Angular Development - Best Practices

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. Aim to be consistent.

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

Why?
Gives reasons for following the previous recommendations.

File structure conventions

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

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

Single responsibility

Apply the single responsibility principle (SRP) to all components, services, and other symbols. This helps make the application 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

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

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

@Component({
  selector: 'app-root',
  template: `
      <h1>{{title}}</h1>
      <pre>{{heroes | json}}</pre>
    `,
  styleUrls: ['../app.component.css'],
})
export class AppComponent implements OnInit {
  title = 'Tour of Heroes';

  heroes: Hero[] = [];

  ngOnInit() {
    getHeroes().then((heroes) => (this.heroes = heroes));
  }
}

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

platformBrowserDynamic().bootstrapModule(AppModule);

const HEROES: Hero[] = [
  {id: 1, name: 'Bombasto'},
  {id: 2, name: 'Tornado'},
  {id: 3, name: 'Magneta'},
];

function getHeroes(): Promise<Hero[]> {
  return Promise.resolve(HEROES); // TODO: get hero data from the server;
}
```

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 application grows, this rule becomes even more important.

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 help find desired code faster and make it easier to understand.

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

Separate file names with dots and dashes

Style 02-02

Do use dashes to separate words in the descriptive name.

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

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

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

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

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

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

Why?
Type names provide pattern matching for any automated tasks.

Symbols and file names

Style 02-03

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

Do use upper camel case for class names.

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

Do append the symbol name with the conventional suffix (such as Component, Directive, Module, Pipe, or Service) for a thing of that type.

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

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

Symbol name	File name

`@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`

`@Directive({ … })`
`export class ValidationDirective { }`
	`validation.directive.ts`

`@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`

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`

Bootstrapping

Style 02-05

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

Do include error handling in the bootstrapping logic.

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

Why?
Follows a consistent convention for the startup logic of an app.

Why?
Follows a familiar convention from other technology platforms.

```
main.ts
```

```
import {AppComponent} from './app/app.component';
import {bootstrapApplication} from '@angular/platform-browser';

bootstrapApplication(AppComponent);
```

Component selectors

Style 05-02

Do use `dashed-case` or `kebab-case` for naming the element selectors of components.

Why?
Keeps the element names consistent with the specification for Custom Elements.

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

```
/* avoid */

@Component({
  standalone: true,
  selector: 'tohHeroButton',
  templateUrl: './hero-button.component.html',
})
export class HeroButtonComponent {}


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

Component custom prefix

Style 02-07

Do use a hyphenated, lowercase element selector value; for example, `admin-users`.

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

Why?
Prevents element name collisions with components in other applications and with native HTML elements.

Why?
Makes it easier to promote and share the component in other applications.

Why?
Components are easy to identify in the DOM.

`app/heroes/hero.component.ts`

```
/* avoid */

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

`app/users/users.component.ts`

```
/* avoid */
```

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

`app/heroes/hero.component.ts`

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

`app/users/users.component.ts`

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

Directive selectors

Style 02-06

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

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

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

Directive custom prefix

Style 02-08

Do spell non-element selectors in lower camel case unless the selector is meant to match a native HTML attribute.

Don't prefix a directive name with ng because that prefix is reserved for Angular and using it could cause bugs that are difficult to diagnose.

Why?
Prevents name collisions.

Why?
Directives are easily identified.

`app/shared/validate.directive.ts`

```
/* avoid */

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

`app/shared/validate.directive.ts`

```

@Directive({
  standalone: true,
  selector: '[tohValidate]',
})
export class ValidateDirective {}
```

Pipe names

Style 02-09

Do use consistent names for all pipes, named after their feature. The pipe class name should use `UpperCamelCase` (the general convention for class names), and the corresponding name string should use `lowerCamelCase`. The name string cannot use hyphens ("dash-case" or "kebab-case").

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

Symbol name	File name
`@Pipe({ standalone: true, name: 'ellipsis' })` `export class EllipsisPipe implements PipeTransform { }`	`ellipsis.pipe.ts`
`@Pipe({ standalone: true, name: 'initCaps' })` `export class InitCapsPipe implements PipeTransform { }`	`init-caps.pipe.ts`

Unit test file names

Style 02-10

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

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

Why?
Provides a consistent way to quickly identify tests.

Why?
Provides pattern matching for karma 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`

Application structure and NgModules

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

All of the application's code goes in a folder named `src`. All feature areas are in their own folder.

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. Use the naming conventions for files in this guide.

Overall structural guidelines

Style 04-06

Do start small but keep in mind where the application is heading down the road.

Do have a near term view of implementation and a long term vision.

Do put all of the application'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 application structure small and easy to maintain in the early stages, while being easy to evolve as the application grows.

Why?
Components often have four files (for example, `*.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
│   │   │   └── 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.routes.ts
│   │   ├── shared
│   │   │   └── init-caps.pipe.ts|spec.ts
│   │   │   └── filter-text.component.ts|spec.ts
│   │   │   └── filter-text.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.routes.ts
│   └── main.ts
│   └── index.html
│   └── …
└── node_modules/…
└── …
```

HELPFUL: While components in dedicated folders are widely preferred, another option for small applications is to keep components flat (not in a dedicated folder). This adds up to four files to the existing folder, but also reduces the folder nesting. Whatever you choose, be consistent.

Folders-by-feature structure

Style 04-07

Do create folders named for the feature area they represent.

Why?
A developer can locate the code and identify what each file represents at a glance. The structure is as flat as it can be and there are no repetitive or redundant names.

Why?
Helps reduce the application from becoming cluttered through organizing the content.

Why?
When there are a lot of files, for example 10+, locating them is easier with a consistent folder structure and more difficult in a flat structure.

For more information, refer to this folder and file structure example.

App root module

IMPORTANT: The following style guide recommendations are for applications based on NgModule. New applications should use standalone components, directives, and pipes instead.

Style 04-08

Do create an NgModule in the application's root folder, for example, in `/src/app` if creating a NgModule based app.

Why?
Every NgModule based application requires at least one root `NgModule`.

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

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

`app/app.module.ts`

```
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],
})
export class AppModule {}
```

Feature modules

Style 04-09

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

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

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

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

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

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

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

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

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

Why?
A feature module can easily be isolated for testing.

Shared feature module

Style 04-10

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

Consider not providing services in shared modules. Services are usually singletons that are provided once for the entire application or in a particular feature module. There are exceptions, however. For example, in the sample code that follows, notice that the SharedModule provides FilterTextService. This is acceptable here because the service is stateless;that is, the consumers of the service aren't impacted by new instances.

Do import all modules required by the assets in the SharedModule; for example, CommonModule and FormsModule.

Why?
`SharedModule` will contain components, directives, and pipes that may need features from another common module; for example, `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 undesirable results.

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

```
project root
├──src
├──├──app
├──├──├── shared
├──├──├──└── shared.module.ts
├──├──├──└── init-caps.pipe.ts|spec.ts
├──├──├──└── filter-text.component.ts|spec.ts
├──├──├──└── filter-text.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],
})
export class SharedModule {}
```

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.

Why?
The folder makes it easy to identify and isolate the feature content.

Components

Components as elements

Style 05-03

Consider giving components an element selector, as opposed to attribute or class selectors.

Why?
Components have templates containing HTML and optional Angular template syntax. They display content. Developers place components on the page as they would native HTML elements and web components.

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

HELPFUL: There are a few cases where you give a component an attribute, such as when you want to augment a built-in element. For example, Material Design uses this technique with <button mat-button>. However, you wouldn't use this technique on a custom element.

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

```
/* avoid */

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

`app/app.component.html`

```
<!-- avoid -->

<div tohHeroButton></div>
```

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

Extract templates and styles to their own files

Style 05-04

Do extract templates and styles into a separate file, when more than 3 lines.

Do name the template file `[component-name].component.html`, where `[component-name]` is the component name.

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

Do specify component-relative URLs, prefixed with `./`.

Why?
Large, inline templates and styles obscure the component's purpose and implementation, reducing readability and maintainability.

Why?
In most editors, syntax hints and code snippets aren't available when developing inline templates and styles. The Angular TypeScript Language Service (forthcoming) promises to overcome this deficiency for HTML templates in those editors that support it; it won't help with CSS styles.

Why?
A component relative URL requires no change when you move the component files, as long as the files stay together.

Why?
The ./ prefix is standard syntax for relative URLs; don't depend on Angular's current ability to do without that prefix.

`app/heroes/heroes.component.ts`

```
/* avoid */

@Component({
  standalone: true,
  selector: 'toh-heroes',
  template: `
    <div>
```

```
      <h2>My Heroes</h2>
      <ul class="heroes">
        @for (hero of heroes | async; track hero) {
          <li (click)="selectedHero=hero">
            <span class="badge">{{hero.id}}</span> {{hero.name}}
          </li>
        }
      </ul>
      @if (selectedHero) {
        <div>
          <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;
    }
  `,
  ],
  imports: [NgFor, NgIf, AsyncPipe, UpperCasePipe],
})
export class HeroesComponent {
  heroes: Observable<Hero[]>;
  selectedHero!: Hero;

  constructor(private heroService: HeroService) {
    this.heroes = this.heroService.getHeroes();
  }
}
```

```
@Component({
  standalone: true,
  selector: 'toh-heroes',
  templateUrl: './heroes.component.html',
  styleUrls: ['./heroes.component.css'],
  imports: [NgFor, NgIf, AsyncPipe, UpperCasePipe],
})
export class HeroesComponent {
  heroes: Observable<Hero[]>;
  selectedHero!: Hero;

  constructor(private heroService: HeroService) {
    this.heroes = this.heroService.getHeroes();
  }
}
```

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({
  standalone: true,
  selector: 'toh-hero-button',
  template: `<button type="button"></button>`,
  inputs: ['label'],
  outputs: ['heroChange'],
})
```

```
export class HeroButtonComponent {
  heroChange = new EventEmitter<any>();
  label: string = '';
}
```

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

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

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({
  standalone: true,
  selector: 'toh-hero-button',
  template: `<button type="button">{{label}}</button>`,
})
export class HeroButtonComponent {
  // Pointless aliases
  @Output('heroChangeEvent') heroChange = new EventEmitter<any>();
  @Input('labelAttribute') label!: string;
}
```

`app/app.component.html`

```
<!-- avoid -->

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

```
@Component({
  standalone: true,
  selector: 'toh-hero-button',
  template: `<button type="button" >{{label}}</button>`,
})
export class HeroButtonComponent {
  // No aliases
  @Output() heroChange = new EventEmitter<any>();
  @Input() label = '';
}
```

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 as 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 {Component, OnInit} from '@angular/core';
import {HttpClient} from '@angular/common/http';

import {Observable} from 'rxjs';
import {catchError, finalize} from 'rxjs/operators';

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

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

@Component({
  standalone: true,
  selector: 'toh-hero-list',
  template: `...`,
})
export class HeroListComponent implements OnInit {
  heroes: Hero[];

  constructor(private http: HttpClient) {
    this.heroes = [];
  }

  getHeroes() {
    this.heroes = [];
    this.http
      .get(heroesUrl)
      .pipe(
        catchError(this.catchBadResponse),
        finalize(() => this.hideSpinner()),
      )
      .subscribe((heroes: Hero[]) => (this.heroes = heroes));
  }
  ngOnInit() {
    this.getHeroes();
  }

  private catchBadResponse(err: any, source: Observable<any>) {
    // log and handle the exception
    return new Observable();
  }

  private hideSpinner() {
    // hide the spinner
  }
}
```

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

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

import {Hero, HeroService} from '../shared';

@Component({
  standalone: true,
  selector: 'toh-hero-list',
  template: `...`,
})
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();
  }
}
```

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({
  standalone: true,
  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>();
}
```

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({
  standalone: true,
  selector: 'toh-hero-list',
  template: `
    <section>
      Our list of heroes:
      @for (hero of heroes; track hero) {
        <toh-hero [hero]="hero"></toh-hero>
      }
      Total powers: {{totalPowers}}<br>
      Average power: {{totalPowers / heroes.length}}
    </section>
  `,
  imports: [NgFor, HeroComponent],
})
export class HeroListComponent {
  heroes: Hero[] = [];
  totalPowers: number = 0;
}
```

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

```
@Component({
  standalone: true,
  selector: 'toh-hero-list',
  template: `
    <section>
      Our list of heroes:
      @for (hero of heroes; track hero) {
        <toh-hero [hero]="hero"></toh-hero>
      }
      Total powers: {{totalPowers}}<br>
      Average power: {{avgPower}}
    </section>
  `,
  imports: [NgFor, HeroComponent],
})
export class HeroListComponent {
  heroes: Hero[];
  totalPowers = 0;

...
  get avgPower() {
    return this.totalPowers / this.heroes.length;
  }
}
```

Initialize inputs

Style 05-18

TypeScript's `--strictPropertyInitialization` compiler option ensures that a class initializes its properties during construction. When enabled, this option causes the TypeScript compiler to report an error if the class does not set a value to any property that is not explicitly marked as optional.

By design, Angular treats all @Input properties as optional. When possible, you should satisfy `--strictPropertyInitialization` by providing a default value.

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

```
@Component({
  standalone: true,
  selector: 'toh-hero',
  template: `...`,
})
export class HeroComponent {
  @Input() id = 'default_id';
}
```

```
If the property is hard to construct a default value for, use ? to explicitly
mark the property as optional.
```

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

```
@Component({
  standalone: true,
  selector: 'toh-hero',
  template: `...`,
})
export class HeroComponent {
  @Input() id?: string;

  process() {
    if (this.id) {
      // ...
    }
  }
}
```

You may want to have a required `@Input` field, meaning all your component users are required to pass that attribute. In such cases, use a default value. Just suppressing the TypeScript error with ! is insufficient and should be avoided because it will prevent the type checker from ensuring the input value is provided.

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

```
@Component({
  standalone: true,
  selector: 'toh-hero',
  template: `...`,
})
export class HeroComponent {
  // The exclamation mark suppresses errors that a property is
  // not initialized.
  // Ignoring this enforcement can prevent the type checker
  // from finding potential issues.
  @Input() id!: string;
}
```

Directives

Use directives to enhance an element

Style 06-01

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

Why?
Attribute directives don't have an associated template.

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

`app/shared/highlight.directive.ts`

```
@Directive({
  standalone: true,
  selector: '[tohHighlight]',
})
export class HighlightDirective {
  @HostListener('mouseover') onMouseEnter() {
    // do highlight work
  }
}
```

app/app.component.html

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

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/method declaration in the directive's class and the metadata in the decorator associated with the directive.

app/shared/validator.directive.ts

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

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

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: HttpClient) {}

  getHeroes() {
    return this.http.get<Hero[]>('api/heroes');
  }
}
```

Providing a service

Style 07-03

Do provide a service with the application root injector in the `@Injectable` decorator of the service.

Why?
The Angular injector is hierarchical.

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

Why?
When you register a service in the `@Injectable` decorator of the service,

optimization tools such as those used by the Angular CLI's production builds can perform tree shaking and remove services that aren't used by your app.

Why?
This is not ideal when two different components need different instances of a service. In this scenario it would be better to provide the service at the component level that needs the new and separate instance.

`src/app/treeshaking/service.ts`

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

@Injectable({
  providedIn: 'root',
})
export class Service {}
```

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(HttpClient) private http: HttpClient,
  ) {}
}
```

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

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

Data Services

Talk to the server through a service

Style 08-01

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

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

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

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

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

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

Lifecycle hooks

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

Implement lifecycle hook interfaces

Style 09-01

Do implement the lifecycle hook interfaces.

Why?
Lifecycle interfaces prescribe typed method signatures. Use those signatures to flag spelling and syntax mistakes.

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

```
/* avoid */

@Component({
  standalone: true,
  selector: 'toh-hero-button',
  template: `<button type="button">OK</button>`,
})
export class HeroButtonComponent {
  onInit() {
    // misspelled
    console.log('The component is initialized');
  }
}
```

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

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

Security of Angular Applications

Angular has built-in protections against common web vulnerabilities, but developers need to follow best practices to ensure their applications are secure. Here's a summary of Angular security best practices:

Preventing Cross-Site Scripting (XSS)

- **Use template interpolation:** Always use the double curly braces {{ }} for displaying data. Angular automatically sanitizes the values to prevent XSS attacks.
- **Avoid innerHTML:** If you must use innerHTML to dynamically add HTML, bind its generation to [innerHTML] and sanitize the content.
- **Server-side validation:** Validate user-submitted data on the server-side to add an extra layer of protection.

HTTP-related Vulnerabilities

- **Cross-Site Request Forgery (CSRF):** Implement CSRF tokens to verify that the user making the request is the legitimate user.
- **Cross-Site Script Inclusion (XSSI):** Ensure your JSON responses are wrapped in a function call to prevent XSSI attacks.

Other Important Practices

- **Avoid risky APIs:** Be cautious when using APIs like ElementRef that can manipulate the DOM directly.
- **Stay updated:** Keep your Angular libraries and dependencies up-to-date to benefit from the latest security patches.
- **Route guards:** Use route guards to protect specific routes and prevent unauthorized access.
- **Content Security Policy (CSP):** Implement CSP to control the resources the browser is allowed to load, reducing the risk of XSS attacks.

Common Vulnerabilities

While Angular has built-in security measures, developers can still introduce vulnerabilities. Here are some common ones:

1. Cross-Site Scripting (XSS)

Improper Sanitization: Failing to sanitize user inputs can allow attackers to inject malicious scripts into your application. This can lead to data theft, session hijacking, or website defacement.

DOM Manipulation: Directly manipulating the DOM using APIs like ElementRef without proper sanitization can also introduce XSS vulnerabilities.

2. HTTP-Related Vulnerabilities

Cross-Site Request Forgery (CSRF): If not properly protected, attackers can trick users into performing unwanted actions on your application, such as changing their password or making a purchase.

Cross-Site Script Inclusion (XSSI): Loading external JSON resources without proper precautions can expose your application to XSSI attacks, potentially allowing attackers to steal sensitive data.

3. Authentication and Authorization

Bypassing Authentication: Weak authentication mechanisms or flaws in implementation can allow attackers to gain unauthorized access to your application.

Insufficient Authorization: Failing to properly restrict access to certain features or data based on user roles can lead to privilege escalation and data breaches.

4. Other Vulnerabilities

Insecure Third-Party Libraries: Using outdated or vulnerable third-party libraries can introduce security risks to your application.

Sensitive Data Exposure: Failing to protect sensitive data like API keys, passwords, or user information can lead to data breaches.

Denial of Service (DoS): Inadequate protection against DoS attacks can make your application unavailable to legitimate users.

It's crucial to be aware of these vulnerabilities and follow security best practices to mitigate the risks and build secure Angular applications.

Advice for Angular 18 Migration

Angular 18 promotes the use of standalone components, offering a streamlined way to build applications without relying heavily on `NgModules`. Here's how you can effectively port your module-based Angular code to this new architecture:

1. Utilize the Angular CLI's Migration Schematic

The Angular CLI provides a powerful schematic to automate much of the conversion process:

```
ng generate @angular/core:standalone
```

This schematic offers several options:

Convert all components, directives, and pipes to standalone: This automatically adds the `standalone: true` flag to your components, directives, and pipes, and populates the imports array with necessary dependencies.

Remove unnecessary `NgModule` classes: This attempts to remove `NgModule`s that are no longer required.

Bootstrap the project using standalone APIs: This configures your application to bootstrap using the standalone APIs.

2. Manual Adjustments and Optimizations

Review and Refactor: After running the schematic, carefully review the converted code. You might need to manually adjust imports, especially for components or services with complex dependencies.

Lazy Loading: If you're using lazy loading, update your routing configuration to use the `loadComponent` option instead of `loadChildren`.

Third-party Libraries: Ensure any third-party libraries you're using are compatible with standalone components. Some libraries might require adjustments or updates.

SCAM Pattern: If you're dealing with a large project, consider migrating incrementally. *Start with components that follow the SCAM (Single Component Angular Module) pattern, as they are easier to convert*.

3. Best Practices for Standalone Components

Explicit Imports: Clearly declare all the dependencies of your standalone component in its imports array.

Scoped Styles: Use component-scoped styles to avoid CSS conflicts and improve maintainability.

Organization: Even without `NgModules`, maintain a good project structure to organize your components, services, and other files.

Example:

Let's say you have a simple component within a module:

```
// app.module.ts

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

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

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

import { MyComponent } from './my.component';


@NgModule({

  declarations: [AppComponent, MyComponent],

  imports: [BrowserModule],

  bootstrap: [AppComponent]

})

export class AppModule { }


// my.component.ts

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

@Component({

  selector: 'app-my',

  template: '<p>My Component</p>'

})

export class MyComponent { }
```

After converting to a standalone component:

```
// app.component.ts

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

import { MyComponent } from './my.component';

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

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [MyComponent, BrowserModule],
  template: '<app-my></app-my>'
})
export class AppComponent { }


// my.component.ts
import { Component } from '@angular/core';
@Component({
  selector: 'app-my',
  standalone: true,
  template: '<p>My Component</p>'
})
export class MyComponent { }


// main.ts
import { bootstrapApplication } from '@angular/platform-browser';
import { AppComponent } from './app/app.component';


bootstrapApplication(AppComponent);
```

By following these steps, you can successfully migrate your Angular application to leverage the benefits of standalone components, such as improved performance, reduced bundle size, and simplified code organization.

Additional Learning Resources

Topic	Resources
Angular Basics	https://angular.dev/tutorials/first-app https://angular.dev/essentials/components https://angular.dev/essentials/managing-dynamic-data https://angular.dev/essentials/rendering-dynamic-templates https://angular.dev/essentials/conditionals-and-loops https://angular.dev/essentials/handling-user-interaction https://angular.dev/essentials/sharing-logic
Angular Guides (for advanced learners)	(follow the trail on the left for every link) https://angular.dev/guide/components https://angular.dev/guide/templates https://angular.dev/guide/directives https://angular.dev/guide/di https://angular.dev/guide/signals https://angular.dev/guide/routing https://angular.dev/guide/forms https://angular.dev/guide/http https://angular.dev/guide/performance https://angular.dev/guide/testing https://angular.dev/guide/i18n https://angular.dev/guide/animations https://angular.dev/guide/experimental/zoneless
Angular Security	https://angular.dev/best-practices/security
Additional Topics	https://angular.dev/ecosystem/service-workers https://angular.dev/ecosystem/web-workers
Angular Material	https://material.angular.io/guide/getting-started https://material.angular.io/components/categories https://material.angular.io/guide/theming https://material.angular.io/guides
TypeScript	https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html https://www.typescriptlang.org/docs/handbook/2/basic-types.html
ES6 Modules	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import/with https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/import
ES6 Features	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind
RxJS	https://www.learnrxjs.io/learn-rxjs/operators

Topic	Resources
Nx	https://nx.dev/getting-started/intro https://nx.dev/getting-started/tutorials/angular-monorepo-tutorial
Testing	https://jestjs.io/docs/getting-started https://jestjs.io/docs/mock-functions https://jestjs.io/docs/tutorial-async https://docs.cypress.io/app/end-to-end-testing/writing-your-first-end-to-end-test https://angular.dev/guide/testing/components-basics https://angular.dev/guide/testing/components-scenarios
Microfrontends	https://martinfowler.com/articles/micro-frontends.html https://www.angulararchitects.io/en/blog/the-microfrontend-revolution-part-2-module-federation-with-angular/
Design Patterns	https://www.cs.unc.edu/~stotts/COMP723-s13/patterns/sum.html https://sourcemaking.com/design_patterns https://en.wikipedia.org/wiki/Builder_pattern https://en.wikipedia.org/wiki/Factory_method_pattern https://en.wikipedia.org/wiki/Adapter_pattern https://en.wikipedia.org/wiki/Composite_pattern https://en.wikipedia.org/wiki/Decorator_pattern https://en.wikipedia.org/wiki/Facade_pattern https://en.wikipedia.org/wiki/Proxy_pattern https://en.wikipedia.org/wiki/Chain-of-responsibility_pattern https://en.wikipedia.org/wiki/Strategy_pattern https://en.wikipedia.org/wiki/Template_method_pattern https://en.wikipedia.org/wiki/Observer_pattern https://en.wikipedia.org/wiki/Mediator_pattern
Microservice Architectural Patterns	https://learn.microsoft.com/en-us/azure/architecture/patterns/#catalog-of-patterns
NgRx	https://ngrx.io/guide/store https://ngrx.io/guide/effects https://ngrx.io/guide/router-store
Immutability	https://www.youtube.com/watch?v=I7IdS-PbEgI&pp=ygUQaW1tdXRhYmxlanMgbWV0YQ== https://immerjs.github.io/immer/ https://angular.love/immutability-importance-in-angular-applications