Angular Best Practices for 2021

GrapeCity Developer Solutions Follow
Aug 23 · 9 min read

. . .

Built with TypeScript by Google developers, Angular is an open-source JavaScript framework designed for building front-end applications.

Angular 2+ is a successor to Angular.js, rewritten from scratch using TypeScript instead of JavaScript, which helped avoid many issues related to JavaScript and ensures following best practices and integrations with IDEs thanks to static typing and the class-based oriented object features of TypeScript.

Angular is not just a framework but an entire platform packed with features that make front-end web and mobile development more manageable. Also, thanks to projects by the community, you can build native apps for mobile (Ionic and NativeScript) and desktop (Electron) devices.

Angular is like other modern JavaScript libraries, such as React and Vue.js, and uses many shared concepts. While React is more popular among web developers worldwide, Angular is suitable for enterprise apps.

This article covers some of the best practices that developers should follow when building Angular applications.

Use Angular CLI

The first thing that you should consider when developing your web application is development tooling. These days, we have modern tools that make front-end web development more straightforward. For Angular, we have many tools, most importantly, the official Angular CLI and Nx, a smart and extensible build framework.

Even though you can create an Angular project without using the official CLI, this is only useful for learning purposes. For real-world development, you should use Angular CLI. It's a command-line interface created by the official team behind Angular, on top of Node.js. It makes it extremely easy to initialize a fully working Angular application from the start, without the hassle of configuring build tools like Webpack. It assists during development by providing the commands for scaffolding constructs such as modules and components, testing (unit, integration, and e2e testing), building the final production bundles, and even helping you with deploying the final app.

Make sure to use Angular CLI to generate your project since it comes with the best practices recommended by the team, or even use Nx if you are building full-stack applications.

Before installing Angular CLI, you must have a recent version of Node.js and npm installed. If you do not, you can use one of the following methods:

- Download the installer for your operating system from the official website
- Use the official package manager for your target system
- Use a Node version management tool such as NVM, enabling you to manage multiple versions of Node on your system. It's also helpful to install packages globally on your machine without using sudo on Linux or MAC and with no extra configuration

Now, install Angular CLI using the following command:

```
npm install -g @angular/cli
```

This command installs the CLI globally on your system.

You can run the ng command to get all the available commands at your disposal and then run ng followed by a particular command and the —help option to display the help file for that command.

You can check the installed version using the following command:

```
ng version
```

Next, run the following command to generate a new project:

```
ng new angular-practices-demo
```

Angular asks you:

- Would you like to add Angular routing? Type "y"
- Which stylesheet format would you like to use? Use the arrow keys to pick SCSS

Use a Scalable and Maintainable Project Structure

If you have done web development before, you know that finding a convenient project structure or architecture is not always easy on the first try. Still, it gets easier as you get more experience building both small and large apps.

For a small application, the default structure generated by Angular CLI is okay. Still, once your project grows, you'll find it difficult to maintain and scale your app correctly.

Here is an excellent article on how to structure the folders of your application, where you start from a barebones Angular project and move to a more organized solid folder structure with separate component and page folders. A page is simply a routed component.

Also, a good practice to follow is architecting your app with a core module, shared module, and feature module for each feature of your application (plus the root application module, which bootstraps the app). You then move the imports in the app module to the core module and leave the app module only for application bootstrapping.

You must place all the singleton services, which should only have one instance for the entire application in the core module. For example, the authentication service should only have one instance for each application so that it can be part of the core module.

In the shared module, you should place common artifacts (components, directives, pipes, and so on) used in multiple modules so that you can import the shared module to use them. The shared module is also a good place for dumb components and pipes that don't inject services but can only receive data through props.

Suppose you're using a UI components library like Angular Material. In this case, this is an excellent place to import and re-export the components that you intend to use throughout the app, so you don't need to repeat imports in each module.

To continue our previously generated project, run the following commands to create core and shared modules:

```
ng generate module core
ng generate module shared
```

Next, let's assume we need two features for product and cart.

Generate two feature modules for them with the same command:

```
ng generate module product
ng generate module cart
```

Next, open the src/app/shared.module.ts file and update it as follows:

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

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

Here, we added the FormsModule to the exports array, so the array exports it to the other modules that import the shared module, but we didn't add it to the imports array. This way, we can give other modules access to FormsModule without importing it directly in the shared **NgModule.**

Next, we re-export **CommonModule** and **FormsModule** to use common directives like **NgIf** and **NgFor** from **CommonModule** and bind component properties with `[(ngModel)]` from modules that import this **SharedModule**.

Next, open the src/app/app.module.ts file and import the core and shared modules as follows:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { CoreModule } from './core/core.module';
import { SharedModule } from './shared/shared.module';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    CoreModule,
    SharedModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Also, remove **CommonModule** from **ProductModule** and **CartModule** and import **SharedModule** since it already exports **CommonModule**.

Keep Up to Date

Angular follows semantic versioning with a new major version released every six months.

Semantic versioning is a convention used for versioning software. It has a major.minor.patch format. Angular increments each part when they release major, minor, or patch changes.

You can follow the news about the latest version of Angular from the CHANGELOG and make sure you keep your Angular version up to date, ensuring you always get the latest features, bug fixes, and performance

enhancements like Ivy.

It would help if you also used this official tool when updating your project from one version to the next.

Strict Mode

We mentioned in the introduction that Angular 2+ adopted TypeScript from the early phases, ensuring the platform — including the framework and the tooling — follows best practices such as dependency injection, which makes testing more manageable, and performance budgets.

The Angular team has moved to apply the strict mode progressively with an option in Angular 10 to enable strict mode by default for all projects starting with Angular 12. This is a best practice now enabled by default, but if you must disable it for learning purposes, you use the — no-strict option when creating a new project.

For existing projects, you enable strict mode in tsconfig.json as follows:

```
{
  "compilerOptions": {
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
  },
  "angularCompilerOptions": {
    "enableI18nLegacyMessageIdFormat": false,
    "strictInjectionParameters": true,
    "strictInputAccessModifiers": true,
    "strictTemplates": true
  }
}
```

Also, thanks to the Ivy compiler and the language service, you'll benefit from the TypeScript's type system in your templates by simply setting strictTemplates to true. This is the default, starting with Angular 12. Check out the official docs for more details.

Make sure to follow the Angular team's recommended security practices and avoid using ElementRef and innerHTML unless you're sure you know what you are doing!

Use Lazy Loading

When using Angular, you should build the so-called SPAs, which refer to single-page applications. This is a modern type of app that's different from the traditional web apps we created before.

Angular loads SPA bundles at once from the server and uses JavaScript or client-side routing to enable users to navigate between different views.

This is the modern approach for building apps today, and this how we build apps with modern frameworks such as Angular, React, and Vue.js.

Angular provides a powerful router with a plethora of features to use for client-side routing. So, building an SPA is easy once you grasp the necessary concepts. However, this impacts performance since we must download the full app bundles from the server. So, when your app size

grows, the downloading time of your application increases!

Here comes the role of lazy-loading, which revolves around the idea of deferring the loading of specific modules when the users of your application access them. This benefits you by reducing the actual downloading size of the application bundles. Lazy-loading also improves the boot time by not loading unused modules when the application first starts, but only when users trigger navigation.

As a best practice, you must lazy-load the feature modules in your application whenever that's possible. You need one feature module to load eagerly during the app start-up to display the initial content. You should lazy-load all other feature modules to boost performance and decrease the initial bundle size.

You can lazy-load a module using the loadChildren property of the Angular router with the dynamic import syntax. But thanks to Ivy, you can also lazy-load a component. Let's see an example!

First, make sure you have a project with Angular routing set up. With Angular CLI, you take care of this by setting the — routing flag for the ng new command when generating a project or answering "y" when prompted if you "Would like to add Angular routing?"

Open the src/app/app-routing.module.ts file and lazy-load the product and cart modules as follows:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { ProductModule } from './product/product.module';
import { CartModule } from './cart/cart.module';


const routes: Routes = [
  { path: 'product', loadChildren: () =>
import('./product/product.module').then(m => m.ProductModule) },
  { path: 'cart', loadChildren: () =>
import('./cart/cart.module').then(m => m.CartModule) }
];


@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

We use the **loadChildren** property of the route configuration combined with the import statement to lazy-load a module.

Now, any components you add to these modules will be lazy-loaded! However, with Ivy, we can lazy-load an Angular component without requiring a module.

First, generate a component using the following command:

```
ng generate component header --module=core
```

The core module imports this.

Open the src/app/app.component.html file and update as follows:

```
<button (click)="lazyLoadHeader()">Load header</button>
<ng-container #header></ng-container>
```

Next, open the src/app/app.component.ts file and update it as follows:

```
import { Component, ComponentFactoryResolver, ViewChild,
ViewContainerRef } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  title = 'angular-practices-demo';
  @ViewChild('header', { read: ViewContainerRef }) headerContainer:
ViewContainerRef | null = null;
  constructor(private factoryResolver: ComponentFactoryResolver) { }

  async lazyLoadHeader() {
    const { HeaderComponent } = await
import('./header/header.component');
    const factory =
this.factoryResolver.resolveComponentFactory(HeaderComponent);
    this.headerContainer?.createComponent(factory);
  }
}
```

When you click the button, you should see “header works!” which means the component lazy-loaded on-demand and rendered!

Unsubscribe from RxJS Observables

When subscribing your components to RxJS Observables, you should always unsubscribe. Otherwise, this causes unwanted memory leaks as the observable stream is open, even after destroying the component using it.

You can do this in multiple ways:

- Unsubscribe the component in the **ngOnDestory** event after destroying the component
- Use the **async pipe** to subscribe to Observables and automatically unsubscribe in templates.

Use ngFor with trackBy

You use the **ngFor** directive to iterate arrays in Angular templates. When you change an array, the complete DOM tree re-renders, which is not performance-wise. To solve this, you must use ngFor with trackBy, which uniquely identifies each DOM element and enables Angular to re-render only the modified element:

Top highlight

```
@Component({
  selector: 'my-app',
  template: `
   <li *ngFor="let product of products; trackBy:productById"></li>
  `
})
export class App {
  products:[];
   {id:0, name: “product 1”},
   {id:1, name: “product 2”}
  ];
```

```
productById(index, product){
   return product.id;
}
```

Conclusion

The Angular team has adopted best practices from the beginning by using TypeScript for Angular development, ensuring types safety, better error handling, and integrations with IDEs. Angular 12 has enabled the strict mode by default, ensuring you follow strict rules that help you build error-free and solid apps. In this article, we have seen some of the best practices that you can follow to build scalable and easily maintainable apps.

Angular JavaScript Webdev