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 unmistakeable 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](https://angular.io/api/core/Component#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](https://angular.io/api/core/NgModule#imports): [[BrowserModule](https://angular.io/api/platform-browser/BrowserModule)],
31. [declarations](https://angular.io/api/core/NgModule#declarations): [AppComponent],
32. [exports](https://angular.io/api/core/NgModule#exports): [AppComponent],
33. [bootstrap](https://angular.io/api/core/NgModule#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.

main.ts

app/app.module.ts

app/app.component.ts

app/heroes/heroes.component.ts

app/heroes/shared/hero.service.ts

app/heroes/shared/hero.model.ts

app/heroes/shared/mock-heroes.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);

As the app grows, this rule becomes even more important. [Back to top](https://angular.io/guide/styleguide#toc)

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.

[Back to top](https://angular.io/guide/styleguide#toc)

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.

[Back to top](https://angular.io/guide/styleguide#toc)

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.

[Back to top](https://angular.io/guide/styleguide#toc)

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](https://angular.io/api/core/Component), [Directive](https://angular.io/api/core/Directive), Module, [Pipe](https://angular.io/api/core/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 |

[Back to top](https://angular.io/guide/styleguide#toc)

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 |

[Back to top](https://angular.io/guide/styleguide#toc)

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));

[Back to top](https://angular.io/guide/styleguide#toc)

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 {}

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 {}

[Back to top](https://angular.io/guide/styleguide#toc)

Component custom prefix

Style 02-07

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 Tour of Heroes and the prefix adminrepresents 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 {}

[Back to top](https://angular.io/guide/styleguide#toc)

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.

[Back to top](https://angular.io/guide/styleguide#toc)

Directive custom prefix

Style 02-08

Do use a custom prefix for the selector of directives (e.g, the prefix toh from Tour of Heroes).

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 {}

[Back to top](https://angular.io/guide/styleguide#toc)

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 |

[Back to top](https://angular.io/guide/styleguide#toc)

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 |

[Back to top](https://angular.io/guide/styleguide#toc)

*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 |

[Back to top](https://angular.io/guide/styleguide#toc)

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 |

[Back to top](https://angular.io/guide/styleguide#toc)

Coding conventions

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

Classes

Style 03-01

Do use upper camel case 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() { }

}

[Back to top](https://angular.io/guide/styleguide#toc)

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 constdeclaration. 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

[Back to top](https://angular.io/guide/styleguide#toc)

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() { }

}

[Back to top](https://angular.io/guide/styleguide#toc)

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. }

[Back to top](https://angular.io/guide/styleguide#toc)

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';

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';

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

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

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