Going Reactive

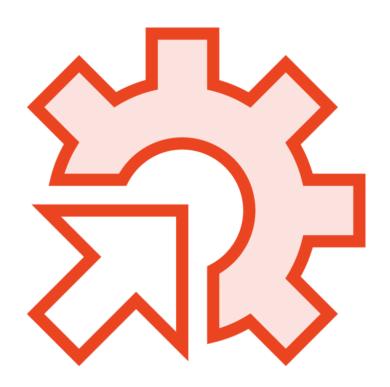


Deborah Kurata Consultant | Speaker | Author | MVP | GDE

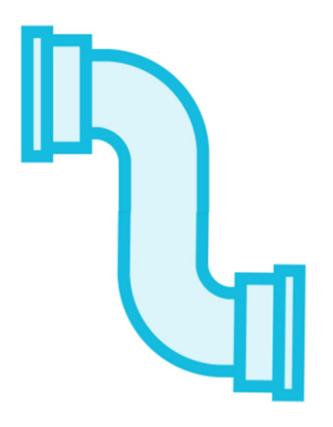
@deborahkurata



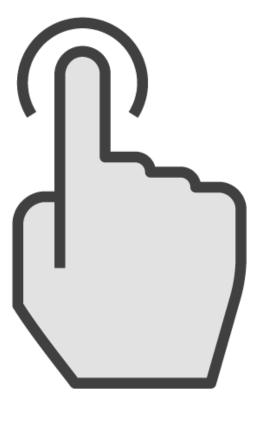
Going Reactive



Working with Observables directly



Leverage RxJS operators



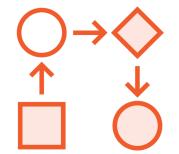
React to actions



Going Reactive



Improves performance



Handles state



Reacts to user actions



Simplifies code



Module Overview



Working with the async pipe

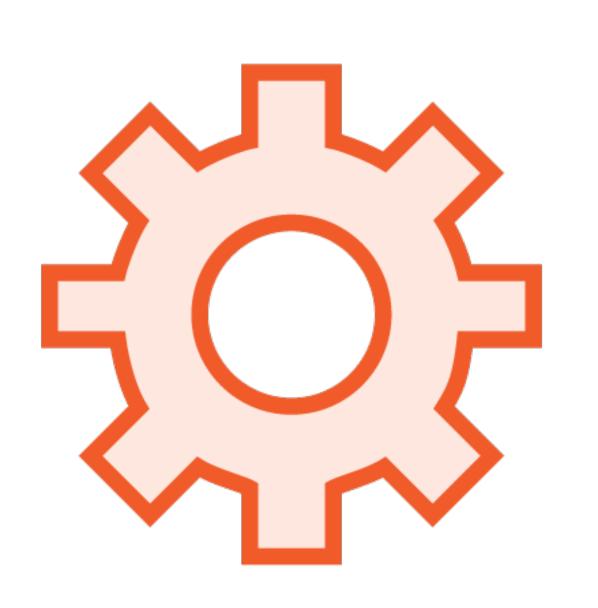
Handling errors

Improving change detection

Declarative pattern for data retrieval



RxJS Features

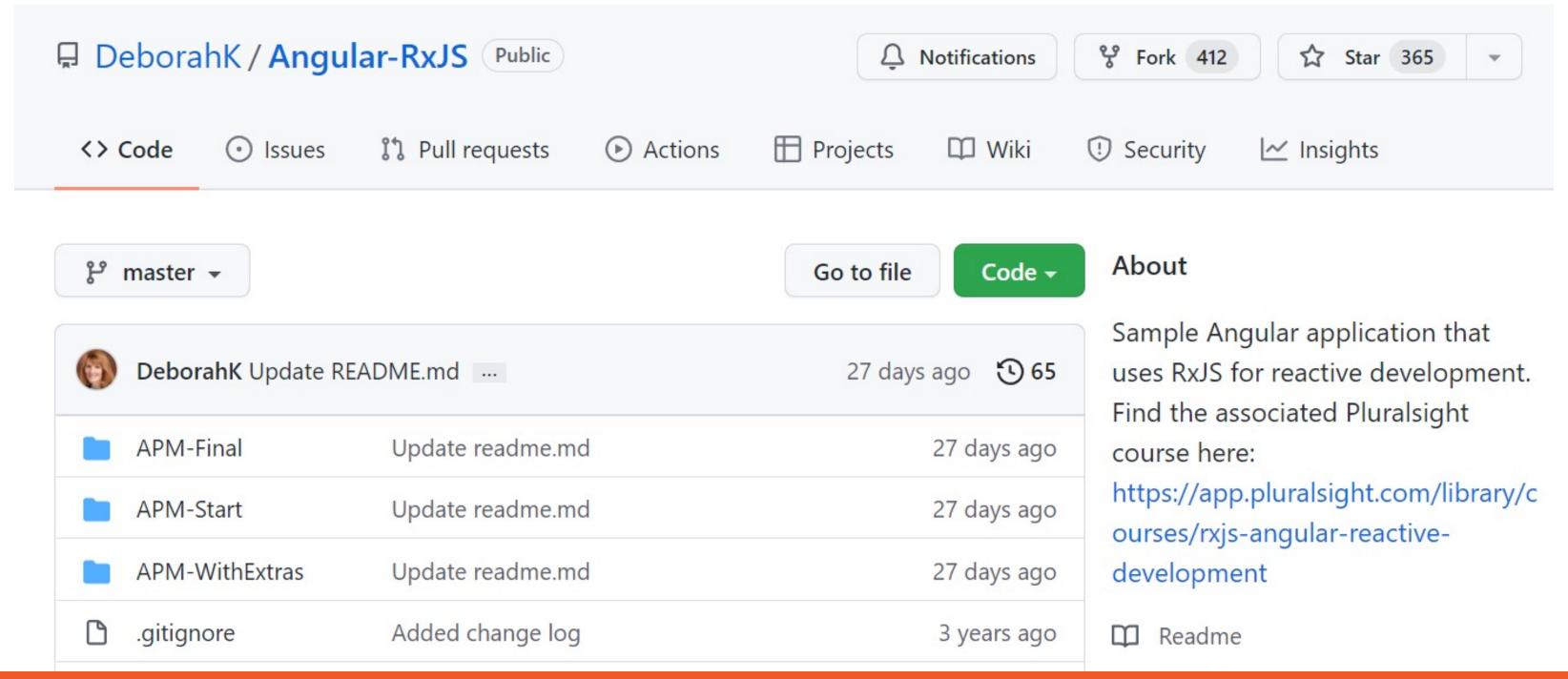


catchError

EMPTY

throwError

GitHub Repository



https://github.com/DeborahK/Angular-RxJS

Demo



Procedural data retrieval pattern



Async Pipe

"products\$ | async"

Subscribes to the Observable when component is initialized

Returns each emitted value

When a new item is emitted, component is marked to be checked for changes

Unsubscribes when component is destroyed



Common Pattern with an Async Pipe

Product List Component

```
products: Product[] = [];
constructor(private productService: ProductService) { }

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

Product List Component

```
products$: Observable<Product[]>;
constructor(private productService: ProductService) { }
ngOnInit(): void {
  this.products$ = this.productService.getProducts();
}
```



Template with an Async Pipe

Product List Template

```
<div *ngIf="products">

        {{ product.productName }}
```

Product List Template

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

        {{ product.productName }}

        </td
```

Handling Observable Errors

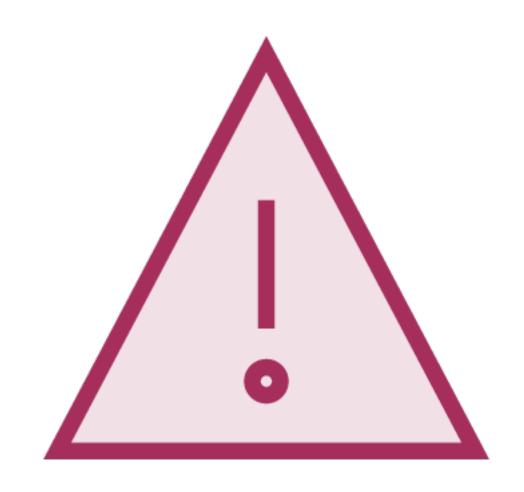
There are places things could go wrong

Catch Observable errors

Error stops the Observable

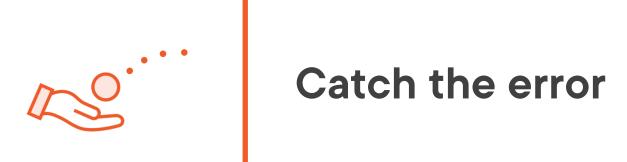
It won't emit any more items

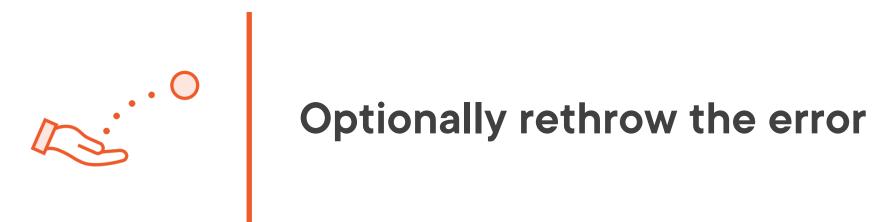
We can't use it anymore





Handling Observable Errors







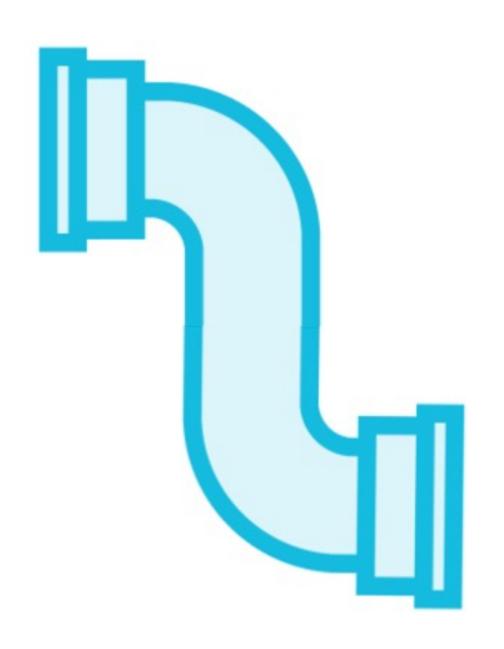
Replace the errored Observable with a new Observable



RxJS Error Handling Features



RxJS Operator: catchError



Catches any errors that occur on an Observable

catchError(this.handleError)

Must be after any operator that could generate an error

Used for catching errors

In the error handler:

- Rethrow the error
- Replace the errored Observable to continue after an error occurs

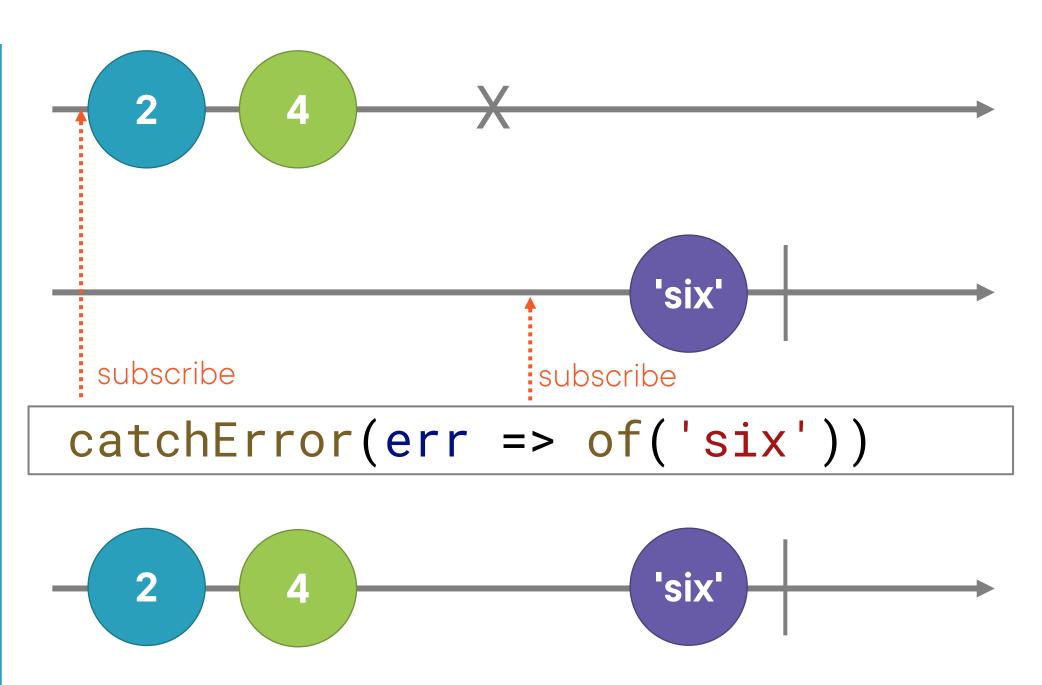


Catching an Error

```
of (2, 4, 6)
  .pipe(
  map(i => {
    if (i === 6) {
     throw 'Error!';
    return i;
   catchError(err => of('six'))
  .subscribe({
    next: x => console.log(x),
    error: err => console.log(err)
  });
```

Marble Diagram: catchError

```
of(2, 4, 6)
  .pipe(
   map(i => {
    if (i === 6) {
      throw 'Error!';
    return i;
   }),
   catchError(err => of('six'))
  .subscribe({
    next: x => console.log(x),
    error: err => console.log(err)
  });
                Console
```



2 4

'six'

RxJS Creation Function: throwError



Creates an Observable that emits no items

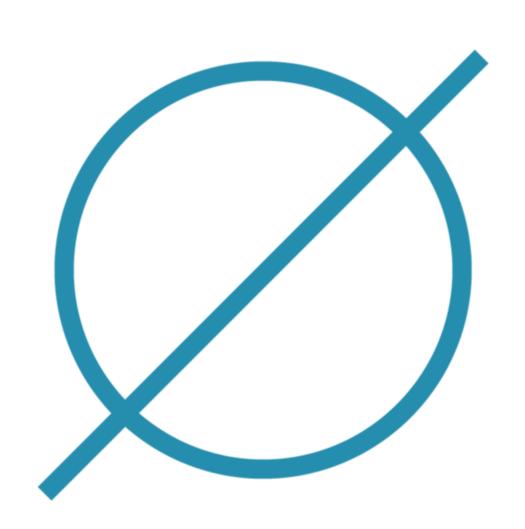
Immediately emits an error notification

Used for

- Propagating an error

Or use the throw statement

RxJS Constant: EMPTY



Is an Observable that emits no items

return EMPTY;

Immediately emits a complete notification

Used for

- Returning an empty Observable

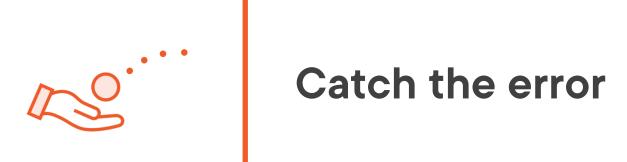


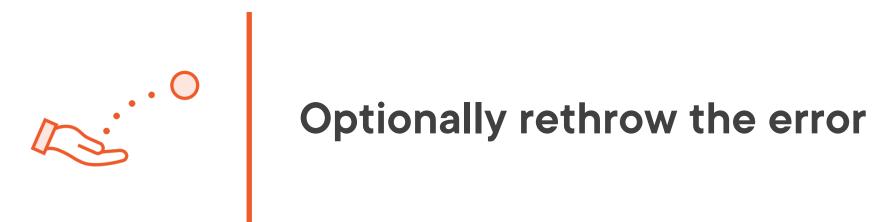
Error Handling: Service

Product Service

```
getProducts(): Observable<Product[]> {
   return this.http.get<Product[]>(this.productsUrl)
        .pipe(
        tap(data => console.log(JSON.stringify(data)))
        );
}
```

Handling Observable Errors







Replace the errored Observable with a new Observable



Error Handling: Service

Product Service

```
getProducts(): Observable<Product[]> {
  return this.http.get<Product[]>(this.productsUrl)
    .pipe(
      tap(data => console.log(JSON.stringify(data))),
      catchError(err => {
```

Replacing an Errored Observable



An Observable created from hard-coded or local data

An Observable that emits an empty value or empty array

The EMPTY RxJS constant

Catch and Replace

Product Service

```
getProducts(): Observable<Product[]> {
  return this.http.get<Product[]>(this.productsUrl)
    .pipe(
      tap(data => console.log(JSON.stringify(data))),
      catchError(err => {
        return of([{ id: 1, productName: 'cart' },
                   { id: 2, productName: 'hammer' }]);
```

Catch and Rethrow

Product Service

```
getProducts(): Observable<Product[]> {
  return this.http.get<Product[]>(this.productsUrl)
    .pipe(
      tap(data => console.log(JSON.stringify(data))),
      catchError(err => {
        console.error(err);
        throw new Error('Could not retrieve'));
```

Error Handling: Component

Product List Component

```
this.productService.getProducts()
   .subscribe(
    products => this.products = products,
    err => this.errorMessage = err
   );
```

Product Component

```
this.products$ = this.productService.getProducts();
```

Error Handling: Component

Product List Component

```
this.productService.getProducts()
   .subscribe({
    next: products => this.products = products,
    error: err => this.errorMessage = err
});
```

Product Component

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

Demo



Handling errors

Benefits of an Async Pipe



Angular uses change detection to track changes to application data so that it knows when to update the UI



Change Detection Strategies

Default

Uses the default checkAlways strategy

Every component is checked when:

- Any change is detected

OnPush

Improves performance by minimizing change detection cycles

Component is only checked when:

- @Input properties change
- Event emits
- A bound Observable emits

```
@Component({
  templateUrl: './product-list.component.html',
  changeDetection: ChangeDetectionStrategy.OnPush
})
```

Procedural Pattern

Product Service

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

Product Component

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

Declarative Pattern: Service

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

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

Declarative Pattern: Component

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

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

Declarative Pattern

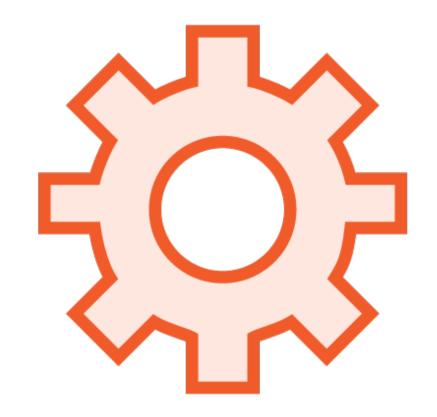
Product Service

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

Product Component

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

Benefits of a Declarative Approach



Leverages the power of RxJS Observables and operators



Combine data streams



React to user actions



RxJS Checklist: Going Reactive



Work with Observables directly

Use an async pipe in the template to display emitted items

Declare Observable properties

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

RxJS Checklist: Handling Parameters



Procedural approach: pass parameters

```
getProducts(pageNumber): Observable<Product[]> {
    return this.http.get<Product[]>(this.productsUrl, {
        params: { page: pageNumber }
    });
}
```

Declarative approach: react to actions

```
products$ = this.requestedPage$
   .pipe(
       switchMap(pageNumber =>
            this.http.get<Product[]>(this.productsUrl, {
            params: { page: pageNumber }
            }))
    );
```

RxJS Checklist: Async Pipe



Automatically subscribes when the component is rendered

Processes each emitted item

Automatically unsubscribes when the component is destroyed

Use the 'as' syntax to map each emitted item to a variable

*ngIf="products\$ | async as products"



RxJS Checklist: Error Handling



Procedural approach: errors handled in the subscribe

```
this.productService.getProducts()
   .subscribe({
    next: products => this.products = products,
    error: err => this.errorMessage = err
});
```

Declarative approach: errors handled in the pipeline

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

RxJS Checklist: Error Handling Features



```
catchError: catches an error
```

```
catchError(this.handleError)
```

throwError: Throws an error along the subscriber chain

```
throwError(() => new Error('Could not retrieve'));
```

EMPTY: Defines an empty Observable that completes

```
return EMPTY;
```



Coming up next...

Mapping returned data

