How you can build clean and robust Angular application?

Arslan Ali Jan 3, 2020 · 14 min read

This article outlines the practices that I use in my application and is related to Angular, Typescript, RxJs and @ngrx/store. I'll also go through some general coding guidelines to help make the application cleaner.

1) trackBy

When using `ngFor` to loop over an array in templates, use it with a `trackBy` function which will return an unique identifier for each item.

Why?

When an array changes, Angular re-renders the whole DOM tree. But if you use `trackBy`, Angular will know which element has changed and will only make DOM changes for that particular element.

Before

```
<li *ngFor="let item of items;">{{ item }}</li>
```

After

```
// in the template

<li *ngFor="let item of items; trackBy: trackByFn">{{ item }}</li>

// in the component

trackByFn(index, item) {
   return item.id; // unique id corresponding to the item
}
```

2) const vs let

Related


Best Practices in Angular Application Development


Application Structure and Best Practices Of Angular—Part 1


It's so easy with Angular
It's a rainy day. I'm bored a…


Angular 12 Routing By Example
Throughout this tutorial, we…

2) const vs let

When declaring variables, use const when the value is not going to be reassigned.

Why?

Using `let` and `const` where appropriate makes the intention of the declarations clearer. It will also help in identifying issues when a value is reassigned to a constant accidentally by throwing a compile time error. It also helps improve the readability of the code.

Before

```
let car = 'ludicrous car';

let myCar = `My ${car}`;
let yourCar = `Your ${car};

if (iHaveMoreThanOneCar) {
   myCar = `${myCar}s`;
}

if (youHaveMoreThanOneCar) {
   yourCar = `${youCar}s`;
}
```

After

```
// the value of car is not reassigned, so we can make it a const
const car = 'ludicrous car';

let myCar = `My ${car}`;
let yourCar = `Your ${car};

if (iHaveMoreThanOneCar) {
   myCar = `${myCar}s`;
}

if (youHaveMoreThanOneCar) {
   yourCar = `${youCar}s`;
}
```

3) Pipeable operators

Use pipeable operators when using RxJs operators.

Why?

Pipeable operators are tree-shakeable meaning only the code we need to execute will be included when they are imported.

This also makes it easy to identify unused operators in the files.

Note: This needs Angular version 5.5+.

Before

```
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/take';

iAmAnObservable
    .map(value => value.item)
    .take(1);
```

After

```
import { map, take } from 'rxjs/operators';

iAmAnObservable
    .pipe(
       map(value => value.item),
       take(1)
     );
```

4) Isolate API hacks

Not all APIs are bullet proof — sometimes we need to add some logic in the code to make up for bugs in the APIs. Instead of having the hacks in components where they are needed, it is better to isolate them in one place — like in a service and use the service from the component.

Why?

This helps keep the hacks "closer to the API", so as close to where the network request is made as possible. This way, less of your code is dealing with the un-hacked code. Also, it is one place where all the hacks live and it is easier to find them. When fixing the bugs in the APIs, it is easier to look for them in one file rather than looking for the hacks that could be spread across the codebase.

You can also create custom tags like API_FIX similar to TODO and tag the fixes with it so it is easier to find.

5) Subscribe in template

Avoid subscribing to observables from components and instead subscribe to the observables from the template.

Why?

`async` pipes unsubscribe themselves automatically and it makes the code simpler by eliminating the need to manually manage subscriptions. It also reduces the risk of accidentally forgetting to unsubscribe a subscription in the component, which would cause a memory leak. This risk can also be mitigated by using a lint rule to detect unsubscribed observables.

This also stops components from being stateful and introducing bugs where the data gets mutated outside of the subscription.

Before

```
// // template

<p>{{ textToDisplay }}</p>

// component

iAmAnObservable
    .pipe(
       map(value => value.item),
       takeUntil(this._destroyed$)
     )
    .subscribe(item => this.textToDisplay = item);
```

After

```
// template

<p>{{ textToDisplay$ | async }}</p>

// component

this.textToDisplay$ = iAmAnObservable
    .pipe(
       map(value => value.item)
     );
```

6) Clean up subscriptions

When subscribing to observables, always make sure you unsubscribe from them appropriately by using operators like `take`, `takeUntil`, etc.

Why?

Failing to unsubscribe from observables will lead to unwanted memory leaks as the observable stream is left open, potentially even after a component has been destroyed / the user has navigated to another page.

Even better, make a lint rule for detecting observables that are not unsubscribed.

Before

```
iAmAnObservable
    .pipe(
       map(value => value.item)
     )
    .subscribe(item => this.textToDisplay = item);
```

After

Using `takeUntil` when you want to listen to the changes until another observable emits a value:

```
private _destroyed$ = new Subject();

public ngOnInit (): void {
    iAmAnObservable
    .pipe(
       map(value => value.item)
      // We want to listen to iAmAnObservable until the component is
destroyed,
       takeUntil(this._destroyed$)
     )
    .subscribe(item => this.textToDisplay = item);
}

public ngOnDestroy (): void {
    this._destroyed$.next();
    this._destroyed$.complete();
}
```

Using a private subject like this is a pattern to manage unsubscribing many observables in the component.

Using `take` ... the first ... emitted by the ... observable

Using `take` when you want only the first value emitted by the observable:

```
iAmAnObservable
    .pipe(
       map(value => value.item),
       take(1),
       takeUntil(this._destroyed$)
    )
    .subscribe(item => this.textToDisplay = item);
```

Note the usage of `takeUntil` with `take` here. This is to avoid memory leaks caused when the subscription hasn't received a value before the component got destroyed. Without `takeUntil` here, the subscription would still hang around until it gets the first value, but since the component has already gotten destroyed, it will never get a value — leading to a memory leak.

7) Use appropriate operators

When using flattening operators with your observables, use the appropriate operator for the situation.

switchMap: when you want to ignore the previous emissions when there is a new emission

mergeMap: when you want to concurrently handle all the emissions

concatMap: when you want to handle the emissions one after the other as they are emitted

exhaustMap: when you want to cancel all the new emissions while processing a previous emisssion

For a more detailed explanation on this, please refer to this article by Nicholas Jamieson.

Why?

Using a single operator when possible instead of chaining together multiple other operators to achieve the same effect can cause less code to be shipped to the user. Using the wrong operators can lead to unwanted behaviour, as different operators handle observables in different ways.

8) Lazy load

When possible, try to lazy load the modules in your Angular application. Lazy loading is when you load something only when it is used, for example, loading a component only when it is to be seen.

Why?

This will reduce the size of the application to be loaded and can improve the application boot time by not loading the modules that are not used.

Before

```
// app.routing.ts

{ path: 'not-lazy-loaded', component: NotLazyLoadedComponent }
```

After

```
// app.routing.ts

{
  path: 'lazy-load',
  loadChildren: 'lazy-load.module#LazyLoadModule'
}

// lazy-load.module.ts
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterModule } from '@angular/router';
import { LazyLoadComponent }   from './lazy-load.component';

@NgModule({
  imports: [
    CommonModule,
    RouterModule.forChild([
         {
             path: '',
             component: LazyLoadComponent
         }
    ])
  ],
  declarations: [
    LazyLoadComponent
  ]
})
export class LazyModule {}
```

9) Avoid having subscriptions inside subscriptions

Sometimes you may want values from more than one observable to perform an action. In this case, avoid subscribing to one observable in the subscribe block of another observable. Instead, use appropriate chaining operators. Chaining operators run on observables from the operator before them. Some chaining operators are: `withLatestFrom`, `combineLatest`, etc.

Before

```
firstObservable$.pipe(
   take(1)
)
.subscribe(firstValue => {
    secondObservable$.pipe(
        take(1)
    )
    .subscribe(secondValue => {
        console.log(`Combined values are: ${firstValue} &
${secondValue}`);
    });
});
```

After

```
firstObservable$.pipe(
    withLatestFrom(secondObservable$),
    first()
)
.subscribe(([firstValue, secondValue]) => {
    console.log(`Combined values are: ${firstValue} &
${secondValue}`);
});
```

Why?

Code smell/readability/complexity: Not using RxJs to its full extent,

suggests developer is not familiar with the RxJs API surface area.

Performance: If the observables are cold, it will subscribe to firstObservable, wait for it to complete, THEN start the second observable's work. If these were network requests it would show as synchronous/waterfall.

10) Avoid any; type everything;

Always declare variables or constants with a type other than `any`.

Why?

When declaring variables or constants in Typescript without a typing, the typing of the variable/constant will be deduced by the value that gets assigned to it. This will cause unintended problems. One classic example is:

```
const x = 1;
const y = 'a';
const z = x + y;

console.log(`Value of z is: ${z}`

// Output
Value of z is 1a
```

This can cause unwanted problems when you expect y to be a number too. These problems can be avoided by typing the variables appropriately.

```
const x: number = 1;
const y: number = 'a';
const z: number = x + y;

// This will give a compile error saying:

Type '"a"' is not assignable to type 'number'.

const y:number
```

This way, we can avoid bugs caused by missing types.

Another advantage of having good typings in your application is that it makes refactoring easier and safer.

Consider this example:

```
public ngOnInit (): void {
    let myFlashObject = {
        name: 'My cool name',
        age: 'My cool age',
        loc: 'My cool location'
    }
    this.processObject(myFlashObject);
}

public processObject(myObject: any): void {
    console.log(`Name: ${myObject.name}`);
    console.log(`Age: ${myObject.age}`);
    console.log(`Location: ${myObject.loc}`);
}

// Output
Name: My cool name
Age: My cool age
```

```
Location: My cool location
```

Let us say, we want to rename the property `loc` to `location` in `myFlashObject`:

```
public ngOnInit (): void {
    let myFlashObject = {
        name: 'My cool name',
        age: 'My cool age',
        location: 'My cool location'
    }
    this.processObject(myFlashObject);
}

public processObject(myObject: any): void {
    console.log(`Name: ${myObject.name}`);
    console.log(`Age: ${myObject.age}`);
    console.log(`Location: ${myObject.loc}`);
}

// Output
Name: My cool name
Age: My cool age
Location: undefined
```

If we do not have a typing on `myFlashObject`, it thinks that the property `loc` on `myFlashObject` is just undefined rather than that it is not a valid property.

If we had a typing for `myFlashObject`, we would get a nice compile time error as shown below:

```
type FlashObject = {
    name: string,
    age: string,
    location: string
}

public ngOnInit (): void {
    let myFlashObject: FlashObject = {
        name: 'My cool name',
        age: 'My cool age',
        // Compilation error
        Type '{ name: string; age: string; loc: string; }' is not
assignable to type 'FlashObjectType'.
        Object literal may only specify known properties, and 'loc'
does not exist in type 'FlashObjectType'.
        loc: 'My cool location'
    }
    this.processObject(myFlashObject);
}

public processObject(myObject: FlashObject): void {
    console.log(`Name: ${myObject.name}`);
    console.log(`Age: ${myObject.age}`)
    // Compilation error
    Property 'loc' does not exist on type 'FlashObjectType'.
    console.log(`Location: ${myObject.loc}`);
}
```

If you are starting a new project, it is worth setting `strict:true` in the `tsconfig.json` file to enable all strict type checking options.

11) Make use of lint rules

`tslint` has various options built in already like `no-any`, `no-magic-numbers`, `no-console`, etc that you can configure in your `tslint.json` to enforce certain rules in your code base.

Why?

Having lint rules in place means that you will get a nice error when you are doing something that you should not be. This will enforce consistency in your application and readability. Please refer here for more rules that you can configure.

Some lint rules even come with fixes to resolve the lint error. If you want to configure your own custom lint rule, you can do that too. Please refer to this article by Craig Spence on how to write your own custom lint rules using TSQuery.

Before

```
public ngOnInit (): void {
    console.log('I am a naughty console log message');
    console.warn('I am a naughty console warning message');
    console.error('I am a naughty console error message');
}

// Output
No errors, prints the below on console window:
I am a naughty console message
I am a naughty console warning message
I am a naughty console error message
```

After

```
// tslint.json
{
    "rules": {
        .......
        "no-console": [
             true,
             "log",    // no console.log allowed
             "warn"    // no console.warn allowed
        ]
   }
}

// ..component.ts

public ngOnInit (): void {
    console.log('I am a naughty console log message');
    console.warn('I am a naughty console warning message');
    console.error('I am a naughty console error message');
}

// Output
Lint errors for console.log and console.warn statements and no error
for console.error as it is not mentioned in the config

Calls to 'console.log' are not allowed.
Calls to 'console.warn' are not allowed.
```

12) Small reusable components

Extract the pieces that can be reused in a component and make it a new one. Make the component as dumb as possible, as this will make it work in more scenarios. Making a component dumb means that the component does not have any special logic in it and operates purely based on the inputs and outputs provided to it.

As a general rule, the last child in the component tree will be the dumbest of all.

Why?

Reusable components reduce duplication of code therefore making it

easier to maintain and make changes.

Dumb components are simpler, so they are less likely to have bugs. Dumb components make you think harder about the public component API, and help sniff out mixed concerns.

13) Components should only deal with display logic

Avoid having any logic other than display logic in your component whenever you can and make the component deal only with the display logic.

Why?

Components are designed for presentational purposes and control what the view should do. Any business logic should be extracted into its own methods/services where appropriate, separating business logic from view logic.

Business logic is usually easier to unit test when extracted out to a service, and can be reused by any other components that need the same business logic applied.

14) Avoid long methods

Long methods generally indicate that they are doing too many things. Try to use the Single Responsibility Principle. The method itself as a whole might be doing one thing, but inside it, there are a few other operations that could be happening. We can extract those methods into their own method and make them do one thing each and use them instead.

Why?

Long methods are hard to read, understand and maintain. They are also prone to bugs, as changing one thing might affect a lot of other things in that method. They also make refactoring (which is a key thing in any application) difficult.

This is sometimes measured as "cyclomatic complexity". There are also some TSLint rules to detect cyclomatic/cognitive complexity, which you could use in your project to avoid bugs and detect code smells and maintainability issues.

15) DRY

Do not Repeat Yourself. Make sure you do not have the same code copied into different places in the codebase. Extract the repeating code and use it in place of the repeated code.

Why?

Having the same code in multiple places means that if we want to make a change to the logic in that code, we have to do it in multiple places. This makes it difficult to maintain and also is prone to bugs where we could miss updating it in all occurrences. It takes longer to make changes to the logic and testing it is a lengthy process as well.

In those cases, extract the repeating code and use it instead. This means only one place to change and one thing to test. Having less duplicate code

only one place to change and one thing to test. Having less duplicate code shipped to users means the application will be faster.

16) Add caching mechanisms

When making API calls, responses from some of them do not change often. In those cases, you can add a caching mechanism and store the value from the API. When another request to the same API is made, check if there is a value for it in the cache and if so, use it. Otherwise, make the API call and cache the result.

If the values change but not frequently, you can introduce a cache time where you can check when it was last cached and decide whether or not to call the API.

Why?

Having a caching mechanism means avoiding unwanted API calls. By only making the API calls when required and avoiding duplication, the speed of the application improves as we do not have to wait for the network. It also means we do not download the same information over and over again.

17) Avoid logic in templates

If you have any sort of logic in your templates, even if it is a simple `&&` clause, it is good to extract it out into its component.

Why?

Having logic in the template means that it is not possible to unit test it and therefore it is more prone to bugs when changing template code.

Before

```
// template
<p *ngIf="role==='developer'"> Status: Developer </p>

// component
public ngOnInit (): void {
    this.role = 'developer';
}
```

After

```
// template
<p *ngIf="showDeveloperStatus"> Status: Developer </p>

// component
public ngOnInit (): void {
    this.role = 'developer';
    this.showDeveloperStatus = true;
}
```

18) Strings should be safe

If you have a variable of type string that can have only a set of values, instead of declaring it as a string type, you can declare the list of possible values as the type.

Why?

By declaring the type of the variable appropriately, we can avoid bugs while writing the code during compile time rather than during runtime.

Before

```
private myStringValue: string;

if (itShouldHaveFirstValue) {
   myStringValue = 'First';
} else {
   myStringValue = 'Second'
}
```

After

```
private myStringValue: 'First' | 'Second';

if (itShouldHaveFirstValue) {
   myStringValue = 'First';
} else {
   myStringValue = 'Other'
}

// This will give the below error
Type '"Other"' is not assignable to type '"First" | "Second"'
(property) AppComponent.myValue: "First" | "Second"
```

Bigger picture

State Management

Consider using @ngrx/store for maintaining the state of your application and @ngrx/effects as the side effect model for store. State changes are described by the actions and the changes are done by pure functions called reducers.

Why?

@ngrx/store isolates all state related logic in one place and makes it consistent across the application. It also has memoization mechanism in place when accessing the information in the store leading to a more performant application. *@ngrx/store* combined with the change detection strategy of Angular leads to a faster application.

Immutable state

When using *@ngrx/store*, consider using ngrx-store-freeze to make the state immutable. *ngrx-store-freeze* prevents the state from being mutated by throwing an exception. This avoids accidental mutation of the state leading to unwanted consequences.

Why?

Mutating state in components leads to the app behaving inconsistently depending on the order components are loaded. It breaks the mental model of the redux pattern. Changes can end up overridden if the store state changes and re-emits. Separation of concerns — components are view layer, they should not know how to change state.

Jest

Jest

Jest is Facebook's unit testing framework for JavaScript. It makes unit testing faster by parallelising test runs across the code base. With its watch mode, only the tests related to the changes made are run, which makes the feedback loop for testing way shorter. Jest also provides code coverage of the tests and is supported on VS Code and Webstorm.

You could use a preset for Jest that will do most of the heavy lifting for you when setting up Jest in your project.

Karma

Karma is a test runner developed by AngularJS team. It requires a real browser/DOM to run the tests. It can also run on different browsers. Jest doesn't need chrome headless/phantomjs to run the tests and it runs in pure Node.

Universal

If you haven't made your app a *Universal* app, now is a good time to do it. Angular Universal lets you run your Angular application on the server and does server-side rendering (SSR) which serves up static pre-rendered html pages. This makes the app super fast as it shows content on the screen almost instantly, without having to wait for JS bundles to load and parse, or for Angular to bootstrap.

It is also SEO friendly, as Angular Universal generates static content and makes it easier for the web crawlers to index the application and make it searchable without executing JavaScript.

Why?

Universal improves the performance of your application drastically. We recently updated our application to do server side rendering and the site load time went from several seconds to tens of milliseconds!!

It also allows your site to correctly show up in social media preview snippets. The first meaningful paint is really fast and makes content visible to the users without any unwanted delays.

Conclusion

Building applications is a constant journey, and there's always room to improve things. This list of optimisations is a good place to start, and applying these patterns consistently will make your team happy. Your users will also love you for the nice experience with your less buggy and performant application.

If you have any comment, question, or recommendation, feel free to post them in the comment section below!

JavaScript Angular Frontend Architecture Best Practices Application Development

Full Stack developer /Lead Developer/Project Manager/marslan.ali@gmail.com

Dec 31, 2019

Docker & Benefits of Docker

Read about the top benefits of Docker, why it became so popular for containerization, like rapid deployment, multi-cloud platforms, and security.

Today, there is a buzz all around about containerization and Docker. What exactly is Docker and how it is related to containerization? What are the top benefits of using Docker? Why did it become so popular? And what are the statistics and successful case studies related to Docker? In this article, I will answer all these questions.

What is Docker & How is it Related to Containerization

Running applications in containers instead of virtual machines is gaining