Mistakes I Made as an Angular Developer

Confessions: what I regret not doing as an angular developer and how you can avoid it!

Bharath Ravi Following
Jul 30, 2020 · 7 min read

Credit— malcoded.com

> "Every time you decide not to refactor bad code you've just written, you incur technical debt" — @codestandards

Being a junior developer when you are self-taught can be hard. Programming and grasping high-level concepts can be tricky even if you have a formal computer science degree. So, most of our time as a junior dev involves trying out things and learning basics. It might not be true if you are part of an organization where there is pre-training. But otherwise, we all know the mess we are/were in. That's totally fine. **The only way to write good code is to start with bad code!**

Top highlight

Most of my early angular applications were messy, bulky, and hard for anyone else to understand. My code may still be the same, but at least I had my learning over the years. Looking back I regret not learning/using these following practices in my angular applications.

Not taking performance and optimization into account

Photo by Austin Distel on Unsplash

This is a broad overview. Everything I explain in the rest of this article is the result of having not being bothered about performance and optimizations as an Angular dev.

When you are a junior dev, getting into a real project can be overwhelming. When I was starting off, my primary focus was only to get my work done. Optimizations and best practices were far from consideration. It took me a good long time to get my head to focus on what matters. I think it's common especially when you are self-taught.

Not Following the DRY principle

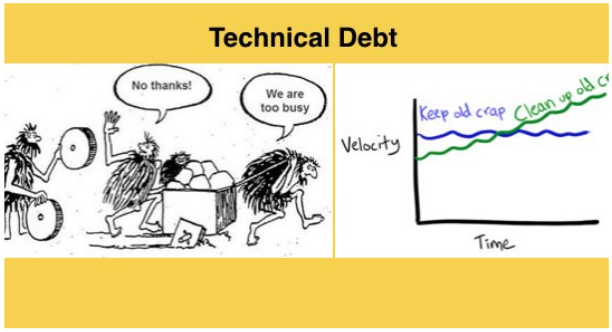
Photo by Nick Fewings on Unsplash

DRY(don't repeat yourself) is a well-known software development principle. It simply refers to not writing any same block of code, that does the same thing, more than once in your application. **It basically says to reuse everything possible instead of writing anything from scratch.**
Just for the reason that you have a custom button element that you already are using in multiple places, it cannot mean that you are following the principle. There are enormous different technics available in angular to make sure of this.

Learn more about the DRY principle here and here.

Not considering technical debt

Not considering technical debt



Credit — https://twitter.com/carnage4life

Let's admit it, deadlines drive us! We all have that manager who constantly asks for updates and says how unhappy the client is on pushing (unattainable) deadlines. So, what do you do? You forget the best practices, copy-paste styles from component to component, and release a working yet flawed application to production. Your manages is happy, the client is happy, the boss is happy!
Things turn upside down when more feature enhancement coming to the application. Now you obviously don't have that time in hand to refactor old code but you also you are pushed to do the same thing again!
This is every developer's dilemma, I suppose. It needs to have a structured approach throughout the organization to get this sorted rather than individual goals. Maybe you can take the lead to **maintain a single style guide going forward**.

Let's take this example code,

```
import { Component, OnInit } from "@angular/core";
import { HttpClient } from "@angular/common/http";

export class CartPage implements OnInit {
  constructor(private http: HttpClient) {}
  // ......
  ngOnInit() {
    this.cart = this.cartService.cart;
    this.getData(); // get employees data
  }
  getData() {
    this.employees$ = this.http.get('/api/employees'); // store in an observable to use in templ
  }
  // ......
}
```

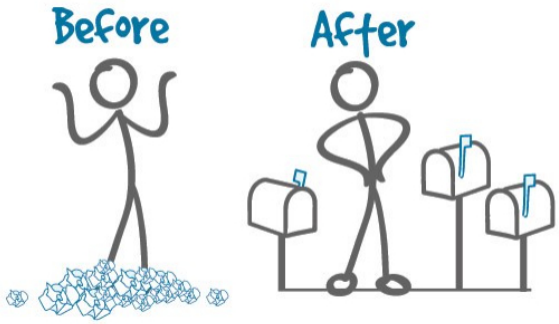
debt.ts hosted with ❤ by GitHub view raw

The above could be a potential code snippet that you have in one of your applications. You did a round of testing and this code works perfectly fine, you push the JIRA ticket to *done* and move to the next implementation.

The problem here is the practice of making `http` calls right on the component level. Instead, we could use a *service* to do this. The benefit of a *service* here is the independence of the data sources. Your component doesn't have to be aware of where the data is coming from. That can help in reusing, isolating, and customizing your code. That right there is a debt you're leaving in your code which in future will haunt you for obvious reasons.

Read more on technical debt here.

Not using `async` pipe and not unsubscribing observables



Credit — datanumen.com

It was late when I realized the importance of unsubscribing from observables. Most of my initial angular snippets looked something like this 👇

```
@Component({
  selector: "students",
  template: `<li *ngFor="let let student of students">{{student.name}}</li>`
})

export class StudentsComponent implements OnInit {
  ngOnInit() {
      this.studentService.loadStudents().subscribe(
      (arg) => {
        this.students = arg['data']
      },
      (err) => {
        console.log(err);
      }
    );
  }
}
```

not-async.ts hosted with ❤ by GitHub — view raw

like you can see, I almost never unsubscribed from observables that are not required anymore. **Here, clearly, the better approach to do this is to unsubscribe on `ngOnDestroy` or even better, to use `async` pipe.**

```
@Component({
  selector: "students",
  template: `<li *ngFor="let let student of (students | async)">{{student.name}}</li>`
})

export class StudentsComponent implements OnInit {
  ngOnInit() {
    this.students =
      this.studentService
      .loadAllCourses()
      .pipe(map((courses) => courses["data"]));
  }
}
```

with-async.ts hosted with ❤ by GitHub — view raw

Read more about unsubscribing here.

Not heavily using rxjs and reactivity

One of the very core philosophy that Angular follow is the concept of reactivity. The framework was built keeping reactivity in mind. Angular has been relying heavily on `rxjs` under the hood.

But for me, personally, `rxjs` was just another library. I have made use of it only when the framework demanded. To the max, I have been using Subjects for handling the global state.
Lately, I learned that **there are an amusing number of ways we can use `rxjs` to make our application reactive, cleaner, light, and simple**.

Let's take a simple example to understand the use case,

```
const listener = document.addEventEventListener(
   element, 'input', handler
);
setTimeout(() => listener(), 2000);

const handler = (event) => {
    const value = event.target.value.toUpperCase();
    if (!value.trim()) {
        return;
    }
    console.log(value);
};
```

wuthout-rxjs.ts hosted with ❤ by GitHub — view raw

Credits — Bits and Pieces

The above code retrieves the values from the DOM element, filter the value, and after two seconds stops listening.

Let's see how we can use `rxjs` to do the same.

```
import { map, filter, takeUntil } from 'rxjs/operators';
import { timer } from 'rxjs';

const values$ = fromEvent(element, 'input').pipe(
    map((event) => event.target.value.toUpperCase()),
    filter((value) => Boolean(value.trim())),
    takeUntil(timer(2000))
);
values$.subscribe(console.log);
```

with-rxjs.ts hosted with ❤ by GitHub — view raw

Credits — Bits and Pieces

We are creating the `value$` observable from the event using `fromEvent`,

mapping through it to convert to uppercase using `map`, filter whitespaces using `filter` and finally cancels subscription after 2 seconds using `takeUntil`.
Such an elegant way to do it! You can now subscribe to this observable anywhere on the template.

You can learn more about `rxjs` in angular here.

Not utilizing the power of Angular

Photo by Nikita Fox on Unsplash

Angular offers a lot, out of the box. That means there's a lot of opportunities that angular provides to write more performant and organized applications. For a simple example, I wasn't using reactive forms even in larger applications for a long time. I wasn't using `pipes` for a long time either.

Using these convenient and powerful features from angular can make your applications elegant and performant — This may include the followings and more,

1. Reactive forms
2. Services and Pipes
3. Custom directives and custom components
4. Decorators and Interfaces
5. Http Interceptors and resolvers
6. Route Guards

Not modularizing business logic

Photo by Victoria Strukovskaya on Unsplash

This isn't anything specific to Angular though modularizing your application makes a lot of sense in Angular. It's no secret that Angular bundle size is heavier comparing other SPA frameworks.
Typically the project requirement starts small. We as developers at that time may not really think and customize our application in a very modular manner. But eventually, more features will get added up and our application will grow bigger.

At some point, we will have a bigger bundle size and our application initial loading and performance gets affected. One of the best performance optimization available in Angular is Lazy loading. The very beginning step in lazy loading if to identity modules and now, you're stuck. You have no clue what components are related to what, how many services are global, or what needs to be in a shared module!

So, no matter how small you start, make sure you modularize your Angular application from the very beginning.

learn more about modularizing your Angular applications here.

Directly manipulating DOM

Photo by Jose Aragones on Unsplash

The basic rule of thumb when it comes to DOM(Document Object Model) is **not to directly manipulate DOM.** Angular provides a number of alternatives to do DOM operations. I have seen applications where people used `JQuery` to manipulate DOM. This is not ideal as our code is tightly coupled with the DOM APIs.

For example, if you're using server-side rendering(SSR) and you have something like `window.onScroll()` on your application, you are going to get an error thrown as the server does know about `window` object

an error throw as the server does know about `window` object.

There are different solutions Angular provides to do these more reliable and efficient. Like, `ElementRef`, `TemplateRef`, `ViewRef`, `ComponentRef` and `ViewContainerRef`.

Evidently, instead of doing this 👇

```
// ...
const element = document.getElementById('mobile-img');
element.style.display = 'none';
}
// ...
```

direct-dom-manipulate.ts hosted with ❤ by GitHub view raw

Let's do this 👇

```
import { ViewChild } from '@angular/core';
// ...

@ViewChild('mobileImg') element: HTMLElement;
// ...

this.renderer.setStyle(this.element, "display", "none");
// ...
```

dom-manipulation.ts hosted with ❤ by GitHub view raw

That's just one example, the basic idea of not manipulating DOM directly is the concept of platform independence. Our Angular application can run on a browser, on a mobile platform, or inside a web worker.

Read more on Angular DOM manipulations here.

Takeaways

1. Keep the performance of your application in mind when architecting your Angular application, no matter how big or small your application can get.
2. Follow the DRY principle.
3. Keep your technical debt as low as possible. Always target to leave nothing unnecessary behind.
4. Use `async` pipe or always remember to unsubscribe from observable once they are not in use.
5. Use `rxjs` when possible.
6. Modularize your application from the beginning.
7. Stop directly manipulating DOM.

. . .

For more articles like this 👇

Bharath Ravi | javaScript Articles

Competent Articles to Level Up Your Web Skills. Personal blog of

4. Use `async` pipe or always remember to unsubscribe from observable

once they are not in use.

5. Use `rxjs` when possible.
6. Modularize your application from the beginning.
7. Stop directly manipulating DOM.

. . .

For more articles like this 👇

4. Use `async` pipe or always remember to unsubscribe from observable once they are not in use.
5. Use `rxjs` when possible.
6. Modularize your application from the beginning.
7. Stop directly manipulating DOM.

. . .

For more articles like this 👇

Bharath Ravi | javaScript Articles

Competent Articles to Level Up Your Web Skills. Personal blog of