Angular guidelines and best practices

Version 1.0.draft - March 2022

Contents

- Typescript coding guide
- HTML template coding guide
- CSS coding guide
- Angular best practices
- VS Code extensions
- External references

Typescript coding guide

Naming conventions

- **DO** use `PascalCase` for types, classes, interfaces, constants and enum values.
- **DO** use `camelCase` for variables, properties and functions
- **DO NOT** prefix interfaces with a capital `I`, see Angular style guide
- **DO NOT** use `_` as a prefix for private properties or parameters. An exception **COULD** be made for backing fields like this:

```
private _foo: string;
get foo() { return this._foo; } // foo is read-only to consumers
```

Ordering

- Within a file, type definitions **SHOULD** come first.
- Within a class, these priorities **SHOULD** be respected:
  - Properties **SHOULD** be found before functions
  - Static symbols **SHOULD** be found before instance symbols
  - Public symbols **SHOULD** be found before private symbols

Coding rules

- **DO** use single quotes `'` for strings.
- The number of lines in the file **SHOULD NOT** exceed 300.
- Always **DO** use strict equality checks: `===` and `!==` instead of `==` or `!=` to avoid comparison pitfalls (see JavaScript equality table).
- **DO** use `[]` instead of `Array` constructor.
- **DO** use `{}` instead of `Object` constructor.
- Always **DO** specify types for function parameters and returns (if applicable, otherwise `void`). **DO NOT** use `any`.
- **DO NOT** export types/functions unless you need to share it across multiple components.

- **DO NOT** introduce new types/values to the global namespace.
- **DO** use arrow functions over anonymous function expressions.

Definitions

In order to infer types from JavaScript modules, TypeScript language supports external type definitions. They are located in the `node_modules/@types` folder.

To manage type definitions, **DO** use standard `npm install|update|remove` commands with the `--save-dev` flag.

Enforcement

Coding rules **SHOULD** be enforced in the project via ESLint. If an Nx workspace is used, then **DO** only configure the `parserOptions.project` option in the project's `.eslintrc.json` when you need to use rules requiring type information.

tsconfig.json

TODO

HTML template coding guide

Naming conventions

- Everything **SHOULD** be named in `kebab-case` (lowercase words separated with a hyphen): tags, attributes, IDs, etc, except for everything bound to Angular such as variables, directives or events which should be in `camelCase`.
- File names **SHOULD** always be in `kebab-case`.

Coding rules

- All templates **SHOULD** be extracted in separate files, when more than 3 lines. Only use inline templates sparingly in very simple components with less than 3 lines of HTML.
- **DO** use double quotes " around attribute values in tags.
- **DO** use a new line for every block, list, or table element, and indent every such child element.
- Clear separation of structure (HTML) from presentation (CSS) from behavior (JavaScript):
  - **DO NOT** ever use inline CSS or JavaScript.
  - **DO NOT** keep any logic in the HTML.

Common pitfalls

- Block-type tags cannot be nested inside inline-type tags: i.e. a `<div>` tag cannot be nested in a `<span>`. This rule also applies regarding the display value of an element.
- HTML is **not** XML: empty tags cannot be self-closing and will result in improper results
  - `<div/>` will be interpreted as a simple `<div>` without closing tag!
  - The only tags that allows self-closing are the one that does not require a closing tag in first place: these are the void elements that do not not accept content `<br>`, `<hr>`, `<img>`, `<input>`, `<meta>`, `<link>` (and others).

CSS coding guide

Naming conventions

- Everything **SHOULD** be named in `kebab-case` (lowercase words separated with a `-`).
- File names **SHOULD** always be in `kebab-case`.

Coding rules

- When using CSS preprocessors such as Less/Sass, the following nesting hierarchy **SHOULD** be used:

```
/* The base component class acts as the namespace, to avoid naming and style
collisions */
.my-component {
  /* Put here all component elements (flat) */
  .my-element {
    /* Use a third-level only for modifiers and state variations */
    &.active { ... }
  }
}
```

- **DO** use single quotes ' for strings.
- **DO** use classes selectors, never use ID or element selectors.
- **DO NOT** use more than 3 levels of nesting.
- **DO NOT** use more than 3 qualifiers.

Best practices

- **DO** use object-oriented CSS (OOCSS):
  - **DO** factorize common code in base class, and extend it, for example:

```
/* Base button class */
.btn { ... }

/* Color variation */
.btn-warning { ... }

/* Size variation */
.btn-small { ... }
```

- **DO** name class by semantic, not style nor function for better reusability: Use `.btn-warning`, not `.btn-orange` nor `.btn-cancel`.
- **DO NOT** undo style. Instead refactor using common base classes and extensions.
- **DO** keep your style scoped:
  - **DO** clearly separate **global** (think `framework`) and **modules** (`components`) style.
  - Global style **SHOULD** only go in `app/theme/*` or `app/helpers.scss` (never in modules).

    - **DO NOT** share styles between modules. If some style may need to be shared, refactor it as a framework component and put it in your global theme.
    - **DO NOT** use wider selectors than needed (always use classes!)
- Avoid rules multiplication
    - The less, the better, **DO** factorize rules whenever it is possible.
    - CSS is code, and like any code frequent refactoring is healthy.
- When ugly hacks cannot be avoided, **DO** place them in `app/hacks.scss`:
    - These ugly hacks **SHOULD** only be temporary.
    - Each hack **SHOULD** be documented with the author name, the problem and hack reason.
    - **DO** limit this file to a reasonable length (~100 lines) and refactor hacks with proper solutions when the limit is reached.

Common pitfalls

- **DO NOT** use the `!important` keyword. Using `!important` is bad practice and **SHOULD** be avoided.
- **DO NOT** use inline style in html, even just for debugging (because we KNOW it will end up in your commit).
- **DO NOT** use browser-specific prefixes: there are tools taking care of that part (e.g. autoprefixer)

Angular best practices

Application folder structure

An angular application **SHOULD** contain 3 main folders. The `app` folder contains all application code, the `assets` folder contains all static content and the `environments` folder contains configuration information.

```
|-- [+] app
        |-- [+] @core
        |-- [+] @shared
        |-- [+] feature-module-1
        |-- [+] feature-module-2
        |-- app.routing.module.ts
        |-- app.component.ts|html|less|spec
        |-- app.module.ts
|-- [+] assets
        |-- [+] scss
        |-- [+] less
        |-- [+] i18n
        |-- [+] images
|-- [+] environments
        |-- environment.ts
        |-- environment.dev.ts
        |-- environment.uat.ts
        |-- environment.prod.ts
```

- The `app` folder **SHOULD** contain only the `app` module, the `app-routing` module and the `app` component.
- The `assets` folder **SHOULD** include

    - CSS related folders (e.g. `scss` or `less`).
    - internationalization files - e.g. translation files.
    - images
    - other static files if needed (e.g. font files)
- The `environments` **SHOULD** contain one general configuration file and one configuration file per environment that the app is deployed.

Core module

The `CoreModule` **SHOULD** contain only singleton services (which is usually the case), universal components and other features where there's only once instance per application. To prevent re-importing the core module elsewhere, you **SHOULD** also add a guard for it in the core module' constructor. The core module **SHOULD NOT** have a routing module.

```
|-- core
    |-- [+] auth
    |-- [+] layout
            |-- [+] header
            |-- [+] footer
    |-- [+] guards
            |-- auth.guard.ts
            |-- no-auth-guard.ts
            |-- admin-guard.ts
    |-- [+] interceptors
            |-- api-prefix.interceptor.ts
            |-- error-handler.interceptor.ts
            |-- http-token.interceptor.ts
    |-- [+] services
            |-- logger.service.ts
    |-- [+] mocks
    |-- core.module.ts
```

- The `auth` folder contains everything to handle the authentication-cycle of the user (from login to logout).
- The `footer` and `header` folders (under `layout` folder) contains the global component-files, statically used across the entire application. These files will appear on every page in the application.
- The `interceptors` folder **SHOULD** contain all the `HttpInterceptor`s used by the app.
- The `guards` folder **SHOULD** contain all the guards used to protect different routes in the application.
- The `services` folder **SHOULD** contain all other additional **singleton** services.
- The `mocks` folder contains all the mock-files of the application. Mocks are specially useful for testing, but can also be used to retrieve fictional data until the back-end is set up.

Shared module

The `SharedModule` is where any shared components, pipes/filters and services **SHOULD** go. The `SharedModule` can be imported in any other module when those items need to be reused. The shared module **SHOULD NOT** have any dependency to the rest of the application and therefore it **SHOULD NOT** rely on any other module.

TODO

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

- The `components` folder **SHOULD** contain all the "shared" components. This are components like `loaders` and `buttons`, which multiple components would benefit from.
- The `directives`, `pipes` and `models` folders **SHOULD** contain the directives, pipes and models used across the application.

Lazy loading feature modules

The application **SHOULD** use lazy-loading, which means that a feature module is not loaded before the user actually accesses its routes. By using the structure described in this section, the main `app-routing` file **SHOULD** contain a lazy reference for each feature module.

```
|-- [-] modules
    |-- [-] home
        |-- [+] components
        |-- [+] pages
                |-- [-] home
                        |-- home.component.ts|html|scss|spec
        |-- [+] services
        |-- home-routing.module.ts
        |-- home.module.ts
```

State management using RxJS

TODO

I18n (internationalization)

TODO

- Directionality (LTR / RTL) should be taken care while designing
- All content should be retreived from the localization files and consumed using translate pipe

Reactive forms

TODO

Lookup caching

TODO

Logging

- For all core and/or complex logic **DO** use `try/catch` blocks and when an error occurs **DO** use the `xxxLogService` to send the information at the backend.
- Error interceptor should be implemented and errors must be logged in backend

TODO

Common Angular development best practices

- **DO** use Angular CLI for creating components, services, etc.
- **DO** setup Prettier formatter as default formatting tool.
- To avoid memory leaks, you **SHOULD** unsubscribe from all subsriptions that are created in a component (either on `ngOnDestroy` or earlier).

VS Code extensions

DO install the following extensions in VS Code:

- **Angular Language Service**: provides a rich editing experience for Angular templates, both inline and external templates.
- **Prettier - Code formatter**: enforces a consistent style by parsing your code and re-printing it with its own rules that take the maximum line length into account, wrapping code when necessary.
- **Auto Rename Tag**: automatically renames paired HTML/XML tag.
- **ESLint**: integrates ESLint to find and fix problems in JavaScript code.
- **GitLens**: git enhancements for VS Code.
- **Nx Console**: is the UI for nx.

External references

- Angular style guide
- TypeScript coding guidelines
- TypeScript Deep Dive Style Guide
- TSConfig Reference