Angular
Folder Structure
& Style guide

Angular Folder Structure for Large Teams

```
src/
├── app/
│   ├── core/ → Global services (auth, logger, layout)
│   ├── shared/ → UI-only, reusable components
│   ├── features/ → Business domains (users, reports, dashboard)
│   │   ├── dashboard/
│   │   ├── reports/
│   │   └── users/
│   └── app.routes.ts → Global routing setup
├── assets/
├── environments/
└── main.ts
```

Organize your project into subdirectories based on the features of your application or common themes to the code in those directories

Avoid creating subdirectories based on the type of code that lives in those directories.
For example, avoid creating directories like components, directives, and services.

Core folder

- Contains App-Wide Services & Singletons
- Examples:
  - Auth service
  - App-wide HTTP interceptors
  - Logging service
  - Layout service
  - Shell or base route guards
  - These services are usually singletons and should be injected via providedIn: 'root'

- *Avoid putting UI or feature-specific logic here*

Shared folder

- The shared folder is strictly for reusable presentational elements
- Examples:
  - UI components
    - Like Button, Modal, Dropdown, …
  - Reusable pipes
    - Like formatDate, truncate, …
  - Directives
    - Like autofocus, debounceClick, …
- *Rule of thumb: If it uses business logic, it doesn't belong here*

Feature folder

- The features folder is the heart of your app
  - Each feature (e.g., reports, users, dashboard) is organized into its own folder
- Each feature contains:
  - Its own routing config
  - State management (optional)
  - Smart and dumb components
  - Scoped services

```
reports/
├── components/
├── services/
├── store/ → Component Store / Signals / NGRX
├── reports.routes.ts
├── reports.ts
```

$\}$ Avoid creating subdirectories based on the type of code that lives in those directories.
But avoid putting so many files into one directory that it becomes hard to read or navigate.
As the number of files in a directory grows, consider splitting further into additional sub-directories

- To generate a new component:
  - `ng g c features/reports/component-name`
  - Or `ng g c features/reports/components/component-name`

AARHUS UNIVERSITY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Angular Routing Strategy for Scalable Apps

- Use route-based lazy loading of features
- Keep features isolated
- Match folder names to route paths
- Use feature-level routing files for navigating between sub features

Style recommendations

Dependency injection

- Prefer the inject function over constructor parameter injection

- The inject function works the same way as constructor parameter injection, but offers several style advantages:
  - inject is generally more readable
  - It's more syntactically straightforward to add comments to injected dependencies
  - inject offers better type inference

Components and directives

- Group Angular-specific properties before methods
  - Components and directives should group Angular-specific properties together, typically near the top of the class declaration
    - This includes injected dependencies, inputs, outputs, and queries
  - This practice makes it easier to find the class's template APIs and dependencies
- Keep components and directives focused on presentation
  - For code that makes sense on its own, decoupled from the UI, prefer refactoring to other files
  - For example, you can factor form validation rules or data transformations into separate functions or classes
- Avoid overly complex logic in templates
  - Angular templates are designed to accommodate JavaScript-like expressions
    - You should take advantage of these expressions to capture relatively straightforward logic directly in template expressions
  - But refactor logic into the TypeScript code when it gets to complicated

Components - visibility

- Use `protected` on class members that are only used by a component's template
  - A component class's public members intrinsically define a public API that's accessible via dependency injection and queries

- Use `readonly` on properties that are initialized by Angular
  - This includes properties initialized by input, model, output, and queries
  - The readonly access modifier ensures that the value set by Angular is not overwritten

Components – Event handlers

- Prefer naming event handlers for the action they perform rather than for the triggering event

```
<!-- PREFER -->
<button (click)="saveUserData()">Save</button>
<!-- AVOID -->
<button (click)="handleClick()">Save</button>
```

- Using meaningful names like this makes it easier to tell what an event does from reading the template

```
<textarea
  (keydown.control.enter)="commitNotes()”
  (keydown.control.space)="showSuggestions()”
>
```

AARHUS UNIVERSITY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Components – Life cycle methods

- Avoid putting long or complex logic inside lifecycle hooks like ngOnInit
- Instead, prefer creating well-named methods to contain that logic and then call those methods in your lifecycle hooks
- Reason:
  - Lifecycle hook names describe when they run, meaning that the code inside doesn't have a meaningful name that describes what the code inside is doing
- Example

```
ngOnInit() {
  this.startLogging();
  this.runBackgroundTask();
}
```

References & Links

- Best Angular Folder Structure for Large Teams (2025 Guide)
  https://blog.stackademic.com/best-angular-folder-structure-for-large-teams-2025-guide-acb61babaf28
- Style guide
  https://angular.dev/style-guide