STYLE GUIDE

> Write Angular with style.

Welcome to the Angular Style Guide

Purpose

Looking for an opinionated guide to Angular syntax, conventions, and application structure? Step right in! This style guide presents our 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 unmistakeable red header.

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 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. App structure and Angular modules
5. Components
6. Directives
7. Services
8. Data services
9. Lifecycle hooks
10. Appendix

Single responsibility

Apply the *Single Responsibility Principle* (SPR) 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

COPY CODE

```
/* avoid */

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { BrowserModule } from '@angular/platform-browser';
import { NgModule, Component, OnInit } from '@angular/core';

class Hero {
  id: number;
  name: string;
}

@Component({
  selector: 'my-app',
```

```
14.   template: `
15.       <h1>{{title}}</h1>
16.       <pre>{{heroes | json}}</pre>
17.     `,
18.   styleUrls: ['app/app.component.css']
19. })
20. class AppComponent implements OnInit {
21.   title = 'Tour of Heroes';
22.
23.   heroes: Hero[] = [];
24.
25.   ngOnInit() {
26.     getHeroes().then(heroes => this.heroes = heroes);
27.   }
28. }
29.
30. @NgModule({
31.   imports: [ BrowserModule ],
32.   declarations: [ AppComponent ],
33.   exports: [ AppComponent ],
34.   bootstrap: [ AppComponent ]
35. })
36. export class AppModule { }
37.
38. platformBrowserDynamic().bootstrapModule(AppModule);
39.
40. const HEROES: Hero[] = [
41.   {id: 1, name: 'Bombasto'},
42.   {id: 2, name: 'Tornado'},
43.   {id: 3, name: 'Magneta'},
44. ];
45.
46. function getHeroes(): Promise<Hero[]> {
47.   return Promise.resolve(HEROES); // TODO: get hero data from the
    server;
48. }
```

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

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

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

platformBrowserDynamic().bootstrapModule(AppModule);
```

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 find desited 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 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

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

`@Component({ ... })`
`export class`
`AppComponent { }`
	app.component.ts

`@Component({ ... })`
`export class`
`HeroesComponent { }`
	heroes.component.ts

`@Component({ ... })`
`export class`
`HeroListComponent { }`
	hero-list.component.ts

`@Component({ ... })`
`export class`
`HeroDetailComponent {`
`}`
	hero-detail.component.ts

	validation.directive.ts

```
@Directive({ ... })
export class
ValidationDirective {
}
```

```
@NgModule({ ... })
export class AppModule
```

app.module.ts

```
@Pipe({ name:
'initCaps' })
export class
InitCapsPipe
implements
PipeTransform { }
```

init-caps.pipe.ts

```
@Injectable()
export class
UserProfileService { }
```

user-profile.service.ts

Back to top

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

`@Injectable()`
`export class`
`HeroDataService { }`
	hero-data.service.ts

`@Injectable()`
`export class`
`CreditService { }`
	credit.service.ts

`@Injectable()`
`export class Logger {`
`}`
	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`.

> **Do** include error handling in the bootstrapping logic.

> **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.

main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

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

platformBrowserDynamic().bootstrapModule(AppModule)
  .then(success => console.log(`Bootstrap success`))
  .catch(err => console.error(err));
```

Back to top

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 will recognize lower camel case.

Back to top

Custom prefix for components

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 **T**our **o**f **H**eroes 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

```
/* avoid */

// HeroComponent is in the Tour of Heroes feature
@Component({
  selector: 'hero'
})
export class HeroComponent {}
```

app/users/users.component.ts

```
/* avoid */

// UsersComponent is in an Admin feature
@Component({
  selector: 'users'
})
export class UsersComponent {}
```

app/heroes/hero.component.ts

```
@Component({
  selector: 'toh-hero'
})
export class HeroComponent {}
```

app/users/users.component.ts

```
@Component({
  selector: 'admin-users'
})
export class UsersComponent {}
```

Custom prefix for directives

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

```
/* avoid */

@Directive({
  selector: '[validate]'
})
export class ValidateDirective {}
```

app/shared/validate.directive.ts

```
@Directive({
  selector: '[tohValidate]'
})
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
`@Pipe({ name: 'ellipsis' }) export class EllipsisPipe implements PipeTransform { }`	ellipsis.pipe.ts
`@Pipe({ name: 'initCaps' }) export class InitCapsPipe implements PipeTransform { }`	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 (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.

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

Back to top

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
`@NgModule({ ... }) export class AppModule { }`	app.module.ts
`@NgModule({ ... }) export class HeroesModule { }`	heroes.module.ts
`@NgModule({ ... }) export class VillainsModule { }`	villains.module.ts
`@NgModule({ ... }) export class AppRoutingModule { }`	app-routing.module.ts
`@NgModule({ ... }) export class HeroesRoutingModule { }`	heroes-routing.module.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. By convention, upper camel case indicates a constructable asset.

app/shared/exception.service.ts

```
/* avoid */

export class exceptionService {
  constructor() { }
}
```

app/shared/exception.service.ts

```
export class ExceptionService {
  constructor() { }
}
```

Back to top

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 itself 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

```
export const mockHeroes   = ['Sam', 'Jill']; // prefer
export const heroesUrl    = 'api/heroes';    // prefer
export const VILLAINS_URL = 'api/villains';  // tolerate
```

Back to top

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.

> **Why?** TypeScript 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

```
/* avoid */

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

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

@Injectable()
export class HeroCollectorService {
  hero: IHero;

  constructor() { }
}
```

app/shared/hero-collector.service.ts

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

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

@Injectable()
export class HeroCollectorService {
  hero: Hero;

  constructor() { }
}
```

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.

app/shared/toast.service.ts

```
/* avoid */

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

```

```
@Injectable()
export class ToastService {
  message: string;

  private _toastCount: number;

  hide() {
    this._toastCount--;
    this._log();
  }

  show() {
    this._toastCount++;
    this._log();
  }

  private _log() {
    console.log(this.message);
  }
}
```

app/shared/toast.service.ts

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

@Injectable()
export class ToastService {
  message: string;

  private toastCount: number;

  hide() {
    this.toastCount--;
    this.log();
  }

```

```
    show() {
      this.toastCount++;
      this.log();
    }

    private log() {
      console.log(this.message);
    }
  }
```

Back to top

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

```
/* avoid */

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

Back to top

Application structure and Angular modules

Have a near-term view of implementation and a long-term vision. Start small but keep in mind where the app is heading down the road.

All of the app's code goes in a folder named `src`. All feature areas are in their own folder, with their own Angular module.

All content is one asset per file. Each component, service, and pipe is in its own file. All third party vendor scripts are stored in another folder and not in the `src` folder. You didn't write them and you don't want them cluttering `src`. Use the naming conventions for files in this guide.

Back to top

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*?

Back to top

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.

Back to top

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.

Back to top

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 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.

Back to top

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 component, `hero-view.component.html` because a component is obviously a view. But if something is not obvious or departs from a convention, then spell it out.

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 `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.

Back to top

Folders-by-feature structure

STYLE 04-07

> **Do** create folders named for the feature area 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 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 (e.g. 10+), locating them is easier with a consistent folder structure and more difficult in a flat structure.

> **Do** create an Angular module for each feature area.

> **Why?** Angular modules make it easy to lazy load routable features.

> **Why?** Angular modules make it easier to isolate, test, and re-use features.

Refer to this _folder and file structure_ example.

Back to top

App *root module*

STYLE 04-08

> **Do** create an Angular module in the app's root folder (e.g., in `/src/app`).

> **Why?** Every app requires at least one root Angular module.

> **Consider** naming the root module `app.module.ts`.

> **Why?** Makes it easier to locate and identify the root module.

app/app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent }    from './app.component';
import { HeroesComponent } from './heroes/heroes.component';

@NgModule({
  imports: [
    BrowserModule,
  ],
  declarations: [
    AppComponent,
    HeroesComponent
  ],
  exports: [ AppComponent ],
  entryComponents: [ AppComponent ]
})
export class AppModule {}
```

Back to top

Feature modules

STYLE 04-09

> **Do** create an Angular module for all distinct features in an application (e.g. `Heroes` feature).

> **Do** place the feature module in the same named folder as the feature area (.e.g `app/heroes`).

Do name the feature module file reflecting the name of the feature area and folder (e.g. `app/heroes/heroes.module.ts`)

Do name the feature module symbol reflecting the name of the feature area, folder, and file (e.g. `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.

Back to top

Shared feature module

STYLE 04-10

Do create a feature module named `SharedModule` in a `shared` folder (e.g. `app/shared/shared.module.ts` defines `SharedModule`).

Do declare components, directives, and pipes in a shared module when those items will be re-used and referenced by the components declared in other feature modules.

Consider using the name SharedModule, when the contents of a shared

module are referenced across the entire application.

Do not provide services in shared modules. Services are usually singletons that are provided once for the entire application or in a particular feature module.

Do import all modules required by the assets in the `SharedModule` (e.g. `CommonModule` and `FormsModule`).

Why? `SharedModule` will contain components, directives and pipes that may need features from another common module (e.g. `ngFor` in `CommonModule`).

Do declare all components, directives, and pipes in the `SharedModule`.

Do export all symbols from the `SharedModule` that other feature modules need to use.

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 undesireable 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
  ...
```

```
import { NgModule }      from '@angular/core';
import { CommonModule }  from '@angular/common';
import { FormsModule }   from '@angular/forms';

import { FilterTextComponent } from './filter-text/filter-text.component';
import { FilterTextService }   from './filter-text/filter-text.service';
import { InitCapsPipe }        from './init-caps.pipe';

@NgModule({
  imports: [CommonModule, FormsModule],
  declarations: [
    FilterTextComponent,
    InitCapsPipe
  ],
  providers: [FilterTextService],
  exports: [
    CommonModule,
    FormsModule,
    FilterTextComponent,
    InitCapsPipe
  ]
})
```

```
23.   export class SharedModule { }
```

Back to top

Core feature module

STYLE 04-11

> **Consider** collecting numerous, auxiliary, single-use classes inside a core module to simplify the apparent structure of a feature module.

> **Consider** calling the application-wide core module, `CoreModule`. Importing `CoreModule` into the root `AppModule` reduces its complexity and emphasizes its role as orchestrator of the application as a whole.

> **Do** create a feature module named `CoreModule` in a `core` folder (e.g. `app/core/core.module.ts` defines `CoreModule`).

> **Do** put a singleton service whose instance wil be shared throughout the application in the `CoreModule` (e.g. `ExceptionService` and `LoggerService`).

> **Do** import all modules required by the assets in the `CoreModule` (e.g. `CommonModule` and `FormsModule`).

> **Why?** `CoreModule` provides one or more singleton services. Angular registers the providers with the app root injector, making a singleton instance of each service available to any component that needs them, whether that component is eagerly or lazily loaded.

> **Why?** `CoreModule` will contain singleton services. When a lazy loaded module imports these, it will get a new instance and not the intended app-wide singleton.

Do gather application-wide, single use components in the `CoreModule`. Import it once (in the `AppModule`) when the app starts and never import it anywhere else. (e.g. `NavComponent` and `SpinnerComponent`).

Why? Real world apps can have several single-use components (e.g., spinners, message toasts, and modal dialogs) that appear only in the `AppComponent` template. They are not imported elsewhere so they're not shared in that sense. Yet they're too big and messy to leave loose in the root folder.

Avoid importing the `CoreModule` anywhere except in the `AppModule`.

Why? A lazily loaded feature module that directly imports the `CoreModule` will make its own copy of services and likely have undesireable results.

Why? An eagerly loaded feature module already has access to the `AppModule`'s injector, and thus the `CoreModule`'s services.

Do export all symbols from the `CoreModule` that the `AppModule` will import and make available for other feature modules to use.

Why? `CoreModule` exists to make commonly used singleton services available for use in the many other modules.

Why? You want the entire app to use the one, singleton instance. You don't want each module to have its own separate instance of singleton services. Yet there is a real danger of that happening accidentally if the `CoreModule` provides a service.

```
src
└─app
  └─core
    ├─core.module.ts
    ├─logger.service.ts|spec.ts
    └─nav
      └─nav.component.ts|html|css|spec.ts
```

```
    spinner
      spinner.component.ts|html|css|spec.ts
      spinner.service.ts|spec.ts
    app.component.ts|html|css|spec.ts
    app.module.ts
    app-routing.module.ts
  main.ts
  index.html
...
```

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent }   from './app.component';
import { HeroesComponent } from './heroes/heroes.component';
import { CoreModule }    from './core/core.module';

@NgModule({
  imports: [
    BrowserModule,
    CoreModule,
  ],
  declarations: [
    AppComponent,
    HeroesComponent
  ],
  exports: [ AppComponent ],
  entryComponents: [ AppComponent ]
})
export class AppModule {}
```

> `AppModule` is a little smaller because many app/root classes have moved to other modules. `AppModule` is stable because you will add future components and providers to other modules, not this one. `AppModule` delegates to imported modules rather than doing work. `AppModule` is focused on its main task, orchestrating the app as a whole.

Back to top

Prevent re-import of the core module

STYLE 04-12

Only the root `AppModule` should import the `CoreModule`.

> **Do** guard against reimporting of `CoreModule` and fail fast by adding guard logic.

> **Why?** Guards against reimporting of the `CoreModule`.

> **Why?** Guards against creating multiple instances of assets intended to be singletons.

```
export function throwIfAlreadyLoaded(parentModule: any, moduleName: string) {
  if (parentModule) {
    throw new Error(`${moduleName} has already been loaded. Import Core modules in the AppModule only.`);
  }
}
```

Back to top

Lazy Loaded folders

STYLE 04-13

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

Never directly import lazy loaded folders

STYLE 04-14

> **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.

Back to top

Components

Component selector names

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.

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

```
/* avoid */

@Component({
  moduleId: module.id,
  selector: 'tohHeroButton',
  templateUrl: './hero-button.component.html'
})
export class HeroButtonComponent {}
```

```
@Component({
  moduleId: module.id,
  selector: 'toh-hero-button',
  templateUrl: './hero-button.component.html'
})
export class HeroButtonComponent {}
```

Back to top

Components as elements

STYLE 05-03

Do give 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 WebComponents.

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

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

```
/* avoid */

@Component({
  moduleId: module.id,
  selector: '[tohHeroButton]',
  templateUrl: './hero-button.component.html'
})
export class HeroButtonComponent {}
```

app/app.component.html

```
<!-- avoid -->

<div tohHeroButton></div>
```

```
@Component({
  moduleId: module.id,
  selector: 'toh-hero-button',
  templateUrl: './hero-button.component.html'
```

```
5.  })
6.  export class HeroButtonComponent {}
```

Back to top

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 `./`, and add `moduleId: module.id` to the component metadata.

> **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 JIT compiler requires the `moduleId` for relative URLs; the AOT compiler, which doesn't need it, safely ignores this property.

> **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

```
/* avoid */

@Component({
  selector: 'toh-heroes',
  template: `
    <div>
      <h2>My Heroes</h2>
      <ul class="heroes">
        <li *ngFor="let hero of heroes | async"
(click)="selectedHero=hero">
          <span class="badge">{{hero.id}}</span> {{hero.name}}
        </li>
      </ul>
      <div *ngIf="selectedHero">
        <h2>{{selectedHero.name | uppercase}} is my hero</h2>
      </div>
    </div>
  `,
  styles: [`
    .heroes {
      margin: 0 0 2em 0; list-style-type: none; padding: 0; width:
15em;
    }
    .heroes li {
      cursor: pointer;
      position: relative;
      left: 0;
      background-color: #EEE;
      margin: .5em;
      padding: .3em 0;
      height: 1.6em;
      border-radius: 4px;
    }
    .heroes .badge {
```

```
        display: inline-block;
        font-size: small;
        color: white;
        padding: 0.8em 0.7em 0 0.7em;
        background-color: #607D8B;
        line-height: 1em;
        position: relative;
        left: -1px;
        top: -4px;
        height: 1.8em;
        margin-right: .8em;
        border-radius: 4px 0 0 4px;
      }
    `]
})
export class HeroesComponent implements OnInit {
  heroes: Observable<Hero[]>;
  selectedHero: Hero;

 constructor(private heroService: HeroService) { }

  ngOnInit() {
    this.heroes = this.heroService.getHeroes();
  }
}
```

```
@Component({
  moduleId: module.id,
  selector: 'toh-heroes',
  templateUrl: './heroes.component.html',
  styleUrls:  ['./heroes.component.css']
})
export class HeroesComponent implements OnInit {
  heroes: Observable<Hero[]>;
  selectedHero: Hero;
```

```
10.
11.     constructor(private heroService: HeroService) { }
12.
13.     ngOnInit() {
14.       this.heroes = this.heroService.getHeroes();
15.     }
16.   }
```

Back to top

Decorate *input* and *output* properties

STYLE 05-12

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

> **Consider** placing `@Input()` or `@Output()` on the same line as the property it decorates.

> **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` or `@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

```
/* avoid */

@Component({
  selector: 'toh-hero-button',
  template: `<button></button>`,
  inputs: [
    'label'
  ],
  outputs: [
    'change'
  ]
})
export class HeroButtonComponent {
  change = new EventEmitter<any>();
  label: string;
}
```

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

```
@Component({
  selector: 'toh-hero-button',
  template: `<button>{{label}}</button>`
})
export class HeroButtonComponent {
  @Output() change = new EventEmitter<any>();
  @Input() label: string;
}
```

Back to top

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

```
/* avoid pointless aliasing */

@Component({
  selector: 'toh-hero-button',
  template: `<button>{{label}}</button>`
})
export class HeroButtonComponent {
  // Pointless aliases
  @Output('changeEvent') change = new EventEmitter<any>();
  @Input('labelAttribute') label: string;
}
```

app/app.component.html

```
<!-- avoid -->

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

```
@Component({
```

```
  selector: 'toh-hero-button',
  template: `<button>{{label}}</button>`
})
export class HeroButtonComponent {
  // No aliases
  @Output() change = new EventEmitter<any>();
  @Input() label: string;
}
```

Back to top

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

```
/* avoid */

export class ToastComponent implements OnInit {

  private defaults = {
    title: '',
    message: 'May the Force be with you'
  };
  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.   }
```

app/shared/toast/toast.component.ts

```
1.   export class ToastComponent implements OnInit {
2.     // public properties
3.     message: string;
4.     title: string;
5.
```

```
    // private fields
    private defaults = {
      title: '',
      message: 'May the Force be with you'
    };
    private toastElement: any;

    // public methods
    activate(message = this.defaults.message, title =
  this.defaults.title) {
      this.title = title;
      this.message = message;
      this.show();
    }

    ngOnInit() {
      this.toastElement = document.getElementById('toh-toast');
    }

    // private methods
    private hide() {
      this.toastElement.style.opacity = 0;
      window.setTimeout(() => this.toastElement.style.zIndex = 0, 400);
    }

    private show() {
      console.log(this.message);
      this.toastElement.style.opacity = 1;
      this.toastElement.style.zIndex = 9999;
      window.setTimeout(() => this.hide(), 2500);
    }
  }
```

Back to top

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

```
/* avoid */

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

import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/catch';
import 'rxjs/add/operator/finally';
import 'rxjs/add/operator/map';

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

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

```

```
15.  export class HeroListComponent implements OnInit {
16.    heroes: Hero[];
17.    constructor(private http: Http) {}
18.    getHeroes() {
19.      this.heroes = [];
20.      this.http.get(heroesUrl)
21.        .map((response: Response) => <Hero[]>response.json().data)
22.        .catch(this.catchBadResponse)
23.        .finally(() => this.hideSpinner())
24.        .subscribe((heroes: Hero[]) => this.heroes = heroes);
25.    }
26.    ngOnInit() {
27.      this.getHeroes();
28.    }
29.
30.    private catchBadResponse(err: any, source: Observable<any>) {
31.      // log and handle the exception
32.      return new Observable();
33.    }
34.
35.    private hideSpinner() {
36.      // hide the spinner
37.    }
38.  }
```

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

```
1.  import { Component, OnInit } from '@angular/core';
2.
3.  import { Hero, HeroService } from '../shared';
4.
5.  @Component({
6.    selector: 'toh-hero-list',
7.    template: `...`
8.  })
9.  export class HeroListComponent implements OnInit {
```

```
    heroes: Hero[];
    constructor(private heroService: HeroService) {}
    getHeroes() {
      this.heroes = [];
      this.heroService.getHeroes()
        .subscribe(heroes => this.heroes = heroes);
    }
    ngOnInit() {
      this.getHeroes();
    }
  }
```

Back to top

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

app/heroes/hero.component.ts

```
/* avoid */

@Component({
  selector: 'toh-hero',
```

```
    template: `...`
  })
  export class HeroComponent {
    @Output() onSavedTheDay = new EventEmitter<boolean>();
  }
```

app/app.component.html

```
<!-- avoid -->

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

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

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.

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

```
/* avoid */

@Component({
  selector: 'toh-hero-list',
  template: `
    <section>
      Our list of heroes:
      <hero-profile *ngFor="let hero of heroes" [hero]="hero">
      </hero-profile>
      Total powers: {{totalPowers}}<br>
      Average power: {{totalPowers / heroes.length}}
    </section>
  `
})
export class HeroListComponent {
  heroes: Hero[];
  totalPowers: number;
}
```

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

```
@Component({
  selector: 'toh-hero-list',
  template: `
    <section>
      Our list of heroes:
      <toh-hero *ngFor="let hero of heroes" [hero]="hero">
      </toh-hero>
      Total powers: {{totalPowers}}<br>
      Average power: {{avgPower}}
    </section>
  `
})
```

```
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 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() {
```

```
      // do highlight work
    }
  }
```

app/app.component.html

```
<div tohHighlight>Bombasta</div>
```

Back to top

HostListener/HostBinding decorators versus *host* metadata

STYLE 06-03

> **Consider** preferring the `@HostListener` and `@HostBinding` to the `host` property of the `@Directive` and `@Component` decorators.

> **Do** be consistent in your choice.

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

app/shared/validator.directive.ts

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

@Directive({
  selector: '[tohValidator]'
})
```

```
export class ValidatorDirective {
  @HostBinding('attr.role') role = 'button';
  @HostListener('mouseenter') onMouseEnter() {
    // do work
  }
}
```

Compare with the less preferred `host` metadata alternative.

> **Why?** The `host` metadata is only one term to remember and doesn't require extra ES imports.

app/shared/validator2.directive.ts

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

@Directive({
  selector: '[tohValidator2]',
  host: {
    'attr.role': 'button',
    '(mouseenter)': 'onMouseEnter()'
  }
})
export class Validator2Directive {
  role = 'button';
  onMouseEnter() {
    // do work
  }
}
```

Back to top

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

```
export class HeroService {
  constructor(private http: Http) { }

  getHeroes() {
    return this.http.get('api/heroes')
      .map((response: Response) => <Hero[]>response.json().data);
  }
}
```

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 injector at the top-most component where they will be shared.

> **Why?** The Angular 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.

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

import { HeroService } from './heroes';

@Component({
  selector: 'toh-app',
  template: `
```

```
      <toh-heroes></toh-heroes>
    `,
  providers: [HeroService]
})
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 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()` syntax is much less verbose compared to using `@Inject()` on each individual constructor parameter.

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

```
/* avoid */

export class HeroArena {
  constructor(
      @Inject(HeroService) private heroService: HeroService,
      @Inject(Http) private http: Http) {}
}
```

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

```
@Injectable()
export class HeroArena {
  constructor(
    private heroService: HeroService,
    private http: Http) {}
}
```

Back to top

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.

Back to top

Lifecycle hooks

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

Back to top

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

```
/* avoid */

@Component({
  selector: 'toh-hero-button',
  template: `<button>OK<button>`
})
export class HeroButtonComponent {
  onInit() { // misspelled
```

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

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

```
@Component({
  selector: 'toh-hero-button',
  template: `<button>OK</button>`
})
export class HeroButtonComponent implements OnInit {
  ngOnInit() {
    console.log('The component is initialized');
  }
}
```

Back to top

Appendix

Useful tools and tips for Angular.

Back to top

Codelyzer

STYLE A-01

> **Do** use codelyzer to follow this guide.

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

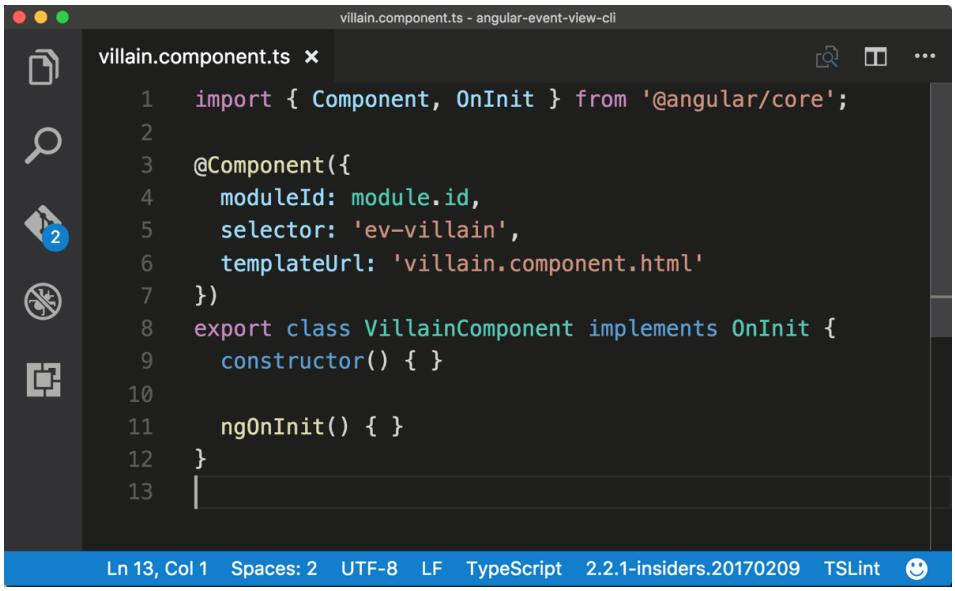
Back to top

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 for Visual Studio Code that follow these styles and guidelines.



> **Consider** using snippets for Atom that follow these styles and guidelines.

> **Consider** using snippets for Sublime Text that follow these styles and guidelines.

> **Consider** using snippets for Vim that follow these styles and guidelines.

Back to top