OBSERVABLES + STRUCTURAL DIRECTIVES = ❤️

HEY

NILS MEHLHORN

freelance software engineer
founder of scenelab.io

www nils-mehlhorn.de

@n_mehlhorn

NGRX BOOK

Pay what you want for the complete learning resource

gum.co/angular-ngrx-book

```
@Component({...})
export class UsersComponent implements OnInit {

    users: User[] = []

    constructor(private userService: UserService) {}

    ngOnInit() {
      this.userService.getAll().subscribe(users => {
        this.users = users
      })
    }
}
```

```
<p>{{ users.length }} users online</p>
```

You forgot to unsubscribe! 🙅‍♀️

... do you have to unsubscribe everytime? 🤔

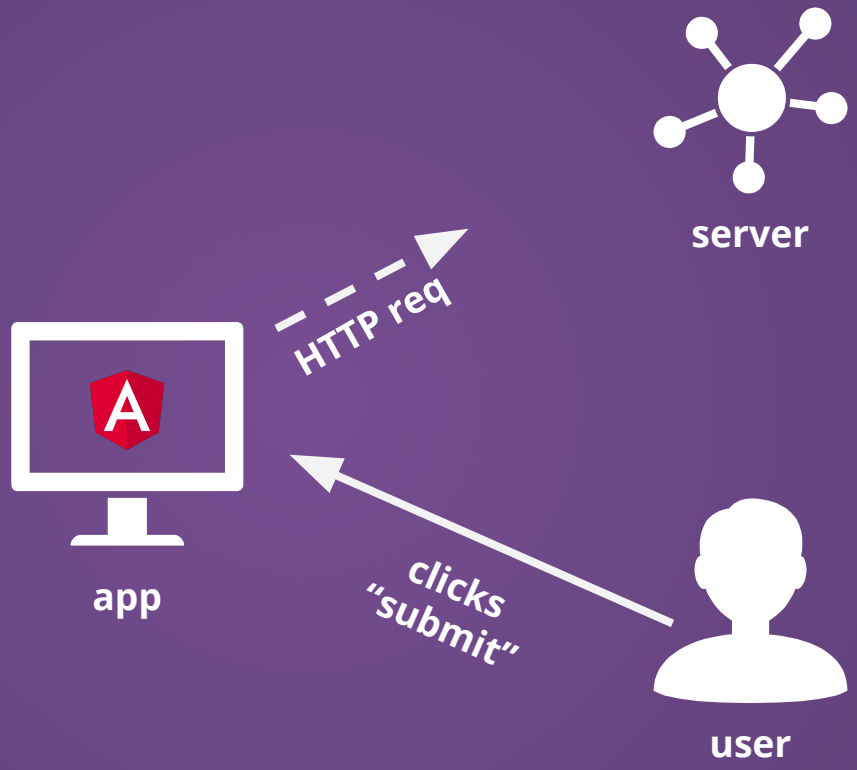
WHY UNSUBSCRIBE?

One-Off Observables

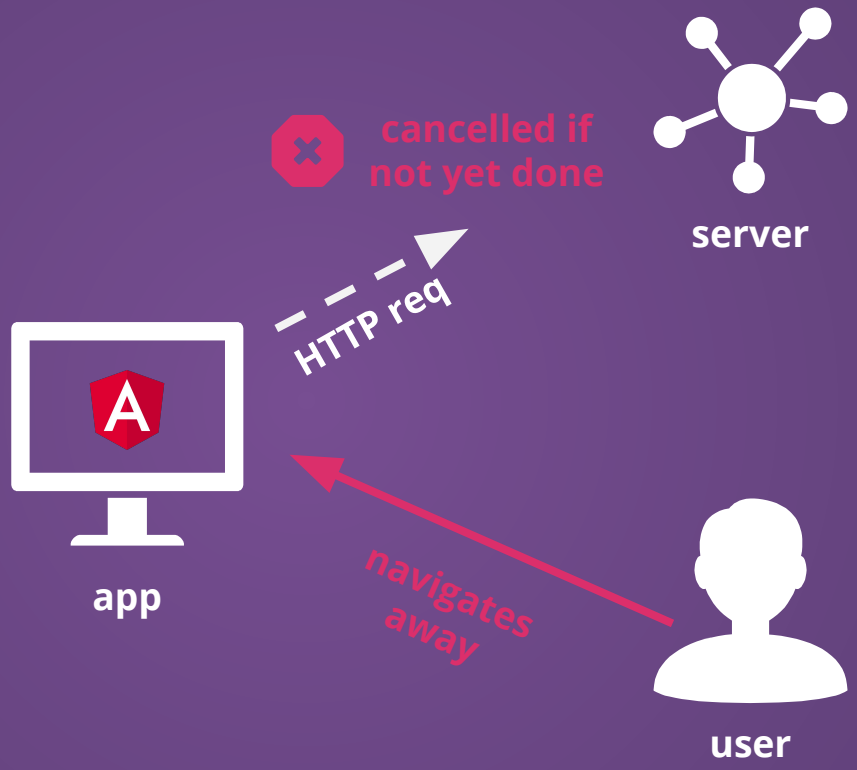
e.g. HTTP request, timer

→ cancellation
→ “observable etiquette”

CANCELLATION



CANCELLATION



WHY UNSUBSCRIBE?

One-Off Observables

e.g. HTTP request, timer

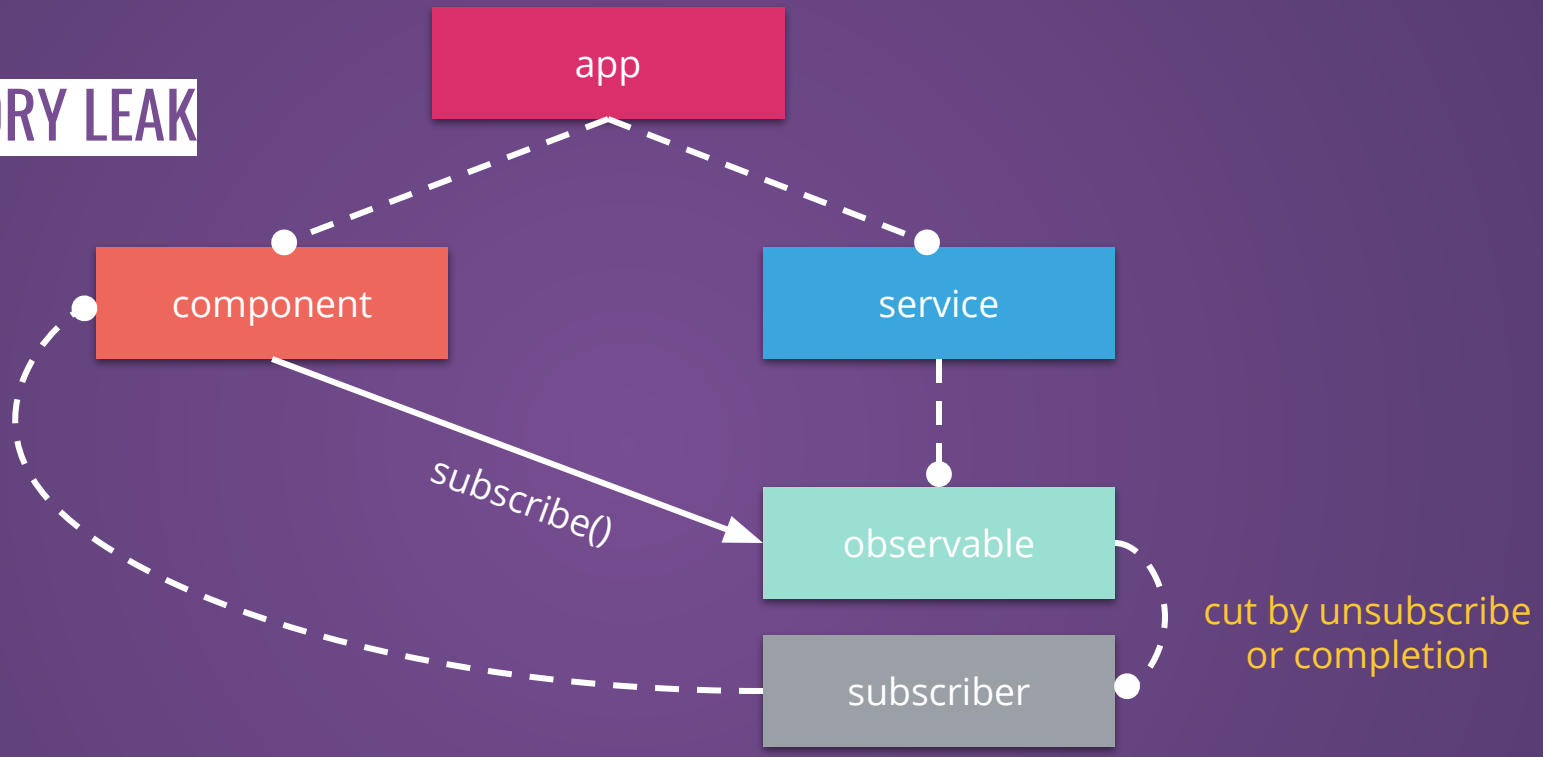
→ cancellation
→ “observable etiquette”

Long-Lived Observables

e.g. store, router events

→ no memory leak

MEMORY LEAK



Recommended Read

How to create a memory leak in Angular -- Kevin Kreuzer

Observable is just a function that takes an observer and returns a function

Ben Lesh

RxJS Lead

callbacks or Subject passed to subscribe()

subscribe()

Observable is just a function that takes an observer and returns a function

cancellation returned by subscribe()

Ben Lesh

RxJS Lead

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

    users: User[]

    subscription: Subscription

    constructor(private userService: UserService) {}

    ngOnInit() {
      this.subscription = this.userService.getAll().subscribe(users => {
        this.users = users
      })
    }

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

IMPERATIVE MANUAL SUBSCRIPTION MANAGEMENT

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

    users: User[]

    destroy$ = new Subject<void>()

    constructor(private userService: UserService) {}

    ngOnInit() {
      this.userService.getAll()
        .pipe(takeUntil(this.destroy$)
        .subscribe(users => {
          this.users = users
        })
    }

    ngOnDestroy() {
      this.destroy$.next()
    }
}
```

DECLARATIVE MANUAL SUBSCRIPTION MANAGEMENT

MANUAL SUBSCRIPTION MANAGEMENT

rxjs-tslint-rules

- 👍 full control
- 👍 access to values from other methods
- 👍 falsy values

- 👎 verbose & error-prone
- 👎 OnPush change detection requires trigger

⚠ required for observables not reflected in view

(e.g. updating a user)

```
@Component({
  ...
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class UsersComponent implements OnInit {

    users$: Observable<User[]>

    constructor(private userService: UserService) {}

    ngOnInit() {
      this.users$ = this.userService.getAll()
    }
}
```

```
<p *ngIf="users$ | async as users; else loading">
  {{ users.length }} users online
</p>
<ng-template #loading>Loading…</ng-template>
```

AUTOMATIC SUBSCRIPTION MANAGEMENT WITH NGIF & ASYNCPIPE

```
@Component({
  ...
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class UsersComponent implements OnInit {

    users$: Observable<User[]>

    constructor(private userService: UserService) {}

    ngOnInit() {
      this.users$ = this.userService.getAll()
    }
}
```

```
<p *ngIf="users$ | async as users; else loading">
  {{ users.length }} users online
</p>
<ng-template #loading>Loading…</ng-template>
```

AUTOMATIC SUBSCRIPTION MANAGEMENT WITH NGIF & ASYNCPIPE

ASYNCPIPE

- unsubscribes
- triggers change detection

```
@Pipe({name: 'async', pure: false})
export class SimpleAsyncPipe implements OnDestroy, PipeTransform {

  private latestValue: any = null
  private subscription: Subscription = null

  constructor(private cd: ChangeDetectorRef) {}

  transform(observable: Observable<any>): any {
    this.subscription = observable.subscribe(value => {
        this.latestValue = value
        this.cd.markForCheck()
    })
    return WrappedValue.wrap(this.latestValue)
  }

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

```
@Component({
  ...
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class UsersComponent implements OnInit {

    users$: Observable<User[]>

    constructor(private userService: UserService) {}

    ngOnInit() {
      this.users$ = this.userService.getAll()
    }
}
```

```
<p *ngIf="users$ | async as users; else loading">
  {{ users.length }} users online
</p>
<ng-template #loading>Loading…</ng-template>
```

AUTOMATIC SUBSCRIPTION MANAGEMENT WITH NGIF & ASYNCPIPE

ONPUSH CHANGE DETECTION

updates view only when

1. @Inputs are reassigned
2. events occur on component or children
3. markForCheck() called

➜ faster due to less updates

source: angular/change_detection_spec.ts L282

```
@Component({
  ...
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class UsersComponent implements OnInit {

    users$: Observable<User[]>

    constructor(private userService: UserService) {}

    ngOnInit() {
      this.users$ = this.userService.getAll()
    }
}
```

```
<p *ngIf="users$ | async as users; else loading">
  {{ users.length }} users online
</p>
<ng-template #loading>Loading…</ng-template>
```

AUTOMATIC SUBSCRIPTION MANAGEMENT WITH NGIF & ASYNCPIPE

Structural directives are responsible for HTML layout.

They shape or reshape the DOM's structure, typically by adding, removing, or manipulating elements.

Angular Docs

STRUCTURAL DIRECTIVES: MICROSYNTAX

```
<p *ngIf="users$ | async as users; else loading">
  {{ users.length }} users online
</p>
<ng-template #loading>Loading…</ng-template>
```

microsyntax desugaring

```
<ng-template [ngIf]="users$ | async as users" [ngIfElse]="loading">
  <p>{{ users.length }} users online</p>
</ng-template>
<ng-template #loading>Loading…</ng-template>
```

STRUCTURAL DIRECTIVES: NGIF

```
<ng-template
  [ngIf]="users$ | async as users"
  [ngIfElse]="loading">
  <p>{{ users.length }} online</p>
</ng-template>
<ng-template #loading>
  Loading…
</ng-template>
```

```
interface NgIfContext<T> {
  $implicit: T
  ngIf: T
}
```

```
@Directive({selector: '[ngIf]'})
export class SimpleNgIf<T> {

  elseTemplate: TemplateRef
  context: NgIfContext<T> = {}

  constructor(private view: ViewContainerRef,
              private template: TemplateRef<NgIfContext<T>>) {}

  @Input()
  set ngIfElse(template: TemplateRef) {
    this.elseTemplate = template
  }

  @Input()
  set ngIf(condition: T) {
    this.context.$implicit = this.context.ngIf = condition
    this.view.clear()
    if (condition) {
      this.view.createEmbeddedView(this.template, this.context)
    } else {
      this.view.createEmbeddedView(this.elseTemplate)
    }
  }
}
```

POP QUIZ: NGIF

explicit binding to ngIf-property

explicit binding to $implicit-property

```
<ng-template [ngIf]="'hello'" let-a="$implicit"
             let-b="ngIf" let-c>
  <p>{{ a }}</p>
  <p>{{ b }}</p>
  <p>{{ c }}</p>
</ng-template>
<p *ngIf="'hello' as d">{{ d }}</p>
```

implicit binding to ngIf-property

implicit binding to $implicit-property

What's the output?

1. hello, ngIf, undefined, hello
2. undefined, undefined, hello, hello
3. hello, hello, hello, hello
4. hello, undefined, hello, hello

```
interface NgIfContext<T> {
  $implicit: T
  ngIf: T
}
```

```
this.context.$implicit =
  this.context.ngIf =
    condition // 'hello'
```

NGIF & ASYNCPIPE

- 👍 succinct
- 👍 OnPush change detection
- 👍 fallback template

- 👎 no falsy values
- 👎 no access to errors
- 👎 same template for loading and error states
- 👎 (no access to values from other methods)

*observe

A Structural Directive for Observables

```
<p *observe="users$ as users; before loadingTemplate; error errorTemplate">
  {{ users.length }} users online
</p>
<ng-template #loadingTemplate>
  <p>Loading ...</p>
</ng-template>
<ng-template #errorTemplate let-error>
  <p>{{ error }}</p>
</ng-template>
```

DEMO

AUTOMATIC SUBSCRIPTION MANAGEMENT WITH OBSERVE

```
<p *observe="users$ as users;
            before loadingTemplate;
            error errorTemplate">
  {{ users.length }} users online
</p>
<ng-template #loadingTemplate>
  <p>Loading ...</p>
</ng-template>
<ng-template #errorTemplate let-error>
  <p>{{ error }}</p>
</ng-template>
```

```
@Directive({
  selector: "[observe]"
})
export class ObserveDirective<T> implements OnDestroy,OnInit {

  constructor(
    private view: ViewContainerRef,
    private nextRef: TemplateRef<ObserveContext<T>>,
    private changes: ChangeDetectorRef
  ) {}
  ...
}
```

```
<p *observe="users$ as users;
             before loadingTemplate;
             error errorTemplate">
  {{ users.length }} users online
</p>
<ng-template #loadingTemplate>
  <p>Loading ...</p>
</ng-template>
<ng-template #errorTemplate let-error>
  <p>{{ error }}</p>
</ng-template>
```

```
@Directive({
  selector: "[observe]"
})
export class ObserveDirective<T> implements OnDestroy,OnInit {

  constructor(
    private view: ViewContainerRef,
    private nextRef: TemplateRef<ObserveContext<T>>,
    private changes: ChangeDetectorRef
  ) {}
  ...
}
```

```
interface ObserveContext<T> {
  $implicit: T
  observe: T
}
```

```
<p *observe="users$ as users;
            before loadingTemplate;
            error errorTemplate">
  {{ users.length }} users online
</p>
<ng-template #loadingTemplate>
  <p>Loading ...</p>
</ng-template>
<ng-template #errorTemplate let-error>
  <p>{{ error }}</p>
</ng-template>
```

```
@Directive({
  selector: "[observe]"
})
export class ObserveDirective<T> implements OnDestroy,OnInit {
  ...
  @Input()
  set observeBefore(ref: TemplateRef<null>) {
    this.beforeRef = ref;
  }

  @Input()
  set observeError(ref: TemplateRef<ErrorContext>) {
    this.errorRef = ref;
  }
}
```

```
<p *observe="users$ as users;
             before loadingTemplate;
             error errorTemplate">
  {{ users.length }} users online
</p>
<ng-template #loadingTemplate>
  <p>Loading ...</p>
</ng-template>
<ng-template #errorTemplate let-error>
  <p>{{ error }}</p>
</ng-template>
```

```
@Directive({
  selector: "[observe]"
})
export class ObserveDirective<T> implements OnDestroy,OnInit {
  ...
  @Input()
  set observeBefore(ref: TemplateRef<null>) {
    this.beforeRef = ref;
  }

  @Input()
  set observeError(ref: TemplateRef<ErrorContext>) {
    this.errorRef = ref;
  }
}
```

```
interface ErrorContext {
  $implicit: Error
}
```

```
<p *observe="users$ as users;
            before loadingTemplate;
            error errorTemplate">
  {{ users.length }} users online
</p>
<ng-template #loadingTemplate>
  <p>Loading ...</p>
</ng-template>
<ng-template #errorTemplate let-error>
  <p>{{ error }}</p>
</ng-template>
```

```
@Directive({
  selector: "[observe]"
})
export class ObserveDirective<T> implements OnDestroy,OnInit {
  ...
  @Input()
  set observe(source: Observable<T>) {
    this.view.createEmbeddedView(this.beforeRef)
    source.pipe(takeUntil(this.destroy$))
    .subscribe(value => {
      this.view.clear()
      this.view.createEmbeddedView(this.nextRef,
        {$implicit: value, observe: value})
      this.changes.markForCheck()
    }, error => {
      this.view.clear()
      this.view.createEmbeddedView(this.errorRef,
        {$implicit: error})
      this.changes.markForCheck()
    })
  }
}
```

OBSERVE

- 👍 succinct
- 👍 OnPush change detection
- 👍 loading & error templates
- 👍 falsy values
- 👍 access to errors

- 👎 (no access to values from other methods)

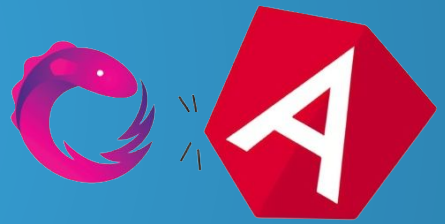
COMPARISON

	OBSERVABLES NOT REFLECTED IN VIEW	FALSY VALUES	ONPUSH SUPPORT	LOADING & ERROR TEMPLATES	ACCESS TO ERRORS
Manual Subscription	YES	YES	MANUAL	MANUAL	MANUAL
NgIf & AsyncPipe	NO	NO	YES	MANUAL	MANUAL
Observe	NO	YES	YES	YES	YES

LIBRARIES

ngx-observe

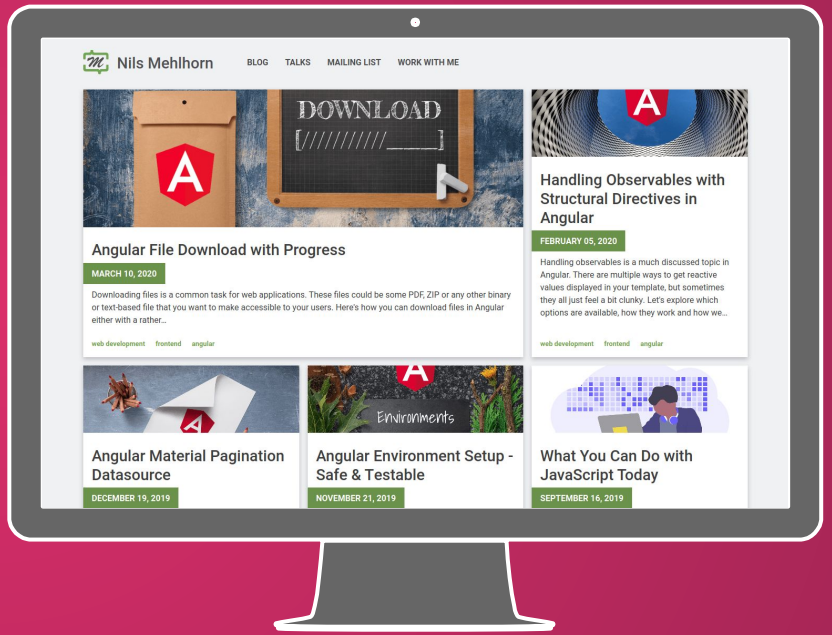
@ngrx/component



@rx-angular/template

THANKS

1. Visit Blog
2. Join Mailing List
3. Follow On Twitter
4. Work With Me



www nils-mehlhorn.de

@n_mehlhorn