Using a Declarative Approach

Deborah Kurata

Developer

https://www.youtube.com/@deborah_kurata

Retrieving Data

Procedural

```
getProducts(): Observable<Product[]> {
  return this.http.get<Product[]>(this.url)
    .pipe(
      tap(data => console.log(data)),
      catchError(err => this.handleError(err))
    );
}
```

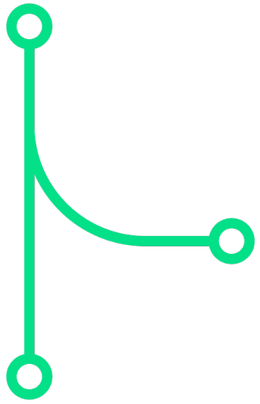
Declarative

```
products$ = this.http.get<Product[]>(this.url)
    .pipe(
      tap(data => console.log(data)),
      catchError(err => this.handleError(err))
    );
```

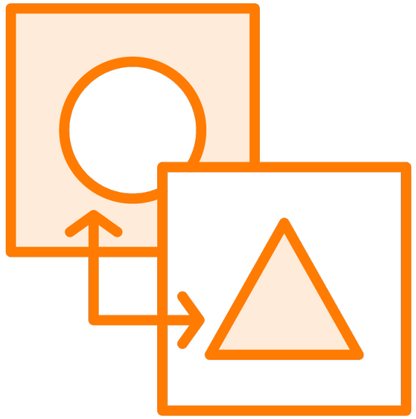
Why?

Benefits of a Declarative Approach

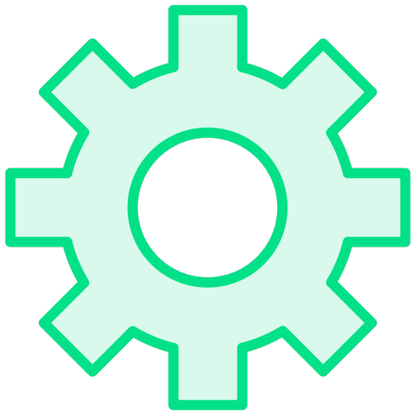
React to user actions and data emissions



Merge data from multiple sources



Share data between components



More easily use features like the async pipe

Retrieving Data - Component

Procedural

```
ngOnInit(): void {
  this.sub = this.productService.getProducts().subscribe(
        products => this.products = products
  );
}
ngOnDestroy(): void {
  this.sub.unsubscribe();
}
```

Declarative

```
products$ = this.productService.products$;
```

What about subscribe and unsubscribe?

Overview

Use a declarative approach

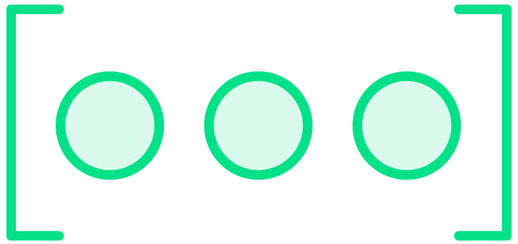
Examine the async pipe

Explore data caching

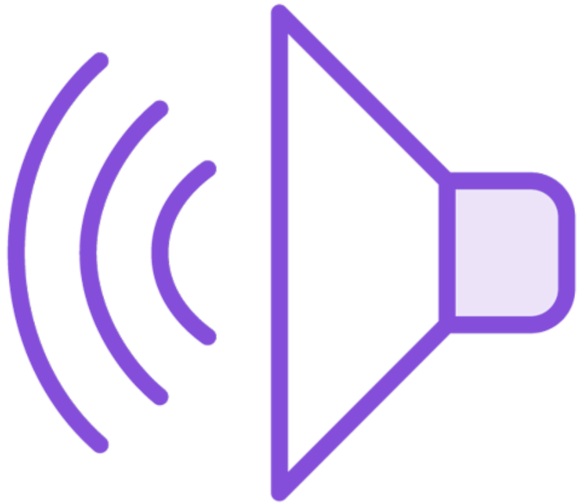
Demo

Transform to a declarative approach

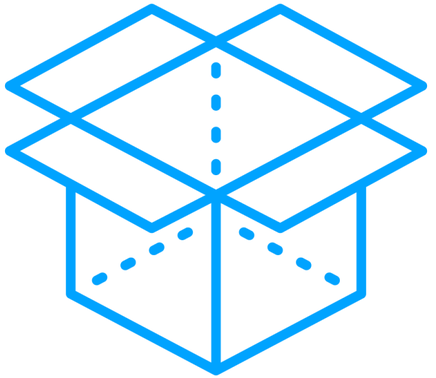
Displaying Retrieved Data in the Template



Copy emitted data to
a local variable,
use the local variable



Work directly with the
emitted data
(async pipe)



Signals!

Async Pipe

Work directly with data emitted from an observable

RxJS pipe

```
this.products$ = this.productService.products$
    .pipe(
      tap(data => console.log(data))
    );
```

Async pipe

```
<button type='button'
        *ngFor='let product of products$ | async'
        (click)='onSelected(product.id)'>
    {{ product.productName }}
</button>
```

Async Pipe

Automatically subscribes and unsubscribes

```
<button type='button'
        *ngFor='let product of products$ | async'
        (click)='onSelected(product.id)'>
    {{ product.productName }}
</button>
```

Async Pipe

Assigns the emitted item to a variable

```
<div *ngIf="products$ | async as products">
   <button type='button'
           *ngFor='let product of products'
            (click)='onSelected(product.id)'>
           {{ product.productName }}
   </button>
 </div>
```

Demo

Display retrieved data

- Work directly with the data emitted from the observable (async pipe)

Benefits of the async Pipe

Automatically handles subscribe and unsubscribe

No need to manage a local variable to hold emitted items

Simplifies our component code

Improve performance with OnPush change detection strategy

Minimize the number of async pipes in a template

Key Point

```
<div *ngIf="products$ | async as products">
```

Why?

Each async pipe is a subscription

How?

Use the as clause
Combine multiple observables, use one async pipe for the set

Caching Observables

Retain retrieved data locally

Reuse previously retrieved data

Stored in memory or external

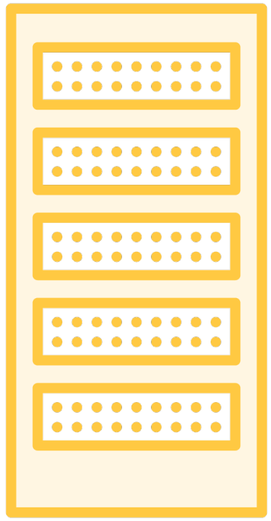
Demo

Why is caching useful?

Advantages of Caching Data

Improves responsiveness

Reduces bandwidth and network consumption



Reduces backend server load

Reduces redundant computations

Classic Caching Pattern

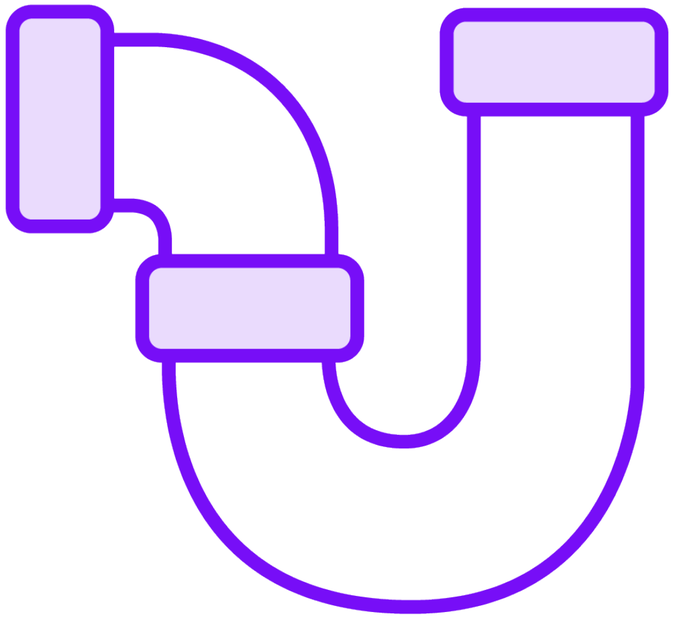
```
private products: Product[] = [];

getProducts(): Observable<Product[]> {
  if (this.products) {
    return of(this.products);
  }
  return this.http.get<Product[]>(this.url)
    .pipe(
      tap(data => this.products = data),
      catchError(err => this.handleError(err))
    );
}
```

Declarative Caching Pattern

```
products$ = this.http.get<Product[]>(this.url)
  .pipe(
    shareReplay(1),
    catchError(err => this.handleError(err))
  );
```

RxJS Operator: shareReplay



Shares its source with other subscribers

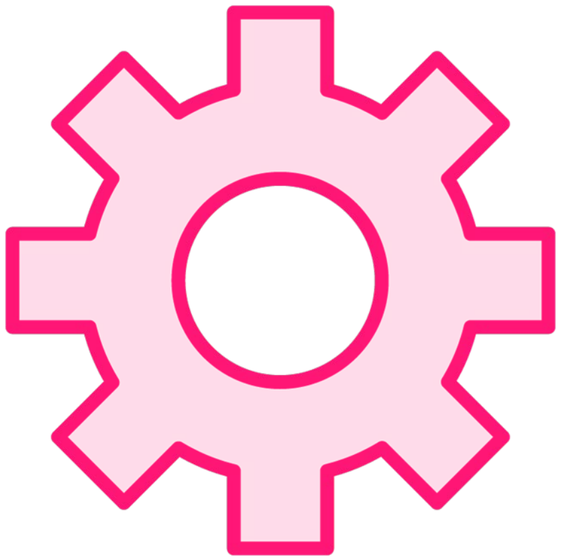
Replays the defined number of emissions to the subscriber

```
shareReplay(1)
```

Used for

- Sharing observables
- Caching data in the application
- Replaying emissions to late subscribers

RxJS Operator: shareReplay



`shareReplay` is a multicast operator

Returns a Subject that shares a single subscription to the underlying source

Takes in an optional buffer size, which is the number of items cached and replayed

On a subscribe, it replays the specified number of emissions

The data stays cached, even after there are no more subscribers

Demo

Cache data with shareReplay

Declarative Approach

Use a declarative approach

- Service

```
products$ = this.http.get<Product[]>(this.url)
  .pipe(
    catchError(err => this.handleError(err))
  );
```

- Component

```
products$ = this.productService.products$
  .pipe(
    catchError(err => {
      this.errorMessage = err;
      return EMPTY;
    })
  );
```

async Pipe

Use the async pipe to display observable emissions in the template

```
<button type='button'
        *ngFor='let product of products$ | async'
        (click)='onSelected(product.id)'>
   {{ product.productName }}
</button>
```

Automatically handles subscribe and unsubscribe

Caching

Add shareReplay to any observable pipeline you wish to share and replay to all new subscribers

```
products$ = this.http.get<Product[]>(this.url)
  .pipe(
    shareReplay(1),
    catchError(err => this.handleError(err))
  );
```

Best Practices

Minimize the number of async pipes in a template

Take care where you add a `shareReplay` in the pipeline

- Before: Processed before caching the data
- After: Re-executed for each subscription

Be sure to consider invalidating the cache you define with `shareReplay`

Cache Invalidation

Clear the cache at some point

Evaluate

- Fluidity of data
- Users' behavior

Consider

- Invalidating the cache on a time interval
- Allowing the user to control when data is refreshed
- Always getting fresh data on update operations

For More Information

Demo code

- https://github.com/DeborahK/angular-rxjs-signals-fundamentals

Angular documentation

- https://angular.io/guide/observables-in-angular#async-pipe

RxJS documentation

- https://rxjs.dev/api/index/function/shareReplay

"Declarative Pattern for Getting Data from an Observable"

- https://youtu.be/0XPxUa8u-LY

Up Next:

Reacting to Actions: Subject and BehaviorSubject