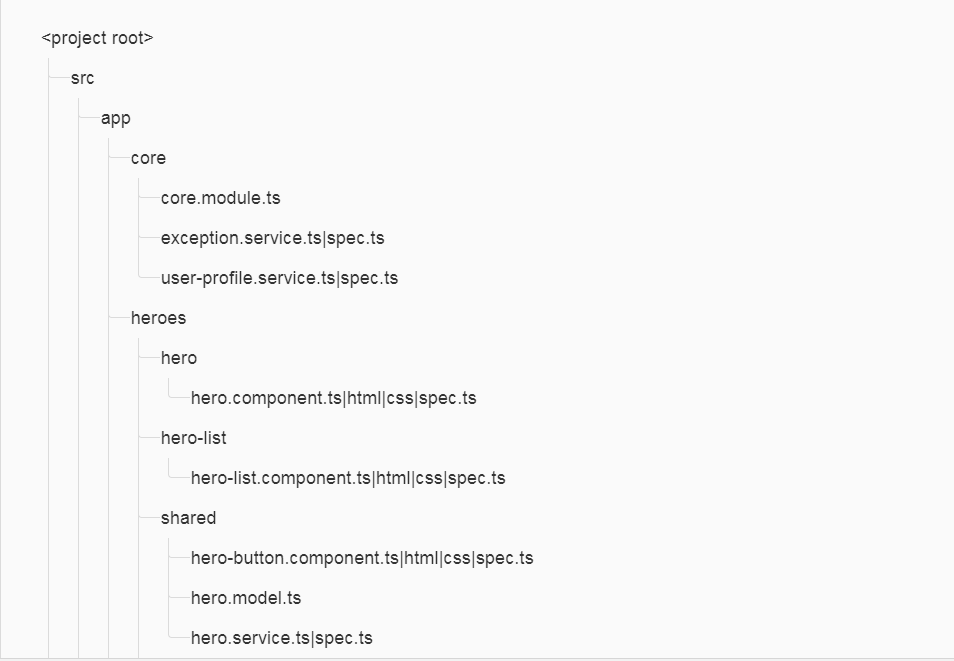
Table of Contents

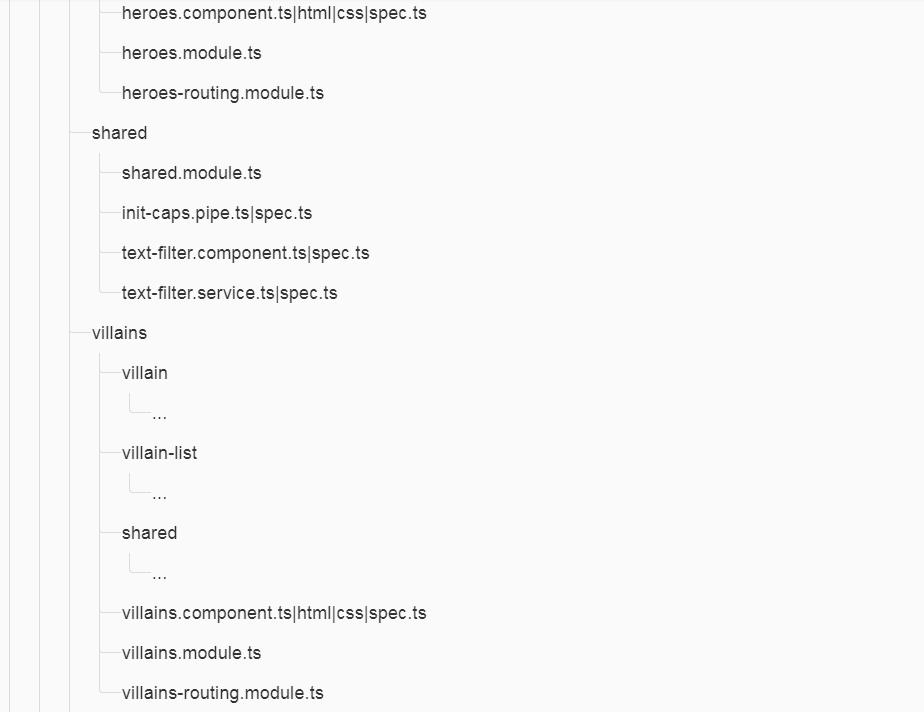
|  |  |
| --- | --- |
| Sl.No. | Title |
| 1. | File Structure conventions |
| 2. | Naming   1. General Naming Guidelines 2. Separate file names with dots and dashes 3. Symbols and file names 4. Service names 5. Bootstrapping 6. Directive selectors 7. Custom prefix for components 8. Pipe names 9. Unit test file names 10. End-to-End (E2E) test file names |
| 3. | Modules   1. App root module 2. Feature module 3. Shared feature module 4. Core feature module 5. Prevent re-import of the core module 6. Put presentation logic in the component class |
| 4. | Directives |
| 5. | Services   1. Services are singletons 2. Single responsibility 3. Providing a service |
| 6. | Data Services |
| 7. | References |

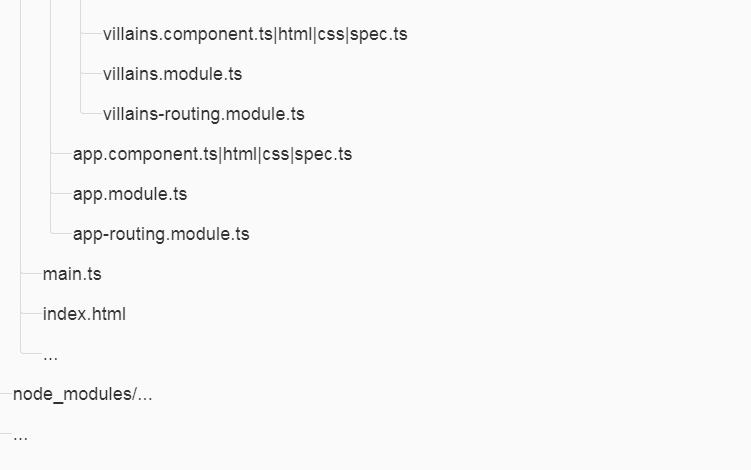
## File structure conventions

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

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







## 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**

**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.

Naming conventions help provide a consistent way to find content at a glance. Consistency within the project is vital. Consistency with a team is important. Consistency across a company provides tremendous efficiency.

The naming conventions should simply help find desired code faster and make it easier to understand.

Names of folders and files should clearly convey their intent. For example, app/heroes/hero-list.component.ts may contain a component that manages a list of heroes.

### **Separate file names with dots and dashes**

**Do** use dashes to separate words in the descriptive name.

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

**Do** use consistent type names for all components following a pattern that describes the component's feature then its type. A recommended pattern is feature.type.ts.

**Do** use conventional type names including .service, .component, .pipe, .module, and .directive. Invent additional type names if you must but take care not to create too many.

Type names provide a consistent way to quickly identify what is in the file.

 Type names make it easy to find a specific file type using an editor or IDE's fuzzy search techniques.

 Unabbreviated type names such as .service are descriptive and unambiguous. Abbreviations such as .srv, .svc, and .serv can be confusing.

 Type names provide pattern matching for any automated tasks.

### **Symbols and file names**

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

**Do** use upper camel case for class names.

**Do** match the name of the symbol to the name of the file.

**Do** append the symbol name with the conventional suffix (such as [Component](https://angular.io/api/core/Component), [Directive](https://angular.io/api/core/Directive), Module, [Pipe](https://angular.io/api/core/Pipe), or Service) for a thing of that type.

**Do** give the filename the conventional suffix (such as .component.ts, .directive.ts, .module.ts, .pipe.ts, or .service.ts) for a file of that type.

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

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

### **Service names**

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

**Do** suffix a service class name with Service. For example, something that gets data or heroes should be called a DataService or a HeroService.

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.

Provides a consistent way to quickly identify and reference services.

Clear service names such as Logger do not require a suffix.

Service names such as Credit are nouns and require a suffix and should be named with a suffix when it is not obvious if it is a service or something else.

|  |  |
| --- | --- |
| **Symbol Name** | **File Name** |
| @[Injectable](https://angular.io/api/core/Injectable)()  export class HeroDataService { } | hero-data.service.ts |
| @[Injectable](https://angular.io/api/core/Injectable)()  export class CreditService { } | credit.service.ts |
| @[Injectable](https://angular.io/api/core/Injectable)()  export class Logger { } | logger.service.ts |

### **Bootstrapping**

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

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

**Avoid** putting app logic in main.ts. Instead, consider placing it in a component or service.

main.ts

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

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

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

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

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

### **Directive selectors**

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

Keeps the names of the properties defined in the directives that are bound to the view consistent with the attribute names.

The Angular HTML parser is case sensitive and recognizes lower camel case.

### **Custom prefix for components**

**Do** use a hyphenated, lowercase element selector value (e.g. admin-users).

**Do** use a custom prefix for a component selector. For example, the prefix toh represents from **T**our **o**f **H**eroes and the prefix adminrepresents an admin feature area.

**Do** use a prefix that identifies the feature area or the app itself.

Prevents element name collisions with components in other apps and with native HTML elements.

Makes it easier to promote and share the component in other apps.

Components are easy to identify in the DOM.

app/heroes/hero.component.ts

/\* avoid \*/

// HeroComponent is in the Tour of Heroes feature

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

selector: 'hero'

})

export class HeroComponent {}

app/users/users.component.ts

/\* avoid \*/

// UsersComponent is in an Admin feature

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

selector: 'users'

})

export class UsersComponent {}

app/heroes/hero.component.ts

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

selector: 'toh-hero'

})

export class HeroComponent {}

app/users/users.component.ts

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

selector: 'admin-users'

})

export class UsersComponent {}

### **Pipe names**

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

Provides a consistent way to quickly identify and reference pipes.

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

### **Unit test file names**

**Do** name test specification files the same as the component they test.

**Do** name test specification files with a suffix of .spec.

Provides a consistent way to quickly identify tests.

Provides pattern matching for [karma](http://karma-runner.github.io/) or other test runners.

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

### **End-to-End (E2E) test file names**

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

Provides a consistent way to quickly identify end-to-end tests.

 Provides pattern matching for test runners and build automation.

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

## Modules

### **App root module**

**Do** create an NgModule in the app's root folder, for example, in /src/app.

Every app requires at least one root NgModule.

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

Makes it easier to locate and identify the root module.

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

### **Feature module**

**Do** create an NgModule for all distinct features in an application; for example, a Heroes feature.

**Do** place the feature module in the same named folder as the feature area; for example, in app/heroes.

**Do** name the feature module file reflecting the name of the feature area and folder; for example, app/heroes/heroes.module.ts.

**Do** name the feature module symbol reflecting the name of the feature area, folder, and file; for example, app/heroes/heroes.module.tsdefines HeroesModule.

A feature module can expose or hide its implementation from other modules.

A feature module identifies distinct sets of related components that comprise the feature area.

A feature module can easily be routed to both eagerly and lazily.

A feature module defines clear boundaries between specific functionality and other application features.

A feature module helps clarify and make it easier to assign development responsibilities to different teams.

A feature module can easily be isolated for testing.

A feature module helps clarify and make it easier to assign development responsibilities to different teams.

A feature module can easily be isolated for testing.

### **Shared feature module**

**Do** create a feature module named SharedModule in a shared folder; for example, app/shared/shared.module.ts defines SharedModule.

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

**Consider** using the name SharedModule when the contents of a shared module are referenced across the entire application.

**Avoid** providing services in shared modules. Services are usually singletons that are provided once for the entire application or in a particular feature module.

**Do** import all modules required by the assets in the SharedModule; for example, [CommonModule](https://angular.io/api/common/CommonModule) and [FormsModule](https://angular.io/api/forms/FormsModule).

 SharedModule will contain components, directives and pipes that may need features from another common module; for example,[ngFor](https://angular.io/api/common/NgForOf) in [CommonModule](https://angular.io/api/common/CommonModule).

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

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

 SharedModule exists to make commonly used components, directives and pipes available for use in the templates of components in many other modules.

**Avoid** specifying app-wide singleton providers in a SharedModule. Intentional singletons are OK. Take care.

 A lazy loaded feature module that imports that shared module will make its own copy of the service and likely have undesirable results.

 You don't want each module to have its own separate instance of singleton services. Yet there is a real danger of that happening if the SharedModule provides a service.

1. import { [NgModule](https://angular.io/api/core/NgModule) } from '@angular/core';
2. import { [CommonModule](https://angular.io/api/common/CommonModule) } from '@angular/common';
3. import { [FormsModule](https://angular.io/api/forms/FormsModule) } from '@angular/forms';
5. import { FilterTextComponent } from './filter-text/filter-text.component';
6. import { FilterTextService } from './filter-text/filter-text.service';
7. import { InitCapsPipe } from './init-caps.pipe';
9. @[NgModule](https://angular.io/api/core/NgModule)({
10. imports: [[CommonModule](https://angular.io/api/common/CommonModule), [FormsModule](https://angular.io/api/forms/FormsModule)],
11. declarations: [
12. FilterTextComponent,
13. InitCapsPipe
14. ],
15. providers: [FilterTextService],
16. exports: [
17. [CommonModule](https://angular.io/api/common/CommonModule),
18. [FormsModule](https://angular.io/api/forms/FormsModule),
19. FilterTextComponent,
20. InitCapsPipe
21. ]
22. })
23. export class SharedModule { }

### **Core feature module**

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

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

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

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

**Do** import all modules required by the assets in the CoreModule (e.g. [CommonModule](https://angular.io/api/common/CommonModule) and [FormsModule](https://angular.io/api/forms/FormsModule)).

 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.

 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).

 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.

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

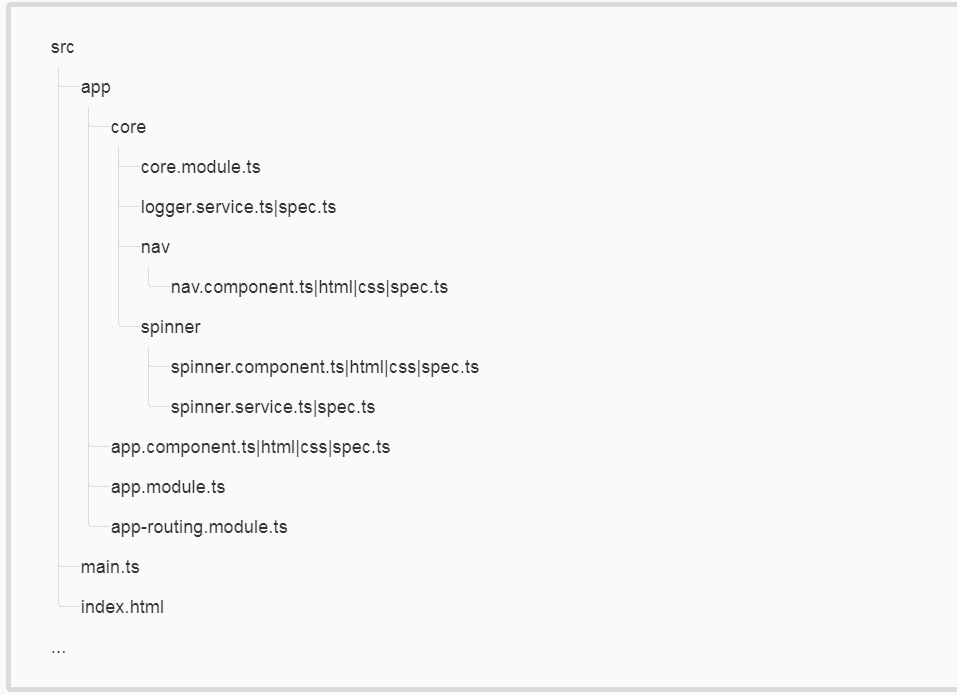
 An eagerly loaded feature module already has access to the AppModule's injector, and thus the CoreModule's services.

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

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

 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.

1. import { [NgModule](https://angular.io/api/core/NgModule) } from '@angular/core';
2. import { [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) } from '@angular/platform-browser';
4. import { AppComponent } from './app.component';
5. import { HeroesComponent } from './heroes/heroes.component';
6. import { CoreModule } from './core/core.module';
8. @[NgModule](https://angular.io/api/core/NgModule)({
9. imports: [
10. [BrowserModule](https://angular.io/api/platform-browser/BrowserModule),
11. CoreModule,
12. ],
13. declarations: [
14. AppComponent,
15. HeroesComponent
16. ],
17. exports: [ AppComponent ],
18. entryComponents: [ AppComponent ]
19. })
20. export class AppModule {}



### **Prevent re-import of the core module**

Only the root AppModule should import the CoreModule.

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

 Guards against reimporting of the CoreModule.

 Guards against creating multiple instances of assets intended to be singletons.

export function throwIfAlreadyLoaded(parentModule: any, moduleName: string) {

if (parentModule) {

throw new Error(`${moduleName} has already been loaded. Import Core modules in the AppModule only.`);

}

}

1. import { [NgModule](https://angular.io/api/core/NgModule), [Optional](https://angular.io/api/core/Optional), [SkipSelf](https://angular.io/api/core/SkipSelf) } from '@angular/core';
2. import { [CommonModule](https://angular.io/api/common/CommonModule) } from '@angular/common';
4. import { LoggerService } from './logger.service';
5. import { NavComponent } from './nav/nav.component';
6. import { throwIfAlreadyLoaded } from './module-import-guard';
8. @[NgModule](https://angular.io/api/core/NgModule)({
9. imports: [
10. [CommonModule](https://angular.io/api/common/CommonModule) // we use [ngFor](https://angular.io/api/common/NgForOf)
11. ],
12. exports: [NavComponent],
13. declarations: [NavComponent],
14. providers: [LoggerService]
15. })
16. export class CoreModule {
17. constructor( @[Optional](https://angular.io/api/core/Optional)() @[SkipSelf](https://angular.io/api/core/SkipSelf)() parentModule: CoreModule) {
18. throwIfAlreadyLoaded(parentModule, 'CoreModule');
19. }
20. }

### **Put presentation logic in the component class**

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

 Logic will be contained in one place (the component class) instead of being spread in two places.

 Keeping the component's presentation logic in the class instead of the template improves testability, maintainability, and reusability.

/\* avoid \*/

@Component({

selector: 'toh-hero-list',

template: `

<section>

Our list of heroes:

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

</hero-profile>

Total powers: {{totalPowers}}<br>

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

</section>

`

})

export class HeroListComponent {

heroes: Hero[];

totalPowers: number;

}

## Directives

### **Use directives to enhance an element**

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

 Attribute directives don't have an associated template.

 An element may have more than one attribute directive applied.

@Directive({

selector: '[tohHighlight]'

})

export class HighlightDirective {

@HostListener('mouseover') onMouseEnter() {

// do highlight work

}

}

## Services

### **Services are singletons**

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

 Services are ideal for sharing methods across a feature area or an app.

 Services are ideal for sharing stateful in-memory data.

app/heroes/shared/hero.service.ts

export class HeroService {

### **Single responsibility**

**Do** create services with a single responsibility that is encapsulated by its context.

**Do** create a new service once the service begins to exceed that singular purpose.

 When a service has multiple responsibilities, it becomes difficult to test.

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

### **Providing a service**

**Do** provide a service with the app root injector in the @[Injectable](https://angular.io/api/core/Injectable) decorator of the service.

 The Angular injector is hierarchical.

 When you provide the service to a root injector, that instance of the service is shared and available in every class that needs the service. This is ideal when a service is sharing methods or state.

 When you register a service in the @[Injectable](https://angular.io/api/core/Injectable) decorator of the service, optimization tools such as those used by the CLI's production builds can perform tree shaking and remove services that aren't used by your app.

 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.

/\* avoid \*/

export class HeroArena {

constructor(

@Inject(HeroService) private heroService: HeroService,

@Inject(Http) private http: Http) {}

}

## Data Services

### **Talk to the server through a service**

**Do** refactor logic for making data operations and interacting with data to a service.

**Do** make data services responsible for XHR calls, local storage, stashing in memory, or any other data operations.

 The component's responsibility is for the presentation and gathering of information for the view. It should not care how it gets the data, just that it knows who to ask for it. Separating the data services moves the logic on how to get it to the data service, and lets the component be simpler and more focused on the view.

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

 The details of data management, such as headers, HTTP methods, caching, error handling, and retry logic, are irrelevant to components and other data consumers.

A data service encapsulates these details. It's easier to evolve these details inside the service without affecting its consumers. And it's easier to test the consumers with mock service implementations.

## References

### https://angular.io/guide/styleguide