6 Ways to Unsubscribe from Observables in Angular

A review of the different ways you can unsubscribe from Observables in Angular

Chidume Nnamdi Follow

Jan 16, 2020 · 10 min read

In this post, I'll review the different ways you can unsubscribe from Observables in Angular apps.

The Downside to Observable Subscription

Observables have the `subscribe` method we call with a callback function to get the values emitted into the Observable. In Angular, we use it in Components/Directives especially in the router module, NgRx, HTTP module.

Now, if we subscribe to a stream the stream will be left open and the callback will be called when values are emitted into it anywhere in the app until they are closed by calling the unsubscribe method.

```
@Component({...})
export class AppComponent implements OnInit {
    subscription: Subscription
    ngOnInit () {
        var observable = Rx.Observable.interval(1000);
        this.subscription = observable.subscribe(x =>
console.log(x));
    }
}
```

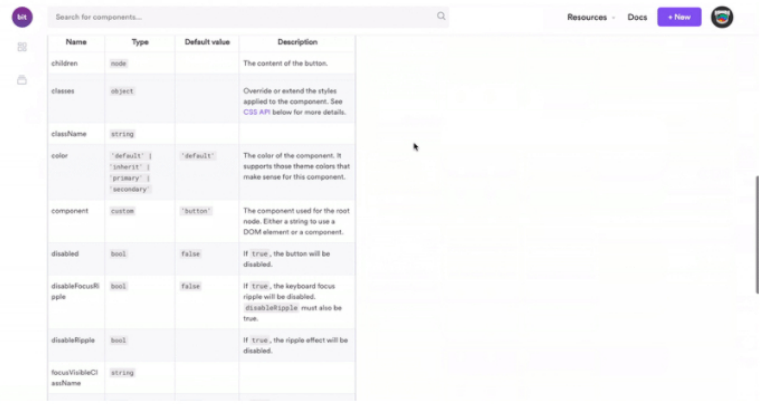
Looking at the above implementation, we called the interval to emit values at the interval of 1sec. We subscribe to it to receive the emitted value, our function callback will log the emitted value on the browser console.

Now, if this AppComponent is destroyed, maybe via navigating away from the component or using the `destroy(...)` method, we will still be seeing the console log on the browser. This is because the AppComponent has been destroyed but the subscription still lives on, it hasn't been canceled.

If a subscription is not closed the function callback attached to it will be continuously called, this poses a huge memory leak and performance issue.

If the function callback in our AppCompoennt subscription had been an expensive function, we will see that the function will still be called despite its parent being destroyed this will eat up resources and slow down the overall app performance.

Tip: Use **Bit** (Github) to "harvest" Angular components from any codebase and share them on bit.dev. Let your team reuse and collaborate on components to write scalable code, speed up development and maintain a consistent UI.



Example: searching for shared components on bit.dev

1. Use the `unsubscribe` method

> A Subscription essentially just has an `unsubscribe()` function to release resources or cancel Observable executions.

To prevent this memory leaks we have to unsubscribe from the subscriptions when we are done with. We do so by calling the `unsubscribe` method in the Observable.

In Angular, we have to unsubscribe from the Observable when the component is being destroyed. Luckily, Angular has a `ngOnDestroy` hook that is called before a component is destroyed, this enables devs to provide the cleanup crew here to avoid hanging subscriptions, open portals, and what nots that may come in the future to bite us in the back.

So, whenever we use Observables in a component in Angular, we should

set up the ngOnDestroy method, and call the unsubscribe method on all of them.

```
@Component({...})
export class AppComponent implements OnInit, OnDestroy {
    subscription: Subscription
    ngOnInit () {
        var observable = Rx.Observable.interval(1000);
        this.subscription = observable.subscribe(x =>
console.log(x));
    }

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

We added ngOnDestroy to our AppCompoennt and called unsubscribe method on the `this.subscription` Observable. When the AppComponent is destroyed (via route navigation, destroy(...), etc) there will be no hanging subscription, the interval will be canceled and there will be no console logs in the browser anymore.

If there are multiple subscriptions:

```
@Component({...})
export class AppComponent implements OnInit, OnDestroy {
    subscription1$: Subscription
    subscription2$: Subscription

    ngOnInit () {
        var observable1$ = Rx.Observable.interval(1000);
        var observable2$ = Rx.Observable.interval(400);
        this.subscription1$ = observable.subscribe(x =>
console.log("From interval 1000" x));
        this.subscription2$ = observable.subscribe(x =>
console.log("From interval 400" x));
    }

    ngOnDestroy() {
        this.subscription1$.unsubscribe()
        this.subscription2$.unsubscribe()
    }
}
```

There are two subscriptions in AppComponent. They are both unsubscribed in the ngOnDestroy hook preventing memory leaks.

We can gather them subscriptions in an array and unsubscribe from them in the `ngOnDestroy`:

```
@Component({...})
export class AppComponent implements OnInit, OnDestroy {
    subscription1$: Subscription
    subscription2$: Subscription
    subscriptions: Subscription[] = []

    ngOnInit () {
        var observable1$ = Rx.Observable.interval(1000);
        var observable2$ = Rx.Observable.interval(400);
        this.subscription1$ = observable.subscribe(x =>
console.log("From interval 1000" x));
        this.subscription2$ = observable.subscribe(x =>
console.log("From interval 400" x));
        this.subscriptions.push(this.subscription1$)
        this.subscriptions.push(this.subscription2$)
    }

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

Observables subscribe method returns an object of RxJS's Subscription type. This Subscription represents a disposable resource. These Subscriptions can be grouped using the `add` method, this will attach a child Subscription to the current Subscription. When a Subscription is unsubscribed, all its children will be unsubscribed as well. We can refactor our AppComponent to use this:

```
@Component({...})
export class AppComponent implements OnInit, OnDestroy {

    subscription: Subscription


    ngOnInit () {
        var observable1$ = Rx.Observable.interval(1000);
        var observable2$ = Rx.Observable.interval(400);
        var subscription1$ = observable.subscribe(x =>
console.log("From interval 1000" x));
        var subscription2$ = observable.subscribe(x =>
console.log("From interval 400" x));
        this.subscription.add(subscription1$)
        this.subscription.add(subscription2$)
    }


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

This will unsubscribe `this.subscripton1$`, and `this.subscripton2$` when the component is destroyed.

2. Use Async `|` Pipe

The `async` pipe subscribes to an `Observable` or `Promise` and returns the latest value it has emitted. When a new value is emitted, the `async` pipe marks the component to be checked for changes. When the component gets destroyed, the `async` pipe unsubscribes automatically to avoid potential memory leaks.

Using it in our AppComponent:

```
@Component({
    ...,
    template: `
        <div>
         Interval: {{observable$ | async}}
        </div>
    `
})
export class AppComponent implements OnInit {
    observable$
    ngOnInit () {
        this.observable$ = Rx.Observable.interval(1000);
    }
}
```

On instantiation, the AppComponent will create an Observable from the interval method. In the template, the Observable `observable$` is piped to the async Pipe. The `async` pipe will subscribe to the `observable$` and display its value in the DOM. `async` pipe will unsubscribe the `observable$` when the AppComponent is destroyed. async Pipe has ngOnDestroy on its class so it is called when the view is contained in is being destroyed.

Using the async pipe is a huge advantage if we are using Observables in

our components because it will subscribe to them and unsubscribe from them. We will not be bothered about forgetting to unsubscribe from them in ngOnDestroy when the component is being killed off.

3. Use RxJS `take*` operators

RxJS have useful operators that we can use in a declarative way to unsubscribe from subscriptions in our Angular project. One of them are the `take*` family operators:

- take(n)
- takeUntil(notifier)
- takeWhile(predicate)

take(n)

This operator makes a subscription happen once. This operator makes a source subscription happen the number of `n` times specified and completes.

`1` is popularly used with the take operator so subscriptions happen once and exit.

This operator will be effective when we want a source Observable to emit once and then unsubscribe from the stream:

```
@Component({
    ...
})
export class AppComponent implements OnInit {
    subscription$
    ngOnInit () {
        var observable$ = Rx.Observable.interval(1000);
        this.subscription$ = observable$.pipe(take(1)).
        subscribe(x => console.log(x))
     }
}
```

The subscription$ will unsubscribe when the interval emits the first value.

Beware, even if the AppComponent is destroyed the `subscription$` will not unsubscribe until the interval emits a value.

So it is best to make sure everything is canceled in the ngOnDestroy hook:

```
@Component({
    ...
})
export class AppComponent implements OnInit, OnDestroy {
    subscription$
    ngOnInit () {
        var observable$ = Rx.Observable.interval(1000);
        this.subscription$ = observable$.pipe(take(1)).
        subscribe(x => console.log(x))
    }

    ngOnDestroy() {
        this.subscription$.unsubscribe()
    }
}
```

or to make sure that the source Observable is fired during the component

or to make sure that the source Observable is fired during the component lifetime.

takeUntil(notifier)

This operator emits values emitted by the source Observable until a notifier Observable emits a value.

```
@Component({...})
export class AppComponent implements OnInit, OnDestroy {
    notifier = new Subject()

    ngOnInit () {
        var observable$ = Rx.Observable.interval(1000);
        observable$.pipe(takeUntil(this.notifier))
        .subscribe(x => console.log(x));
    }

    ngOnDestroy() {
        this.notifier.next()
        this.notifier.complete()
    }
}
```

We have an extra notifier Subject, this is what will emit to make the `this.subscription` unsubscribe. See, we pipe the observable to takeUntil before we subscribe. The takeUntil will emit the values emitted by the interval until the notifier Subject emits, it will then unsubscribe the observable$. The best place to make the notifier to emit so the observable$ is canceled is in the ngOnDestroy hook.

takeWhile(predicate)

This operator will emit the value emitted by the source Observable so long as the emitted value passes the test condition of the predicate.

```
@Component({...})
export class AppComponent implements OnInit {
    ngOnInit () {
        var observable$ = Rx.Observable.interval(1000);
        observable$.pipe(takeWhile(value => value < 10))
        .subscribe(x => console.log(x));
    }
}
```

We pipe the observable$ to go through takeWhile operator. The takeWhile operator will pass the values so long as the values as less than 10. If it encounters a value greater than or equal to 10 the operator will unsubscribe the observable$.

If the interval doesn't emit up to 9 values before the AppComponent is destroyed, the observable$ subscription will still be open until the interval emits 10 before it is destroyed. So for safety, we add the ngOnDestroy hook to unsubscribe observable$ when the component is destroyed.

```
@Component({...})
export class AppComponent implements OnInit, OnDestroy {
    subscription$
    ngOnInit () {
        var observable$ = Rx.Observable.interval(1000);
        this.subscription$ = observable$.pipe(takeWhile(value =>
value < 10))
        .subscribe(x => console.log(x));
    }

    ngOnDestroy() {
        this.subscription$.unsubscribe()
```

```
    }
}
```

4. Use RxJS `first` operator

This operator is like the concatenation of take(1) and takeWhile 😁

If called with no value, it emits the first value emitted by the source Observable and completes. If it is called with a predicate function, it emits the first value of the source Observable that pass the test condition of the predicate function and complete.

```
@Component({...})
export class AppComponent implements OnInit {
    observable$
    ngOnInit () {
        this.observable = Rx.Observable.interval(1000);
        this.observable$.pipe(first())
        .subscribe(x => console.log(x));
    }
}
```

The observable$ will complete if the interval emits its first value. That means, in our console, we will see only `1` log and nothing else. At point will stop emitting values.

```
@Component({...})
export class AppComponent implements OnInit {
    observable$
    ngOnInit () {
        this.observable$ = Rx.Observable.interval(1000);
        this.observable$.pipe(first(val => val === 10))
        .subscribe(x => console.log(x));
    }
}
```

Here, first will not emit a value until interval emits a value that is equal to 10, then it will complete the observable$. In the console will see only one log and no more.

In the first example, if the AppComponent is destroyed before first receives a value from observable$, the subscription will still be open until the interval emits its first value.

Also, in the second example, if the AppComponent is destroyed before interval produces a value that passes the first operator condition, the subscription will still be open up until interval emits 10.

So to ensure safety, we have to explicitly cancel the subscriptions in the ngOnDestroy hook when the component is destroyed.

5. Use Decorator to automate Unsubscription

We are humans, we can forget, it is our nature. Most methods we have seen here relies on the ngOnDestroy hook to make sure we clean up the subscriptions in the component when destroyed. We can forget to clean them up in the ngOnDestroy, maybe due to deadline leaning dangerously near and we have a grumpy client or a psycho client that knows where you live oh boy

We can leverage the usefulness of Decorator in our Angular projects to help us add the ngOnDestroy method in our components and unsubscribe from all the subscriptions in our component automatically.

Here is an implementation that will be of huge help:

```
function AutoUnsub() {
    return function(constructor) {
        const orig = constructor.prototype.ngOnDestroy
        constructor.prototype.ngOnDestroy = function() {
            for(const prop in this) {
                const property = this[prop]
                if(typeof property.subscribe === "function") {
                    property.unsubscribe()
                }
            }
            orig.apply()
        }
    }
}
```

Top highlight

This AutoUnsub is a class decorator that can be applied to classes in our Angular project. See, it saves the original ngOnDestroy hook, then creates a new one and hook it into the class it is applied on. So, when the class is being destroyed the new hook is called. Inside it, the functions scan through the properties of the class, if it finds an Observable property it will unsubscribe from it. Then it calls the original ngOnDestroy hook in the class if present.

```
@Component({
    ...
})
@AutoUnsub
export class AppComponent implements OnInit {
    observable$
    ngOnInit () {
        this.observable$ = Rx.Observable.interval(1000);
        this.observable$.subscribe(x => console.log(x))
    }
}
```

We apply it on our AppComponent class no longer bothered about unsubscribing the observable$ in the ngOnDestroy, the Decorator will do it for us.

There are downsides to this(what isn't in this world), it will be a problem if we have non-subscribing Observable in our component.

6. Use `tslint`

Some might need a reminder by `tslint`, to remind us in the console that our components or directives should have a ngOnDestroy method when it detects none.

We can add a custom rule to tslint to warn us in the console during linting and building if it finds no ngOnDestroy hook in our components:

```
// ngOnDestroyRule.ts

import * as Lint from "tslint"
import * as ts from "typescript"
import * as tsutils from "tsutils"

export class Rule extends Lint.Rules.AbstractRule {
```

```
    public static metadata: Lint.IRuleMetadata = {
        ruleName: "ng-on-destroy",
        description: "Enforces ngOnDestory hook on
component/directive/pipe classes",
        optionsDescription: "Not configurable.",
        options: null,
        type: "style",
        typescriptOnly: false
    }

    public static FAILURE_STRING = "Class name must have the
ngOnDestroy hook";

    public apply(sourceFile: ts.SourceFile): Lint.RuleFailure[] {
        return this.applyWithWalker(new NgOnDestroyWalker(sourceFile,
Rule.metadata.ruleName, void this.getOptions()))
    }
}

class NgOnDestroyWalker extends Lint.AbstractWalker {

    visitClassDeclaration(node: ts.ClassDeclaration) {
        this.validateMethods(node)
    }

    validateMethods(node: ts.ClassDeclaration) {
        const methodNames =
node.members.filter(ts.isMethodDeclaration).map(m =>
m.name!.getText());
        const ngOnDestroyArr = methodNames.filter( methodName =>
methodName === "ngOnDestroy")
        if( ngOnDestroyArr.length === 0)
            this.addFailureAtNode(node.name, Rule.FAILURE_STRING);
    }
}
```

If we have a component like this without ngOnDestroy:

```
@Component({
    ...
})
export class AppComponent implements OnInit {
    observable$
    ngOnInit () {
        this.observable$ = Rx.Observable.interval(1000);
        this.observable$.subscribe(x => console.log(x))
    }
}
```

Linting AppComponent will warn us about the missing ngOnDestroy hook:

```
$ ng lint
Error at app.component.ts 12: Class name must have the ngOnDestroy
hook
```

💯 Helpful Links 💯

Conclusion

We have seen ways that we can unsubscribe from our Angular projects. Like we have learned, hanging or open subscriptions may introduce memory leaks, bugs, unwanted behavior or performance cost to our Angular apps. Find the best possible ways for you here and plug those

leaks today.

Social Media links

- Twitter
- Facebook
- LinkedIn

Learn More

JavaScript Angular Webdev Web Development Frontend

WRITTEN BY

Chidume Nnamdi 🔥💻🎵🎮

Follow

I am available for gigs, full-time and part-time jobs, contract roles | 📫:kurtwanger40@gmail.com | Author of “Understanding JavaScript” https://gum.co/LikDD 📕

Bits and Pieces

Follow

The best of web and frontend development articles, tutorials, and news. Love JS? Follow to get the best stories.

More From Medium