Workshop

Angular Subjects

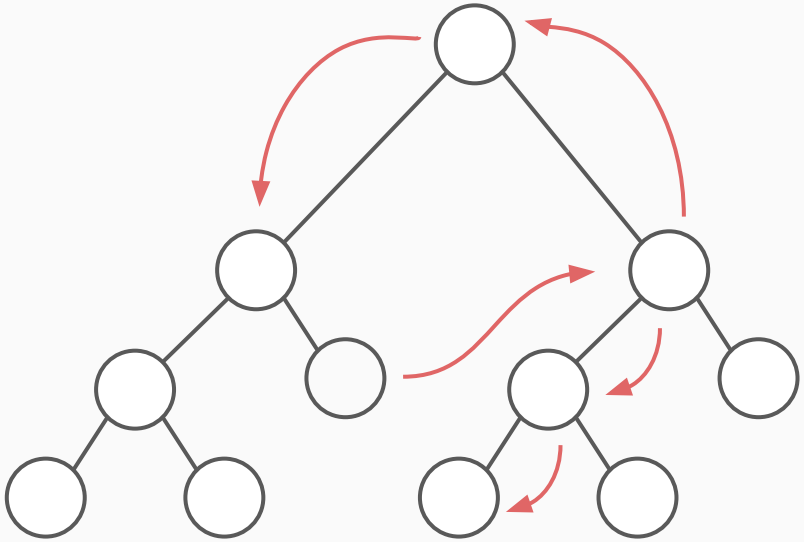
Angular Subjects

Helps to manage the state of your application

Why?

- Unidirectional data flow
- Predictable state changes and rendering
- Helping you application to be more “reactive”

Why?



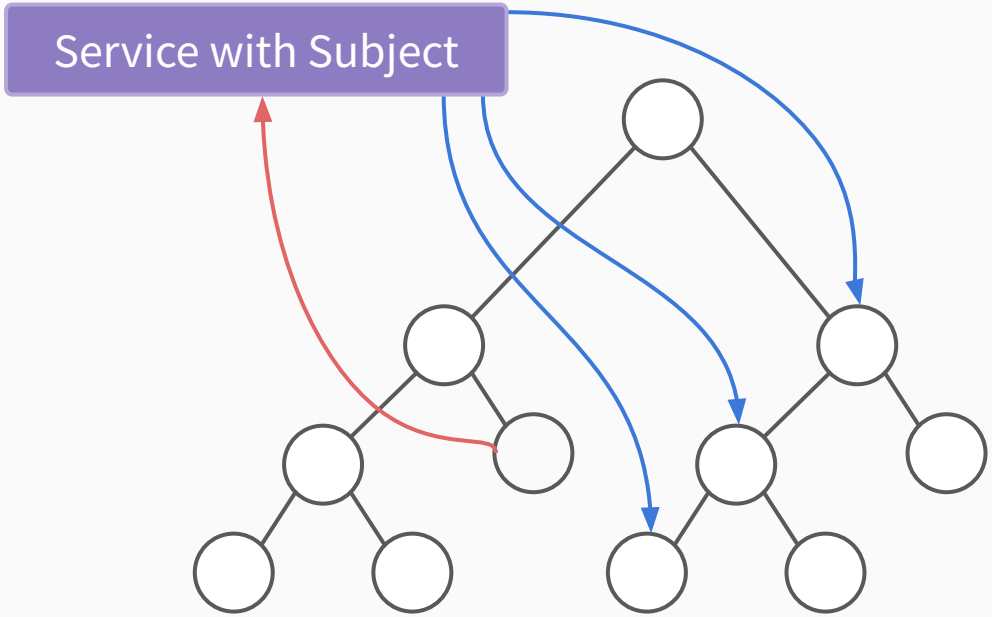
This is how we manage state at the moment:

@Input()
@Output()

State management with Subjects

- Subjects are Observables but also Observers themselves
- Components can subscribe to Subjects
- Subjects **can emit data** too

State management with Subjects



Everything is dispatched from and to **one global store**

Creating a Subject


```
let subject = new Subject<string>();

// We subscribe to the subject
subject.subscribe((data) => {
 console.log(`Hello ${data}`)
});


subject.next('Angular');
// Hello Angular
```

Task

Create a HeaderService with a Subject

Subjects are multicast

<code>

```
let subject = new Subject<string>();
subject.subscribe((data) => {
 console.log(`Subscriber 1 received ${data}`);
});

subject.subscribe((data) => {
 console.log(`Subscriber 2 received ${data}`);
});

subject.next('Hello Angular');

// Subscriber 1 received Hello Angular
// Subscriber 2 received Hello Angular
```

Don’t expose Subjects directly !!!

- Subscribers will be able to “mess up” with your Subjects
- Return an Observable:

```
private subject = new Subject<string>();

observable$ = this.subject.asObservable();
```

Task

Change Headertitle on Navigation

Using Subjects to unsubscribe

- We need to unsubscribe of all Subscriptions (otherwise we might geht memory leaks)
- But that can get really messy:

```
subsription1 = observable1$.subscribe((data) => {});
subsription2 = observable2$.subscribe((data) => {});
subsription3 = observable3$.subscribe((data) => {});
subsription4 = observable4$.subscribe((data) => {});

//ngOnDestroy:
subsription1.unsubscribe()
subsription2.unsubscribe()
subsription3.unsubscribe()
subsription4.unsubscribe()
```

Using Subjects to unsubscribe

<code>

```
let destroy$ = new Subject<boolean>();

this.apiService.getObservable().pipe(
     takeUntil(this.destroy$)
)
.subscribe((data) => {
  ...
});


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

Task

Use takeUntil()-Pattern

Other Subjects?

- A simple subject is not keeping the state
- Subscribers of subjects after value was emitted are not getting it

BehaviourSubject

- BehaviourSubject always stores the last emitted Value
- It needs a default Value to

```
private behaviourSubject = new BehaviourSubject<string>('default');
```

ReplaySubject

- ReplaySubjects always stores the last emitted Value**s**
- It needs the amount of Values it should store

```
private replaySubject = new ReplaySubject<string>(11);
```