RxJS in Angular: Reactive Development

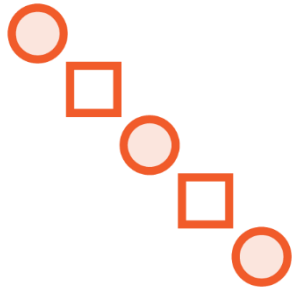
INTRODUCTION

Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR | MVP | GDE

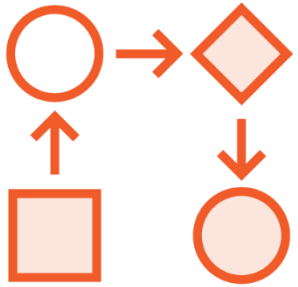
@deborahkurata | blogs.msmvps.com/deborahk/

Goals for This Course

Add clarity



Examine reactive patterns



Improve state management

Merge RxJS streams

Minimize subscriptions

Improve UI performance

Module Overview

What is RxJS?

How is RxJS used in Angular?

What is reactive development?

Getting the most from this course

Course outline

What Is RxJS?

Reactive Extensions for JavaScript

Reactive Extensions were originally developed by Microsoft as Rx.NET

RxJava, RxPy, Rx.rb, RxJS

"RxJS is a library for composing asynchronous and event-based programs by using observable sequences."

https://rxjs.dev/guide/overview

Manage Data as It Flows through Time

Collect

Pipe through a set of operations

- Transform
- Filter
- Process

Combine

Cache

"RxJS is a library for composing asynchronous and event-based programs by using observable sequences."

https://rxjs.dev/guide/overview

"RxJS is a library for composing asynchronous and event-based programs by using observable sequences."

https://rxjs.dev/guide/overview

RxJS is a library for composing observable streams and optionally processing each item in the stream using an extensive set of operators.

Why RxJS Instead Of...

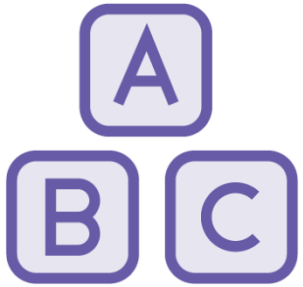
Callbacks

Promises

`async/await`

Why RxJS?

One technique to rule them all



Compositional

Watchful

Lazy

Handles errors

Cancellable

Angular uses RxJS.

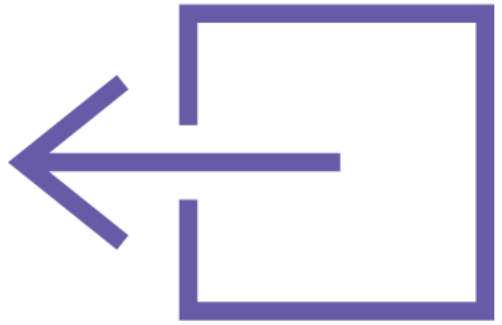
How Is RxJS Used in Angular?

Routing

```
this.route.paramMap
this.route.data
this.router.events
```

Reactive Forms

```
this.productForm.valueChanges
```



HttpClient

```
getProducts(): Observable<Product[]> {
  return this.http.get<Product[]>(this.url);
}
```

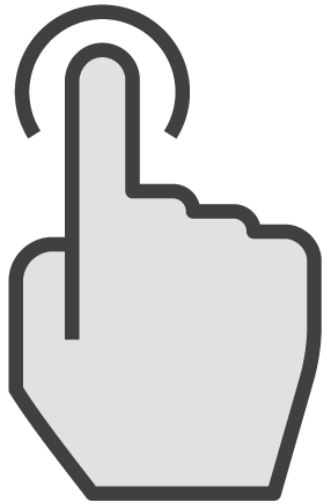
"... a declarative programming paradigm concerned with data streams and the propagation of change."

Wikipedia
https://en.wikipedia.org/wiki/Reactive_programming 4/15/19

"The essence of functional reactive programming is to specify the dynamic behavior of a value completely at the time of declaration."

Heinrich Apfelmus
https://apfelmus.nfshost.com/blog/2011/03/28-essence-frp.html

Reactive Development



Quick to react to user interactions

Resilient to failure

Reactive to state changes

Prerequisites

Required

- Components
- Templates
- Services
- Observables / HttpClient

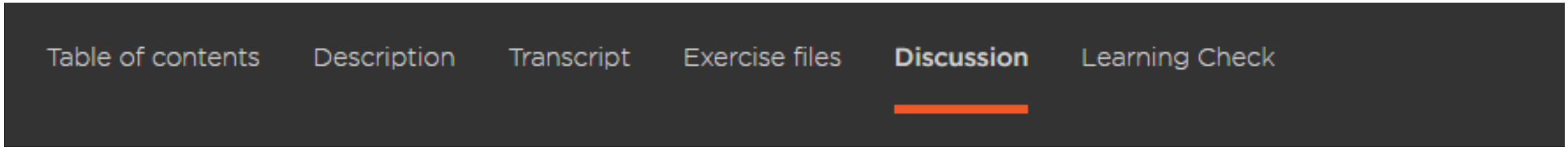
Suggested

- Angular: Getting Started
- Angular: First Look

Not Required

- Extensive knowledge of RxJS

Thoughts? Comments? Questions?



@deborahkurata

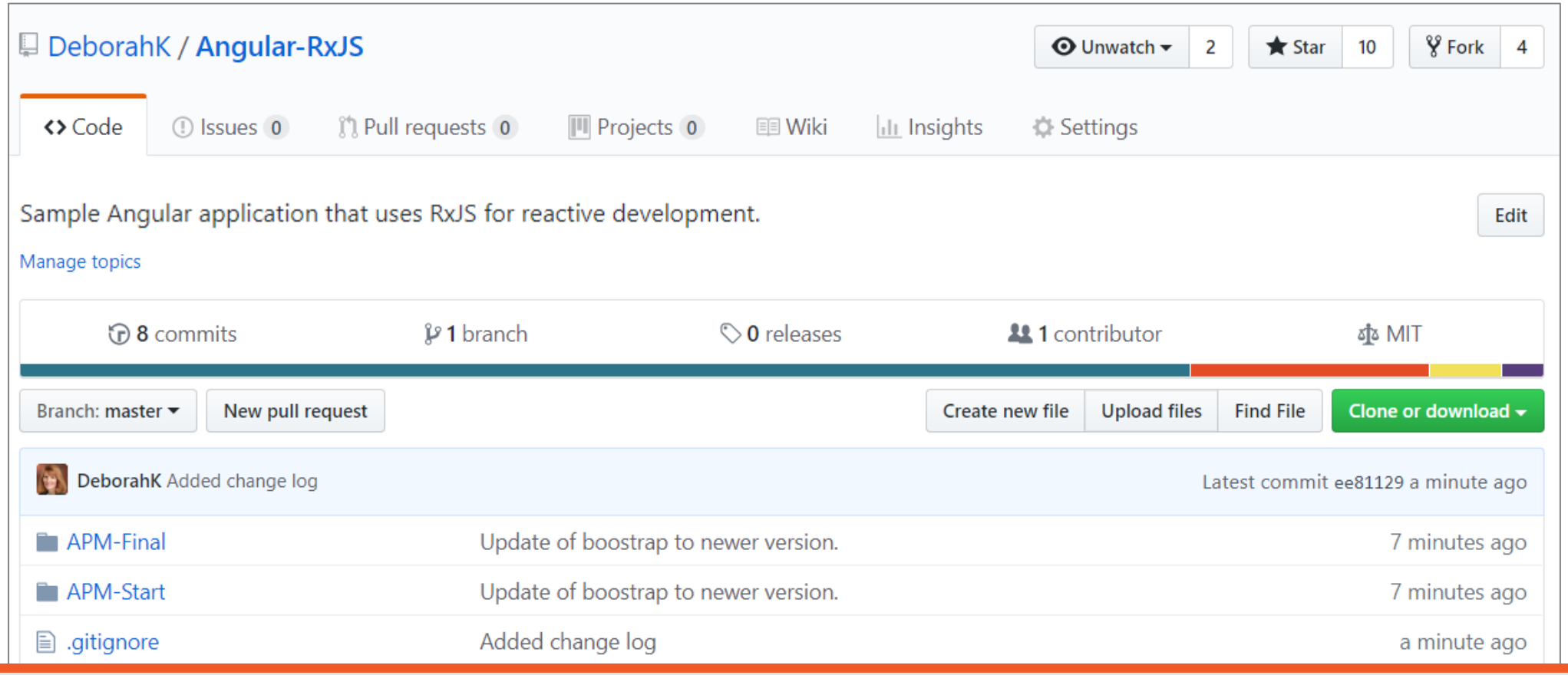
Checklist

Review module concepts

Code along assistance

Revisit as you build

GitHub Repository



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

Course Outline

RxJS terms and syntax

RxJS operators

Going reactive

Mapping returned data

Combining streams

Reacting to actions

Caching Observables

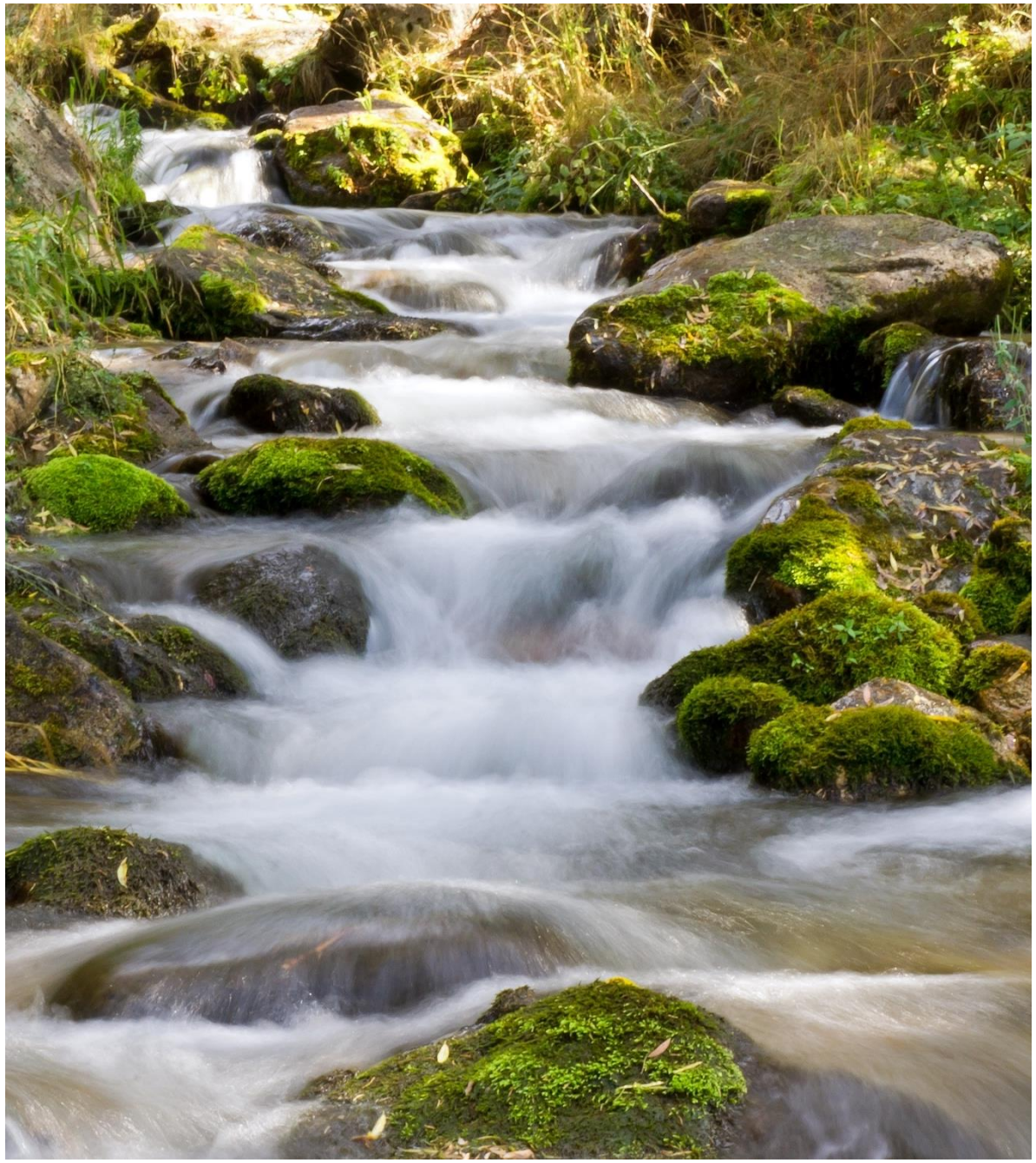
Higher-order mapping operators

Combining all the streams

RxJS Terms and Syntax

Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR | MVP | GDE

@deborahkurata | blogs.msmvps.com/deborahk/



Processing Observable Streams

Start the stream

- Emit items into the stream

Items pass through a set of operations

As an observer

- Next item, process it
- Error occurred, handle it
- Complete, you're done

Stop the stream

Processing Observable Streams

Apple Factory	RxJS
Start the stream - Emits items	**Subscribe** - Emits items
Items pass through a set of operations	**Pipe through a set of operators**
As an observer - Next item, process it - Error occurred, handle it - Complete, you're done	**Observer** - next() - error() - complete()
Stop the stream	**Unsubscribe**

Module Overview

Observer/Subscriber

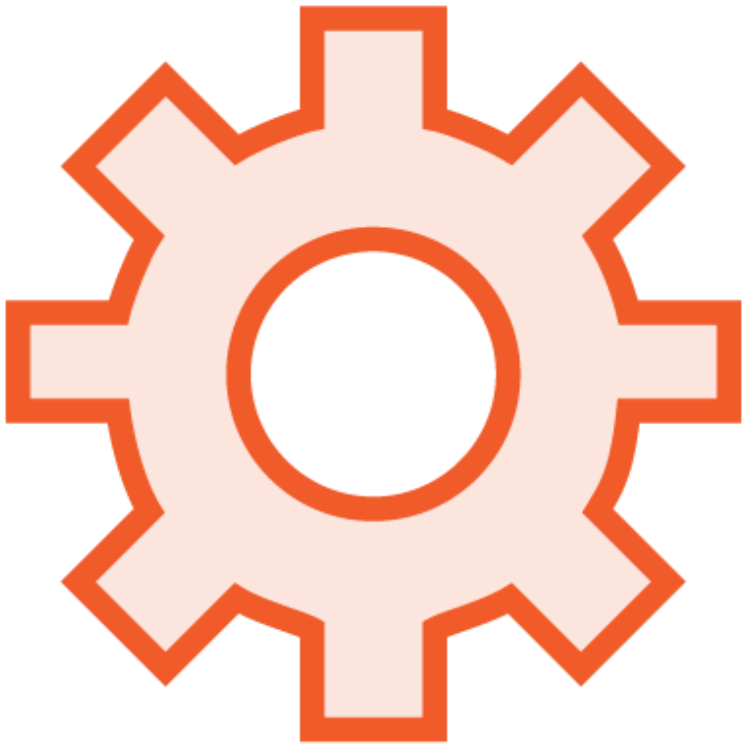
Observable stream (Observable)

Starting the Observable stream / Subscription

Stopping the Observable stream

Creation functions

RxJS Features



```
of

from
```

Observer

As an observer

- Next item, process it
- Error occurred, handle it
- Complete, you're done

Observer

As an observer

Next item, process it

Error occurred, handle it

Complete, you're done

Observer

next()

error()

complete()

Observes the stream and responds to its notifications

Observer

"Observer: is a collection of callbacks that knows how to listen to values delivered by the Observable."

"A JavaScript object that defines the handlers for the notifications you receive."

In RxJS, an Observer is also defined as an interface with next, error, and complete methods.

Observer

- next()
- error()
- complete()

Observes the stream and responds to its notifications

Subscriber

Subscriber

- next()
- error()
- complete()

Observer that can unsubscribe from an Observable

Observer

- next()
- error()
- complete()

Observes the stream and responds to its notifications

Observer

```
const observer = {
  next: apple => console.log(`Apple was emitted ${apple}`),
  error: err => console.log(`Error occurred: ${err}`),
  complete: () => console.log(`No more apples, go home`)
};
```

Observable Stream

Stream of apples moving on a conveyor

Observable Stream

Stream of apples moving on a conveyor

Any stream of data, optionally produced over time

- Numbers
- Strings
- Events
- Object literals
- Response returned from an HTTP request
- Other Observable streams

Observable Stream

Also called:
- An Observable sequence
- An Observable
- A stream

Observables can be synchronous or asynchronous

Observables can emit a finite or infinite number of values

Any stream of data, optionally produced over time

- Numbers
- Strings
- Events
- Object literals
- Response returned from an HTTP request
- Other Observable streams

Observable

```
const observer = {
  next: apple => console.log(`Apple was emitted ${apple}`),
  error: err => console.log(`Error occurred: ${err}`),
  complete: () => console.log(`No more apples, go home`)
};
```

```
const appleStream = new Observable(appleObserver => {
  appleObserver.next('Apple 1');
  appleObserver.next('Apple 2');
  appleObserver.complete();
});
```

Subscription

Start the stream

- Emits items

Items pass through a set of operations

As an observer

- Next item, process it
- Error occurred, handle it
- Complete, you're done

Stop the stream

Subscription

Start the stream

Emits items

Items pass through a set of operations

As an observer

Next item, process it

Error occurred, handle it

Complete, you're done

Stop the stream

Call subscribe() on the Observable

MUST subscribe to start the Observable stream

Subscription

```
const observer = {
  next: apple => console.log(`Apple was emitted ${apple}`),
  error: err => console.log(`Error occurred: ${err}`),
  complete: () => console.log(`No more apples, go home`)
};
```

```
const appleStream = new Observable(appleObserver => {
  appleObserver.next('Apple 1');
  appleObserver.next('Apple 2');
  appleObserver.complete();
});
```

```
const sub = appleStream.subscribe(observer);
```

Subscription

```
const observer = {
  next: apple => console.log(`Apple was emitted ${apple}`),
  error: err => console.log(`Error occurred: ${err}`),
  complete: () => console.log(`No more apples, go home`)
};
```

```
const sub = appleStream.subscribe(observer);
```

```
const sub = appleStream.subscribe(
  apple => console.log(`Apple was emitted ${apple}`),
  err => console.log(`Error occurred: ${err}`),
  () => console.log(`No more apples, go home`)
);
```

Subscription

```
const appleStream = new Observable(appleObserver => {
  appleObserver.next('Apple 1');
  appleObserver.next('Apple 2');
  appleObserver.complete();
});
```

```
const sub = appleStream.subscribe(
  apple => console.log(`Apple was emitted ${apple}`),
  err => console.log(`Error occurred: ${err}`),
  () => console.log(`No more apples, go home`)
);
```

Stopping the Stream

Start the stream

Emit items

Items pass through a set of operations

As an observer

Next item, process it

Error occurred, handle it

Complete, you're done

Stop the stream

Stopping an Observable Stream

Technique	Completes?

Unsubscribe

```
const sub = appleStream.subscribe(observer);
```

```
sub.unsubscribe();
```

Properly unsubscribing from each Observable prevents memory leaks

Subscription

```
const appleStream = new Observable(appleObserver => {
  appleObserver.next('Apple 1');
  appleObserver.next('Apple 2');
  appleObserver.complete();
});
```

```
const sub = appleStream.subscribe(
  apple => console.log(`Apple was emitted ${apple}`),
  err => console.log(`Error occurred: ${err}`),
  () => console.log(`No more apples, go home`)
);
```

```
sub.unsubscribe();
```

In Angular, we often work with Observables that Angular creates for us.

Creating an Observable

```
const appleStream = new Observable(appleObserver => {
  appleObserver.next('Apple 1');
  appleObserver.next('Apple 2');
  appleObserver.complete();
});
```

```
const appleStream = of('Apple1', 'Apple2');
```

```
const appleStream = from(['Apple1', 'Apple2']);
```

# of vs. from

```
const apples = ['Apple 1', 'Apple 2'];
```

```
of(apples);
// [Apple1, Apple2]
```

```
from(apples);
// Apple1 Apple2
```

```
of(...apples);
// Apple1 Apple2
```

Creating an Observable

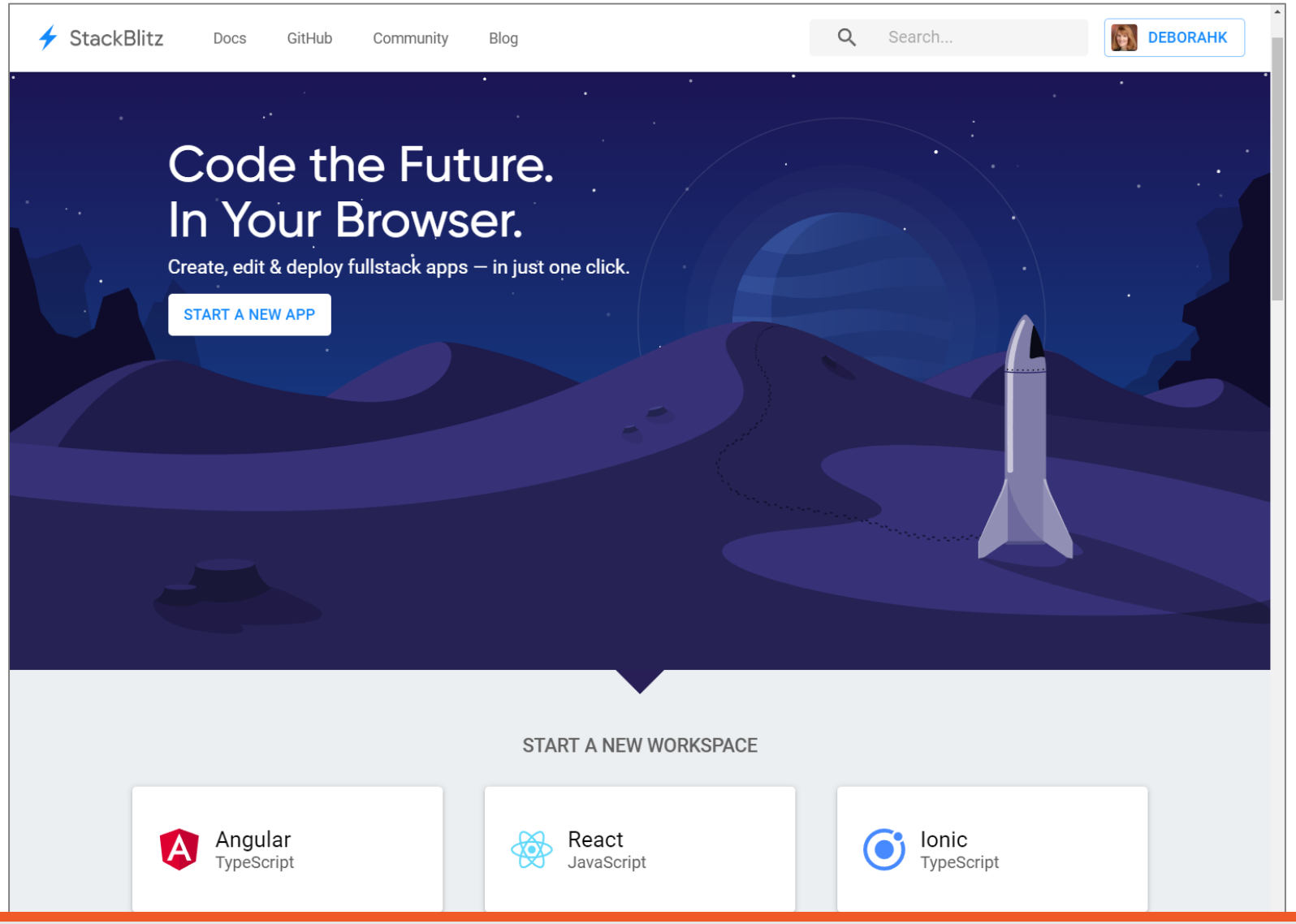
```
@ViewChild('para') par: ElementRef;

ngAfterViewInit() {
  const parStream = fromEvent(this.par.nativeElement, 'click')
                              .subscribe(console.log)
}
```

```
const num = interval(1000).subscribe(console.log);
```

Demo

Creation functions:

- `of`
- `from`



https://stackblitz.com

Terms

Observable

- Any stream of data

Observer

- Observes the stream
- Methods to process notifications from the stream: next(), error(), complete()

Subscriber

- An Observer that can unsubscribe

Subscription

- Represents the execution of an Observable
- subscribe() returns a Subscription

Creating an Observable

Constructor

Creation functions

- of, from, fromEvent, interval, ...
- Create an Observable from anything

Returned from an Angular feature

- Forms: valueChanges
- Routing: paramMap
- HTTP: get
- ...

Starting an Observable

Call subscribe!

Pass in an Observer

- next(), error(), complete()

```
const sub = appleStream.subscribe(
  apple => console.log(`Emitted: ${apple}`),
  err => console.log(`Error: ${err}`),
  () => console.log(`No more apples, go home`)
);
```

Stopping an Observable

Call complete() on the Observer

Use a creation function that completes

- of, from, ...

Use an operator that completes

- take, ...

Throw an error

Call unsubscribe() on the Subscription

```
sub.unsubscribe();
```

RxJS Operators

Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR | MVP | GDE

@deborahkurata | blogs.msmvps.com/deborahk/

RxJS Operators

Start the stream

Emits items

Items pass through a set of operations

As an observer

Next item, process it

Error occurred, handle it

Complete, you're done

Stop the stream

RxJS Operators

Start the stream

- Emits items

Items pass through a set of operations

As an observer

- Next item, process it
- Error occurred, handle it
- Complete, you're done

Stop the stream

Items are piped through a set of operators

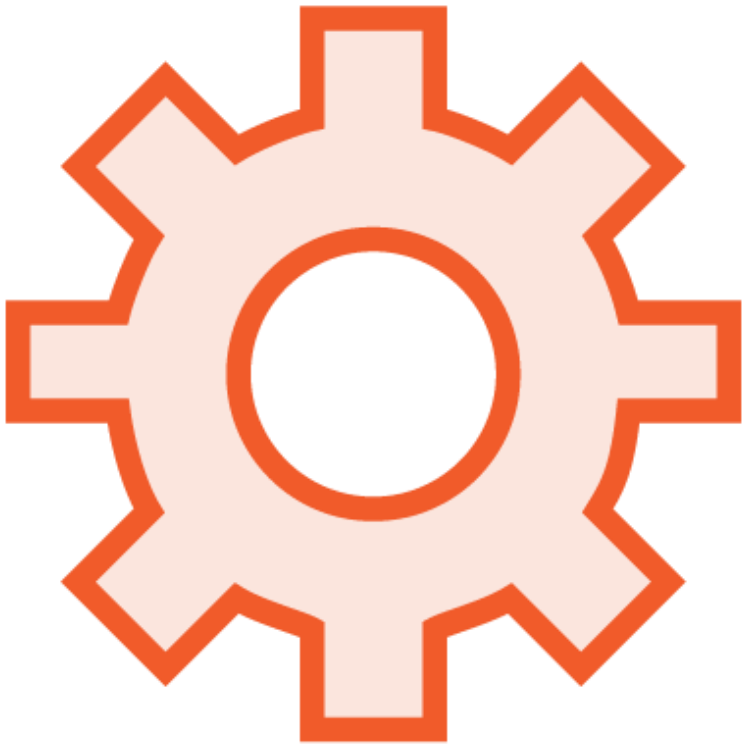
- Fashioned after .NET LINQ operators
- Similar to array methods such as filter and map

Module Overview

RxJS Operators

- Overview
- Documentation
- Examples

RxJS Features

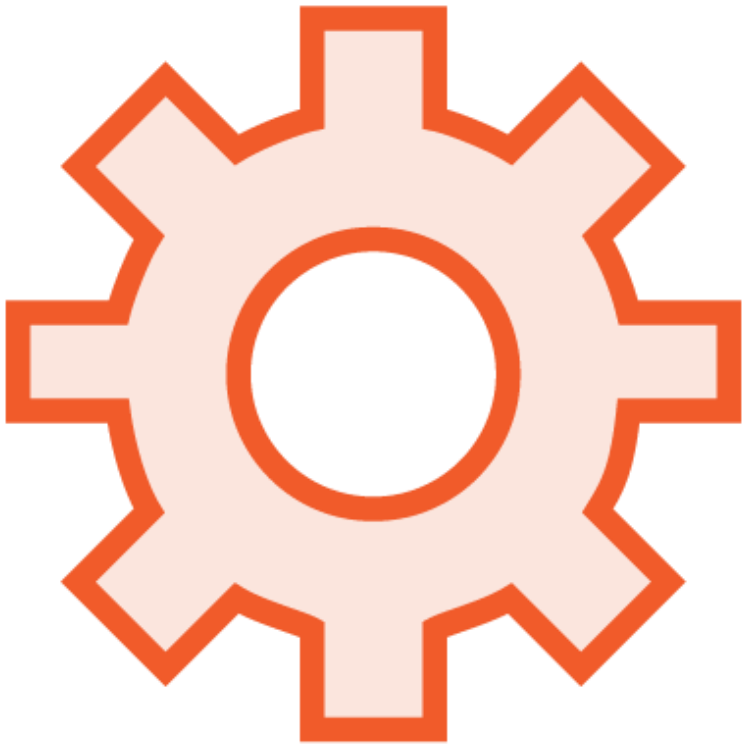


`map`

`tap`

`take`

RxJS Operators



An operator is a function

Used to transform and manipulate items in an Observable stream

Apply operators in sequence using the Observable's pipe method

RxJS Operators

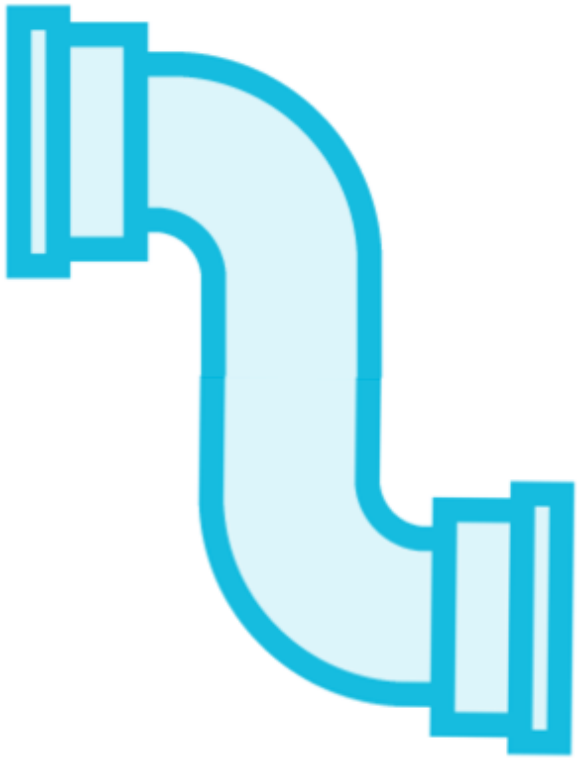
```
of(2, 4, 6)
  .pipe(
     map(item => item * 2),
     tap(item => console.log(item)),
     take(2)
  ).subscribe(console.log);
```

RxJS Operators

- audit
- auditTime
- buffer
- bufferCount
- bufferTime
- bufferToggle
- bufferWhen
- catchError
- combineAll
- ~~combineLatest (deprecated)~~
- ~~concat (deprecated)~~
- concatAll
- concatMap
- concatMapTo
- count
- debounce
- debounceTime
- defaultIfEmpty
- delay
- delayWhen
- dematerialize
- distinct
- distinctUntilChanged
- distinctUntilKeyChanged
- elementAt
- endWith
- every
- exhaust
- exhaustMap
- expand
- filter
- finalize
- find
- findIndex
- first
- flatMap
- groupBy
- ignoreElements
- isEmpty
- last
- map
- mapTo
- materialize
- max
- ~~merge (deprecated)~~
- mergeAll
- mergeMap
- mergeMapTo
- mergeScan
- min
- multicast
- observeOn
- onErrorResumeNext
- pairwise
- ~~partition (deprecated)~~
- pluck
- publish
- publishBehavior
- publishLast
- publishReplay
- ~~race (deprecated)~~
- reduce
- refCount
- repeat
- repeatWhen
- retry
- retryWhen
- sample
- sampleTime
- scan
- sequenceEqual
- share
- shareReplay
- single
- skip
- skipLast
- skipUntil
- skipWhile
- startWith
- subscribeOn
- switchAll
- switchMap
- switchMapTo
- take
- takeLast
- takeUntil
- takeWhile
- tap
- throttle
- throttleTime
- throwIfEmpty
- timeInterval
- timeout
- timeoutWith
- timestamp
- toArray
- window
- windowCount
- windowTime
- windowToggle
- windowWhen
- withLatestFrom
- ~~zip (deprecated)~~
- zipAll

https://rxjs.dev

RxJS Operator: `map`



Transforms each emitted item

```
map(item => item * 2)
```

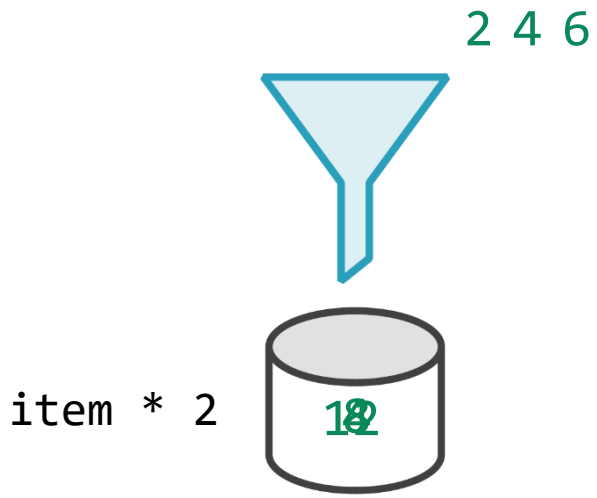
For each item in the source, one mapped item is emitted

Used for

- Making changes to each item

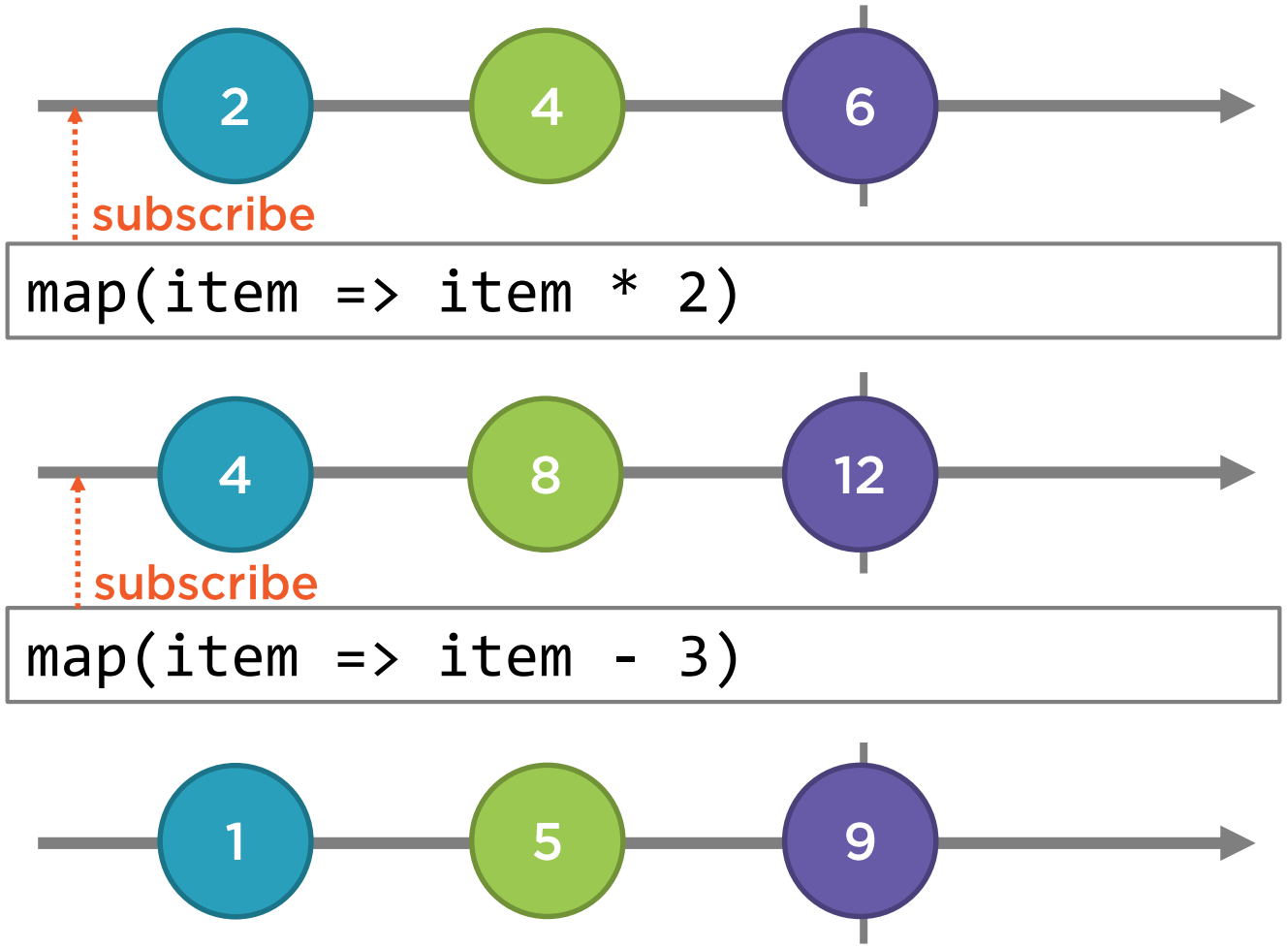
RxJS Operator: **map**

```
of(2, 4, 6)
    .pipe(
      map(item => item * 2)
    ).subscribe(console.log);
```

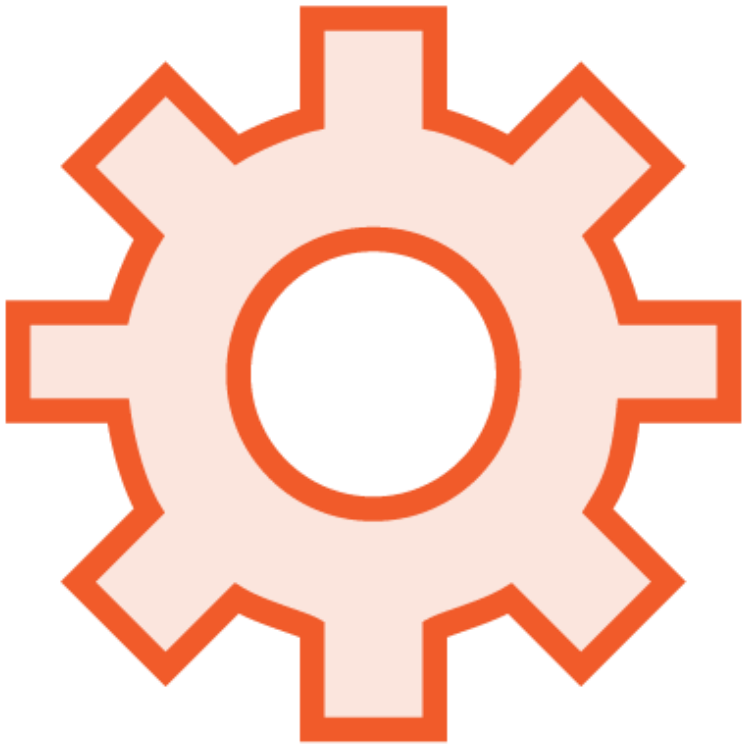


Marble Diagram: `map`

```
of(2, 4, 6)
    .pipe(
      map(item => item * 2),
      map(item => item - 3)
    ).subscribe(console.log);
```



RxJS Operator: `map`



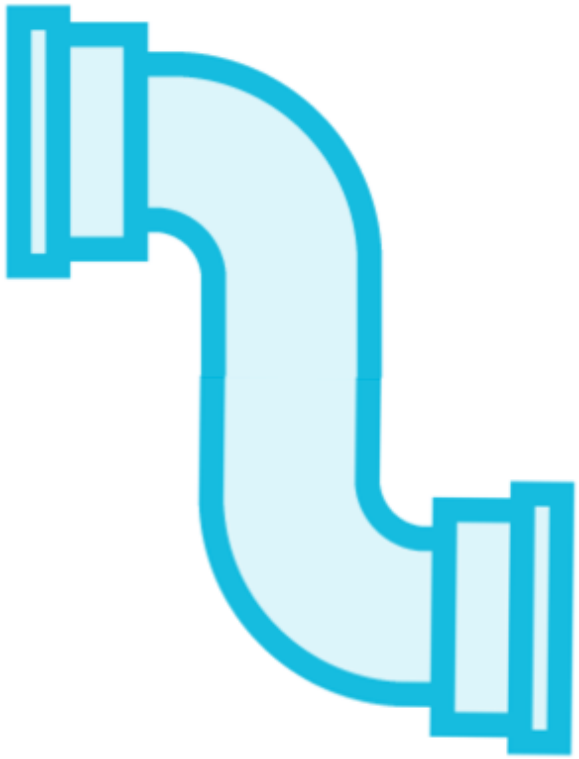
`map` **is a transformation operator**

- Takes in an input stream, subscribes
- Creates an output stream

When an item is emitted

- Item is transformed as specified by a provided function
- Item is emitted to the output stream

RxJS Operator: `tap`



Taps into a stream without modifying it

```
tap(item => console.log(item))
```

Used for

- Debugging
- Performing actions outside of the flow of data

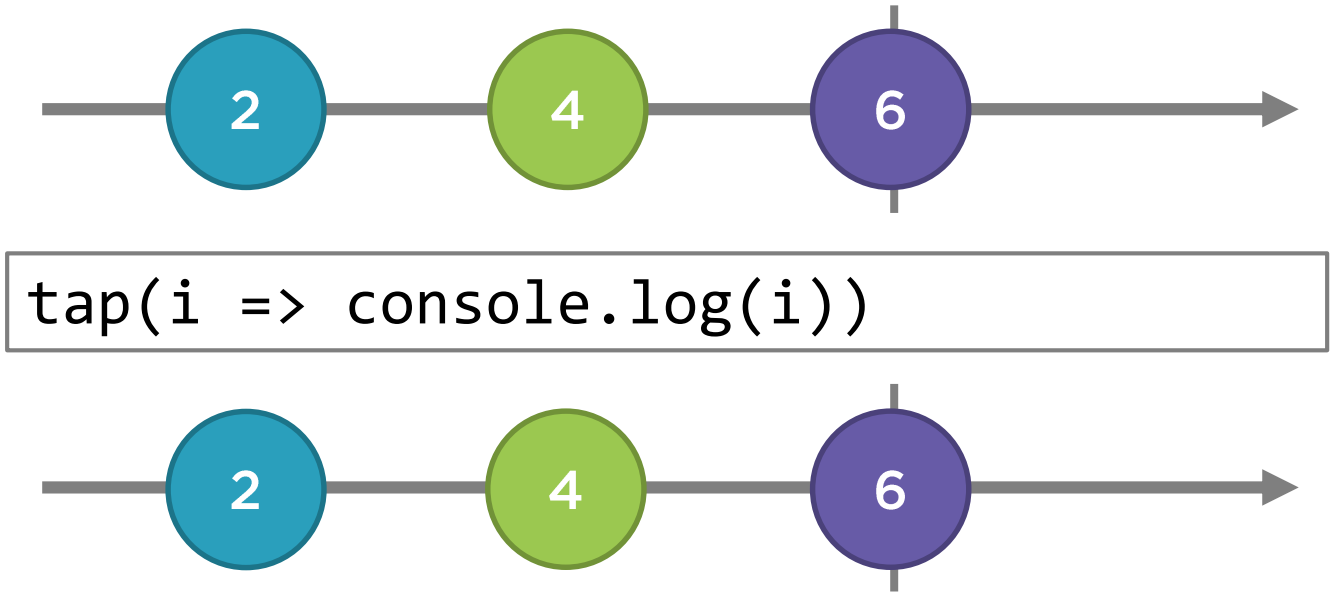
RxJS Operator: `tap`

```
of(2, 4, 6)
    .pipe(
      tap(item => console.log(item)),
      map(item => item * 2),
      tap(item => console.log(item)),
      map(item => item - 3),
      tap(item => console.log(item))
    ).subscribe();
```

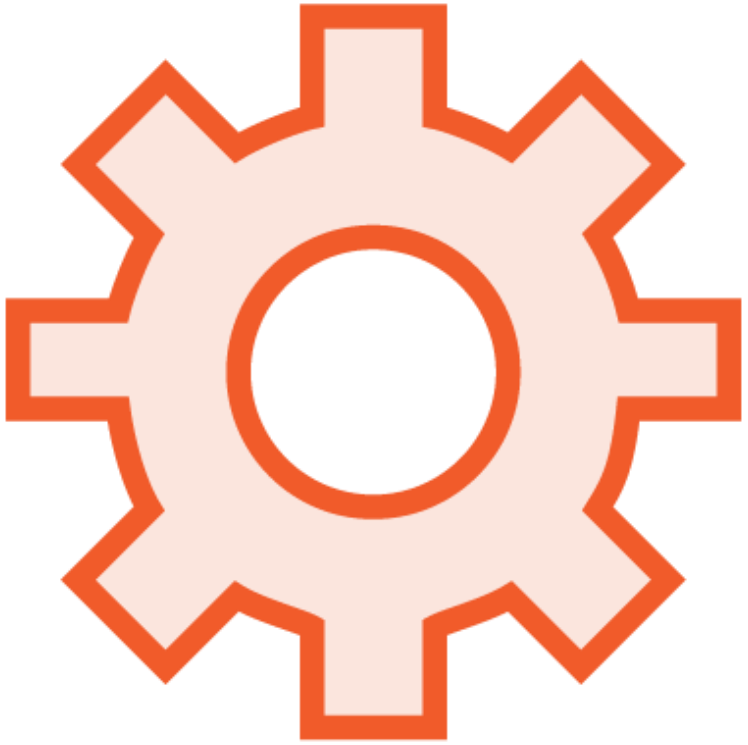
2
4
1
4
8
5
6
12
9

Marble Diagram: `tap`

```
of(2, 4, 6)
    .pipe(
      tap(i => console.log(i))
    ).subscribe(console.log);
```



RxJS Operator: `tap`



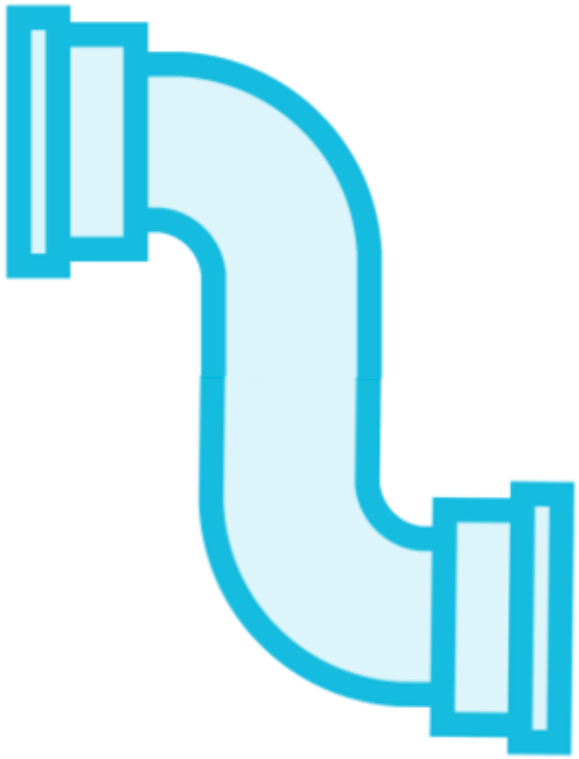
`tap` is a utility operator

- Takes in an input stream, subscribes
- Creates an output stream

When an item is emitted

- Performs a side effect as specified by a provided function
- Item is emitted to the output stream

RxJS Operator: `take`



Emits a specified number of items

```
take(2)
```

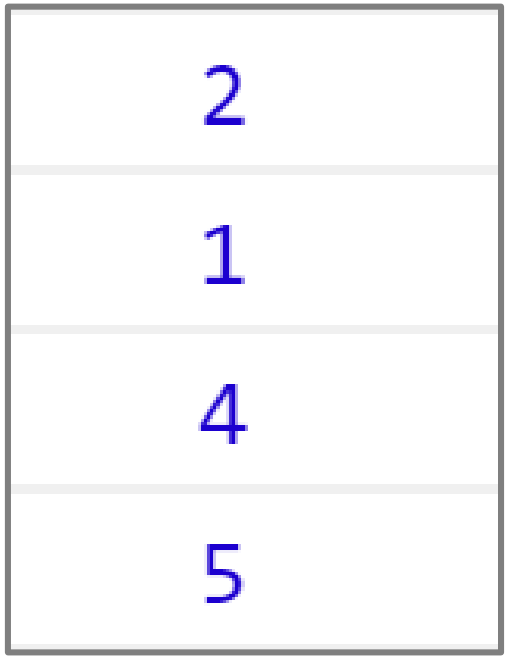
Used for

- Taking a specified number of items
- Limiting unlimited streams

RxJS Operator: take

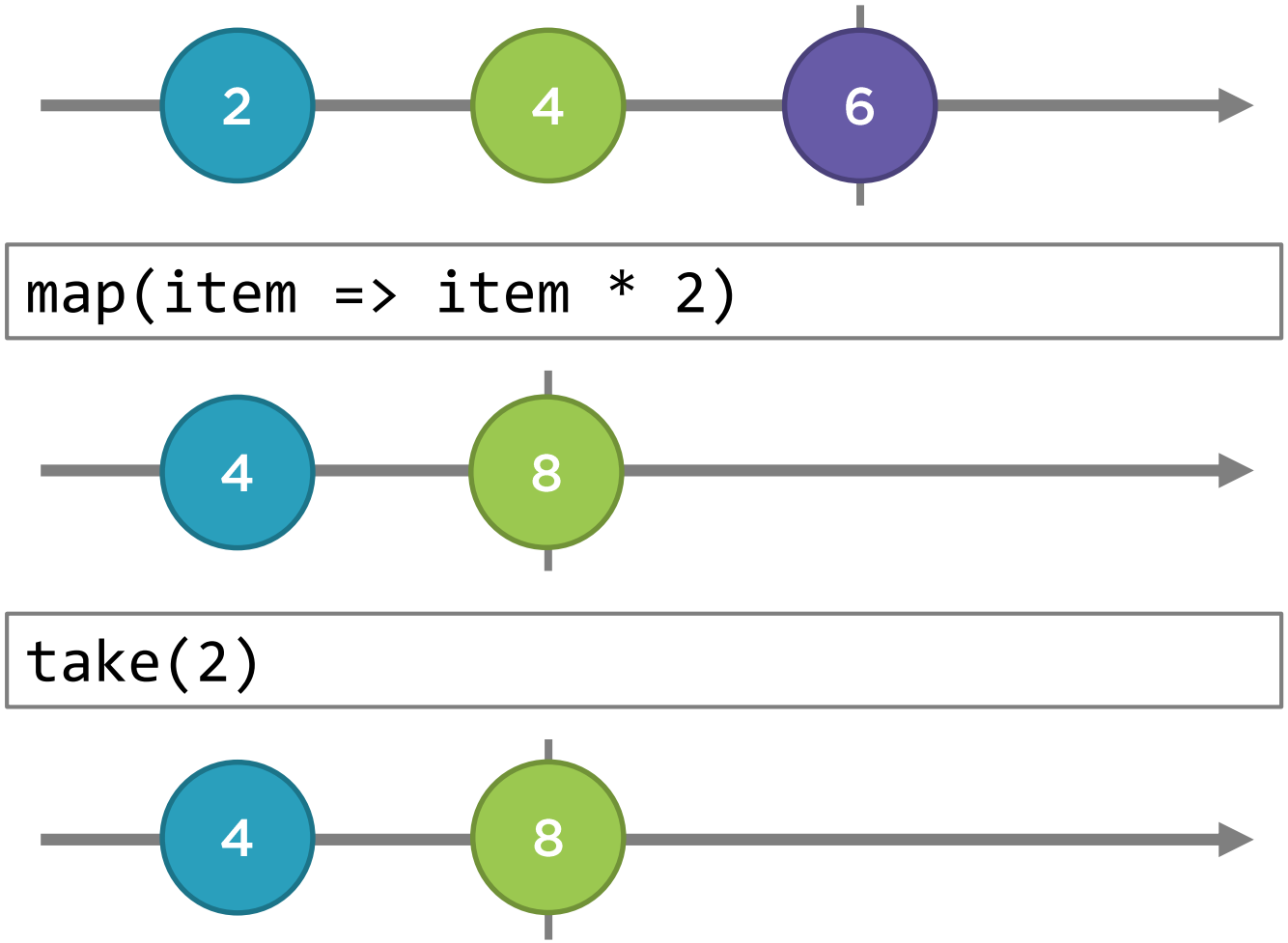
```
of(2, 4, 6)
    .pipe(
      take(2)
    ).subscribe(console.log);  // 2 4
```

```
of(2, 4, 6)
    .pipe(
      tap(item => console.log(item)),
      map(item => item * 2),
      take(2),
      map(item => item - 3),
      tap(item => console.log(item))
    ).subscribe();
```

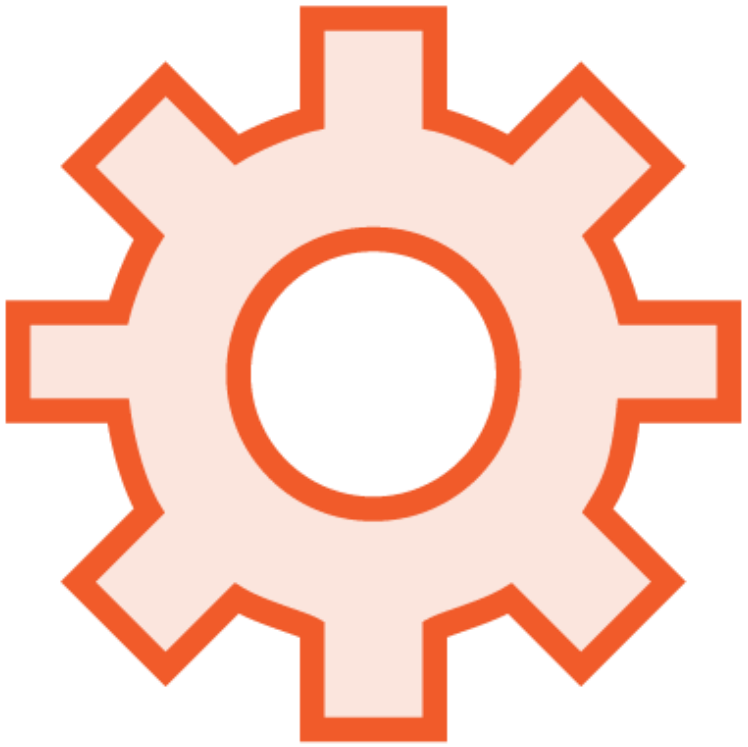


Marble Diagram: `map` and `take`

```
of(2, 4, 6)
    .pipe(
      map(item => item * 2),
      take(2)
    ).subscribe(console.log);
```



RxJS Operator: take



take **is a filtering operator**

- Takes in an input stream, subscribes
- Creates an output stream

When an item is emitted

- Counts the item
  - If <= specified number, emits item to the output stream
  - When it equals the specified number, it completes

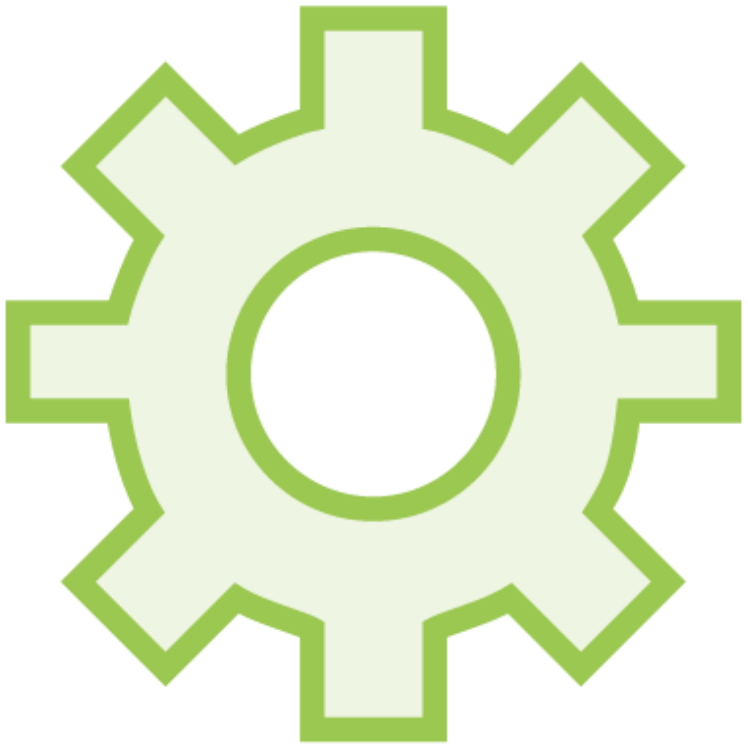
Only emits the defined number of items

Demo

RxJS Operators:

- `map`
- `tap`
- `take`

Operators

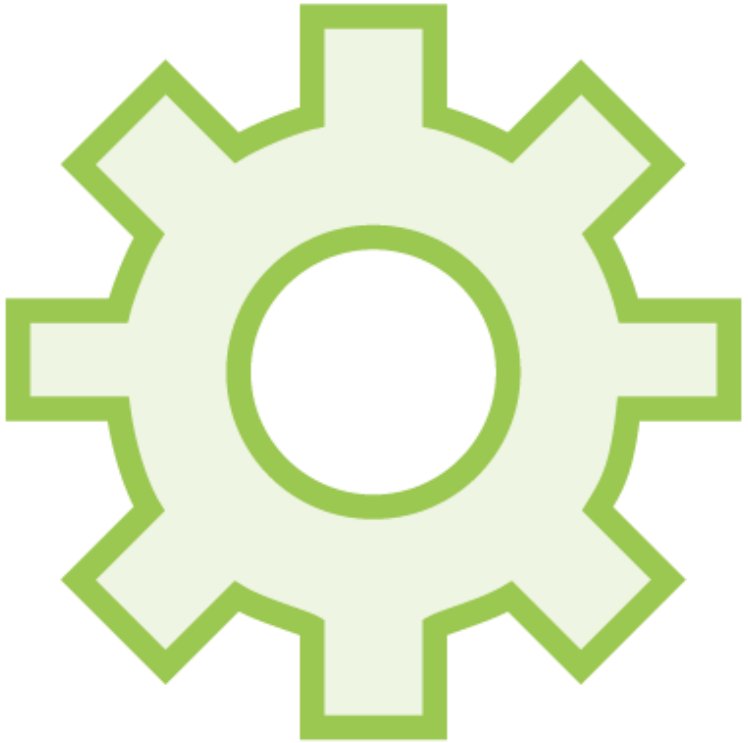


Use the Observable pipe method ...

... To pipe emitted items through a sequence of operators

```
from([20, 15, 10, 5]).pipe(
  tap(item => console.log(item)),
  take(3),
  map(item => item * 2),
  map(item => item - 10),
  map(item => {
    if (item === 0) {
      throw new Error('zero detected');
    }
    return item;
  })
);
```

RxJS Features



map: Transforms each emitted item

```
map(item => item * 2)
```

tap: Taps into the stream without modifying it

```
tap(item => console.log(item))
```

take: Emits the specified number of items and completes

```
take(2)
```

Reacting to Actions

Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR | MVP | GDE

@deborahkurata | blogs.msmvps.com/deborahk/

Acme Product Management Home Product List Product List (Alternate UI)

Product List

- Display All -

- Display All -
Garden
Toolbox
Gaming

Add Product

	Code	Category	Price	In Stock
[illegible]	GDN-0011	Garden	$29.92	15
Garden Cart	GDN-0023	Garden	$49.49	2
Hammer	TBX-0048	Toolbox	$13.35	8
Saw	TBX-0022	Toolbox	$17.33	6
Video Game Controller	GMG-0042	Gaming	$53.93	12

Module Overview

Filtering a stream

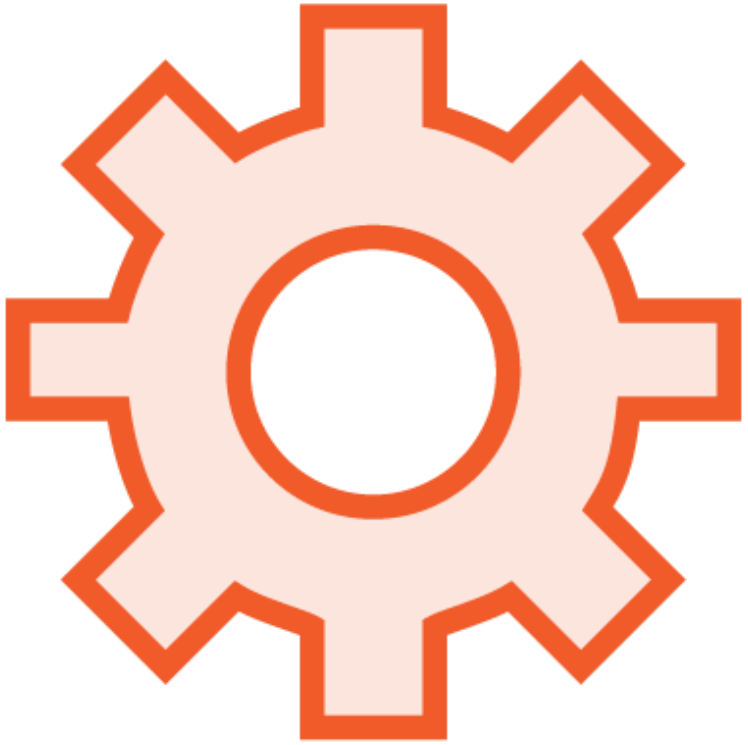
Data stream vs. action stream

Subject and BehaviorSubject

Reacting to actions

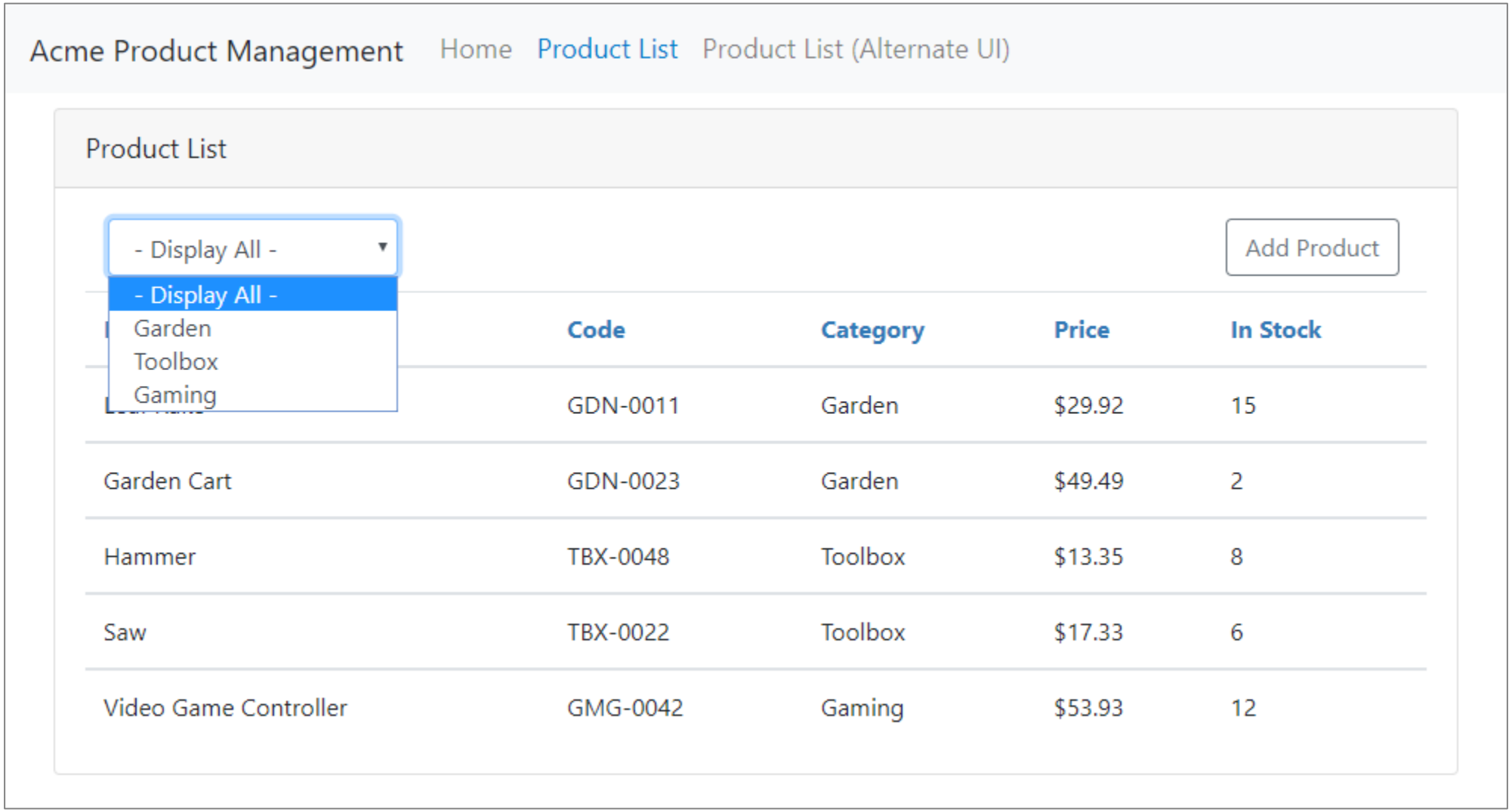
Starting with an initial value

RxJS Features

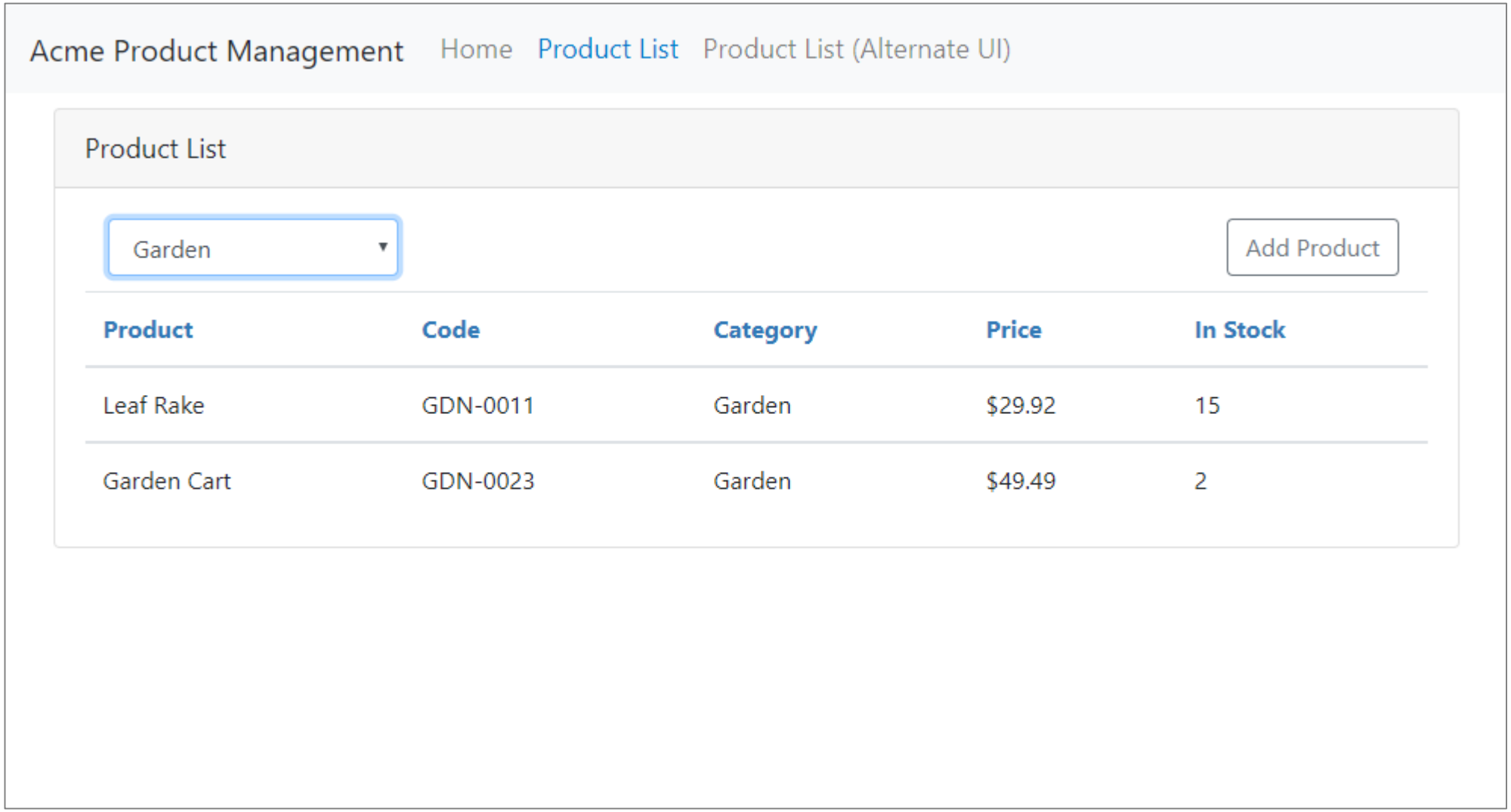


- `filter`
- `startWith`
- `Subject`
- `BehaviorSubject`

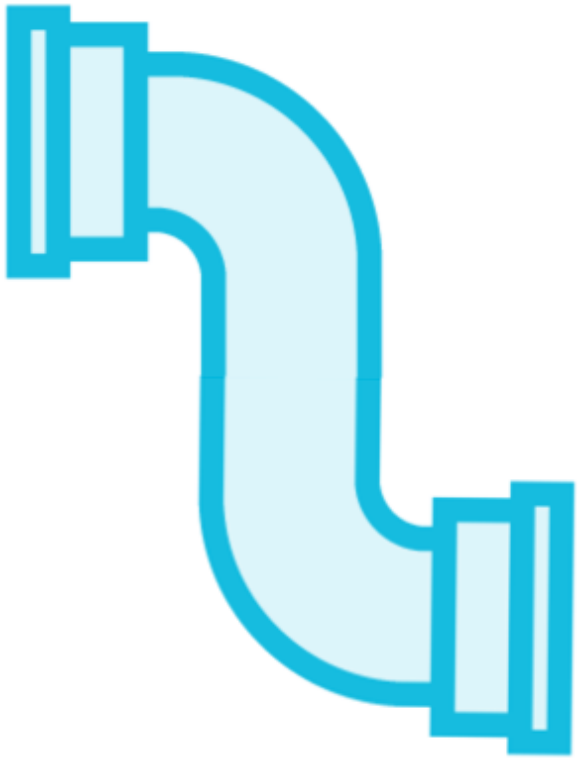
Filtering a Stream



Filtering a Stream



RxJS Operator: `filter`



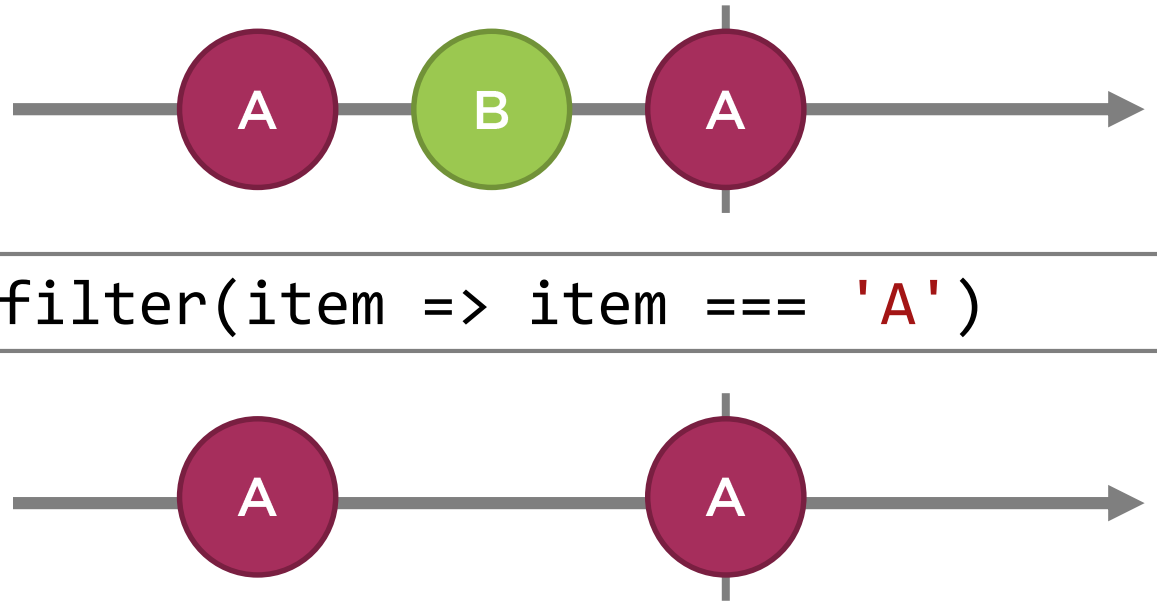
Filters to the items that match criteria specified in a provided function

```
filter(item => item === 'Apple')
```

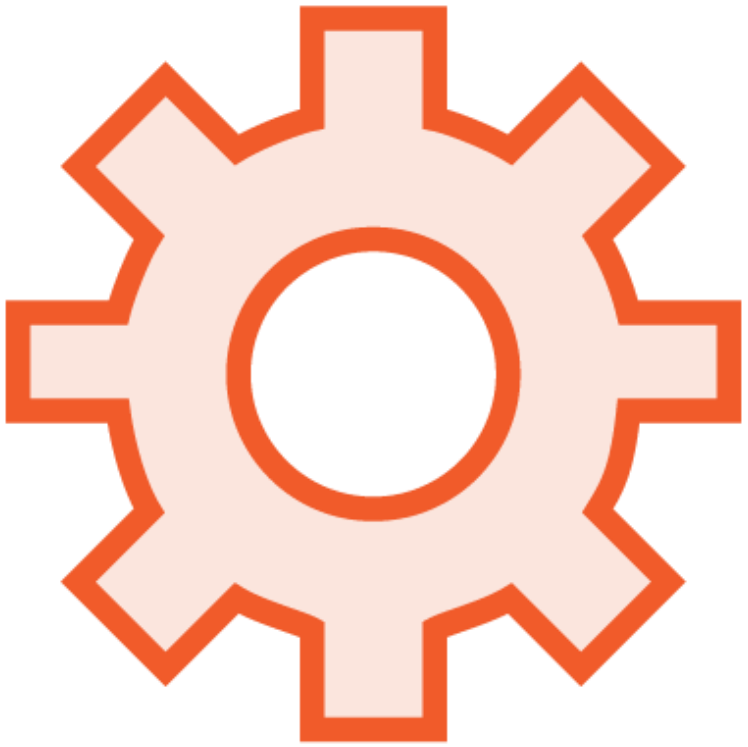
Similar to the array filter method

Marble Diagram: `filter`

```
of('A', 'B', 'A')
  .pipe(
    filter(item => item === 'A'),
  );
```



RxJS Operator: `filter`



`filter` is a transformation operator

- Takes in an input stream, subscribes
- Creates an output stream

When a source item is emitted

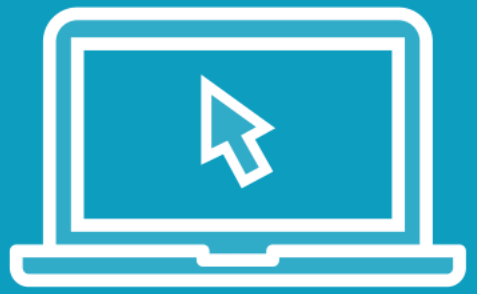
- Item is evaluated as specified by the provided function
- If the evaluation returns true, item is emitted to the output stream

Demo

Filtering a stream: Demo I

- Hard-coded category

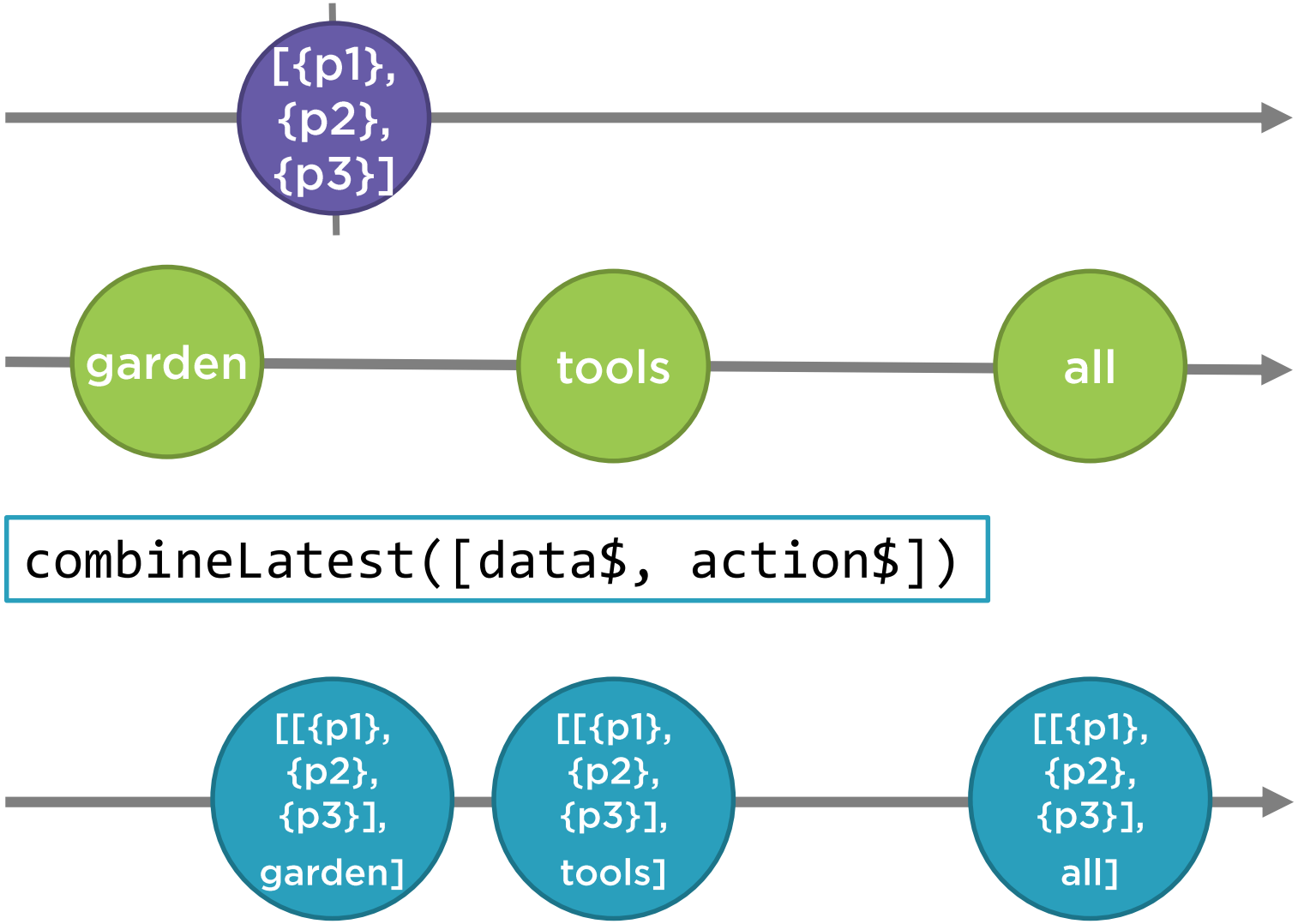
Demo



Filtering a stream: Demo II

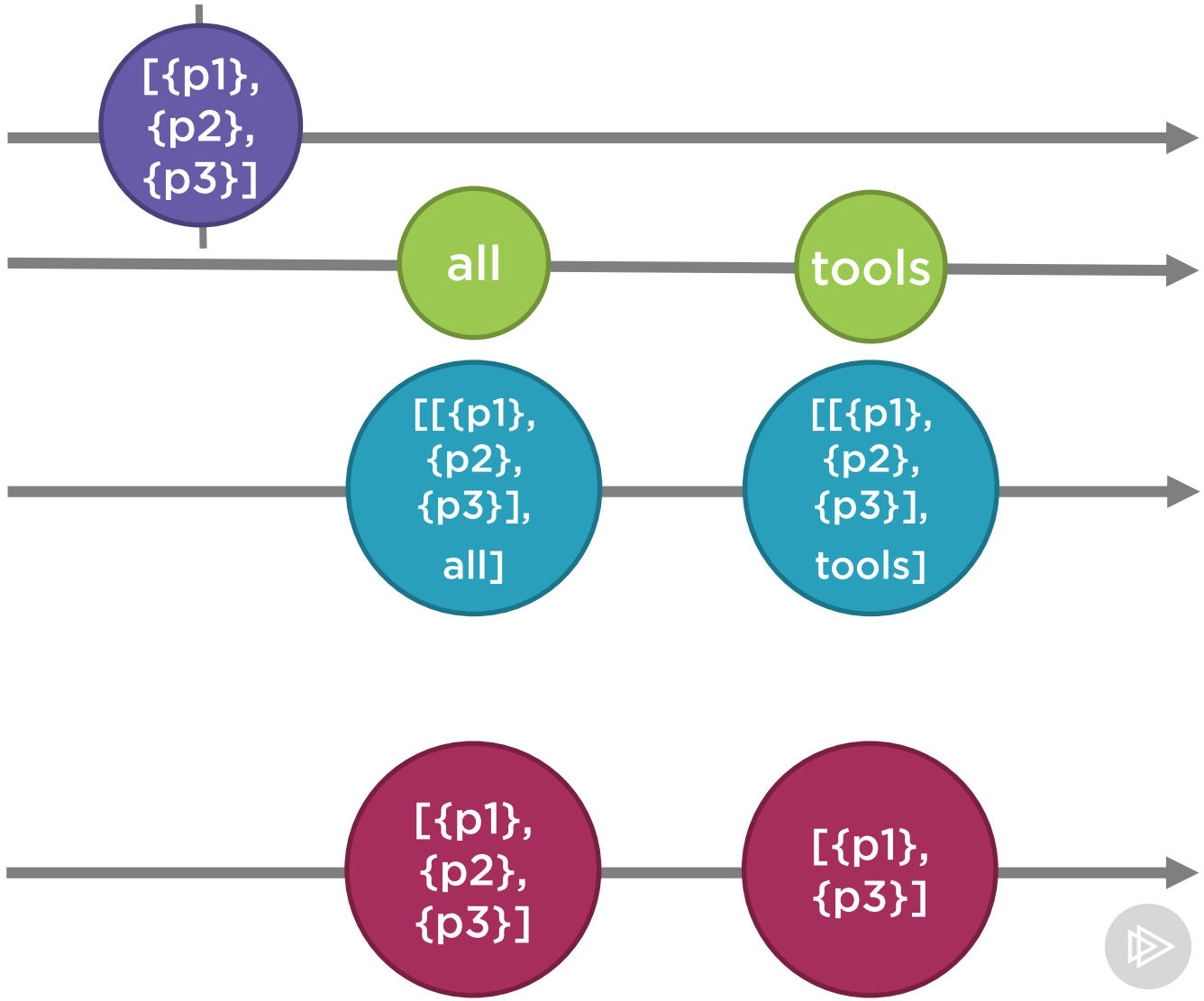
- Dropdown list of categories

Data Stream vs. Action Stream



Data Stream vs. Action Stream

```
products$ = combineLatest([

  this.productService.products$,

  this.action$



])
.pipe(
  map(([products, category]) =>
   products.filter(product =>
     product.category === category)
  )
);
```



Creating Streams

Data Stream

```
products$ = this.http.get<Product[]>(this.productsUrl)
```

Action Stream

```
action$ = ???
```

- Use a built-in stream
- `fromEvent`
- Subject/BehaviorSubject

Observable and Observer

Observable Stream

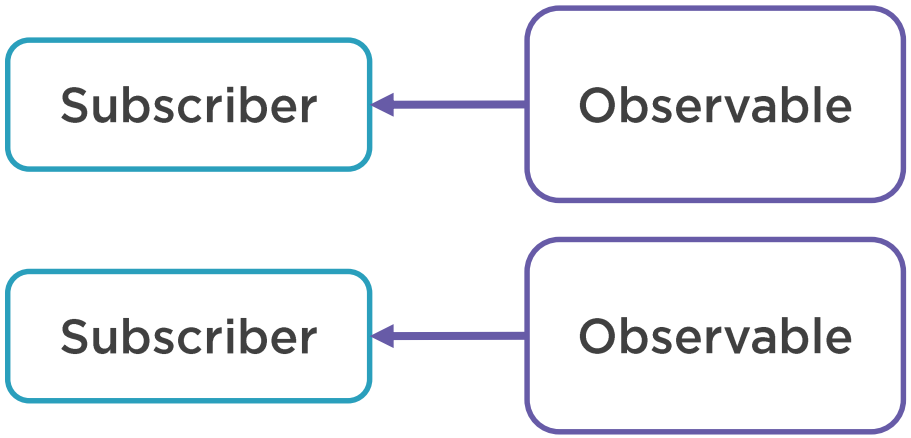
Observer:

next()

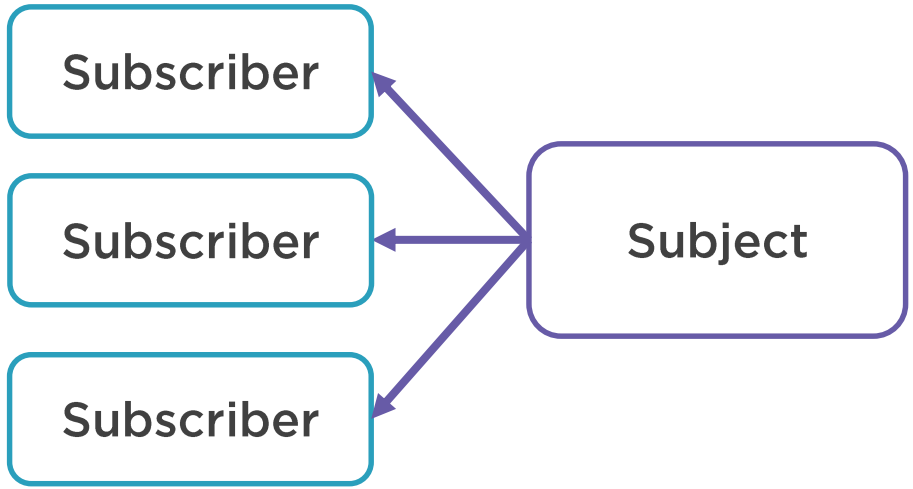
error()

complete()

Unicast vs. Multicast



Observable is unicast



Subject is multicast

Subject

```
private categorySelectedSubject = new Subject<number>();
categorySelectedAction$ = this.categorySelectedSubject.asObservable();
```

```
onSelected(categoryId): void {
  this.categorySelectedSubject.next(+categoryId);
}
```

```
products$ = combineLatest([
  this.productService.products$,
  this.categorySelectedAction$
])
.pipe(
  map(([products, categoryId]) =>
   products.filter(product =>
     categoryId ? product.categoryId === categoryId : true)
  )
);
```

BehaviorSubject

```
private categorySelectedSubject = new BehaviorSubject<number>(0);
categorySelectedAction$ = this.categorySelectedSubject.asObservable();
```

```
onSelected(categoryId): void {
  this.categorySelectedSubject.next(+categoryId);
}
```

```
products$ = combineLatest([
  this.productService.products$,
  this.categorySelectedAction$
])
.pipe(
  map(([products, categoryId]) =>
   products.filter(product =>
     categoryId ? product.categoryId === categoryId : true)
  )
);
```

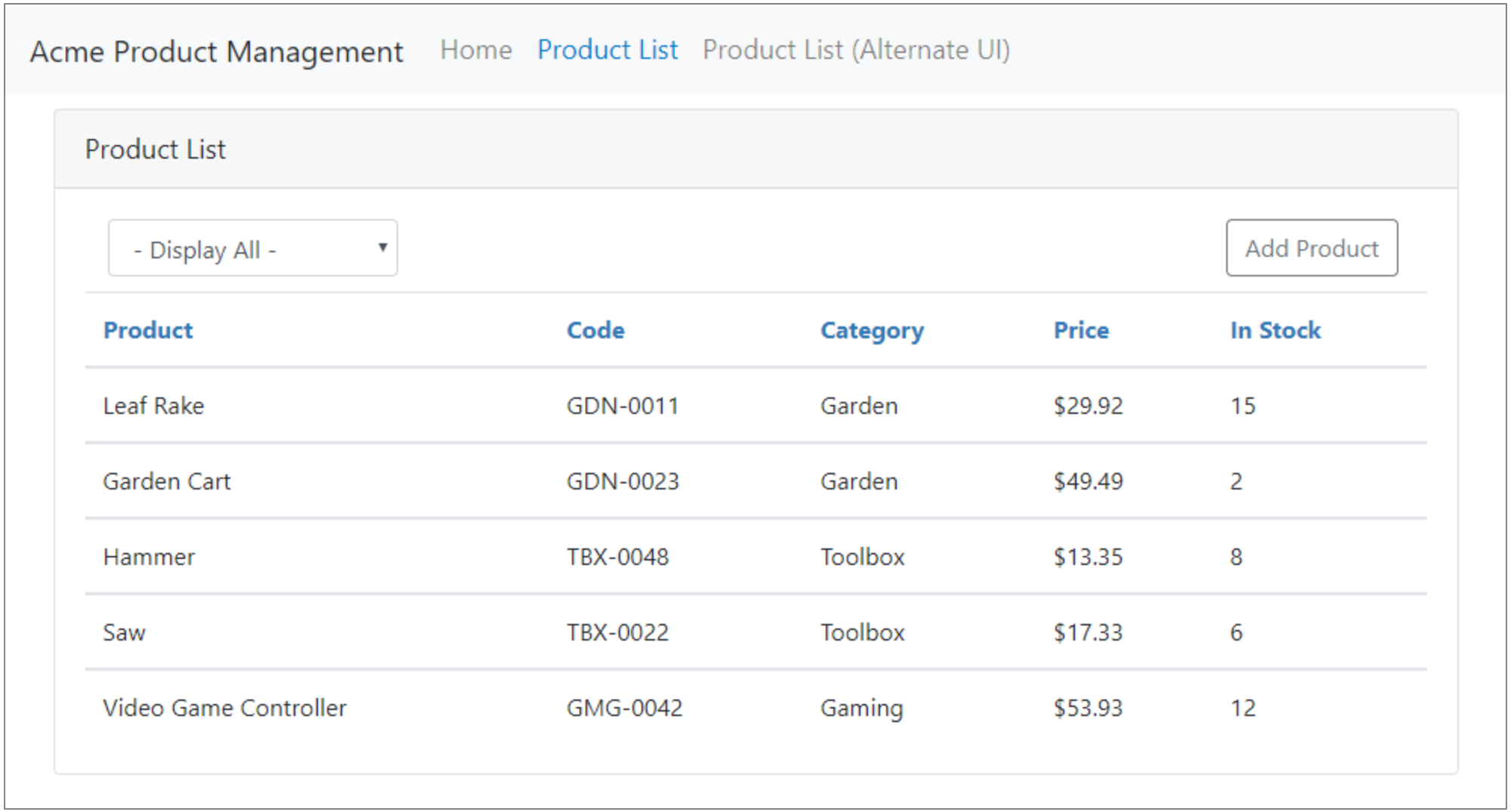
Reacting to Actions

Create an action stream (Subject/BehaviorSubject)

Combine the action stream and data stream

Emit a value to the action stream when an action occurs

Starting with an Initial Value

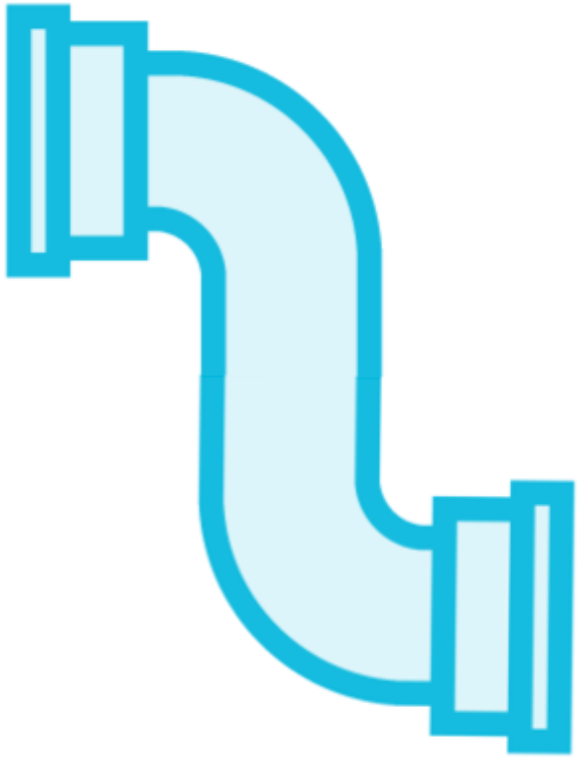


Starting with an Initial Value

```
this.categorySelectedAction$.pipe(startWith(0))
```

```
private categorySelectedSubject = new BehaviorSubject<number>(0);
categorySelectedAction$ = this.categorySelectedSubject.asObservable();
```

RxJS Operator: `startWith`

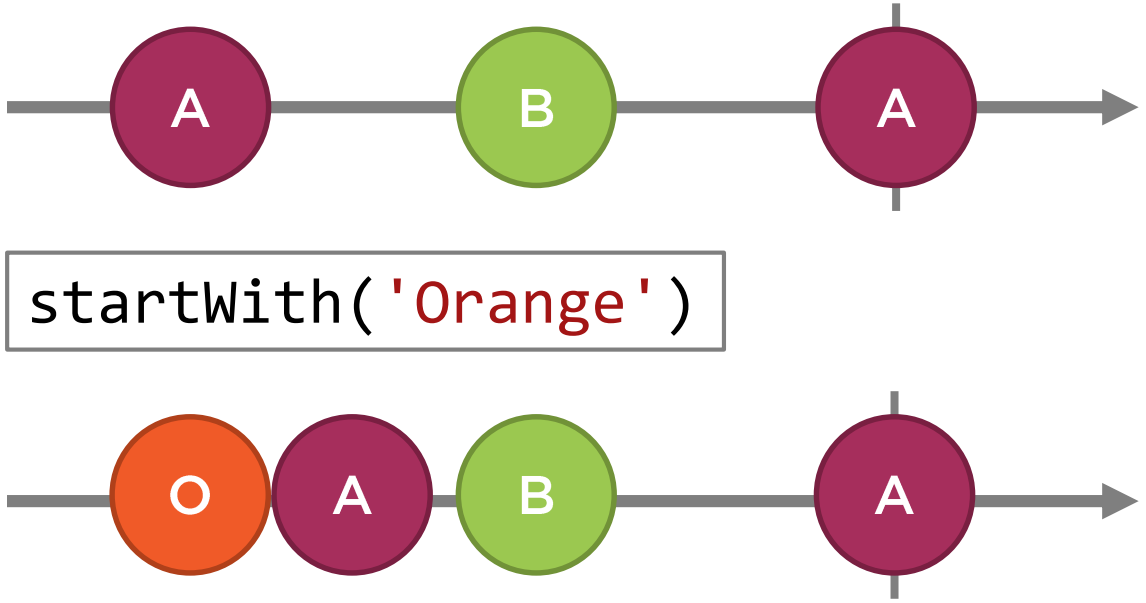


Provides an initial value

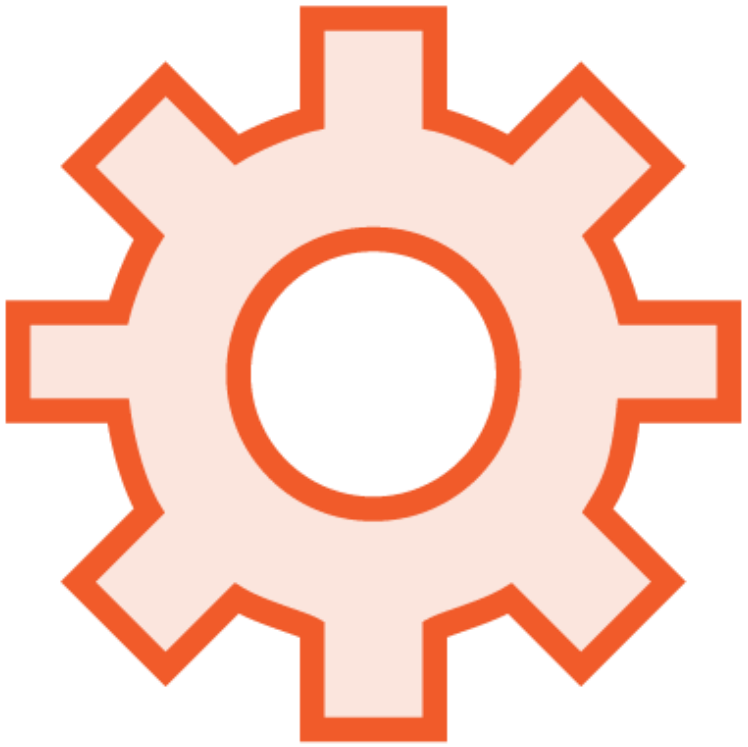
```
startWith('Orange')
```

Marble Diagram: `startWith`

```
of('A', 'B', 'A')
  .pipe(
    startWith('O'),
  );
```



RxJS Operator: `startWith`



`startWith` **is a combination operator**

- Takes in an input stream, subscribes
- Creates an output stream

When a source item is emitted

- If it's the first item, it emits the specified initial value(s), then ...
- It emits the item to the output stream

Initial value(s) must be the same type as the input Observable

Reacting to Actions

Create an action stream (Subject/BehaviorSubject)

```
selSubject = new Subject<number>();
selectedAction$ = this.selSubject.asObservable();
```

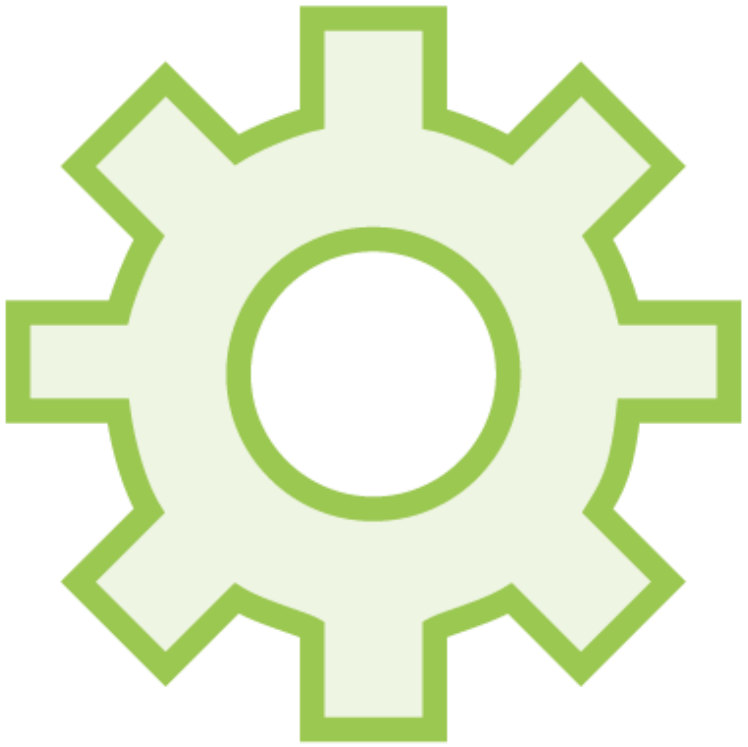
Combine the action and data streams

```
products$ = combineLatest([
  this.productService.products$,
  this.selectedAction$
]).pipe(...);
```

Emit a value to the action stream when an action occurs

```
onSelected(id): void {
  this.selSubject.next(+id);
}
```

RxJS Features



`filter`: Only emits items that match criteria

```
filter(item => item === 'Apple')
```

`startWith`: Defines an initial value emitted before the input stream values

```
startWith('Orange')
```

Subject/BehaviorSubject

`Subject`: **Special type of Observable that is both an Observable and an Observer**

```
selectedSubject = new Subject<number>();
```

`BehaviorSubject`: **Special type of Subject that emits an initial value**

```
selectedSubject = new BehaviorSubject<number>(0);
```

Reacting to Actions: Examples

Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR | MVP | GDE

@deborahkurata | blogs.msmvps.com/deborahk/

Acme Product Management Home Product List Product List (Alternate UI)

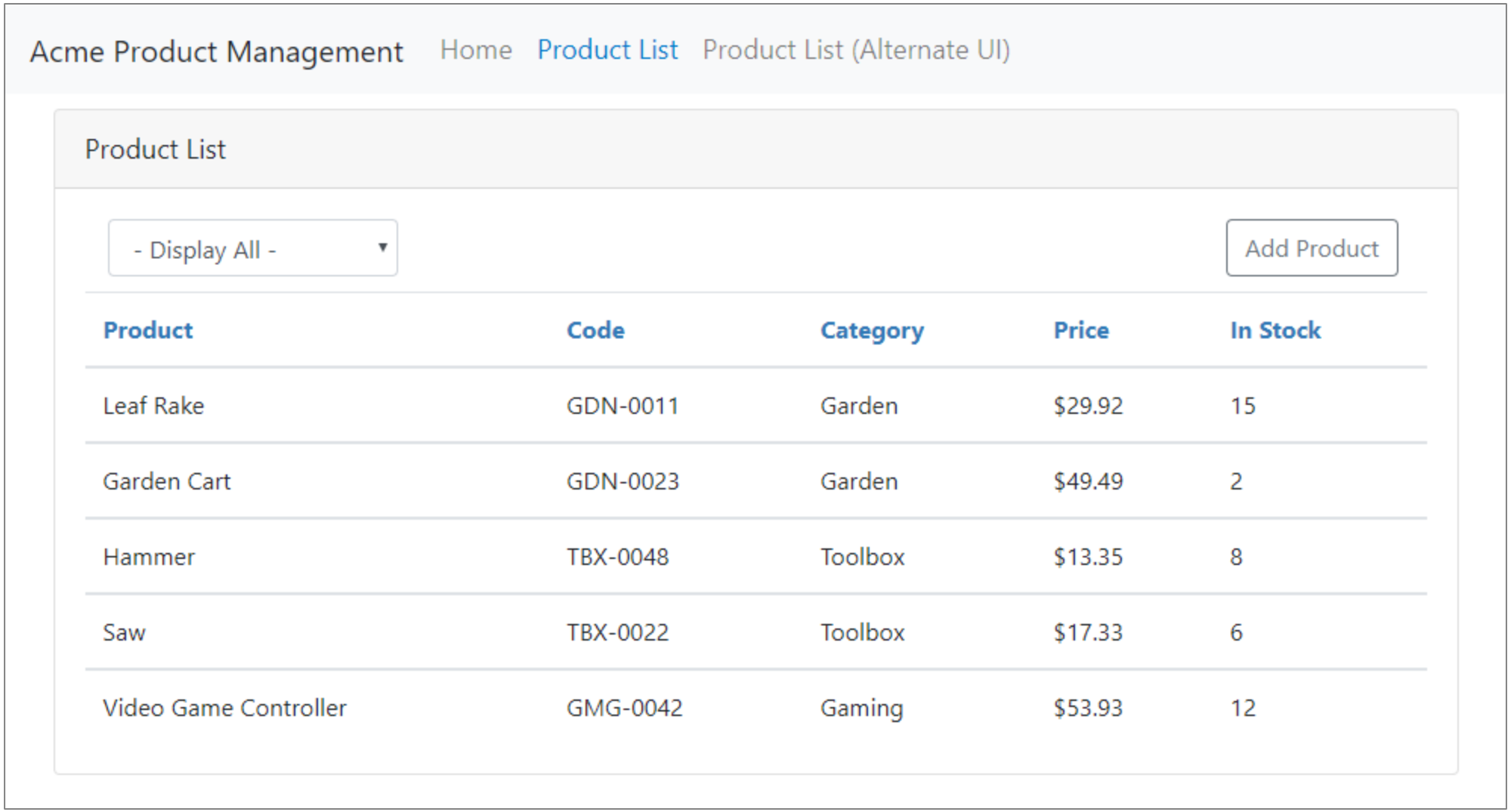
Products

- Leaf Rake (Garden)
- Garden Cart (Garden)
- Hammer (Toolbox)
- Saw (Toolbox)
- Video Game Controller (Gaming)

Product Detail for: Hammer

Name: Hammer
Code: TBX-0048
Category: Toolbox
Description: Curved claw steel hammer
Price: $13.35
In Stock: 8

Supplier	Cost	Minimum Quantity
Acme General Supply	$2.00	24
Acme Tool Supply	$4.00	12


Acme Product Management
Home
Product List
Product List (Alternate UI)
Product List
- Display All -
Add Product
Product
Code
Category
Price
In Stock
Leaf Rake
GDN-0011
Garden
$29.92
15
Garden Cart
GDN-0023
Garden
$49.49
2
Hammer
TBX-0048
Toolbox
$13.35
8
Saw
TBX-0022
Toolbox
$17.33
6
Video Game Controller
GMG-0042
Gaming
$53.93
12

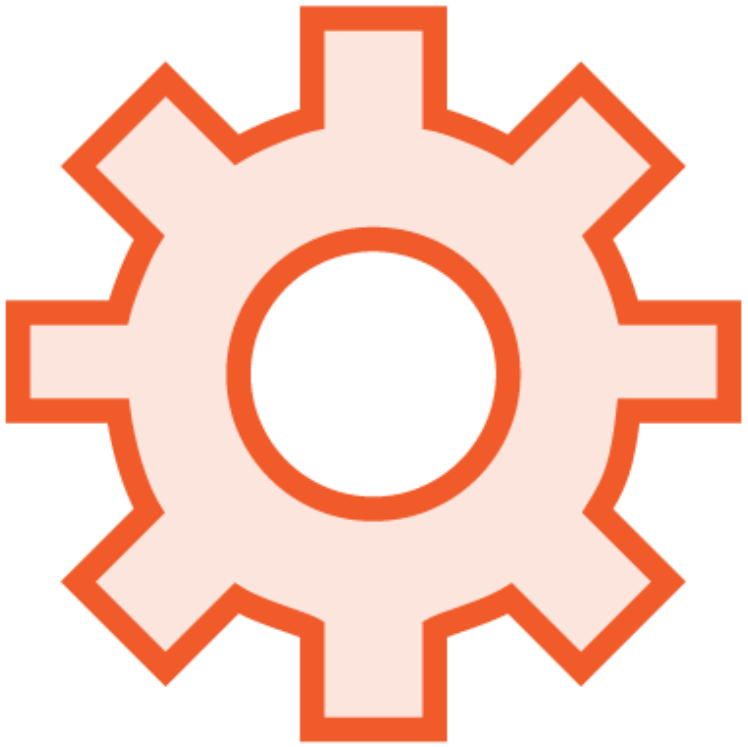
Module Overview

React to selections

React to errors

React to add operations

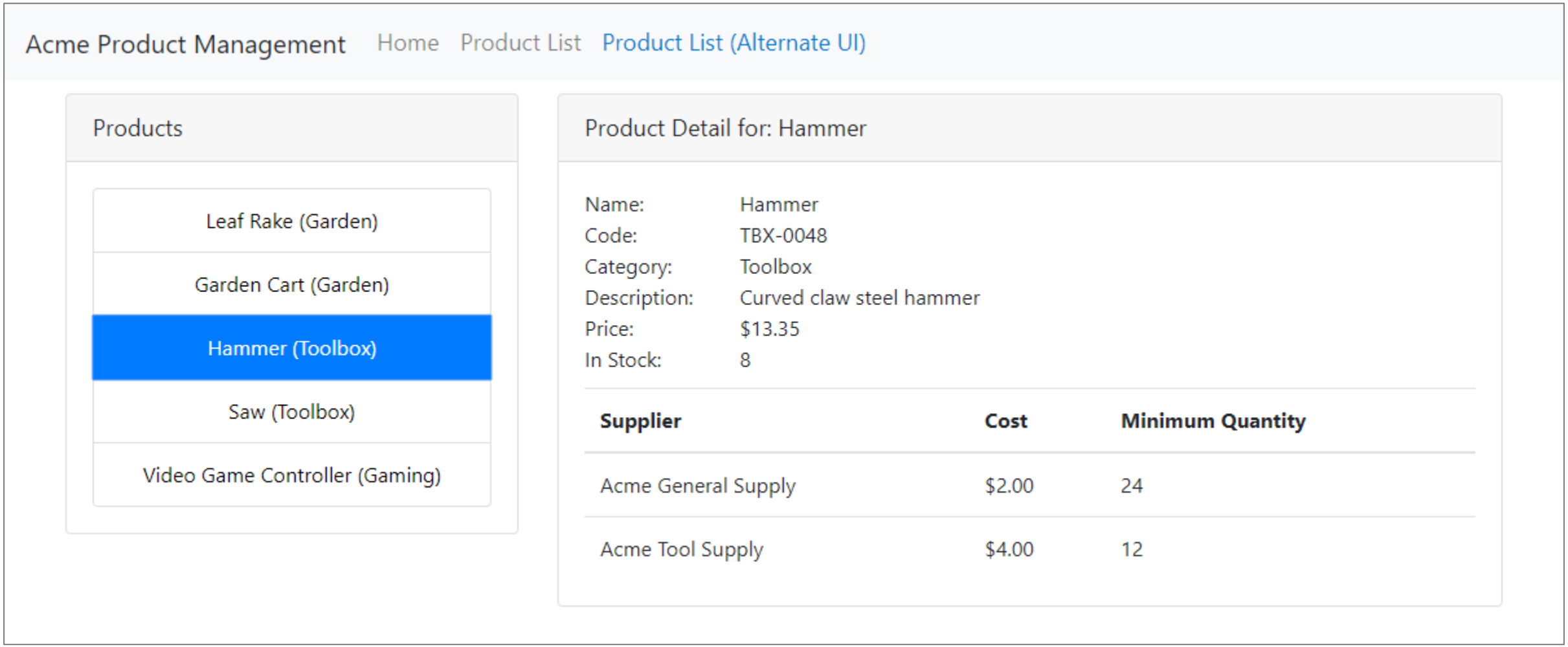
RxJS Features



`merge`

`scan`

Reacting to a Selection



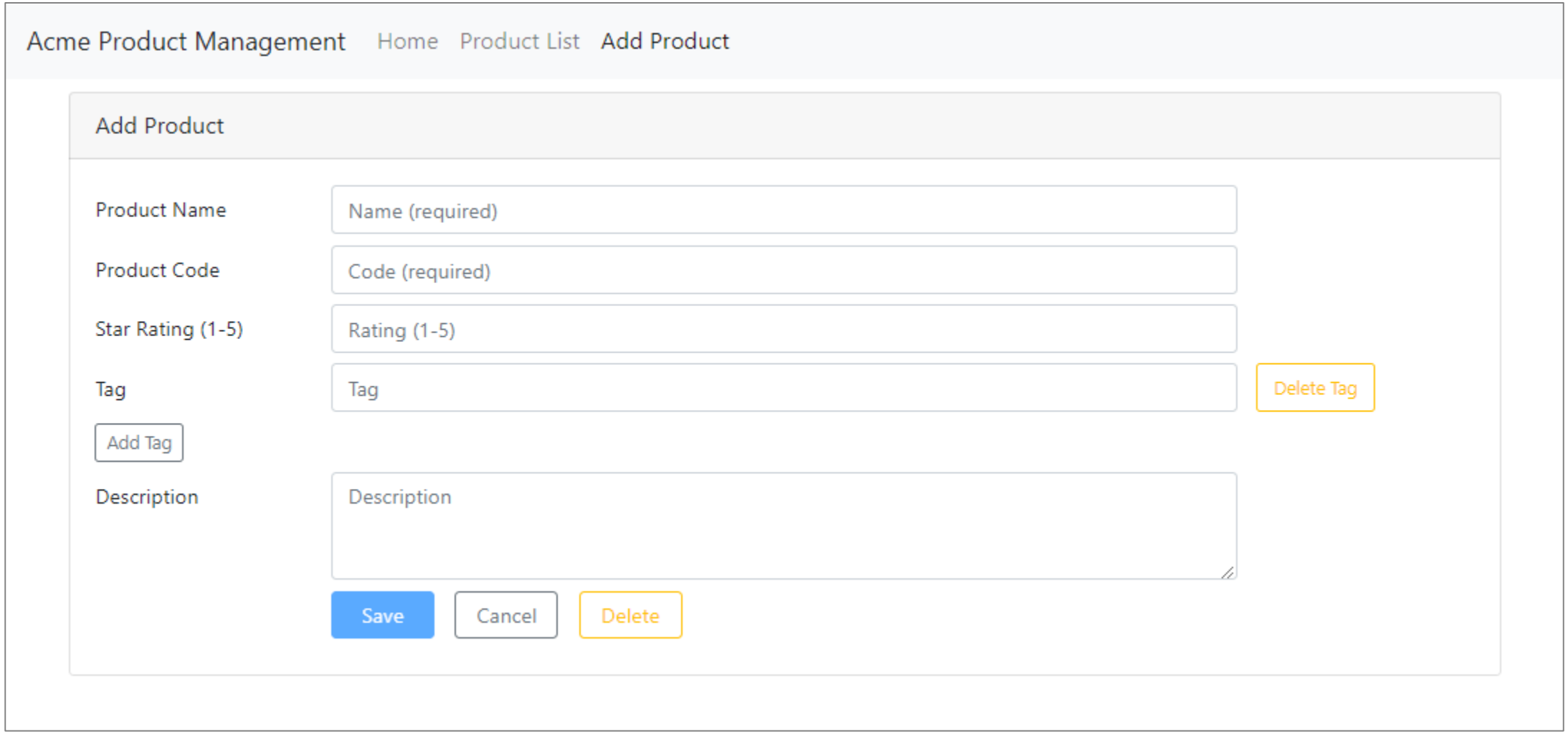
Demo

Reacting to a selection

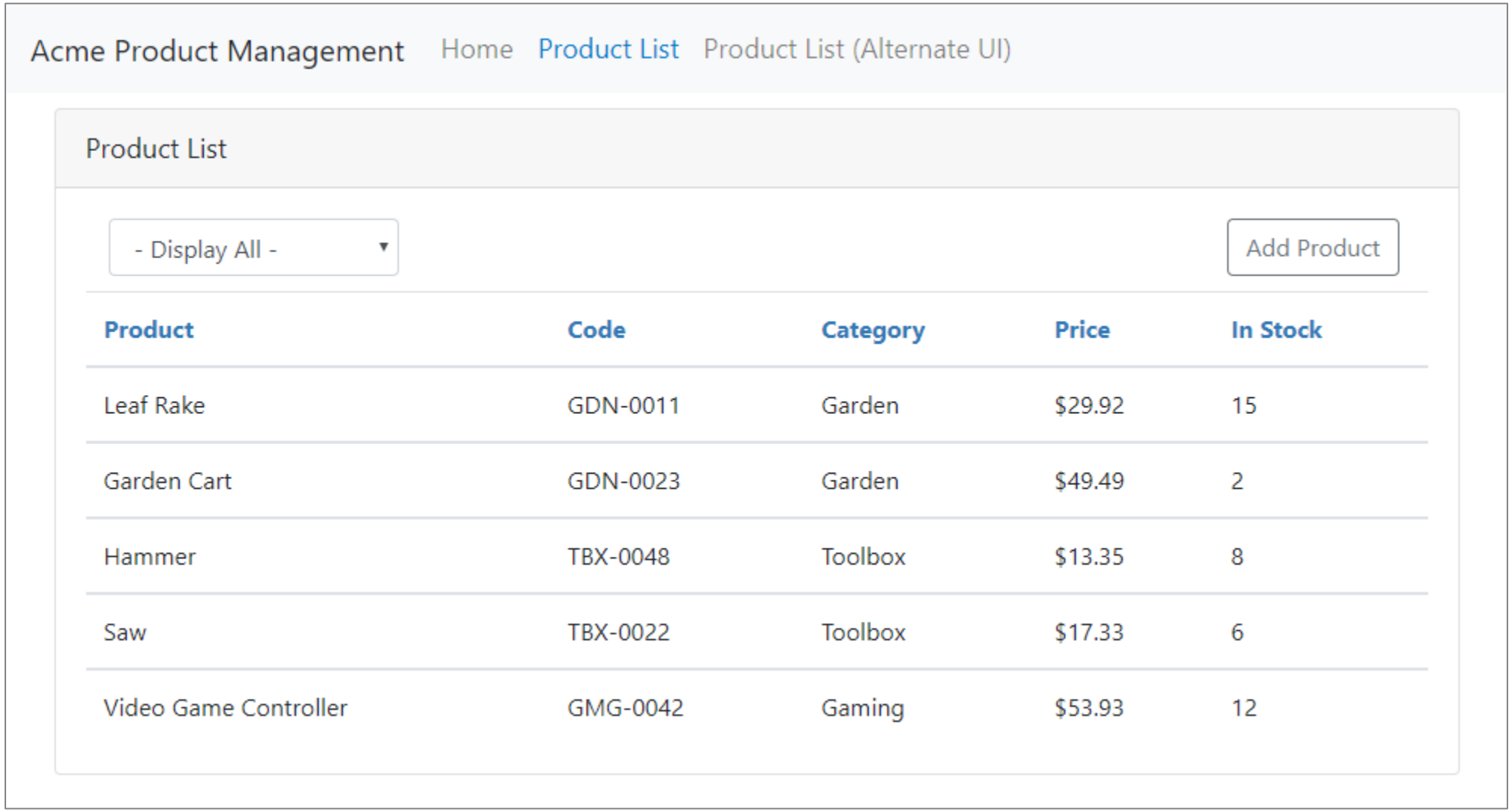
Demo

Reacting to an error

Reacting to an Add Operation



Reacting to an Add Operation



RxJS Creation Function: `merge`

Combines multiple streams by merging their emissions

```
merge(a$, b$, c$)
```

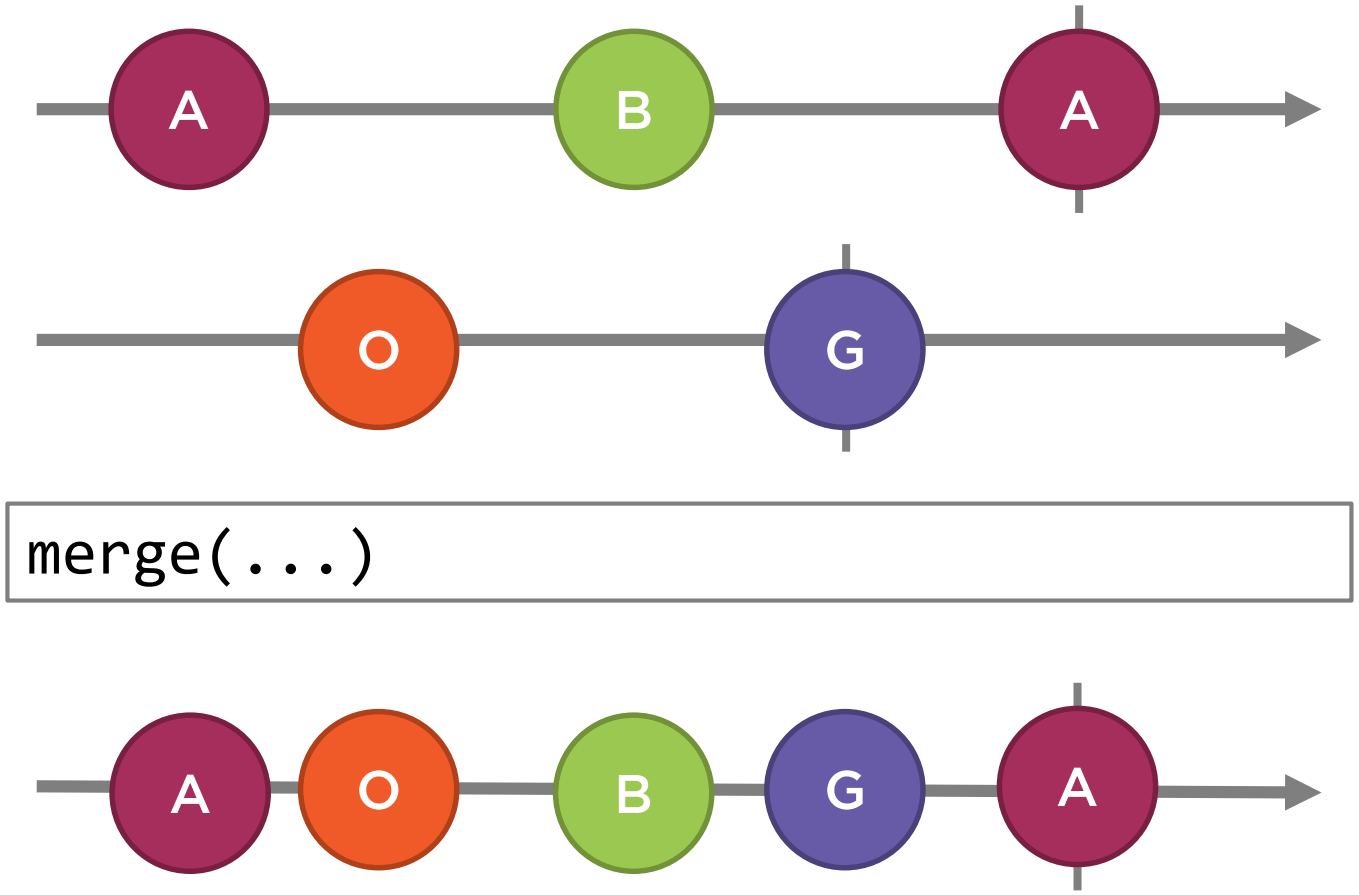
Static creation function, not a pipeable operator

Used for

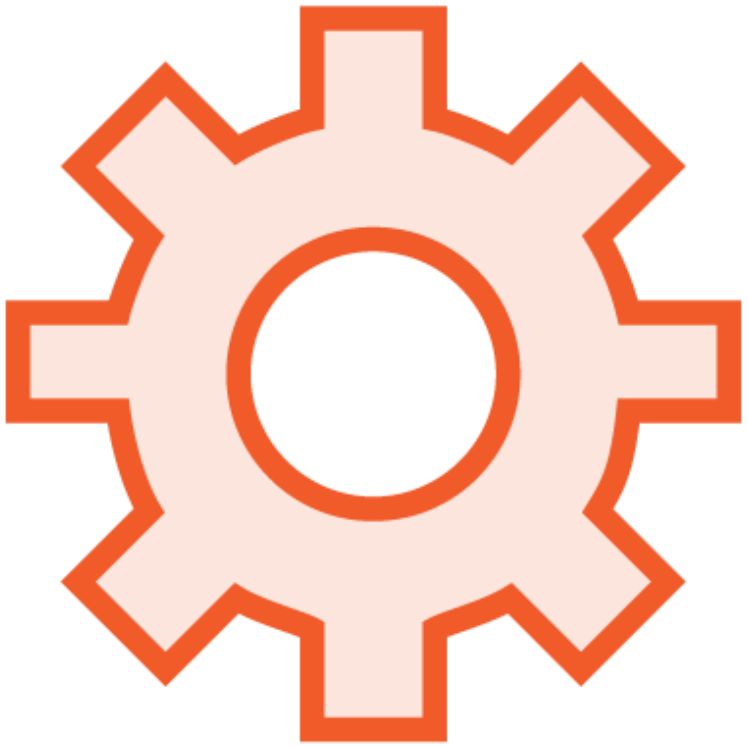
- Combining sequences of similar types to blend their emitted values

Marble Diagram: `merge`

```
merge(
  of('A', 'B', 'A'),
  of('O', 'G'),
  );
```



RxJS Creation Function: `merge`



`merge` **is a combination function**

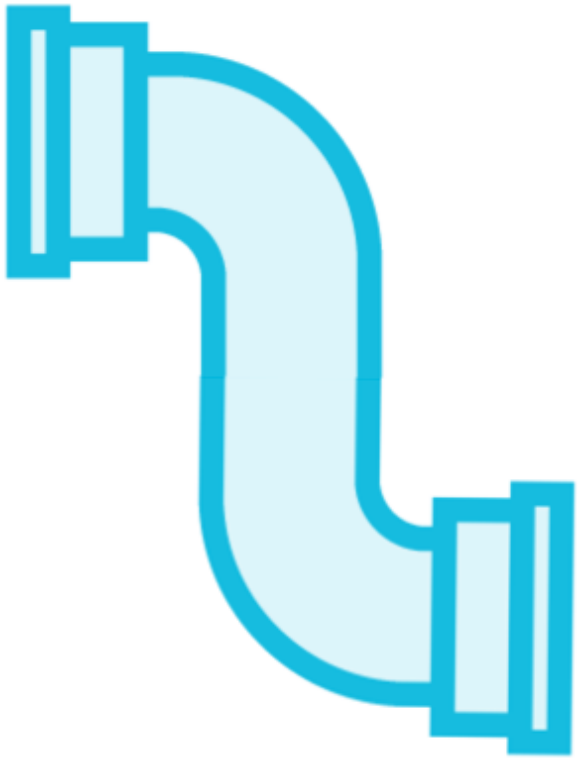
- Takes in a set of streams, subscribes
- Creates an output stream

When an item is emitted from any stream

- Item is emitted to the output stream

Completes when all input streams complete

RxJS Operator: scan



Accumulates items in a stream

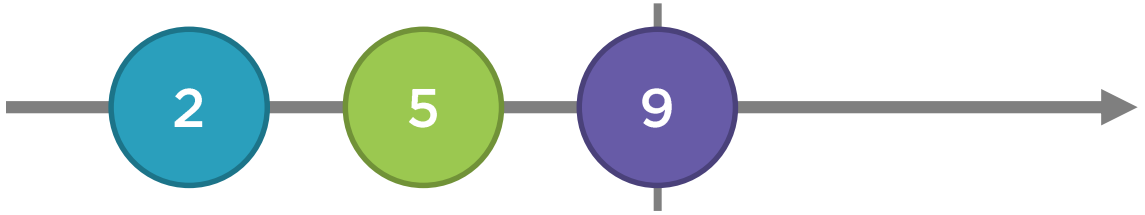
```
scan((acc, curr) => acc + curr)
```

Used for

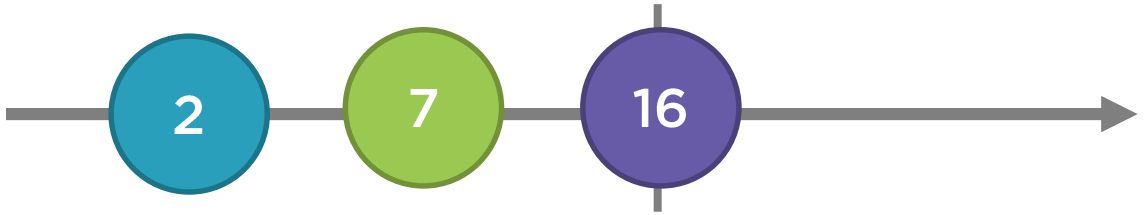
- Totaling amounts
- Accumulating items into an array

Marble Diagram: **scan**

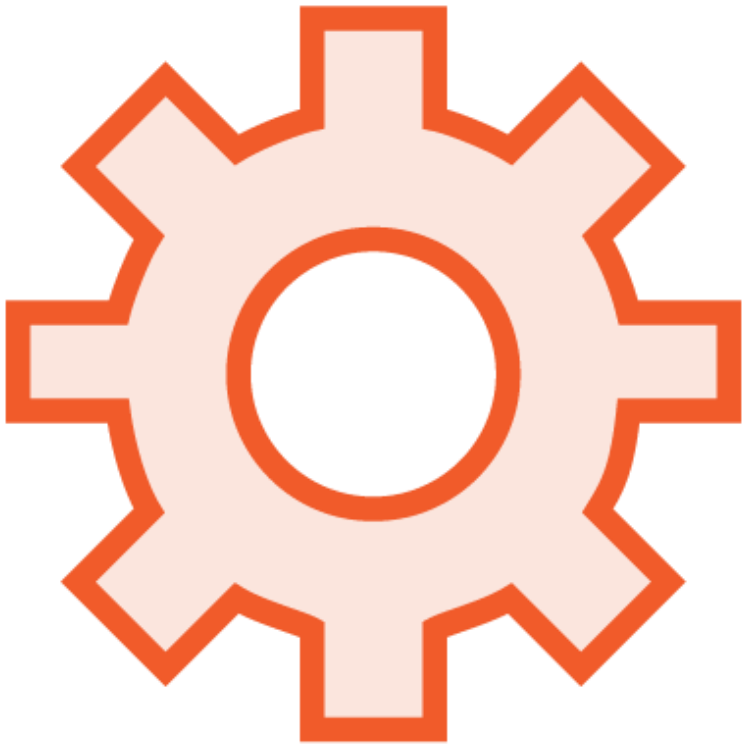
```
of(2, 5, 9)
  .pipe(
    scan((acc, curr) => acc + curr),
  );
```




```
scan((acc, curr) => acc + curr)
```



RxJS Operator: **scan**



scan **is a transformation operator**

- Takes in an input stream, subscribes
- Creates an output stream

When a item is emitted

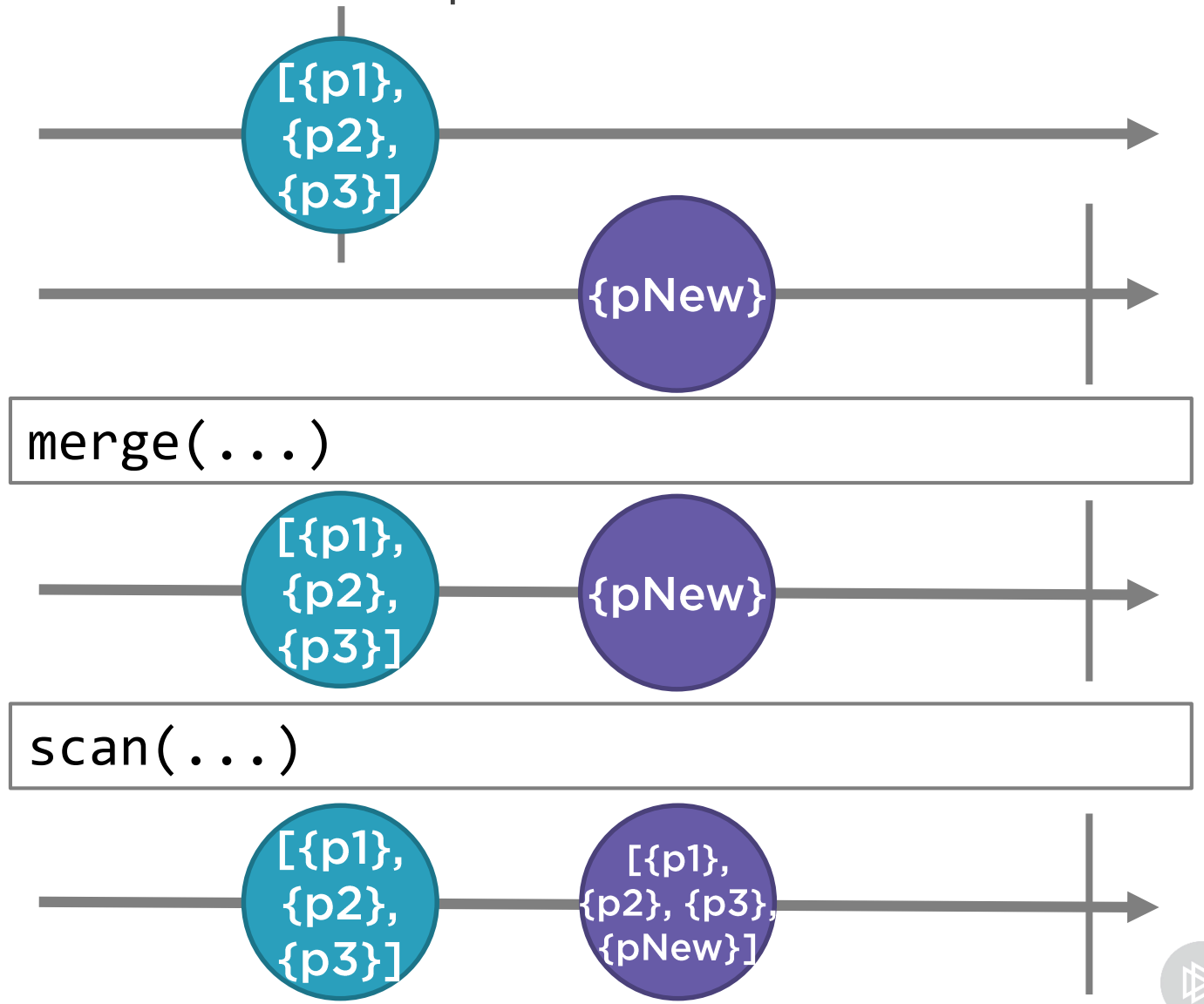
- Item is accumulated as specified by a provided function
- Intermediate result is emitted to the output stream

Reacting to an Add Operation

```
merge(
  this.products$,
  this.insertAction$
)
 .pipe(
    scan((acc: Product[],
          value: Product) =>
           [...acc, value])
  );
```

Reacting to an Add Operation

```
merge(
  this.products$,

  this.insertAction$

)




 .pipe(
    scan((acc: Product[],
          value: Product) =>
           [...acc, value])
  );
```



Demo

Reacting to an add operation

Reacting to Actions

Create an action stream (Subject/BehaviorSubject)

```
selSubject = new Subject<number>();
selectedAction$ = this.selSubject.asObservable();
```

Combine the action and data streams

```
products$ = combineLatest([
  this.productService.products$,
  this.selectedAction$
]).pipe(...);
```

Emit a value to the action stream when an action occurs

```
onSelected(id): void {
  this.selSubject.next(+id);
}
```

Reacting to a Selection

```
private pSelSubject = new BehaviorSubject<number>(0);
pSelAction$ = this.pSelSubject.asObservable();

selectedProduct$ = combineLatest([
  this.productsWithCategory$,
  this.pSelAction$
])
 .pipe(
   map(([products, selectedProductId]) =>
      products.find(product => product.id ===
                                          selectedProductId)
   )
 );
```

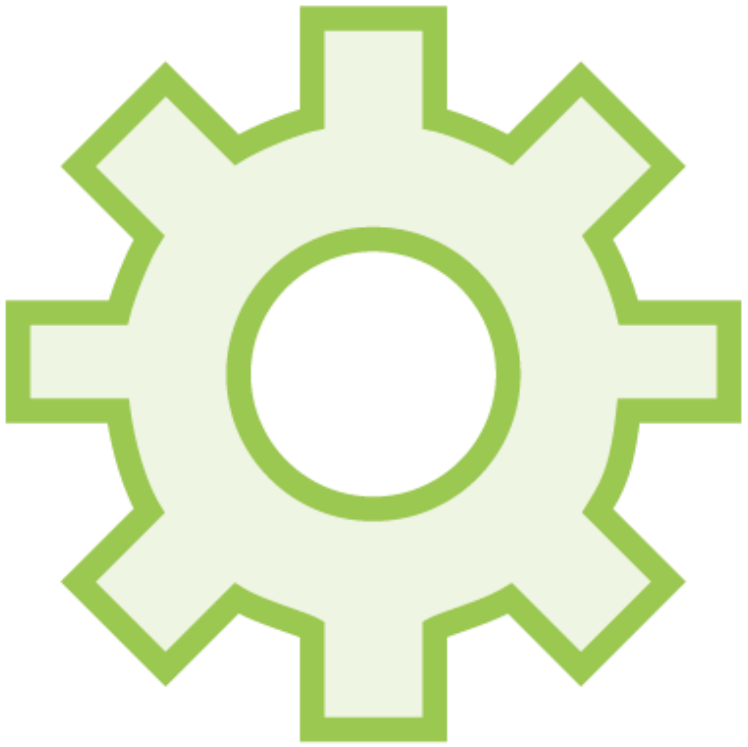
```
selProdChanged(selectedProductId){
  this.pSelSubject.next(selectedProductId);
}
```

Reacting to an Error

```
private errorSubject = new Subject<string>();
error$ = this.errorSubject.asObservable();

product$ = this.productService.selectedProduct$
  .pipe(
    catchError(err => {
      this.errorSubject.next(err);
      return EMPTY;
  })
);
```

```
<div
  *ngIf="error$ | async as errorMessage">
  {{errorMessage}}
</div>
```

RxJS Features



merge: Merges the emissions of multiple streams

```
merge(a$, b$, c$)
```

scan: Applies an accumulator function

```
scan((acc, curr) => acc + curr)
```

Reacting to an Add Operation

```
merge(
  this.products$,
  this.insertAction$
)
 .pipe(
    scan((acc: Product[],
          value: Product) =>
           [...acc, value])
 );
```

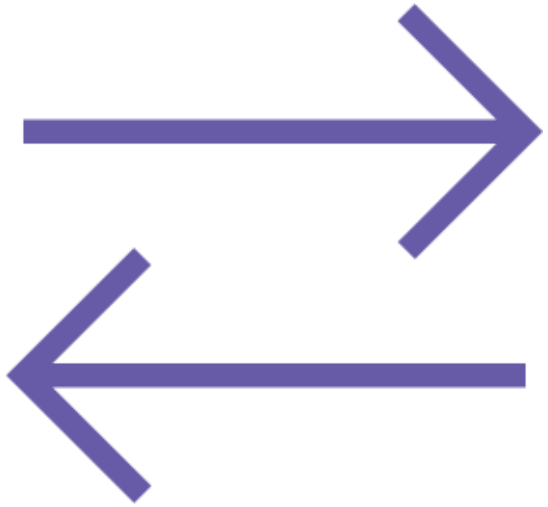
Going Reactive

Deborah Kurata

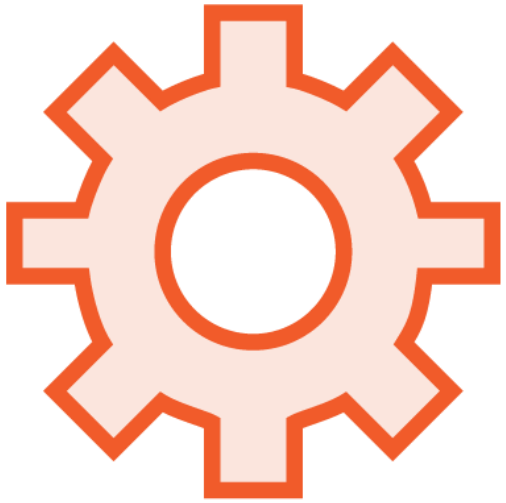
CONSULTANT | SPEAKER | AUTHOR | MVP | GDE

@deborahkurata | blogs.msmvps.com/deborahk/

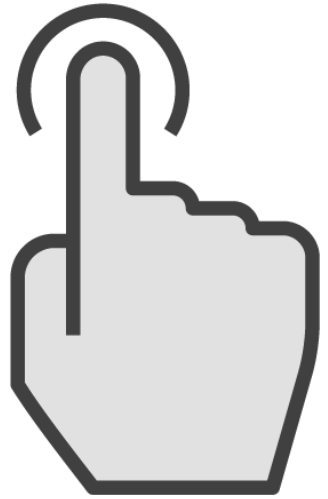
Going Reactive



Focus on async data streams



Leverage RxJS operators



React to actions

Module Overview

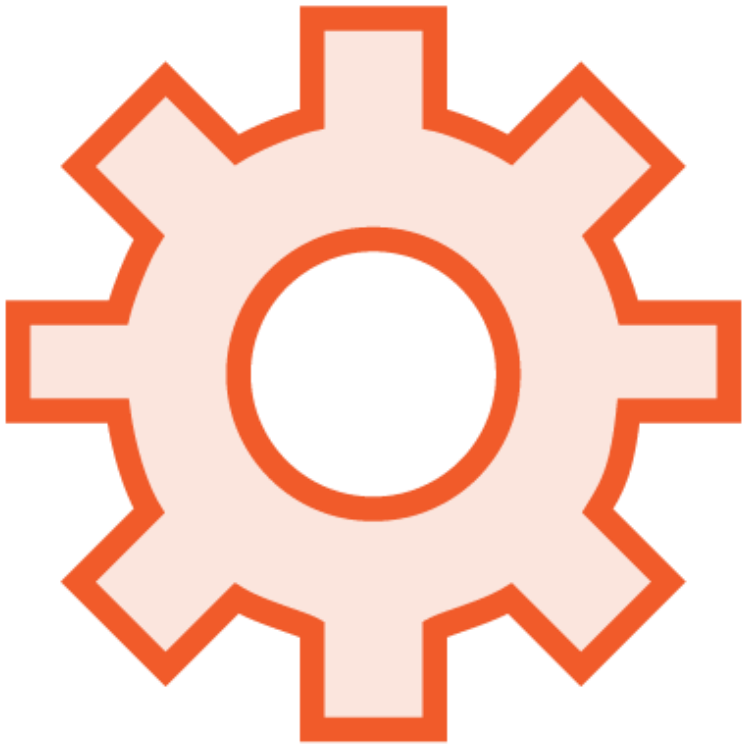
Working with the async pipe

Handling errors

Improving change detection

Declarative pattern for data retrieval

RxJS Features

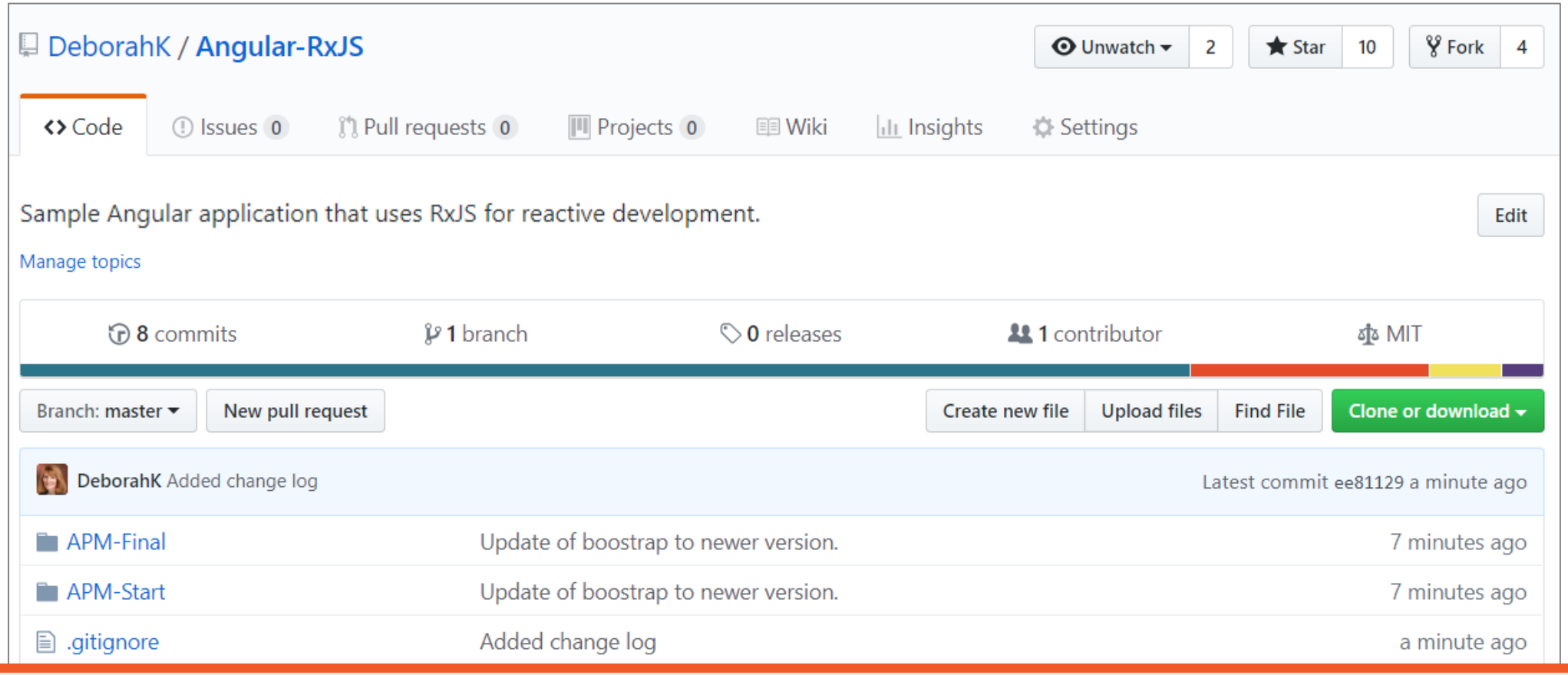


`catchError`

`EMPTY`

`throwError`

GitHub Repository



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

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

Product List Component

```
products$: Observable<Product[]>;

constructor(private productService: ProductService) { }

ngOnInit() {
 this.products$ = this.productService.getProducts();
}
```

Template with an Async Pipe

Product List Template

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

<table>
  <tr *ngFor="let product of products">
    <td>{{ product.productName }}</td>
    <td>{{ product.productCode }}</td>
  </tr>
</table>
```

Product List Template

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

<table>
  <tr *ngFor="let product of products">
    <td>{{ product.productName }}</td>
    <td>{{ product.productCode }}</td>
  </tr>
</table>
```

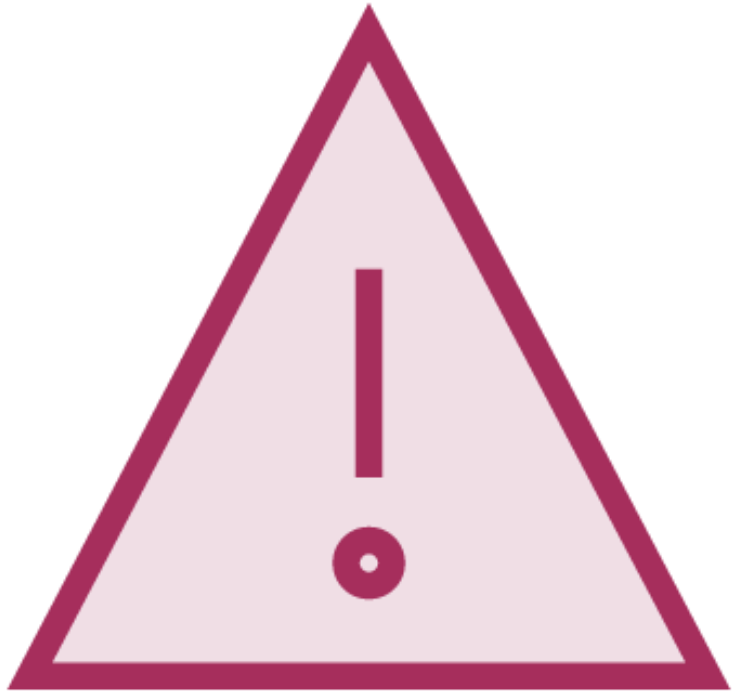
Handling Errors

Catch Observable errors

Error stops the Observable

It won't emit any more items

We can't use it anymore

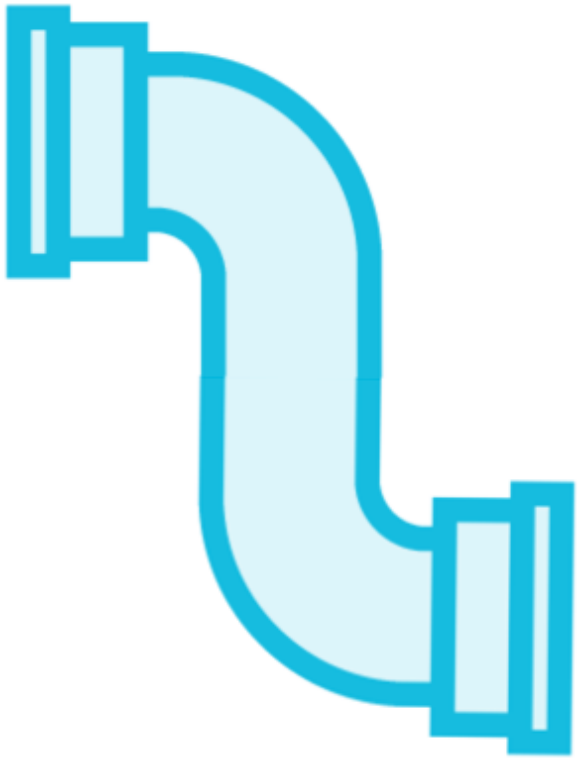


Handling Errors

Catch and Replace

Catch and Rethrow

RxJS Operator: `catchError`



Catches any errors that occur on an Observable

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

Used for catching errors and

- Rethrowing an error
- Or replacing the errored Observable to continue after an error occurs

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

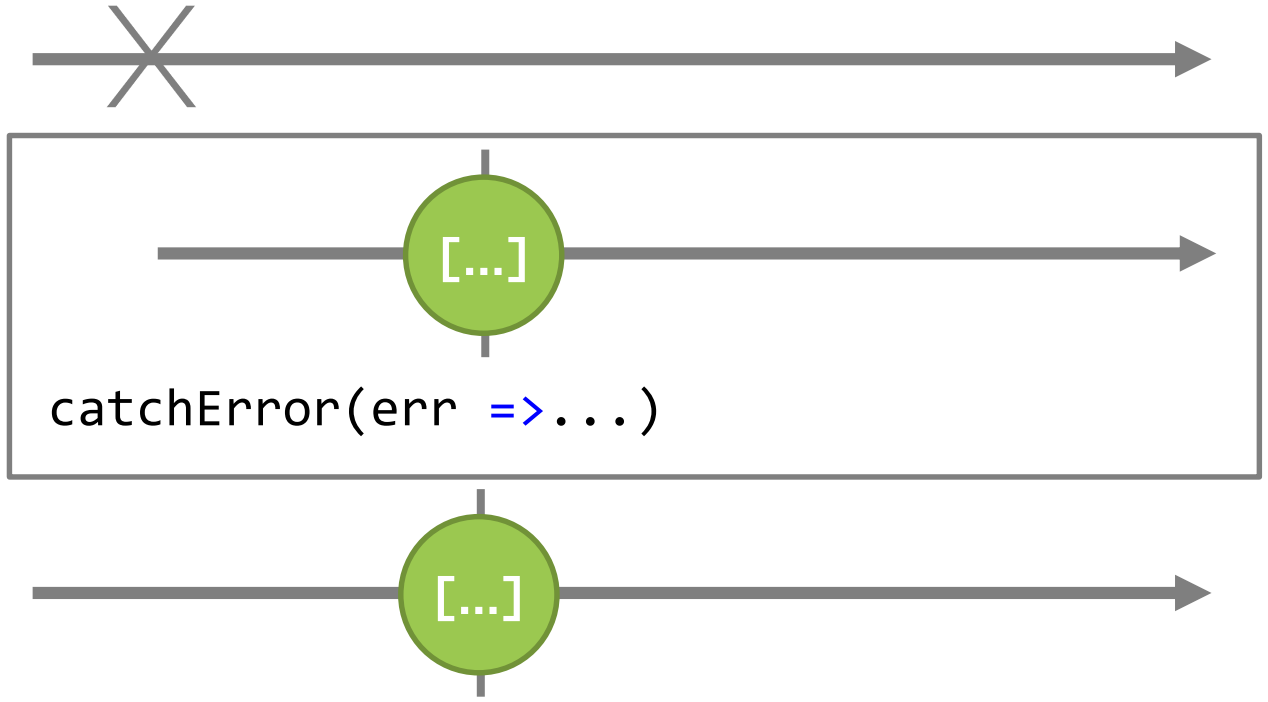
```
return this.http.get<Product[]>(this.productsUrl)
  .pipe(
    catchError(err => {
      console.error(err);
      return of([{ id: 1, productName: 'cart'},
                 { id: 2, productName: 'hammer'}]);
  });
```

Product List Component

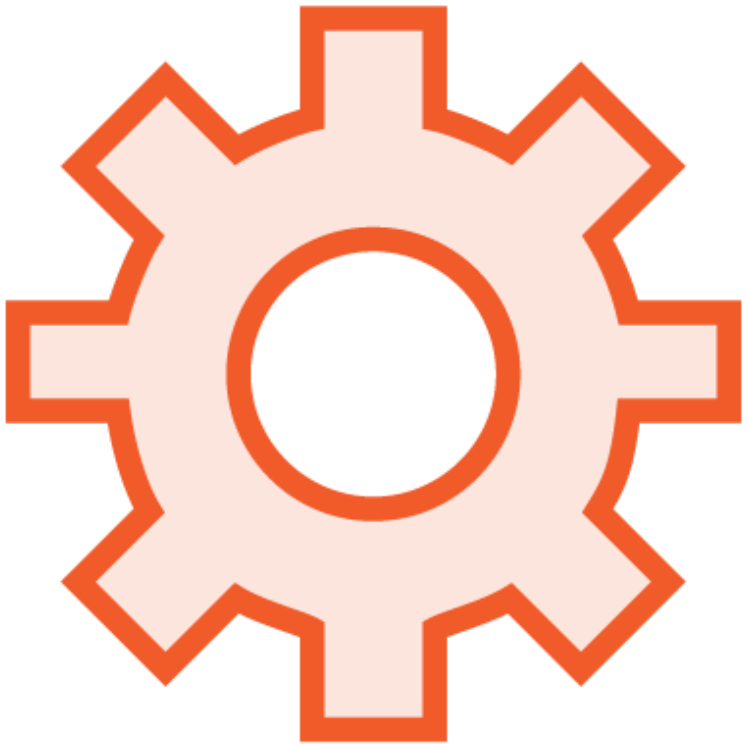
```
ngOnInit() {
 this.productService.getProducts()
  .subscribe(
    products => this.products = products,
    err => this.errorMessage = err
  );
}
```

Marble Diagram: `catchError`

```
return this.http.get<Product[]>(this.url)
 .pipe(
  catchError(err => {
   console.error(err);
   return of(
    [{ id: 1, productName: 'cart'},
     { id: 2, productName: 'hammer'}
    ]);
  })
 );
```



RxJS Operator: `catchError`



`catchError` is an error handling operator

- Takes in an input stream, subscribes
- Creates an output stream

When a source item is emitted

- Item is emitted to the output stream

If an error occurs

- Catches the error
- Unsubscribes from the input stream
- Returns a replacement Observable
- Optionally rethrows the error

Handling Errors

Catch and Replace

Catch and Rethrow

Catch and Rethrow

Product Service

```
return this.http.get<Product[]>(this.productsUrl)
  .pipe(
    catchError(err => {
      console.error(err);
      return throwError(err);
  });
```

RxJS Creation Function: `throwError`

Creates an Observable that emits no items

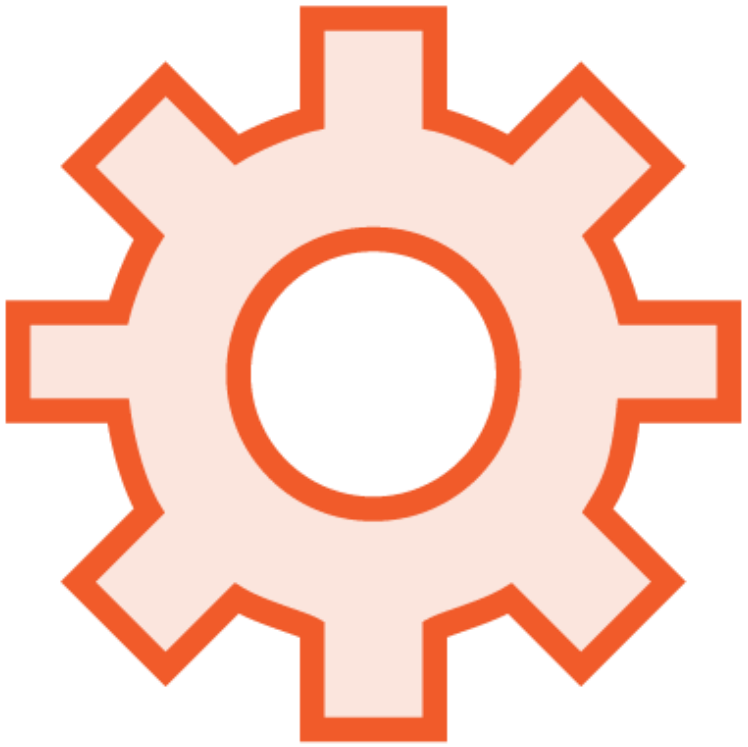
And immediately emits an error notification

```
throwError(err)
```

Used for

- Propagating an error

RxJS Creation Function: `throwError`



throwError is a creation function

Creates an Observable that emits no items

- Observable<never>

Immediately emits an error notification

Common Pattern with Error Handling

Product Service

```
private productsUrl = 'api/products';

getProducts(): Observable<Product[]> {
  return this.http.get<Product[]>(this.productsUrl)
    .pipe(
      catchError(this.handleError)
    );
}

private handleError(err) {
  // ...
  return throwError(errorMessage);
}
```

Error Handling

Product List Component

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

Product List Component

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

RxJS Constant: **EMPTY**

Returns an Observable that emits no items

And immediately emits a complete notification

```
return EMPTY;
```

Used for

- Returning an empty Observable

Error Handling

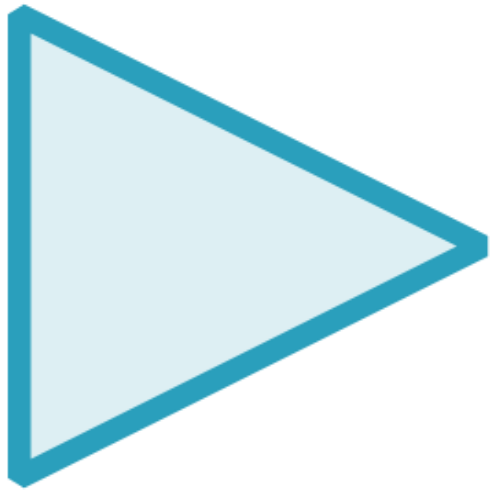
Product List Component

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

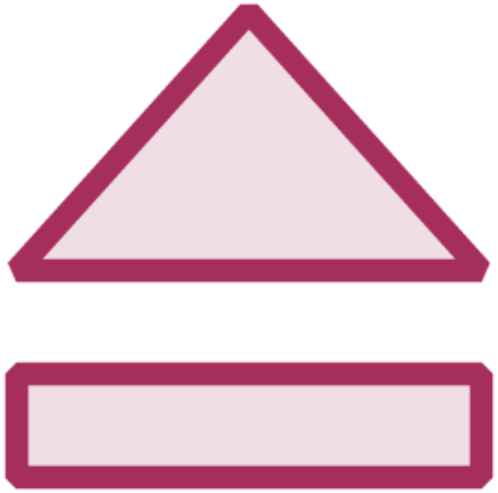
Demo

Handling errors

Benefits of an Async Pipe



No need to subscribe



No need to unsubscribe

Improve change detection

Change Detection Strategies

Angular uses change detection to track changes to application data structures ...

... So it knows when to update the UI with changed data

Uses the default chec

Every component is checked when:

- Any change is detected

nance 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
})
```

Common Pattern

Product Service

```
private productsUrl = 'api/products';
getProducts(): Observable<Product[]> {
  return this.http.get<Product[]>(this.productsUrl)
    .pipe(
      catchError(this.handleError)
    );
}
```

Product List Component

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

Declarative Pattern

Product Service

```
private productsUrl = 'api/products';

products$ = this.http.get<Product[]>(this.productsUrl);
```

Product List Component

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

Declarative Pattern with Error Handling

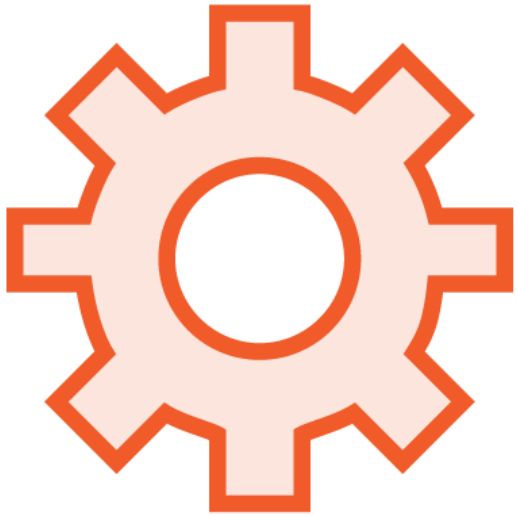
Product Service

```
private productsUrl = 'api/products';

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

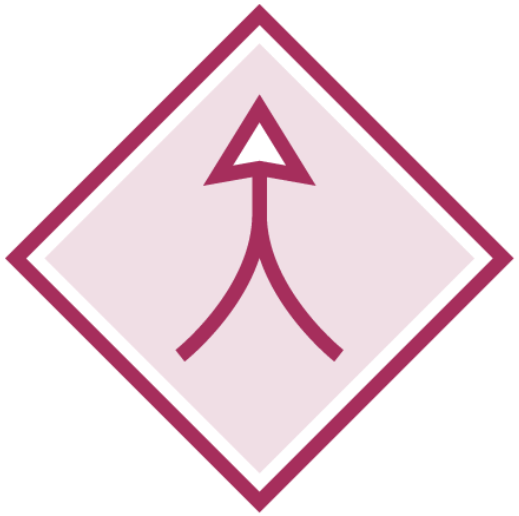
Product List Component

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

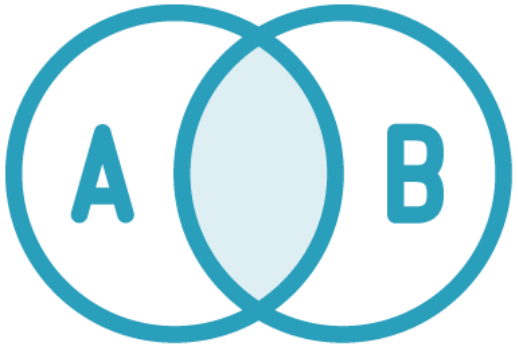
Benefits of a Declarative Approach



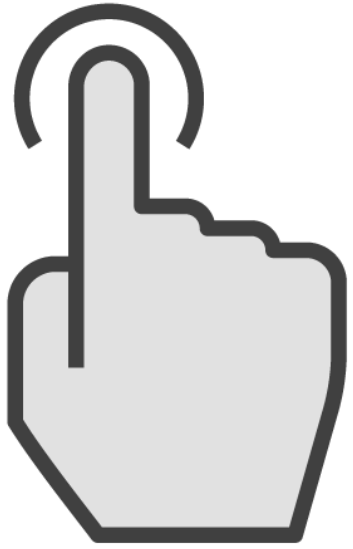
Leverages the power of RxJS Observables and operators



Effectively combine streams



Easily share Observables



Readily react to actions

Checklist: Data Retrieval

Define the shape of the data

- Interface or Class

```
export interface Product {
  id: number;
  productName: string;
  productCode: string;
  categoryId: number;
  description: string;
}
```

Checklist: Data Retrieval

Build a service

- Set a property to the Observable returned from http.get
- Use the type argument to map the response to the desired shape
- When the response is received, it's emitted and the Observable completes
- Pipe through desired operators

```
private productsUrl = 'api/products';

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

Checklist: Data Retrieval

In a component, assign the service property to a local property

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

Use OnPush change detection

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

Checklist: Data Retrieval

In the template, use an async pipe

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

<table>
  <tr *ngFor="let product of products">
    <td>{{ product.productName }}</td>
    <td>{{ product.productCode }}</td>
  </tr>
</table>
```

Checklist: Handling Errors

Catch and replace

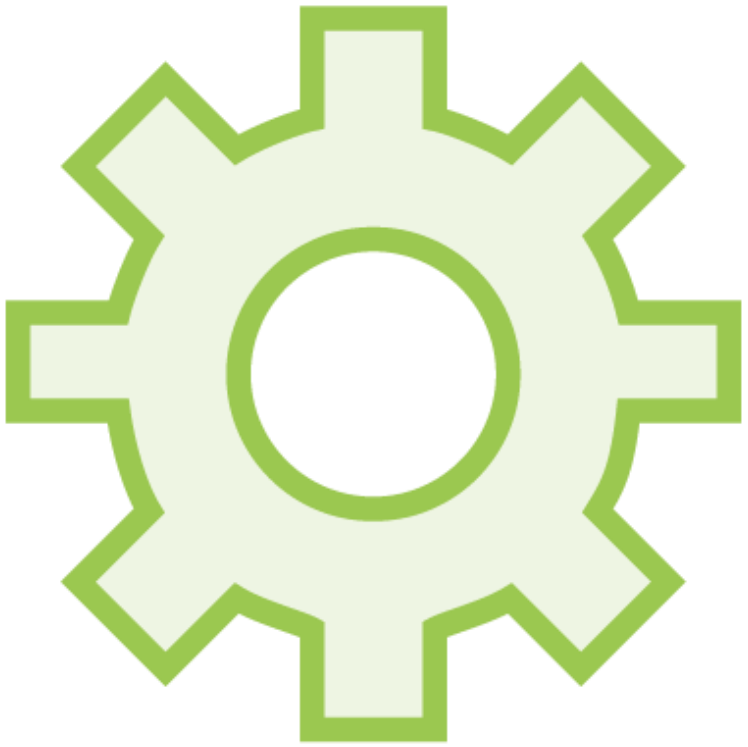
- An Observable that emits an alternate set of data
- An Observable that emits an empty set
- EMPTY

```
catchError(err => {
   this.errorMessage = err;
   return EMPTY;
})
```

Catch and rethrow

```
catchError(err => {
   console.error(err);
   return throwError(err);
})
```

RxJS Features



catchError: Catches any error and replaces the error Observable with a new Observable

throwError: Creates an Observable that emits no items and immediately emits an error notification

```
products$ = this.http.get<Product[]>(this.productsUrl)
  .pipe(
    catchError(err => {
      console.error(err);
      return throwError(err);
  });
```

Mapping Returned Data

Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR | MVP | GDE

@deborahkurata | blogs.msmvps.com/deborahk/

Why Map Returned Data?

Modify a value: `price = price * 1.5`

Transform a value: 'Y' -> true; 'N' -> false

Change a field name: p_nm -> productName

Add a calculated field: `profit = price - cost`

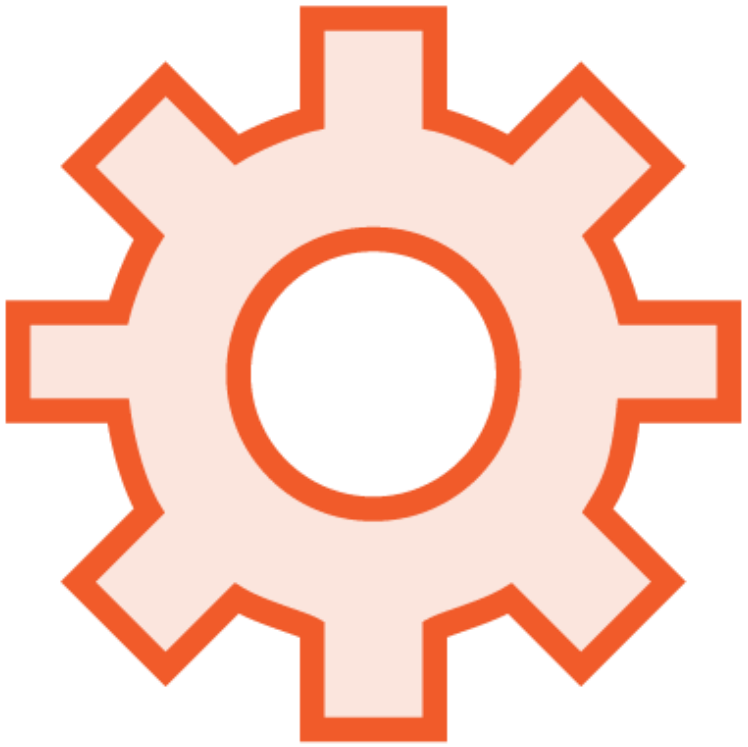
Simple mapping
doesn't always work.

Module Overview

Mapping an HTTP response

Mapping array elements

RxJS Features



`map`

Mapping a Simple Observable

```
of(2, 4, 6)
    .pipe(
      map(item => item * 2)
    ).subscribe(console.log); // 4 8 12
```

Mapping an HTTP Response

Product Service

```
private productsUrl = 'api/products';

product$ =
  this.http.get<Product>(`${this.productsUrl}/${id}`);
```

Response

{hammer}

Response

[{cart}, {hammer}]

Backend Server

```
export interface Product {
  id: number;
  productName: string;
  productCode: string;
}
```

```
{
  id: undefined,
  productId: 2,
  productName: undefined
  name: 'cart',
  productCode: 'GDN-0023'
}
```

```
{
  productId: 2,
  name: 'cart',
  productCode: 'GDN-0023'
}
```

Mapping an HTTP Response

Map the emitted array

Map each element in the array

Transform each element

Mapping an Emitted Array

```
products$ = this.http.get<Product[]>(this.productsUrl)
  .pipe(
    map(products =>
        ...
    ),
    catchError(this.handleError)
);
```

Mapping an HTTP Response

Map the emitted array

Map each element in the array

Mapping Array Elements

```
products$ = this.http.get<Product[]>(this.productsUrl)
 .pipe(
   map(products =>
     products.map(product => ...)
   ),
   catchError(this.handleError)
);
```

Mapping an HTTP Response

Map the emitted array

Map each element in the array

Transform each element

Transforming Array Elements

```
products.map(product => ({
   id: product.id,
   productName: product.productName,
   productCode: product.productCode,
   description: product.description,
   price: product.price * 1.5,
   searchKey: [product.productName]
}) as Product)
```

```
products.map(product => ({
   ...product,
   price: product.price * 1.5,
   searchKey: [product.productName]
}) as Product)
```

Checklist: Mapping an HTTP Response

- Map the emitted array
- Map the array elements
- Transform each array element

Checklist: Mapping an Emitted Array

Map the array

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

Checklist: Mapping Array Elements

Use the array's map method

```
products$ = this.http.get<Product[]>(this.url)
  .pipe(
    map(products =>
      products.map(product => { ... })
    )
  );
```

Checklist: Transforming Array Elements

- Define an object literal of the desired type
- Leverage the spread operator
- Replace or add fields as needed
- Use the as clause to type the result

```
products.map(product => ({
   ...product,
   price: product.price * 1.5,
   searchKey: [product.productName]
}) as Product)
```

Higher-order Mapping Operators

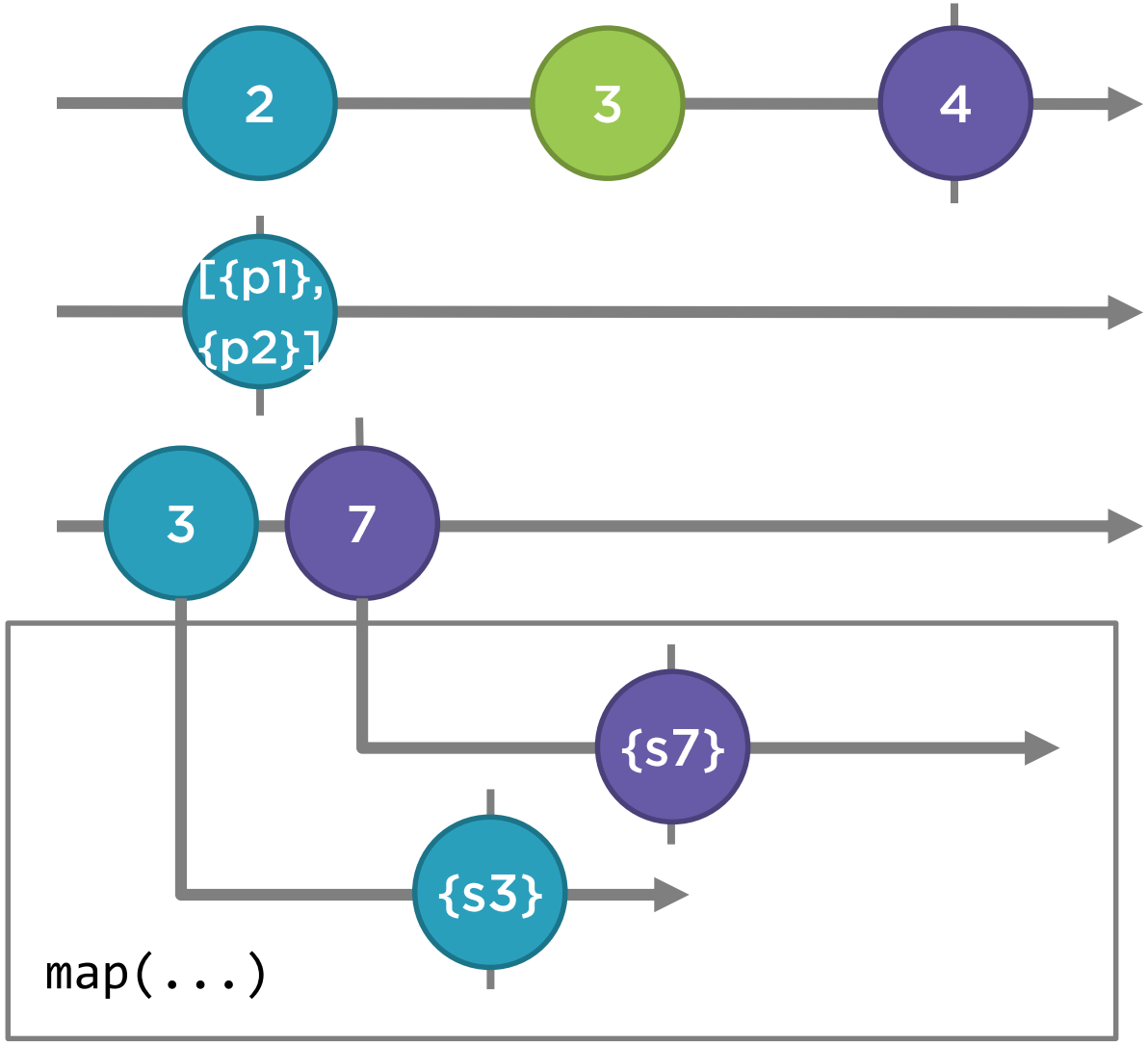
Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR | MVP | GDE
@deborahkurata | blogs.msmvps.com/deborahk/

Observables

```
of(2, 3, 4)
  .subscribe();
```

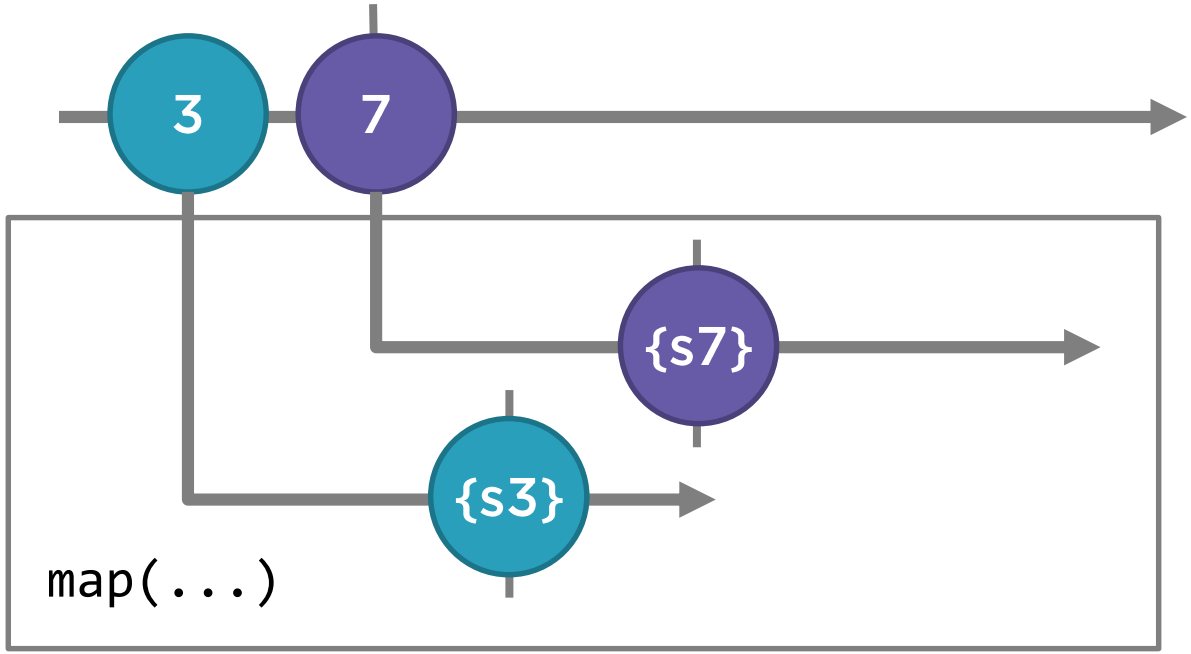
```
this.http.get<Product[]>(this.url)
  .subscribe();
```

```
of(3, 7)
 .pipe(
  map(id => this.http.get<Supplier>
              (`${this.url}/${id}`)
 )).subscribe();
```



Higher-order Observables

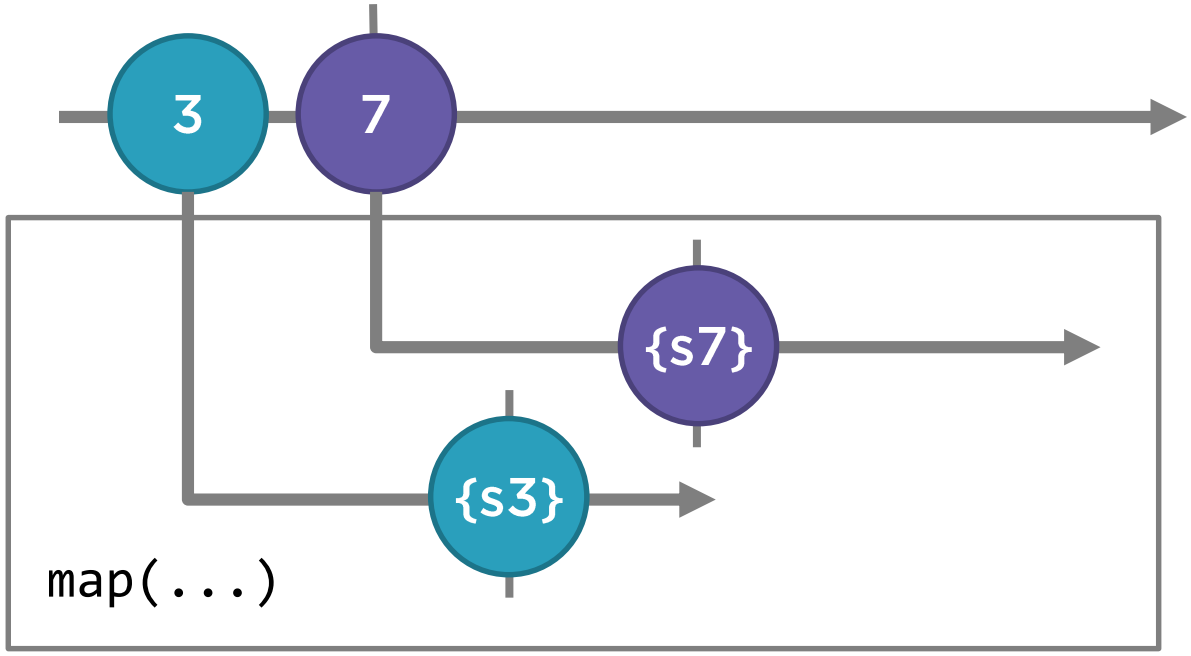
```
of(3, 7)
 .pipe(
  map(id => this.http.get<Supplier>
              (`${this.url}/${id}`)
 )).subscribe();
```



Higher-order Observables

```
of(3, 7)
 .pipe(
  map(id => this.http.get<Supplier>
              (`${this.url}/${id}`)
 )).subscribe();
```

```
of(3, 7)
 .pipe(
  map(id => this.http.get<Supplier>
              (`${this.url}/${id}`)
 )).subscribe(o =>
      o.subscribe()
 );
```

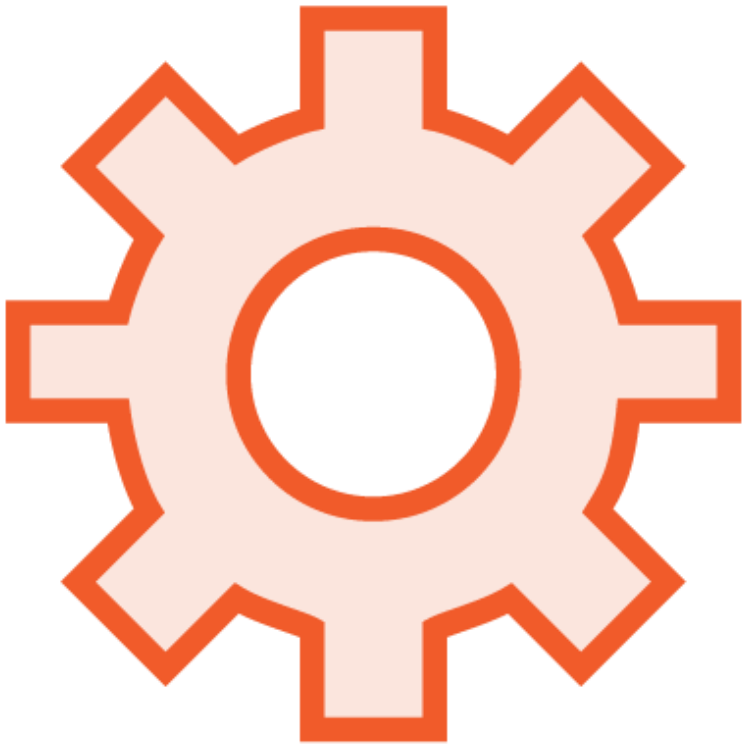


Higher-order mapping operators transform higher-order Observables.

Module Overview

Higher-order mapping operators

RxJS Features



`concatMap`

`mergeMap`

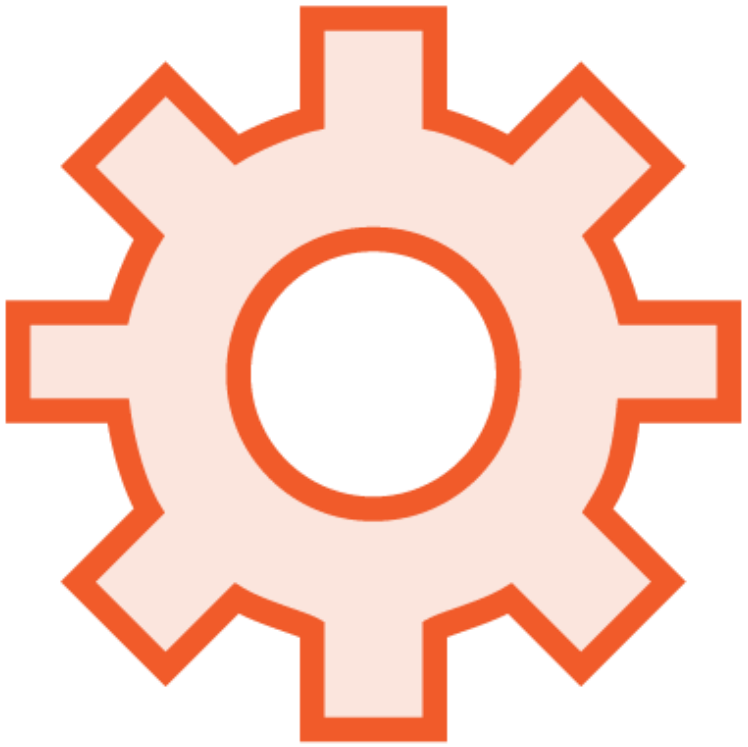
`switchMap`

Higher-order Mapping Operators

```
export interface Product {
  id: number;
  productName: string;
  productCode?: string;
  description?: string;
  price?: number;
  categoryId?: number;
  category?: string;
  supplierIds?: number[];
}
```

```
of(1, 5, 8)
 .pipe(
   map(id => this.http.get<Supplier>(`${this.url}/${id}`))
 ).subscribe(console.log);
```

Higher-order RxJS Mapping Operators



Family of operators: xxxMap()

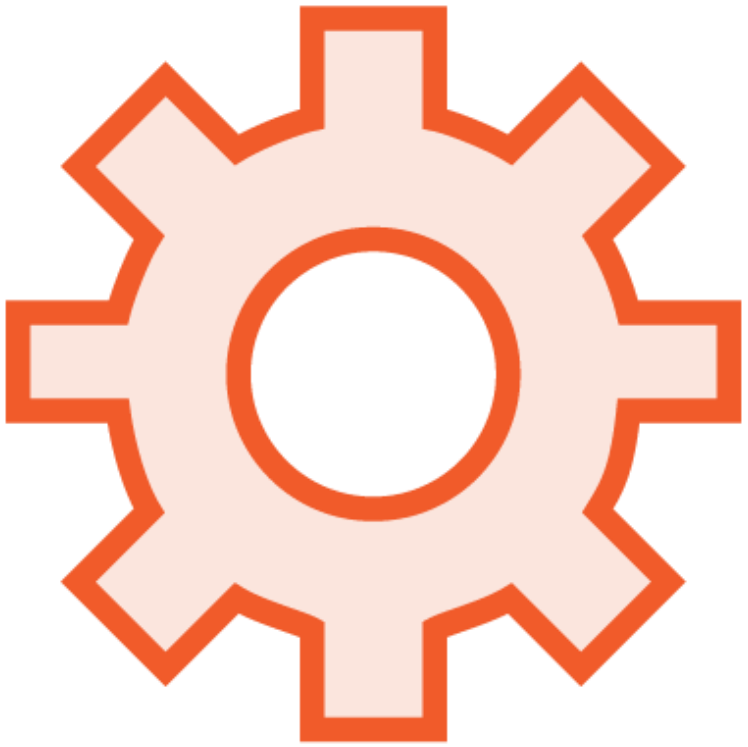
Map each value

- From a source (outer) Observable
- To a new (inner) Observable

Automatically subscribe/unsubscribe from inner Observables

Emit the resulting values to the output Observable

Higher-order RxJS Mapping Operators

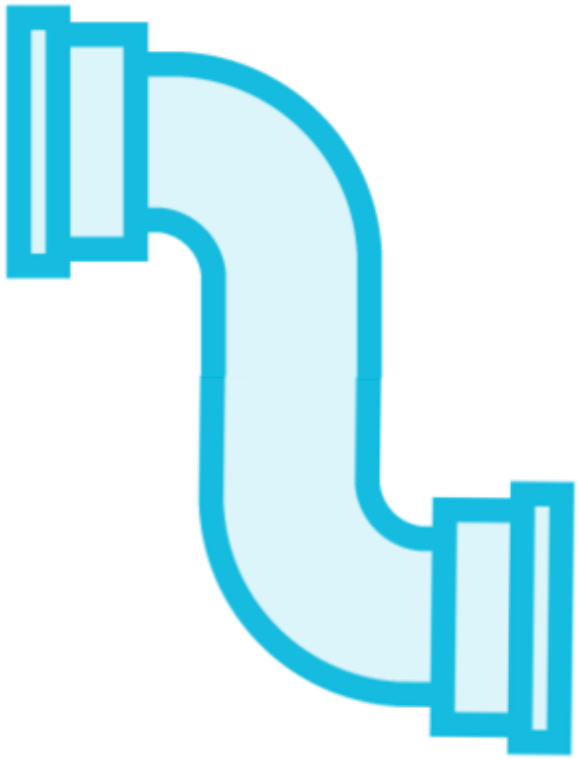


`concatMap`

`mergeMap`

`switchMap`

RxJS Operator: `concatMap`



Higher-order mapping + concatenation

Transforms each emitted item to a new (inner) Observable as defined by a function

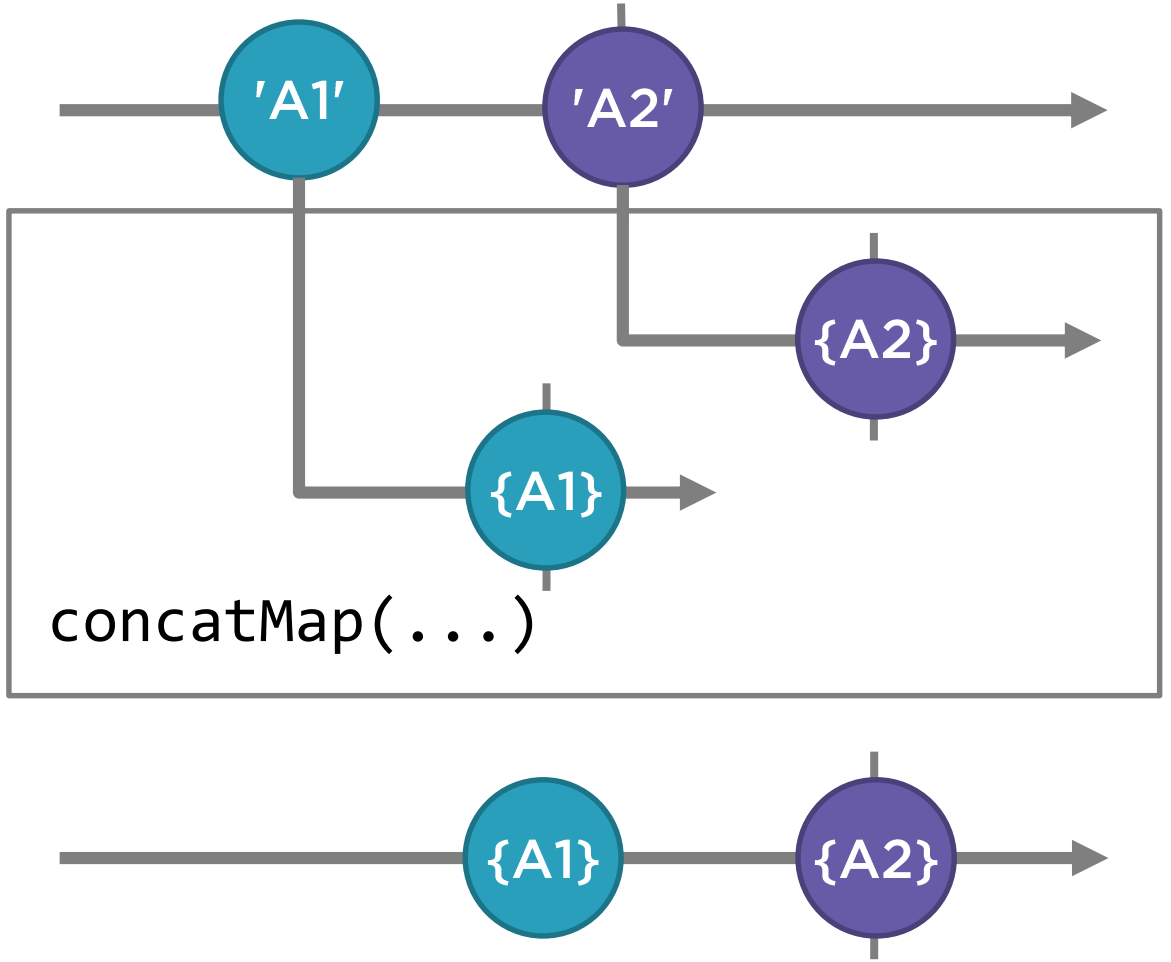
```
concatMap(i => of(i))
```

It waits for each inner Observable to complete before processing the next one

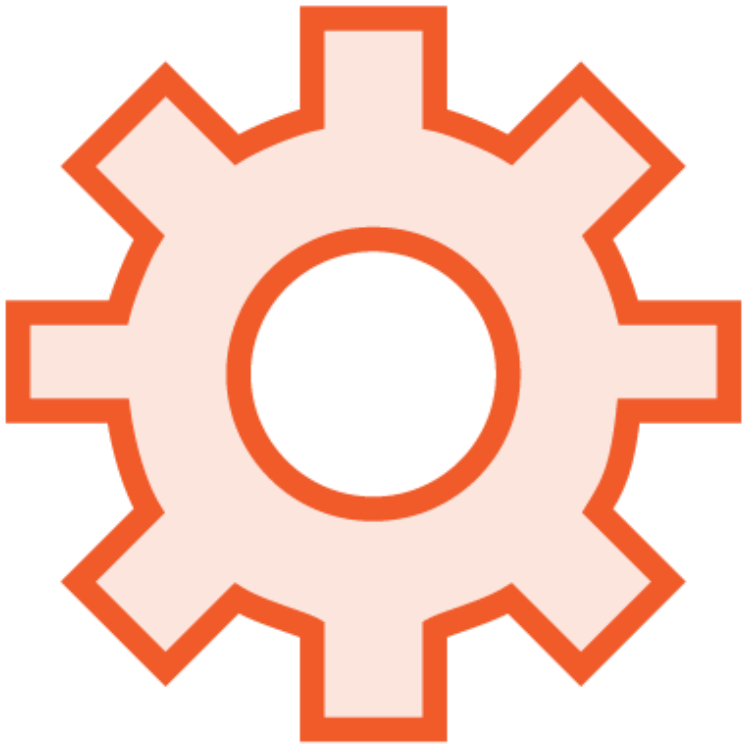
Concatenates their results in sequence

Marble Diagram: concatMap

```
of('A1', 'A2')
 .pipe(
  concatMap(id =>
 this.http.get<Apple>(`${this.url}/${id}`))
 ).subscribe(console.log);
```



RxJS Operator: concatMap



concatMap is a transformation operator

- Takes in an input stream, subscribes
- Creates an output stream

When an item is emitted, it's queued

- Item is mapped to a inner Observable as specified by a provided function
- Subscribes to inner Observable
- Waits!
- Inner Observable emissions are concatenated to the output stream
- When inner Observable completes, processes the next item

Use concatMap

To wait for the prior Observable to complete before starting the next one

To process items in sequence

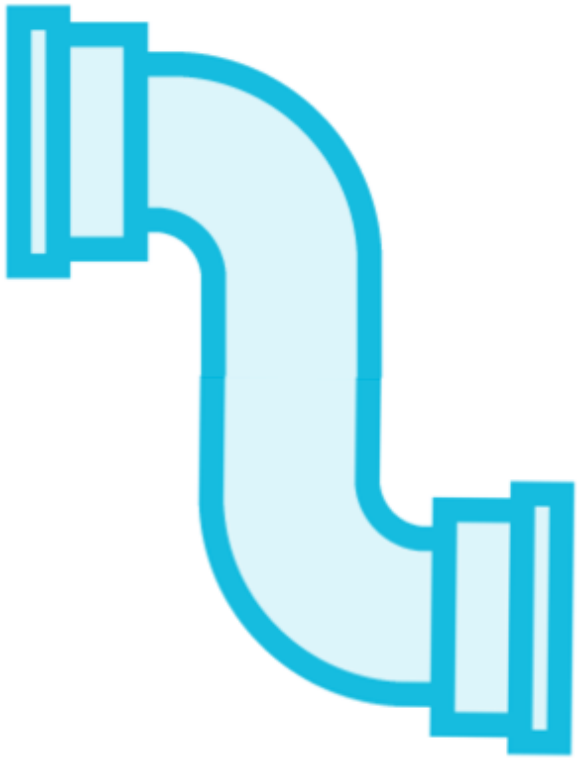
Examples:

- From a set of ids, get data in sequence
- From a set of ids, update data in sequence

Demo

`concatMap`

RxJS Operator: `mergeMap`



Higher-order mapping + merging

Transforms each emitted item to a new (inner) Observable as defined by a function

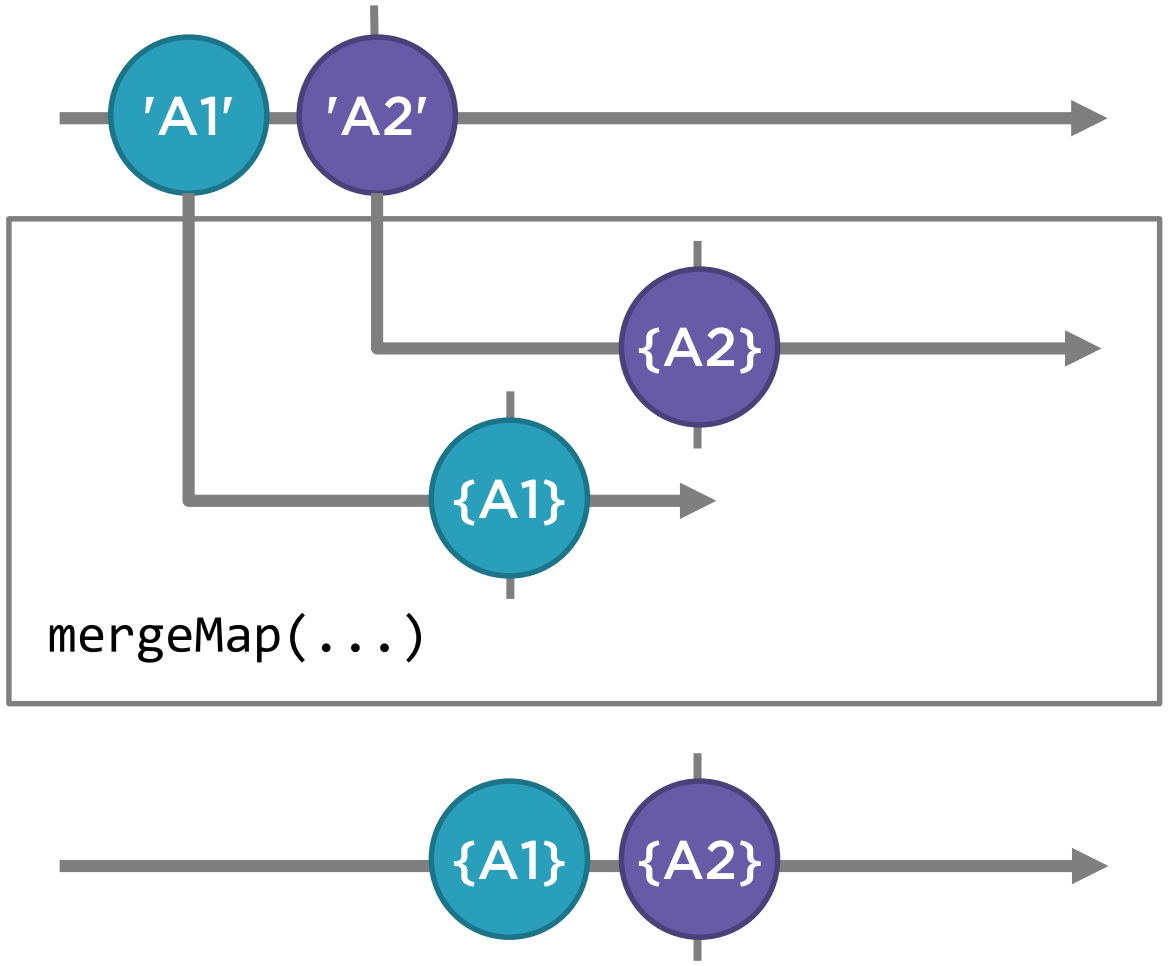
```
mergeMap(i => of(i))
```

It executes inner Observables in parallel

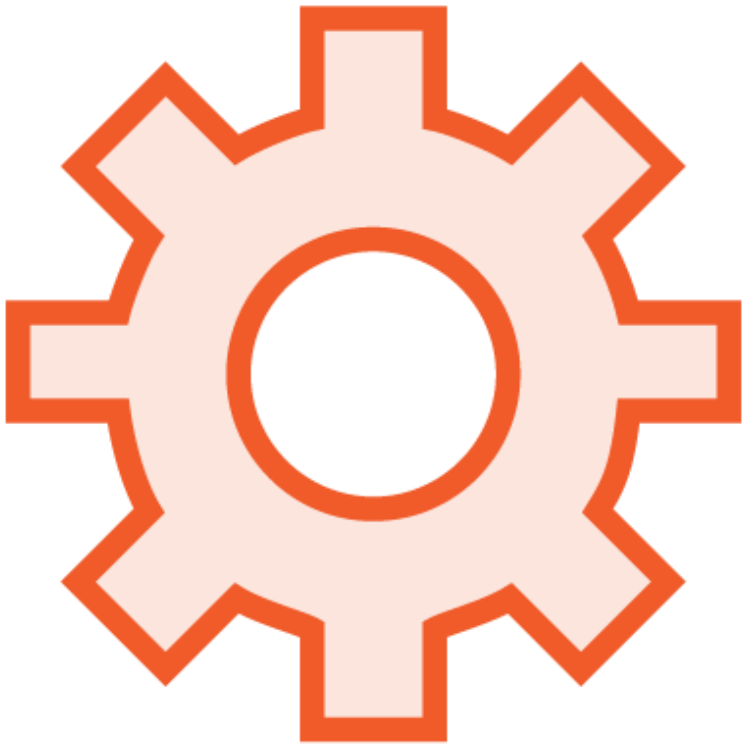
And merges their results

Marble Diagram: `mergeMap`

```
of('A1', 'A2')
 .pipe(
  mergeMap(id =>
 this.http.get<Apple>(`${this.url}/${id}`))
 ).subscribe(console.log);
```



RxJS Operator: `mergeMap (flatMap)`



`mergeMap` **is a transformation operator**

- Takes in an input stream, subscribes
- Creates an output stream

When each item is emitted

- Item is mapped to a inner Observable as specified by a provided function
- Subscribes to inner Observable
- Inner Observable emissions are merged to the output stream

Use mergeMap

To process in parallel

When order doesn't matter

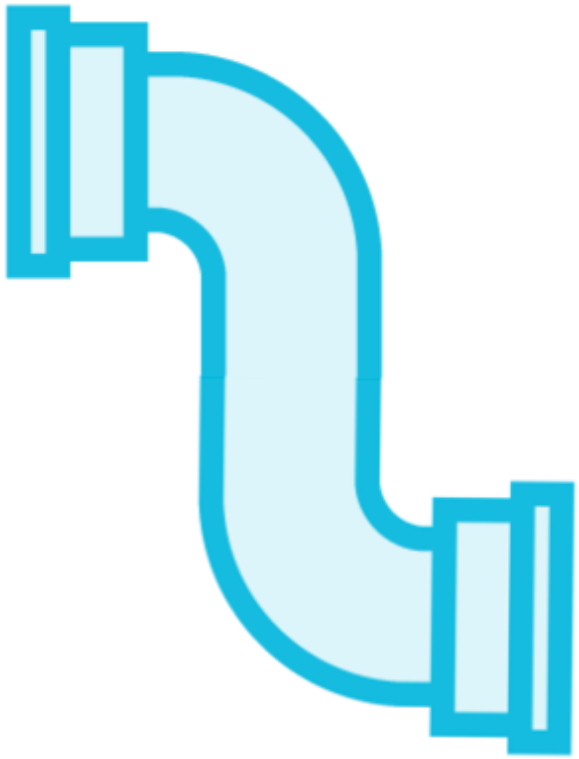
Examples:

- From a set of ids, retrieve data (order doesn't matter)

Demo

`mergeMap`

RxJS Operator: `switchMap`



Higher-order mapping + switching

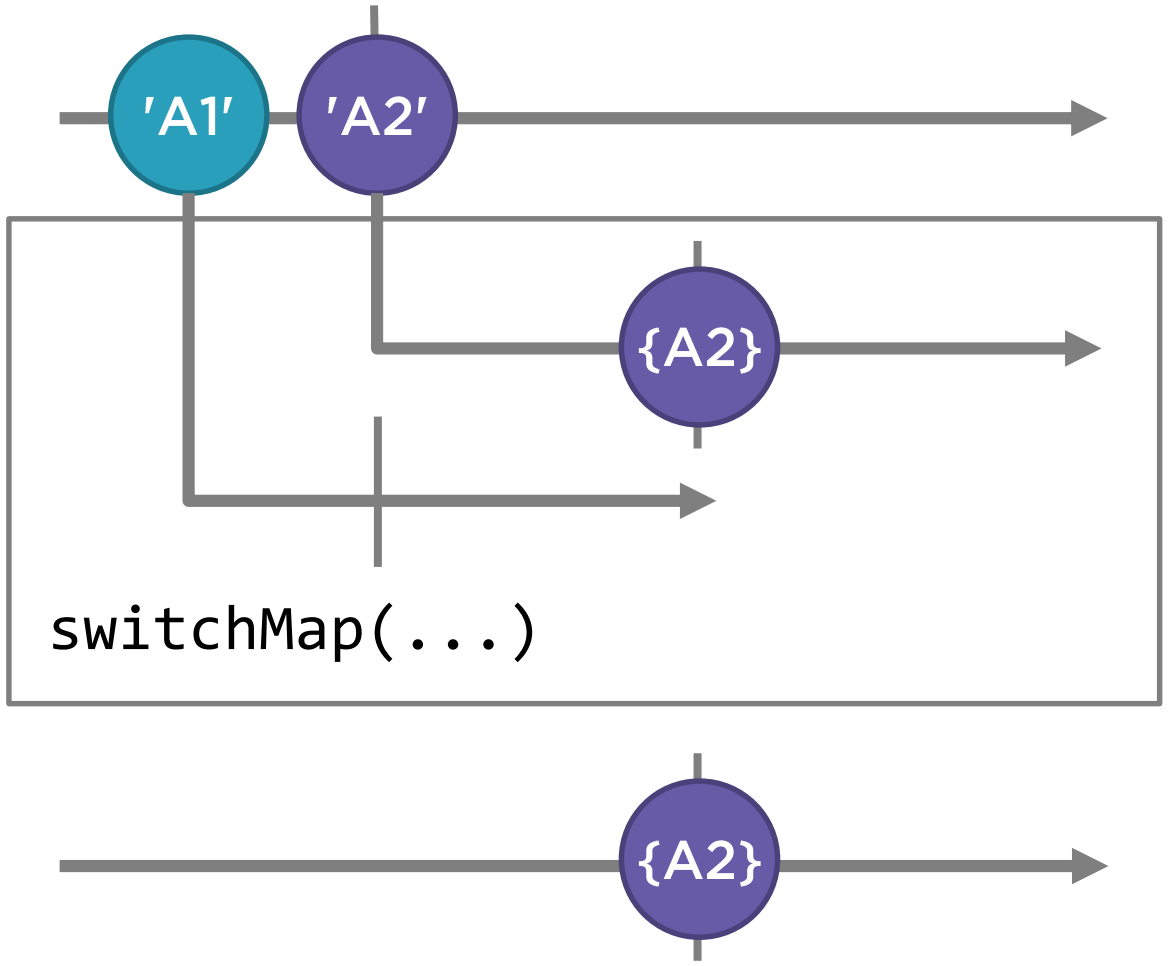
Transforms each emitted item to a new (inner) Observable as defined by a function

```
switchMap(i => of(i))
```

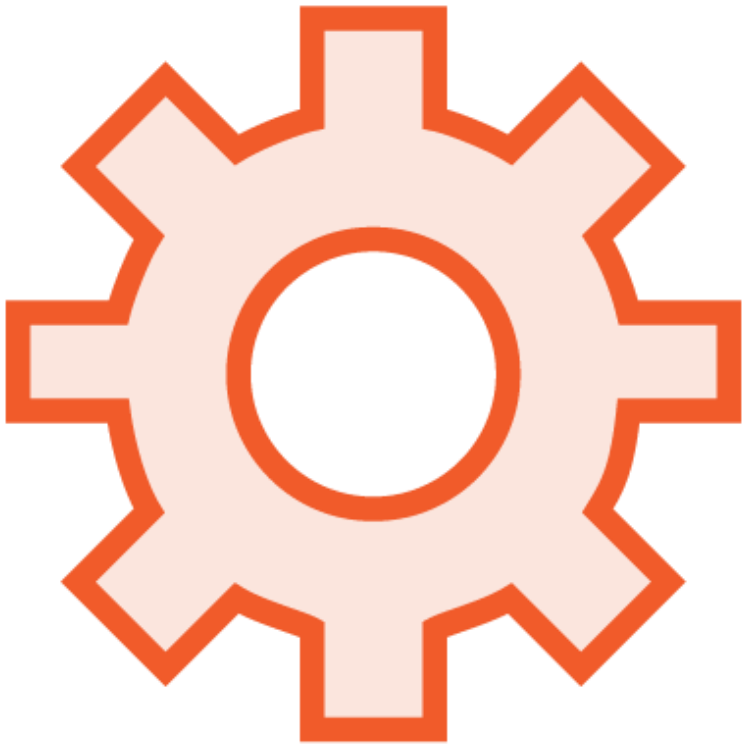
Unsubscribes the prior inner Observable and switches to the new inner Observable

Marble Diagram: `switchMap`

```
of('A1', 'A2')
 .pipe(
  switchMap(id =>
 this.http.get<Apple>(`${this.url}/${id}`))
 ).subscribe(console.log);
```



RxJS Operator: switchMap



switchMap is a transformation operator

- Takes in an input stream, subscribes
- Creates an output stream

When each item is emitted

- Item is mapped to an inner Observable as specified by a provided function
- Unsubscribes from prior inner Observable
- Subscribes to new inner Observable
- Inner Observable emissions are merged to the output stream

Use `switchMap`

To stop any prior Observable before switching to the next one

Examples:

- Type ahead or auto completion
- User selection from a list

Demo

`switchMap`

Higher-Order Observable

Source/outer Observable

Inner Observable

```
of('A1', 'A2')
 .pipe(
   mergeMap(id => this.http.get<Apple>(`${this.url}/${id}`))
 );
```

Higher-order mapping operator

Item emitted from outer Observable

{A1} {A2}

Higher-Order Mapping

Use higher-order mapping operators

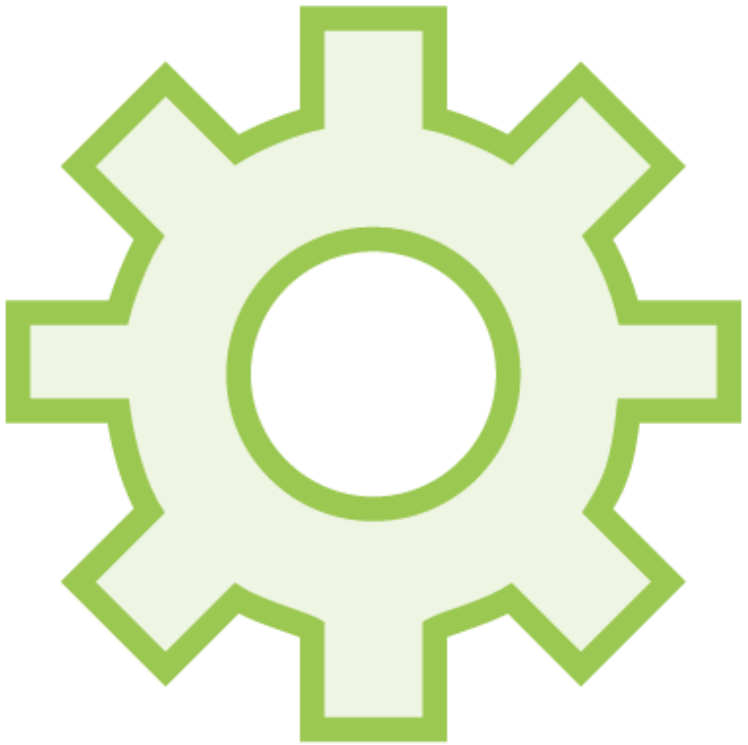
- To map emitted items to a new Observable
- Automatically subscribe to and unsubscribe from that Observable
- And emit the results to the output stream

Higher-order mapping operator functions

- Take in an item and return an Observable

Use instead of nested subscribes

Higher-Order Mapping Operators



`concatMap`

- Waits for inner Observable to complete before processing the next one

`mergeMap`

- Processes inner Observables in parallel

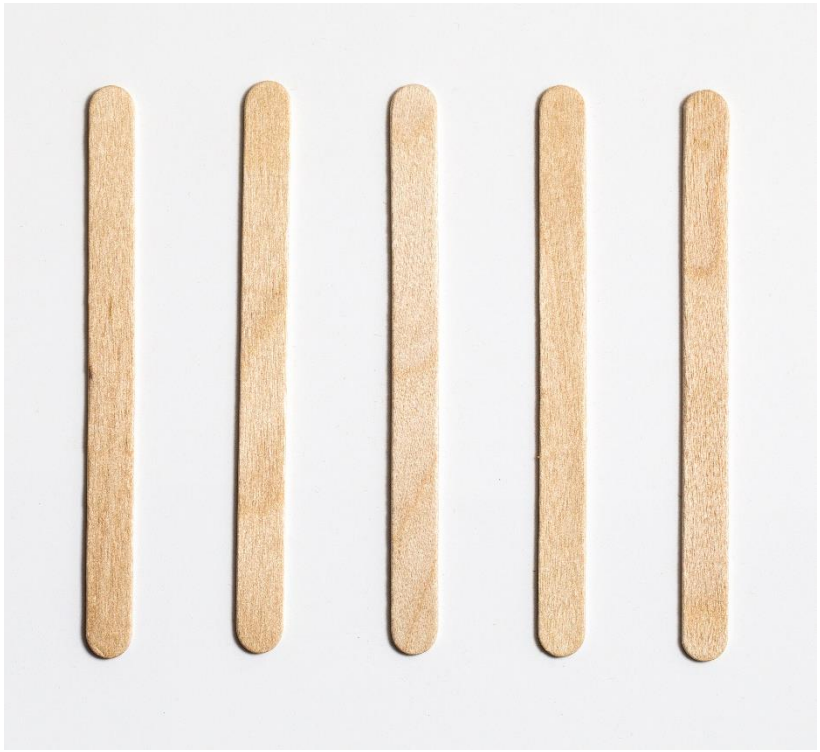
`switchMap`

- Unsubscribes from the prior inner Observable and switches to the new one

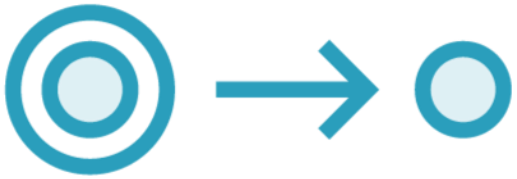
Combining Streams

Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR | MVP | GDE

@deborahkurata | blogs.msmvps.com/deborahk/

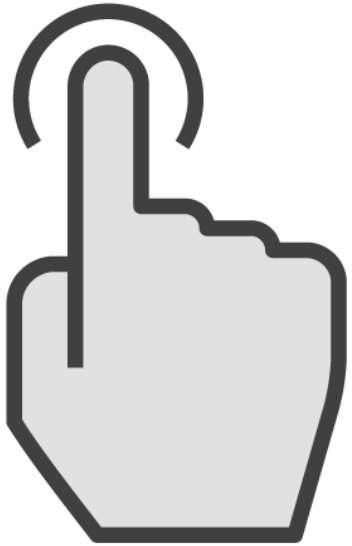


Why Combine Observable Streams?



Map id to a string

Work with multiple data sources



React to actions

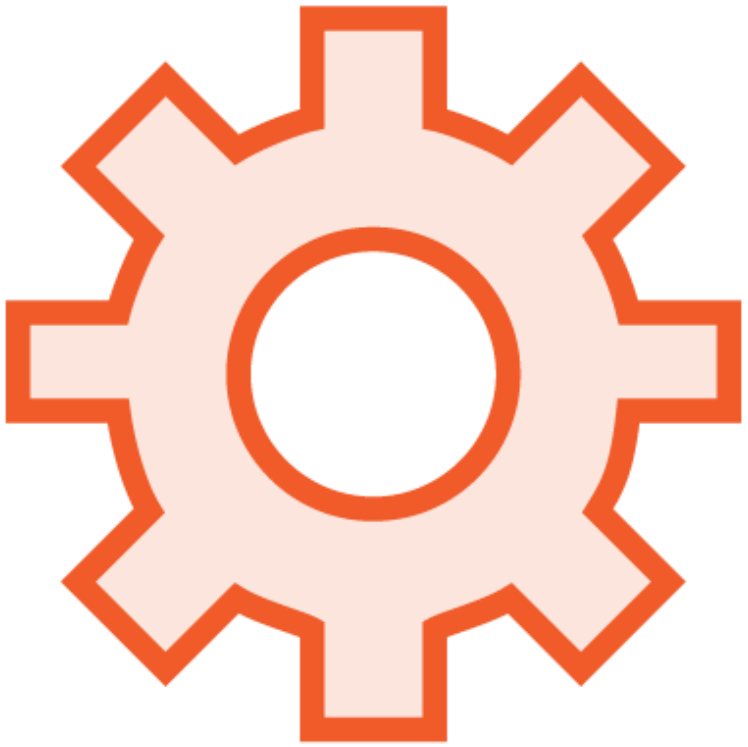
Simplify template code

Module Overview

Combination operators

Combining streams to map an id to a string

RxJS Features

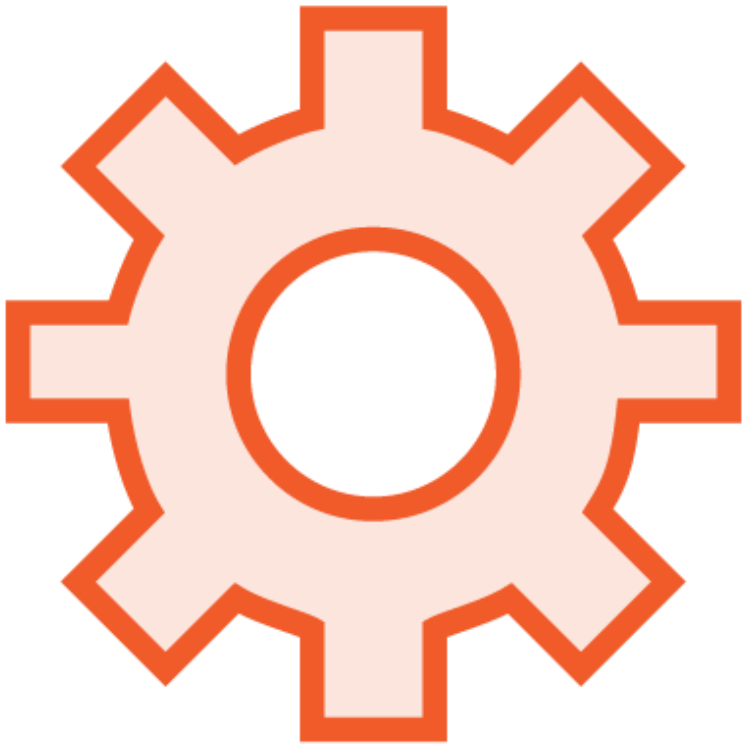


`combineLatest`

`forkJoin`

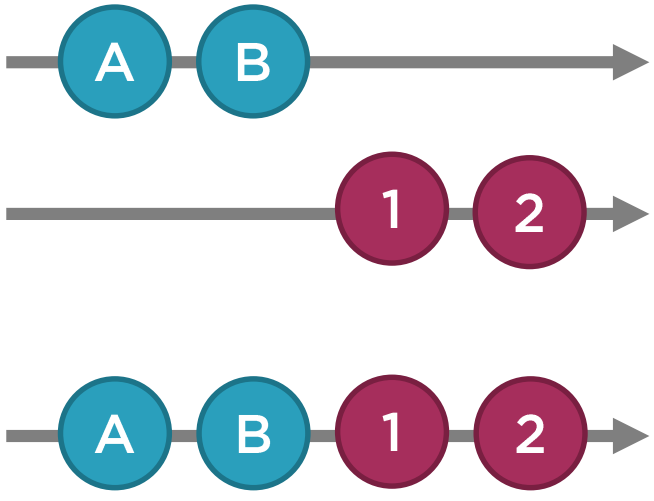
`withLatestFrom`

Combination Operators/Functions

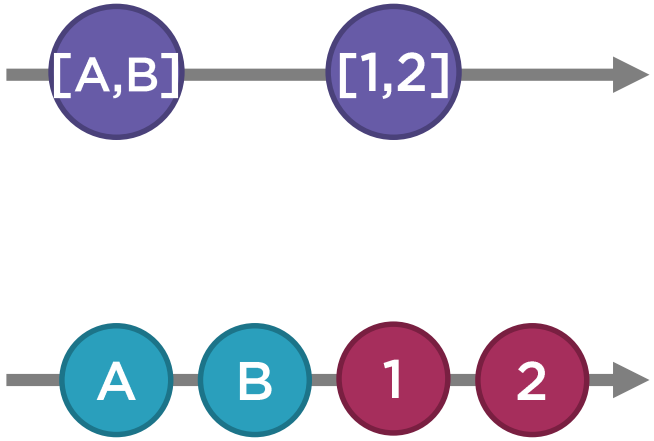


Combine information from multiple Observable streams

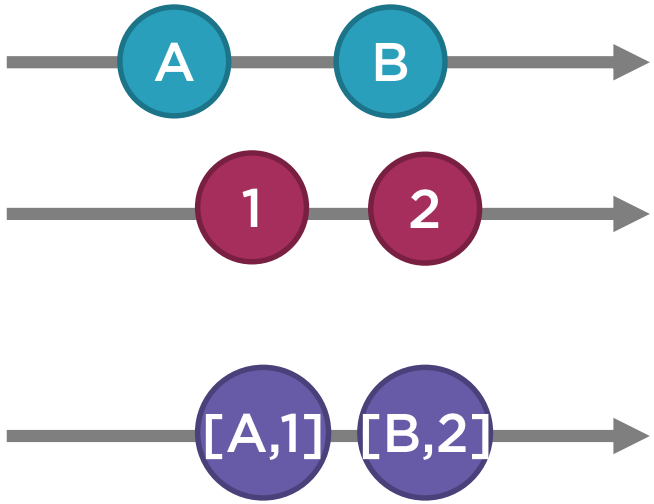
Types of Combination Operators/Functions



Combine to a single stream



Flatten higher-order Observables



Emit a combined value

RxJS Creation Function: `combineLatest`

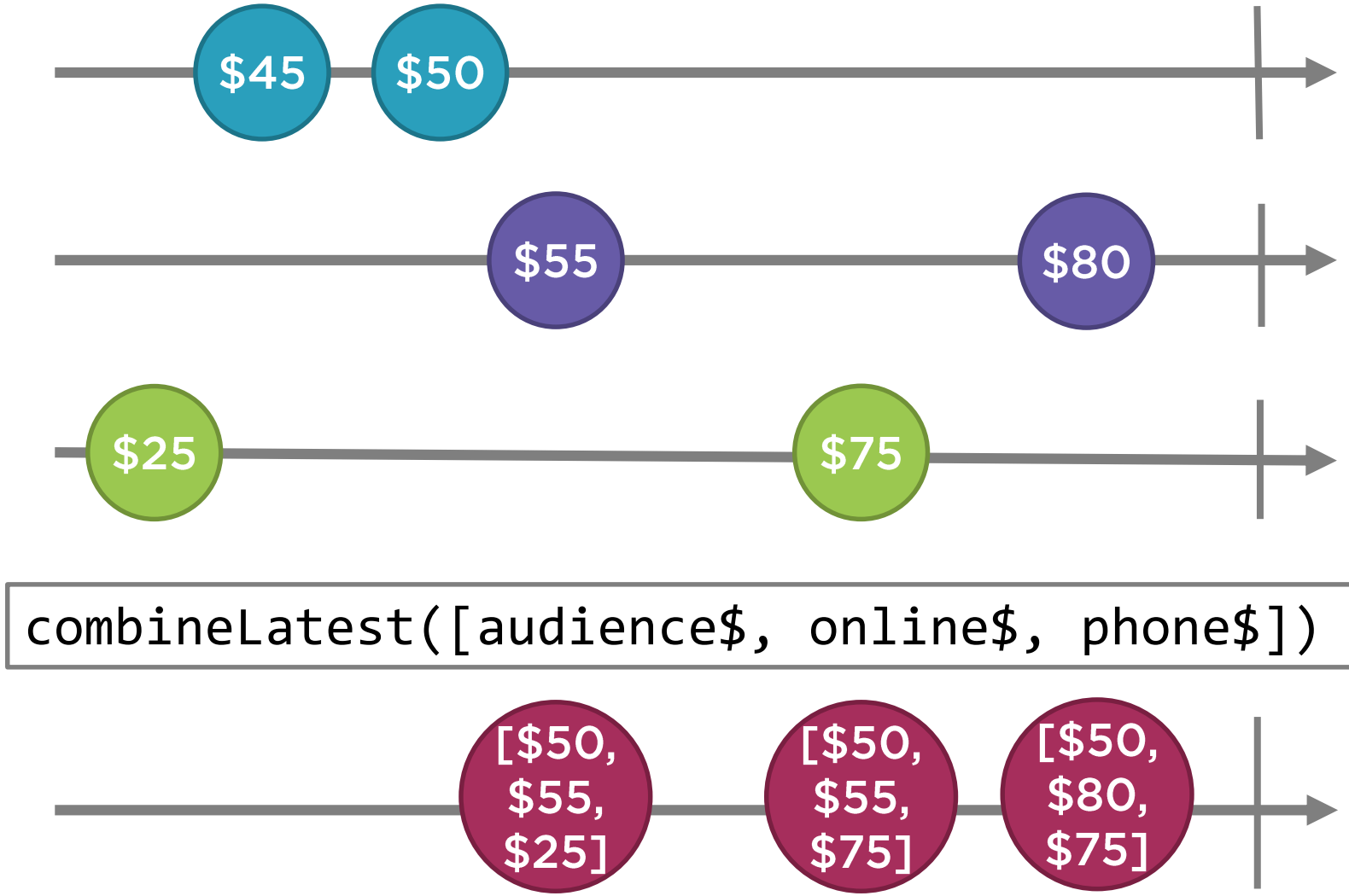
Creates an Observable whose values are defined:

- Using the latest values from each input Observable

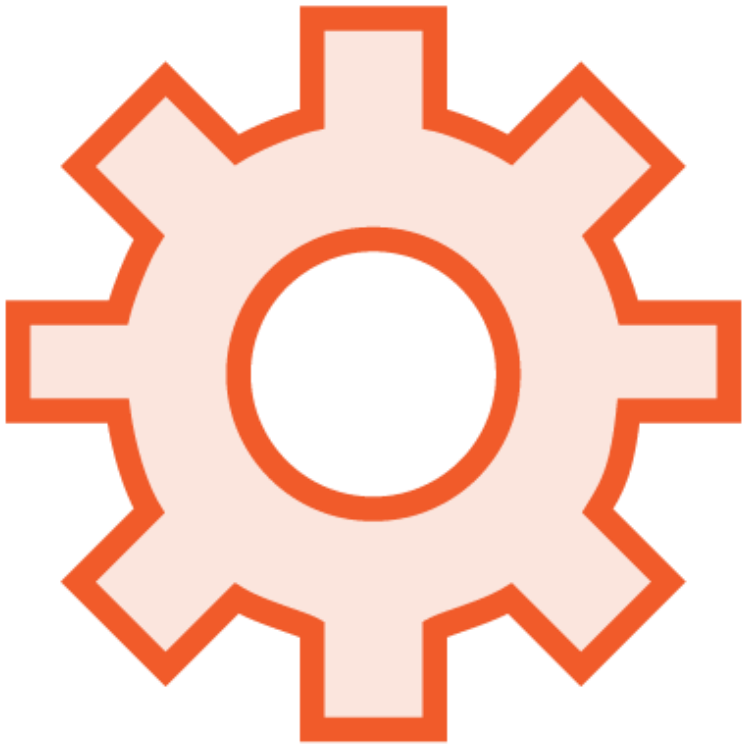
```
combineLatest([a$, b$, c$])
```

Static creation function, not a pipeable operator

Marble Diagram: `combineLatest`



RxJS Creation Function: `combineLatest`



`combineLatest` **is a combination function**

- Takes in a set of streams, subscribes
- Creates an output stream

When an item is emitted from any stream

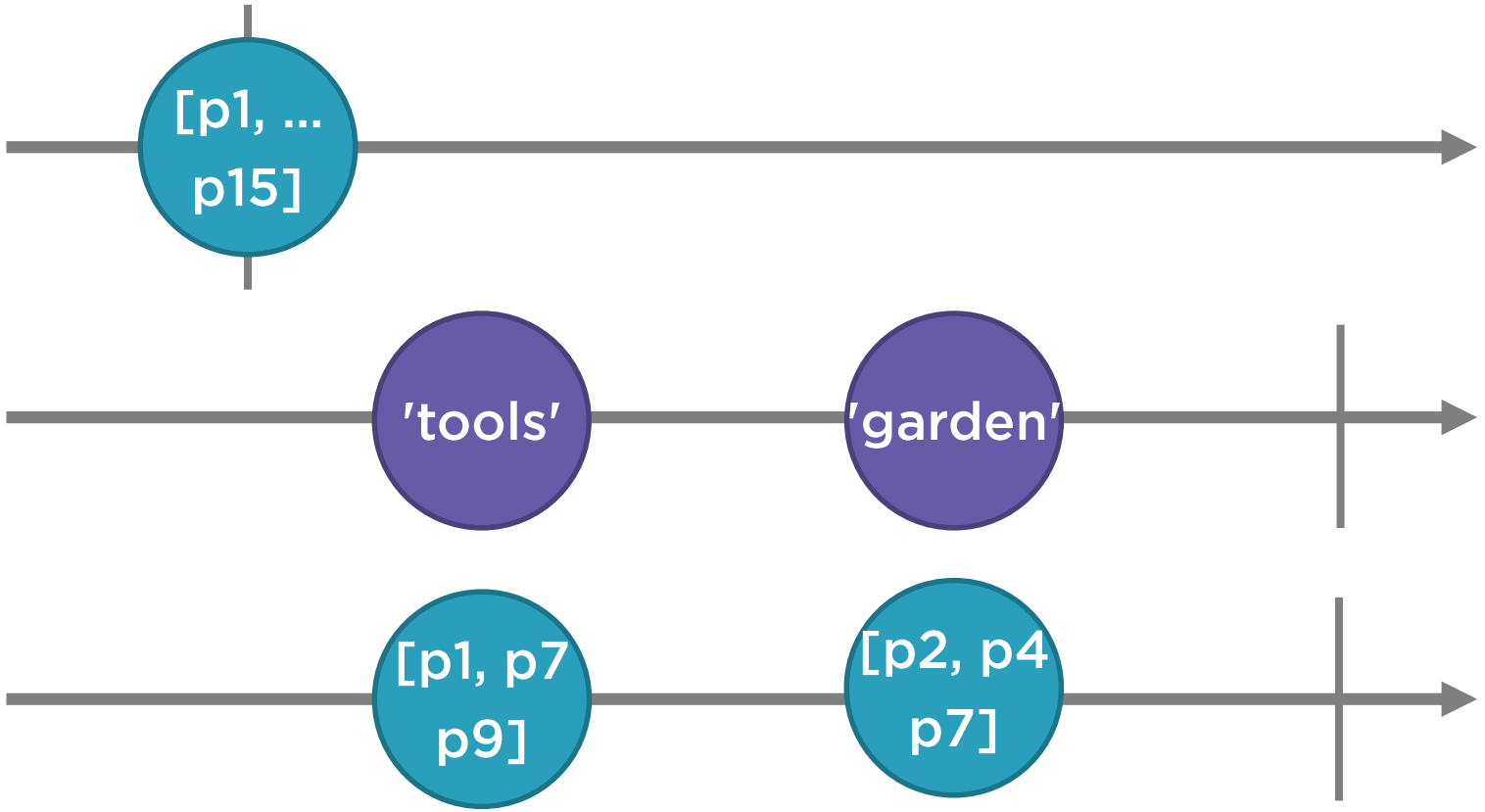
- If all streams have emitted at least once
- Emits a value to the output stream

Completes when all input streams complete

Emitted value combines the latest emitted value from each input stream into an array

Use `combineLatest`

To re-evaluate state when an action occurs



RxJS Creation Function: `forkJoin`

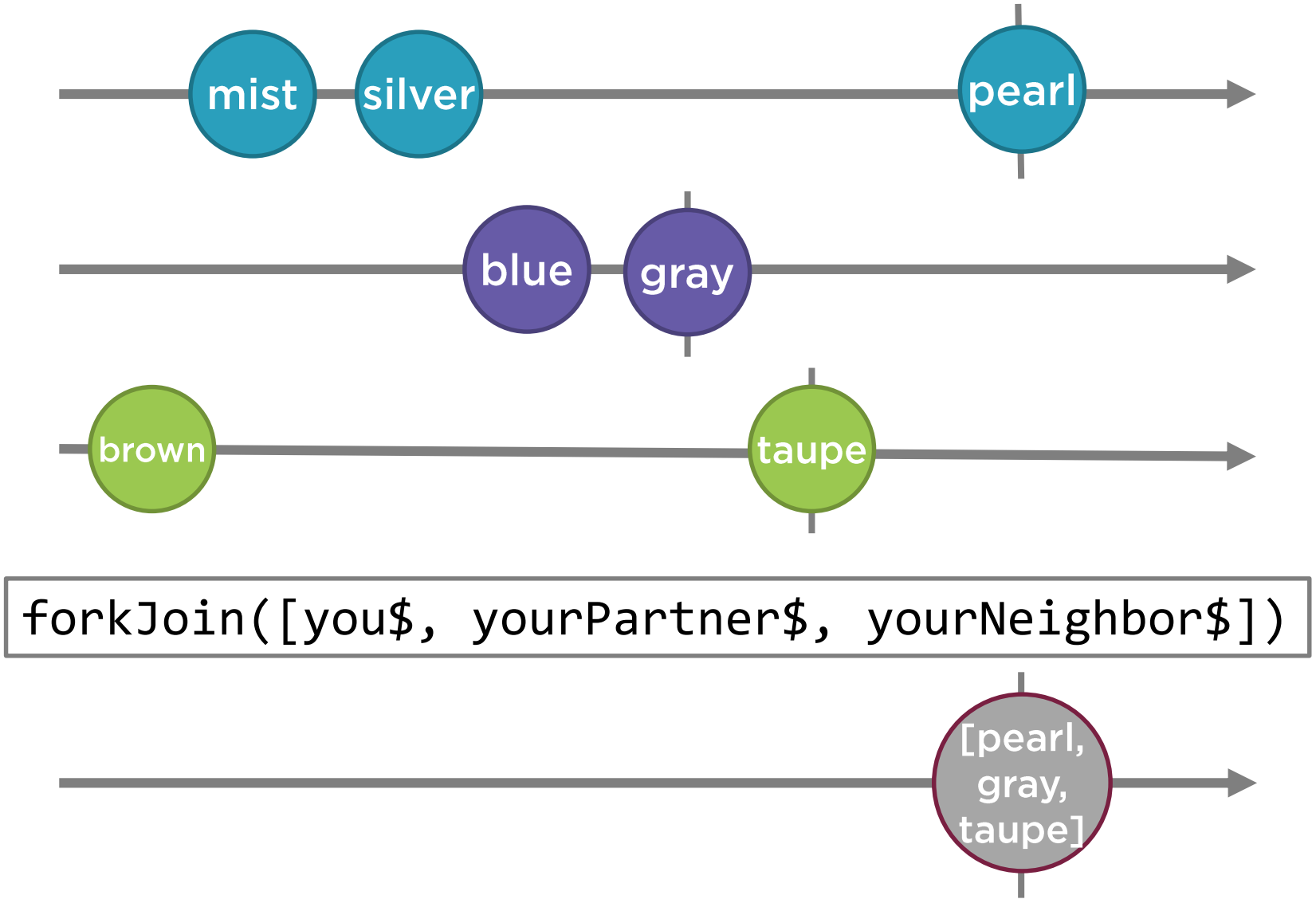
Creates an Observable whose value is defined

- Using the last value from each input Observable

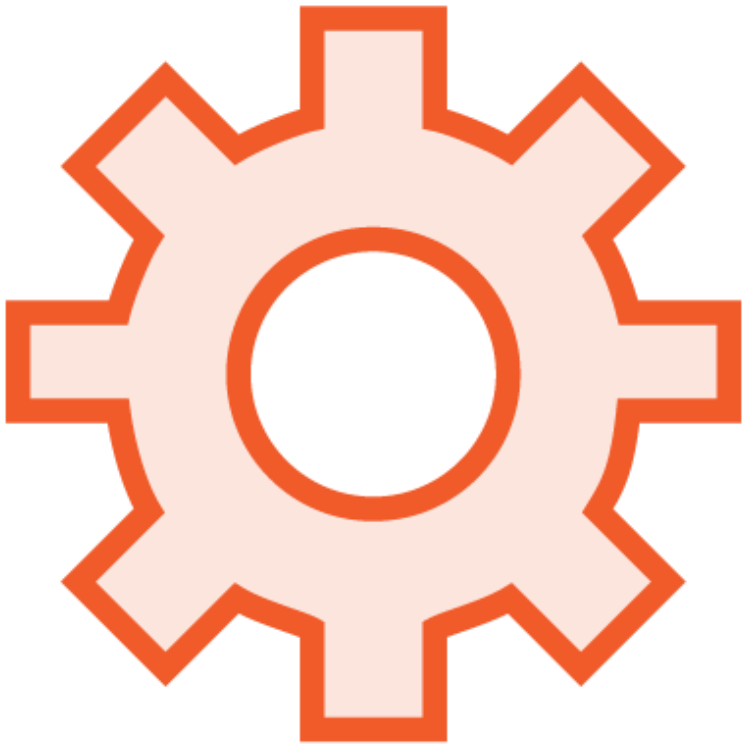
```
forkJoin([a$, b$, c$])
```

Static creation function, not a pipeable operator

Marble Diagram: `forkJoin`



RxJS Function: `forkJoin`



`forkJoin` **is a combination function**

- Takes in a set of streams, subscribes
- Creates an output stream

When all input streams complete

- Emits a value to the output stream
- And completes

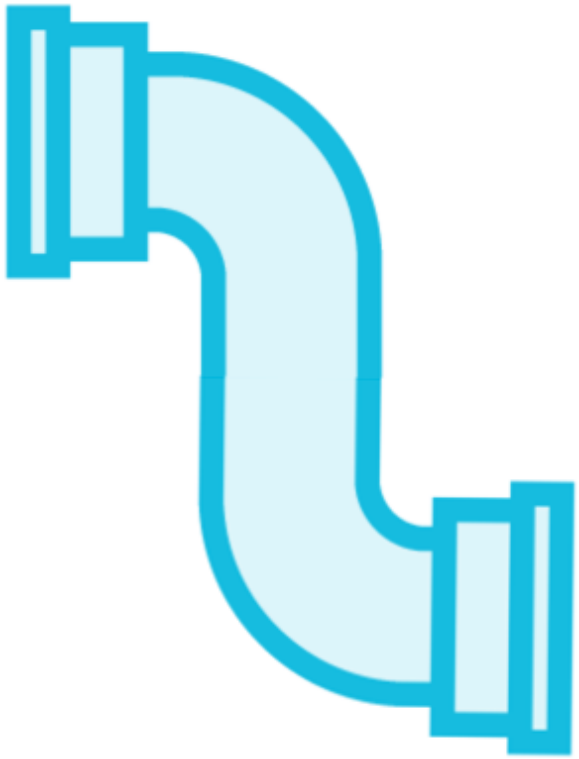
Emitted value combines the last emitted value from each input stream into an array

Use `forkJoin`

To wait to process any results until all streams are complete

Don't use when working with streams that don't complete

RxJS Operator: `withLatestFrom`



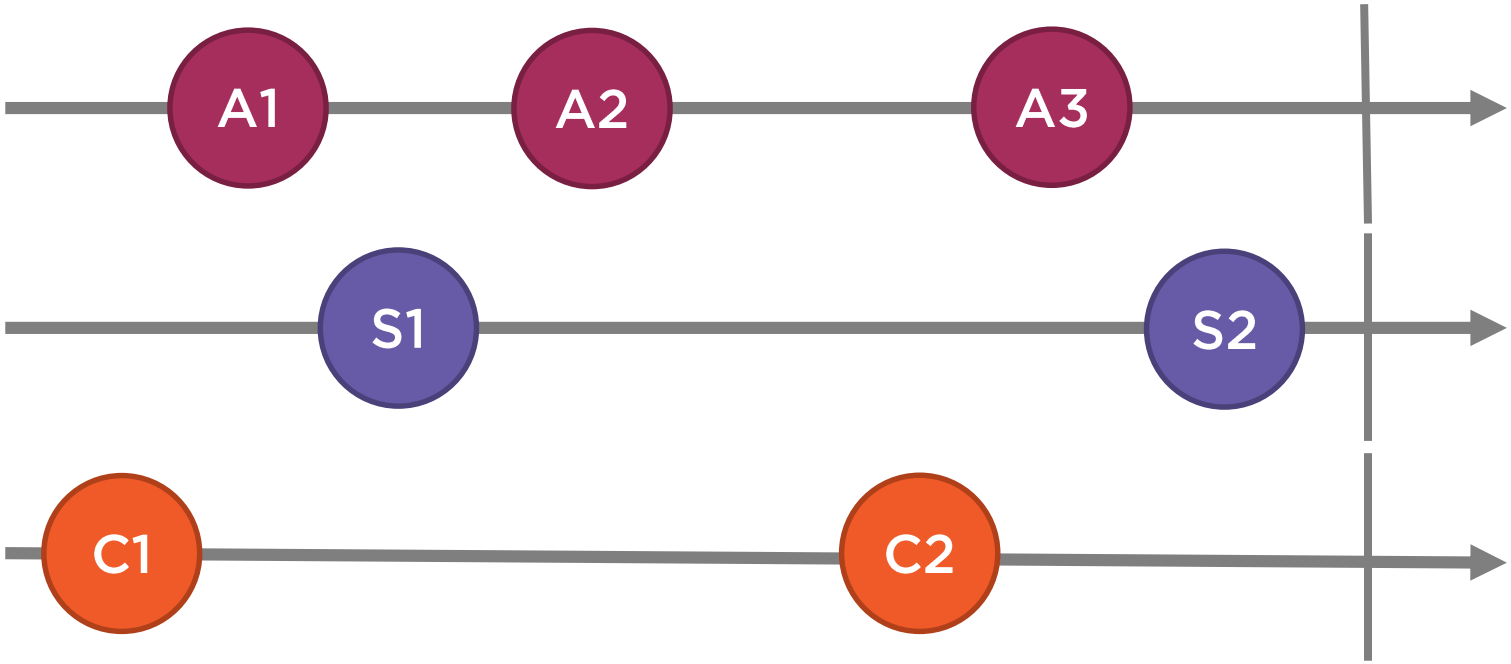
Creates an Observable whose values are defined

- Using the latest values from each input Observable
- But only when the source stream emits

```
a$.pipe(withLatestFrom(b$, c$))
```

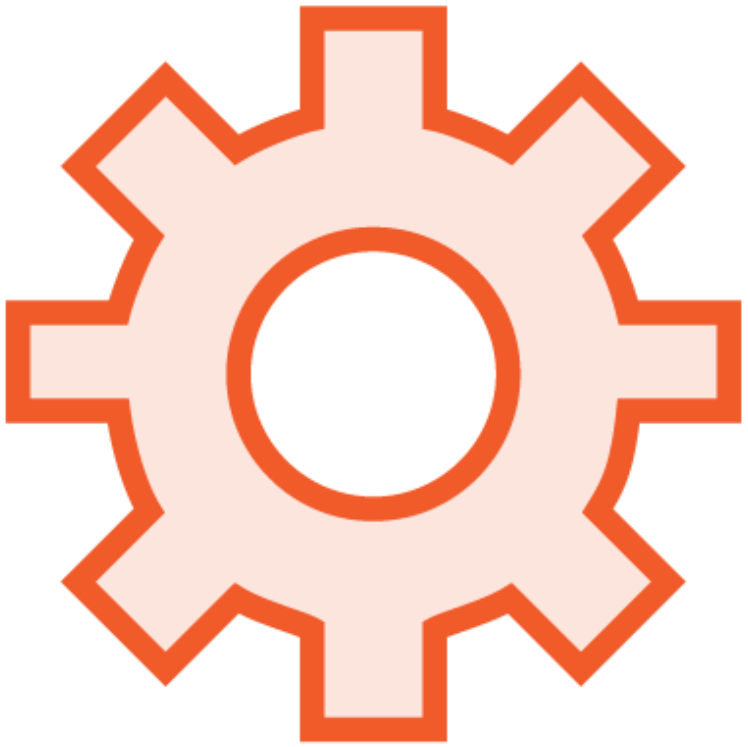
Pipeable operator

Marble Diagram: `withLatestFrom`




```
apples$.pipe(withLatestFrom(sticks$, caramel$))
```

RxJS Operator: `withLatestFrom`



`withLatestFrom` **is a combination operator**

- Takes in a set of streams, subscribes
- Creates an output stream

When an item is emitted from source stream

- If all streams have emitted at least once
- Emits a value to the output stream

Completes when the source stream completes

Emitted value combines the latest emitted value from each input stream into an array

Use `withLatestFrom`

To react to changes in only one stream

To regulate the output of the other streams

Acme Product Management Home Product List Product List (Alternate UI)

Product List

- Display All -

Add Product

Product	Code	Category	Price	In Stock
Leaf Rake	GDN-0011	1	$29.92	15
Garden Cart	GDN-0023	1	$49.49	2
Hammer	TBX-0048	3	$13.35	8
Saw	TBX-0022	3	$17.33	6
Video Game Controller	GMG-0042	5	$53.93	12

Acme Product Management Home Product List Product List (Alternate UI)

Product List

- Display All -

Add Product

Product	Code	Category	Price	In Stock
Leaf Rake	GDN-0011	Garden	$29.92	15
Garden Cart	GDN-0023	Garden	$49.49	2
Hammer	TBX-0048	Toolbox	$13.35	8
Saw	TBX-0022	Toolbox	$17.33	6
Video Game Controller	GMG-0042	Gaming	$53.93	12

Mapping an Id to a String

```
export interface Product {
 id: number;
 productName: string;
 productCode?: string;
 description?: string;
 price?: number;
 categoryId?: number;
}
```

```
export interface Product {
 id: number;
 productName: string;
 productCode?: string;
 description?: string;
 price?: number;
 categoryId?: number;
 category?: string;
}
```

Mapping an Id to a String

```
products$=this.http.get<Product[]>(this.url)
 .pipe(
   map(products =>
    products.map(product => ({
      ...product,
      price: product.price * 1.5,
      category: ???
   }) as Product)
 ));
```

```
category: this.getCategory(product.categoryId)
```

```
productCategories$ = this.http.get<ProductCategory[]>(this.productCategoriesUrl);
```

Combining the Streams

```
productsWithCategory$ = combineLatest([
 this.products$,
 this.productCategories$
]);
```

[product[], category[]]

```
productsWithCategory$ = forkJoin([
 this.products$,
 this.productCategories$
]);
```

[product[], category[]]

```
productsWithCategory$ = this.products$
 .pipe(
   withLatestFrom(this.productCategories$)
 );
```

[product[], category[]]

Mapping an Id to a String

```
product$ = this.http.get<Product[]>(this.url);
```

```
productCategories$ = this.http.get<ProductCategory[]>(this.productCategoriesUrl);
```

```
productsWithCategory$ = combineLatest([
 this.products$,
 this.productCategories$
])
 .pipe(
   map(([products, categories]) =>
    products.map(product => ({
      ...product,
      price: product.price * 1.5,
      category: categories.find(
       c => product.categoryId === c.id
      ).name
   }) as Product))
 );
```

Mapping an Id to a String

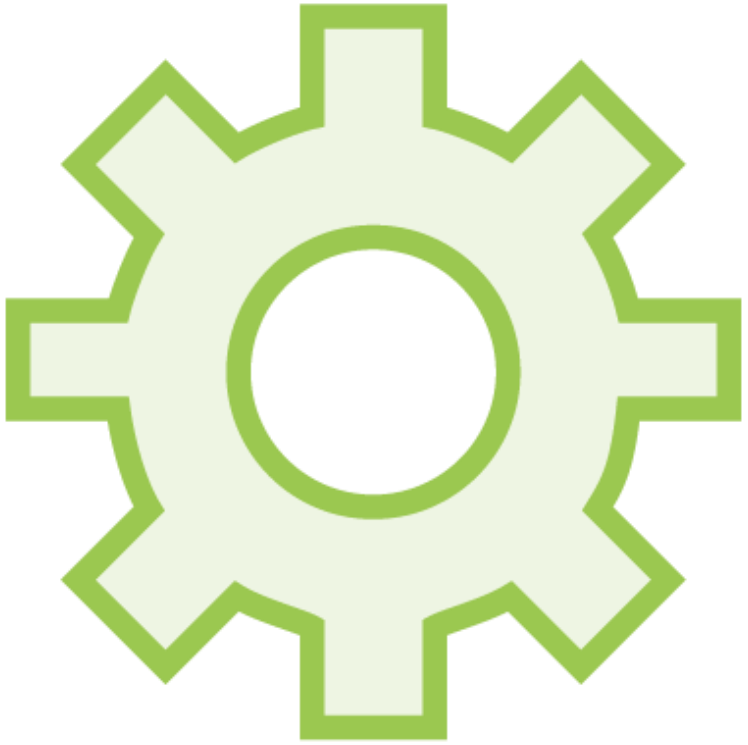
```
productsWithCategory$ = combineLatest([
 this.products$,
 this.productCategories$
])
 .pipe(
   map(([products, categories]) =>
    products.map(product => ({
     ...product,
     price: product.price * 1.5,
     category: categories.find(
      c => product.categoryId === c.id
     ).name
   }) as Product))
 );
```

[product[], category[]]

Demo

Combining streams
to map an id to a string

Combining Observable Streams



combineLatest: Emits any time a new value is emitted from **any** of them

```
combineLatest([a$, b$, c$])
```

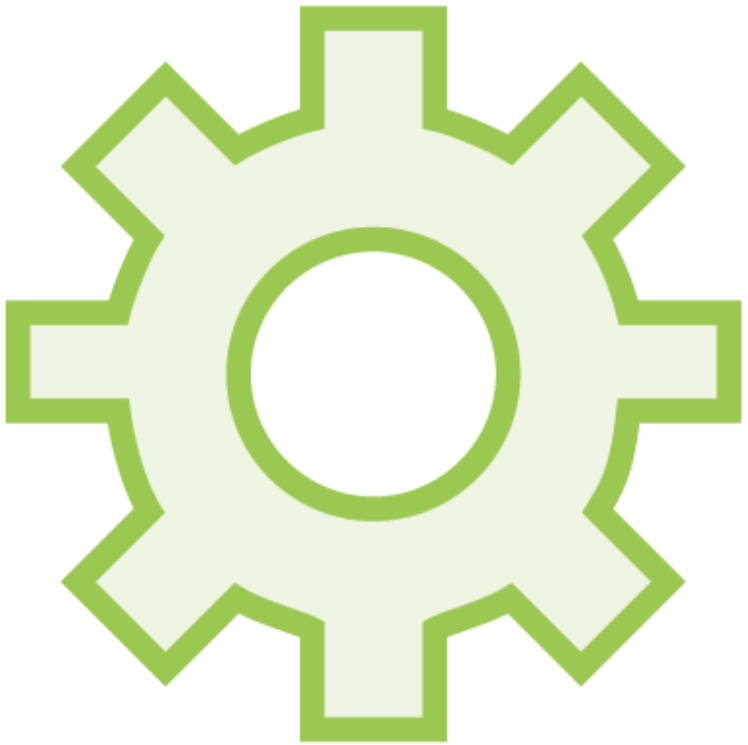
forkJoin: Emits only the **last** emitted values

```
forkJoin([a, b$, c$])
```

withLatestFrom: Emits any time a new value is emitted from the **source** stream

```
a$.pipe(withLatestFrom(b$, c$))
```

Emitted Item



```
combineLatest([a$, b$, c$])

forkJoin([a$, b$, c$])

a$.pipe(withLatestFrom(b$, c$))
```

```
[a1, b1, c1]
```

Mapping an Id to a String

Combine the streams

Map the items

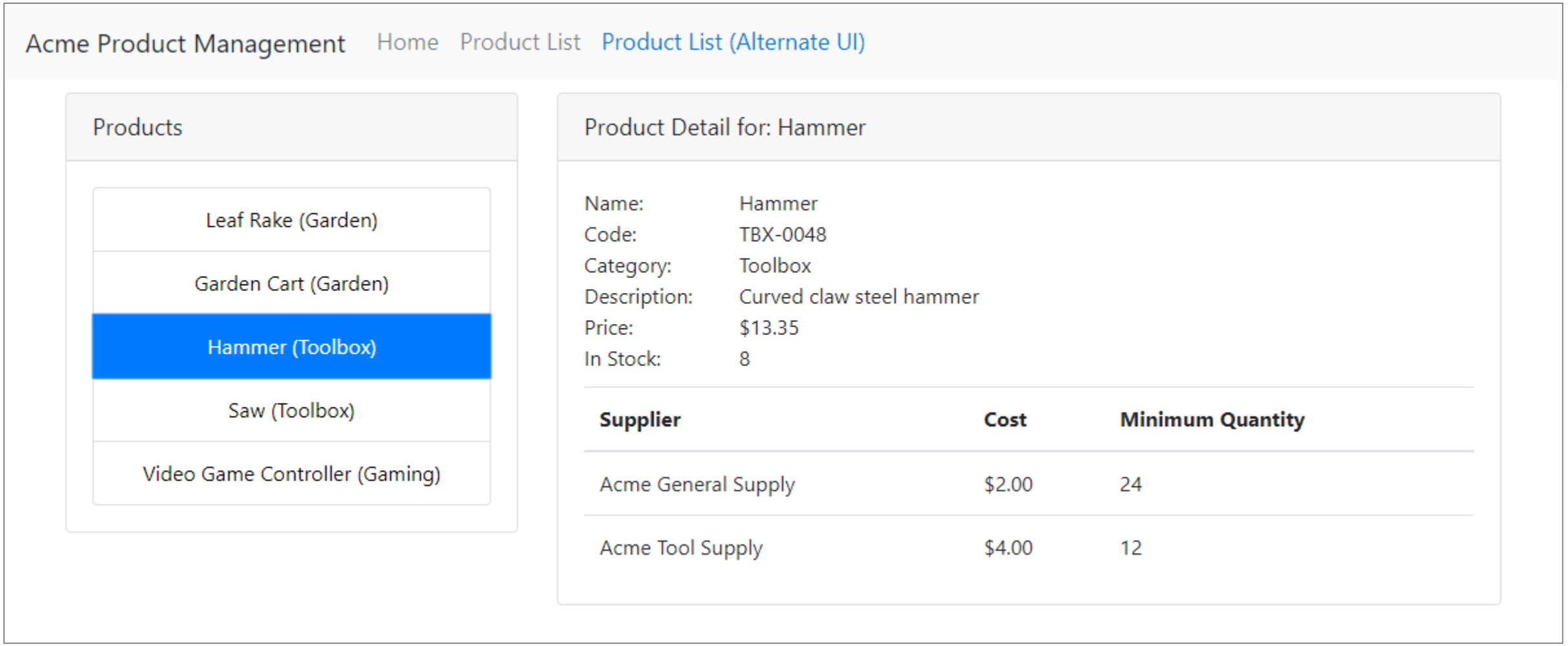
```
productsWithCategory$ = combineLatest(
 this.products$,
 this.productCategories$
)
 .pipe(
   map([products, categories] =>
    products.map(product => ({
     ...product,
     category: categories.find(
      c => product.categoryId === c.id
     ).name
   }) as Product))
 );
```

Combining All the Streams

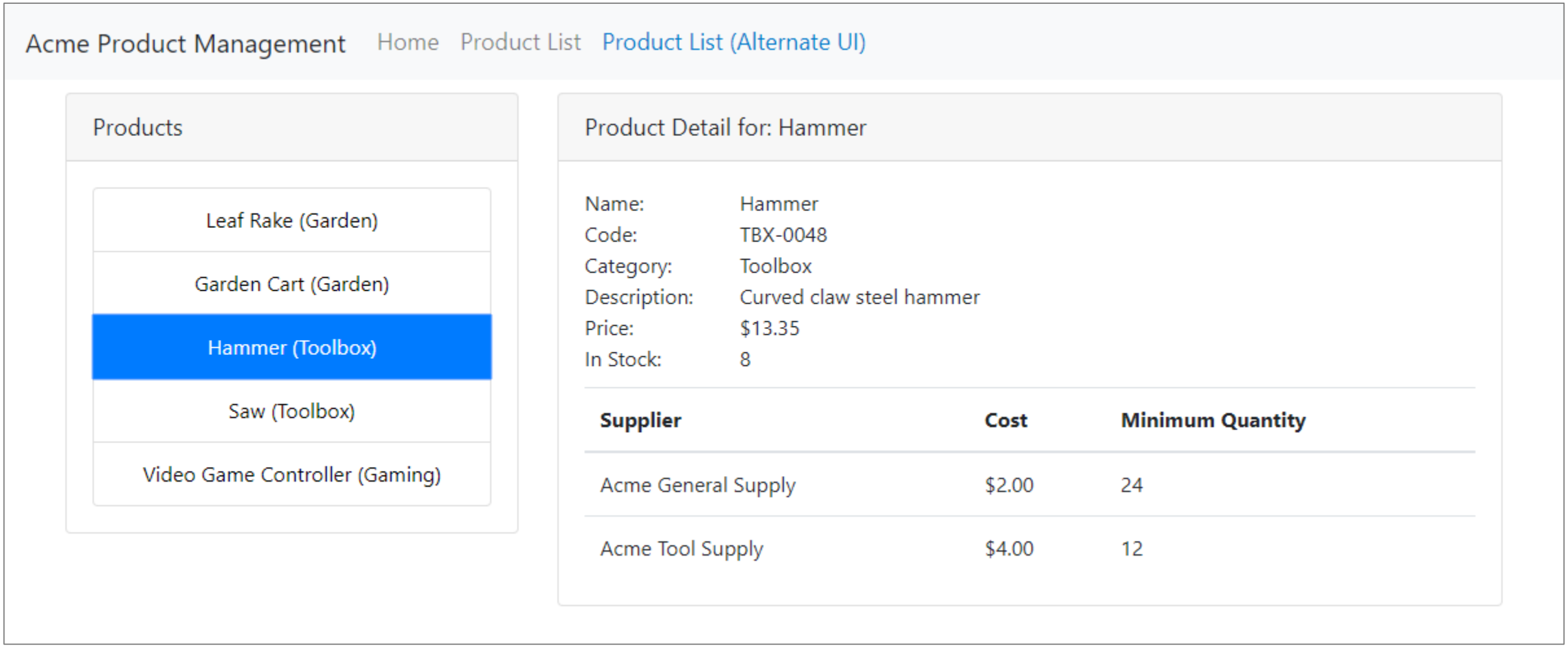
Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR | MVP | GDE

@deborahkurata | blogs.msmvps.com/deborahk/

Multiple Data Sources



Creating Ancillary Streams



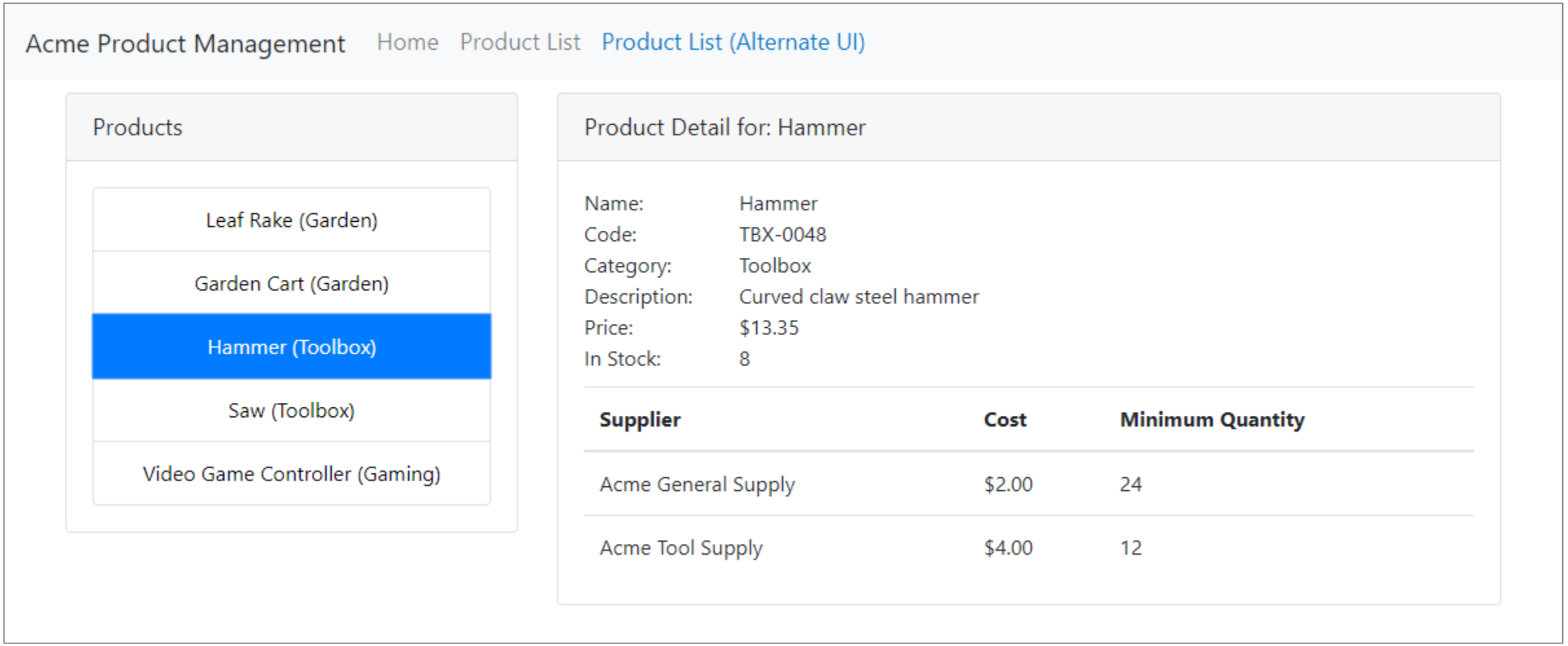
Module Overview

Handling related data

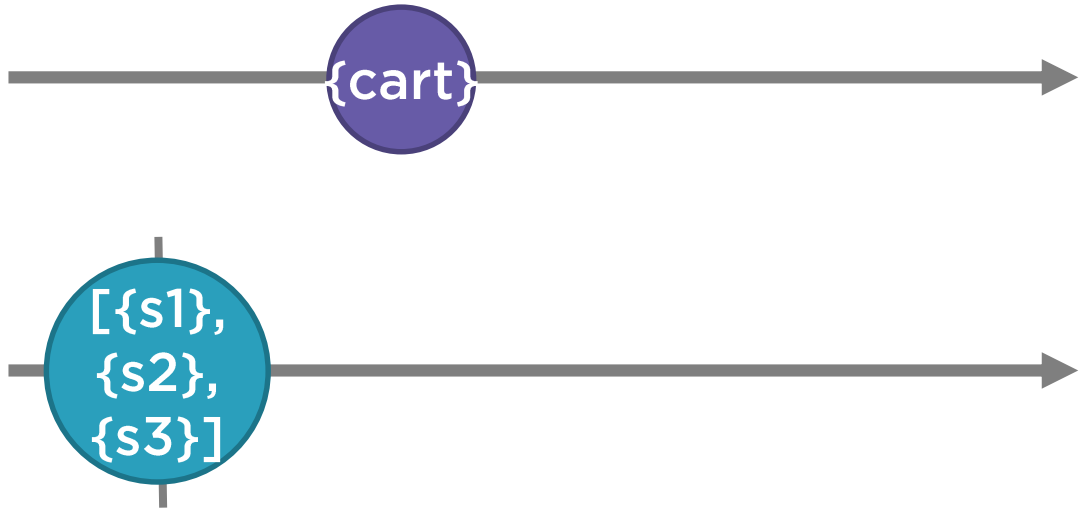
Creating ancillary streams

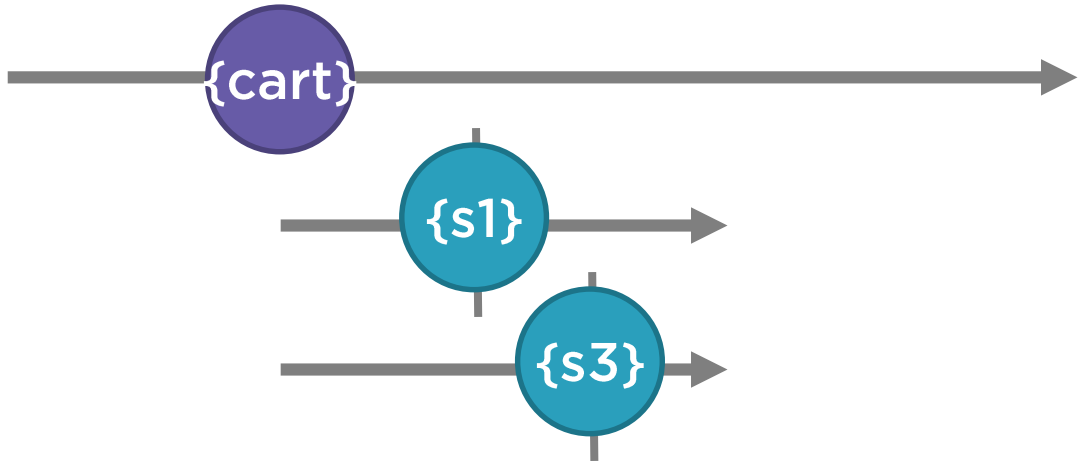
Combining all the streams

Related Data Streams



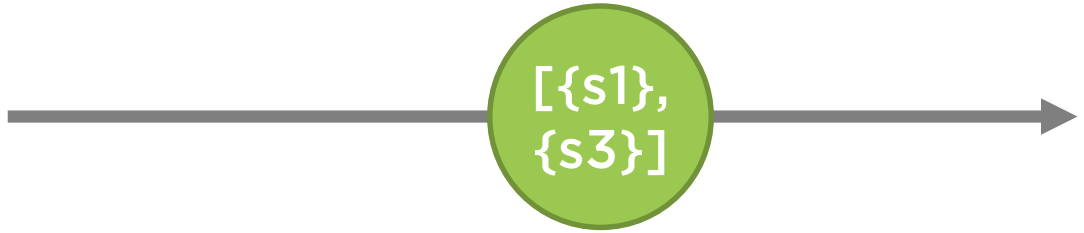
Related Data Streams






```
combineLatest(product$,
              suppliers$)
```

```
mergeMap(...)
```



Get it all

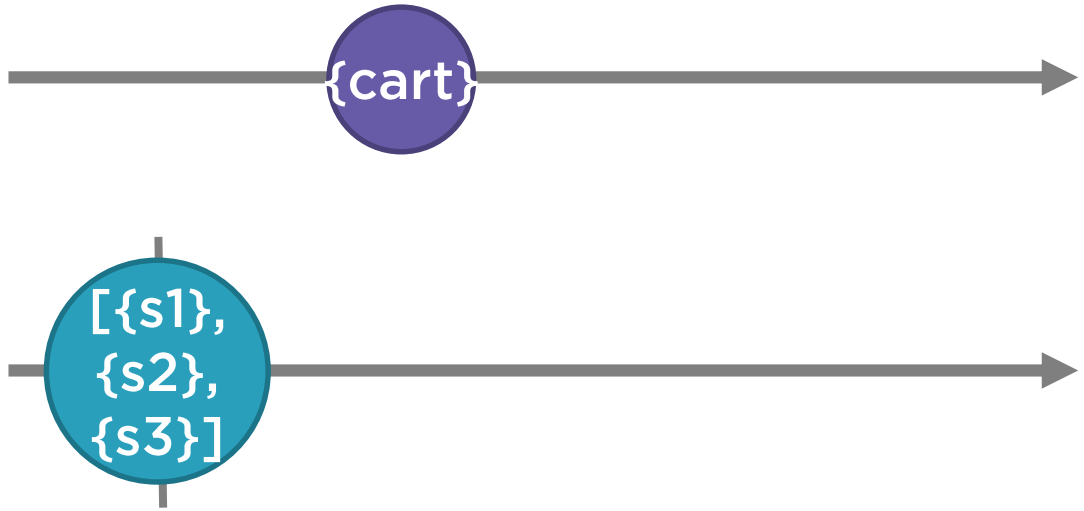
Just in time

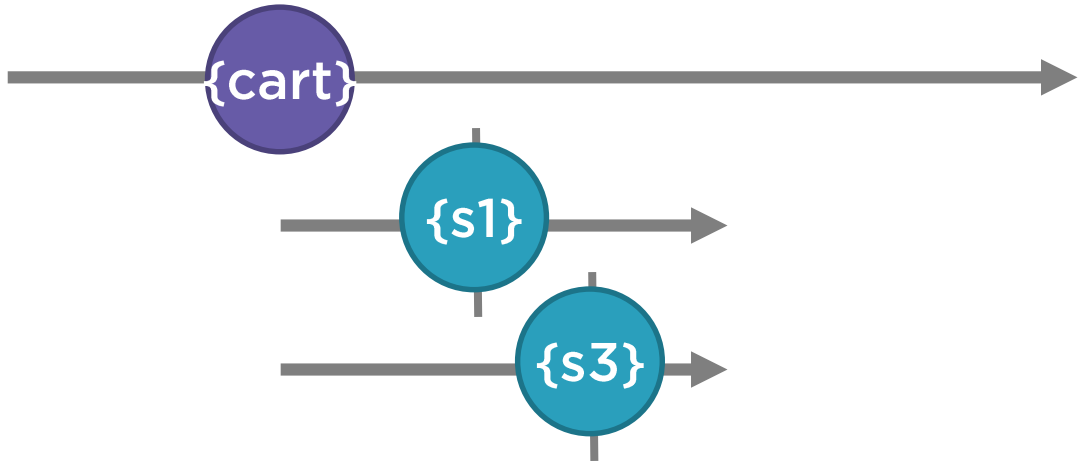
Demo

Related data streams: get it all

- Get all data from the related dataset
- Find the related items in that dataset

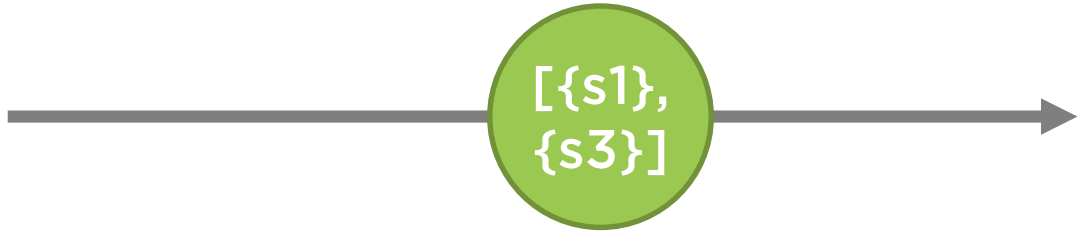
Related Data Streams






```
combineLatest(product$,
              suppliers$)
```

```
mergeMap(...)
```

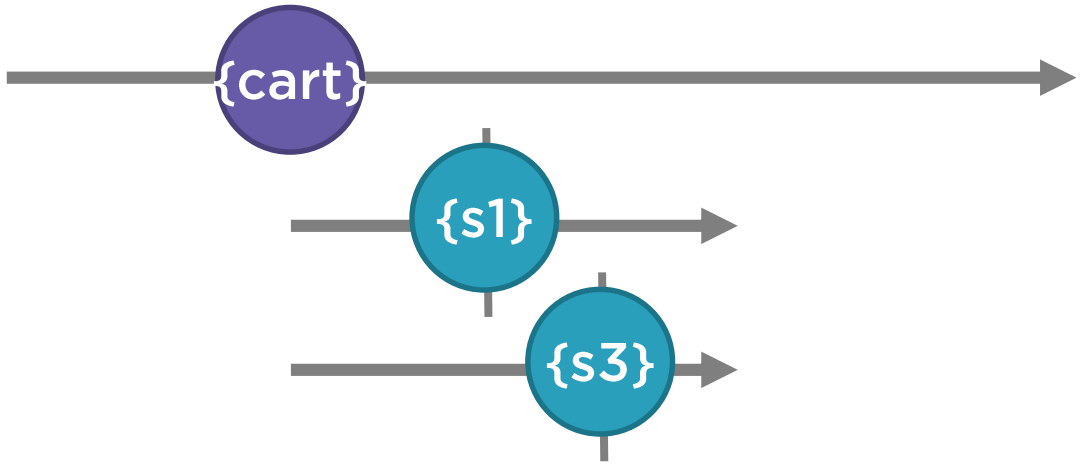


Get it all

Just in time

Related Data Stream: Just in Time

```
export interface Product {
  id: number;
  productName: string;
  productCode?: string;
  description?: string;
  price?: number;
  categoryId?: number;
  category?: string;
  quantityInStock?: number;
  searchKey?: string[];
  supplierIds?: number[];
}
```



Related Data Stream: Just in Time

```
supplierWithMergeMap$ = of(1, 5, 8)
 .pipe(
   mergeMap(supplierId => this.http.get<Supplier>(`${this.url}/${supplierId}`))
 );
```

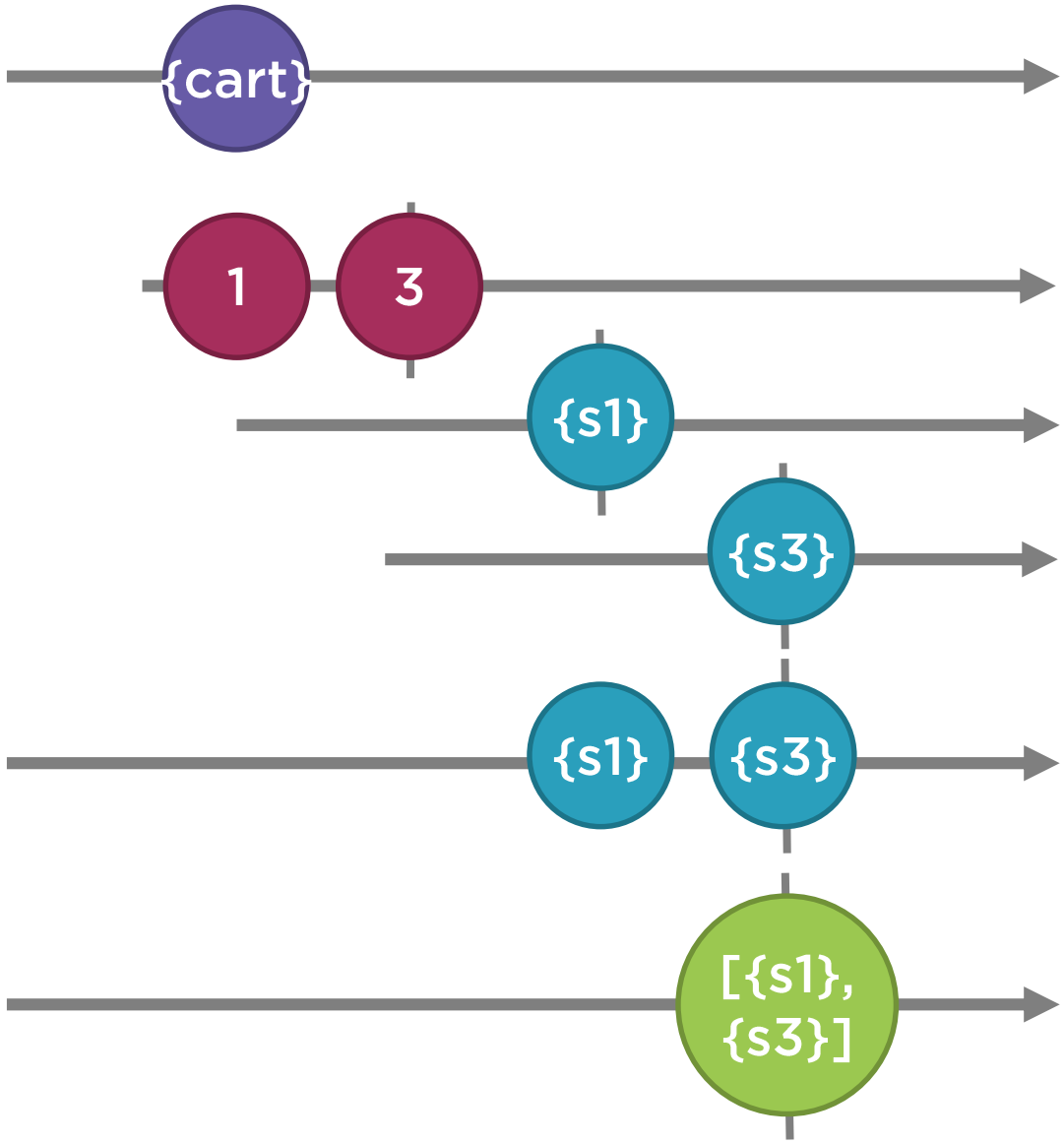
```
selectedProductSuppliers$ = this.selectedProduct$
 .pipe(
   mergeMap(product =>
    from(product.supplierIds)
     .pipe(
       mergeMap(supplierId => this.http.get<Supplier>(`${this.url}/${supplierId}`)),
       toArray()
 )));
```

Related Data Stream: Just in Time

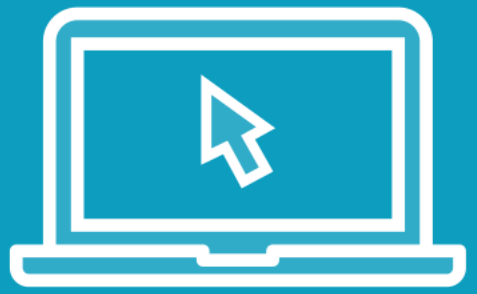
```
this.selectedProduct$
 .pipe(
   mergeMap(product =>
    from(product.supplierIds)
     .pipe(
       mergeMap(supplierId =>
         this.http.get<Supplier>(…)),
       toArray()
 )));
```

Related Data Stream: Just in Time

```
this.selectedProduct$
 .pipe(
   mergeMap(product =>

    from(product.supplierIds)
     .pipe(


       mergeMap(supplierId =>
         this.http.get<Supplier>(…)),




       toArray()
 )));
```



Demo

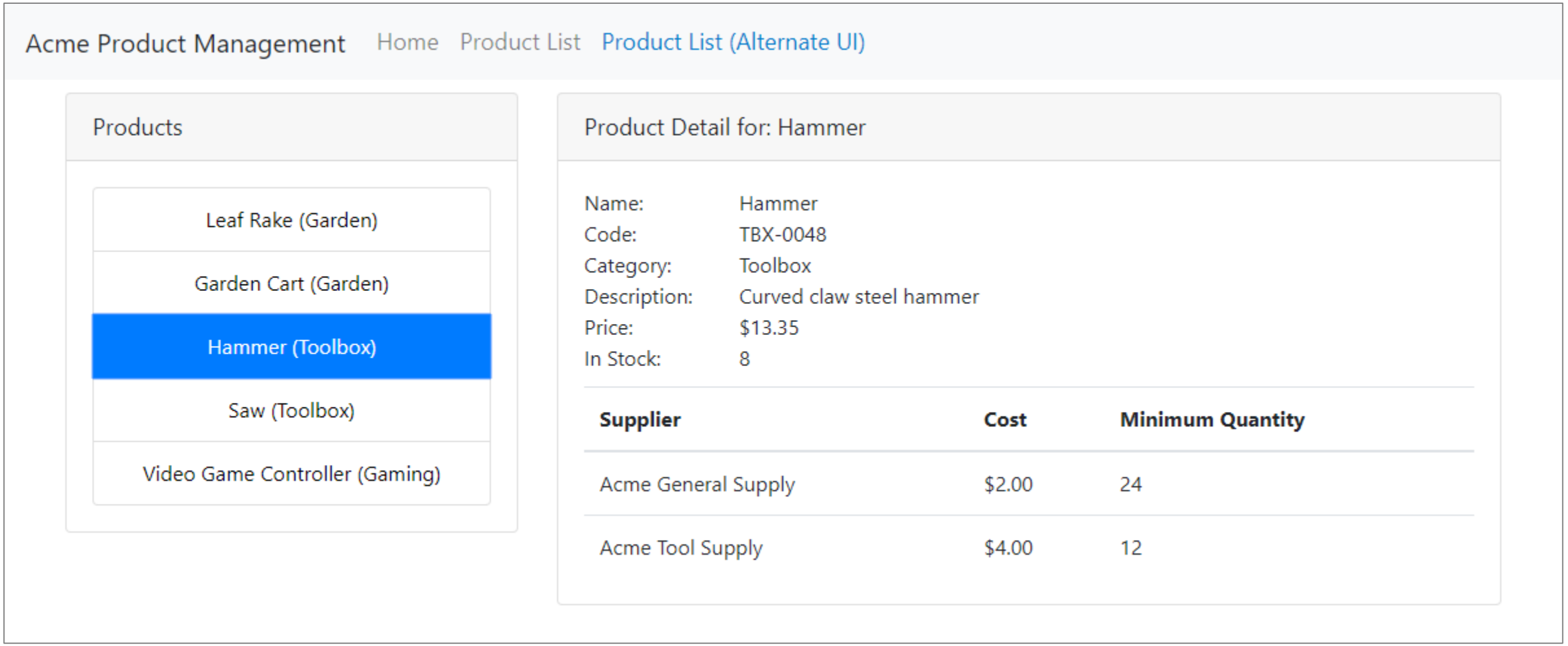


Related data streams: just in time

Related Data Stream

Get It All	Just in Time
Declarative pattern	More complex code
Combine streams	Higher-order mapping operators
Displays instantly	Display delay
Gets all data	Only retrieves required data

Defining Ancillary Streams



Defining Ancillary Streams

```
pageTitle$ = this.product$
  .pipe(
    map((p: Product) =>
          p ? `Product Detail for: ${p.productName}` : null)
  );
```

Multiple Async Pipes

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

```
<table *ngIf="productSuppliers$ | async as productSuppliers">
```

```
<div *ngIf="pageTitle$ | async as pageTitle">
  {{pageTitle}}
</div>
```

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

Related Data Streams

Get it all

- Gets all data from the related dataset
- Filters that dataset to find the related data

Just in time

- Gets just the related data items as needed

Related Data Streams: Get It All

```
selectedProductSuppliers$ = combineLatest(
 this.selectedProduct$,
 this.supplierService.suppliers$
).pipe(
  map(([selectedProduct, suppliers]) =>
    suppliers.filter(supplier =>
      selectedProduct.supplierIds.includes(supplier.id)
)));
```

Related Data Streams: Just in Time

```
selectedProductSuppliers$ = this.selectedProduct$
 .pipe(
    filter(product => Boolean(product)),
    switchMap(product =>
      from(product.supplierIds)
        .pipe(
          mergeMap(supplierId =>
this.http.get<Supplier>(`${url}/${supplierId}`)),
          toArray()
)));
```

Combining All the Streams

```
vm$ = combineLatest([
 this.product$,
 this.productSuppliers$,
 this.pageTitle$
])
 .pipe(
   map(([product, productSuppliers, pageTitle]) =>
      ({ product, productSuppliers, pageTitle }))
 );
```

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

Caching Observables

Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR | MVP | GDE

@deborahkurata | blogs.msmvps.com/deborahk/


Alarm
Net
HDD
IR
Esc
Fn
Enter
100-240VAC

Caching Observables

Retain retrieved data locally

Reuse previously retrieved data

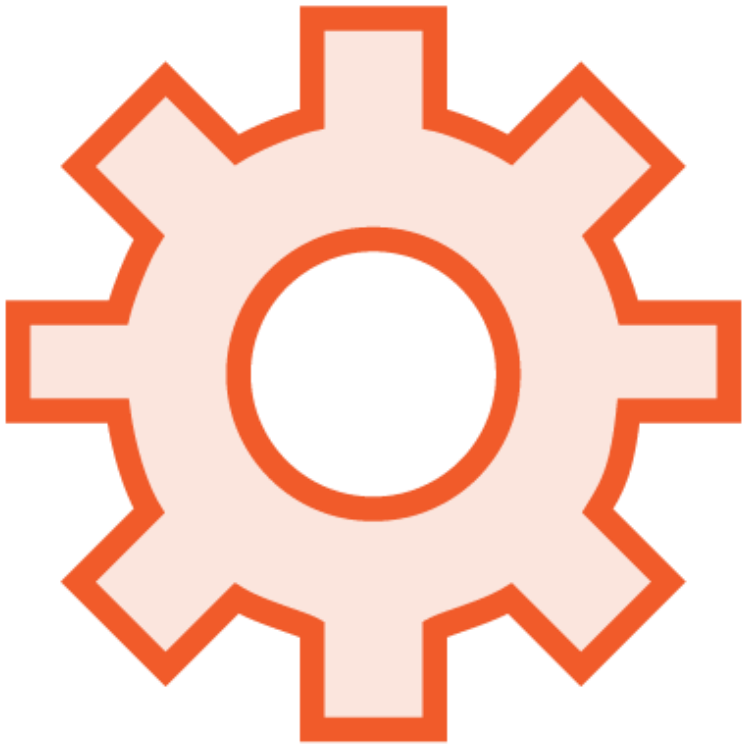
Stored in memory or external

Module Overview

Why caching?

Patterns for data caching

RxJS Features



`shareReplay`

Retrieving Data

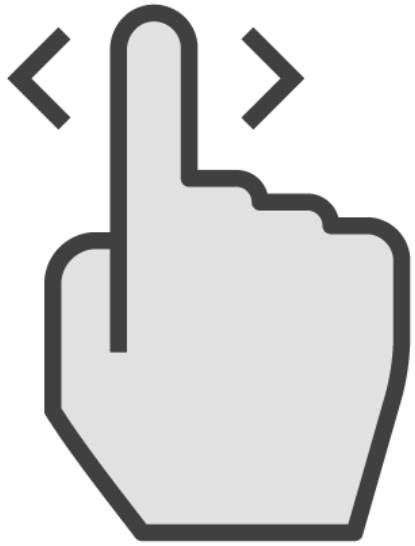
Product List Component

```
constructor(private productService: ProductService) { }

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

Product List Template

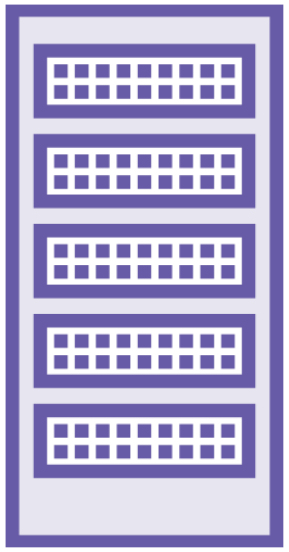
```
<div *ngIf="products$ | async as products">
<table>
  <tr *ngFor="let product of products">
    <td>{{ product.productName }}</td>
    <td>{{ product.productCode }}</td>
  </tr>
</table>
```

Advantages of Caching Data

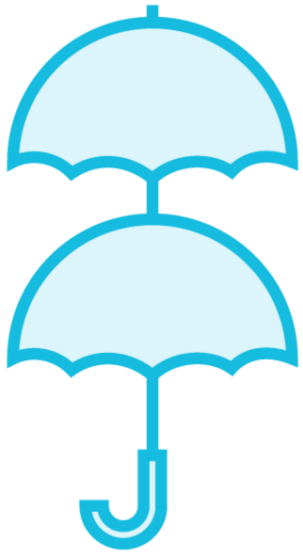


Improves responsiveness

Reduces bandwidth and network consumption



Reduces backend server load



Reduces redundant computations

Classic Caching Pattern

Product Service

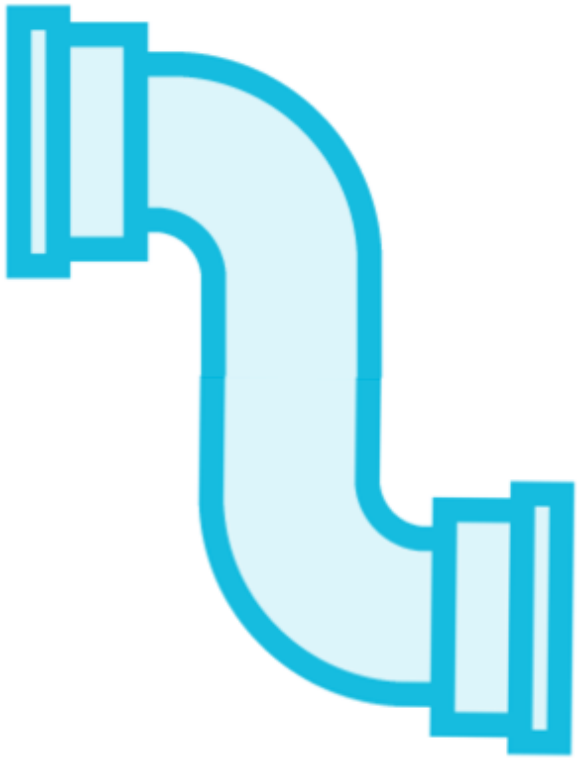
```
private products: Product[];

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

Declarative Caching Pattern

Product Service

```
private productsUrl = 'api/products';

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

RxJS Operator: `shareReplay`



Shares the stream with other subscribers

Replays the defined number of emissions on subscription

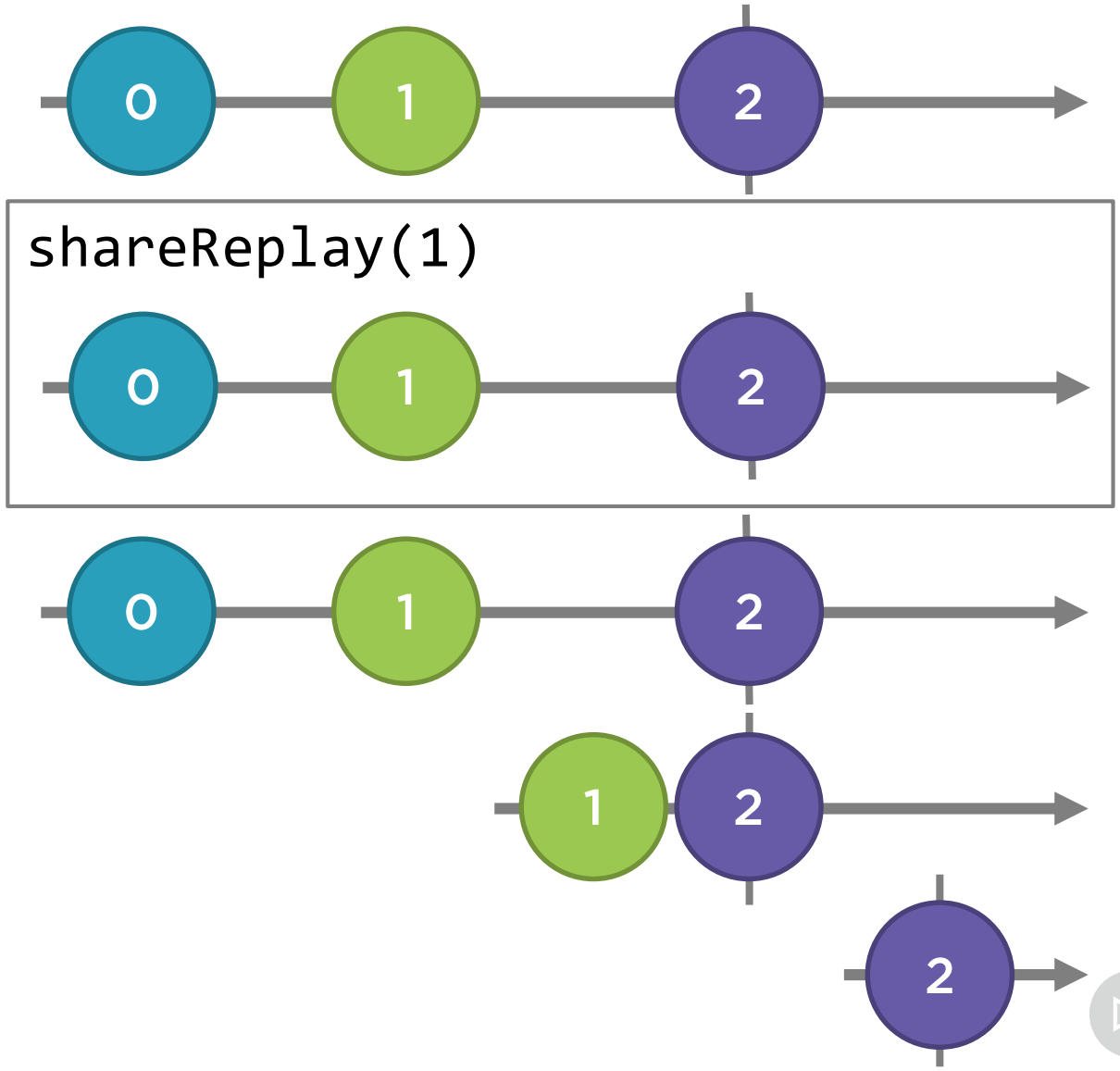
```
shareReplay(1)
```

Used for

- Caching data in the application

Marble Diagram: shareReplay

```
a$ = interval(1000)
 .pipe(
  take(3),
  shareReplay(1)
 )
```

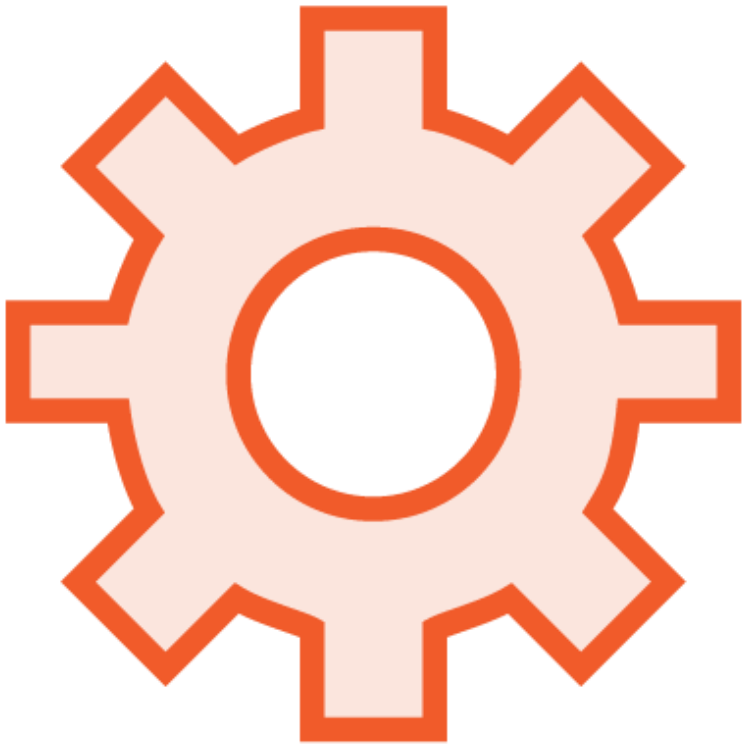



```
a$.subscribe(console.log);
```

```
a$.subscribe(console.log);
```

```
a$.subscribe(console.log);
```

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 a specified number of emissions

The items stays cached forever, even after there are no more subscribers

Demo

Caching data with `shareReplay`

Caching Observables

Use shareReplay on any stream you wish to share and replay to all new subscribers

```
productCategories$ = this.http.get<ProductCategory[]>(this.url)
  .pipe(
    tap(data => console.log('categories', data)),
    shareReplay(1),
    catchError(this.handleError)
  );
```

Cache Invalidation

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

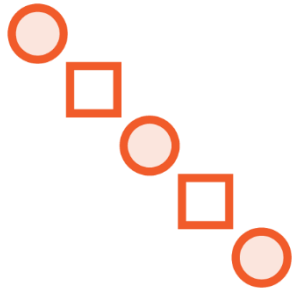
Final Words

Deborah Kurata
CONSULTANT | SPEAKER | AUTHOR | MVP | GDE

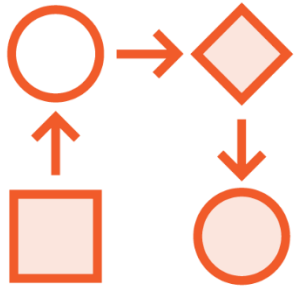
@deborahkurata | blogs.msmvps.com/deborahk/

Goals for This Course

Add clarity



Evaluate current patterns



Improve state management

Merge RxJS streams

Minimize subscriptions

Improve UI performance

Key points, tips and common issues

RxJS Operators

Import from 'rxjs/operators'

```
import { map, catchError } from 'rxjs/operators';
```

Specify operators within the pipe method

```
of(2, 4, 6)
  .pipe(
     map(item => item * 2),
     tap(item => console.log(item))
  );
```

RxJS Creation Functions

Import from 'rxjs'

```
import { combineLatest, of } from 'rxjs';
```

Be careful not to import the operator with the same name

```
import { combineLatest } from 'rxjs/operators';
```

```
vm$ = combineLatest([
  t
  t
  t
])
  .pipe(
    filter(([product]) => Boolean(product)),
    map(([product, productSuppliers, pageTitle]) =>
      ({ product, productSuppliers, pageTitle }))
  );
```

any

Property 'pipe' does not exist on type 'OperatorFunction<unknown, [unknown, Observable<string> | Observable<Product> | Observable<Supplier[]>]>'. ts(2339)

Quick Fix... Peek Problem

Debugging Observables

Use the tap operator

```
tap(data => console.log(JSON.stringify(data)))
```

Hover over the Observable to view the type

```
products$ = this.http.get<Product[]>(this.productsUrl)
(property) ProductService.products$: Observable<Product[]>
products$ = this.http.get<Product[]>(this.productsUrl)
  .pipe(
    tap(data => console.log('Products', JSON.stringify(data))),
    catchError(this.handleError)
  );
```

Data Streams

Emits one item, the response

After emitting the response, the stream completes

The response is often an array

To transform the array elements:

- Map the emitted array
- Map each array element
- Transform each array element

Action Streams

Only emits if it is active

If the stream is stopped, it won't emit

An unhandled error causes the stream to stop

Catch the error and replace the errored Observable

- Don't replace an errored action Observable with EMPTY
- Replace with a default or empty value

Combination Operators (Array)

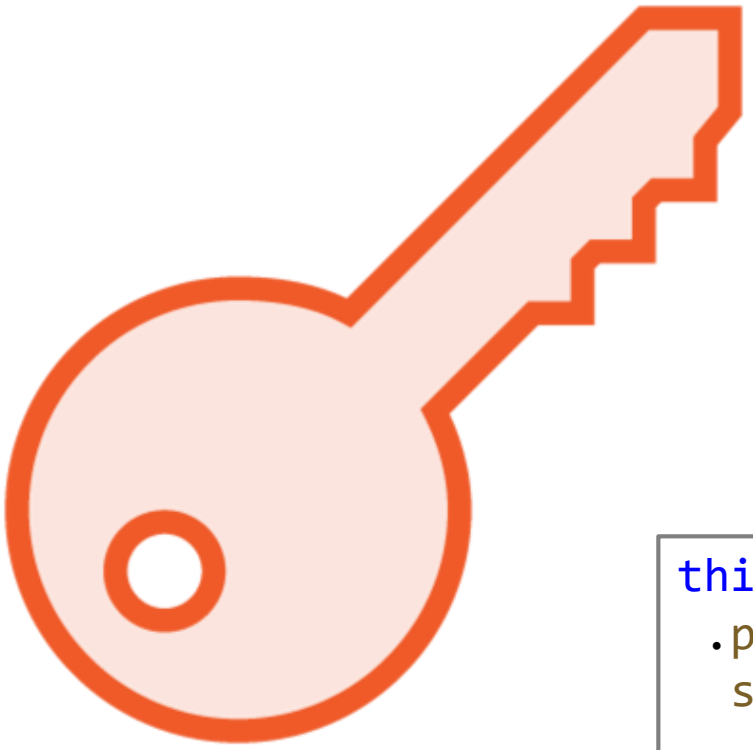
Don't emit until each source stream emits at least once

- `combineLatest`
- `forkJoin`
- `withLatestFrom`

Action stream created with a `Subject` does not immediately emit

When combining with an action stream, consider using a `BehaviorSubject` since it emits a default value

Complete Notifications



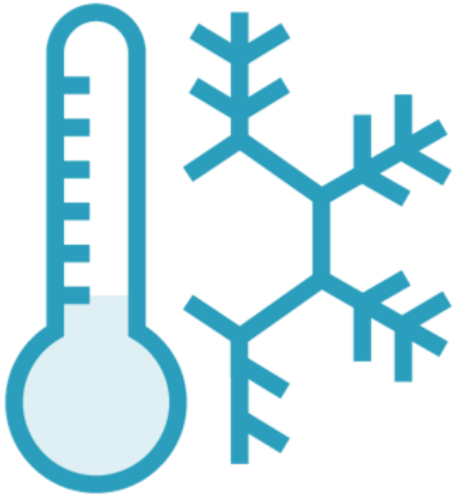
Some functions/operators require the input Observable(s) complete before they emit

- `forkJoin`
- `toArray`

Take care when using these functions/operators with action streams, which often don't complete

```
this.selectedProduct$
 .pipe(
  switchMap(selectedProduct =>
   from(selectedProduct.supplierIds)
    .pipe(
     mergeMap(id => this.http.get<Supplier>(`${this.url}/${id}`)),
     toArray()
 )));
```

A Few More Terms



Cold Observable

- Doesn't emit until subscribed to
- Unicast

```
products$ = this.http.get<Product[]>(url)
                .subscribe();
```

Hot Observable

- Emits without subscribers
- Multicast

```
productSubject = new Subject<number>();
this.productSubject.next(12)
```

Learning More

Pluralsight courses

- Angular Component Communication
- Angular NgRx: Getting Started
- RxJS: Getting Started
- Learning RxJS Operators by Example Playbook

RxJS documentation

- rxjs.dev

@deborahkurata