Angular/RxJs When should I unsubscribe from `Subscription`

Asked 3 years, 2 months ago Active 11 days ago Viewed 142k times

620

389

When should I store the `Subscription` instances and invoke `unsubscribe()` during the NgOnDestroy life cycle and when can I simply ignore them?

Saving all subscriptions introduces a lot of mess into component code.

HTTP Client Guide ignore subscriptions like this:

```
getHeroes() {
  this.heroService.getHeroes()
                   .subscribe(
                     heroes => this.heroes = heroes,
                     error =>  this.errorMessage = <any>error);
}
```

In the same time Route & Navigation Guide says that:

> Eventually, we'll navigate somewhere else. The router will remove this component from the DOM and destroy it. We need to clean up after ourselves before that happens. Specifically, we must unsubscribe before Angular destroys the component. Failure to do so could create a memory leak.
>
> We unsubscribe from our `Observable` in the `ngOnDestroy` method.

```
private sub: any;

ngOnInit() {
  this.sub = this.route.params.subscribe(params => {
     let id = +params['id']; // (+) converts string 'id' to a number
     this.service.getHero(id).then(hero => this.hero = hero);
   });
}
```

```
    this.sub.unsubscribe();
  }
```

angular rxjs observable subscription angular-component-life-cycle

edited Dec 16 '17 at 10:27
Jota.Toledo
15.2k 7 35 55

asked Jun 24 '16 at 7:52
Sergey Tihon
4,576 3 16 25

16 I guess `Subscription` s to `http-requests` can be ignored, as they only call `onNext` once and then they call `onComplete` . The `Router` instead calls `onNext` repeatedly and might never call `onComplete` (not sure about that...). Same goes for `Observable` s from `Event` s. So I guess those should be `unsubscribed` . – Springrbua Jun 24 '16 at 8:41

1 @gt6707a The stream completes (or does not complete) independent of any observation of that completion. The callbacks (the observer) provided to the subscription function do not determine if resources are allocated. It is the call to `subscribe` itself that potentially allocates resources upstream. – seangwright Dec 16 '16 at 3:31

18 Answers

901

--- Edit 4 - Additional Resources (2018/09/01)

In a recent episode of Adventures in Angular Ben Lesh and Ward Bell discuss the issues around how/when to unsubscribe in a component. The discussion starts at about 1:05:30.

Ward mentions `right now there's an awful takeUntil dance that takes a lot of machinery` and Shai Reznik mentions `Angular handles some of the subscriptions like http and routing` .

In response Ben mentions that there are discussions right now to allow Observables to hook into the Angular component lifecycle events and Ward suggests an Observable of lifecycle events that a component could subscribe to as a way of knowing when to complete Observables maintained as component internal state.

That said, we mostly need solutions now so here are some other resources.

2. Lightweight npm package that exposes an Observable operator that takes a component instance (`this`) as a parameter and automatically unsubscribes during `ngOnDestroy` . https://github.com/NetanelBasal/ngx-take-until-destroy
3. Another variation of the above with slightly better ergonomics if you are not doing AOT builds (but we should all be doing AOT now). https://github.com/smnbbrv/ngx-rx-collector
4. Custom directive `*ngSubscribe` that works like async pipe but creates an embedded view in your template so you can refer to the 'unwrapped' value throughout your template. https://netbasal.com/diy-subscription-handling-directive-in-angular-c8f6e762697f

I mention in a comment to Nicholas' blog that over-use of `takeUntil()` could be a sign that your component is trying to do too much and that separating your existing components into **Feature** and **Presentational** components should be considered. You can then `| async` the Observable from the Feature component into an `Input` of the Presentational component, which means no subscriptions are necessary anywhere. Read more about this approach here

--- Edit 3 - The 'Official' Solution (2017/04/09)

I spoke with Ward Bell about this question at NGConf (I even showed him this answer which he said was correct) but he told me the docs team for Angular had a solution to this question that is unpublished (though they are working on getting it approved). He also told me I could update my SO answer with the forthcoming official recommendation.

The solution we should all use going forward is to add a `private ngUnsubscribe = new Subject();` field to all components that have `.subscribe()` calls to `Observable` s within their class code.

We then call `this.ngUnsubscribe.next(); this.ngUnsubscribe.complete();` in our `ngOnDestroy()` methods.

The secret sauce (as noted already by @metamaker) is to call `takeUntil(this.ngUnsubscribe)` before each of our `.subscribe()` calls which will guarantee all subscriptions will be cleaned up when the component is destroyed.

Example:

```
import { Component, OnDestroy, OnInit } from '@angular/core';
// RxJs 6.x+ import paths
import { filter, startWith, takeUntil } from 'rxjs/operators';
import { Subject } from 'rxjs';
import { BookService } from '../books.service';

@Component({
    selector: 'app-books',
    templateUrl: './books.component.html'
})
```

```
    constructor(private booksService: BookService) { }

    ngOnInit() {
        this.booksService.getBooks()
            .pipe(
               startWith([]),
               filter(books => books.length > 0),
               takeUntil(this.ngUnsubscribe)
            )
            .subscribe(books => console.log(books));

        this.booksService.getArchivedBooks()
            .pipe(takeUntil(this.ngUnsubscribe))
            .subscribe(archivedBooks => console.log(archivedBooks));
    }

    ngOnDestroy() {
        this.ngUnsubscribe.next();
        this.ngUnsubscribe.complete();
    }
}
```

Note: It's important to add the `takeUntil` operator as the last one to prevent leaks with intermediate observables in the operator chain.

--- Edit 2 (2016/12/28)

Source 5

The Angular tutorial, the Routing chapter now states the following: "The Router manages the observables it provides and localizes the subscriptions. The subscriptions are cleaned up when the component is destroyed, protecting against memory leaks, so we don't need to unsubscribe from the route params Observable." - Mark Rajcok

Here's a discussion on the Github issues for the Angular docs regarding Router Observables where Ward Bell mentions that clarification for all of this is in the works.

--- Edit 1

Source 4

In this video from NgEurope Rob Wormald also says you do not need to unsubscribe from Router Observables. He also mentions the `http` service and `ActivatedRoute.params` in this video from November 2016.

TLDR:

For this question there are (2) kinds of `Observables` - **finite** value and **infinite** value.

`http` `Observables` produce **finite** (1) values and something like a DOM `event listener` `Observables` produce **infinite** values.

If you manually call `subscribe` (not using async pipe), then `unsubscribe` from **infinite** `Observables` .

Don't worry about **finite** ones, `RxJs` will take care of them.

Source 1

I tracked down an answer from Rob Wormald in Angular's Gitter here.

He states (i reorganized for clarity and emphasis is mine)

> if its **a single-value-sequence** (like an http request) the **manual cleanup is unnecessary** (assuming you subscribe in the controller manually)
>
> i should say "if its a **sequence that completes**" (of which single value sequences, a la http, are one)
>
> **if its an infinite sequence**, **you should unsubscribe** which the async pipe does for you

Also he mentions in this youtube video on Observables that `they clean up after themselves` ... in the context of Observables that `complete` (like Promises, which always complete because they are always producing 1 value and ending - we never worried about unsubscribing from Promises to make sure they clean up `xhr` event listeners, right?).

Source 2

Also in the Rangle guide to Angular 2 it reads

> In most cases we will not need to explicitly call the unsubscribe method unless we want to cancel early or our Observable has a longer lifespan than our subscription. The default behavior of Observable operators is to dispose of the subscription as soon as .complete() or .error() messages are published. Keep in mind that RxJS was designed to be used in a "fire and forget" fashion most of the time.

When does the phrase `our Observable has a longer lifespan than our subscription` apply?

I read this as meaning if we subscribe to an `http` request or an observable that emits 10 values and our component is destroyed before that `http` request returns or the 10 values have been emitted, we are still ok!

When the request does return or the 10th value is finally emitted the `Observable` will complete and all resources will be cleaned up.

Source 3

If we look at this example from the same Rangle guide we can see that the `Subscription` to `route.params` does require an `unsubscribe()` because we don't know when those `params` will stop changing (emitting new values).

The component could be destroyed by navigating away in which case the route params will likely still be changing (they could technically change until the app ends) and the resources allocated in subscription would still be allocated because there hasn't been a `completion` .

edited Sep 1 '18 at 17:12

answered Dec 16 '16 at 4:11
seangwright
11.3k 4 28 41

13 Calling `complete()` by itself doesn't appear to clean up the subscriptions. However calling `next()` and then `complete()` does, I believe `takeUntil()` only stops when a value is produced, not when the sequence is ended. – Firefly Apr 11 '17 at 8:53

2 @seangwright A quick test with a member of type `Subject` inside a component and toggling it with `ngIf` to trigger `ngOnInit` and `ngOnDestroy` shows, that the subject and its subscriptions will never complete or get disposed (hooked up a `finally` -operator to the subscription). I must call `Subject.complete()` in `ngOnDestroy` , so the subscriptions can clean up after themselves. – Lars Apr 11 '17 at 9:17

2 Your --- *Edit 3* is very insightful, thanks! I just have a followup question: if using the `takeUnitl` approach, we never have to manually unsubscribe from any observables? Is that the case? Furthermore, why do we need to call `next()` in the `ngOnDestroy` , why not just call `complete()` ? – uglycode Apr 22 '17 at 10:21

5 @seangwright That's disappointing; the additional boilerplate is annoying. – spongessuck Apr 27 '17 at 20:30

3 **Edit 3** discussed in context of events at medium.com/@benlesh/rxjs-dont-unsubscribe-6753ed4fda87 – HankCa May 29 '17 at 2:07

76

You don't need to have bunch of subscriptions and unsubscribe manually. Use Subject and takeUntil combo to handle subscriptions like a boss:

```
import { Subject } from "rxjs"
```

```
  moduleId: __moduleName,
  selector: "my-view",
  templateUrl: "../views/view-route.view.html"
})
export class ViewRouteComponent implements OnInit, OnDestroy {
  componentDestroyed$: Subject<boolean> = new Subject()

  constructor(private titleService: TitleService) {}

  ngOnInit() {
    this.titleService.emitter1$
      .pipe(takeUntil(this.componentDestroyed$))
      .subscribe((data: any) => { /* ... do something 1 */ })

    this.titleService.emitter2$
      .pipe(takeUntil(this.componentDestroyed$))
      .subscribe((data: any) => { /* ... do something 2 */ })

    //...

    this.titleService.emitterN$
      .pipe(takeUntil(this.componentDestroyed$))
      .subscribe((data: any) => { /* ... do something N */ })
  }

  ngOnDestroy() {
    this.componentDestroyed$.next(true)
    this.componentDestroyed$.complete()
  }
}
```

Alternative approach, which was proposed by @acumartini in comments, uses takeWhile instead of takeUntil. You may prefer it, but mind that this way your Observable execution will not be cancelled on ngDestroy of your component (e.g. when you make time consuming calculations or wait for data from server). Method, which is based on takeUntil, doesn't have this drawback and leads to immediate cancellation of request. Thanks to @AlexChe for detailed explanation in comments.

So here is the code:

```
@Component({
  moduleId: __moduleName,
  selector: "my-view",
  templateUrl: "../views/view-route.view.html"
```

```
 constructor(private titleService: TitleService) {}

 ngOnInit() {
  this.titleService.emitter1$
   .pipe(takeWhile(() => this.alive))
   .subscribe((data: any) => { /* ... do something 1 */ })

  this.titleService.emitter2$
   .pipe(takeWhile(() => this.alive))
   .subscribe((data: any) => { /* ... do something 2 */ })

  // ...

  this.titleService.emitterN$
   .pipe(takeWhile(() => this.alive))
   .subscribe((data: any) => { /* ... do something N */ })
 }

 // Probably, this.alive = false MAY not be required here, because
 // if this.alive === undefined, takeWhile will stop. I
 // will check it as soon, as I have time.
 ngOnDestroy() {
  this.alive = false
 }
}
```

edited Aug 14 at 12:15
fridoo
2,792 3 13 28

answered Mar 9 '17 at 12:35
metamaker
1,340 1 13 17

1 If he just use a bool to keep the state, how to make "takeUntil" works as expected? – Val Apr 24 '17 at 3:38

4 I think there is a significant difference between using `takeUntil` and `takeWhile` . The former unsubscribes from the source observable immediately when fired, while the latter unsubscribes only as soon as next value is produced by the source observable. If producing a value by the source observable is a resource consuming operation, choosing between the two may go beyond style preference. See the plunk – Alex Che Aug 22 '17 at 16:40

1 @AlexChe thanks for providing interesting plunk! This is very valid point for general usage of `takeUntil` vs `takeWhile` , however, not for our specific case. When we need to unsubscribe listeners **on component destruction**, we are just checking boolean value like `() => alive` in `takeWhile` , so any time/memory consuming operations are not used and difference is pretty much about styling (ofc, for this specific case). – metamaker Aug 31 '17 at 10:28

`ngOnDestroy` is called during our component destruction. Thus the mining `Observable` function is able to cancel it's operation immediately during this process. – Alex Che Aug 31 '17 at 14:19

1 OTOH, if we use `takeWhile` , in the `ngOnDestory` we just set the boolean variable. But the mining `Observable` function might still work for up to one day, and only then during it's `next` call will it realize that there are no subscriptions active and it needs to cancel. – Alex Che Aug 31 '17 at 14:23

60

The Subscription class has an interesting feature:

> Represents a disposable resource, such as the execution of an Observable. A Subscription has one important method, unsubscribe, that takes no argument and just disposes the resource held by the subscription.
> **Additionally, subscriptions may be grouped together through the add() method, which will attach a child Subscription to the current Subscription. When a Subscription is unsubscribed, all its children (and its grandchildren) will be unsubscribed as well.**

You can create an aggregate Subscription object that groups all your subscriptions. You do this by creating an empty Subscription and adding subscriptions to it using its `add()` method. When your component is destroyed, you only need to unsubscribe the aggregate subscription.

```
@Component({ ... })
export class SmartComponent implements OnInit, OnDestroy {
  private subscriptions = new Subscription();

  constructor(private heroService: HeroService) {
  }

  ngOnInit() {
    this.subscriptions.add(this.heroService.getHeroes().subscribe(heroes => this.heroes
= heroes));
    this.subscriptions.add(/* another subscription */);
    this.subscriptions.add(/* and another subscription */);
    this.subscriptions.add(/* and so on */);
  }

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

6,865 3 42 63

I'm using this approach. Wondering if this is better than using the approach with takeUntil(), like in the accepted answer.. drawbacks ? – Manuel Di Iorio Sep 19 '17 at 20:28

No drawbacks that I'm aware of. I don't think this is better, just different. – Steven Liekens Sep 19 '17 at 20:56

1 See medium.com/@benlesh/rxjs-dont-unsubscribe-6753ed4fda87 for further discussion on the official `takeUntil` approach versus this approach of collecting subscriptions and calling `unsubscribe` . (This approach seems a lot cleaner to me.) – Josh Kelley Mar 29 '18 at 20:31

1 One small benefit of this answer: you don't have to check if `this.subscriptions` is null – user2023861 Aug 24 '18 at 19:58

1 Just avoid the chaining of add methods like `sub = subsciption.add(..).add(..)` because in many cases it produces unexpected results github.com/ReactiveX/rxjs/issues/2769#issuecomment-345636477 – Evgeniy Generalov Sep 29 '18 at 13:51

26

Some of the best practices regarding observables unsubscriptions inside Angular components:

A quote from `Routing & Navigation`

> When subscribing to an observable in a component, you almost always arrange to unsubscribe when the component is destroyed.
>
> There are a few exceptional observables where this is not necessary. The ActivatedRoute observables are among the exceptions.
>
> The ActivatedRoute and its observables are insulated from the Router itself. The Router destroys a routed component when it is no longer needed and the injected ActivatedRoute dies with it.
>
> Feel free to unsubscribe anyway. It is harmless and never a bad practice.

And in responding to the following links:

- (1) Should I unsubscribe from Angular 2 Http Observables?
- (2) Is it necessary to unsubscribe from observables created by Http methods?
- (3) RxJS: Don’t Unsubscribe
- (4) The easiest way to unsubscribe from Observables in Angular
- (5) Documentation for RxJS Unsubscribing

- (8) A comment about the `http` observable

I collected some of the best practices regarding observables unsubscriptions inside Angular components to share with you:

- `http` observable unsubscription is conditional and we should consider the effects of the 'subscribe callback' being run after the component is destroyed on a case by case basis. We know that angular unsubscribes and cleans the `http` observable itself (1), (2). While this is true from the perspective of resources it only tells half the story. Let's say we're talking about directly calling `http` from within a component, and the `http` response took longer than needed so the user closed the component. The `subscribe()` handler will still be called even if the component is closed and destroyed. This can have unwanted side effects and in the worse scenarios leave the application state broken. It can also cause exceptions if the code in the callback tries to call something that has just been disposed of. However at the same time occasionally they are desired. Like, let's say you're creating an email client and you trigger a sound when the email is done sending - well you'd still want that to occur even if the component is closed (8).
- No need to unsubscribe from observables that complete or error. However, there is no harm in doing so(7).
- Use `AsyncPipe` as much as possible because it automatically unsubscribes from the observable on component destruction.
- Unsubscribe from the `ActivatedRoute` observables like `route.params` if they are subscribed inside a nested (Added inside tpl with the component selector) or dynamic component as they may be subscribed many times as long as the parent/host component exists. No need to unsubscribe from them in other scenarios as mentioned in the quote above from `Routing & Navigation` docs.
- Unsubscribe from global observables shared between components that are exposed through an Angular service for example as they may be subscribed multiple times as long as the component is initialized.
- No need to unsubscribe from internal observables of an application scoped service since this service never get's destroyed, unless your entire application get's destroyed, there is no real reason to unsubscribe from it and there is no chance of memory leaks. (6).

  Note: Regarding scoped services, i.e component providers, they are destroyed when the component is destroyed. In this case, if we subscribe to any observable inside this provider, we should consider unsubscribing from it using the `OnDestroy` lifecycle hook which will be called when the service is destroyed, according to the docs.
- Use an abstract technique to avoid any code mess that may be resulted from unsubscriptions. You can manage your subscriptions with `takeUntil` (3) or you can use this `npm` package mentioned at (4) The easiest way to unsubscribe from Observables in Angular.
- Always unsubscribe from `FormGroup` observables like `form.valueChanges` and `form.statusChanges`
- Always unsubscribe from observables of `Renderer2` service like `renderer2.listen`
- Unsubscribe from every observable else as a memory-leak guard step until Angular Docs explicitly tells us which observables are unnecessary to be unsubscribed (Check issue: (5) Documentation for RxJS Unsubscribing (Open)).
- Bonus: Always use the Angular ways to bind events like `HostListener` as angular cares well about removing the event listeners if needed and prevents any potential memory leak due to event bindings

A nice final tip: If you don't know if an observable is being automatically unsubscribed/completed or not, add a `complete` callback to `subscribe(...)` and check if it gets called when the component is destroyed.

edited Oct 2 '18 at 12:00

answered Aug 7 '18 at 18:03

Mouneer
6,058 2 27 41

Answer for No. 6 is not quite right. Services are destroyed and their `ngOnDestroy` is called when the service is provided at a level other than the root level e.g. provided explicitly in a component that later gets removed. In these cases you should unsubscribe from the services inner observables – Drenai Sep 9 '18 at 7:50

@Drenai, thanks for your comment and politely I don't agree. If a component is destroyed, the component, service and the observable will be all GCed and the unsubscription will be useless in this case unless you keep a reference for the observable anywhere away from the component (Which is not logical to leak the component states globally despite scoping the service to the component) – Mouneer Sep 11 '18 at 16:02

If the service being destroyed has a subscription to an observable belonging to another service higher up in the DI hierarchy, then GC won't occur. Avoid this scenario by unsubscribing in `ngOnDestroy` , which is always called when services are destroyed github.com/angular/angular/commit/… – Drenai Sep 11 '18 at 22:54

1 @Drenai, Check the updated answer. – Mouneer Oct 2 '18 at 12:00

2 @Tim First of all, `Feel free to unsubscribe anyway. It is harmless and never a bad practice.` and regarding your question, it depends. If the child component is initiated multiple times (For example, added inside `ngIf` or being loaded dynamically), you must unsubscribe to avoid adding multiple subscriptions to the same observer. Otherwise no need. But I prefer unsubscribing inside the child component as this makes it more reusable and isolated from how it could be used. – Mouneer Oct 8 '18 at 17:03

15

It depends. If by calling `someObservable.subscribe()` , you start holding up some resource that must be manually freed-up when the lifecycle of your component is over, then you should call `theSubscription.unsubscribe()` to prevent memory leak.

Let's take a closer look at your examples:

`getHero()` returns the result of `http.get()` . If you look into the angular 2 source code, `http.get()` creates two event listeners:

```
_xhr.addEventListener('load', onLoad);
_xhr.addEventListener('error', onError);
```

and by calling `unsubscribe()` , you can cancel the request as well as the listeners:

```
_xhr.abort();
```

Note that `_xhr` is platform specific but I think it's safe to assume that it is an `XMLHttpRequest()` in your case.

Normally, this is enough evidence to warrant a manual `unsubscribe()` call. But according this WHATWG spec, the `XMLHttpRequest()` is subject to garbage collection once it is "done", even if there are event listeners attached to it. So I guess that's why angular 2 official guide omits `unsubscribe()` and lets GC clean up the listeners.

As for your second example, it depends on the implementation of `params` . As of today, the angular official guide no longer shows unsubscribing from `params` . I looked into src again and found that `params` is a just a BehaviorSubject. Since no event listeners or timers were used, and no global variables were created, it should be safe to omit `unsubscribe()` .

The bottom line to your question is that always call `unsubscribe()` as a guard against memory leak, unless you are certain that the execution of the observable doesn't create global variables, add event listeners, set timers, or do anything else that results in memory leaks.

When in doubt, look into the implementation of that observable. If the observable has written some clean up logic into its `unsubscribe()` , which is usually the function that is returned by the constructor, then you have good reason to seriously consider calling `unsubscribe()` .

edited Dec 1 '16 at 7:14

answered Dec 1 '16 at 7:09
Chuanqi Sun
612 5 20

6

Angular 2 official documentation provides an explanation for when to unsubscribe and when it can be safely ignored. Have a look at this link:

https://angular.io/docs/ts/latest/cookbook/component-communication.html#!#bidirectional-service

Look for the paragraph with the heading **Parent and children communicate via a service** and then the blue box:

> Notice that we capture the subscription and unsubscribe when the AstronautComponent is destroyed. This is a memory-leak guard step. There is no actual risk in this app because the lifetime of a AstronautComponent is the same as the lifetime of the app itself. That would not always be true in a more complex application.
>
> We do not add this guard to the MissionControlComponent because, as the parent, it controls the lifetime of the MissionService.

answered Jun 29 '16 at 11:08

Cerny

125 3

3 as a component you never know whether you're a child or not. therefore you should always unsubscribe from subscriptions as best practice. – SeriousM Oct 29 '16 at 17:57

1 The point about MissionControlComponent is not really about whether it's a parent or not, it's that the component itself provides the service. When MissionControl gets destroyed, so does the service and any references to the instance of the service, thus there is no possibility of a leak. – ender Nov 10 '16 at 20:44

5

Based on : [Using Class inheritance to hook to Angular 2 component lifecycle]

Another generic approach:

```
export abstract class UnsubscribeOnDestroy implements OnDestroy {
  protected d$: Subject<any>;

  constructor() {
    this.d$ = new Subject<void>();

    const f = this.ngOnDestroy;
    this.ngOnDestroy = () => {
      f();
      this.d$.next();
      this.d$.complete();
    };
  }

  public ngOnDestroy() {
    // no-op
  }

}
```

Run code snippet   Expand snippet

```
@Component({
    selector: 'my-comp',
    template: ``
})
export class RsvpFormSaveComponent extends UnsubscribeOnDestroy implements OnInit {

    constructor() {
        super();
    }

    ngOnInit(): void {
      Observable.of('bla')
      .takeUntil(this.d$)
      .subscribe(val => console.log(val));
    }
}
```

Run code snippet   Expand snippet

edited Feb 23 '18 at 14:54

answered Apr 12 '17 at 11:04
JoG
346 4 14

1 This does NOT work correctly. Please be careful when using this solution. You are missing a `this.componentDestroyed$.next()` call like the accepted solution by sean above... – philn Feb 23 '18 at 12:00

3

Since seangwright's solution (Edit 3) appears to be very useful, I also found it a pain to pack this feature into base component, and hint other project teammates to remember to call super() on ngOnDestroy to activate this feature.

This answer provide a way to set free from super call, and make "componentDestroyed$" a core of base component.

```
class BaseClass {
    protected componentDestroyed$: Subject<void> = new Subject<void>();
    constructor() {
        this.ngOnDestroy = () => {
            this.componentDestroyed$.next();
            this.componentDestroyed$.complete();
            _$();
        }
    }

    /// placeholder of ngOnDestroy. no need to do super() call of extended class.
    ngOnDestroy() {}
}
```

And then you can use this feature freely for example:

```
@Component({
    selector: 'my-thing',
    templateUrl: './my-thing.component.html'
})
export class MyThingComponent extends BaseClass implements OnInit, OnDestroy {
    constructor(
        private myThingService: MyThingService,
    ) { super(); }

    ngOnInit() {
        this.myThingService.getThings()
            .takeUntil(this.componentDestroyed$)
            .subscribe(things => console.log(things));
    }

    /// optional. not a requirement to implement OnDestroy
    ngOnDestroy() {
        console.log('everything works as intended with or without super call');
    }

}
```

edited Apr 24 '17 at 5:34

answered Apr 24 '17 at 4:30

Val

11.4k 3 45 67

The official Edit #3 answer (and variations) works well, but the thing that gets me is the 'muddying' of the business logic around the

Here's another approach using wrappers.

> **Warining:** experimental code

File *subscribeAndGuard.ts* is used to create a new Observable extension to wrap `.subscribe()` and within it to wrap `ngOnDestroy()`. Usage is the same as `.subscribe()`, except for an additional first parameter referencing the component.

```
import { Observable } from 'rxjs/Observable';
import { Subscription } from 'rxjs/Subscription';

const subscribeAndGuard = function(component, fnData, fnError = null, fnComplete = null)
{

  // Define the subscription
  const sub: Subscription = this.subscribe(fnData, fnError, fnComplete);

  // Wrap component's onDestroy
  if (!component.ngOnDestroy) {
    throw new Error('To use subscribeAndGuard, the component must implement
ngOnDestroy');
  }
  const saved_OnDestroy = component.ngOnDestroy;
  component.ngOnDestroy = () => {
    console.log('subscribeAndGuard.onDestroy');
    sub.unsubscribe();
    // Note: need to put original back in place
    // otherwise 'this' is undefined in component.ngOnDestroy
    component.ngOnDestroy = saved_OnDestroy;
    component.ngOnDestroy();

  };

  return sub;
};

// Create an Observable extension
Observable.prototype.subscribeAndGuard = subscribeAndGuard;

// Ref: https://www.typescriptlang.org/docs/handbook/declaration-merging.html
declare module 'rxjs/Observable' {
  interface Observable<T> {
    subscribeAndGuard: typeof subscribeAndGuard;
  }
```

Here is a component with two subscriptions, one with the wrapper and one without. The only caveat is it **must implement OnDestroy** (with empty body if desired), otherwise Angular does not know to call the wrapped version.

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import 'rxjs/Rx';
import './subscribeAndGuard';

@Component({
  selector: 'app-subscribing',
  template: '<h3>Subscribing component is active</h3>',
})
export class SubscribingComponent implements OnInit, OnDestroy {

  ngOnInit() {

    // This subscription will be terminated after onDestroy
    Observable.interval(1000)
      .subscribeAndGuard(this,
        (data) => { console.log('Guarded:', data); },
        (error) => { },
        (/*completed*/) => { }
      );

    // This subscription will continue after onDestroy
    Observable.interval(1000)
      .subscribe(
        (data) => { console.log('Unguarded:', data); },
        (error) => { },
        (/*completed*/) => { }
      );
  }

  ngOnDestroy() {
    console.log('SubscribingComponent.OnDestroy');
  }
}
```

A demo plunker is here

An additional note: Re Edit 3 - The 'Official' Solution, this can be simplified by using takeWhile() instead of takeUntil() before subscriptions, and a simple boolean rather than another Observable in ngOnDestroy.

```
  iAmAlive = true;
  ngOnInit() {

    Observable.interval(1000)
      .takeWhile(() => { return this.iAmAlive; })
      .subscribe((data) => { console.log(data); });
  }

  ngOnDestroy() {
    this.iAmAlive = false;
  }
}
```

edited May 17 '17 at 22:03

answered May 16 '17 at 23:06
Richard Matsen
10.1k 2 12 44

3

Following the answer by @seangwright, I've written an abstract class that handles "infinite" observables' subscriptions in components:

```
import { OnDestroy } from '@angular/core';
import { Subscription } from 'rxjs/Subscription';
import { Subject } from 'rxjs/Subject';
import { Observable } from 'rxjs/Observable';
import { PartialObserver } from 'rxjs/Observer';

export abstract class InfiniteSubscriberComponent implements OnDestroy {
  private onDestroySource: Subject<any> = new Subject();

  constructor() {}

  subscribe(observable: Observable<any>): Subscription;

  subscribe(
    observable: Observable<any>,
    observer: PartialObserver<any>
  ): Subscription;

  subscribe(
    observable: Observable<any>,
    next?: (value: any) => void,
  subscribe(observable: Observable<any>, ...subscribeArgs): Subscription {
    return observable
      .takeUntil(this.onDestroySource)
      .subscribe(...subscribeArgs);
  }

  ngOnDestroy() {
    this.onDestroySource.next();
    this.onDestroySource.complete();
  }
}
```

To use it, just extend it in your angular component and call the `subscribe()` method as follows:

```
this.subscribe(someObservable, data => doSomething());
```

It also accepts the error and complete callbacks as usual, an observer object, or not callbacks at all. Remember to call `super.ngOnDestroy()` if you are also implementing that method in the child component.

Find here an additional reference by Ben Lesh: [RxJS: Don’t Unsubscribe]().

answered Apr 19 '18 at 23:34

Mau Muñoz
61 5

2

I tried seangwright's solution (Edit 3)

That is not working for Observable that created by timer or interval.

However, i got it working by using another approach:

```
import { Component, OnDestroy, OnInit } from '@angular/core';
import 'rxjs/add/operator/takeUntil';
import { Subject } from 'rxjs/Subject';
import { Subscription } from 'rxjs/Subscription';
import 'rxjs/Rx';
```

```
   selector: 'my-thing',
   templateUrl: './my-thing.component.html'
})
export class MyThingComponent implements OnDestroy, OnInit {
   private subscriptions: Array<Subscription> = [];

  constructor(
     private myThingService: MyThingService,
   ) { }

  ngOnInit() {
    const newSubs = this.myThingService.getThings()
        .subscribe(things => console.log(things));
    this.subscriptions.push(newSubs);
  }

  ngOnDestroy() {
    for (const subs of this.subscriptions) {
      subs.unsubscribe();
   }
 }
}
```

answered Apr 11 '17 at 20:00

Jeff Tham
66 4

2

I like the last two answers, but I experienced an issue if the the subclass referenced `"this"` in `ngOnDestroy` .

I modified it to be this, and it looks like it resolved that issue.

```
export abstract class BaseComponent implements OnDestroy {
    protected componentDestroyed$: Subject<boolean>;
    constructor() {
        this.componentDestroyed$ = new Subject<boolean>();
        let f = this.ngOnDestroy;
        this.ngOnDestroy = function()  {
            // without this I was getting an error if the subclass had
            // this.blah() in ngOnDestroy
            f.bind(this)();
            this.componentDestroyed$.next(true);
    }
    /// placeholder of ngOnDestroy. no need to do super() call of extended class.
    ngOnDestroy() {}
}
```

edited Sep 28 '17 at 19:51
Aniruddha Das
7,074 9 56 71

answered Apr 26 '17 at 17:58
Scott Williams
21 4

you need to use the arrow function in order to bind the 'this': `this.ngOnDestroy = () => { f.bind(this)(); this.componentDestroyed$.complete(); };` – Damsorian Sep 14 '17 at 16:00

1

You usually need to unsubscribe when the components get destroyed, but Angular is going to handle it more and more as we go, for example in new minor version of Angular4, they have this section for routing unsubscribe:

> ## Do you need to unsubscribe?
>
> As described in the ActivatedRoute: the one-stop-shop for route information section of the Routing & Navigation page, the Router manages the observables it provides and localizes the subscriptions. The subscriptions are cleaned up when the component is destroyed, protecting against memory leaks, so you don't need to unsubscribe from the route paramMap Observable.

Also the example below is a good example from Angular to create a component and destroy it after, look at how component implements OnDestroy, if you need onInit, you also can implements it in your component, like implements `OnInit, OnDestroy`

```
import { Component, Input, OnDestroy } from '@angular/core';
import { MissionService } from './mission.service';
import { Subscription }   from 'rxjs/Subscription';

@Component({
  selector: 'my-astronaut',
  template: `
    <p>
      {{astronaut}}: <strong>{{mission}}</strong>
      <button
        (click)="confirm()"
    </p>
  `
})

export class AstronautComponent implements OnDestroy {
  @Input() astronaut: string;
  mission = '<no mission announced>';
  confirmed = false;
  announced = false;
  subscription: Subscription;

  constructor(private missionService: MissionService) {
    this.subscription = missionService.missionAnnounced$.subscribe(
      mission => {
        this.mission = mission;
        this.announced = true;
        this.confirmed = false;
    });
  }

  confirm() {
    this.confirmed = true;
    this.missionService.confirmMission(this.astronaut);
  }

  ngOnDestroy() {
    // prevent memory leak when component destroyed
    this.subscription.unsubscribe();
  }
}
```

answered Oct 16 '17 at 11:52

Alireza
61.5k 15 197 133

2 Confused. What are you saying here? You(Angular recent docs/notes) seem to say that Angular takes care of it and then later to confirm that unsubscribe is a good pattern. Thanks. – jamie Oct 18 '17 at 6:51

Another short addition to the above mentioned situations is:

exists or a new subscription to an all new stream is required (refresh, etc.) is a good example for unsubscription.

answered Jun 16 '18 at 12:49
Krishna Ganeriwal
1,079 9 14

1

In case unsubscribe is needed the following operator for observable pipe method can be used

```
import { Observable, Subject } from 'rxjs';
import { takeUntil } from 'rxjs/operators';
import { OnDestroy } from '@angular/core';

export const takeUntilDestroyed = (componentInstance: OnDestroy) => <T>(observable:
Observable<T>) => {
  const subjectPropertyName = '__takeUntilDestroySubject__';
  const originalOnDestroy = componentInstance.ngOnDestroy;
  const componentSubject = componentInstance[subjectPropertyName] as Subject<any> || new
Subject();

  componentInstance.ngOnDestroy = (...args) => {
    originalOnDestroy.apply(componentInstance, args);
    componentSubject.next(true);
    componentSubject.complete();
  };

  return observable.pipe(takeUntil<T>(componentSubject));
};
```

it can be used like this:

```
import { Component, OnDestroy, OnInit } from '@angular/core';
import { Observable } from 'rxjs';

@Component({ template: '<div></div>' })
export class SomeComponent implements OnInit, OnDestroy {

  ngOnInit(): void {
    const observable = Observable.create(observer => {
      observer.next('Hello');
    })
```

```
      .pipe(takeUntilDestroyed(this))
      .subscribe(val => console.log(val));
  }

  ngOnDestroy(): void {
  }
}
```

The operator wraps ngOnDestroy method of component.

Important: the operator should be the last one in observable pipe.

edited Feb 21 at 10:32

answered Feb 20 at 15:07

Oleg Polezky
535 7 12

0

in SPA application at **ngOnDestroy** function (angular lifeCycle) For each **subscribe** you need to **unsubscribe** it. advantage => to prevent the state from becoming too heavy.

for example: in component1 :

```
import {UserService} from './user.service';

private user = {name: 'test', id: 1}

constructor(public userService: UserService) {
    this.userService.onUserChange.next(this.user);
}
```

in service:

```
import {BehaviorSubject} from 'rxjs/BehaviorSubject';

public onUserChange: BehaviorSubject<any> = new BehaviorSubject({});
```

in component2:

```
private onUserChange: Subscription;

constructor(public userService: UserService) {
    this.onUserChange = this.userService.onUserChange.subscribe(user => {
        console.log(user);
    });
}

public ngOnDestroy(): void {
    // note: Here you have to be sure to unsubscribe to the subscribe item!
    this.onUserChange.unsubscribe();
}
```

answered Apr 19 at 4:50

mojtaba ramezani

635 3 9

0

For handling subscription I use a "Unsubscriber" class.

Here is the Unsubscriber Class.

```
export class Unsubscriber implements OnDestroy {
  private subscriptions: Subscription[] = [];

  addSubscription(subscription: Subscription | Subscription[]) {
    if (Array.isArray(subscription)) {
      this.subscriptions.push(...subscription);
    } else {
      this.subscriptions.push(subscription);
    }
  }

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

And You can use this class in any component / Service / Effect etc.

Example:

```
class SampleComponent extends Unsubscriber {
    constructor () {
        super();
    }

    this.addSubscription(subscription);
}
```

answered May 20 at 10:33

Pratiyush

1 1

0

You can use latest `Subscription` class to unsubscribe for the Observable with not so messy code.

We can do this with `normal variable` but it will be `override the last subscription` on every new subscribe so avoid that, and this approach is very much useful when you are dealing with more number of Obseravables, and type of Obeservables like `BehavoiurSubject` and `Subject`

Subscription

> Represents a disposable resource, such as the execution of an Observable. A Subscription has one important method, unsubscribe, that takes no argument and just disposes the resource held by the subscription.

you can use this in two ways,

- you can directly push the subscription to Subscription Array

```
subscriptions:Subscription[] = [];
```

```
{
            //...
   }));

   this.subscription.push(this.dataService.getFileTracker().subscribe((param: any) => {
        //...
    }));
 }

 ngOnDestroy(){
    // prevent memory leak when component destroyed
    this.subscriptions.forEach(s => s.unsubscribe());
  }
```

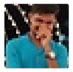
- using **`add()`** of `Subscription`

```
subscriptions = new Subscription();

this.subscriptions.add(subscribeOne);
this.subscriptions.add(subscribeTwo);

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

A **`Subscription`** can hold child subscriptions and safely unsubscribe them all. This method handles possible errors (e.g. if any child subscriptions are null).

Hope this helps.. :)

answered Sep 9 at 7:15



ganesh045

2,065 2 6 19