *exhaustMap:* when you want to cancel all the new emissions while processing a previous emisssion

Why?

Using a single operator when possible instead of chaining together multiple other operators to achieve the same effect can cause less code to be shipped to the user. Using the wrong operators can lead to unwanted behaviour, as different operators handle observables in different ways.

Diagnostic Messages

● Use a period at the end of a sentence.
● Use indefinite articles for indefinite entities.
● Definite entities should be named (this is for a variable name, type name, etc..).
● When stating a rule, the subject should be in the singular (e.g. "An external module cannot..." instead of "External modules cannot...").
● Use present tense.

Diagnostic Message Codes

Diagnostics are categorized into general ranges. If adding a new diagnostic message, use the first integral number greater than the last used number in the appropriate range.
- 1000 range for syntactic messages
- 2000 for semantic messages
- 4000 for declaration emit messages
- 5000 for compiler options messages
- 6000 for command line compiler messages - 7000 for noImplicitAny messages

General Constructs

● Do not use ECMAScript 5 functions; instead use those found in core.ts.
● Do not use for..in statements; instead, use ts.forEach, ts.forEachKey and ts.forEachValue. Be aware of their slightly different semantics.
● Try to use ts.forEach, ts.map, and ts.filter instead of loops when it is not strongly inconvenient.

Style

● Use arrow functions over anonymous function expressions.
● Only surround arrow function parameters when necessary.
● For example, (x) => x + x is wrong but the following are correct:
  - x => x + x
  - (x,y) => x + y
  - <T>(x: T, y: T) => x === y
● Always surround loop and conditional bodies with curly braces. Statements on the same line are allowed to omit braces.
● Open curly braces always go on the same line as whatever necessitates them.

● Parenthesized constructs should have no surrounding whitespace.
A single space follows commas, colons, and semicolons in those constructs. For example:
for (var i = 0, n = str.length; i < 10; i++) { }
if (x < 10) { }
function f(x: number, y: string): void { }
Use a single declaration per variable statement
● (i.e. use var x = 1; var y = 2; over var x = 1, y = 2;).
● else goes on a separate line from the closing curly brace.
● Use 4 spaces per indentation