LevelUp Angular Style Guide

* **Single Responsibility Principle**
  + **Do** define one thing, such as a service or component, per file.
  + **Consider** limiting files to 400 lines of code.
* **Small Functions**
  + **Do** define small functions
  + **Consider** limiting to no more than 75 lines.
* **General Naming Guidelines**
  + **Do** use consistent names for all symbols.
  + **Do** follow a pattern that describes the symbol's feature then its type. The recommended pattern is feature.type.ts.
* **File names with dots and dashes**
  + **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.
* **Symbols and Filenames** 
  + Code generation, like @angular/cli, should handle symbol suffixing for developers
  + **Do** use consistent names for all assets named after what they represent.
  + **Do** use upper camel case for class names, i.e. AppComponent.
  + **Do** match the name of the symbol to the name of the file.
  + **Do** append the symbol name with the conventional suffix (such as Component, Directive, Module, Pipe, or Service) for a thing of that type.
  + **Do** give the filename the conventional suffix (such as .component.ts, .directive.ts, .module.ts, .pipe.ts, or .service.ts) for a file of that type.
* **Service Names**
  + **Do** use consistent names for all services named after their feature.
  + **Do** suffix a service class name with Service. For example, something that gets data or heroes should be called a DataService, LoggerService or a HeroService.
* **Bootstrapping**
  + **Do** put bootstrapping and platform logic for the app in a file named main.ts.
  + **Do** include error handling in the bootstrapping logic.
* **Selectors**
  + Lowercase with hyphens if exists (such as app-root)
* **Custom Prefixes for Component Selectors**
  + **Do** use a hyphenated, lowercase element selector value (e.g. admin-users).
  + **Do** use a prefix that identifies the feature area or the app itself.
  + **Do** use a custom prefix for a component selector. For example, the prefix toh represents from **T**our **o**f **H**eroes and the prefix admin represents an admin feature area.
* **Custom Prefixes for Directives**
  + **Do** use a custom prefix for the selector of directives (e.g, the prefix toh from **T**our **o**f **H**eroes).
  + **Do** spell non-element selectors in lower camel case unless the selector is meant to match a native HTML attribute.
* **Pipe Names**
  + **Do** use consistent names for all pipes, named after their feature.
* **Unit Test Files Names**
  + **Do** name test specification files the same as the component they test.
  + **Do** name test specification files with a suffix of .spec.
* **End-to-End Test File Names**
  + **Do** name end-to-end test specification files after the feature they test with a suffix of .e2e-spec.
* **Module Names**
  + **Do** append the symbol name with the suffix Module.
  + **Do** give the file name the .module.ts extension.
  + **Do** name the module after the feature and folder it resides in.
* **Routing Module Name**
  + See Module Names
  + **Do** suffix a *RoutingModule* class name with RoutingModule.
  + **Do** end the filename of a *RoutingModule* with -routing.module.ts.
* **Class Names**
  + **Do** use upper camel case when naming classes such as AppComponent.
* **Constants**
  + **Do** declare variables with const if their values should not change during the application lifetime.
  + **Consider** spelling const variables in lower camel case.
* **Interfaces**
  + **Do** name an interface using upper camel case.
  + **Consider** naming an interface without an I prefix.
  + **Consider** using a class instead of an interface.
* **Property and Method Names**
  + **Do** use lower camel case to name properties and methods.
  + **Avoid** prefixing private properties and methods with an underscore.
* **Import (Angular import statement) Line Spacing**
  + **Consider** leaving one empty line between third party imports and application imports.
  + **Consider** listing import lines alphabetized by the module. (\*\* look to automate sorting)
  + **Consider** listing destructured imported symbols alphabetically.
* **Application Structure and Angular Modules**
  + **All** app code goes in the src folder.
  + **All** feature areas are in their own folder with their own Angular Module.
  + **All** third party vendor scripts are stored in another folder and not in the src folder.
* **Folders-by-feature structure**
  + **Do** create folders named for the feature area they represent.
  + **Do** create an Angular module for each feature area.
  + **Do** create an Angular module in the app's root folder, for example, in /src/app.
  + **Consider** naming the root module app.module.ts.
* **Feature modules**
  + **Do** create an Angular module for all distinct features in an application; for example, a Heroes feature.
  + **Do** place the feature module in the same named folder as the feature area; for example, in app/heroes.
  + **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.
* **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** not provide services in shared modules. Services are usually singletons that are provided once for the entire application or in a particular feature module.
  + **Do** import all modules required by the assets in the SharedModule; for example, CommonModule and FormsModule.
  + **Why?** SharedModule will contain components, directives and pipes that may need features from another common module; for example, ngFor in CommonModule.
  + **Do** declare all components, directives, and pipes in the SharedModule.
  + **Do** export all symbols from the SharedModule that other feature modules need to use.
* **Core feature module**
  + **Consider** collecting numerous, auxiliary, single-use classes inside a core module to simplify the apparent structure of a feature module.
  + **Consider** calling the application-wide core module, CoreModule. Importing CoreModule into the root AppModule reduces its complexity and emphasizes its role as orchestrator of the application as a whole.
  + **Do** create a feature module named CoreModule in a core folder (e.g. app/core/core.module.ts defines CoreModule).
  + **Do** put a singleton service whose instance will be shared throughout the application in the CoreModule (e.g. ExceptionService and LoggerService).
  + **Do** import all modules required by the assets in the CoreModule (e.g. CommonModule and FormsModule).
  + **Do** gather application-wide, single use components in the CoreModule. Import it once (in the AppModule) when the app starts and never import it anywhere else. (e.g. NavComponent and SpinnerComponent).
  + **Do** export all symbols from the CoreModule that the AppModule will import and make available for other feature modules to use.
  + Only the root AppModule should import the CoreModule.
  + **Do** guard against reimporting of CoreModule and fail fast by adding guard logic.
* **Lazy Loaded folders**
  + **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.
* **Never directly import lazy loaded folders**
  + **Avoid** allowing modules in sibling and parent folders to directly import a module in a *lazy loaded feature*.
  + **Why?** Directly importing and using a module will load it immediately when the intention is to load it on demand.
* **Component selector names**
  + **Do** use *dashed-case* or *kebab-case* for naming the element selectors of components.
* **Components as elements**
  + **Do** give components an *element* selector, as opposed to *attribute* or *class* selectors.
* **Extract templates and styles to their own files**
  + **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 ./.
* **Decorate input and output properties**
  + **Do** use the @Input() and @Output() class decorators instead of the inputs and outputs properties of the @Directive and @Componentmetadata:
  + **Consider** placing @Input() or @Output() on the same line as the property it decorates.
* **Avoid aliasing inputs and outputs**
  + **Avoid** *input* and *output* aliases except when it serves an important purpose.
  + **Why?** Two names for the same property (one private, one public) is inherently confusing.
* **Sequence for Properties and Methods**
  + Do place properties up top followed by methods.
  + Do place private members after public members, alphabetized.
* **Delegate complex component logic to services**
  + **Do** limit logic in a component to only that required for the view. All other logic should be delegated to services.
  + **Do** move reusable logic to services and keep components simple and focused on their intended purpose.
* **Don't prefix output properties**
  + **Do** name events without the prefix on.
  + **Do** name event handler methods with the prefix on followed by the event name.
* **Put presentation logic in the component class**
  + **Do** put presentation logic in the component class, and not in the template.
* **Use directives to enhance an element**
  + **Do** use attribute directives when you have presentation logic without a template.
* **HostListener/HostBinding decorators versus host metadata**
  + Consider preferring the @HostListener and @HostBinding to the host property of the @Directive and @Component decorators.
  + **Do** be consistent in your choice.
* **Services are singletons**
  + **Do** use services as singletons within the same injector. Use them for sharing data and functionality.
  + **Do** create services with a single responsibility that is encapsulated by its context.
  + **Do** create a new service once the service begins to exceed that singular purpose.
  + **Do** provide services to the Angular injector at the top-most component where they will be shared.
* **Use the @Injectable() class decorator**
  + **Do** use the @Injectable() class decorator instead of the @Inject parameter decorator when using types as tokens for the dependencies of a service.
* **Talk to the server through a service**
  + **Do** refactor logic for making data operations and interacting with data to a service.
  + **Do** make data services responsible for XHR calls, local storage, stashing in memory, or any other data operations.
* **Implement lifecycle hook interfaces**
  + **Do** implement the lifecycle hook interfaces.
* **Code style enforcement**
  + **Do** use [codelyzer](https://www.npmjs.com/package/codelyzer) to follow this guide.
* **Code Snippets**
  + **Do** use file templates or snippets to help follow consistent styles and patterns. Here are templates and/or snippets for some of the web development editors and IDEs. Found here [snippets](https://marketplace.visualstudio.com/items?itemName=johnpapa.Angular2)