Shriram Value Services



Coding Standards for Angular 2

Table of Contents

[1. Single responsibility 4](#_Toc485893447)

[1.1 Rule of one 4](#_Toc485893448)

[1.2. Small functions 4](#_Toc485893449)

[2. Naming 4](#_Toc485893450)

[2.1. General naming guidelines 4](#_Toc485893451)

[2.2. Separate file names with dots and dashes 5](#_Toc485893452)

[2.3. Symbols and file names 5](#_Toc485893453)

[2.4. Service names 5](#_Toc485893454)

[2.5. Bootstrapping 6](#_Toc485893455)

[2.6. Directive selectors 6](#_Toc485893456)

[2.7. Custom prefix for components 6](#_Toc485893457)

[2.8. Custom prefix for directives 7](#_Toc485893458)

[2.9. Pipe names 7](#_Toc485893459)

[2.10. Unit test file names 8](#_Toc485893460)

[2.11. End-to-End (E2E) test file names: 8](#_Toc485893461)

[2.12. Angular NgModule names 8](#_Toc485893462)

[3. Coding conventions 8](#_Toc485893463)

[3.1. Variable & Function 9](#_Toc485893464)

[3.2. Enum 9](#_Toc485893465)

[3.3. Array 9](#_Toc485893466)

[3.4. Classes 9](#_Toc485893467)

[3.5. Constants 10](#_Toc485893468)

[3.6. Interfaces 10](#_Toc485893469)

[3.7. Properties and methods 11](#_Toc485893470)

[3.8. Import line spacing 12](#_Toc485893471)

[4. Application structure and Angular modules 12](#_Toc485893472)

[4.1. LIFT 12](#_Toc485893473)

[4.2. Locate 13](#_Toc485893474)

[4.3. Identify 13](#_Toc485893475)

[4.4. Flat 13](#_Toc485893476)

[4.5. T-DRY (Try to be DRY) 13](#_Toc485893477)

[4.6. Overall structural guidelines 14](#_Toc485893478)

[4.7. Folders-by-feature structure 15](#_Toc485893479)

[4.8. App root module 16](#_Toc485893480)

[4.9. Feature modules 16](#_Toc485893481)

[4.10. Shared feature module 17](#_Toc485893482)

[4.11. Core feature module 18](#_Toc485893483)

[4.12. Prevent re-import of the core module 18](#_Toc485893484)

[4.13. Lazy Loaded folders 19](#_Toc485893485)

[4.14. Never directly import lazy loaded folders 19](#_Toc485893486)

[5. Components 19](#_Toc485893487)

[5.1 Component selector names 19](#_Toc485893488)

[5.2. Components as elements 19](#_Toc485893489)

[5.3. Extract templates and styles to their own files 20](#_Toc485893490)

[5.4. Decorate input and output properties 21](#_Toc485893491)

[5.5. Avoid aliasing inputs and outputs 22](#_Toc485893492)

[5.6. Member sequence 23](#_Toc485893493)

[5.7. Delegate complex component logic to services 24](#_Toc485893494)

[5.8. Don't prefix output properties 25](#_Toc485893495)

[5.9. Put presentation logic in the component class 26](#_Toc485893496)

[6. Directives 27](#_Toc485893497)

[6.1. Use directives to enhance an element 27](#_Toc485893498)

[6.2. HostListener/HostBinding decorators versus host metadata 27](#_Toc485893499)

[7. Services 28](#_Toc485893500)

[7.1. Services are singletons 28](#_Toc485893501)

[7.2. Single responsibility 29](#_Toc485893502)

[7.3. Providing a service 29](#_Toc485893503)

[7.4. Use the @Injectable() class decorator 29](#_Toc485893504)

[8. Data Services 31](#_Toc485893505)

[8.1. Talk to the server through a service 31](#_Toc485893506)

[9. Lifecycle hooks 31](#_Toc485893507)

[9.1. Implement lifecycle hook interfaces 31](#_Toc485893508)

[10. Best Practices 32](#_Toc485893509)

# 1. Single responsibility

* Apply the [*single responsibility principle* (SRP)](https://wikipedia.org/wiki/Single_responsibility_principle) to all components, services, and other symbols. This helps make the app cleaner, easier to read and maintain, and more testable.

## 1.1 Rule of one

* **Consider** limiting files to 400 lines of code.

##### ***Explanation***

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

## 1.2. Small functions

* **Consider** limiting to no more than 75 lines.

***Explanation***

* 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.

# 2. Naming

* Naming conventions are hugely important to maintainability and readability. This guide recommends naming conventions for the file name and the symbol name.

## 2.1. 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.

***Explanation***

* 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.
* The naming conventions should simply help find desired code faster and make it easier to understand.
* Names of folders and files should clearly convey their intent. For example, app/heroes/hero-list.component.ts may

## 2.2. Separate 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.

***Explanation***

* Type names provide a consistent way to quickly identify what is in the file.
* Type names make it easy to find a specific file type using an editor or IDE's fuzzy search techniques.
* Unabbreviated type names such as .service are descriptive and unambiguous. Abbreviations such as .srv, .svc, and .serv can be confusing.
* Type names provide pattern matching for any automated tasks.

## 2.3. Symbols and file names

* **Do** use consistent names for all assets named after what they represent.
* **Do** use upper camel case for class names.
* **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.

***Explanation***

* Consistent conventions make it easy to quickly identify and reference assets of different types.

|  |  |
| --- | --- |
| ***Symbol Name*** | ***File Name*** |
| *export class AppComponent { }* | *app.component.ts* |
| *export class HeroesComponent { }* | *heroes.component.ts* |

## 2.4. 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 or a HeroService.

*Explanation*

* Provides a consistent way to quickly identify and reference services.
* Clear service names such as Logger do not require a suffix.
* Service names such as Credit are nouns and require a suffix and should be named with a suffix when it is not obvious if it is a service or something else.

|  |  |
| --- | --- |
| ***Symbol Name*** | ***File Name*** |
| *export class HeroDataService { }* | *hero-data.service.ts* |
| *export class CreditService { }* | *credit.service.ts* |

## 2.5. 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.

*Explanation*

* Follows a consistent convention for the startup logic of an app.
* Follows a familiar convention from other technology platforms.

*import { platformBrowserDynamic } from '@angular/platform-browserdynamic';*

*import { AppModule } from './app/app.module';*

*platformBrowserDynamic().bootstrapModule(AppModule)*

*.then(success => console.log(`Bootstrap success`))*

*.catch(err => console.error(err));*

## 2.6. Directive selectors

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

***Explanation***

* Keeps the names of the properties defined in the directives that are bound to the view consistent with the attribute names.
* The Angular HTML parser is case sensitive and recognizes lower camel case.

## 2.7. Custom prefix for components

* **Do** use a hyphenated, lowercase element selector value (e.g. admin-users).
* **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 adminrepresents an admin feature area.
* **Do** use a prefix that identifies the feature area or the app itself.

***Explanation***

* Prevents element name collisions with components in other apps and with native HTML elements.
* Makes it easier to promote and share the component in other apps.
* Components are easy to identify in the DOM.

*/\* avoid \*/*

*// UsersComponent is in an Admin feature*

*@Component({*

*selector: 'users'*

*})*

*export class UsersComponent {}*

*/\* correct code \*/*

*@Component({*

*selector: 'admin-users'*

*})*

*export class UsersComponent {}*

## 2.8. Custom prefix 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.

***Explanation***

* Prevents name collisions.
* Directives are easily identified.

*/\* avoid \*/*

*@Directive({*

*selector: '[validate]'*

*})*

*export class ValidateDirective {}*

*/\* correct code \*/*

*@Directive({*

*selector: '[tohValidate]'*

*})*

*export class ValidateDirective {}*

## 2.9. Pipe names

* **Do** use consistent names for all pipes, named after their feature.

***Explanations***

* Provides a consistent way to quickly identify and reference pipes.

|  |  |
| --- | --- |
| ***Symbol name*** | ***File name*** |
| export class EllipsisPipe implements PipeTransform { } | ellipsis.pipe.ts |
| *export class InitCapsPipe implements PipeTransform { }* | *init-caps.pipe.ts* |

## 2.10. Unit test file names

* **Do** name test specification files the same as the component they test.
* **Do** name test specification files with a suffix of .spec.

***Explanation***

* Provides a consistent way to quickly identify tests.
* Provides pattern matching for [karma](http://karma-runner.github.io/) or other test runner.

|  |  |
| --- | --- |
| ***Symbol Name*** | ***File Name*** |
| *Components* | *heroes.component.spec.ts* |
| *Services* | *logger.service.spec.ts* |
| *Pipes* | *ellipsis.pipe.spec.ts* |

## 2.11. End-to-End (E2E) test file names:

* **Do** name end-to-end test specification files after the feature they test with a suffix of .e2e-spec.

***Explanation***

* Provides a consistent way to quickly identify end-to-end tests.
* Provides pattern matching for test runners and build automation.

|  |  |
| --- | --- |
| ***Symbol Name*** | ***File Name*** |
| *End-to-End Tests* | *app.e2e-spec.ts* |

## 2.12. Angular NgModule 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.

***Explanation:***

* Provides a consistent way to quickly identify and reference modules.
* Upper camel case is conventional for identifying objects that can be instantiated using a constructor.
* Easily identifies the module as the root of the same named feature.
* **Do** suffix a RoutingModule class name with RoutingModule.
* **Do** end the filename of a RoutingModule with -routing.module.ts.
* A RoutingModule is a module dedicated exclusively to configuring the Angular router. A consistent class and file name convention make these modules easy to spot and verify.

|  |  |
| --- | --- |
| ***Symbol Name*** | ***File Name*** |
| *export class AppModule { }* | *app.module.ts* |
| *export class HeroesModule { }* | *heroes.module.ts* |

# 3. Coding conventions

* Have a consistent set of coding, naming, and whitespace conventions.

## 3.1. Variable & Function

* **Do** use camel case when naming variables & function.
* **Do** use proper data type of the variable being used.
* **Do** specify the function syntax properly and make sure it matches with the function definition
* **Do** specify the scope of the variable (private, public, protected)

*/\* avoid \*/*

*var FooVar;*

*function FooFunction() {*

*return 1;*

*}*

*/\* correct code \*/*

*private fooVar : number;*

*function fooFunction() : number {*

*return 1;*

*}*

## 3.2. Enum

* **Do** use Pascal Case when naming Enum.

*/\* avoid \*/*

*enum color;*

*/\* correct code \*/*

*enum Color;*

## 3.3. Array

* **Do** annotate array as foo: number[] instead of foo: Array<number>

*/\* avoid \*/*

*foo: Array<number>;*

*/\* correct code \*/*

*foo: number[];*

## 3.4. Classes

* **Do** use upper camel case when naming classes.

***Explanation***

* Follows conventional thinking for class names.
* Classes can be instantiated and construct an instance. By convention, upper camel case indicates a constructible asset.

*/\* avoid \*/*

*export class exceptionService {*

*constructor() { }*

*}*

*/\* correct code \*/*

*export class ExceptionService {*

*constructor() { }*

*}*

## 3.5. Constants

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

***Explanation:***

* Conveys to readers that the value is invariant.
* TypeScript helps enforce that intent by requiring immediate initialization and by preventing subsequent re-assignment.
* **Consider** spelling const variables in lower camel case.
* Lower camel case variable names (heroRoutes) are easier to read and understand than the traditional UPPER\_SNAKE\_CASE names (HERO\_ROUTES).
* The tradition of naming constants in UPPER\_SNAKE\_CASE reflects an era before the modern IDEs that quickly reveal the constdeclaration. 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. It is rarely worth the effort to change them at the risk of breaking existing code and documentation.app/shared/d

*export const mockHeroes = ['Sam', 'Jill']; // prefer*

*export const heroesUrl = 'api/heroes'; // prefer*

*export const VILLAINS\_URL = 'api/villains'; // tolerate*

## 3.6. 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.

***Explanation***

* [TypeScript guidelines](https://github.com/Microsoft/TypeScript/wiki/Coding-guidelines) discourage the Iprefix.
* A class alone is less code than a class-plus-interface.
* A class can act as an interface (use implements instead of extends).
* An interface-class can be a provider lookup token in Angular dependency injection.

*/\* avoid \*/*

*import { Injectable } from '@angular/core';*

*import { IHero } from './hero.model.avoid';*

*@Injectable()*

*export class HeroCollectorService {*

*hero: IHero;*

*constructor() { }*

*}*

*/\* correct code \*/*

*import { Injectable } from '@angular/core';*

*import { Hero } from './hero.model';*

*@Injectable()*

*export class HeroCollectorService {*

*hero: Hero;*

*constructor() { }*

*}*

## 3.7. Properties and methods

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

***Explanation***

* Follows conventional thinking for properties and methods.
* JavaScript lacks a true private property or method.
* TypeScript tooling makes it easy to identify private vs. public properties and methods.

*/\* avoid \*/*

*message: string;*

*private \_toastCount: number;*

*private \_log() {*

*console.log(this.message);*

*}*

*/\* correct code \*/*

*message: string;*

*private toastCount: number;*

*private log() {*

*console.log(this.message);*

*}*

## 3.8. 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 de-structured imported symbols alphabetically.

***Explanation***

* The empty line separates *your* stuff from *their* stuff.
* Alphabetizing makes it easier to read and locate symbols.

/\* avoid \*/

*import { ExceptionService, SpinnerService, ToastService } from'../../core';*

*import { Http } from '@angular/http';*

*import { Hero } from './hero.model';*

*/\* correct code \*/*

*import { Injectable } from '@angular/core';*

*import { Http } from '@angular/http';*

*import { Hero } from './hero.model';*

*import { ExceptionService, SpinnerService, ToastService } from '../../core';*

# 4. Application structure and Angular modules

* Have a near-term view of implementation and a long-term vision. Start small but keep in mind where the app is heading down the road.
* All of the app's code goes in a folder named src. All feature areas are in their own folder, with their own Angular module.
* All content is one asset per file. Each component, service, and pipe is in its own file. All third party vendor scripts are stored in another folder and not in the src folder. You didn't write them and you don't want them cluttering src. Use the naming conventions for files in this guide.

## 4.1. LIFT

* **Do** structure the app such that you can **L**ocate code quickly, **I**dentify the code at a glance, keep the **F**lattest structure you can, and **T**ry to be DRY.
* **Do** define the structure to follow these four basic guidelines, listed in order of *importance.*

***Explanation***

* LIFT Provides a consistent structure that scales well, is modular, and makes it easier to increase developer efficiency by finding code quickly. To confirm your intuition about a particular structure, ask: can I quickly open and start work in all of the related files for this feature?

## 4.2. Locate

* **Do** make locating code intuitive, simple and fast.

***Explanation***

* To work efficiently you must be able to find files quickly, especially when you do not know (or do not remember) the file names. Keeping related files near each other in an intuitive location saves time. A descriptive folder structure makes a world of difference to you and the people who come after you.

## 4.3. 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.

***Explanation***

* Spend less time hunting and pecking for code, and become more efficient. Longer file names are far better than short-but-obscureabbreviated names.
* It may be advantageous to deviate from the one-thing-per-file rule when you have a set of small, closely-related features that are better discovered and understood in a single file than as multiple files. Be wary of this loophole.

## 4.4. 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.

***Explanation***

* No one wants to search for a file through seven levels of folders. A flat structure is easy to scan.
* On the other hand, [psychologists believe](https://en.wikipedia.org/wiki/The_Magical_Number_Seven,_Plus_or_Minus_Two) that humans start to struggle when the number of adjacent interesting things exceeds nine. So when a folder has ten or more files, it may be time to create subfolders.
* Base your decision on your comfort level. Use a flatter structure until there is an obvious value to creating a new folder.

## 4.5. T-DRY (Try to be DRY)

* **Do** be DRY (Don't Repeat Yourself).
* **Avoid** being so DRY that you sacrifice readability.

***Explanation***

* 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.

## 4.6. Overall structural guidelines

* **Do** start small but keep in mind where the app is heading down the road.
* **Do** have a near term view of implementation and a long term vision.
* **Consider** creating a folder for a component when it has multiple accompanying files (.ts, .html, .css and .spec).

***Explanation***

* Helps keep the app structure small and easy to maintain in the early stages, while being easy to evolve as the app grows.
* Components often have four files (e.g. \*.html, \*.css, \*.ts, and \*.spec.ts) and can clutter a folder quickly.
* Here is a compliant folder and file structure:

<< Project Name >>cameo

         asssets

o   inv-core-themes.css

         distribution

         languages

o   i18n

  en.json

  ta.json

         modules

o   root-module

o   your-module-1

  comp-1

         comp-1.ts

         comp-1.html

         comp-1.css

  comp-2

         comp-2.ts

         comp2-html

         comp2.css

  comp-n

         comp-n.ts

         comp-n.html

         comp-m.css

  tsconfig.json

o   your-module-2

  comp-1

  comp-n

  tsconfig.json

         node\_modules

         packages

o   node\_modules

o   bower\_components

         services

o   models

         shared-components

         index.html

         main.ts

         package.json

         systemjs.config.js

         tsconfig.json

While components in dedicated folders are widely preferred, another option for small apps is to keep components flat (not in a dedicated folder). This adds up to four files to the existing folder, but also reduces the folder nesting. Whatever you choose be consistent.

## 4.7. Folders-by-feature structure

* **Do** create folders named for the feature area they represent.
* A developer can locate the code and identify what each of the file represents at a glance. The structure is as flat as it can be and there are no repetitive or redundant names.
* The LIFT guidelines are all covered.
* Helps reduce the app from becoming cluttered through organizing the content and keeping them aligned with the LIFT guidelines.
* 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.
* **Do** create an Angular module for each feature area.
* Angular modules make it easy to lazy load routable features.
* Angular modules make it easier to isolate, test, and re-use features.

## 4.8. App root module

* **Do** create an Angular module in the app's root folder.

***Explanation***

* Every app requires at least one root Angular module.
* **Consider** naming the root module root.module.ts.
* Makes it easier to locate and identify the root module.

*import { NgModule } from '@angular/core';*

*import { BrowserModule } from '@angular/platform-browser';*

*import { AppComponent } from './app.component';*

*@NgModule({*

*imports: [*

*BrowserModule,*

*],*

*declarations: [*

*AppComponent,*

*],*

*exports: [ AppComponent ],*

*})*

*export class RootModule {}*

## 4.9. 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

***Explanation***

* A feature module can expose or hide its implementation from other modules.
* A feature module identifies distinct sets of related components that comprise the feature area.
* A feature module can easily be routed to both eagerly and lazily.
* A feature module defines clear boundaries between specific functionality and other application features.
* A feature module helps clarify and make it easier to assign development responsibilities to different teams.
* A feature module can easily be isolated for testing.

## 4.10. 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.

***Explanation***

* 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.
* SharedModule exists to make commonly used components, directives and pipes available for use in the templates of components in many other modules.
* **Avoid** specifying app-wide singleton providers in a SharedModule. Intentional singletons are OK. Take care.
* A lazy loaded feature module that imports that shared module will make its own copy of the service and likely have undesirable results.
* You don't want each module to have its own separate instance of singleton services. Yet there is a real danger of that happening if the SharedModule provides a service.

*import { NgModule } from '@angular/core';*

*import { CommonModule } from '@angular/common';*

*import { FormsModule } from '@angular/forms';*

*import { FilterTextComponent } from './filter-text/filter-text.component';*

*import { FilterTextService } from './filter-text/filter-text.service';*

*import { InitCapsPipe } from './init-caps.pipe';*

*@NgModule({*

*imports: [CommonModule, FormsModule],*

*declarations: [*

*FilterTextComponent,*

*InitCapsPipe*

*],*

*providers: [FilterTextService],*

*exports: [*

*CommonModule,*

*FormsModule,*

*FilterTextComponent,*

*InitCapsPipe*

*]*

*})*

*export class SharedModule { }*

## 4.11. 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).
* **Avoid** importing the CoreModule anywhere except in the AppModule.
* **Do** export all symbols from the CoreModule that the AppModule will import and make available for other feature modules to use.

## 4.12. Prevent re-import of the core module

* Only the root AppModule should import the CoreModule.
* **Do** guard against reimporting of CoreModule and fail fast by adding guard logic.
* Guards against reimporting of the CoreModule.
* Guards against creating multiple instances of assets intended to be singletons.

## 4.13. 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.

***Explanation***

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

## 4.14. Never directly import lazy loaded folders

* **Avoid** allowing modules in sibling and parent folders to directly import a module in a lazy loaded feature.

*Explanation*

* Directly importing and using a module will load it immediately when the intention is to load it on demand.

# 5. Components

## 5.1 Component selector names

* **Do** use dashed-case or kebab-case for naming the element selectors of components.
* Keeps the element names consistent with the specification for [Custom Elements](https://www.w3.org/TR/custom-elements/).

*/\* avoid \*/*

*@Component({*

*selector: 'tohHeroButton',*

*templateUrl: './hero-button.component.html'*

*})*

*export class HeroButtonComponent {}*

*/\* correct code \*/*

*@Component({*

*selector: 'toh-hero-button',*

*templateUrl: './hero-button.component.html'*

*})*

*export class HeroButtonComponent {}*

## 5.2. Components as elements

* **Do** give components an element selector, as opposed to attribute or class selectors.

***Explanation***

* 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.

*/\* avoid \*/*

*@Component({*

*selector: '[tohHeroButton]',*

*templateUrl: './hero-button.component.html'*

*})*

*export class HeroButtonComponent {}*

*/\* correct code \*/*

*@Component({*

*selector: 'toh-hero-button',*

*templateUrl: './hero-button.component.html'*

})

export class HeroButtonComponent {}

## 5.3. 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 ./.

***Explanation***

* 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.
* A component relative URL requires no change when you move the component files, as long as the files stay together.
* The ./ prefix is standard syntax for relative URLs; don't depend on Angular's current ability to do without that prefix.

*/\* avoid \*/*

*@Component({*

*selector: 'toh-heroes',*

*template: `*

*<div>*

*<h2>My Heroes</h2>*

*<ul class="heroes">*

*<li \*ngFor="let hero of heroes | async"(click)="selectedHero=hero">*

*`,*

*styles: [`*

*.heroes {*

*margin: 0 0 2em 0; list-style-type: none; padding: 0; width: 15em;*

*}*

*.heroes li {*

*cursor: pointer;*

*position: relative;*

*left: 0;*

*background-color: #EEE;*

*margin: .5em;*

*padding: .3em 0;*

*height: 1.6em;*

*border-radius: 4px;*

*}*

*`]*

*})*

*export class HeroesComponent implements OnInit {}*

*/\* correct code \*/*

*@Component({*

*selector: 'toh-heroes',*

*templateUrl: './heroes.component.html',*

*styleUrls: ['./heroes.component.css']*

*})*

*export class HeroesComponent implements OnInit {}*

## 5.4. Decorate input and output properties

* **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.

***Explanation***

* 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 \*/*

*@Component({*

*selector: 'toh-hero-button',*

*template: `<button></button>`,*

*inputs: [*

*'label'*

*],*

*outputs: [*

*'change'*

*]*

*})*

*export class HeroButtonComponent {*

*change = new EventEmitter<any>();*

*label: string;*

*}*

*/\* correct code \*/*

*@Component({*

*selector: 'toh-hero-button',*

*template: `<button>{{label}}</button>`*

*})*

*export class HeroButtonComponent {*

*@Output() change = new EventEmitter<any>();*

*@Input() label: string;*

*}*

## 5.5. Avoid aliasing inputs and outputs

* **Avoid** input and output aliases except when it serves an important purpose.

**E*xplanation***

* 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.

*/\* avoid pointless aliasing \*/*

*@Component({*

*selector: 'toh-hero-button',*

*template: `<button>{{label}}</button>`*

*})*

*export class HeroButtonComponent {*

*// Pointless aliases*

*@Output('changeEvent') change = new EventEmitter<any>();*

*@Input('labelAttribute') label: string;*

*}app/app.component.html*

*<!-- avoid -->*

*<toh-hero-button labelAttribute="OK" (changeEvent)="doSomething()">*

*</toh-hero-button>*

*<!—correct code -->*

*@Component({*

*selector: 'toh-hero-button',*

*template: `<button>{{label}}</button>`*

*})*

*export class HeroButtonComponent {*

*// No aliases*

*@Output() change = new EventEmitter<any>();*

*@Input() label: string;*

*}*

## 5.6. Member sequence

* **Do** place properties up top followed by methods.
* **Do** place private members after public members, alphabetized.

***Explanation***

* Placing members in a consistent sequence makes it easy to read and helps instantly identify which members of the component serve which purpose.

*/\* avoid \*/*

*export class ToastComponent implements OnInit {*

*private defaults = {*

*title: '',*

*message: 'May the Force be with you'*

*};*

*message: string;*

*title: string;*

*private toastElement: any;*

*// private methods*

*private hide() {*

*this.toastElement.style.opacity = 0;*

*window.setTimeout(() => this.toastElement.style.zIndex = 0, 400);*

*}*

*}*

*/\* correct code \*/*

*export class ToastComponent implements OnInit {*

*// public properties message: string;*

*title: string*

*// private fields*

*private defaults = {*

*title: '',*

*message: 'May the Force be with you'*

*};*

*private toastElement: any;*

*// public methods*

*activate(message = this.defaults.message, title = this.defaults.title) {*

*this.title = title;*

*this.message = message;*

*this.show();*

*}}*

## 5.7. 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.

***Explanation***

* Logic may be reused by multiple components when placed within a service and exposed via a function.
* Logic in a service can more easily be isolated in a unit test, while the calling logic in the component can be easily mocked.
* Removes dependencies and hides implementation details from the component.
* Keeps the component slim, trim, and focused.

*/\* avoid \*/*

*import { Http, Response } from '@angular/http';*

*import { Observable } from 'rxjs/Observable';*

*import 'rxjs/add/operator/catch';*

*import 'rxjs/add/operator/finally';*

*import 'rxjs/add/operator/map';*

*import { Hero } from '../shared/hero.model';*

*const heroesUrl = 'http://angular.io';*

*export class HeroListComponent implements OnInit {*

*heroes: Hero[];*

*constructor(private http: Http) {}*

*getHeroes() {*

*this.heroes = [];*

*this.http.get(heroesUrl)*

*.map((response: Response) => <Hero[]>response.json().data)*

*.catch(this.catchBadResponse)*

*.finally(() => this.hideSpinner())*

*.subscribe((heroes: Hero[]) => this.heroes = heroes);*

*}*

*}*

*/\* correct \*/*

*import { Component, OnInit } from '@angular/core';*

*import { Hero, HeroService } from '../shared';*

*export class HeroListComponent implements OnInit {*

*heroes: Hero[];*

*constructor(private heroService: HeroService) {}*

*getHeroes() {*

*this.heroes = [];*

*this.heroService.getHeroes()*

*.subscribe(heroes => this.heroes = heroes);*

*}}*

## 5.8. 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.

***Explanation***

* This is consistent with built-in events such as button clicks.
* Angular allows for an [alternative syntax](https://angular.io/template-syntax.html#binding-syntax) on-\*. If the event itself was prefixed with on this would result in an on-onEventbinding expression.

*/\* avoid \*/*

*export class HeroComponent {*

*@Output() onSavedTheDay = new EventEmitter<boolean>();*

*}*

*/\* avoid \*/*

*<toh-hero (onSavedTheDay)="onSavedTheDay($event)"></toh-hero>*

*/\* correct code \*/*

*export class HeroComponent {*

*@Output() savedTheDay = new EventEmitter<boolean>();*

*}*

## 5.9. Put presentation logic in the component class

* **Do** put presentation logic in the component class, and not in the template.

***Explanation***

* Logic will be contained in one place (the component class) instead of being spread in two places.
* Keeping the component's presentation logic in the class instead of the template improves testability, maintainability, and reusability.

*/\* avoid \*/*

*@Component({*

*selector: 'toh-hero-list',*

*template: `*

*<section>*

*Our list of heroes:*

*<hero-profile \*ngFor="let hero of heroes" [hero]="hero">*

*</hero-profile>*

*Total powers: {{totalPowers}}<br>*

*Average power: {{totalPowers / heroes.length}}*

*</section>*

*`*

*})*

*export class HeroListComponent {*

*heroes: Hero[];*

*totalPowers: number;*

*}*

*/\* correct code \*/*

*@Component({*

*selector: 'toh-hero-list',*

*template: `*

*<section>*

*Our list of heroes:*

*<toh-hero \*ngFor="let hero of heroes" [hero]="hero">*

*</toh-hero>*

*Total powers: {{totalPowers}}<br>*

*Average power: {{avgPower}}*

*</section>*

*`*

*})*

*export class HeroListComponent {*

*heroes: Hero[];*

*totalPowers: number;*

*get avgPower() {*

*return this.totalPowers / this.heroes.length;*

*}*

*}*

# 6. Directives

## 6.1. Use directives to enhance an element

* **Do** use attribute directives when you have presentation logic without a template.

***Explanation***

* Attribute directives don't have an associated template.
* An element may have more than one attribute directive applied.

*@Directive({*

*selector: '[tohHighlight]'*

*})*

*export class HighlightDirective {*

*@HostListener('mouseover') onMouseEnter() {*

*// do highlight work*

*}*

*}*

*<div tohHighlight>Bombasta</div>*

## 6.2. 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.

***Explanation***

* The property associated with @HostBinding or the method associated with @HostListener can be modified only in a single place—in the directive's class. If you use the host metadata property, you must modify both the property declaration inside the controller, and the metadata associated with the directive.

*import { Directive, HostBinding, HostListener } from '@angular/core';*

*@Directive({*

*selector: '[tohValidator]'*

*})*

*export class ValidatorDirective {*

*@HostBinding('attr.role') role = 'button';*

*@HostListener('mouseenter') onMouseEnter() {*

*// do work*

*}*

*}*

Compare with the less preferred host metadata alternative.

The host metadata is only one term to remember and doesn't require extra ES imports.

p/shared/validator2.directive.ts

*import { Directive } from '@angular/core';*

*@Directive({*

*selector: '[tohValidator2]',*

*host: {*

*'attr.role': 'button',*

*'(mouseenter)': 'onMouseEnter()'*

*}*

*})*

*export class Validator2Directive {*

*role = 'button';*

*onMouseEnter() { //do work }*

*}*

# 7. Services

## 7.1. Services are singletons

* **Do** use services as singletons within the same injector. Use them for sharing data and functionality.

***Explanation***

* Services are ideal for sharing methods across a feature area or an app.
* Services are ideal for sharing stateful in-memory data.

*export class HeroService {*

*constructor(private http: Http) { }*

*getHeroes() {*

*return this.http.get('api/heroes')*

*.map((response: Response) => <Hero[]>response.json().data);*

*}*

*}*

## 7.2. Single responsibility

* **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.

***Explanation***

* When a service has multiple responsibilities, it becomes difficult to test.
* When a service has multiple responsibilities, every component or service that injects it now carries the weight of them all.

## 7.3. Providing a service

* **Do** provide services to the Angular injector at the top-most component where they will be shared.
* The Angular injector is hierarchical.

*Explanation*

* 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.
* 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.

*import { Component } from '@angular/core';*

*import { HeroService } from './heroes';*

*@Component({*

*selector: 'toh-app',*

*template: `*

*<toh-heroes></toh-heroes>`,*

*providers: [HeroService]*

*})*

*export class AppComponent {}*

## 7.4. 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.

*Explanation*

* 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.

*/\* avoid \*/*

*export class HeroArena {*

*constructor(*

*@Inject(HeroService) private heroService: HeroService,*

*}*

*/\* correct code \*/*

*@Injectable()*

*export class HeroArena {*

*constructor(*

*private heroService: HeroService,*

*}*

# 8. Data Services

## 8.1. 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.

*Explanation*

* The component's responsibility is for the presentation and gathering of information for the view. It should not care how it gets the data, just that it knows who to ask for it. Separating the data services moves the logic on how to get it to the data service, and lets the component be simpler and more focused on the view.
* This makes it easier to test (mock or real) the data calls when testing a component that uses a data service.
* The details of data management, such as headers, HTTP methods, caching, error handling, and retry logic, are irrelevant to components and other data consumers.
* A data service encapsulates these details. It's easier to evolve these details inside the service without affecting its consumers. And it's easier to test the consumers with mock service implementations.

# 9. Lifecycle hooks

* Use Lifecycle hooks to tap into important events exposed by Angular.

## 9.1. Implement lifecycle hook interfaces

* **Do** implement the lifecycle hook interfaces.

***Explanation***

* Lifecycle interfaces prescribe typed method signatures. use those signatures to flag spelling and syntax mistakes.

*/\* avoid \*/*

*export class HeroButtonComponent {*

*onInit() { // misspelled*

*console.log('The component is initialized');*

*}*

*}*

*/\* correct code \*/*

*export class HeroButtonComponent implements OnInit {*

*ngOnInit() {*

*console.log('The component is initialized');*

*}*

*}*

# 10. Best Practices

* **Do** use *Var* cautiously as it hoists the variable at the top of the enclosing function block
* **Do** use *Let* instead of *Var*
* **Do** use *Any* cautiously as it defeats the very purpose of static type. Use it only when the type of variable is unknown
* **Avoid** using *DOMSanitizer* as it opens the security risk
* **Do** map the service result (service data contract) to your local object that is bound to view. This reduces the effort needed when the data contract changes