

# STYLE GUIDE

Angular 2 for TypeScript ▾

Write Angular 2 with style.

Welcome to the Angular 2 Style Guide

## Purpose

If you are looking for an opinionated style guide for syntax, conventions, and structuring Angular applications, then step right in.

The purpose of this style guide is to provide guidance on building Angular applications by showing the conventions we use and, more importantly, why we choose them.

## 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, we 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 we should almost never do. Code examples to *avoid* have an unmistakable red header.

## File Structure Conventions

Some code examples display a file that has one or more similarly named companion files. (e.g. `hero.component.ts` and `hero.component.html`).

The guideline will use 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.

## Table of Contents

1. [Single Responsibility](#)
2. [Naming](#)
3. [Coding Conventions](#)
4. [Application Structure](#)
5. [Components](#)
6. [Directives](#)
7. [Services](#)
8. [Data Services](#)
9. [Lifecycle Hooks](#)
10. [Routing](#)
11. [Appendix](#)

## Single Responsibility

We apply the [Single Responsibility Principle](#) to all Components, Services, and other symbols we create. This helps make our app cleaner, easier to read and maintain, and more testable.

### Rule of One

#### Style 01-01

**Do** define one thing (e.g. 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 Component 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.*

```

1.  /* avoid */
2.
3.  import { bootstrap }           from '@angular/platform-browser-dynamic';
4.  import { Component, OnInit } from '@angular/core';
5.
6.  class Hero {
7.    id: number;
8.    name: string;
9.  }
10.
11. @Component({
12.   selector: 'my-app',
13.   template: `
14.     <h1>{{title}}</h1>
15.     <pre>{{heroes | json}}</pre>
16.   `,
17.   styleUrls: ['app/app.component.css']
18. })
19. class AppComponent implements OnInit {
20.   title = 'Tour of Heroes';
21.
22.   heroes: Hero[] = [];
23.
24.   ngOnInit() {
25.     getHeroes().then(heroes => this.heroes = heroes);
26.   }
27. }
28.
29. bootstrap(AppComponent, []);
30.
31. const HEROES: Hero[] = [
32.   {id: 1, name: 'Bombasto'},
33.   {id: 2, name: 'Tornado'},
34.   {id: 3, name: 'Magnetia'},
35. ];
36.
37. function getHeroes(): Promise<Hero[]> {
38.   return Promise.resolve(HEROES); // TODO: get hero data from the server;
39. }

```

Better to redistribute the component and supporting activities into their own dedicated files.

app/main.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

```

1.  import { bootstrap }   from '@angular/platform-browser-dynamic';
2.
3.  import { AppComponent } from './app.component';
4.
5.  bootstrap(AppComponent, []);

```

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

[Back to top](#)

## 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](#)

## 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 us find our 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](#)

### Separate File Names with Dots and Dashes

#### Style 02-02

**Do** use dashes to separate words.

**Do** use dots to separate the descriptive name from the type.

**Do** use consistent 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 suffixes for the types including `*.service.ts`, `*.component.ts`, `*.pipe.ts`. Invent other suffixes where desired, but take care in having too many.

**Why?** Provides a consistent way to quickly identify what is in the file.

**Why?** Provides a consistent way to quickly find a specific file using an editor or IDE's fuzzy search techniques.

**Why?** Provides pattern matching for any automated tasks.

[Back to top](#)

## Components and Directives

### Style 02-03

**Do** use consistent names for all assets named after what they represent.

**Do** use upper camel case for symbols. Match the name of the symbol to the naming of the file.

**Do** append the symbol name with the suffix that it represents.

**Why?** Provides a consistent way to quickly identify and reference assets.

**Why?** Upper camel case is conventional for identifying objects that can be instantiated using a constructor.

**Why?** The `Component` suffix is more commonly used and is more explicitly descriptive.

Symbol Name	File Name
<pre>@Component({ ... }) export class AppComponent {}</pre>	app.component.ts
<pre>@Component({ ... }) export class HeroesComponent</pre>	heroes.component.ts
<pre>@Component({ ... }) export class HeroListComponent</pre>	hero-list.component.ts
<pre>@Component({ ... }) export class HeroDetailComponent</pre>	hero-detail.component.ts
<pre>@Directive({ ... }) export class ValidationDirective</pre>	validation.directive.ts

[Back to top](#)

## Service Names

### Style 02-04

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

**Do** use upper camel case for services.

**Do** suffix services with `Service` when it is not clear what they are (e.g. when they are nouns).

**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
<pre>@Injectable() export class HeroDataService {}</pre>	hero-data.service.ts
<pre>@Injectable() export class CreditService {}</pre>	credit.service.ts
<pre>@Injectable() export class LoggerService {}</pre>	logger.service.ts

[Back to top](#)

## Bootstrapping

### Style 02-05

**Do** put bootstrapping and platform logic for the app in a file named `main.ts`.

**Avoid** putting app logic in the `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.

[Back to top](#)

## Directive Selectors

### Style 02-06

**Do** Use lower camel case for naming the selectors of our 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 2 HTML parser is case sensitive and will recognize lower camel case.

[Back to top](#)

## Custom Prefix for Components

## Style 02-07

**Do** use a custom prefix for the selector of our components. 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 name collisions.

**Why?** Makes it easier to promote and share our feature in other apps.

**Why?** Our Components and elements are easily identified.

AVOID: `app/heroes/hero.component.ts`

```
1. /* avoid */
2.
3. // HeroComponent is in the Tour of Heroes feature
4. @Component({
5.   selector: 'hero'
6. })
7. export class HeroComponent {}
```



AVOID: `app/users/users.component.ts`

```
1. /* avoid */
2.
3. // UsersComponent is in an Admin feature
4. @Component({
5.   selector: 'users'
6. })
7. export class UsersComponent {}
```



`app/heroes/hero.component.ts`

```
1. @Component({
2.   selector: 'toh-hero'
3. })
4. export class HeroComponent {}
```



`app/users/users.component.ts`

```
1. @Component({
2.   selector: 'admin-users'
3. })
4. export class UsersComponent {}
```



## Custom Prefix for Directives

### Style 02-08

**Do** use a custom prefix for the selector of our directives (for instance below we use the prefix `toh` from **Tour of Heroes**).

**Why?** Prevents name collisions.

**Why?** Our Directives are easily identified.



AVOID: app/shared/validate.directive.ts

```
1. /* avoid */
2.
3. @Directive({
4.   selector: '[validate]'
5. })
6. export class ValidateDirective {}
```



app/shared/validate.directive.ts

```
1. @Directive({
2.   selector: '[tohValidate]'
3. })
4. export class ValidateDirective {}
```



[Back to top](#)

## 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
<pre>@Pipe({ name: 'ellipsis' }) export class EllipsisPipe implements PipeTransform { }</pre>	ellipsis.pipe.ts
<pre>@Pipe({ name: 'initCaps' }) export class InitCapsPipe implements PipeTransform { }</pre>	init-caps.pipe.ts

[Back to top](#)

## 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](#) or other test runners.

Symbol Name	File Name
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](#)

## End to End 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.ts`.

**Why?** Provides a consistent way to quickly identify end-to-end tests.

**Why?** Provides pattern matching for test runners and build automation.

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

[Back to top](#)

## Coding Conventions

Have 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. We often use upper camel case to indicate a constructable asset.

AVOID: app/shared/exception.service.ts

```
1. /* avoid */
2.
3. export class exceptionService {
4.   constructor() { }
5. }
```



app/shared/exception.service.ts

```
1. export class ExceptionService {
2.   constructor() { }
3. }
```



[Back to top](#)

## Constants

### Style 03-02

**Do** use uppercase with underscores when naming constants.

**Why?** Follows conventional thinking for constants.

**Why?** Constants can easily be identified.

AVOID: app/shared/data.service.ts

```
1. /* avoid */
2.
3. export const heroesUrl = 'api/heroes';
4. export const villainsUrl = 'api/villains';
```



app/shared/data.service.ts

```
1. export const HEROES_URL = 'api/heroes';
2. export const VILLAIN_URL = 'api/villains';
```



[Back to top](#)

## Interfaces

### Style 03-03

**Do** name an interface using upper camel case.

**Consider** naming an interface without an `I` prefix.

**Why?** When we use types, we can often simply use the class as the type.

AVOID: app/shared/hero-collector.service.ts

```
1. /* avoid */
2.
3. import { Injectable } from '@angular/core';
4.
5. import { IHero } from './hero.model.avoid';
6.
7. @Injectable()
8. export class HeroCollectorService {
9.   hero: IHero;
10.
11.   constructor() { }
12. }
```



app/shared/hero-collector.service.ts

```
1. import { Injectable } from '@angular/core';
2.
3. import { Hero } from './hero.model';
4.
5. @Injectable()
6. export class HeroCollectorService {
7.   hero: Hero;
8.
9.   constructor() { }
10. }
```



[Back to top](#)

## 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.

AVOID: app/shared/toast/toast.service.ts

```
1.  /* avoid */
2.
3.  import { Injectable } from '@angular/core';
4.
5.  @Injectable()
6.  export class ToastService {
7.    message: string;
8.
9.    private _toastCount: number;
10.
11.    hide() {
12.      this._toastCount--;
13.      this._log();
14.    }
15.
16.    show() {
17.      this._toastCount++;
18.      this._log();
19.    }
20.
21.    private _log() {
22.      console.log(this.message);
23.    }
24. }
```



app/shared/toast/toast.service.ts

```
1.  import { Injectable } from '@angular/core';
2.
3.  @Injectable()
4.  export class ToastService {
5.    message: string;
6.
7.    private toastCount: number;
8.
9.    hide() {
10.      this.toastCount--;
11.      this.log();
12.    }
13.
14.    show() {
15.      this.toastCount++;
16.      this.log();
17.    }
18.
19.    private log() {
20.      console.log(this.message);
21.    }
22. }
```



[Back to top](#)

## Import Destructuring Spacing

### Style 03-05

**Do** leave one whitespace character inside of the `import` statements' curly braces when destructuring.

**Why?** Whitespace makes it easier to read the imports.

AVOID: app/+heroes/shared/hero.service.ts

```
1. /* avoid */
2.
3. import {Injectable} from '@angular/core';
4. import {Http, Response} from '@angular/http';
5.
6. import {Hero} from './hero.model';
7. import {ExceptionService, SpinnerService, ToastService} from '../..//shared';
```



app/+heroes/shared/hero.service.ts

```
1. import { Injectable } from '@angular/core';
2. import { Http, Response } from '@angular/http';
3.
4. import { Hero } from './hero.model';
5. import { ExceptionService, SpinnerService, ToastService } from
  '../..//shared';
```



[Back to top](#)

## Import Line Spacing

### Style 03-06

**Do** leave one empty line between third party imports and imports of code we created.

**Do** list import lines alphabetized by the module.

**Do** list destructured imported assets alphabetically.

**Why?** The empty line makes it easy to read and locate imports.

**Why?** Alphabetizing makes it easier to read and locate imports.

AVOID: app/+heroes/shared/hero.service.ts

```
1. /* avoid */
2.
3. import { ExceptionService, SpinnerService, ToastService } from
  '../..//shared';
4. import { Http, Response } from '@angular/http';
5. import { Injectable } from '@angular/core';
6. import { Hero } from './hero.model';
```



app/+heroes/shared/hero.service.ts

```
1. import { Injectable } from '@angular/core';
2. import { Http, Response } from '@angular/http';
3.
4. import { Hero } from './hero.model';
5. import { ExceptionService, SpinnerService, ToastService } from
  '../..//shared';
```



[Back to top](#)

## Application Structure

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 `app`. All content is 1 feature per file. Each component, service, and pipe is in its own file. All 3rd party vendor scripts are stored in another folder and not in the `app` folder. We didn't write them and we don't want them cluttering our app. Use the naming conventions for files in this guide.

[Back to top](#)

## LIFT

### Style 04-01

**Do** structure the app such that we can **L**ocate our code quickly, **I**dentify the code at a glance, keep the **F**lattest structure we 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. Another way to check our app structure is to ask ourselves: How quickly can we open and work in all of the related files for a feature?

[Back to top](#)

## Locate

### Style 04-02

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

**Why?** We find this to be super important for a project. If we cannot find the files we need to work on quickly, we will not be able to work as efficiently as possible, and the structure will need to change. We may not know the file name or where its related files are, so putting them in the most intuitive locations and near each other saves a ton of time. A descriptive folder structure can help with this.

[Back to top](#)

## Identify

### Style 04-03

**Do** name the file such that we 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?** We spend less time hunting and pecking for code, and become more efficient. If this means we want longer file names, then so be it.

There are deviations of the 1 per file rule when we have a set of very small features that are all related to each other, as they are still easily identifiable.

[Back to top](#)

## Flat

### Style 04-04

**Do** keep a flat folder structure as long as possible.

**Consider** creating folders when we get to seven or more files.

**Why?** Nobody wants to search seven levels of folders to find a file. In a folder structure there is no hard and fast number rule, but when a folder has seven to ten files, that may be time to create subfolders. We base it on our comfort level. Use a flatter structure until there is an obvious value (to help the rest of LIFT) in creating a new folder.

[Back to top](#)

## T-DRY (Try to be DRY)

### Style 04-05

**Do** be DRY (Don't Repeat Yourself)

**Avoid** being so DRY that we sacrifice readability.

**Why?** Being DRY is important, but not crucial if it sacrifices the others in LIFT, which is why we call it T-DRY. We don't want to type `hero-view.component.html` for a view because, well, it's obviously a view. If it is not obvious or by convention, then we name it.

[Back to top](#)

## 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 `app`.



**Consider** creating a folder for each component including its `.ts`, `.html`, `.css` and `.spec` file.

**Why?** Helps us 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.

#### Overall Folder and File Structure

```
src
├── app
│   ├── +heroes
│   │   ├── hero
│   │   │   ├── hero.component.ts | html | css | spec.ts
│   │   │   └── index.ts
│   │   ├── hero-list
│   │   │   ├── hero-list.component.ts | html | css | spec.ts
│   │   │   └── index.ts
│   │   ├── shared
│   │   │   ├── hero.model.ts
│   │   │   ├── hero.service.ts | spec.ts
│   │   │   └── index.ts
│   │   ├── heroes.component.ts | html | css | spec.ts
│   │   └── index.ts
│   ├── shared
│   │   └── ...
│   ├── app.component.ts | html | css | spec.ts
│   └── main.ts
├── index.html
└── ...
```

While we prefer our Components to be in their own dedicated folder, 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. Be consistent.

[Back to top](#)

**Shared Folder**

Style 04-07

**Do** put all shared files within a component feature in a `shared` folder.

**Consider** creating a folder for each component including its `.ts`, `.html`, `.css` and `.spec` file.

**Why?** Separates shared files from the components within a feature.

**Why?** Makes it easier to locate shared files within a component feature.

#### Shared Folder

```
src
├── app
│   ├── +heroes
│   │   ├── hero
│   │   │   └── ...
│   │   ├── hero-list
│   │   │   └── ...
│   │   └── shared
│   │       ├── hero-button
│   │       │   └── ...
│   │       ├── hero.model.ts
│   │       ├── hero.service.ts | spec.ts
│   │       └── index.ts
│   ├── heroes.component.ts | html | css | spec.ts
│   └── index.ts
├── shared
│   ├── exception.service.ts | spec.ts
│   ├── index.ts
│   └── nav
│       └── ...
├── app.component.ts | html | css | spec.ts
├── main.ts
└── index.html
...
```

[Back to top](#)

## Folders-by-Feature Structure

### Style 04-08

**Do** create folders named for the feature they represent.

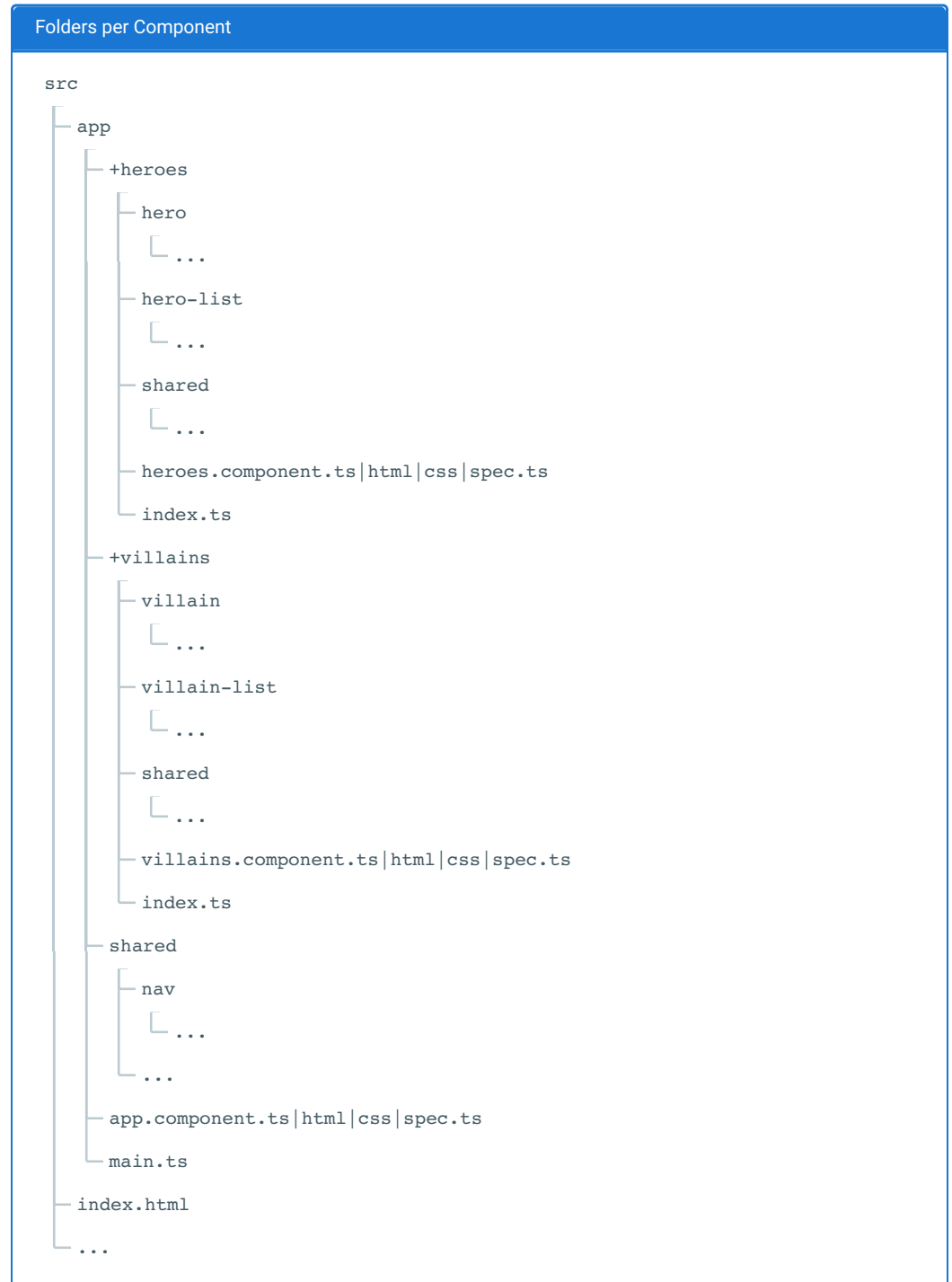
**Why?** A developer can locate the code, identify what each file represents at a glance, the structure is as flat as it can be, and there is no repetitive nor redundant names.

**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 (e.g. 10+) locating them is easier with a consistent folder structures and more difficult in flat structures.

Below is an example of a small app with folders per component.



[Back to top](#)

## Style 04-09

**Do** put components that define the overall layout in a `shared` folder.

**Do** put shared layout components in their own folder, under the `shared` folder.

**Why?** We need a place to host our layout for our app. Our navigation bar, footer, and other aspects of the app that are needed for the entire app.

**Why?** Organizes all layout in a consistent place re-used throughout the application.

### Folder for Layout Components

```
src
├── app
│   ├── +heroes
│   │   └── ...
│   ├── shared
│   │   ├── nav
│   │   │   ├── index.ts
│   │   │   └── nav.component.ts | html | css | spec.ts
│   │   ├── footer
│   │   │   ├── index.ts
│   │   │   └── footer.component.ts | html | css | spec.ts
│   │   ├── index.ts
│   │   └── ...
│   ├── app.component.ts | html | css | spec.ts
│   └── main.ts
├── index.html
└── ...
```

[Back to top](#)

## Create and Import Barrels

### Style 04-10

**Do** create a file that imports, aggregates, and re-exports items. We call this technique a **barrel**.

**Do** name this barrel file `index.ts`.

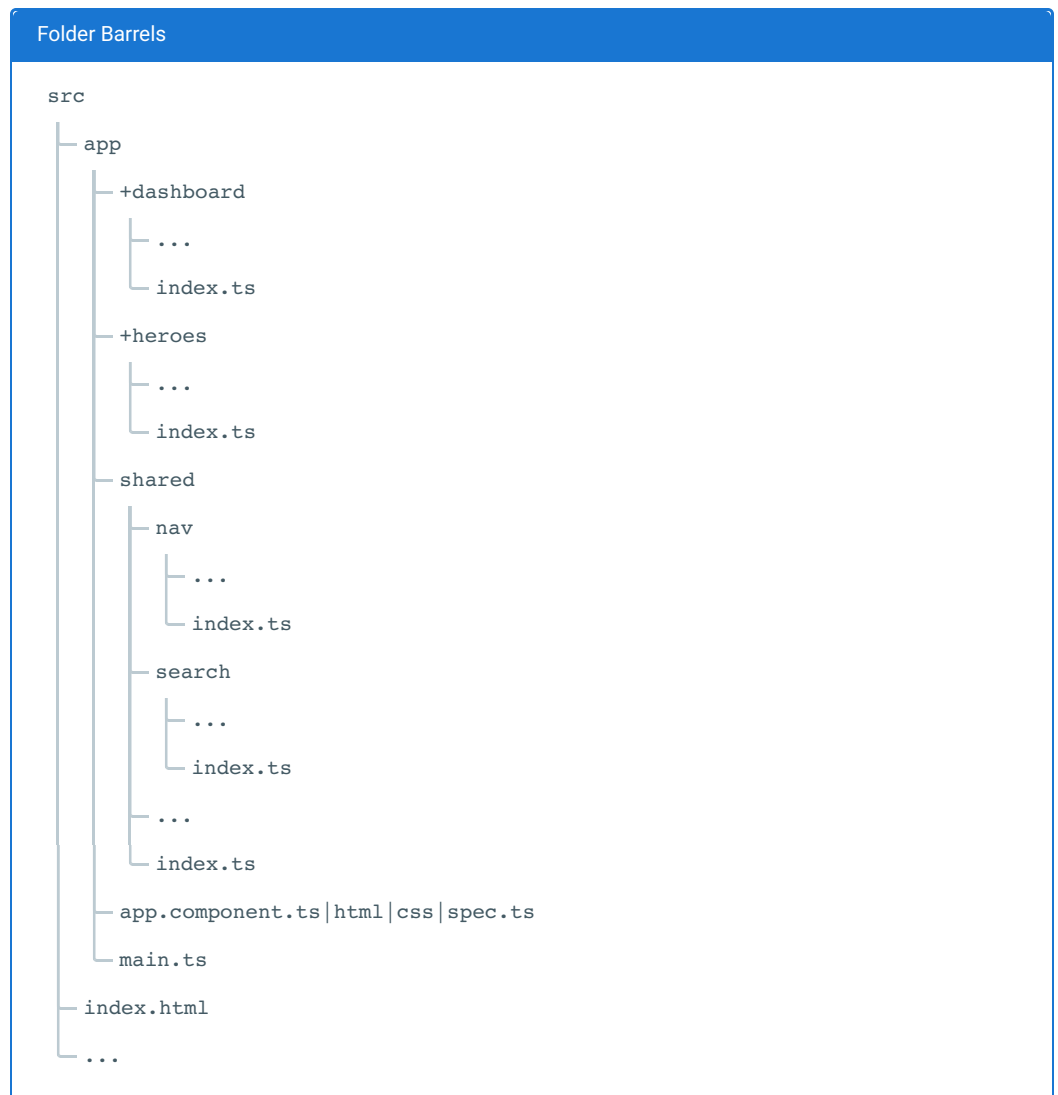
**Why?** A barrel aggregates many imports into a single import.

**Why?** A barrel reduces the number of imports a file may need.

**Why?** A barrel shortens import statements.

app/shared/index.tsapp/shared/filter-text/index.tsapp/shared/modal/index.tsapp/shared/nav/index.tsapp/shared/spinner/index.tsapp/shared/toast/index.ts

```
1. export * from './config';
2. export * from './entity.service';
3. export * from './exception.service';
4. export * from './filter-text';
5. export * from './init-caps.pipe';
6. export * from './modal';
7. export * from './nav';
8. export * from './spinner';
9. export * from './toast';
```



```

1.  /* avoid */
2.
3.  import { Component, OnInit } from '@angular/core';
4.
5.  import { CONFIG } from '../shared/config';
6.  import { EntityService } from '../shared/entity.service';
7.  import { ExceptionService } from '../shared/exception.service';
8.  import { FilterTextComponent } from '../shared/filter-text/filter-
  text.component';
9.  import { InitCapsPipe } from '../shared/init-caps.pipe';
10. import { SpinnerService } from '../shared/spinner/spinner.service';
11. import { ToastService } from '../shared/toast/toast.service';
12.
13. @Component({
14.   selector: 'toh-heroes',
15.   templateUrl: 'app/+heroes/heroes.component.html'
16. })
17. export class HeroesComponent implements OnInit {
18.   constructor() { }
19.
20.   ngOnInit() { }
21. }

```



```

1.  import { Component, OnInit } from '@angular/core';
2.
3.  import {
4.    CONFIG,
5.    EntityService,
6.    ExceptionService,
7.    FilterTextComponent,
8.    InitCapsPipe,
9.    SpinnerService,
10.   ToastService
11. } from '../shared';
12.
13. @Component({
14.   selector: 'toh-heroes',
15.   templateUrl: 'app/+heroes/heroes.component.html'
16. })
17. export class HeroesComponent implements OnInit {
18.   constructor() { }
19.
20.   ngOnInit() { }
21. }

```



[Back to top](#)

## 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.

[Back to top](#)

## Prefix Lazy Loaded Folders with +

### Style 04-12

**Do** prefix the name of a *lazy loaded folder* with a (+) e.g., `+dashboard/`.

**Why?** Lazy loaded code paths are easily identifiable by their `+` prefix.

**Why?** Lazy loaded code paths are easily distinguishable from non lazy loaded paths.

**Why?** If we see an `import` path that contains a `+`, we can quickly refactor to use lazy loading.

#### Lazy Loaded Folders

```
src
├── app
│   └── +dashboard
│       ├── dashboard.component.ts | html | css | spec.ts
│       └── index.ts
```

[Back to top](#)

## Never Directly Import Lazy Loaded Folders

### Style 04-13

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

**Why?** Directly importing a module loads it immediately when our intention is to load it on demand.

AVOID: `app/app.component.ts`

```
import { HeroesComponent } from './+heroes';
```



[Back to top](#)

## Lazy Loaded Folders May Import From a Parent

### Style 04-14

**Do** allow lazy loaded modules to import a module from a parent folder.

**Why?** A parent module has already been loaded by the time the lazy loaded module imports it.

app/heroes/heroes.component.ts

```
import { Logger } from '../shared/logger.service';
```



[Back to top](#)

## Use Component Router to Lazy Load

Style 04-15

**Do** use the Component Router to lazy load routable features.

**Why?** That's the easiest way to load a module on demand.

[Back to top](#)

## Components

### Components Selector Naming

Style 05-02

**Do** use `kebab-case` for naming the element selectors of our components.

**Why?** Keeps the element names consistent with the specification for [Custom Elements](#).

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

```
1. /* avoid */
2.
3. @Component({
4.   selector: 'tohHeroButton'
5. })
6. export class HeroButtonComponent {}
```



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

app/app.component.html

```
1. @Component({
2.   selector: 'toh-hero-button'
3. })
4. export class HeroButtonComponent {}
```



[Back to top](#)

## Components as Elements



## Style 05-03

**Do** define Components as elements via the selector.

**Why?** Components have templates containing HTML and optional Angular template syntax. They are most associated with putting content on a page, and thus are more closely aligned with elements.

**Why?** Components are derived from Directives, and thus their selectors can be elements, attributes, or other selectors. Defining the selector as an element provides consistency for components that represent content with a template.

**Why?** It is easier to recognize that a symbol is a component vs a directive by looking at the template's html.

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

```
1. /* avoid */
2.
3. @Component({
4.   selector: '[tohHeroButton]'
5. })
6. export class HeroButtonComponent {}
```



AVOID: app/heroes/hero-button/hero-button.component.html

```
1. <!-- avoid -->
2.
3. <div tohHeroButton></div>
```



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

app/app.component.html

```
1. @Component({
2.   selector: 'toh-hero-button'
3. })
4. export class HeroButtonComponent {}
```



[Back to top](#)

## Extract Template 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 our component name.

**Do** name the style file `[component-name].component.css`, where [component-name] is our component name.

**Why?** Syntax hints for inline templates in (.js and .ts) code files are not supported by some editors.

**Why?** A component file's logic is easier to read when not mixed with inline template and styles.



```

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

```

app/heroes/heroes.component.tsapp/heroes/heroes.component.htmlapp/heroes/heroes.component.css

```
1. @Component({
2.   selector: 'toh-heroes',
3.   templateUrl: 'heroes.component.html',
4.   styleUrls: ['heroes.component.css']
5. })
6. export class HeroesComponent implements OnInit {
7.   heroes: Hero[];
8.   selectedHero: Hero;
9.
10.  ngOnInit() { }
11. }
```

[Back to top](#)

## Decorate Input and Output Properties Inline

### Style 05-12

Do use `@Input` and `@Output` instead of the `inputs` and `outputs` properties of the `@Directive` and `@Component` decorators:

Do place the `@Input()` or `@Output()` on the same line as the property they decorate.

**Why?** It is easier and more readable to identify which properties in a class are inputs or outputs.

**Why?** If we ever need to rename the property or event name associated to `@Input` or `@Output` we can modify it on 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 makes for shorter code and still easily identifies the property as an input or output.

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

```
1. /* avoid */
2.
3. @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<any>();
15.   label: string;
16. }
```

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

```
1. @Component({
2.   selector: 'toh-hero-button',
3.   template: `<button>OK</button>`
4. })
5. export class HeroButtonComponent {
6.   @Output() change = new EventEmitter<any>();
7.   @Input() label: string;
8. }
```



[Back to top](#)

## Avoid Renaming Inputs and Outputs

### Style 05-13

**Avoid** renaming inputs and outputs, when possible.

**Why?** May lead to confusion when the output or the input properties of a given directive are named a given way but exported differently as a public API.

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

```
1. /* avoid */
2.
3. @Component({
4.   selector: 'toh-hero-button',
5.   template: `<button>{{label}}</button>`
6. })
7. export class HeroButtonComponent {
8.   @Output('changeEvent') change = new EventEmitter<any>();
9.   @Input('labelAttribute') label: string;
10. }
```



AVOID: app/app.component.html

```
1. <!-- avoid -->
2.
3. <toh-hero-button labelAttribute="OK" (changeEvent)="doSomething()">
4. </toh-hero-button>
```



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

app/app.component.html

```
1. @Component({
2.   selector: 'toh-hero-button',
3.   template: `<button>{{label}}</button>`
4. })
5. export class HeroButtonComponent {
6.   @Output() change = new EventEmitter<any>();
7.   @Input() label: string;
8. }
```



[Back to top](#)

## Member Sequence

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 we instantly identify which members of the component serve which purpose.

AVOID: app/shared/toast/toast.component.ts

```
1.  /* avoid */
2.
3.  export class ToastComponent implements OnInit {
4.
5.      private defaults = {
6.          title: '',
7.          message: 'May the Force be with You'
8.      };
9.      message: string;
10.     title: string;
11.     private toastElement: any;
12.
13.     ngOnInit() {
14.         this.toastElement = document.getElementById('toh-toast');
15.     }
16.
17.     // private methods
18.     private hide() {
19.         this.toastElement.style.opacity = 0;
20.         window.setTimeout(() => this.toastElement.style.zIndex = 0, 400);
21.     }
22.
23.     activate(message = this.defaults.message, title = this.defaults.title) {
24.         this.title = title;
25.         this.message = message;
26.         this.show();
27.     }
28.
29.     private show() {
30.         console.log(this.message);
31.         this.toastElement.style.opacity = 1;
32.         this.toastElement.style.zIndex = 9999;
33.
34.         window.setTimeout(() => this.hide(), 2500);
35.     }
36. }
```



```

1. export class ToastComponent implements OnInit {
2.     // public properties
3.     message: string;
4.     title: string;
5.
6.     // private fields
7.     private defaults = {
8.         title: '',
9.         message: 'May the Force be with You'
10.    };
11.    private toastElement: any;
12.
13.    // public methods
14.    activate(message = this.defaults.message, title = this.defaults.title) {
15.        this.title = title;
16.        this.message = message;
17.        this.show();
18.    }
19.
20.    ngOnInit() {
21.        this.toastElement = document.getElementById('toh-toast');
22.    }
23.
24.    // private methods
25.    private hide() {
26.        this.toastElement.style.opacity = 0;
27.        window.setTimeout(() => this.toastElement.style.zIndex = 0, 400);
28.    }
29.
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. }

```



[Back to top](#)

## Put Logic in 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.

```

1.  /* avoid */
2.
3.  import { OnInit } from '@angular/core';
4.  import { Http, Response } from '@angular/http';
5.  import { Observable } from 'rxjs/Observable';
6.
7.  import { Hero } from '../shared/hero.model';
8.
9.  const heroesUrl = 'http://angular.io';
10.
11. export class HeroListComponent implements OnInit {
12.   heroes: Hero[];
13.   constructor(private http: Http) {}
14.   getHeroes() {
15.     this.heroes = [];
16.     this.http.get(heroesUrl)
17.       .map((response: Response) => <Hero[]>response.json().data)
18.       .catch(this.catchBadResponse)
19.       .finally(() => this.hideSpinner())
20.       .subscribe((heroes: Hero[]) => this.heroes = heroes);
21.   }
22.   ngOnInit() {
23.     this.getHeroes();
24.   }
25.
26.   private catchBadResponse(err: any, source: Observable<any>) {
27.     // log and handle the exception
28.     return new Observable();
29.   }
30.
31.   private hideSpinner() {
32.     // hide the spinner
33.   }
34. }

```



```

1.  import { Component, OnInit } from '@angular/core';
2.
3.  import { Hero, HeroService } from '../shared/index';
4.
5.  @Component({
6.    selector: 'toh-hero-list',
7.    template: `...`
8.  })
9.  export class HeroListComponent implements OnInit {
10.   heroes: Hero[];
11.   constructor(private heroService: HeroService) {}
12.   getHeros() {
13.     this.heroes = [];
14.     this.heroService.getHeroes()
15.       .subscribe(heroes => this.heroes = heroes);
16.   }
17.   ngOnInit() {
18.     this.getHeros();
19.   }
20. }

```



[Back to top](#)

**Don't Prefix Output Properties**

## Style 05-16

**Do** name events without the prefix `on`.

**Do** name our 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** `on-*`. If the event itself was prefixed with `on` this would result in an `on-onEvent` binding expression.

AVOID: app/heroes/hero.component.ts

```
1. /* avoid */
2.
3. @Component({
4.   selector: 'toh-hero',
5.   template: `...`
6. })
7. export class HeroComponent {
8.   @Output() onSaveTheDay = new EventEmitter<boolean>();
9. }
```



AVOID: app/app.component.html

```
1. <!-- avoid -->
2.
3. <toh-hero (onSavedTheDay)="onSavedTheDay($event)"></toh-hero>
```



app/heroes/hero.component.ts

app/app.component.html

```
1. export class HeroComponent {
2.   @Output() savedTheDay = new EventEmitter<boolean>();
3. }
```



[Back to top](#)

## 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.



```

1.  /* avoid */
2.
3.  @Component({
4.    selector: 'toh-hero-list',
5.    template: `
6.      <section>
7.        Our list of heroes:
8.        <hero-profile *ngFor="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. }

```



```

1.  @Component({
2.    selector: 'toh-hero-list',
3.    template: `
4.      <section>
5.        Our list of heroes:
6.        <hero-profile *ngFor="let hero of heroes" [hero]="hero">
7.        </hero-profile>
8.        Total powers: {{totalPowers}}<br>
9.        Average power: {{avgPower}}
10.      </section>
11.  `
12. })
13. export class HeroListComponent {
14.   heroes: Hero[];
15.   totalPowers: number;
16.
17.   get avgPower() {
18.     return this.totalPowers / this.heroes.length;
19.   }
20. }

```



[Back to top](#)

## Directives

[Back to top](#)

### Use Directives to Enhance an Existing Element

#### Style 06-01

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

**Why?** Attributes directives don't have an associated template.

**Why?** An element may have more than one attribute directive applied.

app/shared/highlight.directive.ts

```
1. @Directive({
2.   selector: '[tohHighlight]'
3. })
4. export class HighlightDirective {
5.   @HostListener('mouseover') onMouseEnter() {
6.     // do highlight work
7.   }
8. }
```



app/app.component.html

```
<div [tohHighlight]>Bombasta</div>
```



[Back to top](#)

## Use HostListener and HostBinding Class Decorators

### Style 06-03

**Do** use `@HostListener` and `@HostBinding` instead of the `host` property of the `@Directive` and `@Component` decorators:

**Why?** The property or method name associated with `@HostBinding` or respectively `@HostListener` should be modified only in a single place - in the directive's class. In contrast if we use `host` we need to modify both the property declaration inside the controller, and the metadata associated to the directive.

**Why?** The metadata declaration attached to the directive is shorter and thus more readable.

AVOID: app/shared/validate.directive.ts

```
1. /* avoid */
2.
3. @Directive({
4.   selector: '[tohValidator]',
5.   host: {
6.     '(mouseenter)': 'onMouseEnter()',
7.     'attr.role': 'button'
8.   }
9. })
10. export class ValidatorDirective {
11.   role = 'button';
12.   onMouseEnter() {
13.     // do work
14.   }
15. }
```



app/shared/validate.directive.ts

```
1. @Directive({
2.   selector: '[tohValidator]'
3. })
4. export class ValidatorDirective {
5.   @HostBinding('attr.role') role = 'button';
6.   @HostListener('mouseenter') onMouseEnter() {
7.     // do work
8.   }
9. }
```



[Back to top](#)

## Services

### Services are Singletons in Same Injector

#### 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

```
1. export class HeroService {
2.   constructor(private http: Http) { }
3.
4.   getHeroes() {
5.     return this.http.get('api/heroes')
6.       .map((response: Response) => <Hero[]>response.json().data);
7.   }
8. }
```



[Back to top](#)

### 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.

[Back to top](#)

## Providing a Service

### Style 07-03

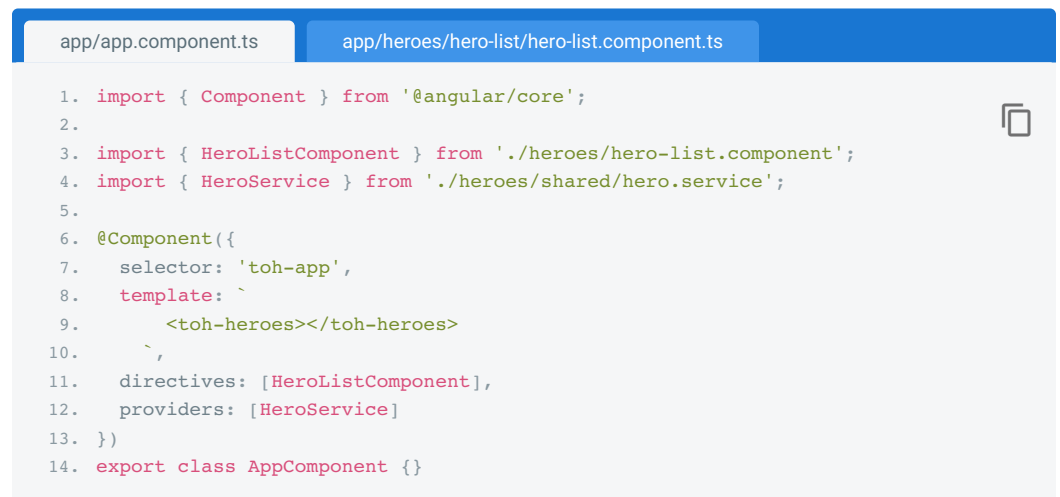
**Do** provide services to the Angular 2 injector at the top-most component where they will be shared.

**Why?** The Angular 2 injector is hierarchical.

**Why?** When providing the service to a top level component, that instance is shared and available to all child components of that top level component.

**Why?** This is ideal when a service is sharing methods or state.

**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.



```
1. import { Component } from '@angular/core';
2.
3. import { HeroListComponent } from './heroes/hero-list.component';
4. import { HeroService } from './heroes/shared/hero.service';
5.
6. @Component({
7.   selector: 'toh-app',
8.   template: `
9.     <toh-heroes></toh-heroes>
10.   `,
11.   directives: [HeroListComponent],
12.   providers: [HeroService]
13. })
14. export class AppComponent {}
```

[Back to top](#)

## Use the @Injectable() Class Decorator

### Style 07-04

**Do** use the `@Injectable` class decorator instead of the `@Inject` parameter decorator when using types as tokens for the dependencies of a service.

**Why?** The Angular DI mechanism resolves all the dependencies of our services based on their types declared with the services' constructors.

**Why?** 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: app/heroes/shared/hero-arena.service.ts

```
1. /* avoid */
2.
3. export class HeroArena {
4.   constructor(
5.     @Inject(HeroService) private heroService: HeroService,
6.     @Inject(Http) private http: Http) {}
7. }
```



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

```
1. @Injectable()
2. export class HeroArena {
3.   constructor(
4.     private heroService: HeroService,
5.     private http: Http) {}
6. }
```



[Back to top](#)

## Data Services

### Separate Data Calls

#### 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?** Data service implementation may have very specific code to handle the data repository. This may include headers, how to talk to the data, or other services such as `Http`. Separating the logic into a data service encapsulates this logic in a single place hiding the implementation from the outside consumers (perhaps a component), also making it easier to change the implementation.

[Back to top](#)

## Lifecycle Hooks

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

[Back to top](#)

## Implement Lifecycle Hooks Interfaces

### Style 09-01

**Do** implement the lifecycle hook interfaces.

**Why?** We get strong typing for the method signatures. The compiler and editor can call our attention to misspellings.

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

```
1. /* avoid */
2.
3. @Component({
4.   selector: 'toh-hero-button',
5.   template: `<button>OK</button>`
6. })
7. export class HeroButtonComponent {
8.   ngOnInit() { // misspelled
9.     console.log('The component is initialized');
10.  }
11. }
```



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

```
1. @Component({
2.   selector: 'toh-hero-button',
3.   template: `<button>OK</button>`
4. })
5. export class HeroButtonComponent implements OnInit {
6.   ngOnInit() {
7.     console.log('The component is initialized');
8.   }
9. }
```



[Back to top](#)

## Routing

Client-side routing is important for creating a navigation flow between a component tree hierarchy, and composing components that are made of many other child components.

[Back to top](#)

## Component Router

### Style 10-01

**Do** separate route configuration into a routing component file, also known as a component router.

**Do** use a `<router-outlet>` in the component router, where the routes will have their component targets display their templates.

**Do** focus the logic in the component router to the routing aspects and its target components.

**Do** extract other logic to services and other components.

**Why?** A component that handles routing is known as the component router, thus this follows the Angular 2 routing pattern.

**Why?** The `<router-outlet>` indicates where the template should be displayed for the target route.

app/app.component.ts

```
1. import { Component } from '@angular/core';
2. import { RouteConfig, ROUTER_DIRECTIVES, ROUTER_PROVIDERS } from
   '@angular/router';
3.
4. import { NavComponent } from '../shared/nav/nav.component';
5. import { DashboardComponent } from '../dashboard/dashboard.component';
6. import { HeroesComponent } from '../heroes/heroes.component';
7. import { HeroService } from '../heroes/shared/hero.service';
8.
9. @Component({
10.   selector: 'toh-app',
11.   templateUrl: 'app/app.component.html',
12.   styleUrls: ['app/app.component.css'],
13.   directives: [ROUTER_DIRECTIVES, NavComponent],
14.   providers: [
15.     ROUTER_PROVIDERS,
16.     HeroService
17.   ]
18. })
19. @RouteConfig([
20.   { path: '/dashboard', name: 'Dashboard', component: DashboardComponent,
     useAsDefault: true },
21.   { path: '/heroes/...', name: 'Heroes', component: HeroesComponent },
22. ])
23. export class AppComponent {}
```



[Back to top](#)

## Appendix

Useful tools and tips for Angular 2.

[Back to top](#)

## Codelyzer

### Style A-01

**Do** use [codelyzer](#) to follow this guide.

**Consider** adjusting the rules in codelyzer to suit your needs.

[Back to top](#)