5 Tips & Best Practices to Organize your Angular Project

Rik de Vos

Apr 15, 2019 · 7 min read ★

Photo from unsplash.com

There are different ways of organizing Angular projects. Many of them follow a pattern of using a shared and core module, which we'll be discussing in this article amongst other tips to keep your codebase clean and maintainable. While this architecture is suited for most projects, larger enterprise projects or tiny one-page apps might favour different approaches. In the end the architecture of each project serves the same purpose: to create a maintainable and scalable codebase.

To demonstrate the tips here, we'll create a sample Angular app. You can follow along or check out the 👉 Github repository here with some more examples.

```
npm install -g @angular/cli@latest
ng new best-practices-app --routing=true --style=scss
cd best-practices-app
```

. . .

1. Use a Shared Module 📕

> *Feature modules are NgModules for the purpose of organizing code.*

Every Angular app has at least one module, the *root module*, by default called `AppModule`. You bootstrap that module to launch the application. As your application grows it doesn't make sense to keep everything in the same module, so you refactor the root module into smaller Feature Modules that represent collections of related functionality. Feature Modules declare their own components, pipes and directives, and provide services which are injected as dependencies into components. They can also export these, or import a different Angular module.

The Shared Module

Even when an application is small you often have some components that are used *everywhere*, like a button component or price pipe. It's a best practise to create a `SharedModule` and declare these often used components, directives and pipes in there. Then `export` these declarations and they will be accessible inside **any** module that imports the `SharedModule`. Furthermore, if you're importing modules that make declarations which need to be available across the entire app (like `FormsModule`, `CommonModule`, or a 3rd-party modules), just `import` and `export` them in the `SharedModule` too.

```
ng g m shared
```

Let's create some shared button components as an example in our `SharedModule`.

```
ng g c shared/button-a
ng g c shared/button-b
```

Next add the shared `imports` and `exports`.

```
// src/app/shared/shared.module.ts

import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { CommonModule } from '@angular/common';
import { ButtonAComponent } from './components/button-a/button-
a.component';
import { ButtonBComponent } from './components/button-b/button-
b.component';

@NgModule({
  declarations: [
    ButtonAComponent,
    ButtonBComponent
  ],
  imports: [
    CommonModule,
    FormsModule
  ],
  exports: [
    ButtonAComponent,
    ButtonBComponent,
    FormsModule
  ]
})
export class SharedModule {}
```

The `SharedModule` should not be providing any services, as these could be instantiated more than once when importing the `SharedModule` multiple times. Instead, move your singleton services into a new module called `CoreModule` *(next chapter)*.

In short, when using a `Shared Module`:

- **DO** `declare` components, pipes, directives, and `export` them.
- **DO** import `FormsModule`, `ReactiveFormsModule` and other (3rd-party) modules you need.
- **DO** import the `SharedModule` into any other Feature Modules.
- **DO NOT** provide app-wide singleton services in your `SharedModule`. Instead move these to the `CoreModule`.

- **DO NOT** import the `SharedModule` into the `AppModule`.

. . .

2. Use a Core Module 🎯

> *A Core Module is an NgModule containing code that will be used to instantiate your app and load some core functionality.*

In the Core Module we commonly place our singleton services and modules that will be used across the app but only need to be imported **once**. Examples are an Authentication Service or LocalStorage Service, but also modules like `HttpClientModule`, `StoreModule.forRoot(…)`, `TranslateModule.forRoot(…)`. The `CoreModule` is then imported into the `AppModule`.

To create the `CoreModule`

```
ng g m core
```

Let's create some singleton services.

```
ng g s core/services/auth
ng g s core/services/local-storage
```

If we now open our `CoreModule`, we can add some modules that should be loaded only once.

```
// src/app/core/core.module.ts

import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  declarations: [],
  providers: [],
  imports: [
    CommonModule,
    HttpClientModule
```

```
  ],
  exports: []
})
export class CoreModule {}
```

> But why are the services I created not provided in the module?

If you open one of the services we created, you'll notice they are set to `providedIn: 'root'`. This is new in Angular 6, and officially called "**Tree-shakable providers**". Instead of the module providing all its services, it is now the service itself declaring where it should be provided. There is no need to add it to the `providers` array in the `@NgModule`, the Angular team even discourages this.

In short, when using a `Core Module`:

- **DO** import modules that should be instantiated once in your app.
- **DO** place services in the module, but do not `provide` them.
- **DO NOT** declare components, pipes, directives.
- **DO NOT** import the `CoreModule` into any modules other than the `AppModule`.

3. Simplify your imports 🖌

When working with many components, things can get messy pretty quickly, especially in the `SharedModule`. If we take a look we can see that each component there is references 3 **(!!!)** times. Within little time the module will be cluttered with many components, modules and pipes. Let's fix this by creating an `index.ts` file inside the `src/app/shared/components` folder and exporting the components.

```
// src/app/shared/components/index.ts

import { ButtonAComponent } from './button-a/button-a.component';
import { ButtonBComponent } from './button-b/button-b.component';

export const components: any[] = [ButtonAComponent,
ButtonBComponent];

export * from './button-a/button-a.component';
export * from './button-b/button-b.component';
```

We're doing 3 things here for each component, which is very easy when using VS Code's IntelliSense:

- Import the Component
- Add it to a named exported object `components`
- Export the Component

We can now import one or more components from anywhere like this:

```
import { ButtonAComponent, ButtonBComponent } from
'/path/to/shared/components';
```

Inside `SharedModule` things are even simpler as we can import the whole components object

```
// shared/shared.module.ts

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

import * as fromComponents from './components';

@NgModule({
  declarations: [...fromComponents.components];
  imports: [CommonModule, FormsModule],
  exports: [FormsModule, ...fromComponents.components]
})
export class SharedModule {}
```

This technique is applicable to all components, containers (smart components), services, pipes, etc, folders in your project. I've added examples in the 👉 GitHub repo.

. . .

4. Shorten your relative paths 🛣️

After the project has grown a little and we have some nested modules, we can end up with code like this:

```
import { FooPipe } from '../../../../../shared/pipes/foo/foo.pipe';
```

This really isn't a nice solution, and when you start moving directories around those relative paths will break. Using absolute paths could work, but then all imports look the same whilst some are relative and some are not. Besides, if we move the `shared` folder around, our paths will still break. Also not ideal. Instead we should aim for something like this:

```
import { FooPipe } from '@shared/pipes/foo/foo.pipe';
```

Is this possible? Absolutely! 🙌 We just have to tell our compiler which paths it should map by modifying our `tsconfig.json`

```
{
...
  "compilerOptions": {
    ...
    "baseUrl": "src",
    "paths": {
      "@env": ["environments/environment"],
      "@shared/*": ["app/shared/*"],
      "@core/*": ["app/core/*"]
    }
  }
}
```

(Don't forget the `baseUrl`)

If you're using a smart IDE like VS Code, it will be clever enough to autocomplete to these paths. *Note: some IDE's may require you to add this configuration to all `tsconfig` files in your repository as they may only read one.*

When combining this tip with **Tip #2,** core services inside the `CoreModule` will be imported like this:

```
import { AuthService, LocalStorageService } from '@core/services';
```

. . .

5. Use SCSS Variables 🎨

This one seems pretty obvious, but it's not always done since it usually requires some Googling. That's why I've summarized the steps:

Create a file `src/styles/variables.scss` and add your SCSS variables, extends, mixins, etc inside:

```
// src/styles/variables.scss

$accent: #0093ff;

%uppercase {
  text-transform: uppercase;
  letter-spacing: 0.3em;
}
```

If we now @import this `variables.scss` file in another SCSS file, we can access these variables:

```
// shared/components/button-a/button-a.component.scss

@import '../../../../styles/variables.scss';

button {
  background: $accent;
}
```

This works fine, but we're again using relative paths 👷. Let's fix this by opening the Angular configuration file `angular.json`. Add the following after both of the `"styles": ["src/styles.scss"]` instances:

```
// angular.json
```

```
"stylePreprocessorOptions": {
  "includePaths": ["src/styles"]
},
```

This tells the SCSS Preprocessor to search inside the `/styles` directory when importing files. We can now simplify the `@import` in our SCSS files.

```
// shared/components/button-a/button-a.component.scss

@import 'variables.scss';

button {
  background: $accent;
}
```

. . .

That's it!

Great, you made it to the end! Now you should have learned some techniques to organize your Angular project a little better. Please support this article with some 👏👏 and share your thoughts below.

. . .

Looking for a job in Amsterdam?

I work for **Sytac** as a Senior Front-end developer and we are looking for medior/senior developers that specialise in Angular, React, Java or Scala. Sytac is a very ambitious consultancy company in the Netherlands that works for a lot of renowned companies in banking, airline, government and retail sectors. You can think of companies like ING, KLM, Deloitte, Ahold Delhaize, ABN AMRO, Flora holland and many more.

From a personal opinion Sytac really sets itself apart with their client portfolio, but also with how they take care of their employees. They do really care about the wellbeing of their employees. Apart from a good salary (50K-75k), you will notice this in regular meetings with the consultant managers but also by the amount of events they organize and all the other perks they offer to keep all employees happy.

If you think you have what it takes to work with the best, send me an email on rik.devos@sytac.io and I'll be happy to tell you more.

Angular JavaScript Software Engineering Programming Front End Development