#### **Angular Style Guide**

#### Style 01-01

* Do define one thing, such as a service or component, per file.
* Consider limiting files to 400 lines of code.

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

Why? 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.

Why? A single component can be the default export for its file which facilitates lazy loading with the router.

The key is to make the code more reusable, easier to read, and less mistake prone.

#### Style 01-02

* Do define small functions
* Consider limiting to no more than 75 lines.

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

Why? Small functions promote reuse.

Why? Small functions are easier to read.

Why? Small functions are easier to maintain.

Why? 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.

Style 02-01(File Naming)

* Do use consistent names for all symbols.
* Do follow a pattern that describes the symbol's feature then its type. The recommended pattern “feature.type.ts”.

Why? Naming conventions help provide a consistent way to find content at a glance. Consistency within the project is vital. Consistency with a team is important. Consistency across a company provides tremendous efficiency.

Why? The naming conventions should simply help find desired code faster and make it easier to understand.

Why? Names of folders and files should clearly convey their intent. For example, app/heroes/hero-list.component.ts may contain a component that manages a list of heroes.

#### Style 02-02

* Do use dashes to separate words in the descriptive name.
* Do use dots to separate the descriptive name from the type.
* Do use consistent type names for all components following a pattern that describes the component's feature then its type. A recommended pattern is “feature.type.ts”.
* Do use conventional type names including .service, .component, .pipe, .module, and .directive. Invent additional type names if you must but take care not to create too many.

Why? Type names provide a consistent way to quickly identify what is in the file.

Why? Type names make it easy to find a specific file type using an editor or IDE's fuzzy search techniques.

Why? Unabbreviated type names such as .service are descriptive and unambiguous. Abbreviations such as .srv, .svc, and .serv can be confusing.

Why? Type names provide pattern matching for any automated tasks.

#### Style 02-03

* Do use lowercase dashed-case or kebab-case for naming the element selectors of components.

Example: “app-root”, “hr-dashboard”

* Do Use lower camel case for naming the selectors of directives.

Example: ‘appValidator’

#### Style 03-01(Naming)

* Do use UpperCamelCase for naming **Class**

Example: “AppDashboard”

* Do use lowerCamelCase for naming **variables, objects, arrays, constants and functions**

Example: “appBaseUrl, setBaseUrl()”.

* Do use lower camel case to name properties object

Why? Lower camel case variable names (appBaseUrl) are easier to read and understand than the traditional UPPER\_SNAKE\_CASE names (APP\_BASE\_URL).

#### Style 04-01(Importing)

* 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.

#### Style 05-01(File Structure)

* All of the app's code goes in a folder named src
* Use LIFT principle for coding

L – Do make locating code intuitive, simple and fast.

I - 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.

F - Do keep a flat folder structure as long as possible.

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

T - Try to be Dry(Don't Repeat Yourself )

#### **Style 05-02**

#### Do create folders named for the feature area they represent.

* Do create an NgModule for each feature area.
* Do create an NgModule in the app's root folder.

Why? A developer can locate the code and identify what each file represents at a glance. The structure is as flat as it can be and there are no repetitive or redundant names.

Why? The LIFT guidelines are all covered.

Why? Helps reduce the app from becoming cluttered through organizing the content and keeping them aligned with the LIFT guidelines.

Why? When there are a lot of files, for example 10+, locating them is easier with a consistent folder structure and more difficult in a flat structure.

Why? NgModules make it easy to lazy load routable features.

Why? NgModules make it easier to isolate, test, and reuse features.

#### Style 05-03(Feature Module)

* Do create an NgModule for all distinct features in an application;
* Do place the feature module in the same named folder as the feature area;
* Do name the feature module file reflecting the name of the feature area and folder; for example, app/heroes/heroes.module.ts.
* Do name the feature module symbol reflecting the name of the feature area, folder, and file; for example,app/heroes/heroes.module.ts defines HeroesModule.

Why? A feature module can expose or hide its implementation from other modules.

Why? A feature module identifies distinct sets of related components that comprise the feature area.

Why? A feature module can easily be routed to both eagerly and lazily.

Why? A feature module defines clear boundaries between specific functionality and other application features.

Why? A feature module helps clarify and make it easier to assign development responsibilities to different teams.

Why? A feature module can easily be isolated for testing.

#### Style 05-04(Shared Feature Module)

* Do create a feature module named SharedModule in a shared folder; for example, app/shared/shared.module.ts defines SharedModule.
* Do declare components, directives, and pipes in a shared module when those items will be re-used and referenced by the components declared in other feature modules.
* Consider using the name SharedModule when the contents of a shared module are referenced across the entire application.
* Do import all modules required by the assets in the SharedModule; for example, [CommonModule](https://angular.io/api/common/CommonModule) and [FormsModule](https://angular.io/api/forms/FormsModule).

#### Style 05-05(Lazy Loading)

* A distinct application feature or workflow may be lazy loaded or loaded on demand rather than when the application starts.
* Do put the contents of lazy loaded features in a lazy loaded folder. A typical lazy loaded folder contains a routing component, its child components, and their related assets and modules.

Why? The folder makes it easy to identify and isolate the feature content.

#### Style 06-01

* 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 ./.

#### Style 06-02

* Do put presentation logic in the component class, and not in the template.
* Use Directive to enhance the element feature
* Write complex logics in services rather than component

#### Style 06-03

* Do use the lifecycle hooks correctly to prevent the memory leakage