ANGULAR CODING STANDARDS

ITS 2021

Contents

1. Project Structure Practices

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
        |
        |-- [+] configs
    |-- assets
        |-- scss
        |-- [+] partials
```

```
        |-- _base.scss
        |-- styles.scss
```

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

1.1 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
        |-- [+] components
        |-- [+] models
        |-- [+] stores
        |-- [+] footer
        |-- [+] guards
        |-- [+] http
        |-- [+] interceptors
        |-- [+] mocks
        |-- [+] services
        |-- [+] header
        |-- core.module.ts
        |-- ensureModuleLoadedOnceGuard.ts
        |-- logger.service.ts
```

1.2 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
```

1.3 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

1.4 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
  }
];
```

1.5 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"
```

1.6 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.

1.7 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.

2.Typescript Guidelines

2.1 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.

2.2 Components

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

2.3 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.

2.4 null and undefined

- Use undefined. Do not use null.

2.5 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.

2.6 Classes

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

2.7 Flags

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

2.8 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).

2.10 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.

2.11 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

2.12 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.

2.13 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.

3.0 Coding Conventions Guidelines

3.1 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.

3.2 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

3.3 Interfaces

- Do name an interface using upper camel case.

3.4 Properties and methods

- Do use lower camel case to name properties and methods.
- Avoid prefixing private properties and methods with an underscore.

3.5 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.

3.6 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.

3.7 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.

3.8 Lifecycle hooks

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

4.0 Deployment

- Remove All console.log and debuggers from Projects files.
- Enable Angular Production Mode before taking build.
- Use AOT compilation build.
- --output-hashing all — hash contents of the generated files and append hash to the file name to facilitate browser cache busting (any change to file content will result in different hash and hence browser is forced to load a new version of the file)
- --extract-css true — extract all the css into separate style-sheet file
- --sourcemaps false — disable generation of source maps
- --named-chunks false — disable using human readable names for chunk and use numbers instead

5.0 External References

https://angular.io/guide/aot-compiler https://angular.io/guide/styleguide
https://wikipedia.org/wiki/Single_responsibility_principle
https://docs.npmjs.com/cli/version

https://stackoverflow.com/questions/273695/git-branch-naming-best-practices https://cli.angular.io/

https://medium.com/beautiful-angular/angular-2-and-environment-variables-59c57ba643be
https://github.com/johnpapa/angular-styleguide/blob/master/a2/README.md
https://github.com/Microsoft/TypeScript/wiki/Coding-guidelines

https://blog.angular-university.io/getting-started-with-angular-setup-a-development-environment- with-yarn-the-angular-cli-setup-an-ide/

https://gist.github.com/digitaljhelms/4287848 https://www.npmjs.com/package/codelyzer

https://medium.com/@tomastrajan/6-best-practices-pro-tips-for-angular-cli-better-developer-exp erience-7b328bc9db81

https://gist.github.com/revett/88ee5abf5a9a097b4c88 https://medium.com/@motcowley/angular-folder-structure-d1809be95542 https://medium.com/medialesson/why-and-how-to-structure-features-in-modules-in-angular-d56 02c6436be