ANGULAR UNIVERSITY

PREMIUM QUALITY
ANGULAR TUTORIALS

angular-university.io

3 Common Rxjs Pitfalls that you might find while building Angular Applications

If you are building an RxJS Angular service layer for the first time, you will likely run into one of these scenarios in the first few services that you build.

In this post we will go over some common trouble situations, explain why the situation occurs and how to fix it. If you have used RxJs before, then try to guess the problem in each scenario, just for fun!

Pitfall 1 - Nothing Happens

We have written our newest service method, all the data will be saved to the backend in one click of a button:

```
@Injectable()
export class LessonsService {

    constructor(private http: Http) {

    }

    createLesson(description) {

        const headers = new Headers();
        headers.append('Content-Type', 'application/json; charset=utf-8');
        headers.append('X-Requested-With','XMLHttpRequest');

        this.http.post('/lessons', JSON.stringify({description}), headers);
    }

```

```
}
```

We will plugin a messages component and an error handling service later to show any error messages that might occur, but right now we will simply trigger the save operation via a click handler:

```
@Component({
    selector:'app',
    template: `
        <button (click)="save()">Save Lesson</button>
        <lessons-list [lessons]="lessons"></lessons-list>
        `
})
export class App {

    constructor(private lessonsService: LessonsService) {
    }

    save() {
        this.lessonsService.createLesson("Introduction to Components");
    }

}
```

And there we have it, the Save functionality is implemented. Except that when we try this out, try to guess what happens ?

...

Nothing happened :-) ! But the method is being called, or so we conclude after some debugging. So what's going on here ?

Pitfall 1 Explanation

An observable itself is just a definition of a way to handle a stream of values. We can think of it as something close to a function. Creating an

observable is somewhat similar to declaring a function, the function itself is just a declaration. Calling the function is something entirely different, as defining a function does not execute its code.

We need to call the function in order for something to happen, and that is also the case with an Observable: we need to subscribe to it in order for it to work!

So in order to fix this issue, we can do the following at the level of the service:

```
1
2  import {Injectable} from "@angular/core";
3  import {Http, Headers} from "@angular/http";
4  import {xhrHeaders} from "./xhr-headers";
5
6
7  @Injectable()
8  export class LessonsService {
9
10     constructor(private http: Http) {
11
12     }
13
14     createLesson(description) {
15         const headers = new Headers();
16         headers.append('Content-Type', 'application/json; charset=utf-8');
17         headers.append('X-Requested-With','XMLHttpRequest');
18         return this.http.post('/lessons', JSON.stringify({description}), he
19     }
20
21 }
```

Notice that now we are returning the Observable that we created, while before the Observable was created and the method returned without the Observable being subscribed to.

In the calling component, we can now add a subscription to this Observable:

```
@Component({
    selector:'app',
    template: `...`
})
export class App {

    constructor(private lessonsService: LessonsService) {

    }

    save() {
        this.lessonsService.createLesson("Introduction to Components")
            .subscribe(
                () => console.log('lesson saved successfully'),
                console.error
            );
    }

}
```

Now that we have a subscription, the HTTP lesson creation POST call will be sent to the backend as expected, which solves the issue.

In order to avoid the issue we need to ensure that we subscribe to the Observables that are returned by the HTTP service, either directly or via the `async` pipe.

But sometimes the issue is that its not one HTTP call that is not made, by the contrary the issue is that too many HTTP calls are made!

Pitfall 2 - Multiple HTTP requests

Speaking of the async pipe, let's give it a try, and pass it in some observable that we get back from the service layer. The observable will emit a list of lessons:

```
1
2  @Injectable()
3  export class LessonsService {
4
5      constructor(private http: Http) {
6
7      }
8
9      loadLessons() {
10         return this.http.get('/lessons')
11             .map(res => res.json());
12     }
13
14 }
15
```

But wait, you also want some error handling just in case the call fails:

```
1  @Component({
2      selector:'app',
3      template: `
4          <button (click)="save()">Save Lesson</button>
5          <lessons-list [lessons]="lessons$ | async"></lessons-list>
6          `
7  })
8  export class App implements OnInit {
9
10     lessons$: Observable<Lesson[]>;
11
12     constructor(private lessonsService: LessonsService) {
13
14     }
15
16     ngOnInit() {
17         this.lessons$ = this.lessonsService.loadLessons();
```

```
18
19          this.lessons$.subscribe(
20              () => console.log('lessons loaded'),
21              console.error
22          );
23      }
24  }
25
```

So we add a subscription to handle the error, and pass in the observable to the async pipe. So what could have gone wrong here ? Nothing, so we try this out and everything is working. But for whatever reason you decide to have a look at the network tab, as you usually leave the Dev Tools opened during development.

And what do you see ?

```
... /lessons GET 200 OK
... /lessons GET 200 OK
```

That's right, duplicate HTTP requests ! The request for reading the data is being issued twice. So what is going on here ?

Pitfall 2 Explanation

It turns out that as the Observable is just a definition, let's remember that in a sense its something close to a function declaration, if we subscribe to it multiple times this means that each time a new HTTP request will be issued, one for each subscription.

Just like calling a function multiple times gives you multiple return values.

And this not what we where trying to implement, we want to issue only one HTTP request, emit the data and handle errors if they occur otherwise print the data to the screen. But what is happening instead is:

- one request gets back and the data is shown on the screen
- another request is sent and if in error a message will be displayed

With this scenario if the errors are intermittent it could even happen that the first request goes through and the second gives an error. So how to fix this ? This is one way to do it in this scenario:

```
ngOnInit() {
    this.lessons$ = this.lessonsService.loadLessons().publishLast().refCount
    this.lessons$.subscribe(
        () => console.log('lessons loaded'),
        console.error
    );
}
```

The `share` operator also works well in this particular case, note that this is just a simple example to illustrate the problem.

Its important to keep in mind this multiple subscription situation when building service layers with the HTTP module. Another situation that happens not so much with the HTTP module but with libraries like AngularFire has to with Observable completion.

Pitfall 3 - Router gets Blocked ?

Imagine that you are building a Router Data Resolver using AngularFire and not the Angular HTTP Library. This issue wouldn't happen with the

HTTP library as we will see in a moment.

Also this router scenario is just an example of the underlying problem, that could manifest itself in other ways. The goal of the data resolver is to load some data before the routing transition completes, this way the data is already available when the new component gets rendered.

Let's have a look at some router configuration for a data resolver:

```
{
  path: 'lessons/:id',
  children: [
      {
          path: 'edit',
          component:  EditLessonComponent,
          resolve: {
            lesson: LessonResolver
          }
      }
  ]
}
```

This configuration means that if we hit a url `/edit` we will load a `EditLessonComponent`, but before we are going to load some data: a list of lessons. For that we define a router data resolver:

```
import {Resolve, ActivatedRouteSnapshot, RouterStateSnapshot} from "@angula
import {Lesson} from "./lesson";
import {Observable} from "rxjs/Rx";
import {Injectable} from "@angular/core";
import {LessonsService} from "./lessons.service";


@Injectable()
export class LessonResolver implements Resolve<Lesson> {


    constructor(private lessonsService: LessonsService) {

    }

    resolve(route:ActivatedRouteSnapshot,
            state:RouterStateSnapshot):Observable<Lesson> {

        return this.lessonsService
            .findLessonByUrl(route.params['id']);
    }

}

```

So we try to trigger the router transition where the resolver is applied, and then try to guess what happens ?

...

Again, nothing happened ! The router transaction did not occur, we stay at the original screen and the UI does not get updated. Its like the transition was never triggered. What could have happened here ?

Pitfall 3 Explanation

This has to do with both the way the router works and the way the AngularFire observable API works. The router in this case will take the observable that we returned and it will subscribe to it. Then it will wait for the data to arrive and for the Observable to complete, and only then the route transition is triggered.

This is because the router needs the data to be available before instantiating the target component. After all one of the goals of the data resolver is to retrieve the data so that the data is already available when the target component gets created.

So this means that if we return an observable in a data resolver, we need to ensure that it completes. How can we do that ? In this case we can for example simply call `first()` and create a derived observable that completes after the first value is emitted:

```
resolve(route:ActivatedRouteSnapshot, state:RouterStateSnapshot):Observable<
  return this.lessonsService
      .findLessonByUrl(route.params['id'])
      .first();
}

```

Observable completion is an important thing to keep in mind

With this the route navigation will complete successfully. This router scenario is just one possible manifestation of what can happen if we accidentally overlook Observable completion.

Completion is something important to happen in many situations that relate to Observables, take for example the RxJs `concat` operator: If we

pass it an Observable that never completes, the concatenation would not happen.

In the case of the Angular HTTP module, the returned observables emit one value and then complete, so this is why the situation above would not happen using the HTTP Module.

But in the case above, because the AngularFire observables are meant to continuously emit the server pushed data changes of the realtime database, they don't complete after a value is emitted so that is why the issue happened.

Conclusions

When writing RxJs code, the majority of the times things just work as expected, but there are a couple of situations that we will probably run into occasionally, especially in the beginning.

If we happen to find ourselves in a situation where something is not working as expected, asking these questions will help us find the solution:

- Has this observable been subscribed to ?
- how many subscriptions does this observable have ?
- when does this observable complete, does it complete ?
- Do I need to unsubscribe from this Observable ?