When to Unsubscribe in Angular

Netanel Basal Follow
Apr 26, 2017 · 2 min read

As you probably know when you subscribe to an observable or event in JavaScript, you *usually* need to unsubscribe at a certain point to release memory in the system. Otherwise, you will have a memory leak.

Let's see the most common cases that you will need to unsubscribe inside the `ngOnDestroy` lifecycle hook.

Forms —

```
export class TestComponent {

  ngOnInit() {
    this.form = new FormGroup({...});
    this.valueChanges  = this.form.valueChanges.subscribe(c
    this.statusChanges = this.form.statusChanges.subscribe(
  }

  ngOnDestroy() {
```

This also applies to any form control.

The Router —

```
export class TestComponent {
  constructor(private route: ActivatedRoute, private router

  ngOnInit() {
    this.route.params.subscribe(console.log);
    this.route.queryParams.subscribe(console.log);
    this.route.fragment.subscribe(console.log);
    this.route.data.subscribe(console.log);
    this.route.url.subscribe(console.log);

    this.router.events.subscribe(console.log);
  }
```

According to the official documentation, Angular should unsubscribe for you, but apparently, there is a bug.

Renderer Service —

```
export class TestComponent {
constructor(private renderer: Renderer2,
            private element : ElementRef) { }

  ngOnInit() {
    this.click = this.renderer.listen(this.element.nativeEl
  }

  ngOnDestroy() {
```

Infinite Observables —

When you have an **infinite** sequence, you should unsubscribe (unless you have a special case), for example when using the `interval()` or the `fromEvent()` observables.

```
export class TestComponent {

  constructor(private element : ElementRef) { }

  interval: Subscription;
  click: Subscription;

  ngOnInit() {
    this.interval = Observable.interval(1000).subscribe(con
    this.click = Observable.fromEvent(this.element.nativeEl
  }

```

Redux Store —

```
export class TestComponent {

  constructor(private store: Store) { }

  todos: Subscription;

  ngOnInit() {
     this.todos = this.store.select('todos').subscribe(cons
  }

```

ngrx/store and redux-angular `select` method returns an observable. Therefore they have to be cleaned.

Don't Unsubscribe

Async pipe —

```
@Component({
  selector: 'test',
  template: `<todos [todos]="todos$ | async"></todos>`
})
export class TestComponent {

  constructor(private store: Store) { }

  ngOnInit() {
```

When the component gets destroyed, the `async` pipe unsubscribes automatically to avoid potential memory leaks.

@HostListener —

```
export class TestDirective {

  @HostListener('click')
  onClick() {
    ....
  }
```

Finite Observable —

When you have a **finite** sequence, **usually** you don't need to unsubscribe, for example when using the `HTTP` service or the `timer` observable.

```
export class TestComponent {

  constructor(private http: Http) { }

  ngOnInit() {
    Observable.timer(1000).subscribe(console.log);
    this.http.get('http://api.com').subscribe(console.log);
  }
```

Final tip —

You should be more declarative and try as little as possible to call the `unsubscribe` method. You can read more about the subject in this

article—RxJS: Don't Unsubscribe.

For example:

```
export class TestComponent {

  constructor(private store: Store) { }

  private componetDestroyed: Subject = new Subject();
  todos: Subscription;
  posts: Subscription;

  ngOnInit() {
     this.todos = this.store.select('todos').takeUntil(this

     this.posts = this.store.select('posts').takeUntil(this
  }
```

Follow me on Medium or Twitter to read more about Angular, Vue and JS!