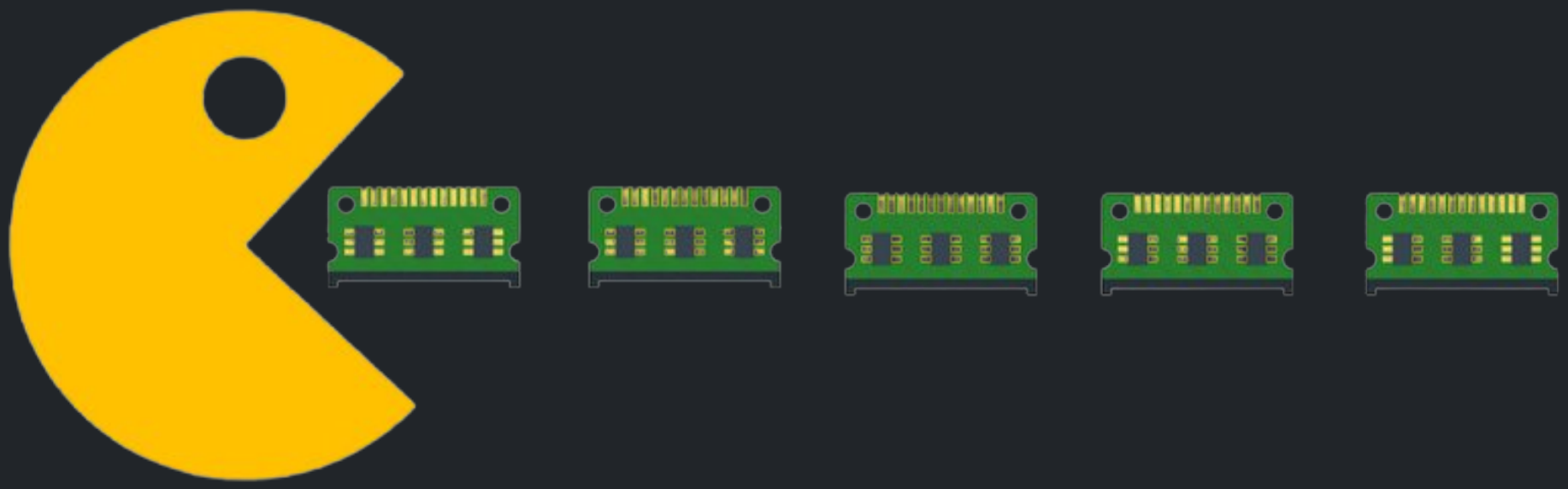
Save it

Like it

Prevent
Memory Leaks
in Angular Observables

@coder_aishya



What is a Memory Leak ?

- Every time you create a subscription to an observable, you're creating a reference to that observable in memory.
- If you're not being careful and unsubscribe then the reference will live in the memory even if the component you created for the observable has been destroyed.
- This is called as a memory leak which can cause a serious problem depends on the size of your application.

@coder_aishya

3 Differrent Ways to Handle Memory Leaks:

1) Unsubscribe using ngOnDestroy

2) TakeUntil Pattern Using Subjects

3) Async Pipe

@coder_aishya

1) Unsubscribe using ngOnDestroy

```
import {Component, OnDestroy, OnInit} from '@angular/core';
import {Subscription} from 'rxjs';
import {AppService} from './app.service';
@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit, OnDestroy{
 title = 'leakage';
 reference: Subscription;
 constructor(private service: AppService) {}
 ngOnInit() {
   this.reference = this.service.getAll()
     .subscribe(res => {
       console.log(res);
     });
 }
 ngOnDestroy() {
   this.reference.unsubscribe();
 }
}
```

@coder_aishya

One problem with unsubscribing using ngOnDestroy() approach is **you have to keep a reference to all the observable you're creating in your component.**

2) TakeUntil Pattern Using Subjects

- takeUntil is a **rxjs operator** which takes an observable as an argument.
- It'll keep the first subscription alive until the second observable finishes
- This is far better and easier than the first approach, the observables used takeUntil will be destroyed once the unsubscribe observable fired.

Take Until Example

```
import {Component, OnDestroy, OnInit} from '@angular/core';
import {Subject} from 'rxjs';
import {takeUntil} from 'rxjs/operators';
import {AppService} from './app.service';
@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit, OnDestroy{
 title = 'leakage';
 private unsubscribe = new Subject();
 constructor(private service: AppService) {}
 ngOnInit() {
   this.service.getAll()
     .pipe(takeUntil(this.unsubscribe))
     .subscribe(res => {
       console.log(res);
     });
 }
 ngOnDestroy() {
   this.unsubscribe.next();
   this.unsubscribe.complete();
 }
}
```

3) Async Pipe

- The async pipe takes an observable as an argument and returns the value from the observable directly to the template.
- So when a component is destroyed we don’t need to do anything because the async pipe will take care of unsubscribing from the observable.
- This is the most performant way of subscribing and unsubscribing to observables.

Async Pipe Example

app.component.ts

```
import {Component, OnInit} from '@angular/core';
import {Observable} from 'rxjs';
import {AppService} from './app.service';
@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit{
 title = 'leakage';
 listItems: Observable<any>;
 constructor(private service: AppService) {}
 ngOnInit() {
   this.listItems = this.service.getAll();
 }
}
```

app.component.html

```
<ul>
 <li *ngFor="let item of listItems | async">{{item.name}}</li>
</ul>
```

Cons of Async Pipe

- can’t reuse a subscription when using an async pipe
- forced to use multiple async pipes for same observable which will create multiple subscriptions

@coder_aishya

Follow me for more

aishwarya-dhuri

coder_aishya