BetterProgramming

Follow 202K Followers

Best Practices for Angular 13

Learn the best practices to build high-quality and scalable Angular applications

Patrick Kalkman Dec 6, 2021 · 10 min read ★

Photo by Lina Trochez on Unsplash

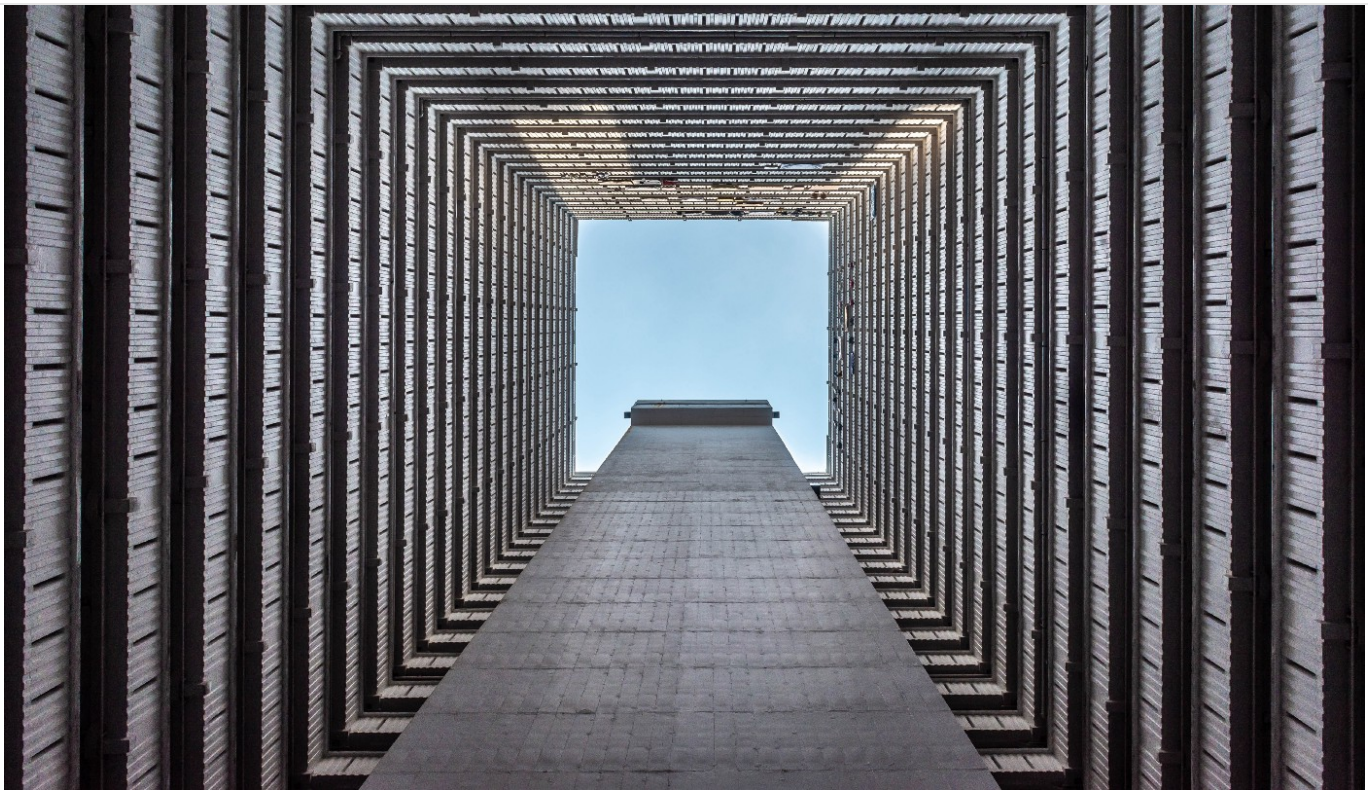
times to improve the structure and decrease the complexity.

This article will show you the best practices we learned during that period. These best practices will prevent running into clarity, scalability, and performance issues when building more significant, more complex applications.

We will start by demonstrating how to organize your Angular project and folders. Next, we will look at Angular components, services, and performance best practices. Following, we will look at how to handle state management. We will look at basic coding best practices at the end of the article.

At the end of this article, you'll know all the essential best practices to help you build high-quality and scalable Angular applications. We use Angular 13, but most of these practices also apply to previous versions.

Application Structure and Organization



Structure by Chor Tsang on Unsplash

There are many ways to organize your Angular application. It is impossible to create a structure that suits every use case. The organization of an application depends on the requirements of the project. Having said that, I think that some guidelines can help.

The Angular style guide uses the LIFT acronym as a guide. It stands for:

- **L**ocate — you must be able to find files quickly
- **I**dentify — name the file such that you instantly know what it contains
- **F**lat — keep a flat folder structure as long as possible.
- **T**ry to be DRY — don't repeat yourself but avoid being so DRY that you sacrifice readability.

Below is the format I use for medium to large Angular applications.

Folder structure of the Angular application

Core Folder and Module

The core folder and module contain shared singleton services and app-level components. The top-level App module uses these App-level components—for example, a navigation component. Creating a Core folder and module prevents your main application folder from becoming too cluttered.

Shared Folder And Module

The shared module contains components, pipes, and directives used and referenced by components in feature modules.

The Feature Folder and Modules

new Angular application using the CLI, it will only contain a single module. When your application grows, it helps to organize it into different feature modules. Having your features in separate modules allows Angular to load them lazily, improving performance.

Styles & Assets

The styles folder contains all the global style sheets of your application. The assets folder contains all the images, vectors, and other types of your application assets.

Angular Services Best Practices

Service by Erik Mclean on Unsplash

Decouple your components from data retrieval

retrieval. Splitting these responsibilities will make the retrieval logic reusable and easier to test. Another result is that your component classes will be lean and efficient.

So, avoid the temptation to retrieve data or call an API directly from your components. Use a service instead.

Add the injectable decorator to all services

When a service needs another service, you inject it via the service's constructor. You then have to add the `@Injectable` decorator to the class. You can add `@Injectable` to all your services even if they don't need another service at the moment. I find it easier to add it directly instead of at the time of adding a service dependency. I often forget it.

Understand how Angular injects Services

The Angular Injector instantiates the dependencies and injects them into components or services. It is therefore essential to understand how this injector works.

When Angular starts, it creates a centralized injector. This injector collects all defined dependencies and maintains all created instances that the application can reuse inside a container. You tell Angular that it can use a service in its DI system by decorating it with @Injectable and listing the service in the Providers section of a Module.

Usually, Angular injects the same instance of a service into your component. There is one exception with feature Modules that are lazily loaded. These modules get a new instance of a service. Remember this when you use lazy loading and use state inside your service.

Performance Best Practices

Performance by Andy Beales on Unsplash

Lazy Loading

Lazy loading is a technique to load an Angular module when needed. For example, we load a feature module when a user selects a specific part of the application. The Angular compilation engine creates separate bundles for each feature module during compilation. This means that the size of your primary bundle decreases, so your application loads faster.

Monitoring Bundle Sizes

We already saw that decreasing the bundle size of your application results in an application that loads quicker. It is therefore essential to continuously monitor the size of your bundles.

Source-map-explorer is a tool that helps to visualize where all the code in your bundle is coming from. You install source-map-explorer with `npm install -g source-map-explorer`. After you generate your production build with source maps using `ng build --configuration production --source-map=true`. You can then visualize the generated js files using source-map-explorer. See below for an example.

Source-map-explorer generates a visualization of your bundles, image by the author.

Ahead-of-time Compilation and the CLI

There are two ways to compile your Angular application. Just-in-Time (JIT) compiles your application in the browser at runtime. Or you could use Ahead-of-Time (AOT), which compiles your application at build time. AOT has been the default since Angular 9.

You can choose the type of compiler via the `aot` property in your `angular.json` configuration file.

It would be best always to use AOT compilation as it compiles your application only once before deployment, which saves time.

Change Detection Strategy

Angular re-renders the view to display the updated data when data changes in a component. Angular makes sure that data in the component and the view are always in sync.

updated data to the view.

There are two change detection strategies: the `Default` strategy and the `OnPush` strategy. With the `Default` strategy, change detection works by checking if the value of template expression has changed for all components.

The second strategy, `OnPush`, compares inputs by reference. Using the `OnPush` strategy will improve the performance of your Angular application.

If you use the `OnPush` method, you must ensure that all your objects become immutable. If you don't and keep working with mutable objects, the complexity of your application increases. It then might introduce bugs that are hard to reproduce.

Enabling the OnPush change detection strategy on your component

If you switch to `OnPush` make sure that your objects become immutable. A good choice for going immutable is to use the Immutable.js library. This library provides immutable primitives for building applications, like immutable objects and lists.

Pure and Impure Pipe Performance

If you are using custom pipes, there are some performance considerations to consider. By default, custom pipes are pure, which means that they only get reevaluated when the object's reference to which the pipe is being applied changes.

Pure and impure pipe performance

A quick way to solve this is to change the pipe to impure by setting the pure property of the Pipe decorator on line 3 to false. This will trigger the pipe with every change in the data.

However, it would be best if you did not do this! An impure pipe sacrifices performance. A better way to fix this problem is to use immutability, where you change the object's reference by building a new one.

Angular Components

Separating Component, CSS, and Template Files

A component consists of a template, styles, and the component typescript code. You can integrate the template and the styles inside the component typescript file. A best practice is that when your HTML is more than four lines, you should separate the HTML template into a separate file.

Prefixing Component Selectors

It is a best practice to prefix your component selectors. If you store a component in a feature module, set the component prefix according to this feature module. If a common component resides in a shared module, use the app name as a selector prefix. Make sure that a prefix length is between 2 and 4 characters.

Prefixing the component selector

Decorating Input and Output Properties

There are two ways to create input and output properties of your components. The recommended way is to decorate your public properties with `@Input`. The second not recommended method is to use metadata.

Delegating Complex Logic to Services

Strive to keep the logic inside your component as simple as possible. Keeping it simple means that more complex logic should move into a separate class, a service.

Component Member Sequence

You should order the elements of your component always the same. I use the following order.

- Public properties
- Private Properties
- Constructor
- Public methods
- Private methods

Implementing Life Cycle Hook Interfaces

When Angular creates and destroys your component, you can attach several life cycle events to your component. Instead of directly implementing the method, you should use the interface. By implementing the interface via `implements [Interface]` you make sure that you implement the event correctly.

Implementing Life Cycle Hook Interfaces

Angular State Management

When your Angular application grows in size, it will need state management. You can implement state management using several patterns and libraries. One of the more straightforward ways is to use the @Input and @Output bindings of components. You can also use Angular services and simple variables to store the temporary data.

Once your application grows, you are going to need something different. We knew that our application would grow to an enterprise application size, so we started with NGXS.

NGXS is a state management pattern + library for Angular. This famous Redux pattern and library made this popular. NGXS works the same as it centralized your application's state and logic. So you have a central place in your application that contains all the state.

For example, I worked on an Angular application for showing movies on Airplanes. The application retrieved all the movies through a service and stored them in the state when the application started. Every component that needed to show movies selected them through the central state.

article to go into the details of using NGXS, so I saved that for another article.

Angular Coding Best Practices

Immutability

There are several best practices for coding in JavaScript or TypeScript. One of the first best practices that we mentioned before is immutability. Immutability is a general recommendation for all your JavaScript code. Immutability refers to programming where you don't change existing objects but create new objects with the needed changes. A JavaScript library that can help is Immutable.js.

In line 3 in the example above, we use Map from immutable to set the key b to the value 50. The result is a new map that contains all the values of map1, and the value of key b changed to 50

Strict Mode

Angular Strict Mode refers to the TypeScript validation that Angular executes when it compiles your app. Strict Mode improves maintainability and helps you catch bugs ahead of time. Strict Mode also allows automatically upgrading Angular using `ng update`.

Strict Mode is on by default when you create a new New Angular project using the CLI.

Small Files and Functions

Keeping your code files and functions small is a general best practice for apps written in any language. So this also applies to Angular applications. Keeping your functions small makes them easier to test, reuse, read, and maintain.

There is no strict limit for files or functions, but the Angular style guide suggests limiting your functions to more than 75 lines.

Single Responsibility principle

You can't create best practices on programming without mentioning the Single Responsibility Principle (SRP). Bob Martin, aka Uncle Bob, said that a single class or module should only have a single responsibility.

When I build a new Angular application, I break down the application into separate components. For each component, I describe its responsibilities and the input and outputs. I always start the description with: "This component is responsible for …." — the amount of text I need to describe the responsibility signals if I need to split the component further.

Conclusion

This article described several best practices that help you build high-quality and scalable Angular applications. We started with the organization of the Angular project. Then we looked at components, services, and performance. We ended the article by looking at general coding practices.

Twice, I referred to the Angular Style Guide. The style guide is an opinionated guide on syntax, conventions, and application structuring. Make sure also to read that guide.

I hope these best practices help you when building your Angular application.