Why you HAVE to unsubscribe from Observable

Higher Order Observables, Subscribing and Unsubscribing

🦊 Reactive Fox 🚀 Follow

Mar 12 · 12 min read

> *Look at the following example and ask yourself: do I need to unsubscribe…or not?*

```
class TokenInterceptor implements HttpInterceptor {
  intercept(
    request: HttpRequest<any>,
    next: HttpHandler
  ): Observable<HttpEvent<any>> {
    return interval(1000).pipe(
      map(() => next.handle(request))
    );
  }
}

const subscription = http.get('https://.../api/get')
  /** Did you unsubscribe? */
  .subscribe(() => subscription.unsubscribe());
```

Yes, many articles already exist on this subject. But, it's important to understand that any code that does not unsubscribe can lead to memory leaks and performance regression. So, let's figure out once and for all when and when not to unsubscribe by examining a few examples.

· · ·

Do not subscribe

The easiest way to **unsubscribe from an Observable** is not to subscribe in the first place. Yes, that's it! Just **don't subscribe** at all. "But how do I get all the data I need?" I hear you say. Easy. In **Angular**, you can just pass your **Observable** to **AsyncPipe**.

AsyncPipe unwraps a value from the **Observable** directly in the template and automatically unsubscribes when the **View** is destroyed.

It looks something like this:

```
<item *ngFor="let item of source | async"></item>
<items [data]="source | async"></items>
Or, that: {{ source | async }}
```

Here. You didn't subscribe, so there is no need to unsubscribe. Hurray!

Now we know how to handle the **Observable**, but what would be the best way to work with **HttpClient**?

. . .

Working with HttpClient

When working with **HttpClient** we might face the situation where we just can’t use **AsyncPipe** for the **Observable**. To dive deeper, let’s look at some examples:

- re-sending requests when parameters are being changed
- receiving data in parallel from several streams
- sending individual requests
- creating, deleting and updating data

• • •

Simple request with subscription and unsubscribing

Let’s consider a situation when there’s an **Http Request** and we need to get some server data. Like this:

```
const request = http.get('https://.../api/get');
```

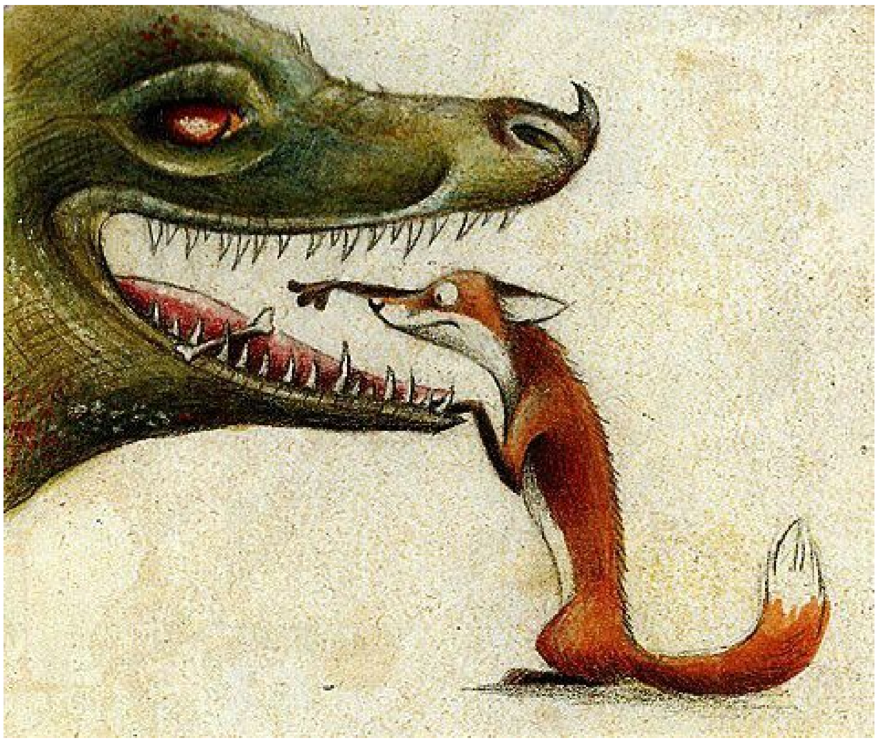
Perfect, now we have a request, so we can subscribe to it. Let’s just do it:

```
request.subscribe((response) => console.log(response));
```

We successfully subscribed, and made a very important mistake. We forgot to unsubscribe! But you may ask, “Why should I do this if **HttpClient** completes the **Observable** after the data is received?”

. . .

Common subscription handling mistakes



Suppose we want to not only receive data, but also to assign it to a component variable:

```
request.subscribe((response) => this.response = response);
```

This is what might happen in this case. If after creating the request but before receiving an answer from the back-end, you deem the component unnecessary and destroy it. Your subscription will maintain the reference to the component thus creating a chance for **memory leaks**.

Here’s another example where a second subscription is created or an asynchronous action is performed inside a subscription:

```
request.subscribe((response) =>
  // Using setInterval to start a timer in the component
  setInterval(() => ..., 1000)
);


request.subscribe((response) =>
  // Using interval to start a timer in the component
  // Subscribing within a subscription is a mega anti-
pattern :)
  interval(1000).subscribe(() => ...)
);
```

If you're still on the fence about unsubscribing from **HttpClient** take a look at this example:

```
class TokenInterceptor implements HttpInterceptor {
  intercept(
    request: HttpRequest<any>,
    next: HttpHandler
  ): Observable<HttpEvent<any>> {
    return interval(1000).pipe(
      map(() => next.handle(request))
    );
  }
}

const subscription = http.get('https://.../api/get')
  /** Did you unsubscribe? */
  .subscribe(() => subscription.unsubscribe());
```

Here if we don't end the subscription, the stream will **never terminate**. Are you absolutely sure you will never face a situation like this?

In real life development you can't be sure this will never happen. As a result you might end up with a **real memory leak** in your application. Create several subscriptions like that, and your memory will never be cleared, the browser will crunch numbers for no useful reason until it crashes.

• • •

Operators take, first, takeWhile…

… **ARE USELESS**! When using them, there’s still no guarantee that the stream will receive any data. At the same time your code has side effects and keeps references to your components in memory. It can also lead to errors when dealing with entities that have been destroyed along with a component.

For example, you can’t be sure your stream receives the emitted response. Or that your event handler doesn’t crash because your **View** had been destroyed.

> *If you don’t need something, just destroy it and let it die.*

. . .

He who has subscribed shall unsubscribe

To dispose of a subscription you can simply use the **unsubscribe()** method:

```
subscription: SubscriptionLike;

ngOnInit() {
  this.subscription = request.subscribe(...);
}

ngOnDestroy() {
  if (this.subscription) {
    this.subscription.unsubscribe();
  }
}
```

That's it. Now we can be sure our subscription will always be ended and we're safe from a memory leak! Note that we **absolutely have to** create the subscription during the initialization of a component and later when it's being destroyed we need to check if the subscription is still live, cancel it and clear all references to it. That sure sounds pretty tedious, don't you think? Check this out:

```
ngOnInit() {
  this.subscription1 = request1.subscribe(...);
  this.subscription2 = request2.subscribe(...);
  // ...
  this.subscriptionX = requestX.subscribe(...);
}

ngOnDestroy() {
  if (this.subscription1) {
    this.subscription1.unsubscribe();
  }
  if (this.subscription2) {
    this.subscription2.unsubscribe();
  }
```

```
    // ...
    if (this.subscriptionX) {
      this.subscriptionX.unsubscribe();
    }
  }
```

It looks terrible! Let’s try to simplify unsubscribing.

. . .

Storing all subscriptions in a list

The simplest way to end all subscriptions at once is to store them all in a list and then just go through it. Let’s just do it:

```
subscriptions: SubscriptionLike[] = [];


ngOnInit() {
  this.subscriptions.push(request.subscribe(...));
  this.subscriptions.push(request.subscribe(...));
  this.subscriptions.push(request.subscribe(...));
}

ngOnDestroy() {
  this.subscriptions.forEach(
    (subscription) => subscription.unsubscribe());
}
```

Ok, fine, we reduced the number of operations and made our code more compact, but it’s still complex! Let’s try something else.

. . .

Using Subscription add

We can add a **teardown effect** to each subscription by calling **subscription.add()**. It will be invoked when the subscription is destroyed. Let’s use this:

```
subscriptions: Subscription = new Subscription();
```

```
ngOnInit() {
  this.subscriptions.add(request.subscribe(...));
  this.subscriptions.add(request.subscribe(...));
  this.subscriptions.add(request.subscribe(...));
}

ngOnDestroy() {
  this.subscriptions.unsubscribe();
}
```

Now we only work with a single subscription. All other subscriptions are added to it, and destroying the component also cancels all of them. But we still have to manage them manually! On to the next idea.

• • •

Using takeUntil

The **takeUntil** operator works as follows: it subscribes to a data source and takes an **Observable** which tells the stream when to end the subscription.

We'll use **ReplaySubject** to emit the last message in case the subscription is ended after the component is destroyed.

Voila, now let's try to use it. The simplest case might look like this:

```
destroy: ReplaySubject<any> = new ReplaySubject<any>(1);

ngOnInit() {
  request.pipe(takeUntil(this.destroy)).subscribe();
  request.pipe(takeUntil(this.destroy)).subscribe();
  request.pipe(takeUntil(this.destroy)).subscribe();
}

ngOnDestroy() {
  this.destroy.next(null);
}
```

And we still have to handle **destroy**! There are a few more ways to tackle our problem, let's go over them.

• • •

Using the AutoUnsubscribe decorator

Let's use the **AutoUnsubscribe** decorator from this library: https://github.com/NetanelBasal/ngx-auto-unsubscribe. Example incoming:

```
@AutoUnsubscribe()
@Component({ selector: '...', template: '...' })
class ExampleComponent implements OnDestroy {
  subscriptions: Subscription = new Subscription();

  ngOnInit() {
    this.subscriptions.add(request.subscribe(...));
    this.subscriptions.add(request.subscribe(...));
    this.subscriptions.add(request.subscribe(...));
  }

  ngOnDestroy() {}
}
```

Our problem is now partially solved: the decorator will automatically dispose subscriptions. In this particular case they are stored in **this.subscriptions**, where the rest of subscriptions go. But we still **have to** create the empty and useless hook **ngOnDestroy()**.

Let's keep improving.

. . .

Using the untilDestroyed operator

The **untilDestroyed()** operator was created by Netanel Basal, here's a link to the package: https://github.com/NetanelBasal/ngx-take-until-destroy. It works similarly to **takeUntil()** but it uses a component reference instead of **Observable**. Using it yields code like this:

```
ngOnInit() {
  request.pipe(untilDestroyed(this)).subscribe();
  request.pipe(untilDestroyed(this)).subscribe();
  request.pipe(untilDestroyed(this)).subscribe();
}
```

```
ngOnDestroy() {}
```

This is so much better! But we still have to create the **ngOnDestroy()** hook! Any more solutions?

· · ·

Using the NgOnDestroy service

The idea here is using the **ngOnDestroy()** hook inherent to all services for automatic unsubscribing. Here's an example of a service set up like that: https://stackblitz.com/edit/angular-auto-unsubscribe-service. Let's try using it:

```
@Component({
  selector: '...',
  template: '...',
  providers: [ NgOnDestroy ]
})
class ExampleComponent {
  constructor(@Self() private destroy: NgOnDestroy) {}

ngOnInit() {
    request.pipe(takeUntil(this.destroy)).subscribe();
    request.pipe(takeUntil(this.destroy)).subscribe();
    request.pipe(takeUntil(this.destroy)).subscribe();
  }
}
```

Now we don't need to create the extra **ngOnDestroy()** hook, but we do need to add services to the list of providers of every component that uses **NgOnDestroy**, and also to read it using **DI** in **constructor**.

And that sums up all the main ways of **ending subscriptions properly**. Don't you think it over complicates things?

· · ·

Don't subscribe!

I mean it.

You probably have better things to do in your life than managing subscriptions.

> *Just stop subscribing!*

. . .

Subscriptionless subscriptions

If you haven't read my previous article, now is a great time to do it. It'll help you to understand the context of this next part.

RxJS in Practice

Writing our own Ngrx

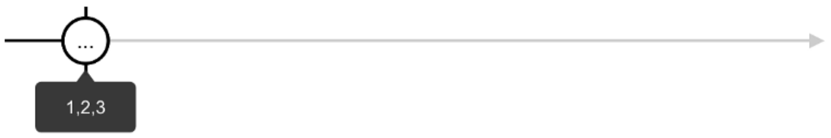
blog.angularindepth.com

We looked at the most basic case of **HttpClient** when we just received data. But we might need to wait for the termination of the previous request? Or to replace it. Or to perform everything asynchronously and process all the results at once. What then?

For cases like this, **RxJS** provides us with convenient **Higher Order Observables**. Here are some of these operators that you're most likely to come across when developing your applications: **switchMap**, **mergeMap**, **concatMap**, **exhaustMap**. Let's explore them one by one and see what their purpose is and when to use them.

Why don't we start with the simplest of all. We'll send requests to our imaginary API from this basic **load()** function that **emulates** the request.

```
function load(): Observable<number[]> {
  return of([ 1, 2, 3 ]).pipe(
    delay(1000)
  );
}
```



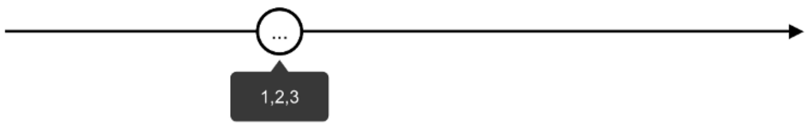
https://rxviz.com/v/58GYqA4O

Now let’s use the data loading function in the **load$** effect. To do this we’ll need **switchMap()**.

switchMap()

switchMap() switches to a new stream every time it receives a message. If it’s already operating on a stream the moment it receives the message, the old stream is terminated.

```
load$: Observable<Action> =
  actions$.pipe(
    ofType('load'),
    switchMap(() => load())
  );
```

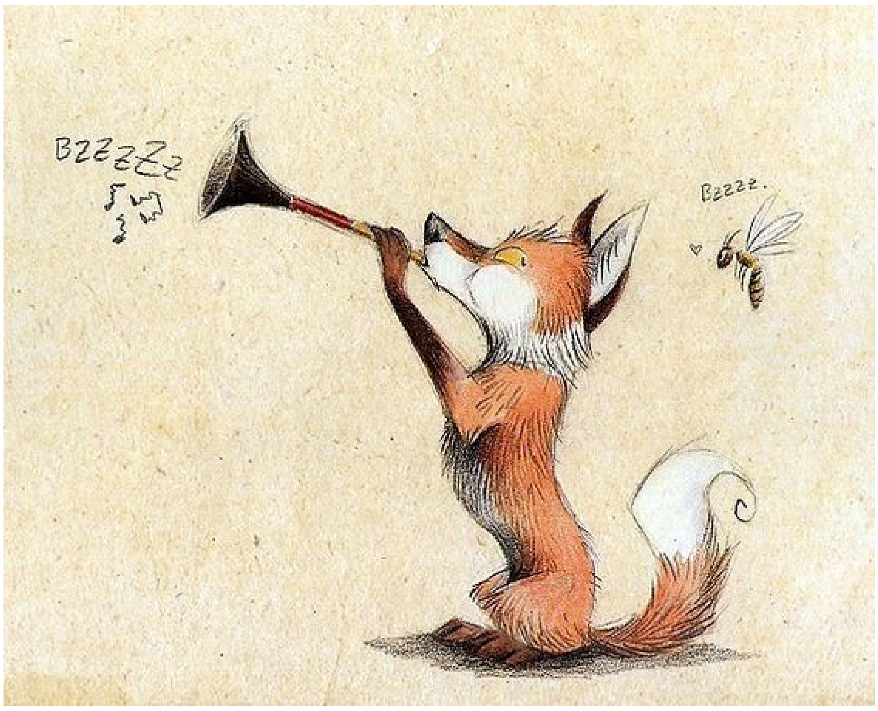


https://rxviz.com/v/MogdZRqO

So far so good! We’ve created our first subscriptionless subscription! But what if we want to add an event model to our application?

• • •

Handling events



The most popular events in applications are **Create**, **Update**, and **Delete**. Let’s implement a simple function to emulate out event channel:

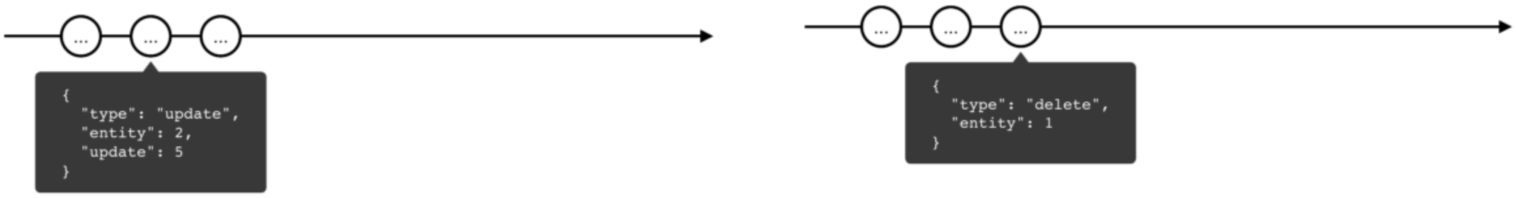
```
events: Event[] = [
  { type: 'create', entity: 4 },
  { type: 'update', entity: 2, update: 5 },
  { type: 'delete', entity: 1 }
];

events$: Observable<Event> = from(events).pipe(
  concatMap(pipe(of, delay(1000))),
  share()
);

function channel(type: string): Observable<Event> {
  return events$.pipe(
    filter(event => event.type === type)
  );
}
```

Here’s what we have done: we declared a test set of events that our channel will send, and wrote a function that reads events of a specific type from the channel. Now we need an effect that would subscribe to events. Let’s call it **channel$**. For this we’ll be using the **mergeMap()** operator.

mergeMap()

The **mergeMap()** method combines new stream messages with the old stream messages that it’s already working with. It’s used when we don’t care about the order of messages, but we do need to receive messages from all streams.

```
channel$: Observable<Action> =
  actions$.pipe(
    ofType('subscribe event'),
    mergeMap((action) => channel(action.payload))
  );
```



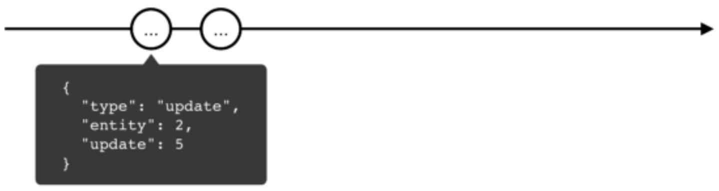
{ "type": "update", "entity": 2, "update": 5 }
{ "type": "delete", "entity": 1 }

https://rxviz.com/v/2ORZezGO

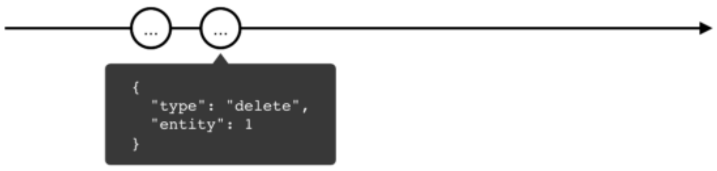
We’re now receiving messages from the channel! But there’s a problem: we can’t unsubscribe from events when we don’t need them anymore. E.g. if we don’t want to receive any more Create events, we can’t stop them. Let’s fix this.

This is where **takeUntil()** comes into play. Let’s just cut into the command stream and wait for the application to initiate unsubscribe.

```
function ofPayload(payload) {
  return filter((action) => payload === action.payload);
}

channel$: Observable<Action> =
  actions$.pipe(
    ofType('subscribe event'),
    mergeMap((action) => {
      const unsubscribe$ = actions$.pipe(
        ofType('unsubscribe event'),
        ofPayload(action.payload)
      );

      return channel(action.payload).pipe(
        takeUntil(unsubscribe$)
      );
    })
  );
```



```
{
  "type": "update",
  "entity": 2,
  "update": 5
}
```



```
{
  "type": "delete",
  "entity": 1
}
```

https://rxviz.com/v/VOKXy6RJ

Now, we might not only subscribe to specific events, but also unsubscribe from them! And our application handles data loading, but let’s assume our back-end is not capable of processing a lot of specific requests.

• • •

Processing queues

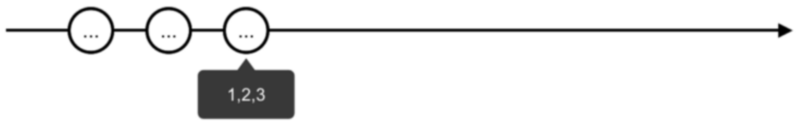
We've already implemented a **load()** function that emulates data loading from the network. We've also implemented a **load$** effect for it that enables the store to load the data.

And it's only **now** that our back-end devs inform us they can't process data loading fast enough, and all these requests have overloaded the back-end. Let's fix that using **concatMap()**!

concatMap()

The **concatMap()** method combines new stream messages with the old stream messages it's been working with. It also waits for the previous stream to terminate before launching a new one. We use it when we care about the order of events and we need to receive all of them.

```
load$: Observable<Action> =
  actions$.pipe(
    ofType('load'),
    concatMap(() => load())
  );
```



https://rxviz.com/v/dJPMKZY8

Perfect! The back-end guys are happy, the front-end does not send all the requests at once, but there’s a BUT here. **Don’t do this in a real application, this is a far-fetched example. :)** At the very least we’d need to cancel all previous events except the first and the last ones.

Buttons! We’ve been neglecting buttons!

. . .

Handling stream losses

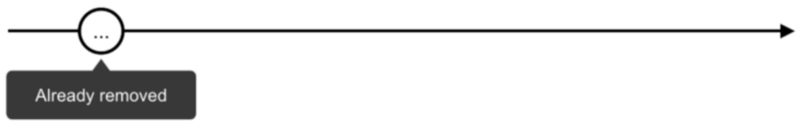
One of the key cases of our application is to handle data interactions properly. E.g. if we have a form or a button that updates an entity, we wouldn’t want to send the same request to the server twice, as it could lead to all sorts of errors.

Let’s write a simple method that will **emulate** deleting an entity or return an error if it has been already deleted.

```
const removedItems = new Set();

function remove(item: number): Observable<Response> {
  const removed = !removedItems.has(item);
  removedItems.add(item);

  const result = removed ?
    of(removed) :
    throwError('Already removed');

  return timer(1000).pipe(switchMapTo(result));
}
```

And let’s try using the **switchMap()** operator for starters:

```
const remove$ =
  actions$.pipe(
    ofType('remove'),
    switchMap((action) =>
      remove(action.payload).pipe(
        catchError((error) => of(error))
      )
    )
  );
```



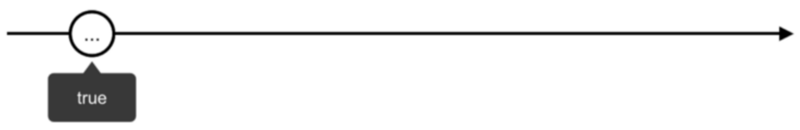
https://rxviz.com/v/0oqM1d1o

As you see, we’re getting an error when we try to send two identical deletion requests. That’s because the second request is overriding the first one. Let’s fix it by using the **exhaustMap()** operator.

exhaustMap()

exhaustMap() processes the new stream only when it's not busy with another one. I.e. if it's already processing a stream the moment it receives a new message, the new stream will be ignored and will never get processed.

```
const remove$ =
  actions$.pipe(
    ofType('remove'),
    exhaustMap((action) =>
      remove(action.payload).pipe(
        catchError((error) => of(error))
      )
    )
  );
```

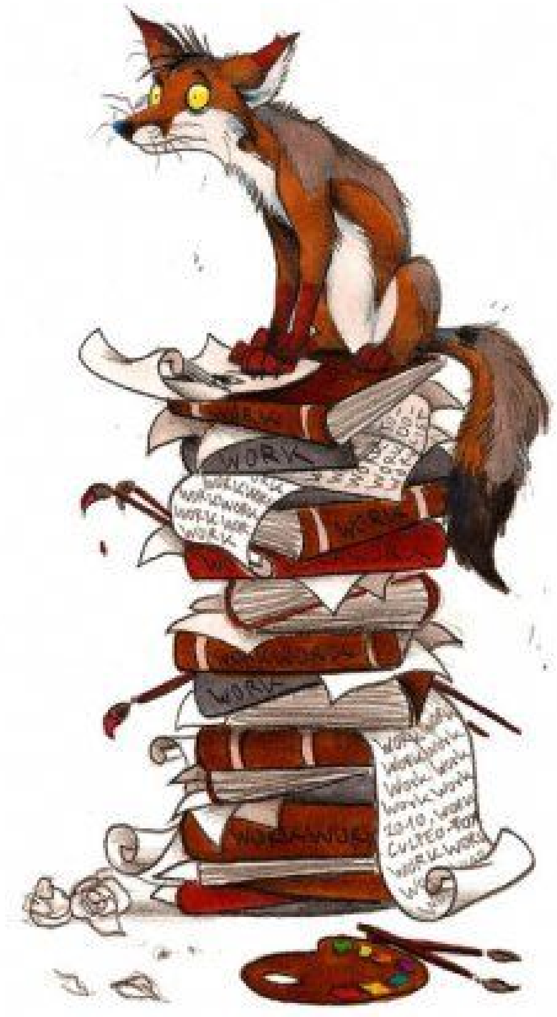


https://rxviz.com/v/EOe2RXjJ

Now we only remove the entity once and don't get any more errors!

• • •

Let's sum it up, shall we?



Today we've talked about subscriptions and unsubscribing, touched upon the topic of **Higher Order Observables**, taught our store to deal with events and to skip repeated actions that can lead to errors. What is the moral here?

Do not manually subscribe! Unless you really have to.

And if you did, **don't forget to unsubscribe**, whether we're talking about **HttpClient** or not.

In most cases use **Higher Order Observables**. Use **AsyncPipe** in your component's template to create a final subscription. This will make your code lighter, and you won't need to worry about ending subscriptions. You'll be able to implement complex solutions in **RxJS** using much more readable declarative style.

Also note how each example has a link to **rxviz.com** in the screenshot description. You can conduct your own experiments there.

. . .

Oh, and most importantly, **takeUntil()** goes in the very end:

RxJS: Avoiding takeUntil Leaks

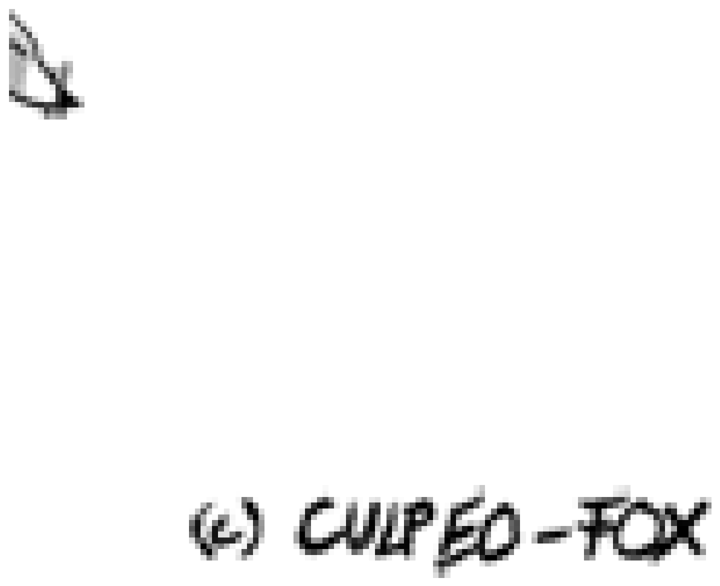
How to unsubscribe without leaking subscriptions

blog.angularindepth.com

```
// Wrong!
pipe(
  takeUntil(read),
  switchMapTo(story)
);
```

```
// Ok!
pipe(
  switchMapTo(story),
  takeUntil(read)
);
```

. . .



You can always reach me via **Twitter**.

Don’t forget to follow me on **Twitter**, **GitHub**, and **Medium**, 👏 **Clap Clap** 👏 this story, and subscribe to **Angular in Depth** without **takeUntil()**!