|  |  |
| --- | --- |
| ANGULAR- Style Guide | DeliverEat  2019 |

Abril 2019

Tabla de contenido

[Introduction 3](#_Toc9347812)

[Style vocabulary 4](#_Toc9347813)

[File structure conventions 4](#_Toc9347814)

[Single responsibility 5](#_Toc9347815)

[Rule of One- Style 01-01 5](#_Toc9347816)

[Small functions- Style 01-02 10](#_Toc9347817)

[Naming 11](#_Toc9347818)

[General Naming Guidelines- Style 02-01 11](#_Toc9347819)

[Separate file names with dots and dashes- Style 02-02 11](#_Toc9347820)

[Symbols and file names- Style 02-03 12](#_Toc9347821)

[Service names- Style 02-04 14](#_Toc9347822)

[Bootstrapping- Style 02-05 15](#_Toc9347823)

[Component selectors- Style 05-02 15](#_Toc9347824)

[Component custom prefix- Style 02-07 17](#_Toc9347825)

[Directive selectors- Style 02-06 18](#_Toc9347826)

[Directive custom prefix- Style 02-08 19](#_Toc9347827)

[Pipe names- Style 02-09 20](#_Toc9347828)

[Unit test file names- Style 02-10 20](#_Toc9347829)

[*End-to-End* (E2E) test file names- Style 02-11 21](#_Toc9347830)

[Angular *NgModule* names- Style 02-12 22](#_Toc9347831)

[Coding conventions 24](#_Toc9347832)

[Classes- Style 03-01 24](#_Toc9347833)

[Constants- Style 03-02 25](#_Toc9347834)

[Interfaces- Style 03-03 26](#_Toc9347835)

[Properties and methods- Style 03-04 27](#_Toc9347836)

[Import line spacing- Style 03-06 29](#_Toc9347837)

[Application structure and NgModules 31](#_Toc9347838)

[*LIFT*- Style 04-01 31](#_Toc9347839)

[Locate- Style 04-02 31](#_Toc9347840)

[Identify-Style 04-03 32](#_Toc9347841)

[Flat- Style 04-04 32](#_Toc9347842)

[*T-DRY* (Try to be *DRY*)- Style 04-05 33](#_Toc9347843)

[Overall structural guidelines- Style 04-06 33](#_Toc9347844)

[Folders-by-feature structure- Style 04-07 36](#_Toc9347845)

[App *root module*- Style 04-08 36](#_Toc9347846)

[Feature modules- Style 04-09 37](#_Toc9347847)

[Shared feature module- Style 04-10 38](#_Toc9347848)

[Lazy Loaded folders- Style 04-11 43](#_Toc9347849)

[Never directly import lazy loaded folders- Style 04-12 43](#_Toc9347850)

[Components 44](#_Toc9347851)

[Components as elements- Style 05-03 44](#_Toc9347852)

[Extract templates and styles to their own files- Style 05-04 45](#_Toc9347853)

[Decorate *input* and *output* properties- Style 05-12 50](#_Toc9347854)

[Avoid aliasing *inputs* and *outputs*- Style 05-13 51](#_Toc9347855)

[Member sequence- Style 05-14 53](#_Toc9347856)

[Delegate complex component logic to services- Style 05-15 56](#_Toc9347857)

[Don't prefix *output* properties- Style 05-16 58](#_Toc9347858)

[Put presentation logic in the component class- Style 05-17 59](#_Toc9347859)

[Directives 62](#_Toc9347860)

[Use directives to enhance an element- Style 06-01 62](#_Toc9347861)

[*HostListener*/*HostBinding* decorators versus *host* metadata- Style 06-03 63](#_Toc9347862)

[Services 65](#_Toc9347863)

[Services are singletons- Style 07-01 65](#_Toc9347864)

[Single responsibility- Style 07-02 65](#_Toc9347865)

[Providing a service-Style 07-03 66](#_Toc9347866)

[Use the @Injectable() class decorator- Style 07-04 67](#_Toc9347867)

[Data Services 68](#_Toc9347868)

[Talk to the server through a service- Style 08-01 68](#_Toc9347869)

[Lifecycle hooks 68](#_Toc9347870)

[Implement lifecycle hook interfaces- Style 09-01 68](#_Toc9347871)

[Appendix 70](#_Toc9347872)

[Codelyzer- Style A-01 70](#_Toc9347873)

[File templates and snippets- Style A-02 70](#_Toc9347874)

# Introduction

Looking for an opinionated guide to Angular syntax, conventions, and application structure? Step right in! This style guide presents preferred conventions and, as importantly, explains why.

# Style vocabulary

Each guideline describes either a good or bad practice, and all have a consistent presentation.

The wording of each guideline indicates how strong the recommendation is.

Do is one that should always be followed. *Always* might be a bit too strong of a word. Guidelines that literally should always be followed are extremely rare. On the other hand, you need a really unusual case for breaking a *Do* guideline.

Consider guidelines should generally be followed. If you fully understand the meaning behind the guideline and have a good reason to deviate, then do so. Please strive to be consistent.

Avoid indicates something you should almost never do. Code examples to *avoid* have an unmistakable red header.

Why? gives reasons for following the previous recommendations.

# File structure conventions

Some code examples display a file that has one or more similarly named companion files. For example*, hero.component.ts* and *hero.component.html.*

The guideline uses the shortcut *hero.component.ts|html|css|spec* to represent those various files. Using this shortcut makes this guide's file structures easier to read and more terse.

# 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.

## Rule of One- Style 01-01

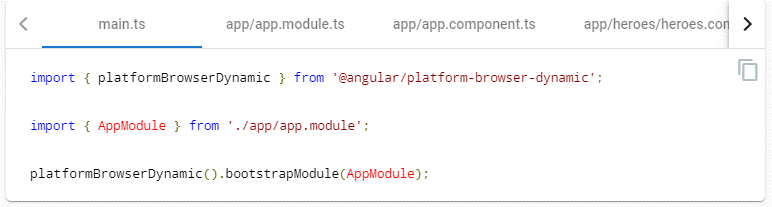
* **Do** define one thing, such as a service or component, per file.
* **Consider** limiting files to 400 lines of code.
* **Why**? One component per file makes it far easier to read, maintain, and avoid collisions with teams in source control.
* **Why**? One component per file avoids hidden bugs that 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.
* The following negative example defines the AppComponent, bootstraps the app, defines the Hero model object, and loads heroes from the server all in the same file. Don't do this.

app/heroes/hero.component.ts

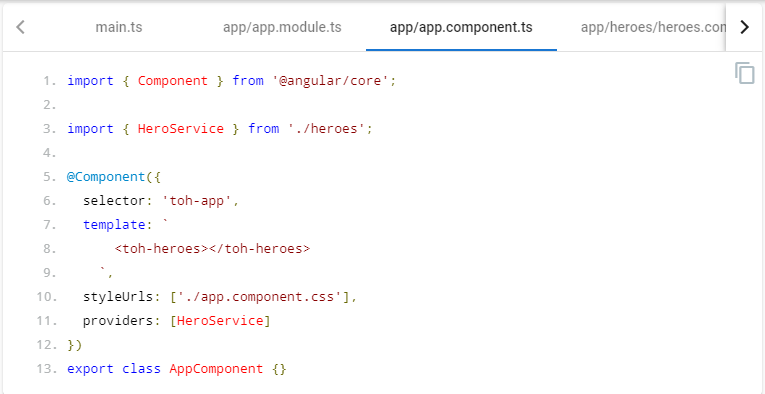
content\_copy

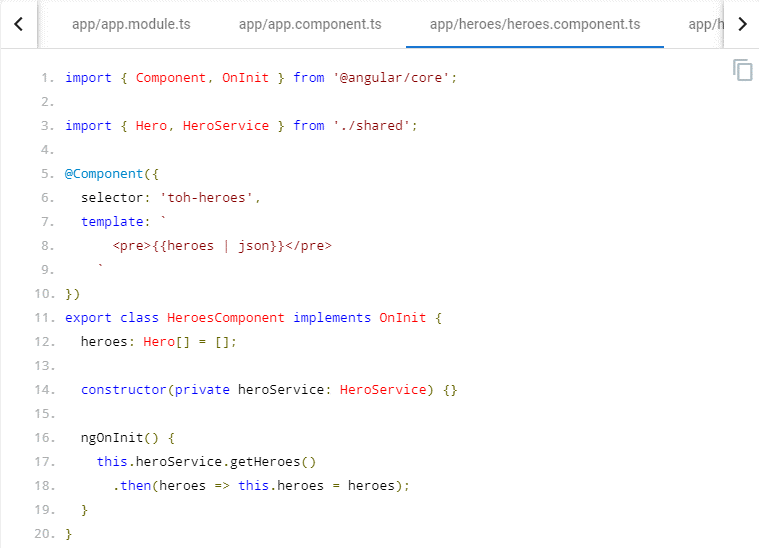
1. /\* avoid \*/
2. import { [Component](https://angular.io/api/core/Component), [NgModule](https://angular.io/api/core/NgModule), [OnInit](https://angular.io/api/core/OnInit) } from '@angular/core';
3. import { [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) } from '@angular/platform-browser';
4. import { [platformBrowserDynamic](https://angular.io/api/platform-browser-dynamic/platformBrowserDynamic) } from '@angular/platform-browser-dynamic';
6. class Hero {
7. id: number;
8. name: string;
9. }
11. @[Component](https://angular.io/api/core/Component)({
12. selector: 'my-app',
13. template: `
14. <h1>{{title}}</h1>
15. <pre>{{heroes | [json](https://angular.io/api/common/JsonPipe)}}</pre>
16. `,
17. [styleUrls](https://angular.io/api/core/Component#styleUrls): ['app/app.component.css']
18. })
19. class AppComponent implements [OnInit](https://angular.io/api/core/OnInit) {
20. title = 'Tour of Heroes';
22. heroes: Hero[] = [];
24. ngOnInit() {
25. getHeroes().then(heroes => (this.heroes = heroes));
26. }
27. }
29. @[NgModule](https://angular.io/api/core/NgModule)({
30. imports: [[BrowserModule](https://angular.io/api/platform-browser/BrowserModule)],
31. declarations: [AppComponent],
32. exports: [AppComponent],
33. bootstrap: [AppComponent]
34. })
35. export class AppModule {}
37. [platformBrowserDynamic](https://angular.io/api/platform-browser-dynamic/platformBrowserDynamic)().bootstrapModule(AppModule);
39. const HEROES: Hero[] = [
40. { id: 1, name: 'Bombasto' },
41. { id: 2, name: 'Tornado' },
42. { id: 3, name: 'Magneta' }
43. ];
45. function getHeroes(): Promise<Hero[]> {
46. return Promise.resolve(HEROES); // TODO: get hero data from the server;
47. }

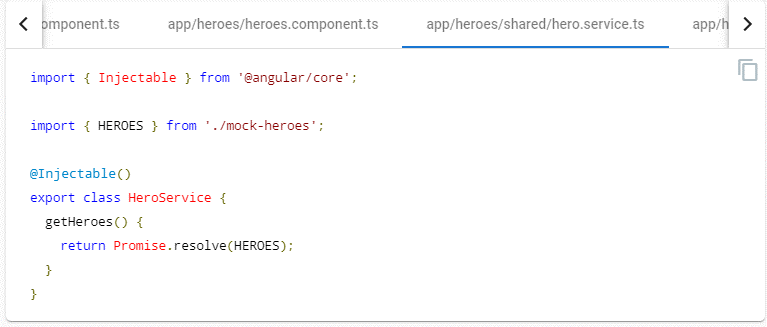
It is a better practice to redistribute the component and its supporting classes into their own, dedicated files.

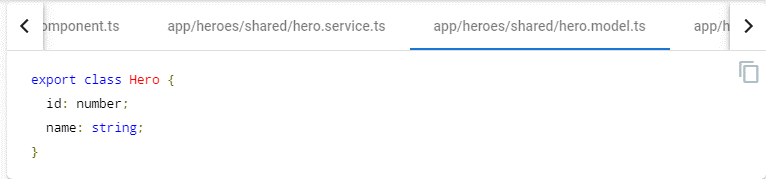


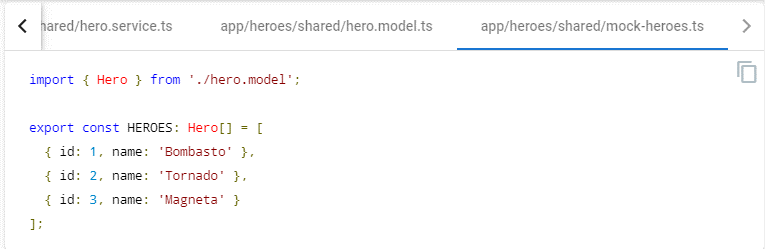












As the app grows, this rule becomes even more important.

## Small functions- 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.

# Naming

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

## General Naming Guidelines- Style 02-01

* 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.
* 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.

## Separate file names with dots and dashes- 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.

## Symbols and file names- Style 02-03

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

|  |  |
| --- | --- |
| **Symbol Name** | **File Name** |
| content\_copy@[Component](https://angular.io/api/core/Component)({ ... })  export class AppComponent { } | app.component.ts |
| content\_copy@[Component](https://angular.io/api/core/Component)({ ... })  export class HeroesComponent { } | heroes.component.ts |
| content\_copy@[Component](https://angular.io/api/core/Component)({ ... })  export class HeroListComponent { } | hero-list.component.ts |
| content\_copy@[Component](https://angular.io/api/core/Component)({ ... })  export class HeroDetailComponent { } | hero-detail.component.ts |
| content\_copy@[Directive](https://angular.io/api/core/Directive)({ ... })  export class ValidationDirective { } | validation.directive.ts |
| content\_copy@[NgModule](https://angular.io/api/core/NgModule)({ ... })  export class AppModule | app.module.ts |
| content\_copy@[Pipe](https://angular.io/api/core/Pipe)({ name: 'initCaps' })  export class InitCapsPipe implements [PipeTransform](https://angular.io/api/core/PipeTransform) { } | init-caps.pipe.ts |
| content\_copy@[Injectable](https://angular.io/api/core/Injectable)()  export class UserProfileService { } | user-profile.service.ts |

## Service names- Style 02-04

* 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.
* A few terms are unambiguously services. They typically indicate agency by ending in "-er". You may prefer to name a service that logs messages Logger rather than LoggerService. Decide if this exception is agreeable in your project. As always, strive for consistency.
* Why? Provides a consistent way to quickly identify and reference services.
* Why? Clear service names such as Logger do not require a suffix.
* Why? 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** |
| content\_copy@[Injectable](https://angular.io/api/core/Injectable)()  export class HeroDataService { } | hero-data.service.ts |
| content\_copy@[Injectable](https://angular.io/api/core/Injectable)()  export class CreditService { } | credit.service.ts |
| content\_copy@[Injectable](https://angular.io/api/core/Injectable)()  export class Logger { } | logger.service.ts |

## Bootstrapping- Style 02-05

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

main.ts

content\_copyimport { [platformBrowserDynamic](https://angular.io/api/platform-browser-dynamic/platformBrowserDynamic) } from '@angular/platform-browser-dynamic';

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

[platformBrowserDynamic](https://angular.io/api/platform-browser-dynamic/platformBrowserDynamic)().bootstrapModule(AppModule)

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

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

## Component selectors- Style 05-02

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

app/heroes/shared/hero-button/hero-button.component.ts

content\_copy/\* avoid \*/

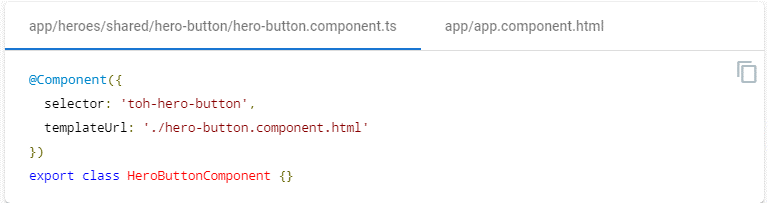
@[Component](https://angular.io/api/core/Component)({

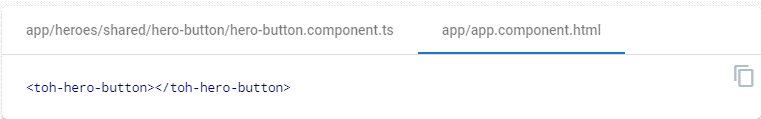
selector: 'tohHeroButton',

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

})

export class HeroButtonComponent {}





## Component custom prefix- Style 02-07

* Do use a hyphenated, lowercase element selector value; for example, admin-users.
* Do use a custom prefix for a component selector. For example, the prefix toh represents from Tour of Heroes and the prefix admin represents an admin feature area.
* Do use a prefix that identifies the feature area or the app itself.
* Why? Prevents element name collisions with components in other apps and with native HTML elements.
* Why? Makes it easier to promote and share the component in other apps.
* Why? Components are easy to identify in the DOM.

app/heroes/hero.component.ts

content\_copy/\* avoid \*/

// HeroComponent is in the Tour of Heroes feature

@[Component](https://angular.io/api/core/Component)({

selector: 'hero'

})

export class HeroComponent {}

app/users/users.component.ts

content\_copy/\* avoid \*/

// UsersComponent is in an Admin feature

@[Component](https://angular.io/api/core/Component)({

selector: 'users'

})

export class UsersComponent {}

app/heroes/hero.component.ts

content\_copy@[Component](https://angular.io/api/core/Component)({

selector: 'toh-hero'

})

export class HeroComponent {}

app/users/users.component.ts

content\_copy@[Component](https://angular.io/api/core/Component)({

selector: 'admin-users'

})

export class UsersComponent {}

## Directive selectors- Style 02-06

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

## Directive custom prefix- Style 02-08

* **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.
* **Why?** Prevents name collisions.
* **Why?** Directives are easily identified.

app/shared/validate.directive.ts

content\_copy/\* avoid \*/

@[Directive](https://angular.io/api/core/Directive)({

selector: '[validate]'

})

export class ValidateDirective {}

app/shared/validate.directive.ts

content\_copy@[Directive](https://angular.io/api/core/Directive)({

selector: '[tohValidate]'

})

export class ValidateDirective {}

## Pipe names- Style 02-09

* **Do** use consistent names for all pipes, named after their feature.
* **Why?** Provides a consistent way to quickly identify and reference pipes.

|  |  |
| --- | --- |
| **Symbol Name** | **File Name** |
| content\_copy@[Pipe](https://angular.io/api/core/Pipe)({ name: 'ellipsis' })  export class EllipsisPipe implements [PipeTransform](https://angular.io/api/core/PipeTransform) { } | ellipsis.pipe.ts |
| content\_copy@[Pipe](https://angular.io/api/core/Pipe)({ name: 'initCaps' })  export class InitCapsPipe implements [PipeTransform](https://angular.io/api/core/PipeTransform) { } | init-caps.pipe.ts |

## Unit test file names- Style 02-10

* **Do** name test specification files the same as the component they test.
* **Do** name test specification files with a suffix of .spec.
* **Why?** Provides a consistent way to quickly identify tests.
* **Why?** Provides pattern matching for [karma](http://karma-runner.github.io/) or other test runners.

|  |  |
| --- | --- |
| **Test Type** | **File Names** |
| Components | heroes.component.spec.ts  hero-list.component.spec.ts  hero-detail.component.spec.ts |
| Services | logger.service.spec.ts  hero.service.spec.ts  filter-text.service.spec.ts |
| Pipes | ellipsis.pipe.spec.ts  init-caps.pipe.spec.ts |

## End-to-End (E2E) test file names- Style 02-11

* **Do** name end-to-end test specification files after the feature they test with a suffix of .e2e-spec.
* **Why?** Provides a consistent way to quickly identify end-to-end tests.
* **Why?** Provides pattern matching for test runners and build automation.

|  |  |
| --- | --- |
| **Test Type** | **File Names** |
| End-to-End Tests | app.e2e-spec.ts  heroes.e2e-spec.ts |

## Angular NgModule names- Style 02-12

* **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.
* **Why?** Provides a consistent way to quickly identify and reference modules.
* **Why?** Upper camel case is conventional for identifying objects that can be instantiated using a constructor.
* **Why?** 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.
* **Why?** 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** |
| content\_copy@[NgModule](https://angular.io/api/core/NgModule)({ ... })  export class AppModule { } | app.module.ts |
| content\_copy@[NgModule](https://angular.io/api/core/NgModule)({ ... })  export class HeroesModule { } | heroes.module.ts |
| content\_copy@[NgModule](https://angular.io/api/core/NgModule)({ ... })  export class VillainsModule { } | villains.module.ts |
| content\_copy@[NgModule](https://angular.io/api/core/NgModule)({ ... })  export class AppRoutingModule { } | app-routing.module.ts |
| content\_copy@[NgModule](https://angular.io/api/core/NgModule)({ ... })  export class HeroesRoutingModule { } | heroes-routing.module.ts |

# Coding conventions

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

## Classes- Style 03-01

* **Do** use upper camel case, also known as PascalCase, when naming classes.
* **Why?** Follows conventional thinking for class names.
* **Why?** Classes can be instantiated and construct an instance. By convention, upper camel case indicates a constructable asset.

app/shared/exception.service.ts

content\_copy/\* avoid \*/

export class exceptionService {

constructor() { }

}

app/shared/exception.service.ts

content\_copyexport class ExceptionService {

constructor() { }

}

## Constants- Style 03-02

* **Do** declare variables with const if their values should not change during the application lifetime.
* **Why?** Conveys to readers that the value is invariant.
* **Why?** TypeScript helps enforce that intent by requiring immediate initialization and by preventing subsequent re-assignment.
* **Consider** spelling const variables in lower camel case.
* **Why?** Lower camel case variable names (heroRoutes) are easier to read and understand than the traditional UPPER\_SNAKE\_CASE names (HERO\_ROUTES).
* **Why?** 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.
* **Why?** 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/data.service.ts

content\_copyexport const mockHeroes = ['Sam', 'Jill']; // prefer

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

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

## Interfaces- Style 03-03

* **Do** name an interface using upper camel case.
* **Consider** naming an interface without an I prefix.
* **Consider** using a class instead of an interface for services and declarables (components, directives, and pipes).
* **Consider** using an interface for data models.
* **Why?** [TypeScript guidelines](https://github.com/Microsoft/TypeScript/wiki/Coding-guidelines) discourage the I prefix.
* **Why?** A class alone is less code than a class-plus-interface.
* **Why?** A class can act as an interface (use implements instead of extends).
* **Why?** An interface-class can be a provider lookup token in Angular dependency injection.

app/shared/hero-collector.service.ts

content\_copy

1. /\* avoid \*/
3. import { [Injectable](https://angular.io/api/core/Injectable) } from '@angular/core';
5. import { IHero } from './hero.model.avoid';
7. @[Injectable](https://angular.io/api/core/Injectable)()
8. export class HeroCollectorService {
9. hero: IHero;
11. constructor() { }
12. }

app/shared/hero-collector.service.ts

content\_copyimport { [Injectable](https://angular.io/api/core/Injectable) } from '@angular/core';

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

@[Injectable](https://angular.io/api/core/Injectable)()

export class HeroCollectorService {

hero: Hero;

constructor() { }

}

## Properties and methods- Style 03-04

* **Do** use lower camel case to name properties and methods.
* **Avoid** prefixing private properties and methods with an underscore.
* **Why?** Follows conventional thinking for properties and methods.
* **Why?** JavaScript lacks a true private property or method.
* **Why?** TypeScript tooling makes it easy to identify private vs. public properties and methods.

app/shared/toast.service.ts

content\_copy

1. /\* avoid \*/
3. import { [Injectable](https://angular.io/api/core/Injectable) } from '@angular/core';
5. @[Injectable](https://angular.io/api/core/Injectable)()
6. export class ToastService {
7. [message](https://angular.io/api/common/http/HttpErrorResponse#message): string;
9. private \_toastCount: number;
11. hide() {
12. this.\_toastCount--;
13. this.\_log();
14. }
16. show() {
17. this.\_toastCount++;
18. this.\_log();
19. }
21. private \_log() {
22. console.log(this.message);
23. }
24. }

app/shared/toast.service.ts

content\_copy

1. import { [Injectable](https://angular.io/api/core/Injectable) } from '@angular/core';
3. @[Injectable](https://angular.io/api/core/Injectable)()
4. export class ToastService {
5. [message](https://angular.io/api/common/http/HttpErrorResponse#message): string;
7. private toastCount: number;
9. hide() {
10. this.toastCount--;
11. this.log();
12. }
14. show() {
15. this.toastCount++;
16. this.log();
17. }
19. private [log](https://angular.io/api/animations/browser/testing/MockAnimationDriver#log)() {
20. console.log(this.message);
21. }
22. }

## Import line spacing- Style 03-06

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

app/heroes/shared/hero.service.ts

content\_copy/\* avoid \*/

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

import { [HttpClient](https://angular.io/api/common/http/HttpClient) } from '@angular/common/[http](https://angular.io/api/common/http)';

import { [Injectable](https://angular.io/api/core/Injectable) } from '@angular/core';

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

app/heroes/shared/hero.service.ts

content\_copyimport { [HttpClient](https://angular.io/api/common/http/HttpClient) } from '@angular/common/[http](https://angular.io/api/common/http)';

import { [Injectable](https://angular.io/api/core/Injectable) } from '@angular/core';

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

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

# Application structure and NgModules

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 NgModule.

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.

## LIFT- Style 04-01

**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.

**Why?** 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?

## Locate- Style 04-02

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

**Why?** 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.

## Identify-Style 04-03

* **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.
* **Why?** Spend less time hunting and pecking for code, and become more efficient. Longer file names are far better than short-but-obscure abbreviated 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.

## Flat- Style 04-04

* **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.
* **Why?** 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.

## T-DRY (Try to be DRY)- Style 04-05

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

## Overall structural guidelines- Style 04-06

* **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.
* **Do** put all of the app's code in a folder named src.
* **Consider** creating a folder for a component when it has multiple accompanying files (.ts, .html, .css and .spec).
* **Why?** Helps keep the app structure small and easy to maintain in the early stages, while being easy to evolve as the app grows.
* **Why?** 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 root>

src

app

core

core.module.ts

exception.service.ts|spec.ts

user-profile.service.ts|spec.ts

heroes

hero

hero.component.ts|html|css|spec.ts

hero-list

hero-list.component.ts|html|css|spec.ts

shared

hero-button.component.ts|html|css|spec.ts

hero.model.ts

hero.service.ts|spec.ts

heroes.component.ts|html|css|spec.ts

heroes.module.ts

heroes-routing.module.ts

shared

shared.module.ts

init-caps.pipe.ts|spec.ts

text-filter.component.ts|spec.ts

text-filter.service.ts|spec.ts

villains

villain

...

villain-list

...

shared

...

villains.component.ts|html|css|spec.ts

villains.module.ts

villains-routing.module.ts

app.component.ts|html|css|spec.ts

app.module.ts

app-routing.module.ts

main.ts

index.html

...

node\_modules/...

...

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.

## Folders-by-feature structure- Style 04-07

* **Do** create folders named for the feature area they represent.
* **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.
* **Do** create an NgModule for each feature area.
* **Why?** NgModules make it easy to lazy load routable features.
* **Why?** NgModules make it easier to isolate, test, and reuse features.

## App root module- Style 04-08

* **Do** create an NgModule in the app's root folder, for example, in /src/app.
* **Why?** Every app requires at least one root NgModule.
* **Consider** naming the root module app.module.ts.
* **Why?** Makes it easier to locate and identify the root module.

app/app.module.ts

content\_copy

1. import { [NgModule](https://angular.io/api/core/NgModule) } from '@angular/core';
2. import { [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) } from '@angular/platform-browser';
4. import { AppComponent } from './app.component';
5. import { HeroesComponent } from './heroes/heroes.component';
7. @[NgModule](https://angular.io/api/core/NgModule)({
8. imports: [
9. [BrowserModule](https://angular.io/api/platform-browser/BrowserModule),
10. ],
11. declarations: [
12. AppComponent,
13. HeroesComponent
14. ],
15. exports: [ AppComponent ],
16. entryComponents: [ AppComponent ]
17. })
18. export class AppModule {}

## Feature modules- Style 04-09

* **Do** create an NgModule 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.
* **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.

## Shared feature module- Style 04-10

* **Do** create a feature module named SharedModule in a shared folder; for example, app/shared/shared.module.tsdefines 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.
* **Consider** not providing services in shared modules. Services are usually singletons that are provided once for the entire application or in a particular feature module. There are exceptions, however. For example, in the sample code that follows, notice that the SharedModule provides FilterTextService. This is acceptable here because the service is stateless;that is, the consumers of the service aren't impacted by new instances.
* **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).
* **Why?** SharedModule will contain components, directives and pipes that may need features from another common module; for example, [ngFor](https://angular.io/api/common/NgForOf) in [CommonModule](https://angular.io/api/common/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.
* **Why?** 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.
* **Why?** A lazy loaded feature module that imports that shared module will make its own copy of the service and likely have undesirable results.
* **Why?** 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.

src

app

shared

shared.module.ts

init-caps.pipe.ts|spec.ts

text-filter.component.ts|spec.ts

text-filter.service.ts|spec.ts

app.component.ts|html|css|spec.ts

app.module.ts

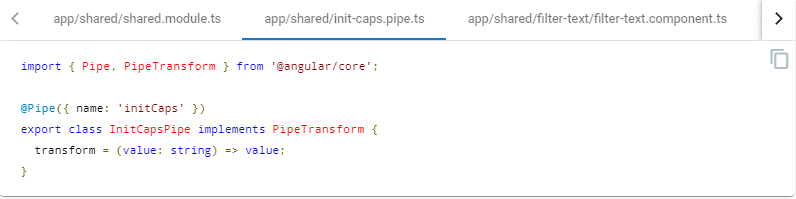
app-routing.module.ts

main.ts

index.html

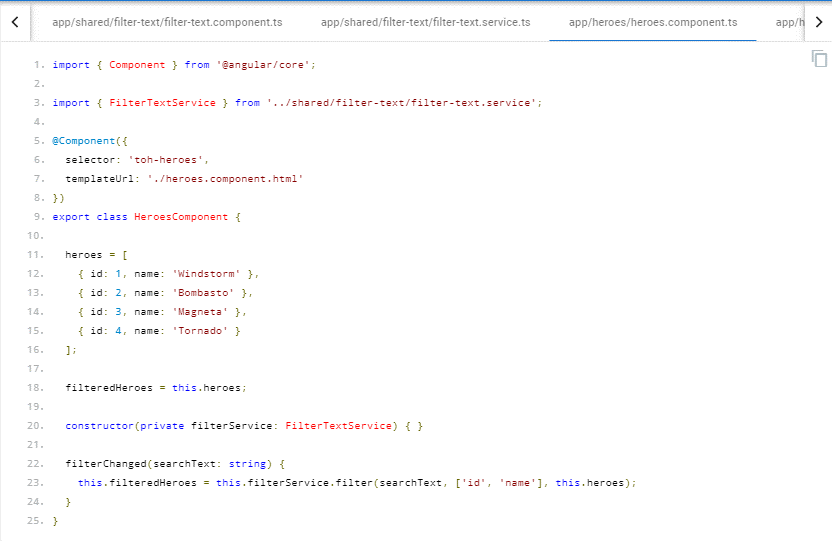
...

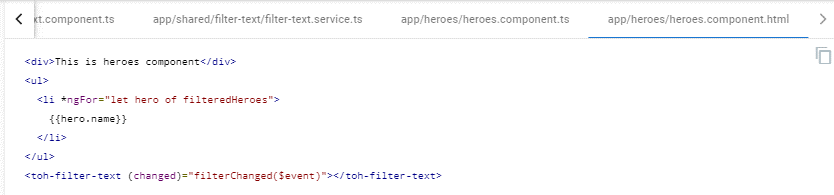












## Lazy Loaded folders- Style 04-11

* 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.

## Never directly import lazy loaded folders- Style 04-12

* **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.

# Components

## Components as elements- Style 05-03

* **Consider** giving components an element selector, as opposed to attribute or class selectors.
* **Why?** 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.
* **Why?** It is easier to recognize that a symbol is a component by looking at the template's html.

There are a few cases where you give a component an attribute, such as when you want to augment a built-in element. For example, [Material Design](https://material.angular.io/components/button/overview) uses this technique with <button mat-button>. However, you wouldn't use this technique on a custom element.

app/heroes/hero-button/hero-button.component.ts

content\_copy/\* avoid \*/

@[Component](https://angular.io/api/core/Component)({

selector: '[tohHeroButton]',

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

})

export class HeroButtonComponent {}

app/app.component.html

content\_copy<!-- avoid -->

<div tohHeroButton></div>

app/heroes/shared/hero-button/hero-button.component.ts

app/app.component.html

content\_copy@[Component](https://angular.io/api/core/Component)({

selector: 'toh-hero-button',

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

})

export class HeroButtonComponent {}

## Extract templates and styles to their own files- Style 05-04

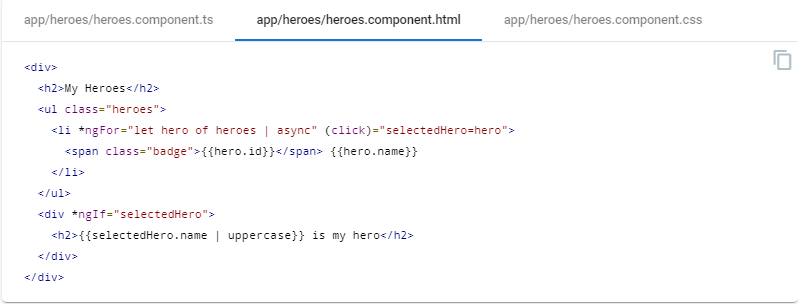
* **Do** extract templates and styles into a separate file, when more than 3 lines.
* **Do** name the template file [component-name].component.html, where [component-name] is the component name.
* **Do** name the style file [component-name].component.css, where [component-name] is the component name.
* **Do** specify component-relative URLs, prefixed with ./.
* **Why?** Large, inline templates and styles obscure the component's purpose and implementation, reducing readability and maintainability.
* **Why?** 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.
* **Why?** A component relative URL requires no change when you move the component files, as long as the files stay together.
* **Why?** The ./ prefix is standard syntax for relative URLs; don't depend on Angular's current ability to do without that prefix.

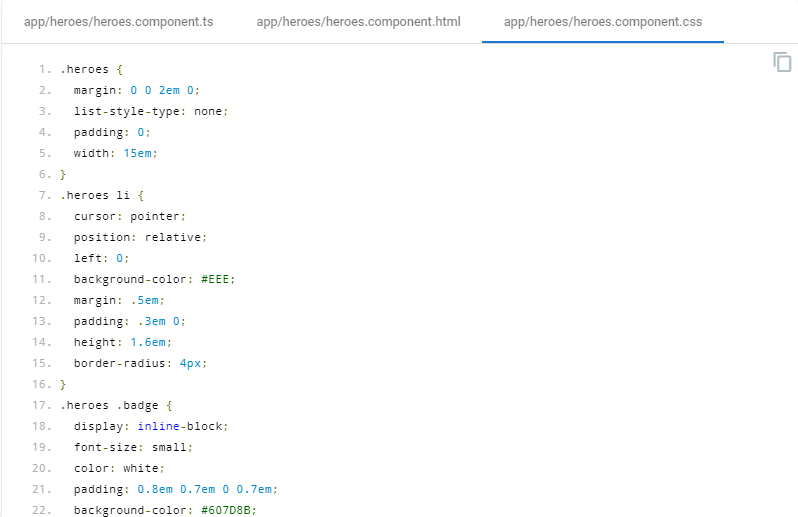
app/heroes/heroes.component.ts

content\_copy

1. /\* avoid \*/
3. @[Component](https://angular.io/api/core/Component)({
4. selector: 'toh-heroes',
5. template: `
6. <div>
7. <h2>My Heroes</h2>
8. <ul class="heroes">
9. <li \*[ngFor](https://angular.io/api/common/NgForOf)="let hero of heroes | async" (click)="selectedHero=hero">
10. <span class="badge">{{hero.id}}</span> {{hero.name}}
11. </li>
12. </ul>
13. <div \*[ngIf](https://angular.io/api/common/NgIf)="selectedHero">
14. <h2>{{selectedHero.name | [uppercase](https://angular.io/api/common/UpperCasePipe)}} is my hero</h2>
15. </div>
16. </div>
17. `,
18. styles: [`
19. .heroes {
20. margin: 0 0 2em 0;
21. list-style-type: none;
22. padding: 0;
23. width: 15em;
24. }
25. .heroes li {
26. cursor: pointer;
27. [position](https://angular.io/api/router/Scroll#position): relative;
28. left: 0;
29. background-color: #EEE;
30. margin: .5em;
31. padding: .3em 0;
32. height: 1.6em;
33. border-radius: 4px;
34. }
35. .heroes .badge {
36. display: inline-block;
37. font-size: small;
38. color: white;
39. padding: 0.8em 0.7em 0 0.7em;
40. background-color: #607D8B;
41. line-height: 1em;
42. [position](https://angular.io/api/router/Scroll#position): relative;
43. left: -1px;
44. top: -4px;
45. height: 1.8em;
46. margin-right: .8em;
47. border-radius: 4px 0 0 4px;
48. }
49. `]
50. })
51. export class HeroesComponent implements [OnInit](https://angular.io/api/core/OnInit) {
52. heroes: Observable<Hero[]>;
53. selectedHero: Hero;
55. constructor(private heroService: HeroService) { }
57. ngOnInit() {
58. this.heroes = this.heroService.getHeroes();
59. }
60. }









## Decorate input and output properties- Style 05-12

* **Do** use the @[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() class decorators instead of the inputs and outputs properties of the@[Directive](https://angular.io/api/core/Directive) and @[Component](https://angular.io/api/core/Component) metadata:
* **Consider** placing @[Input](https://angular.io/api/core/Input)() or @[Output](https://angular.io/api/core/Output)() on the same line as the property it decorates.
* **Why?** It is easier and more readable to identify which properties in a class are inputs or outputs.
* **Why?** If you ever need to rename the property or event name associated with @[Input](https://angular.io/api/core/Input) or @[Output](https://angular.io/api/core/Output), you can modify it in a single place.
* **Why?** The metadata declaration attached to the directive is shorter and thus more readable.
* **Why?** 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.

app/heroes/shared/hero-button/hero-button.component.ts

content\_copy

1. /\* avoid \*/
3. @[Component](https://angular.io/api/core/Component)({
4. selector: 'toh-hero-button',
5. template: `<button></button>`,
6. inputs: [
7. 'label'
8. ],
9. outputs: [
10. 'change'
11. ]
12. })
13. export class HeroButtonComponent {
14. change = new [EventEmitter](https://angular.io/api/core/EventEmitter)<any>();
15. label: string;
16. }

app/heroes/shared/hero-button/hero-button.component.ts

content\_copy@[Component](https://angular.io/api/core/Component)({

selector: 'toh-hero-button',

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

})

export class HeroButtonComponent {

@[Output](https://angular.io/api/core/Output)() change = new [EventEmitter](https://angular.io/api/core/EventEmitter)<any>();

@[Input](https://angular.io/api/core/Input)() label: string;

}

## Avoid aliasing inputs and outputs- Style 05-13

* **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.
* **Why?** You should use an alias when the directive name is also an input property, and the directive name doesn't describe the property.

app/heroes/shared/hero-button/hero-button.component.ts

content\_copy/\* avoid pointless aliasing \*/

@[Component](https://angular.io/api/core/Component)({

selector: 'toh-hero-button',

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

})

export class HeroButtonComponent {

// Pointless aliases

@[Output](https://angular.io/api/core/Output)('changeEvent') change = new [EventEmitter](https://angular.io/api/core/EventEmitter)<any>();

@[Input](https://angular.io/api/core/Input)('labelAttribute') label: string;

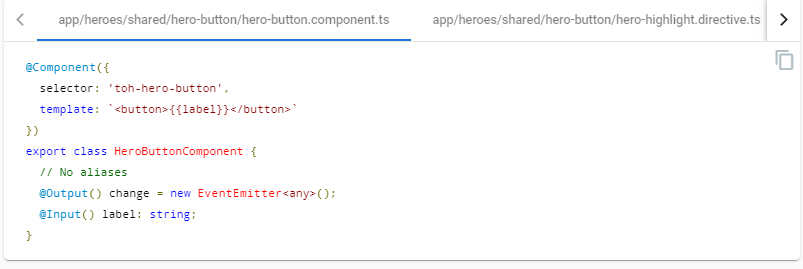
}

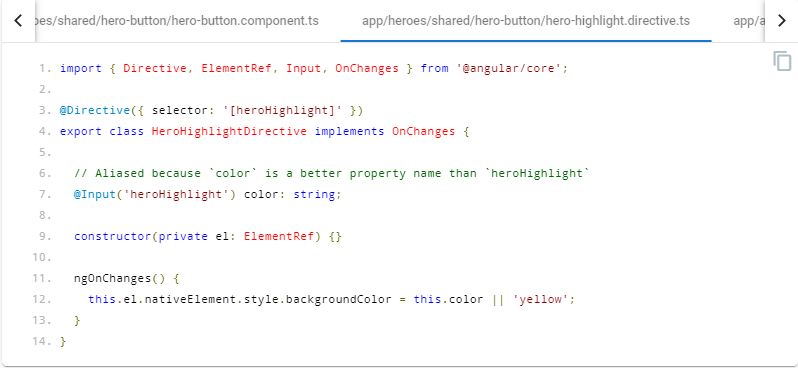
app/app.component.html

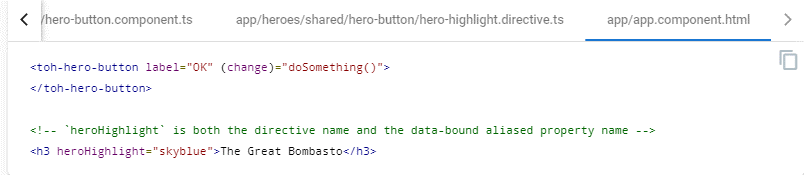
content\_copy<!-- avoid -->

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

</toh-hero-button>







## Member sequence- Style 05-14

* **Do** place properties up top followed by methods.
* **Do** place private members after public members, alphabetized.
* **Why?** Placing members in a consistent sequence makes it easy to read and helps instantly identify which members of the component serve which purpose.

app/shared/toast/toast.component.ts

content\_copy

1. /\* avoid \*/
3. export class ToastComponent implements [OnInit](https://angular.io/api/core/OnInit) {
5. private defaults = {
6. title: '',
7. [message](https://angular.io/api/common/http/HttpErrorResponse#message): 'May the Force be with you'
8. };
9. [message](https://angular.io/api/common/http/HttpErrorResponse#message): string;
10. title: string;
11. private toastElement: any;
13. ngOnInit() {
14. this.toastElement = document.getElementById('toh-toast');
15. }
17. // private methods
18. private hide() {
19. this.toastElement.style.opacity = 0;
20. window.setTimeout(() => this.toastElement.style.zIndex = 0, 400);
21. }
23. activate([message](https://angular.io/api/common/http/HttpErrorResponse#message) = this.defaults.message, title = this.defaults.title) {
24. this.title = title;
25. this.message = [message](https://angular.io/api/common/http/HttpErrorResponse#message);
26. this.show();
27. }
29. private show() {
30. console.log(this.message);
31. this.toastElement.style.opacity = 1;
32. this.toastElement.style.zIndex = 9999;
34. window.setTimeout(() => this.hide(), 2500);
35. }
36. }

app/shared/toast/toast.component.ts

content\_copy

1. export class ToastComponent implements [OnInit](https://angular.io/api/core/OnInit) {
2. // public [properties](https://angular.io/api/core/DebugElement#properties)
3. [message](https://angular.io/api/common/http/HttpErrorResponse#message): string;
4. title: string;
6. // private fields
7. private defaults = {
8. title: '',
9. [message](https://angular.io/api/common/http/HttpErrorResponse#message): 'May the Force be with you'
10. };
11. private toastElement: any;
13. // public methods
14. activate([message](https://angular.io/api/common/http/HttpErrorResponse#message) = this.defaults.message, title = this.defaults.title) {
15. this.title = title;
16. this.message = [message](https://angular.io/api/common/http/HttpErrorResponse#message);
17. this.show();
18. }
20. ngOnInit() {
21. this.toastElement = document.getElementById('toh-toast');
22. }
24. // private methods
25. private hide() {
26. this.toastElement.style.opacity = 0;
27. window.setTimeout(() => this.toastElement.style.zIndex = 0, 400);
28. }
30. private show() {
31. console.log(this.message);
32. this.toastElement.style.opacity = 1;
33. this.toastElement.style.zIndex = 9999;
34. window.setTimeout(() => this.hide(), 2500);
35. }
36. }

## Delegate complex component logic to services- Style 05-15

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

app/heroes/hero-list/hero-list.component.ts

content\_copy

1. /\* avoid \*/
3. import { [OnInit](https://angular.io/api/core/OnInit) } from '@angular/core';
4. import { [Http](https://angular.io/api/http/Http), [Response](https://angular.io/api/common/http/HttpEventType#Response) } from '@angular/[http](https://angular.io/api/common/http)';
6. import { Observable } from 'rxjs';
7. import { catchError, finalize, map } from 'rxjs/operators';
9. import { Hero } from '../shared/hero.model';
11. const heroesUrl = '[http](https://angular.io/api/common/http)://angular.io';
13. export class HeroListComponent implements [OnInit](https://angular.io/api/core/OnInit) {
14. heroes: Hero[];
15. constructor(private [http](https://angular.io/api/common/http): [Http](https://angular.io/api/http/Http)) {}
16. getHeroes() {
17. this.heroes = [];
18. this.http.get(heroesUrl).pipe(
19. map((response: [Response](https://angular.io/api/common/http/HttpEventType#Response)) => <Hero[]>response.json().data),
20. catchError(this.catchBadResponse),
21. finalize(() => this.hideSpinner())
22. ).subscribe((heroes: Hero[]) => this.heroes = heroes);
23. }
24. ngOnInit() {
25. this.getHeroes();
26. }
28. private catchBadResponse(err: any, [source](https://angular.io/): Observable<any>) {
29. // [log](https://angular.io/api/animations/browser/testing/MockAnimationDriver#log) and handle the exception
30. return new Observable();
31. }
33. private hideSpinner() {
34. // hide the spinner
35. }
36. }

app/heroes/hero-list/hero-list.component.ts

content\_copy

1. import { [Component](https://angular.io/api/core/Component), [OnInit](https://angular.io/api/core/OnInit) } from '@angular/core';
3. import { Hero, HeroService } from '../shared';
5. @[Component](https://angular.io/api/core/Component)({
6. selector: 'toh-hero-list',
7. template: `...`
8. })
9. export class HeroListComponent implements [OnInit](https://angular.io/api/core/OnInit) {
10. heroes: Hero[];
11. constructor(private heroService: HeroService) {}
12. getHeroes() {
13. this.heroes = [];
14. this.heroService.getHeroes()
15. .subscribe(heroes => this.heroes = heroes);
16. }
17. ngOnInit() {
18. this.getHeroes();
19. }
20. }

## Don't prefix output properties- Style 05-16

* **Do** name events without the prefix on.
* **Do** name event handler methods with the prefix on followed by the event name.
* **Why?** This is consistent with built-in events such as button clicks.
* **Why?** Angular allows for an [alternative syntax](https://angular.io/guide/template-syntax#binding-syntax) on-\*. If the event itself was prefixed with on this would result in an on-onEvent binding expression.

app/heroes/hero.component.ts

content\_copy/\* avoid \*/

@[Component](https://angular.io/api/core/Component)({

selector: 'toh-hero',

template: `...`

})

export class HeroComponent {

@[Output](https://angular.io/api/core/Output)() onSavedTheDay = new [EventEmitter](https://angular.io/api/core/EventEmitter)<boolean>();

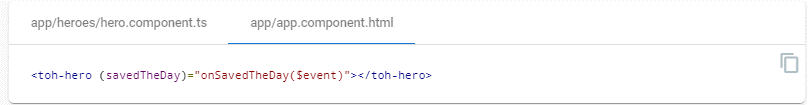
}

app/app.component.html

content\_copy<!-- avoid -->

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





## Put presentation logic in the component class- Style 05-17

* **Do** put presentation logic in the component class, and not in the template.
* **Why?** Logic will be contained in one place (the component class) instead of being spread in two places.
* **Why?** Keeping the component's presentation logic in the class instead of the template improves testability, maintainability, and reusability.

app/heroes/hero-list/hero-list.component.ts

content\_copy

1. /\* avoid \*/
3. @[Component](https://angular.io/api/core/Component)({
4. selector: 'toh-hero-list',
5. template: `
6. <section>
7. Our list of heroes:
8. <hero-profile \*[ngFor](https://angular.io/api/common/NgForOf)="let hero of heroes" [hero]="hero">
9. </hero-profile>
10. Total powers: {{totalPowers}}<br>
11. Average power: {{totalPowers / heroes.length}}
12. </section>
13. `
14. })
15. export class HeroListComponent {
16. heroes: Hero[];
17. totalPowers: number;
18. }

app/heroes/hero-list/hero-list.component.ts

content\_copy

1. @[Component](https://angular.io/api/core/Component)({
2. selector: 'toh-hero-list',
3. template: `
4. <section>
5. Our list of heroes:
6. <toh-hero \*[ngFor](https://angular.io/api/common/NgForOf)="let hero of heroes" [hero]="hero">
7. </toh-hero>
8. Total powers: {{totalPowers}}<br>
9. Average power: {{avgPower}}
10. </section>
11. `
12. })
13. export class HeroListComponent {
14. heroes: Hero[];
15. totalPowers: number;
17. get avgPower() {
18. return this.totalPowers / this.heroes.length;
19. }
20. }

# Directives

## Use directives to enhance an element- Style 06-01

* **Do** use attribute directives when you have presentation logic without a template.
* **Why?** Attribute directives don't have an associated template.
* **Why?** An element may have more than one attribute directive applied.

app/shared/highlight.directive.ts

content\_copy@[Directive](https://angular.io/api/core/Directive)({

selector: '[tohHighlight]'

})

export class HighlightDirective {

@[HostListener](https://angular.io/api/core/HostListener)('mouseover') onMouseEnter() {

// do highlight work

}

}

app/app.component.html

content\_copy<div tohHighlight>Bombasta</div>

## HostListener/HostBinding decorators versus host metadata- Style 06-03

* **Consider** preferring the @[HostListener](https://angular.io/api/core/HostListener) and @[HostBinding](https://angular.io/api/core/HostBinding) to the [host](https://angular.io/api/core/Directive#host) property of the @[Directive](https://angular.io/api/core/Directive) and @[Component](https://angular.io/api/core/Component)decorators.
* **Do** be consistent in your choice.
* **Why?** The property associated with @[HostBinding](https://angular.io/api/core/HostBinding) or the method associated with @[HostListener](https://angular.io/api/core/HostListener) can be modified only in a single place—in the directive's class. If you use the [host](https://angular.io/api/core/Directive#host) metadata property, you must modify both the property/method declaration in the directive's class and the metadata in the decorator associated with the directive.

app/shared/validator.directive.ts

content\_copyimport { [Directive](https://angular.io/api/core/Directive), [HostBinding](https://angular.io/api/core/HostBinding), [HostListener](https://angular.io/api/core/HostListener) } from '@angular/core';

@[Directive](https://angular.io/api/core/Directive)({

selector: '[tohValidator]'

})

export class ValidatorDirective {

@[HostBinding](https://angular.io/api/core/HostBinding)('attr.role') role = 'button';

@[HostListener](https://angular.io/api/core/HostListener)('mouseenter') onMouseEnter() {

// do work

}

}

* Compare with the less preferred [host](https://angular.io/api/core/Directive#host) metadata alternative.
* **Why?** The [host](https://angular.io/api/core/Directive#host) metadata is only one term to remember and doesn't require extra ES imports.

app/shared/validator2.directive.ts

content\_copy

1. import { [Directive](https://angular.io/api/core/Directive) } from '@angular/core';
3. @[Directive](https://angular.io/api/core/Directive)({
4. selector: '[tohValidator2]',
5. [host](https://angular.io/api/core/Directive#host): {
6. '[attr.role]': 'role',
7. '(mouseenter)': 'onMouseEnter()'
8. }
9. })
10. export class Validator2Directive {
11. role = 'button';
12. onMouseEnter() {
13. // do work
14. }
15. }

# Services

## Services are singletons- Style 07-01

* **Do** use services as singletons within the same injector. Use them for sharing data and functionality.
* **Why?** Services are ideal for sharing methods across a feature area or an app.
* **Why?** Services are ideal for sharing stateful in-memory data.

app/heroes/shared/hero.service.ts

content\_copyexport class HeroService {

constructor(private [http](https://angular.io/api/common/http): [Http](https://angular.io/api/http/Http)) { }

getHeroes() {

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

map((response: [Response](https://angular.io/api/common/http/HttpEventType#Response)) => <Hero[]>response.json()));

}

}

## Single responsibility- Style 07-02

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

## Providing a service-Style 07-03

* **Do** provide a service with the app root injector in the @[Injectable](https://angular.io/api/core/Injectable) decorator of the service.
* **Why?** The Angular injector is hierarchical.
* **Why?** When you provide the service to a root injector, that instance of the service is shared and available in every class that needs the service. This is ideal when a service is sharing methods or state.
* **Why?** When you register a service in the @[Injectable](https://angular.io/api/core/Injectable) decorator of the service, optimization tools such as those used by the [Angular CLI's](https://angular.io/cli) production builds can perform tree shaking and remove services that aren't used by your app.
* **Why?** 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.

src/app/treeshaking/service.ts

content\_copy@[Injectable](https://angular.io/api/core/Injectable)({

providedIn: 'root',

})

export class Service {

}

## Use the @Injectable() class decorator- Style 07-04

* **Do** use the @[Injectable](https://angular.io/api/core/Injectable)() class decorator instead of the @[Inject](https://angular.io/api/core/Inject) parameter decorator when using types as tokens for the dependencies of a service.
* **Why?** The Angular Dependency Injection (DI) mechanism resolves a service's own dependencies based on the declared types of that service's constructor parameters.
* **Why?** When a service accepts only dependencies associated with type tokens, the @[Injectable](https://angular.io/api/core/Injectable)() syntax is much less verbose compared to using @[Inject](https://angular.io/api/core/Inject)() on each individual constructor parameter.

app/heroes/shared/hero-arena.service.ts

content\_copy/\* avoid \*/

export class HeroArena {

constructor(

@[Inject](https://angular.io/api/core/Inject)(HeroService) private heroService: HeroService,

@[Inject](https://angular.io/api/core/Inject)([Http](https://angular.io/api/http/Http)) private [http](https://angular.io/api/common/http): [Http](https://angular.io/api/http/Http)) {}

}

app/heroes/shared/hero-arena.service.ts

content\_copy@[Injectable](https://angular.io/api/core/Injectable)()

export class HeroArena {

constructor(

private heroService: HeroService,

private [http](https://angular.io/api/common/http): [Http](https://angular.io/api/http/Http)) {}

}

# Data Services

## Talk to the server through a service- Style 08-01

* **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.
* **Why?** 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.
* **Why?** This makes it easier to test (mock or real) the data calls when testing a component that uses a data service.
* **Why?** 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.

# Lifecycle hooks

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

## Implement lifecycle hook interfaces- Style 09-01

* **Do** implement the lifecycle hook interfaces.
* **Why?** Lifecycle interfaces prescribe typed method signatures. Use those signatures to flag spelling and syntax mistakes.

app/heroes/shared/hero-button/hero-button.component.ts

content\_copy/\* avoid \*/

@[Component](https://angular.io/api/core/Component)({

selector: 'toh-hero-button',

template: `<button>OK<button>`

})

export class HeroButtonComponent {

[onInit](https://angular.io/)() { // misspelled

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

}

}

app/heroes/shared/hero-button/hero-button.component.ts

content\_copy@[Component](https://angular.io/api/core/Component)({

selector: 'toh-hero-button',

template: `<button>OK</button>`

})

export class HeroButtonComponent implements [OnInit](https://angular.io/api/core/OnInit) {

ngOnInit() {

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

}

}

# Appendix

Useful tools and tips for Angular.

## Codelyzer- Style A-01

* **Do** use [codelyzer](https://www.npmjs.com/package/codelyzer) to follow this guide.
* **Consider** adjusting the rules in codelyzer to suit your needs.

## File templates and snippets- Style A-02

* **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.

**Consider** using [snippets](https://marketplace.visualstudio.com/items?itemName=johnpapa.Angular2) for [Visual Studio Code](https://code.visualstudio.com/) that follow these styles and guidelines.

**Consider** using [snippets](https://atom.io/packages/angular-2-typescript-snippets) for [Atom](https://atom.io/) that follow these styles and guidelines.

**Consider** using [snippets](https://github.com/orizens/sublime-angular2-snippets) for [Sublime Text](http://www.sublimetext.com/) that follow these styles and guidelines.

**Consider** using [snippets](https://github.com/mhartington/vim-angular2-snippets) for [Vim](http://www.vim.org/) that follow these styles and guidelines.