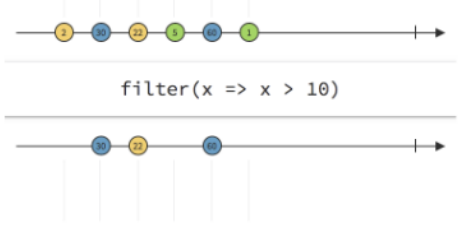
AMC Workshop

RxJS

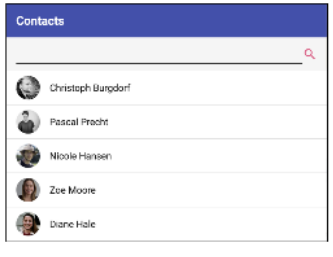
Filter Operator

Emit only those Observable stream values that pass a predicate test



StackBlitz Demo

RxJS Lab 1: Basic Search



- Add Search input controls above contacts lists
- Listen for input control value changes and call search()
- Update contacts$ with search() results

Lab Exercise

Reactive JavaScript: **RxJS**

Understanding **observable** and **operator** concepts and how they play a role in Angular applications is critical for successful Angular development.

In this course, developers will learn how to create, compose, and consume observables as streams for data-push process.

Observables

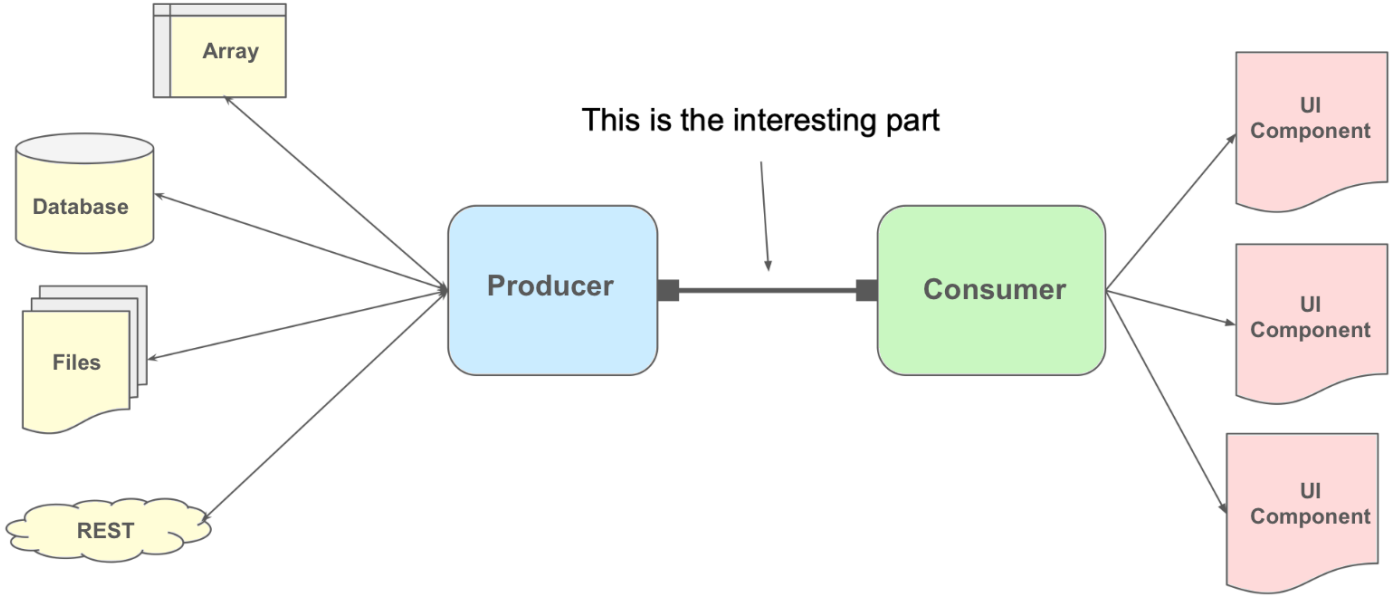
Give us a powerful way to **encapsulate**, **transport**, and **transform** data from user interactions to create powerful & immersive experiences.

What we will cover?

1. Observable Concepts
2. Consuming Observables
3. RxJS Operators
4. Creation Operators

Data: Producers and Consumers

1



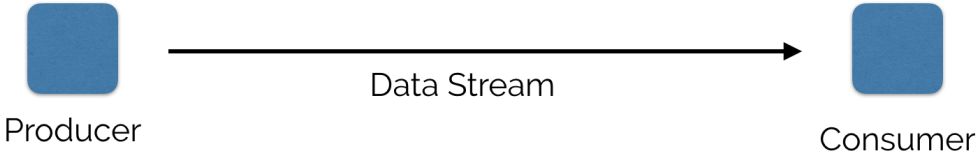
This connection is almost always asynchronous.

Pulling data from the producer is easy.... we do this already.
Push data from the producer is much harder... here, we must 'react'.

What are **Observables**?

1

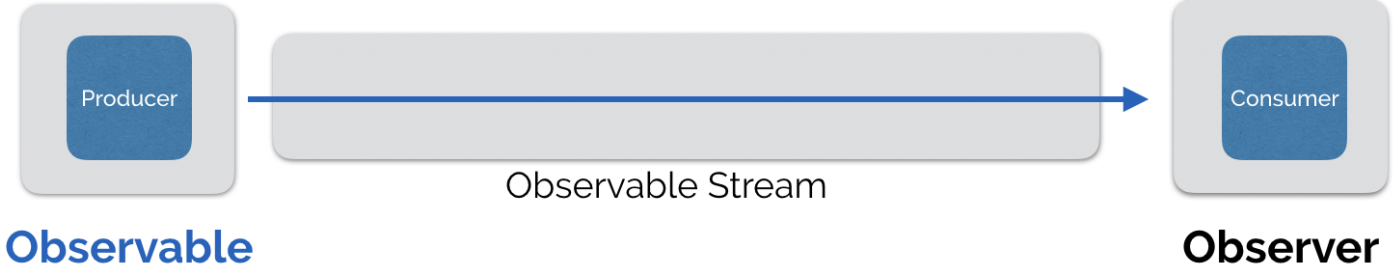
- Are **proxies** for future values
- Way of managing **sequences** of values or events
- **From producer**, **lazy push** stream of multiple values over time



- Consumer **Reacts** to values emitted from observable pipe

What are **Observables**?

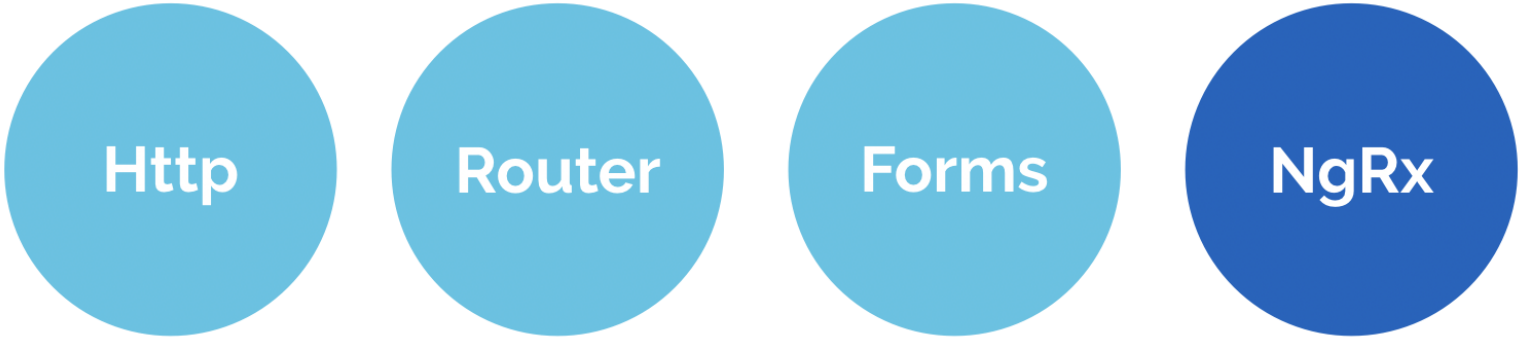
1



Values are emitted **synchronously** or **asynchronously**

Observables can be found in Angular

1



And change detection optimization using **ngZones**!

Why are **Observables** useful?

1

Flow

can control **when**, **how**, and **what** values output

Transformation

can **change the values** that pass through

Purity

produce values using **pure functions**; lesss prone to error

Pure Functions ?

1

- Analogous to **mathematical functions**
- A given input will always yield exactly the same out.
- No **mutation** and no **side effects**
- **Testability**
- **Composability**
- **Parallelism**

Push vs Pull

1

	Single	Multiple
Pull	`Function`	`Iterator`
Push	`Promise`	`Observable`

Functions and Observables

1

Function

deferred (lazy) evaluated computation that **synchronously** returns a single value

(pull mechanism)

Observable

deferred computation that can **synchronously** or **asynchronously** returns **0 - n values**

(push mechanism)

Speaker notes

The only way to defer a computation is to wrap it in a function!

Functions produces data when called
Observables may produce value(s) when called, may defer producing

Events, Promises, & Observables

1

Data Pull	Event Listeners	Promises	Observables
Single-Event	✔	✔	✔
Multi-Event	✔	—	✔
Chain	—	✔	✔
Object Inst	—	✔	✔
Sync	✔	—	✔
Async	—	✔	✔
Cancellable	✔	—	✔
Lazy	—	—	✔

Speaker notes

Promise delivers data to callbacks on the promise’s time (if it feels like calling us back)
(Mention how they compare to events)
Events track subscribers, share side side effects, and have eager execution regardless of subscribers

Observables are functions...

1

Function

deferred computation that synchronously returns a single value

(pull mechanism)

Observable

deferred computation that can synchronously or asynchronously returns 0 - n values

(push mechanism)

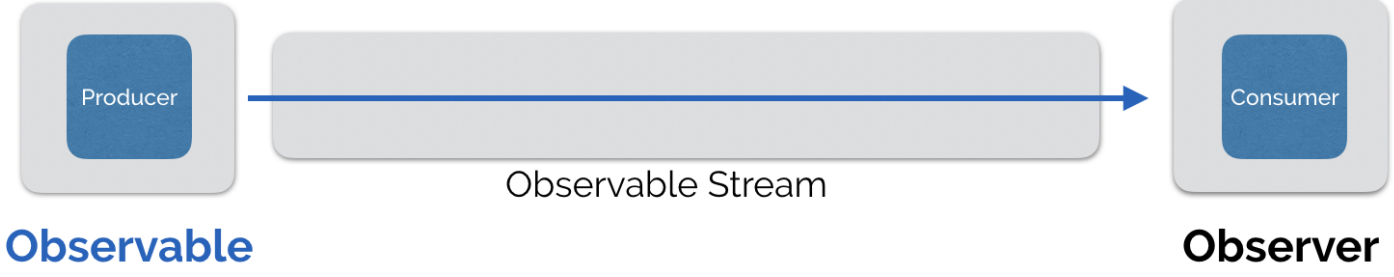
Speaker notes

The only way to defer a computation is to wrap it in a function!

Functions produces data when called
Observables may produce value(s) when called, may defer producing

Creation: Observables are Functions

1



Observables are **functions**[1] that accept an **observer** and returns a (cancellation) **function**

Accepts an **observer** in order to notify the external observer about internal activity....

1) Functions can be lazy triggered...

Observables are Functions

1

```
function myObservable( observer ) {

  // Producer Activity
  // Notify observer using callbacks

  return () => {

      // Cancel Connection to Producer
      // Stop Producer activity (optional)

  };

}
```

Speaker notes

So if an Observable is a function, how do we add meta features to it?
* lazy
* composition (chaining), etc
* Use Observable creation functions

Observables are Functions

1

```
const myObservable$ = Observable.create( observer => {

  // Producer Activity
  // Notify observer using callbacks

  return () => {

      // Cancel Connection to Producer
      // Stop Producer activity (optional)

  };

});
```

Speaker notes

Observable.create() creates object instance with `.subscribe( )`
Observable.create() is used
to support LAZY activating Producer notifications,
to support cancellations to stop Producer connections

Promise(s)

```
let myPromise = new Promise(( resolve, reject ) => {

  // Producer Activity

  resolve(
    /* Notify observer using callbacks */
  );

});
```

Producer activity is started immediately

Observable(s)

```
const myObservable$ = Observable.create( observer => {

  // Producer Activity
  // Notify observer using callbacks

  return () => {

      // Cancel Connection to Producer
      // Stop Producer activity (optional)

  };

});
```

Producer activity is deferred

Speaker notes

Observable.create() creates object instance with `.subscribe( )`
Observable.create() is used
* to support LAZY activating Producer notifications,
* to support cancellations to stop Producer connections
Use `< instance$ >.subscribe( )` nothing happens

Observables are Functions

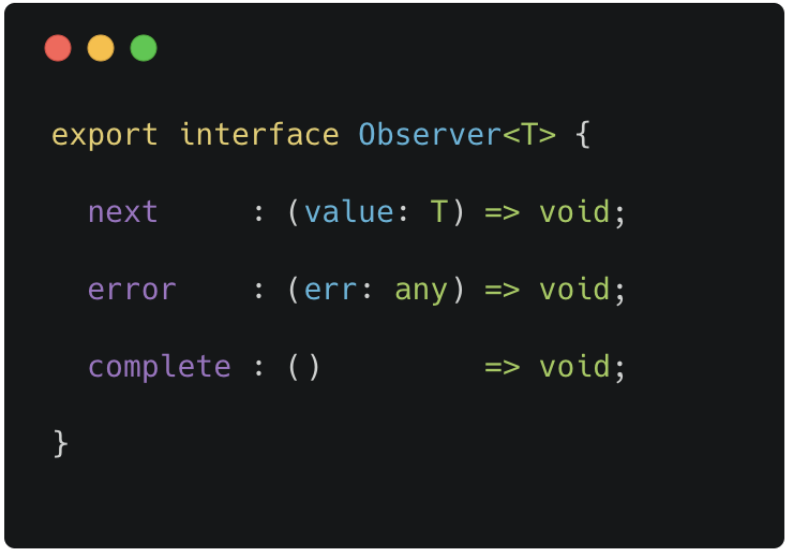
Like a function with zero (0) arguments that allow returning multiple values... over time.

Trigger the observable by using **.subscribe()**

Speaker notes

You “call” it but can’t pass any params to it (the observer doesn’t count)
If you don’t call subscribe, nothing will happen
calling subscribe is analogous to calling a function
Two subscribes can trigger two separate side effects
Event emitters share the same side effects and have eager execution

The Observer *Interface*



```
export interface Observer<T> {

  next     : (value: T) => void;

  error    : (err: any) => void;

  complete : ()         => void;

}
```

Observable.subscribe(watcher: Observer)

The Observer interface is a contract to register notification callbacks.

next * (error | complete)?

Speaker notes

The API contract for an Observable. This is the behavior we can expect to see.
Sends next notifications zero to infinite amount of times until it sends either error or complete
Strictly adheres to this contract (no more next after an error or complete)
The errorIcomplete is optional…could just keep going until you disconnect

* Next sends a value like number, string, object, etc
* Error sends a JavaScript error or exception

Observable Execution

1

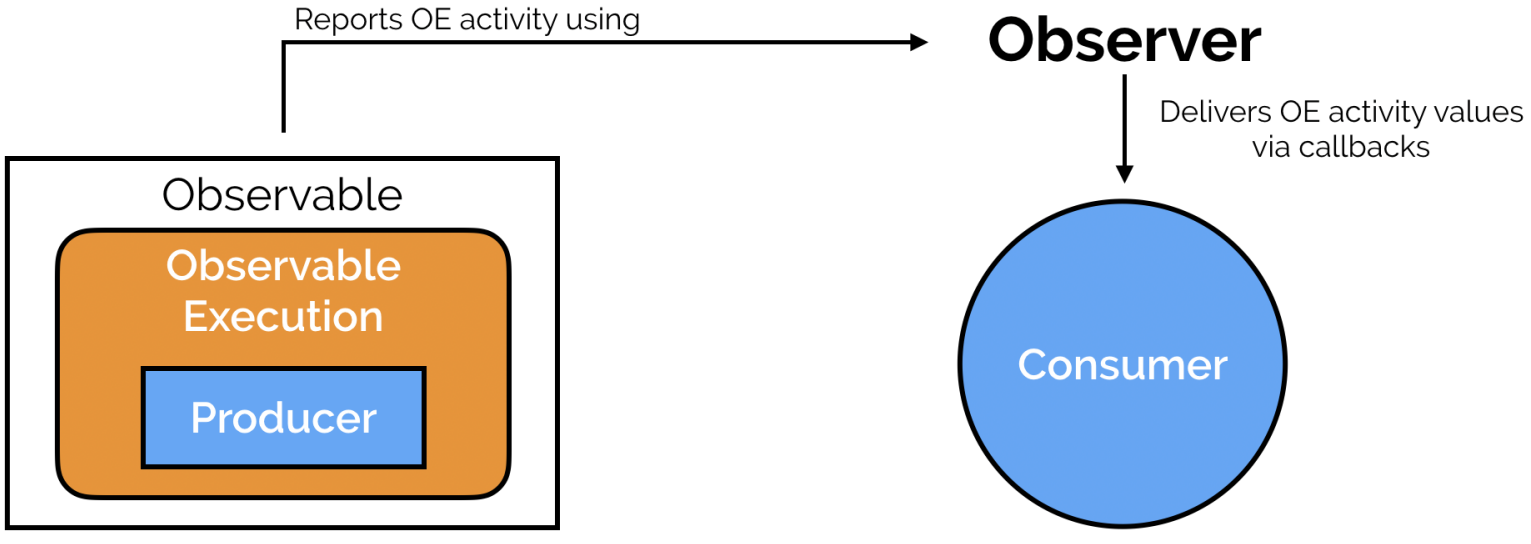
An observable execution is a lazy computation that happen **for EACH** Observer that subscribes.

```
Observable.create(observer => {
    try {
      observer.next(1);
      observer.next(2);
      observer.next(3);
      observer.complete();
    } catch (err) {
      observer.error(err);
    }
});
```

Speaker notes

So something is going to produce these values…the observable execution
Plain Observables are unicast and get their own independent execution
An observer is simply a set of callback functions that the Observable execution can call
And a call to subscribe can run that observable execution

Producers & Consumers



Speaker notes

* Producers are those observables found in Angular (forms, router, httpclient, etc) and ones we make
* Consumers are the subscriptions to those, where we get pushed the values and react to them

What we will cover?

1. Observable Concepts
2. Consuming Observables
3. RxJS Operators
4. Creation Operators

Speaker notes

- Let’s start with the consumer side of things, since this is the most common work we will find ourselves doing with RxJS in Angular

Consume Observables using Subscriptions

2

Subscribing to an Observable is analogous to calling a Function

Observables are able to deliver values either **synchronously** or **asynchronously**.

Speaker notes

A subscription either starts the observable execution or connects to a running one (depending on the observable type)
That may result in the delivery of value(s), it may continue with values after that, or it may even deliver value(s) and complete right away. That is the rules of the observable contract, right?

Observers

Consumers can listen for emitted Observable values using **callbacks**.

Three (3) callbacks or an object can used as arguments to the **subscribe()** function

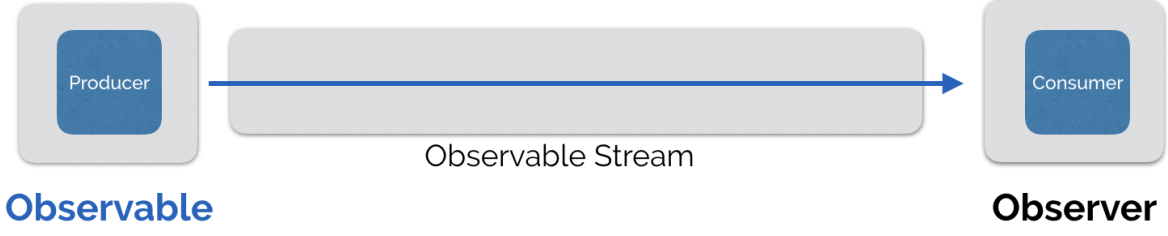
```
export declare class Observable<T>  {

  static create: Function;

  subscribe(
     next?     : (value: T)   => void,
     error?    : (error: any) => void,
     complete?: ()            => void
    ) : Subscription;

}
```

Speaker notes

A subscribe call takes up to 3 callback functions that will get used as an observer
These are callbacks that will get run based on the event the observable execution runs
Can hand it an object, multiple params that are the callback for each (subscribe is an overloaded method)

Common **Subscribe()** Example

2




```
const source$ = Observable
  .create(observer => {
    // Announce producer activity
    observer.next(1);
    observer.complete();
  });
```

Producer uses the **observer channel** to push values to a single consumer

```
// Only watch for emitted values

source$.subscribe(value => {
  // Report producer activity
  console.log(value);
});
```

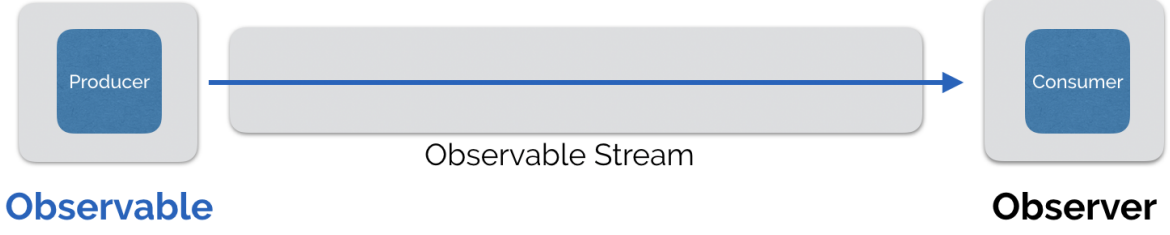
The subscribe registers a **callback bridge** to the consumer. **Consumer** listens and reacts to values

Speaker notes

The most common way to subscribe is to hand it the callback for the next event to work with values pushed

Error Handling

2




```
const source$ = Observable
  .create(observer => {
    observer.next(1);

    observer.error("oops");
  });
```

```
source$.subscribe(
  value   => console.log(value),
  message => console.error(message)
);
```

Watch for emitted values AND **errors**

Speaker notes

We can also include a second callback function for errors.
Now there are several ways to do it, a lengthly lesson in its own right
* Can handle errors here
* Can leverage operators to catch, retry, etc.

Just want to mention it here to make you aware…

Disposing on Observable Execution

Connect to Producer

Save the **Subscription**


```
const connection : Subscription = source$.subscribe(
  value => this.item = value;
);
```

source$ mediates access to the producer...

Disconnects from Producer

Aborts **Observable Execution**


```
connection.unsubscribe();
```

Speaker notes

When we subscribe, we need to think about the need to unsubscribe
Not unsubscribing can lead to memory leaks
The observable is in charge of what it does in its observable execution, also in charge of cleaning up if unsubscribe is called
The subscribe call returns an object with an unsubscribe method. Calling that method will tell the observable that you are done with it. Hopefully it will clean up after itself!

Unsubscribe... When or Always ?

2

Must Unsubscribe

FormControl

Custom Observables (depends)

3rd-Party Observables (depends)

Unsubscribe Not Needed

HttpClient

Router

Some operators; eg. **take**

When using the **async** pipe!

Speaker notes

Unsubscribe
Control, Group and Array
(depends) if that observable execution completes
Note the HttpClient completes
Router params, data, etc all have some Angular lifecycle hooks that clean themselves up
Will talk about operators later

Code without the **async** pipe...

2

```
@Component({
  selector : 'ticket-map',
  template : `
    <ticket-card *ngIf="ticket"[data]="ticket">
   </ticket-card>

    <button (click)="loadTicket()">Load All Tickets</button>
  `
})
export class NxWorkshopMap {
  ticket : Ticket;
  constructor(private ticketHttp : TicketService) { }

  loadTicket(ticketId = 0) {
    const pendingTicket$ = this.ticketHttp.loadTicket(ticketID);

    pendingTicket$.subscribe(
      (ticket) => this.ticket = ticket
    );
  }

}
```

Speaker notes

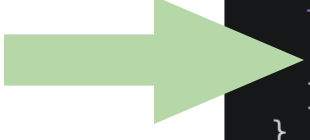
Here’s a typical pattern we may find ourselves doing, we subscribe to get to the value and then use that value.
What are the problems here:

* Manual extraction of data
* No cancellations
* No cleanup on ngOnDestroy()
* Zombies

Power with the **async** pipe...

2

```
<ticket-card
    *ngIf="(ticket$ | async) as currentTicket"
    [dataProvider]="currentTicket" >
</ticket-card>
```



```
export class NxWorkshopMap {
  ticket$ : Observable<Ticket>;

  constructor(private ticketService : TicketService) {
    this.loadTicket(1);
  }

  loadTicket(ticketId) {
    this.ticket$ = this.ticketService.loadTicket(ticketId);
  }
}
```

Speaker notes

Can feed it an observable
The async pipe does the subscribe call
It also does the unsubscribe call when component is destroyed!
And because it’s working with Observables, new values pushed to it will mark for check
Can use the `as` syntax to get a handle to the value from it for use in template

Power with the **async** pipe...

2

```
<ticket-card
    *ngIf="(ticket$ | async) as currentTicket"
    [dataProvider]="currentTicket" >
</ticket-card>

<button (click)="loadTicket(2)">
    Load All Tickets
</button>
```

```
export class TicketsExplorer {
  ticket$ : Observable<Ticket>;

  constructor(private ticketService : TicketService) {
    this.loadTicket(1);
  }

  loadTicket(ticketId) {
    this.ticket$ = this.ticketService.loadTicket(ticketId);
  }
}
```

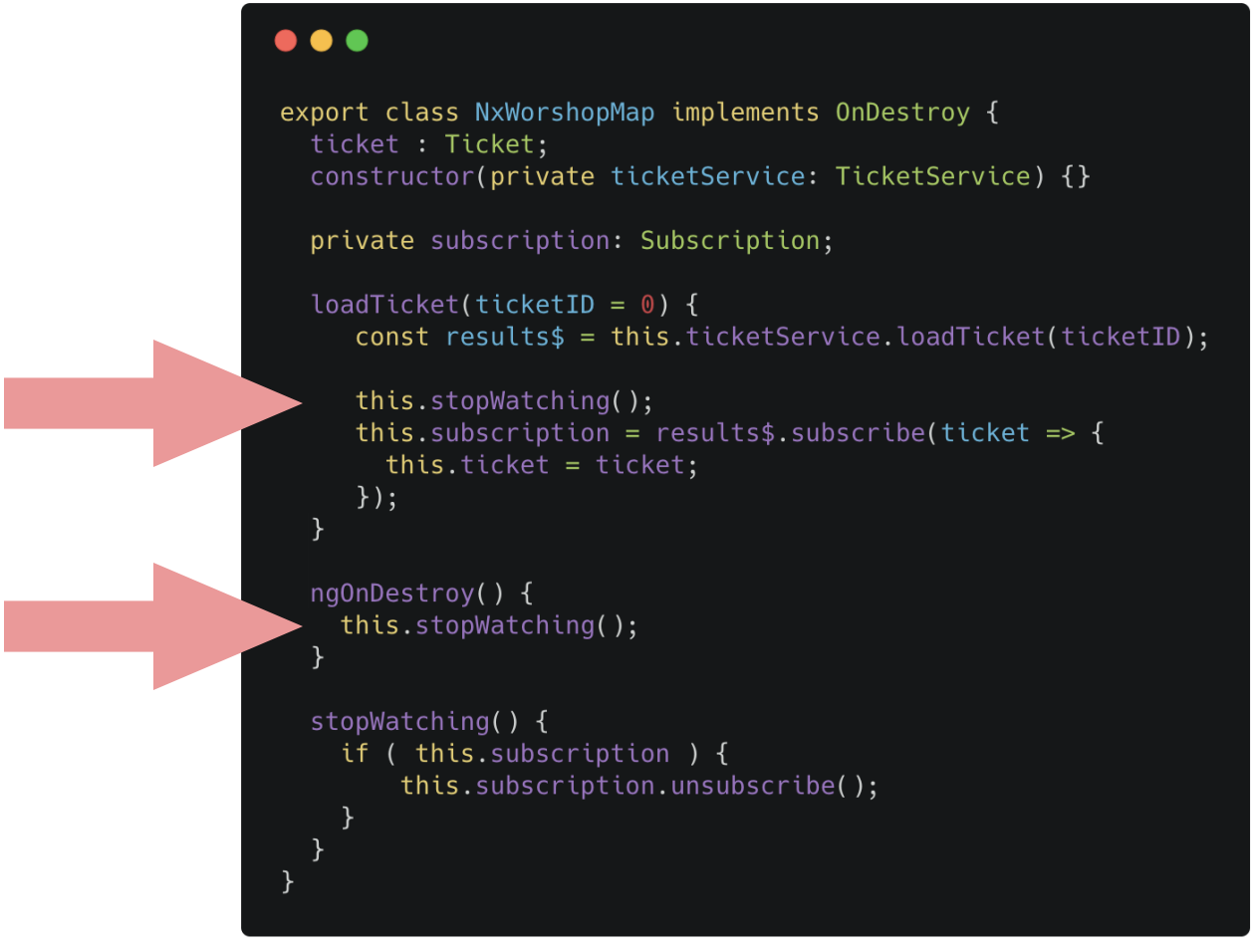
A change to the expression (eg a new Observable) is auto-handled by the async pipe!

Speaker notes

It will also handle unsubscribing and subscribing if we change the expression value
So if we had a button click to load a different ticket…
It will unsubscribe on change to expression
And subscribe to the new expression value

Traditional **Unsubscribe**()

2



```
export class NxWorshopMap implements OnDestroy {
  ticket : Ticket;
  constructor(private ticketService: TicketService) {}

  private subscription: Subscription;

  loadTicket(ticketID = 0) {
     const results$ = this.ticketService.loadTicket(ticketID);

     this.stopWatching();
     this.subscription = results$.subscribe(ticket => {
       this.ticket = ticket;
     });
  }

  ngOnDestroy() {
    this.stopWatching();
  }

  stopWatching() {
    if ( this.subscription ) {
        this.subscription.unsubscribe();
    }
  }
}
```

This is a typical solution seen in the community.

This is horrible... don't do it!

Using the **untilViewDestroyed** operator

2


```
export class TicketsExplorer {
  ticket : Ticket;

  constructor(
    private ticketService: TicketService,
    private elRef: ElementRef) {

      this.loadTicket(1);
  }

  loadTicket(ticketId) {
    const ticket$ = this.ticketService.loadTicket(ticketId);
    ticket$.pipe(
      untilViewDestroyed(this.elRef)
    )
    .subscribe(ticket => {
      this.ticket = ticket
    })
  }
}
```

untilViewDestroyed RxJS Operator

2

```typescript
export function untilViewDestroyed<T>(element: ElementRef): (source: Observable<T>) => Observable<T> {
  const destroyed$ = watchElementDestroyed(element.nativeElement);
  return (source: Observable<T>) => source.pipe(takeUntil(destroyed$));
}

/**
 * Unique hashkey
 */
const destroy$ = 'destroy$';

/**
 * Use MutationObserver to watch for Element being removed from the DOM: destroyed
 * When destroyed, stop subscriptions upstream.
 */
function watchElementDestroyed(nativeEl: Element, delay: number = 20): Observable<boolean> {
  if (!nativeEl[destroy$]) {
    const stop$ = new ReplaySubject<boolean>();
    const hasBeenRemoved = isElementRemoved(nativeEl);

    nativeEl[destroy$] = stop$.asObservable();
    setTimeout(() => {
      const domObserver = new MutationObserver((records: MutationRecord[]) => {
        if (records.some(hasBeenRemoved)) {
          stop$.next(true);
          stop$.complete();

          domObserver.disconnect();
          nativeEl[destroy$] = null;
        }
      });

      domObserver.observe(nativeEl.parentNode, { childList: true });
    }, delay);
  }

  return nativeEl[destroy$];
}
```

[Gist Source]

HttpClient

- Event-driven **stream** of notifications
- Enables functional programming structures

- Auto-conversion to **JSON** objects
- **Strong-typing** for request and response objects
- Request and Response **interceptors**

- Improved **Error Handling** w/ retry()
- **Progress Events** (upload & download)

Speaker notes

Okay, so let’s talk about the HttpClient in Angular
It is a great use case of Observables
It does the http call stuff real well (json handling, strong typing, extension through interceptors)
And we can do advanced stuff like retry and progressive up/down

HttpClient

2

- Returns **Observables** for get, post, put, delete
- They are not active
- Designed to **auto-complete**

Speaker notes

And when it comes to http methods, it makes use of observables
These are observables with an observable execution just waiting to be run
And the observable execution will complete once its done with its logic

Subscribe to start & complete...

2

Configure the Http observable to prepare to call, then **start** and **complete** by calling **.subscribe()**

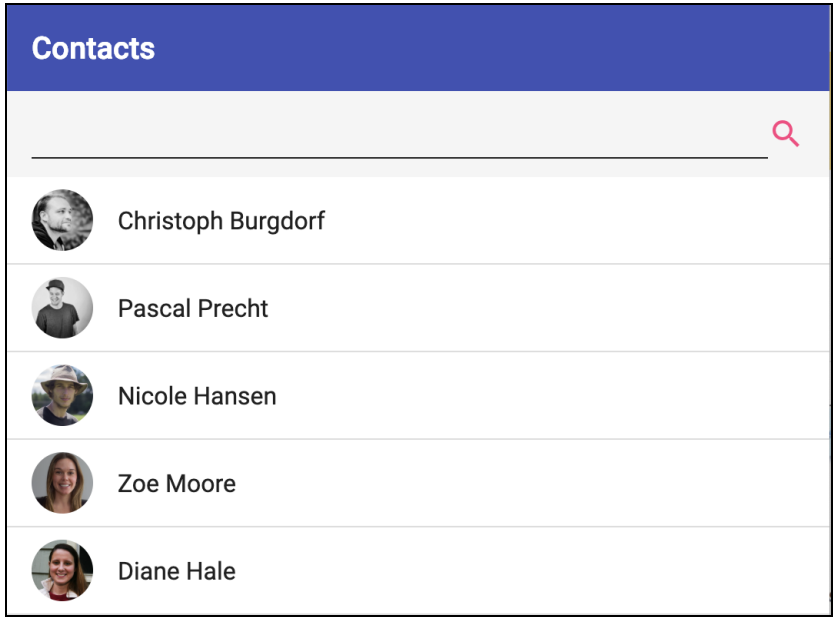
```
const url = `${this.apiEndpoint}/tickets/${ticketID}`;

// Configure pending HTTP call
const ticket$ = this.http.get<Ticket>(url);

// Start and complete HTTP call
ticket$.subscribe(
  (ticket:Ticket) => this.ticket = ticket
);
```

In this case ^, the **Observable Execution** will be the actual internal call to the remote REST endpoint

Speaker notes

So we can think of the subscribe as the function call to do the get, post, put, delete :)
And if you remember your function call training!
The HttpClient observable, when you call subscribe, will run its observable execution which will make an async call and upon finish will next and complete
And for the HttpClient, each call starts its own observable execution
* Two subscribes can equal two network requests

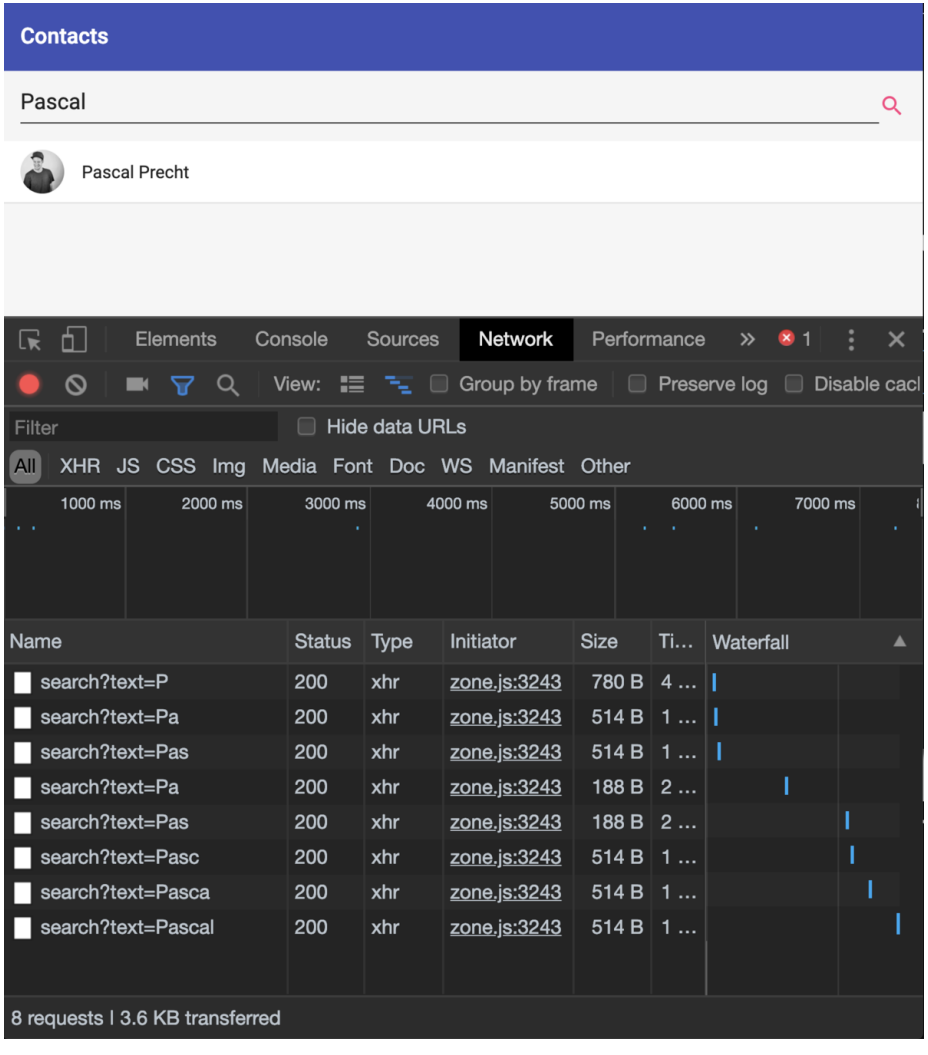
RxJS Lab 1: Basic Search



- Add Search input controls above contacts lists
- Listen for input control value changes and call search()
- Update contacts$ with search() results

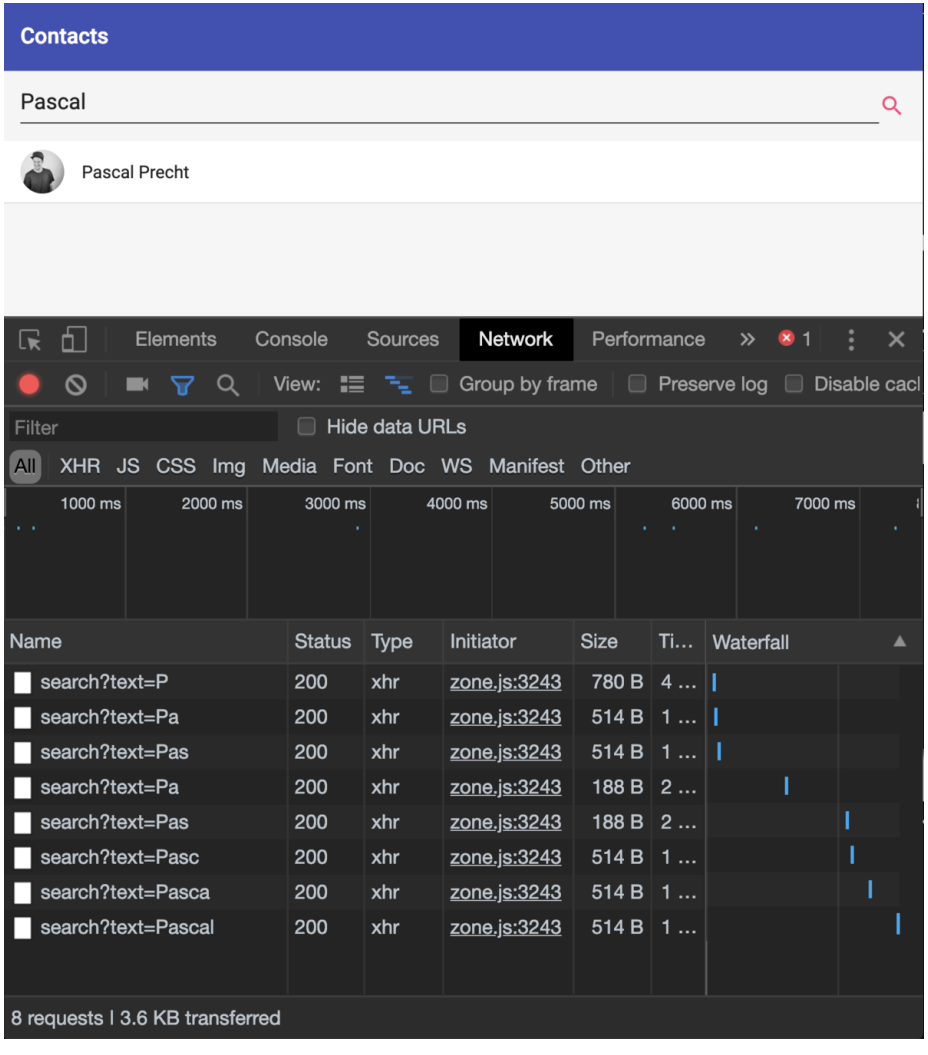
Lab Exercise

Problems with Basic Search



This simple search implementation comes with a couple of problems:

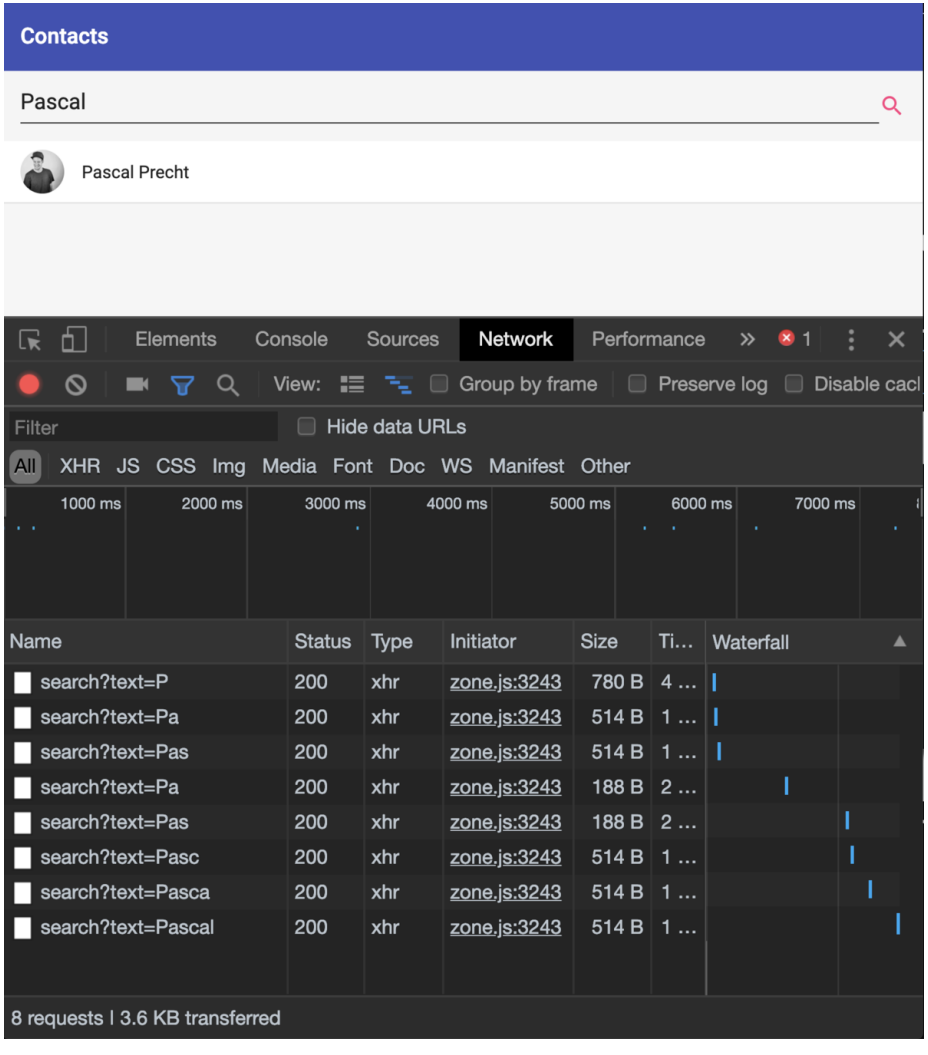
Problems with Basic Search



This simple search implementation comes with a couple of problems:

- We hit the server on every keystroke

Problems with Basic Search

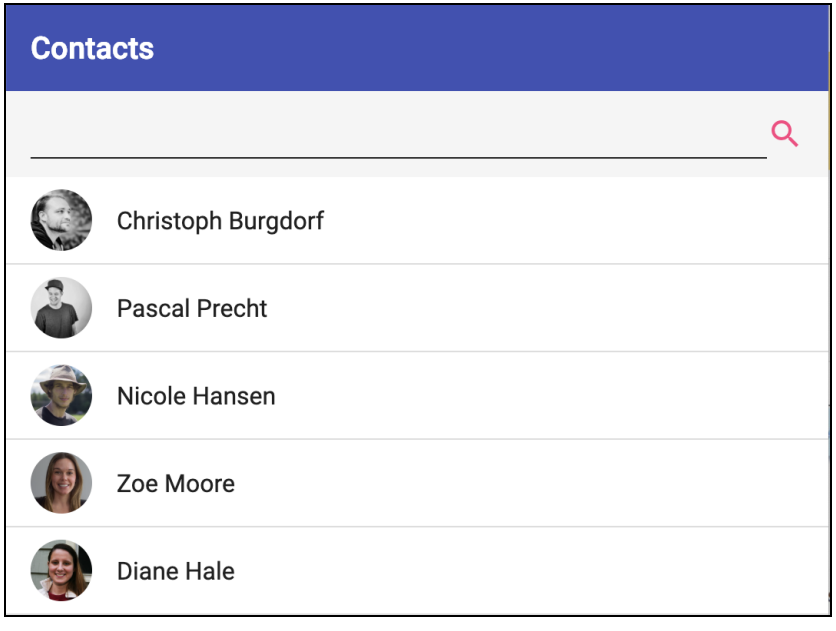


This simple search implementation comes with a couple of problems:

- We hit the server on every keystroke
- We hit the server unnecessarily

Solutions with Basic Search

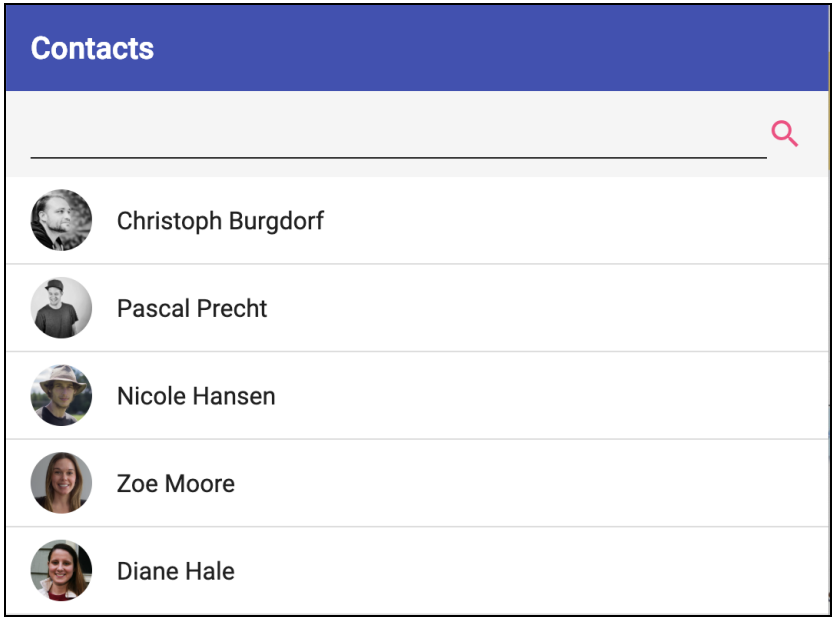
If we can somehow treat the input control as an 'observable stream', we could then use the power of RxJS operators to solve our **problems.**



To get input values via an Observable stream, we could us:

Solutions with Basic Search

If we can somehow treat the input control as an 'observable stream', we could then use the power of RxJS operators to solve our **problems.**

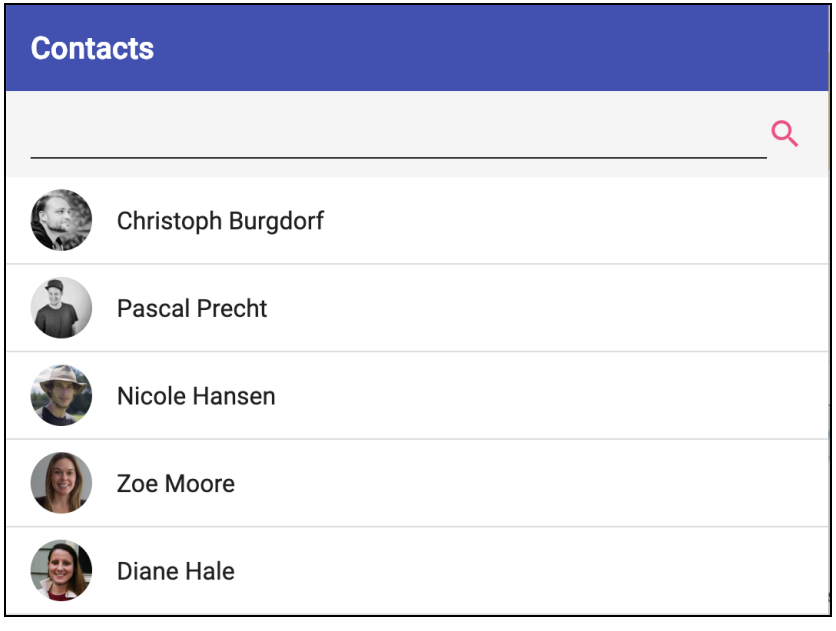


To get input values via an Observable stream, we could us:

- Reactive **FormControls**

Solutions with Basic Search

If we can somehow treat the input control as an 'observable stream', we could then use the power of RxJS operators to solve our **problems.**



To get input values via an Observable stream, we could us:

- Reactive **FormControls**
- Use a **Subject** to create a stream that emits input value changes

But first...

1. Observable Concepts
2. Consuming Observables
3. RxJS Operators
4. Creation Operators

Speaker notes

So we learned how to subscribe and what is going on there.
Let’s talk about operators, and how they give us, as consumers, a ton of power

RxJS **Operators**

A function that creates a **new** Observable based on the current observable.

Why ?

Allows custom projection functions to **project**

or block each value in the stream into another value

RxJS **Operators**

- Are pure functions
- **Input read-only** observable
- **Output new** observable (generated)
- Used to **chain** observables

Why ?

Immutability: the input Observable remains **unchanged** and can be **reused**!

RxJS **Operators**

Operator chaining is an implementation of the **Pipeline** pattern

- Operators support **chaining** observables
- Each operator is applied to the **output observable** from the **previous** operator.
- Operator **order** is important

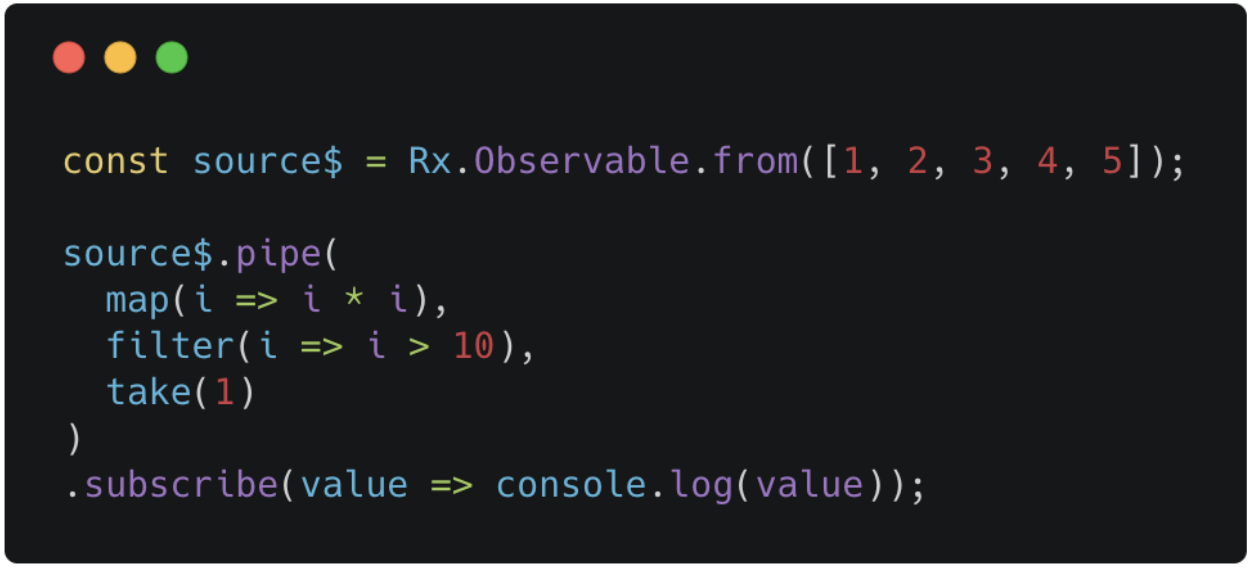
Speaker notes

A value produced at the source, will sequentially flow through the operators of an observable chain.

Each operator in the chain decides if it will emit the value again (filter) and/or if it will modify the value (transformation).

Operators may also choose to append or prepend values to the stream or combine it with another stream.

RxJS **Operators:** Subscription Avalanche



```
const source$ = Rx.Observable.from([1, 2, 3, 4, 5]);

source$.pipe(
  map(i => i * i),
  filter(i => i > 10),
  take(1)
)
.subscribe(value => console.log(value));
```

- Each operator **subscribes to the incoming** Observable
- Each operator **outputs a new Observable**

A chain of RxJs operators is a chain of subscribers...

Observable Data Flows

Push vs Pull

- Data is **pulled** from a Array
- Data is transformed with operations: **map(), filter(), reduce()**

Observables are similar to Lists...

- Data is **pushed** through an Observable
- Data is transformed with operations: **map(), filter(), reduce()**

Speaker notes

Previous Observable stays unmodified

Consider traditional Array **Operators**:


```
[1, 2, 3, 4].filter(i ⇒ i % 2);

['a', ‘b', 'c'].map(i ⇒ i.toUpperCase());
```

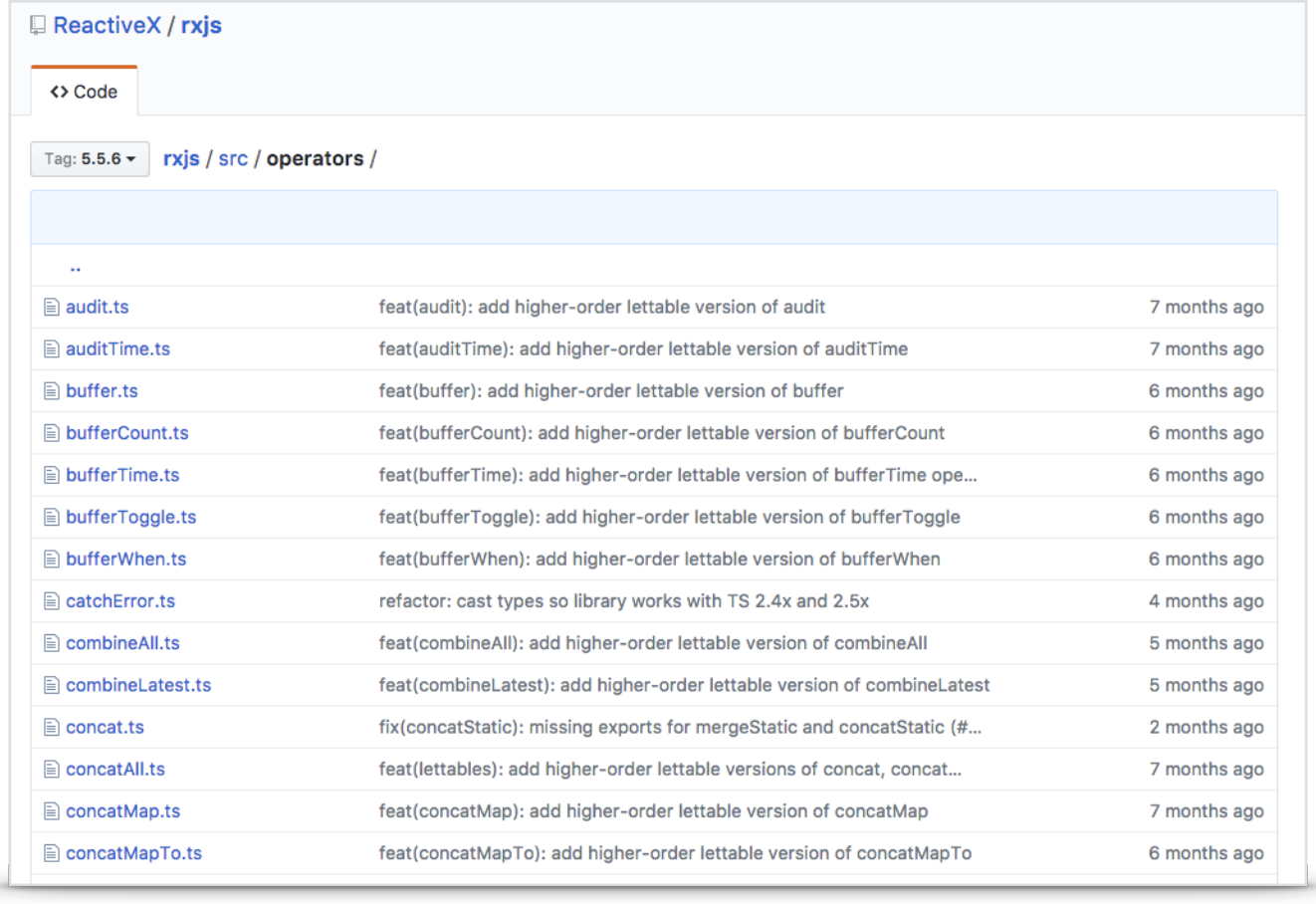
Speaker notes

RxJS Operators are inspired by these
But more than just Array-like ones…
(might want to try and explain how this can trip you up though, because Observable filter will get 1 then 2 then 3.

The array filter will get the entire set at once. Array operators [in general] iterate the entire list...

RxJS comes with a bunch of **Operators**

3



ReactiveX / rxjs

<> Code

Tag: 5.5.6 ▾ rxjs / src / operators /

..		
audit.ts	feat(audit): add higher-order lettable version of audit	7 months ago
auditTime.ts	feat(auditTime): add higher-order lettable version of auditTime	7 months ago
buffer.ts	feat(buffer): add higher-order lettable version of buffer	6 months ago
bufferCount.ts	feat(bufferCount): add higher-order lettable version of bufferCount	6 months ago
bufferTime.ts	feat(bufferTime): add higher-order lettable version of bufferTime ope...	6 months ago
bufferToggle.ts	feat(bufferToggle): add higher-order lettable version of bufferToggle	6 months ago
bufferWhen.ts	feat(bufferWhen): add higher-order lettable version of bufferWhen	6 months ago
catchError.ts	refactor: cast types so library works with TS 2.4x and 2.5x	4 months ago
combineAll.ts	feat(combineAll): add higher-order lettable version of combineAll	5 months ago
combineLatest.ts	feat(combineLatest): add higher-order lettable version of combineLatest	5 months ago
concat.ts	fix(concatStatic): missing exports for mergeStatic and concatStatic (#...	2 months ago
concatAll.ts	feat(lettables): add higher-order lettable versions of concat, concat...	7 months ago
concatMap.ts	feat(concatMap): add higher-order lettable version of concatMap	7 months ago
concatMapTo.ts	feat(concatMapTo): add higher-order lettable version of concatMapTo	6 months ago

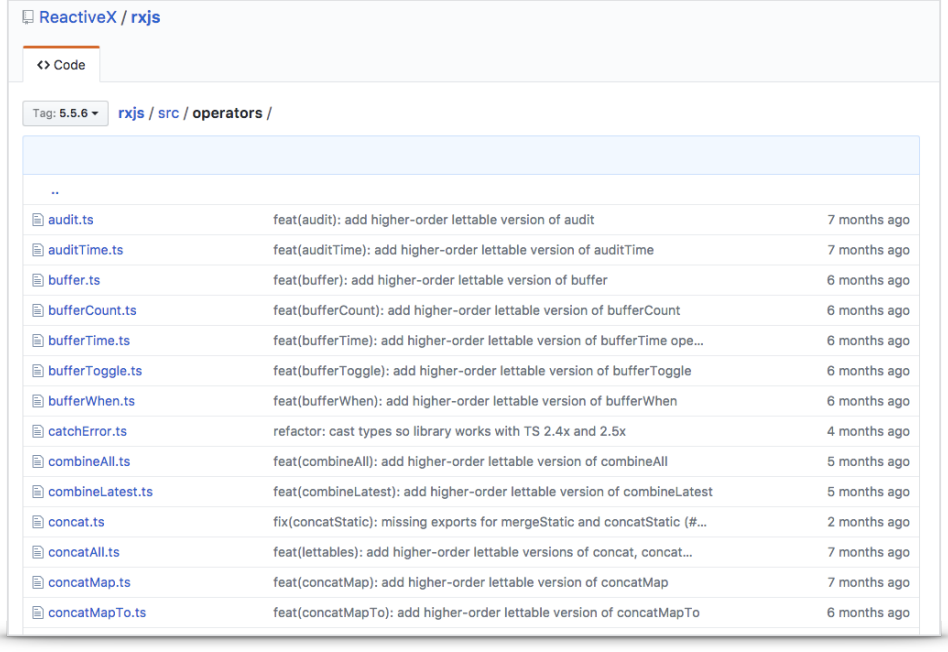
> 105 operators...

Speaker notes

So many operators... plus it is super easy to create our own custom operators.
Like the `untilViewDestroyed()` operator.

Types of **Operators**

3



ReactiveX / rxjs

Code

Tag: 5.5.6 rxjs / src / operators /

..		
audit.ts	feat(audit): add higher-order lettable version of audit	7 months ago
auditTime.ts	feat(auditTime): add higher-order lettable version of auditTime	7 months ago
buffer.ts	feat(buffer): add higher-order lettable version of buffer	6 months ago
bufferCount.ts	feat(bufferCount): add higher-order lettable version of bufferCount	6 months ago
bufferTime.ts	feat(bufferTime): add higher-order lettable version of bufferTime ope...	6 months ago
bufferToggle.ts	feat(bufferToggle): add higher-order lettable version of bufferToggle	6 months ago
bufferWhen.ts	feat(bufferWhen): add higher-order lettable version of bufferWhen	6 months ago
catchError.ts	refactor: cast types so library works with TS 2.4x and 2.5x	4 months ago
combineAll.ts	feat(combineAll): add higher-order lettable version of combineAll	5 months ago
combineLatest.ts	feat(combineLatest): add higher-order lettable version of combineLatest	5 months ago
concat.ts	fix(concatStatic): missing exports for mergeStatic and concatStatic (#...	2 months ago
concatAll.ts	feat(lettables): add higher-order lettable versions of concat, concat...	7 months ago
concatMap.ts	feat(concatMap): add higher-order lettable version of concatMap	7 months ago
concatMapTo.ts	feat(concatMapTo): add higher-order lettable version of concatMapTo	6 months ago

- Transformation
- Filtering
- Combination
- Many others...

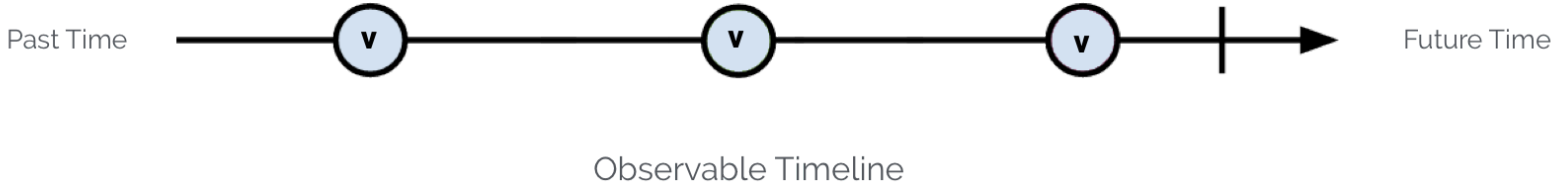
Developers really only need to know **4 - 6 operator**s to get started!

Marble Diagrams

Diagrams help visualize Observable events and RxJS operators!

RxMarbles.com

Marble Diagrams



Observable Timeline

Marble Diagrams - Text Based

Marble syntax is a string that represents events happening over **virtual** time.

```
- 1 - - 2 - - 3 - - |
```

Speaker notes

This sytnax is very useful for marble testing... more later

Observer Interface -> Marble Diagrams

next * (error | complete)?

```
1,2,3      #          |
```

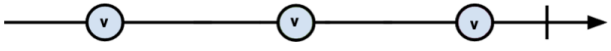
Speaker notes

Within the marble sequence, we can represent the observable contract for the observer:
* Values for the next
* Hash for the error
* Pipe for the complete

Observable Visualizations

3

```
Observable.create(observer => {

  observer.next(1);
  observer.next(2);
  observer.next(3);

  observer.complete();

});
```



```
- 1 - - 2 - - 3 - |
```

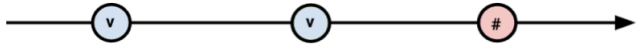
Speaker notes

Three values then complete
In the text one, dashes can be used to simulate an amount of time (for testing)

Observable Visualizations

3

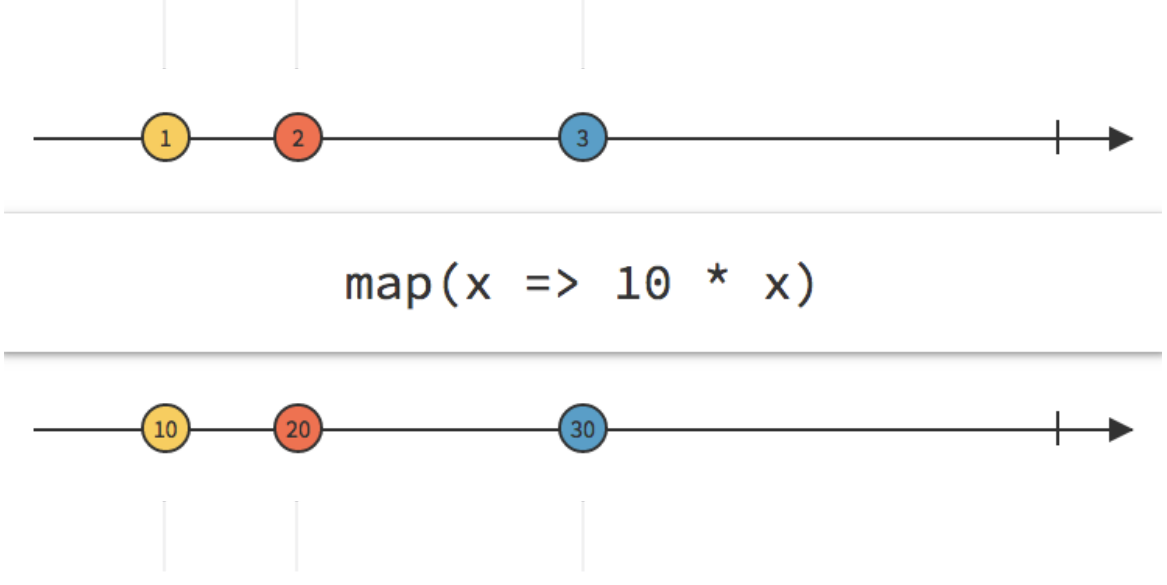
```
Observable.create(observer => {

  observer.next(1);
  observer.next(2);

  observer.error('oops!');

});
```




```
- 1 - - 2 - - #
```

Map Operator

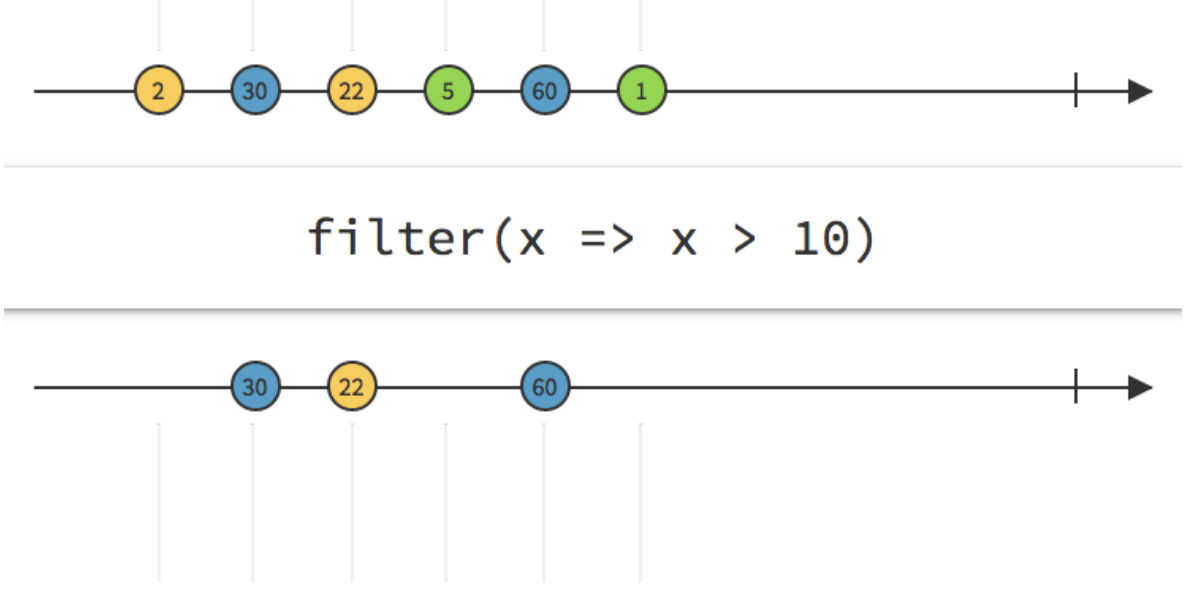
Transform the items emitted by an Observable by applying a function to each item



StackBlitz Demo

Filter Operator

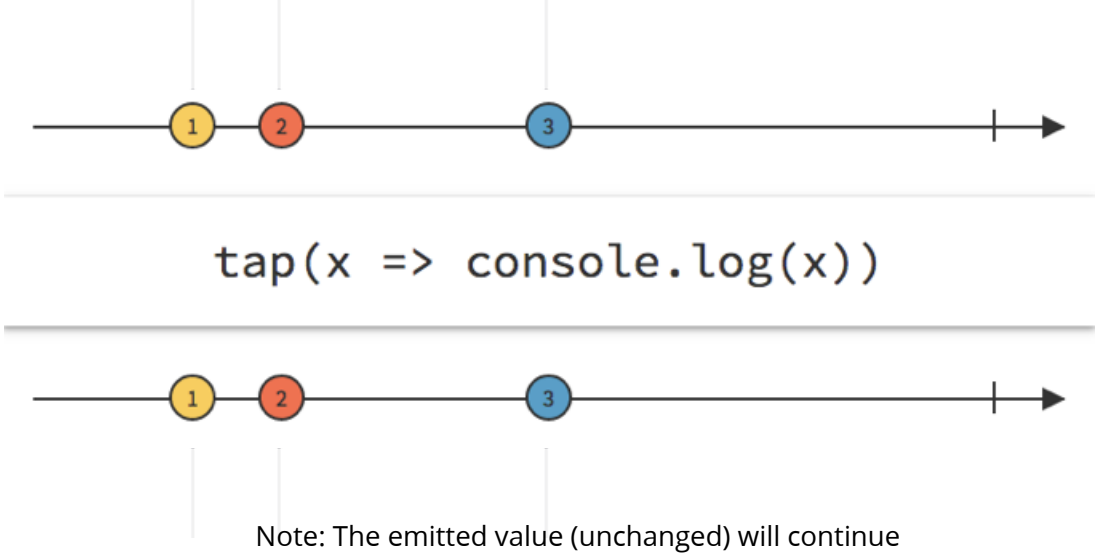
Emit only those Observable stream values that pass a predicate test



StackBlitz Demo

Tap Operator

Register an action to take for each value emitted from an Observable

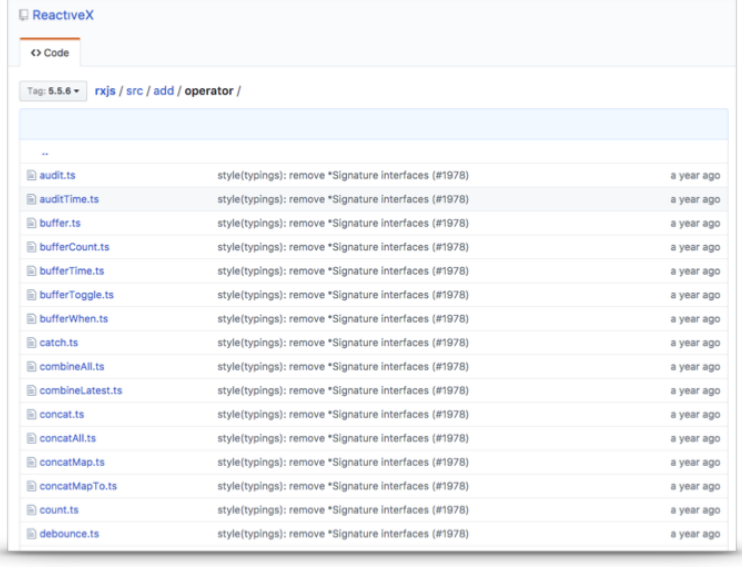


Note: The emitted value (unchanged) will continue to propagate through the pipe.

StackBlitz Demo

Importing RxJS Operators

3




```
import 'rxjs/add/operator/map';
```

Don't do this...Why?

These ^ imports monkey-patch the Observable prototype!

Speaker notes

Import them where you use them

Different from how it used to be (if you are familiar with that)
Will get compile errors if you don’t…so that’s a win!

Importing RxJS Operators

Speaker notes

When it comes to using operators we can import them
Will get compile errors if you don't…so that's a win!

Make sure you get the path rxjs/operators…do not use `rxjs/add/operators`

Using RxJS Operators

- RxJS 5.x introduced a **.pipe()** used to chain multiple (1...n) operators
- Can compose multiple **operators** into one (1) statement
- Composing operators keeps together

easy to reuse
easy to maintain

Speaker notes

Developers must use the pipe to chain operators BEFORE the subscribe()

Function **Chaining** == Function **Composition**

3

```
// Function chaining
// requires each f() to return `x`

const v1 = x.f1().f2().f3()


// Function composition

const v2 = f3(f2(f1(x)));
```

Speaker notes

As English left-to-right readers, function chaining is more intuitive and natural...

RxJS Operator chaining is composition...

RxJS has a **pipe()** method that makes it easy to compose a chain of multiple operators into a single operation.

```
notifications$.pipe(

  filter(m => m.forUserId === 1),
  map(m => m.body)

)
```

Speaker notes

So we use the pipe method and pass it a list of operators we want to run in an order
And each operator will return an observable that will get handed to the next operator

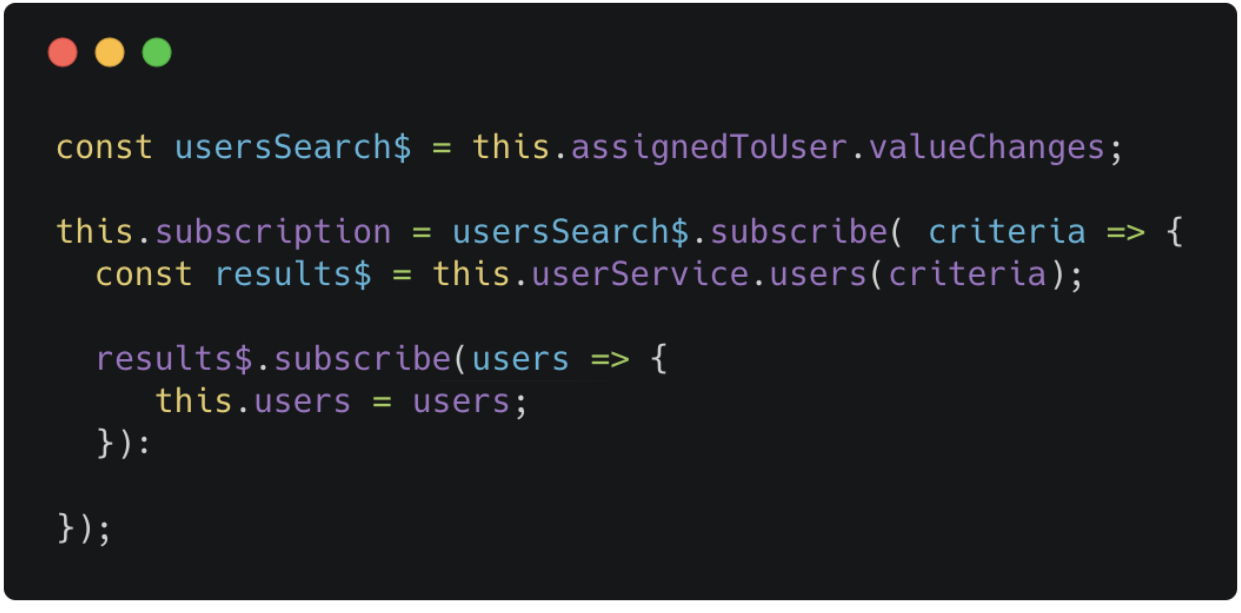
RxJS Operator Composition

3

```
const ticket$ : Observable<Ticket> = this.http.get<Ticket>(url);

const details$: Observable<string> = ticket$.pipe(
        filter(t => t.ticketID === 1),
        map(t => t.details)
      );



details$.subscribe(
  // extract the value from the stream to
  // render in the template

  (details:string) => this.ticketDetails = details
);
```

- **filter()** conditionally blocks stream propagation
- **map()** allows us to extract specific data parts or transform data

Using HttpClient - **Problem #1**



```
const usersSearch$ = this.assignedToUser.valueChanges;

this.subscription = usersSearch$.subscribe( criteria => {
  const results$ = this.userService.users(criteria);

  results$.subscribe(users => {
     this.users = users;
  }):

});
```

Problem: code calls to the server EVERY keystroke

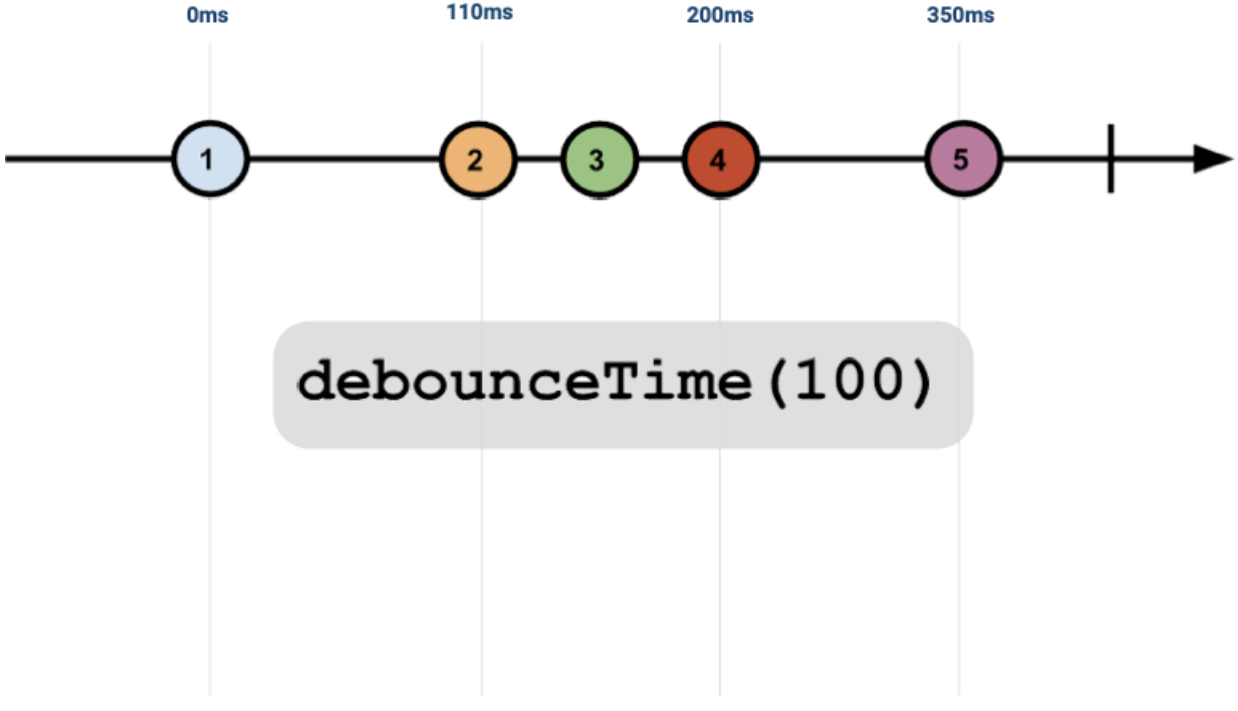
We also have 2 other issues... any guesses?

Speaker notes

Beyond just doing some transformation or filter, we can use operators to build elegant solutions
So we have this issue right now with our implementation…we are constantly making http calls
(show problem in browser)
Issues: need debounce, need distinctUntilChanged, guard against out-of-order

DebounceTime Operator

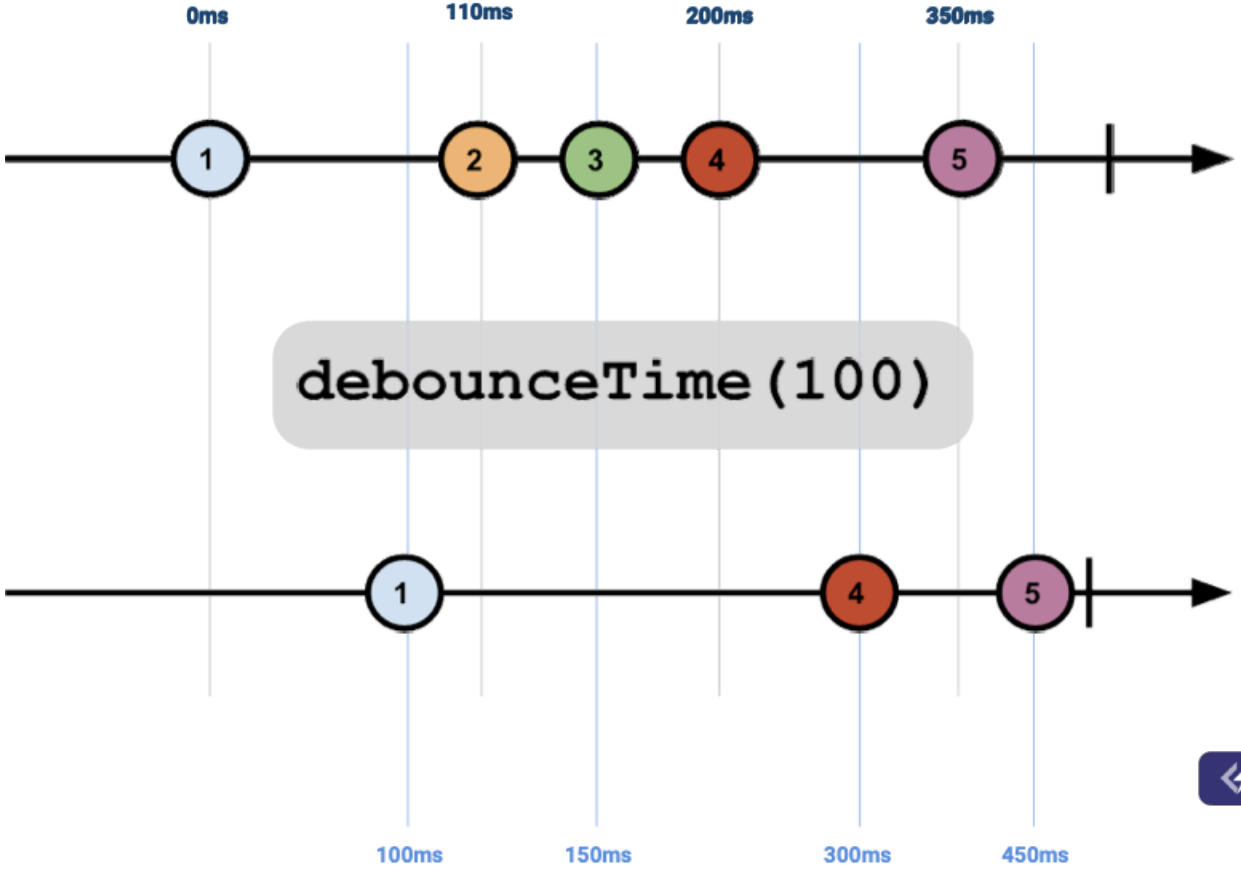
Filter the value emitted by an Observable until a particular timespan has passed without another value emitted.



StackBlitz Demo

DebounceTime Operator

Filter the value emitted by an Observable until a particular timespan has passed without another value emitted.



StackBlitz Demo

Speaker notes

With debounceTime we are saying, send me a value and as soon as I get one I am going to do something with it… and I'm going to bounce any other value that comes through within the next x amount of milliseconds

Oh and if you do send me another value within that time, guess what, I'm starting the timer all over again and won't accept another value for ANOTHER x milliseconds. :)

DebounceTime Operator

3

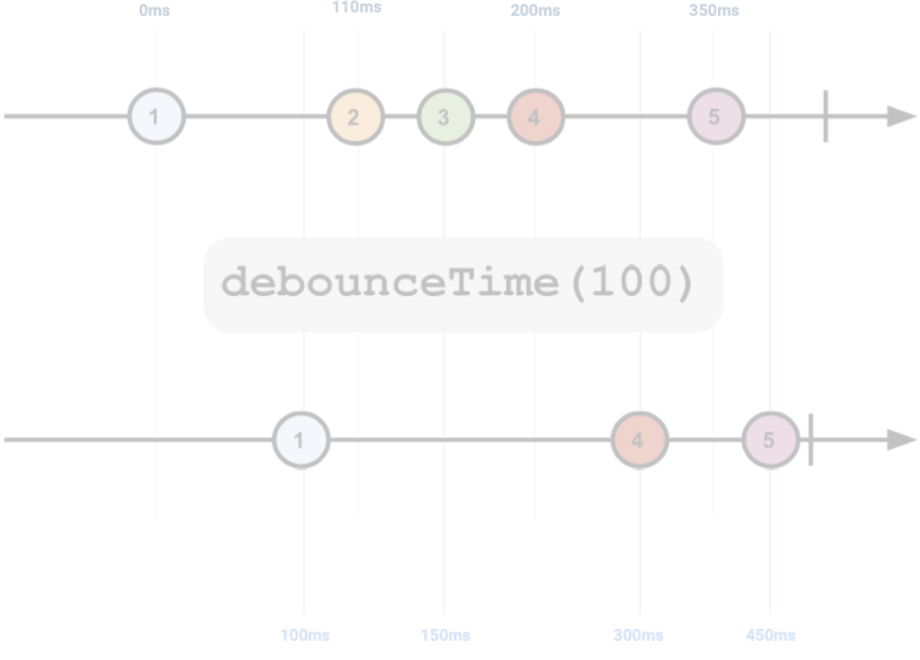
The **debounceTime** operator works by temporarily, internally caching the last emitted value. This means that debounceTime internally **manages state**.

Imagine doing this without an operator... !

DebounceTime Operator

The **debounceTime** operator works by temporarily, internally caching the last emitted value. This means that debounceTime internally **manages state**.

Imagine doing this without an operator... !



Fix Using HttpClient

3

```
@Component({})
class ContactsListComponent implements OnInit {
   term$: Observable<string>;       // stream of values from input control

  ngOnInit() {
    const validCriteria$ = this.terms$.pipe(
      debounceTime(400),
    );

    validCriteria$.subscribe(term => {
      this.contacts$ = this.search(term)
    });

  }
}
```

Fix: debounce the call to wait until the user stops typing

Speaker notes

But we can solve that with operators!
The fix is to debounce to wait for a break in keystrokes
Operator to do that logic
debounceTime

Using HttpClient - **Problem #2**

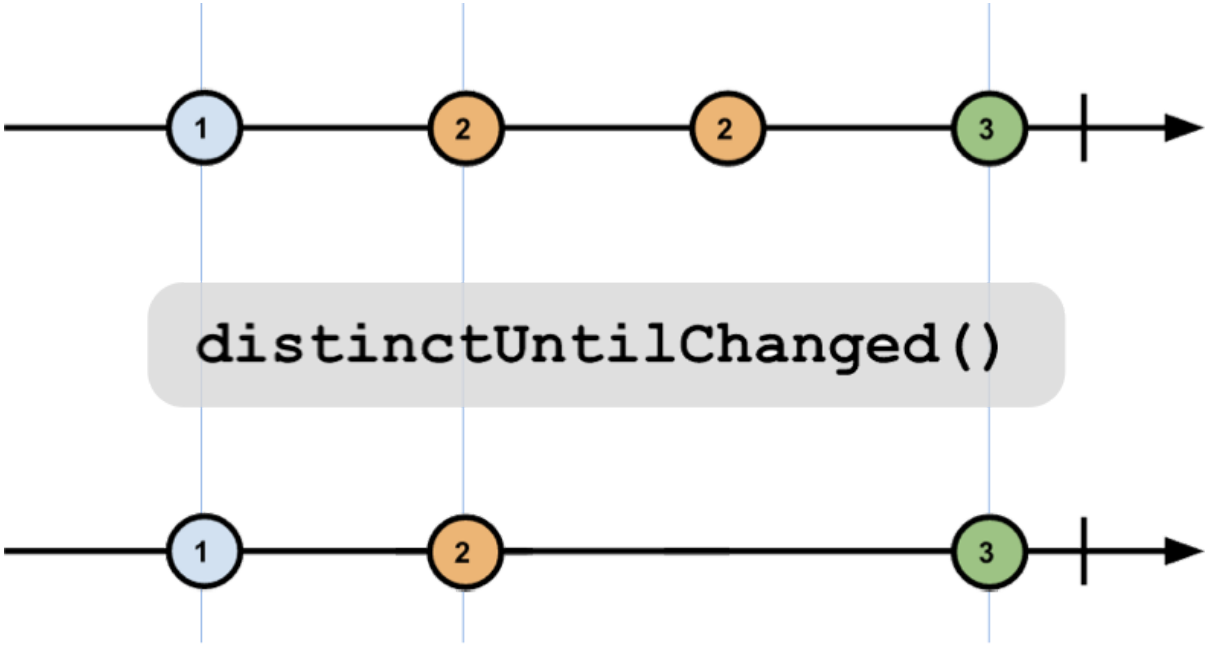

```
@Component({})
class ContactsListComponent implements OnInit {
   term$: Observable<string>;        // stream of values from input control

  ngOnInit() {
    const validCriteria$ = this.terms$.pipe(
      debounceTime(400),
    );

    validCriteria$.subscribe(term => {
      this.contacts$ = this.search(term)
    });

  }
}
```

Problem: user can type, then delete, and retype... with the same value after debounce.

DistinctUntilChanged Operator

3

Suppress duplicate items emitted consecutively by an Observable.



StackBlitz Demo

DistinctUntilChanged Operator

The **distinctUntilChanged** operator works by temporarily, internally caching the last emitted value and comparing to the current value. This means that distinctUntilChanged internally **manages state**.

Imagine doing this without an operator... !

Fix Using HttpClient


```
@Component({})
class ContactsListComponent implements OnInit {
   term$: Observable<string>;       // stream of values from input control

  ngOnInit() {
    const validCriteria$ = this.terms$.pipe(
      debounceTime(400),
      distinctUntilChanged()
    );

    validCriteria$.subscribe(term => {
      this.contacts$ = this.search(term)
    });

  }
}
```

Fix: on call server with a value if it is different than the previous value.

Speaker notes

But now we have a new scenario that is not ideal…if the user does some typing and deleting and then stops on the same value that they started with, the value pushes through and the same http call we just made gets made again.
Operator to do that logic
distinctUntilChanged

How do we consume DOM Events as a Stream

3

```
<mat-toolbar>

  <mat-form-field dividerColor="accent" class="trm-search-container">
    <input matInput (input)="search($event.target.value)" type="text">
  </mat-form-field>

  <mat-icon color="accent">search</mat-icon>
</mat-toolbar>
<mat-list>
  ...
</mat-list>
```

```
@Component({})
class ContactsListComponent implements OnInit {
   term$: Observable<string>;       // stream of values from input control
  }
}
```

Speaker notes

To use the power of RxJS operators, we need observables.
How do we listen of input events and send those through an observable stream?

RxJS Subject<T>

Subject implements both **Observable** and **Observer** interfaces.

- Use Observer interface to ***emit*** values using **next()**
- Use Observable ***read*** values using **pipe()** and **subscribe()**

```
import {Subject, Observable} from 'rxjs';

@Component({})
class ContactsListComponent implements OnInit {
  contacts$ : Observable<Contact[]>;

  ngOnInit() {
    const terms$ = this.channel.asObservable().pipe(
      debounceTime(400)
      distinctUntilChanged()
    );

    term$.subscribe(term => {
      this.contacts$ = this.search(term)
    });
  }

  private channel = new Subject<string>();
}
```

contact-list.component.ts

Speaker notes

Here in our imperative code, we prepare a stream to receive `search terms` and call the HttpService/REST server

Subject<T>

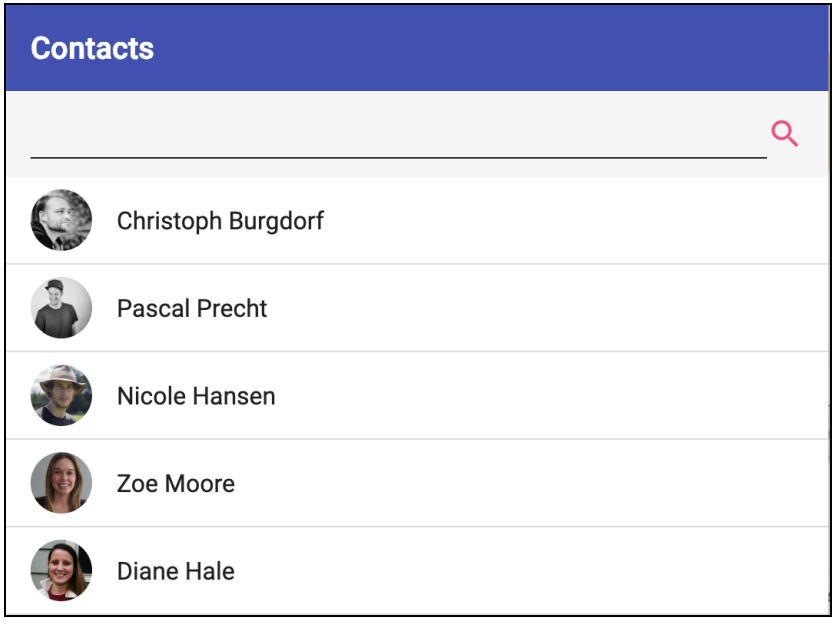
contact-list.component.html

```
<mat-toolbar>
  <mat-input-container>

    <input matInput
      (input)="channel.next($event.target.value)">

  </mat-input-container>
</mat-toolbar>

<mat-list>
  ...
</mat-list>
```

Speaker notes

When the values of the input control change,
we use the native DOM event to + the `channel` to emit the new search term.

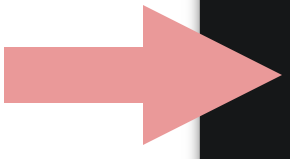
RxJS Lab 2: Throttle Search Requests



- Reduce server thrashing
- Use RxJS operators: **debounceTime**, **distinctUntilChanged**
- Use **Subject**<T>

Lab Exercise

Bad subscribe()



```
import {Subject, Observable} from 'rxjs';

@Component({})
class ContactsListComponent implements OnInit {
  contacts$ : Observable<Contact[]>;

  ngOnInit() {
    const terms$ = this.channel.asObservable().pipe(
      debounceTime(400)
      distinctUntilChanged()
    );

    term$.subscribe(term => {
      this.contacts$ = this.contactService.search(term)
    });
  }

  private channel = new Subject<string>();
}
```

Speaker notes

When using the async pipe, we SHOULD avoid using `subscribe()`

Refactor to remove subscribe...

```
import {Subject, Observable} from 'rxjs';

@Component({})
class ContactsListComponent implements OnInit {
  contacts$ : Observable<Contact[]>;

  ngOnInit() {
    const terms$ = this.channel.asObservable().pipe(
      debounceTime(400)
      distinctUntilChanged()
    );

    this.contacts$ = term$.pipe(
      map(term => this.contactService.search(term))
    )
  }

  private channel = new Subject<string>();
}
```

Refactor to remove subscribe...

```
import {Subject, Observable} from 'rxjs';

@Component({})
class ContactsListComponent implements OnInit {
  private channel = new Subject<string>();
  private term$ = this.channel.asObservable();

  contacts$: Observable<Contact[]> = this.term$.pipe(
    debounceTime(400)
    distinctUntilChanged()
    map(term => this.contactsService.search(term))
  );

}
```

Speaker notes

This looks much cleaner. Cool...

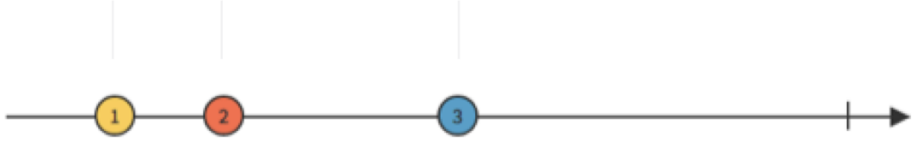
Problem #1: Nested Observables

```
import {Subject, Observable} from 'rxjs';

@Component({})
class ContactsListComponent implements OnInit {
  private channel = new Subject<string>();
  private term$ = this.channel.asObservable();

  contacts$: Observable<Contact[]> = this.term$.pipe(
    debounceTime(400)
    distinctUntilChanged()
    map(term => this.contactsService.search(term))
  );

}
```

Speaker notes

Except this does not work... why?
The stream is supposed to return Contact[], yet map returns an Observable instead of a array

Types coming through Observable streams

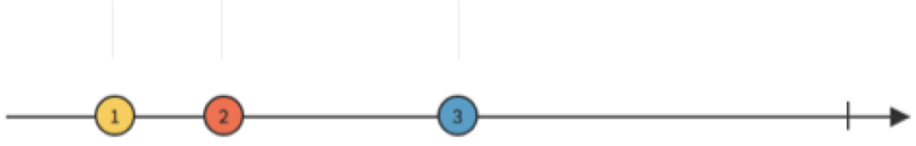


An observable may emit value of **any** JavaScript type:

- number
- string
- boolean
- object
- class instance
- etc.

StackBlitz Demo

Types coming through Observable streams



Since observables may emit value of **any** JavaScript type:

What if the types emitted are **Observables**?

StackBlitz Demo

Speaker notes

But they don’t have to be your basic types…they could be anything, including observables! What???

Types coming through Observable streams

3

```
import {Subject, Observable} from 'rxjs';

@Component({})
class ContactsListComponent implements OnInit {
  private channel = new Subject<string>();
  private term$ = this.channel.asObservable();

  contacts$: Observable<Contact[]> = this.term$.pipe(
    debounceTime(400)
    distinctUntilChanged()
    map(term => this.contactsService.search(term))
  );

}
```

This will not work because **contactsService.search()** returns an observable.

Speaker notes

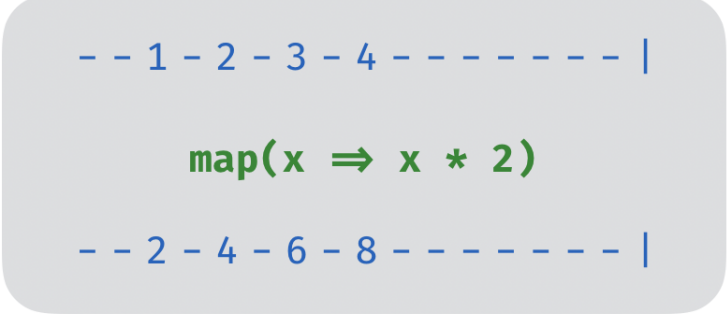
So you could imagine, and maybe you have got into this situation (I certainly did a lot)…
We can use this pipe and these operators to set up the flow of our observable work to be catered towards what we need, surely we could do something there to not have to subscribe to multiple streams? Let's just use map to get the incoming form value and change it to return the users from that user service stream…
But this doesn't work the way we might think…

Types coming through Observable streams

3

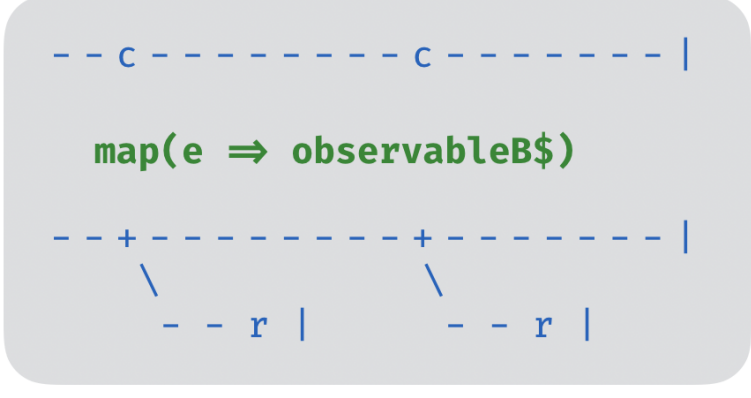
1st Order

- Observable of standard types
- Easy to work with values
- Single Subscribe()



Higher Order

- Observable of Observable
- Could use nested subscribes()... **No!**
- Need to flatten if we want to get values

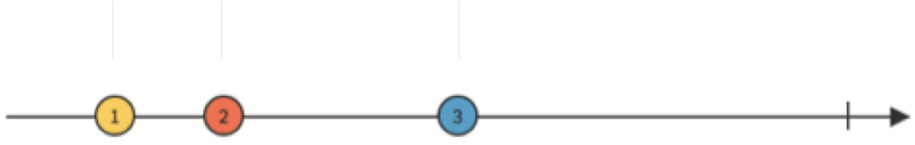


+ is an inner observable

Speaker notes

Need to understand these because we can easily get confused as we start to try different operators…
So we can classify observables based on what types they return…first order for those basic values and higher order for ones that return observables.
Using first order is common for transforming data
Higher order example is new data from form field that needs to subscribe to httpclient observable
Consider the userSearch we did earlier: the `assignedToUser.valueChange` observable is subscribed and the value

Common **1st-Order** Operators



map, filter, reduce

(when working with an emitted value)

Speaker notes

we use these all the time
Can find ourselves returning a value from here…or could find ourselves returning an observable

Common **Higher Order** Values

If we do a **map(e => observableB$)**

and end up with a higher order...

We could manually flatten to get the value(s) from those inner observables. This would require a nested subscribe()...

How do we avoid nested subscribes() **AND** flatten the nested observables?

Flattening and Mapping **Higher-Order** Operators

switchMap, mergeMap, concatMap, forkJoin

(when you need to combine multiple observables into a single operation and get their nested values)

Higher-Order Scenarios

switchMap, mergeMap, concatMap, forkJoin

- Nested subscribes...
- Combining observables...
- Cancelling observable in favor of another...
- Sequencing observables...

Speaker notes

So how do we get ourselves into these scenarios where we have higher order observables?
Well we can see it in the code we just wrote where we subscribe to the form value and then subscribe to the http call
We aslo end up in scenarios where we want to work with values from multiple streams
And we will have times where we want to start a new instance of an existing stream and cancel the previous.

Problem #1: Nested Observables

3

```
import {Subject, Observable} from 'rxjs';

@Component({})
class ContactsListComponent implements OnInit {
  private channel = new Subject<string>();
  private term$ = this.channel.asObservable();

  contacts: Contact[];

  ngOnInit() {
    this.term$.pipe(
      debounceTime(400)
      distinctUntilChanged()
      map(term => this.contactsService.search(term))
    ).subscribe(contacts => {
      this.contacts = contacts;
    });
  }

}
```

Speaker notes

Here we are not using a async pipe, so we subscribe to manually extract the raw data
But this does not work because map() returns an Observable.

Bad Solution:

Use Nested Subscribes

```
export class ContactsListComponent {
  contacts: Contacts[];

  ngOnInit() {
    const searchCriteria$ = this.channel.asObservable();

    searchCriteria$.pipe(
      map(criteria => {
        let pending: Subscription;
        const cancelPending = () => {
          if (pending) pending.unsubscribe();
          pending = null;
        };

        cancelPending();

        return Observable.create(observer => {
          try {
            const response$ = this.contactsService.search(criteria);

            pending = response$.subscribe(contacts => {
              observer.next(contacts);
              observer.complete();
            });

          } catch (e) {
            observer.error(e);
          }

          return () => {
            cancelPending();
            observer.complete();
          };
        });

      })
    );
  }
}
```

3

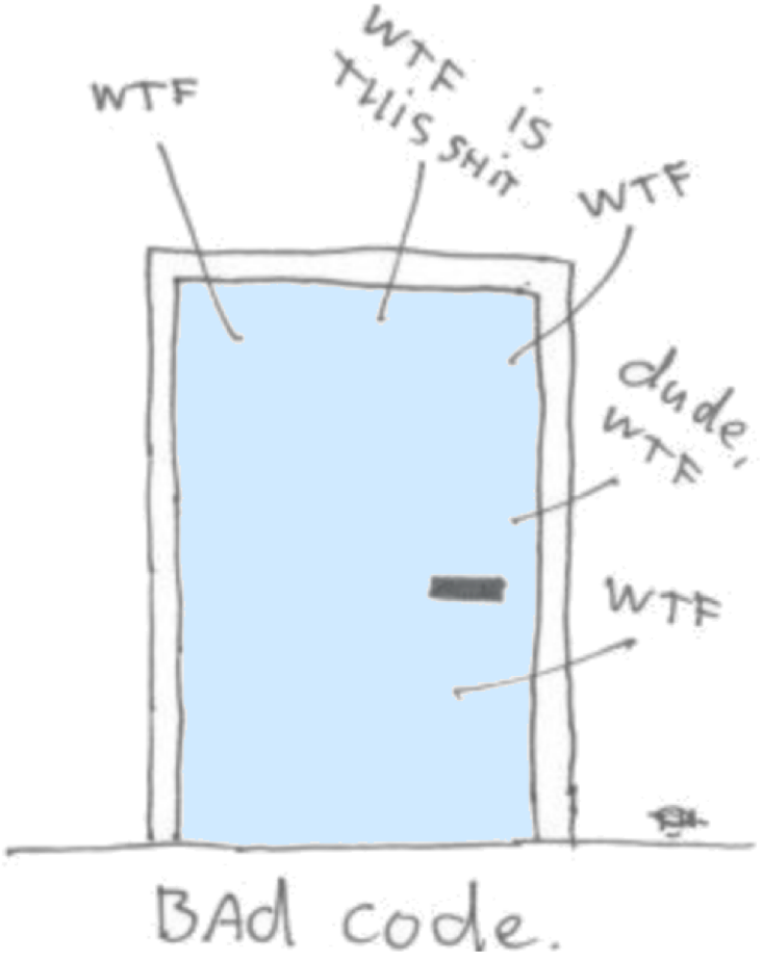
Speaker notes

Take a moment to study this code...
Not only is this `nasty` code, we also
have an out-of-order potential issue here. Bad!

Bad Solution:

3

Use Nested Subscribes



Bad Solution:

Use Nested Subscribes

```
export class SearchTicketsComponent {
  users: string[];

  ngOnInit() {
    const searchCriteria$ = this.assignedToUser.valueChanges;

    searchCriteria$.pipe(
      map(criteria => {
        let pending: Subscription;
        const cancelPending = () => {
          if (pending) pending.unsubscribe();
          pending = null;
        };

        cancelPending();

        return Observable.create(observer => {
          try {
            const response$ = this.userService.users(criteria);

            pending = response$.subscribe(list => {
              observer.next(list);
              observer.complete();
            });

          } catch (e) {
            observer.error(e);
          }

          return () => {
            cancelPending();
            observer.complete();
          };
        });

      })
    );
  }
}
```

3

Speaker notes

Let's walk through this code.
I am sure you will agree that this is non-trivial code. A better solution is needed...

Good Solution: Using **switchMap** operator.

3

```
import {Subject, Observable} from 'rxjs';

@Component({})
class ContactsListComponent implements OnInit {
  private channel = new Subject<string>();
  private term$ = this.channel.asObservable();

  contacts: Contact[];

  ngOnInit() {
    this.term$.pipe(
      debounceTime(400)
      distinctUntilChanged()
      switchMap(term => this.contactsService.search(term))
    ).subscribe(contacts => {
      this.contacts = contacts;
    });
  }

}
```

When we get a new value...

switch to watch a nested observable

and use that nested observable's data

Best Solution: SwitchMap + **async** pipe

3

```
import {Subject, Observable} from 'rxjs';

@Component({})
class ContactsListComponent implements OnInit {
  private channel = new Subject<string>();
  private term$ = this.channel.asObservable();

  contacts$: Observable<Contact[]> = this.term$.pipe(
    debounceTime(400)
    distinctUntilChanged()
    switchMap(term => this.contactsService.search(term))
  );
}
```

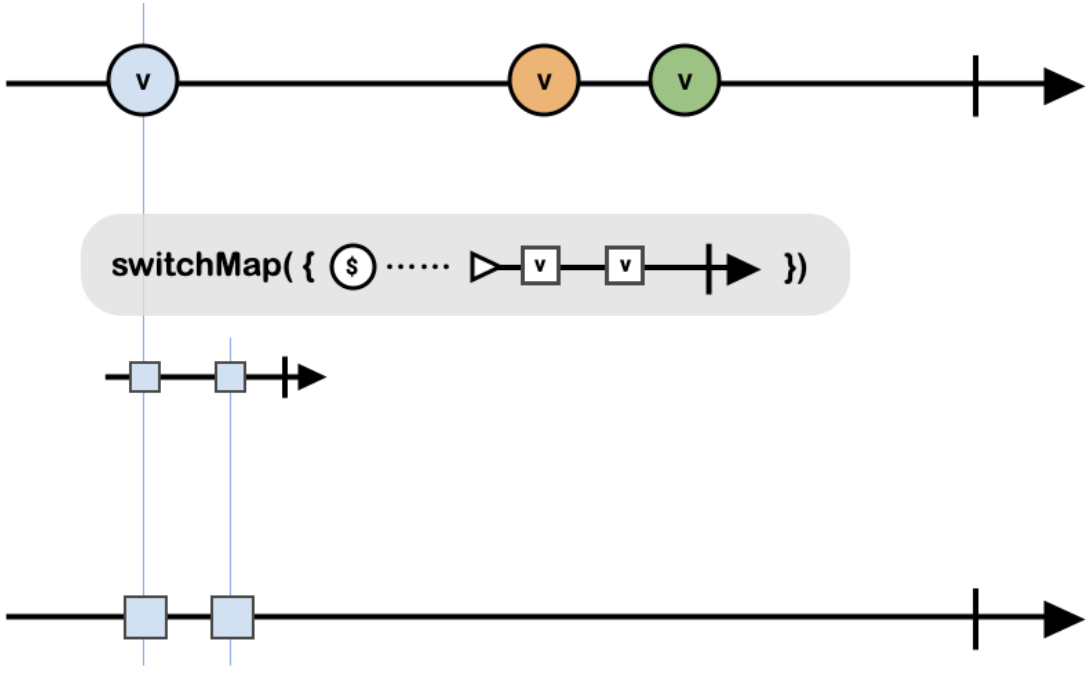
```
<mat-list>
  <a mat-list-item
    title="View {{contact.name}} details"
    [routerLink]="['/contact', contact.id]"
    *ngFor="let contact of contacts$ | async; trackBy:trackByContactId">

    <img mat-list-avatar
         class="circle"
         alt="Picture of {{contact.name}}"
         [src]="contact.image" >
    <h3 mat-line>{{contact.name}}</h3>
  </a>
</mat-list>
```

Speaker notes

The previous solution had a memory leak... were still missing subscription management.
Here we solved some stuff with operators so now by the time we call subscribe we have the data shape we want…
so we can ditch that subscribe in the class in favor of the async pipe!
No subscription management
We are simply composing the workflow for the data we care about

switchMap Operator

Transform the items emitted by an Observable into inner Observables, complete previous inner Observable, emit values of the inner observable.

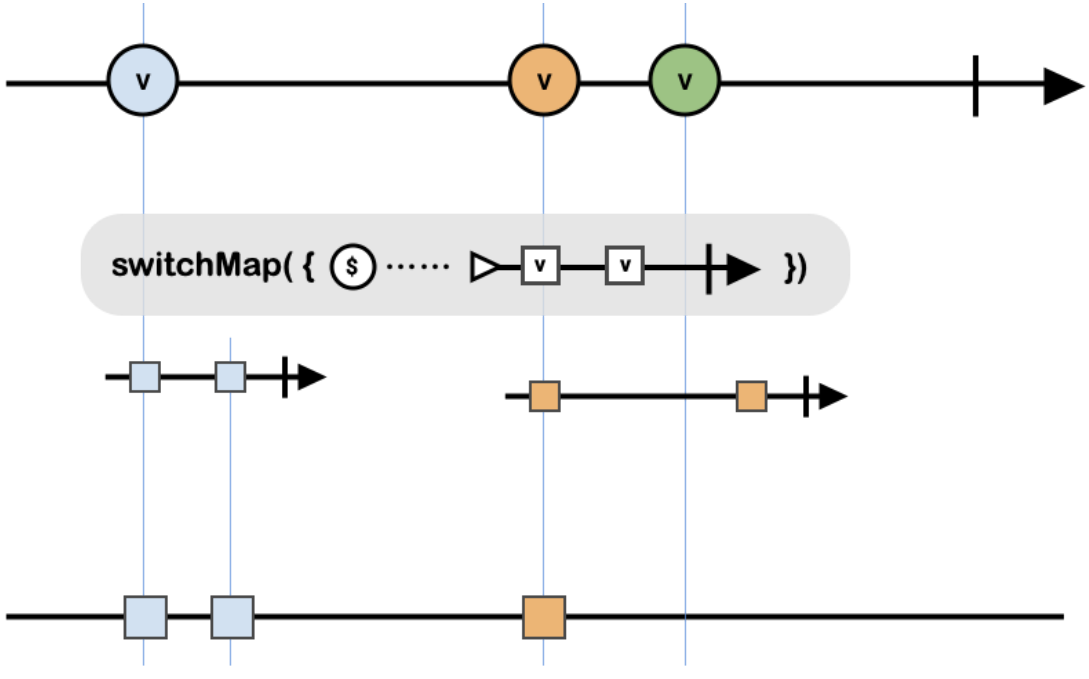


Speaker notes

Let’s break down what switchMap is doing…
So if we are subscribed to new values coming from a form field, and we use each new value to call an Http service that returns a set of items…

switchMap Operator

Transform the items emitted by an Observable into inner Observables, complete previous inner Observable, emit values of the inner observable.



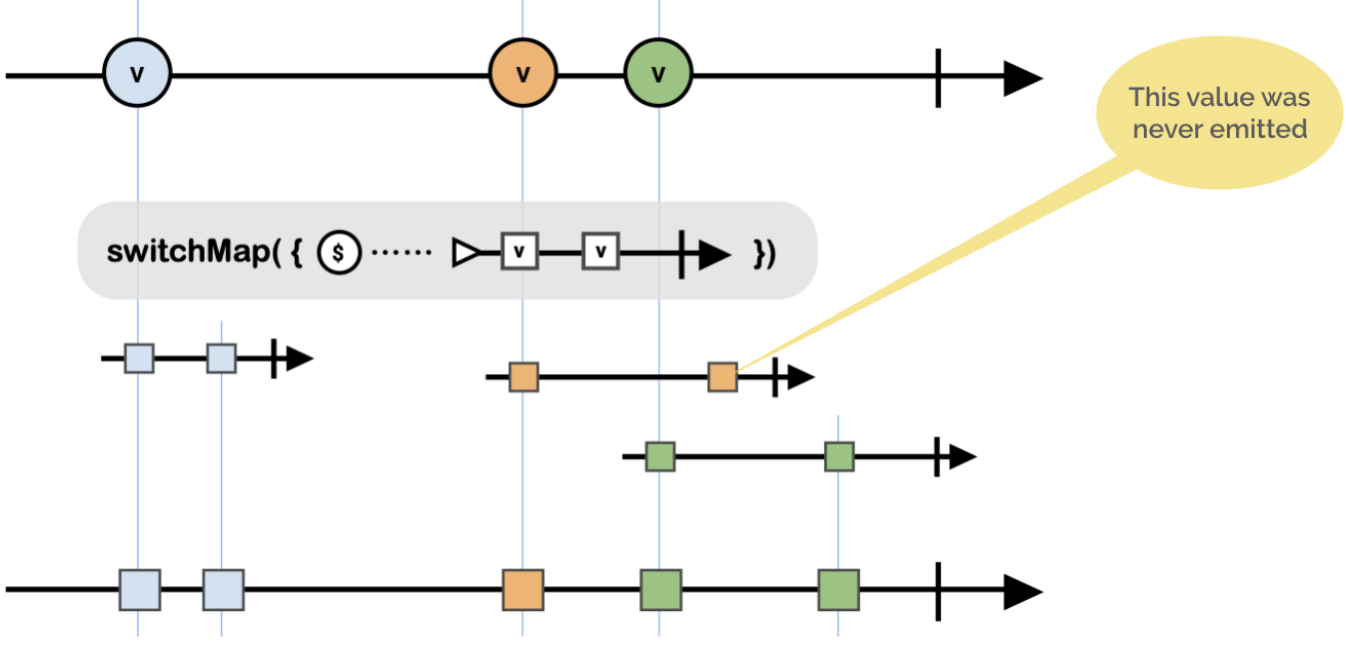
Speaker notes

When a new value comes in it will unsubscribe from the previous observable, then will switch to subscribe to the new one
And remember, the unsubscribe will start a teardown, so if that observable has teardown logic it will run that (the HttpClient will cancel it’s call in its Observable execution) winner winner chicken dinner!

switchMap Operator

3

Transform the items emitted by an Observable into inner Observables, complete previous inner Observable, emit values of the inner observable.



Speaker notes

And if we happen to use switchMap on an observable that keeps running even if you unsubscribe, well that's fine. It can keep doing its thing…but our flow is no longer connected to it, so it won't receive any additional values from it.

Let's look at a scenario in which switchMap would be useful.
If our application's cart shows the total cost of the items plus the shipping, each change to the cart's content would

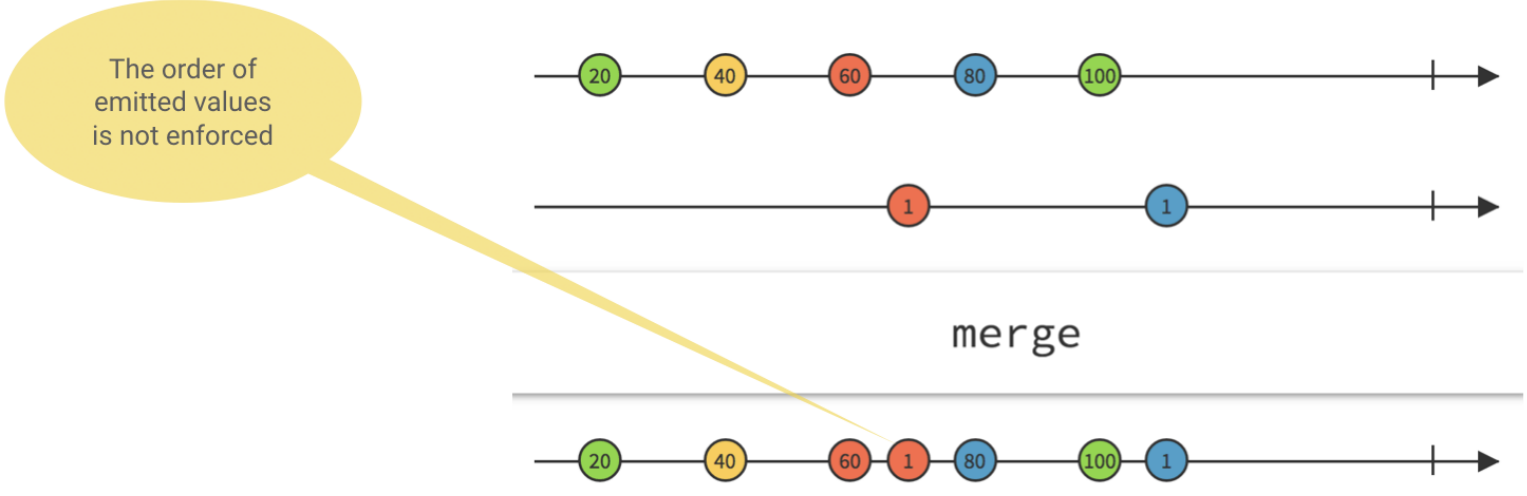
Other common operators

switchMap, mergeMap, concatMap, forkJoin

- Nested subscribes...
- Combining observables...
- Cancelling observable in favor of another...
- Sequencing observables...

Merge

Combine multiple observables by merging their emissions.



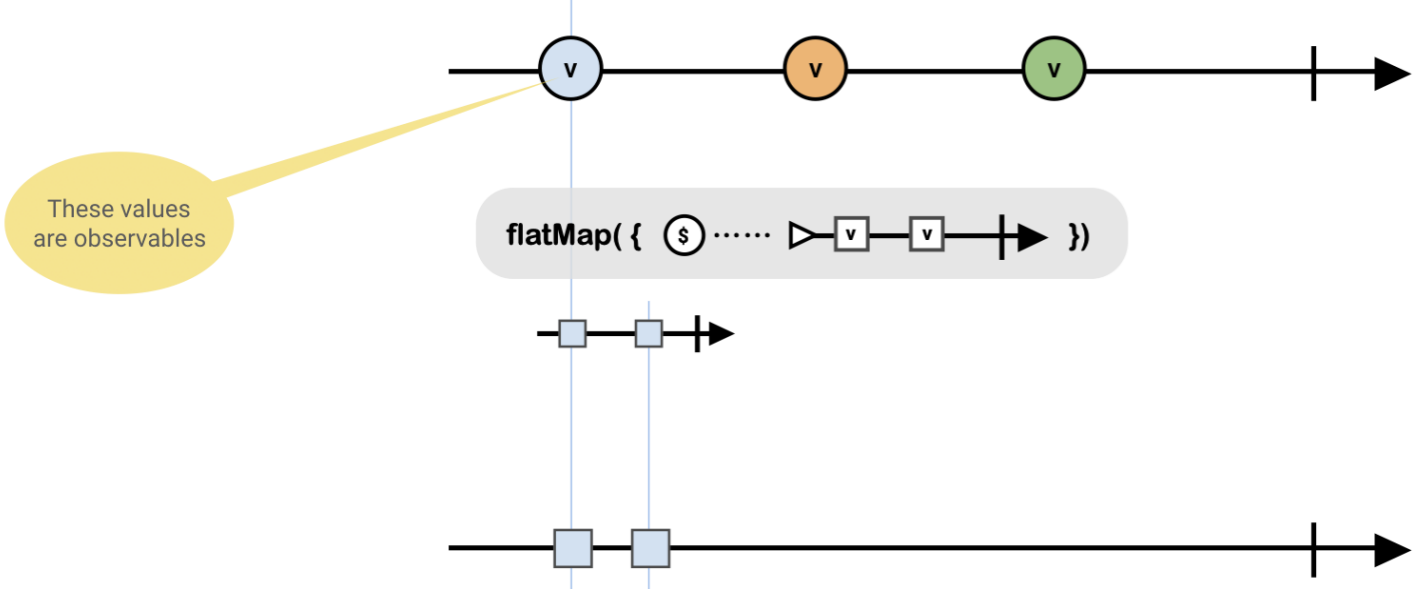
StackBlitz Demo

Speaker notes

We can get the values from multiple observables fed through a single observable
(read bubble)
What does it solve

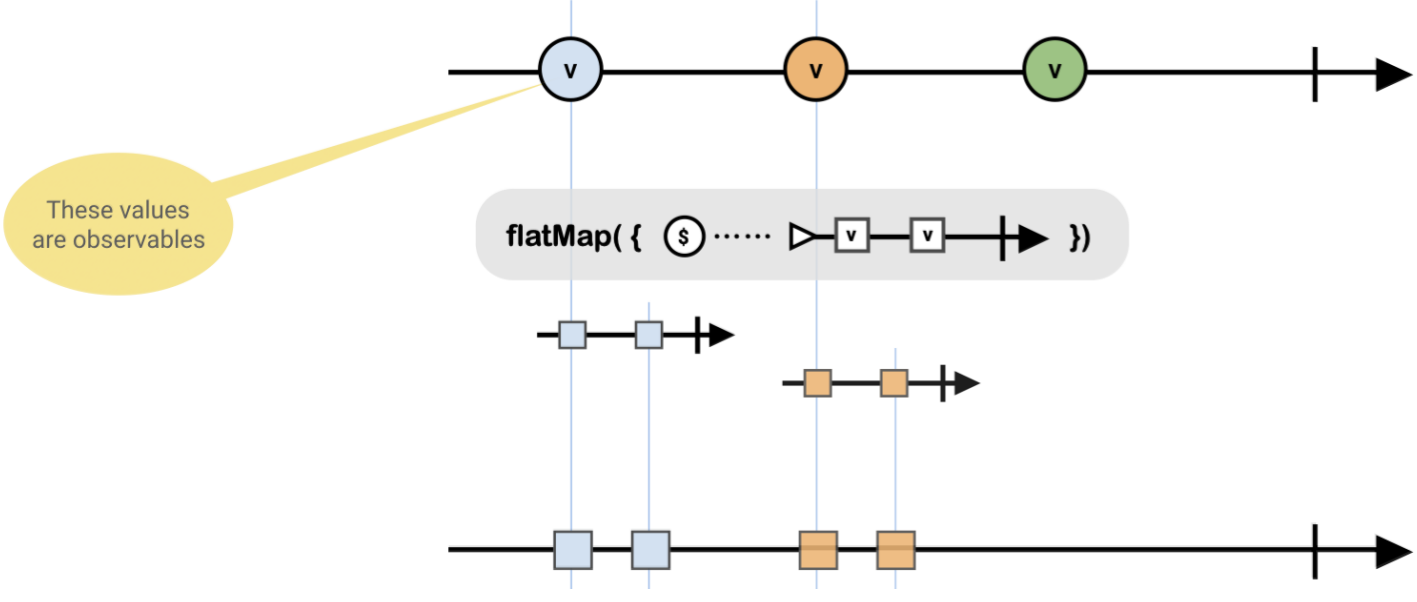
mergeMap Operator (aka flatMap)

When the items emitted by an Observable are **inner observables**, then **merge** the items of those inner observables into the outer stream.



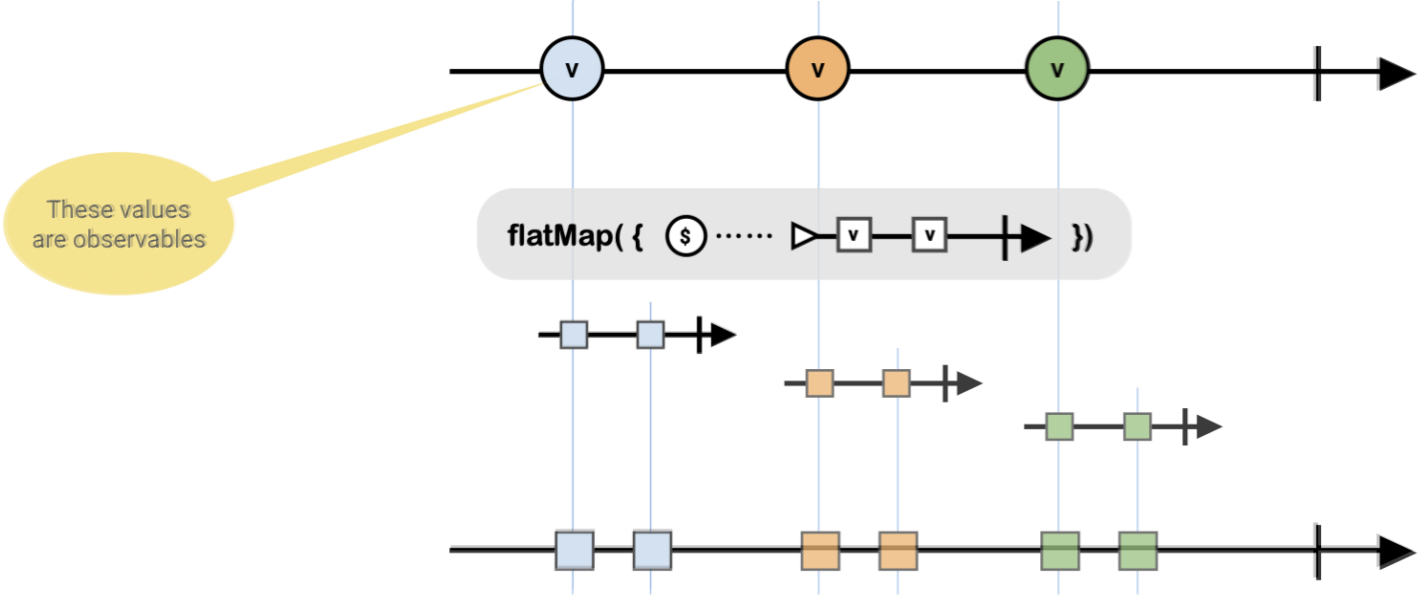
mergeMap Operator (aka flatMap)

When the items emitted by an Observable are **inner observables**, then **merge** the items of those inner observables into the outer stream.



mergeMap Operator (aka flatMap)

When the items emitted by an Observable are **inner observables**, then **merge** the items of those inner observables into the outer stream.



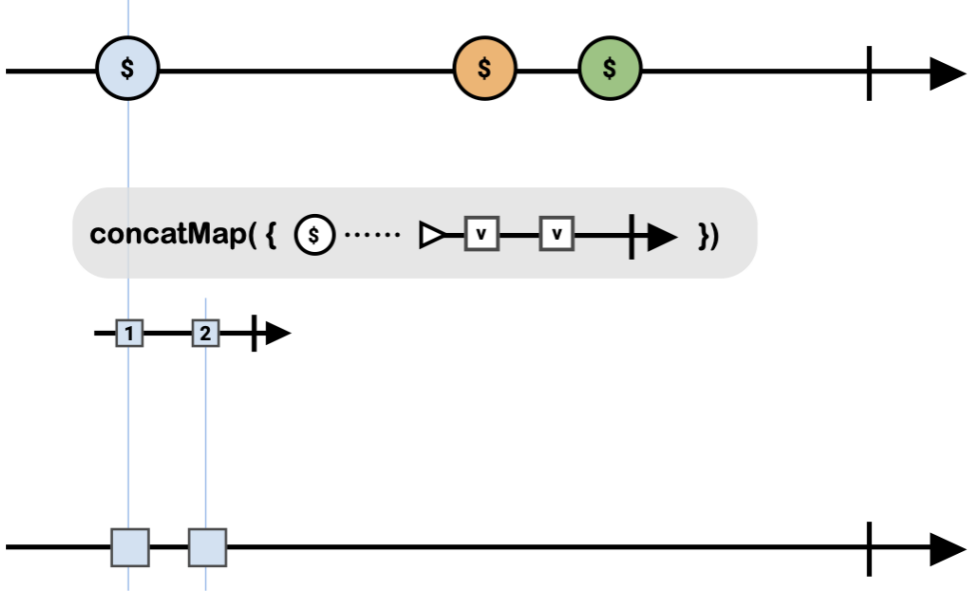
Speaker notes

For example, if the user clicks the remove buttons of the first and second items of a Cart.

It’s possible that the removal of the second item might occur before the removal of the first.
With our cart, the ordering of the removals doesn’t matter, so using mergeMap instead of switchMap fixes a bug where the remove was cancelled.

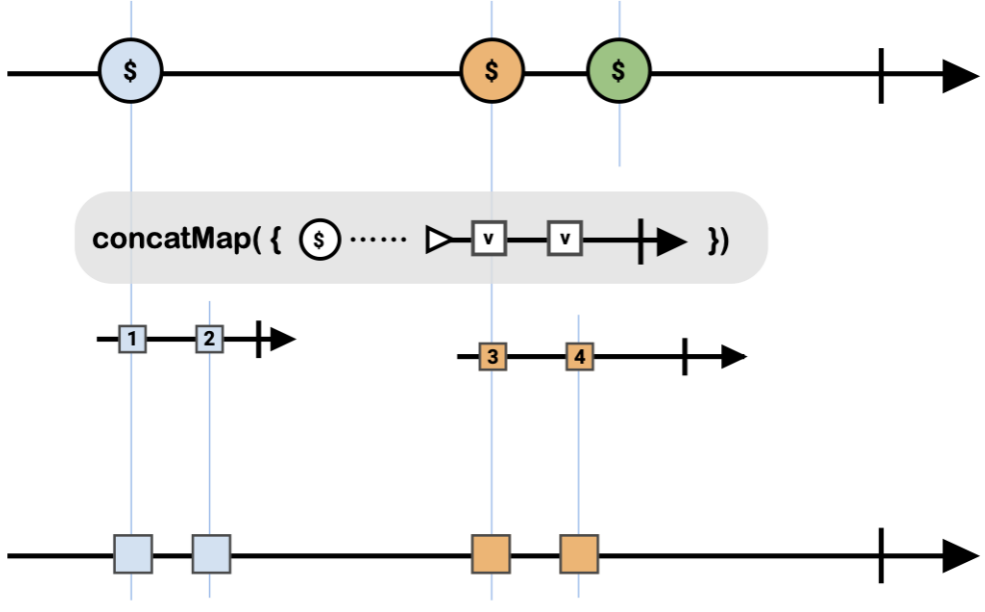
concatMap Operator

Sequence observables by subscribing to the Observables in order. Only when the previous subscription **completes,** then subscribe to the next in the queue.Do not cancel previous subscriptions.



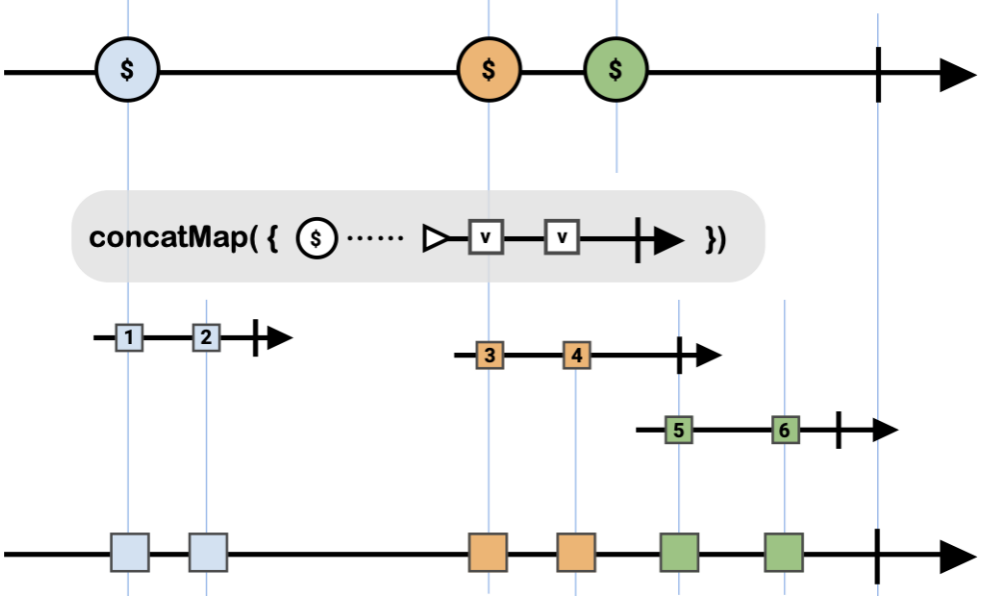
concatMap Operator

Sequence observables by subscribing to the Observables in order. Only when the previous subscription **completes,** then subscribe to the next in the queue.Do not cancel previous subscriptions.



concatMap Operator

Sequence observables by subscribing to the Observables in order. Only when the previous subscription **completes,** then subscribe to the next in the queue.Do not cancel previous subscriptions.

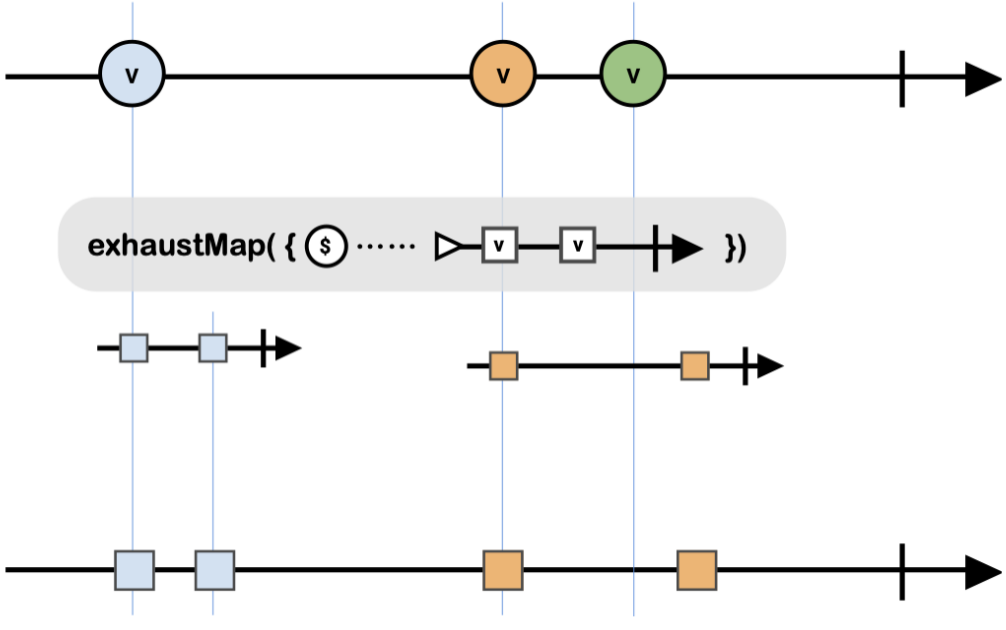


Speaker notes

For example, if our shopping cart has a button for increasing an item’s quantity, it’s important that the dispatched actions are handled in the correct order.
Otherwise, the quantities in the frontend’s cart could end up out-of-sync with the quantities in the backend’s cart.

exhaustMap Operator

Subscribe **first** to the inner Observables; ignore all other outer events until inner completes. Do not queue pending events.



Speaker notes

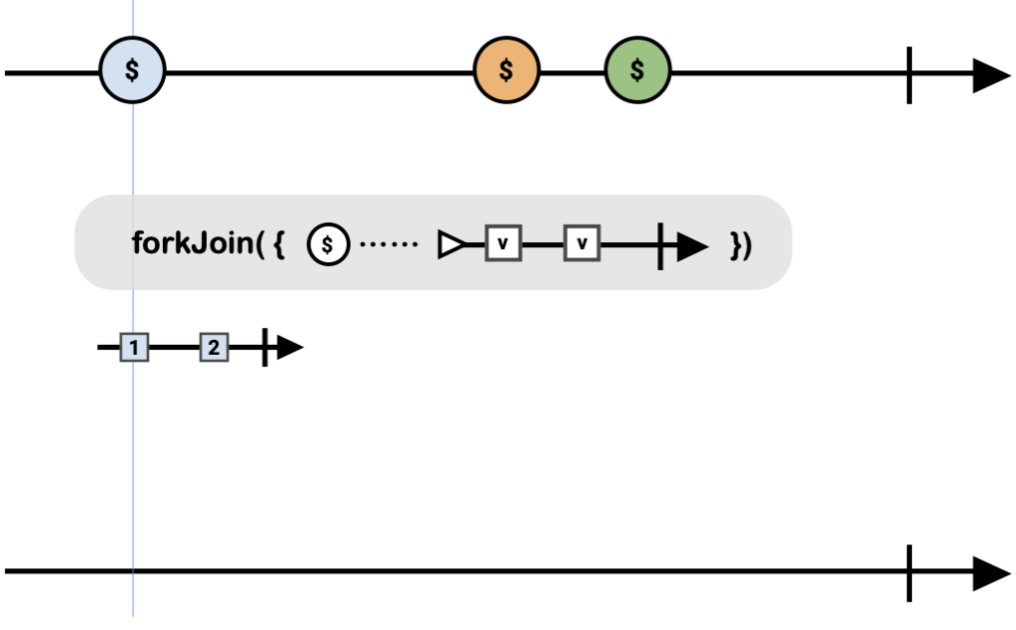
Let’s look at a scenario in which exhaustMap could be used.
There’s a particular type of user with whom developers should be familiar: the incessant button Refresh clicker.
When the incessant button clicker clicks a button and nothing happens, they click it again. And again. And again.

When you need a **Higher-Order** operator

- use **concatMap** with events should be sequenced with ordering preserved... events are **not aborted** nor ignored
- use **mergeMap** with events when ordering is unimportant. ... events are **not aborted** nor ignored
- use **switchMap** with events that should be **aborted** when another event of the same type arrives.
- use **exhaustMap** with events that should ignored while an event of the same type is still pending **(not complete).**

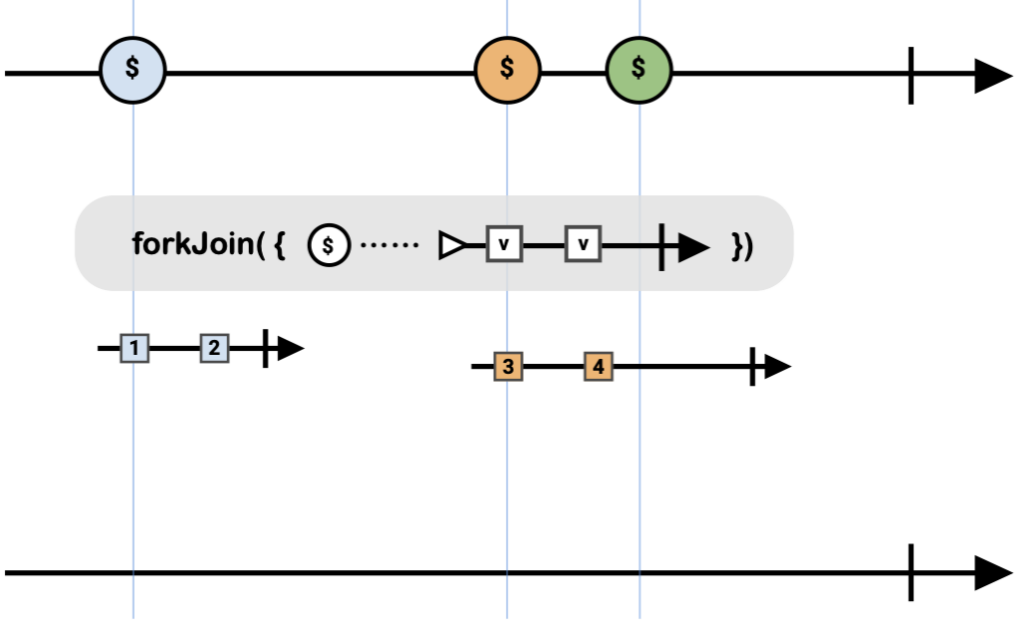
forkJoin Operator

When **all** of the Observables **complete** then an array of the the last value emitted from each.



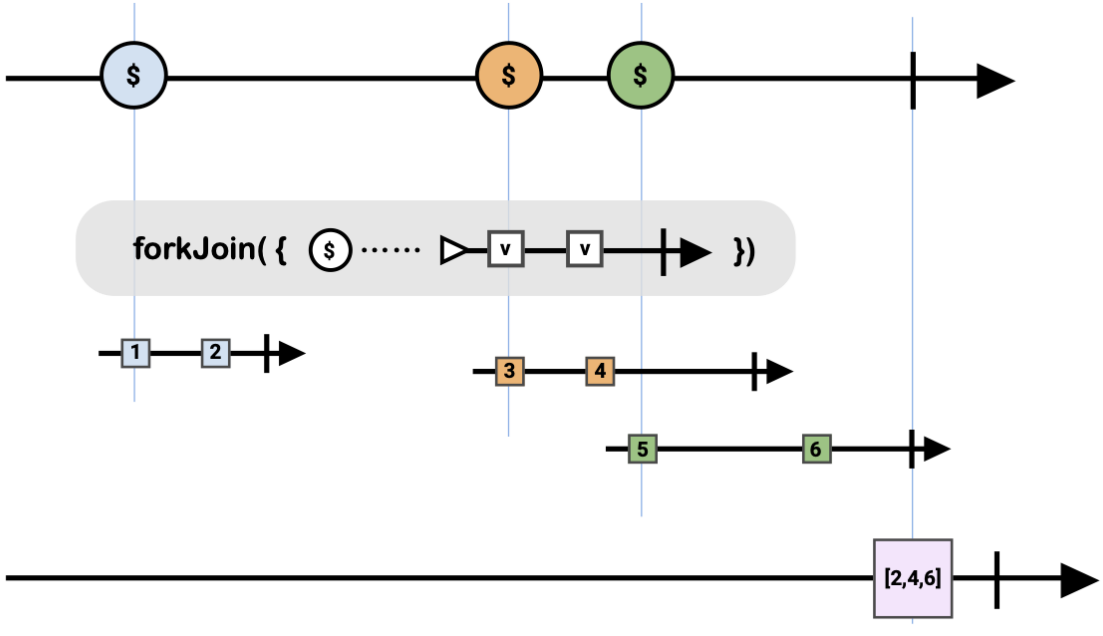
forkJoin Operator

When **all** of the Observables **complete** then an array of the the last value emitted from each.



forkJoin Operator

When **all** of the Observables **complete** then an array of the the last value emitted from each.



RxJS Lab 3: Use switchMap

```
import {Subject, Observable} from 'rxjs';

@Component({})
class ContactsListComponent implements OnInit {
  private channel = new Subject<string>();
  private term$ = this.channel.asObservable();

  contacts$: Observable<Contact[]> = this.term$.pipe(
    debounceTime(400)
    distinctUntilChanged()
    switchMap(term => this.contactsService.search(term))
  );
}
```

- Use **switchMap** operator
- Use **merge**(...) for initial loads + search loads
- Keep using **async** pipe
- Remove use of **subscribe()**

Lab Exercise

RxJS Lab 4: Refactor to Service

```
@Component({
  selector: 'trm-contacts-list',
  templateUrl: './contacts-list.component.html',
  styleUrls: ['./contacts-list.component.css']
})
export class ContactsListComponent implements OnInit {

  contacts$: Observable<Array<Contact>>;
  private terms$ = new Subject<string>();

  constructor(private contactsService: ContactsService) {}

  ngOnInit () {
    this.contacts$ = merge(
      this.contactsService.search(this.terms$),
      this.contactsService.getContacts()
    );
  }
}
```

Lab Exercise

What we will cover?

1. Observable Concepts
2. Consuming Observables
3. RxJS Operators
4. Creation Operators

Why would we **create** Observables?

4

- Need to deliver values over time
- Some async operation
- Shared State
- Testing (mocks)
- Cancelable operations

Speaker notes

A timer
Long running computation that we may want to cancel or transform
Something we can change and have other components get pushed updates
Mocking observables like http calls, forms, etc

Ways to **create** Observables:

- **of( )**
- from( )
- interval( )
- timer()
- **fromEvent( )**

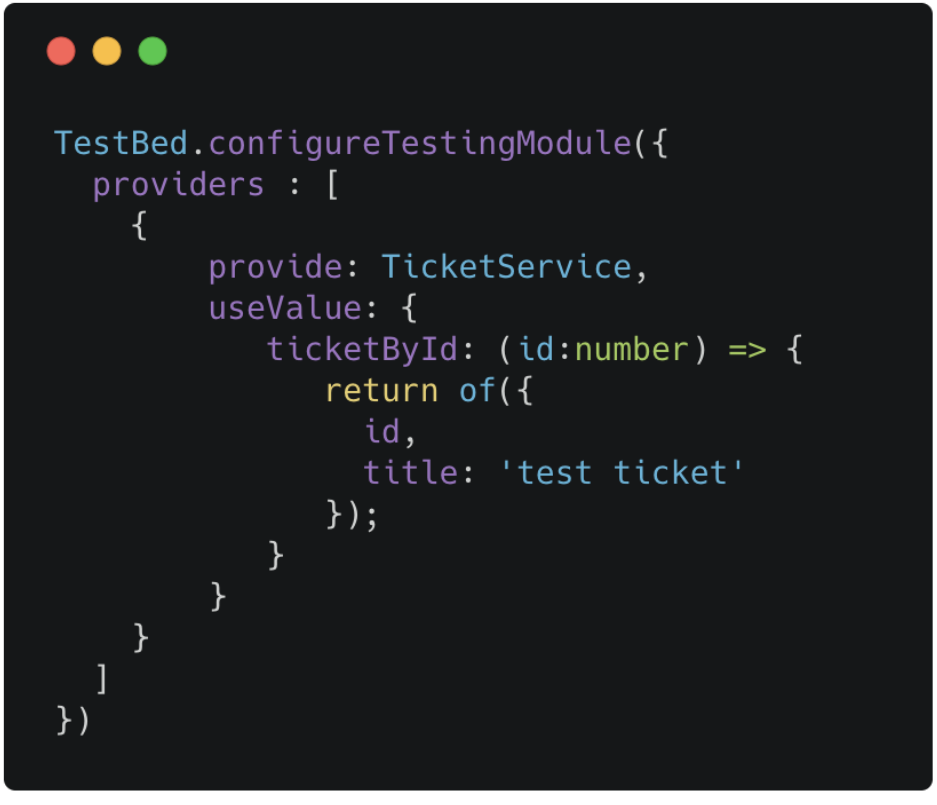
... and more

- **Observable.create( )**
- new Observable( )

Speaker notes

creation operators are great for testing, easy to create mock streams
.create method is usually what we'll use when we want to roll our own observable logic (observable execution)

Using creation **of( )** operator



```
TestBed.configureTestingModule({
  providers : [
    {
        provide: TicketService,
        useValue: {
           ticketById: (id:number) => {
              return of({
                id,
                title: 'test ticket'
              });
           }
        }
    }
  ]
})
```

Here we are building a mock HttpClient service!

Creating your Own Observable

4

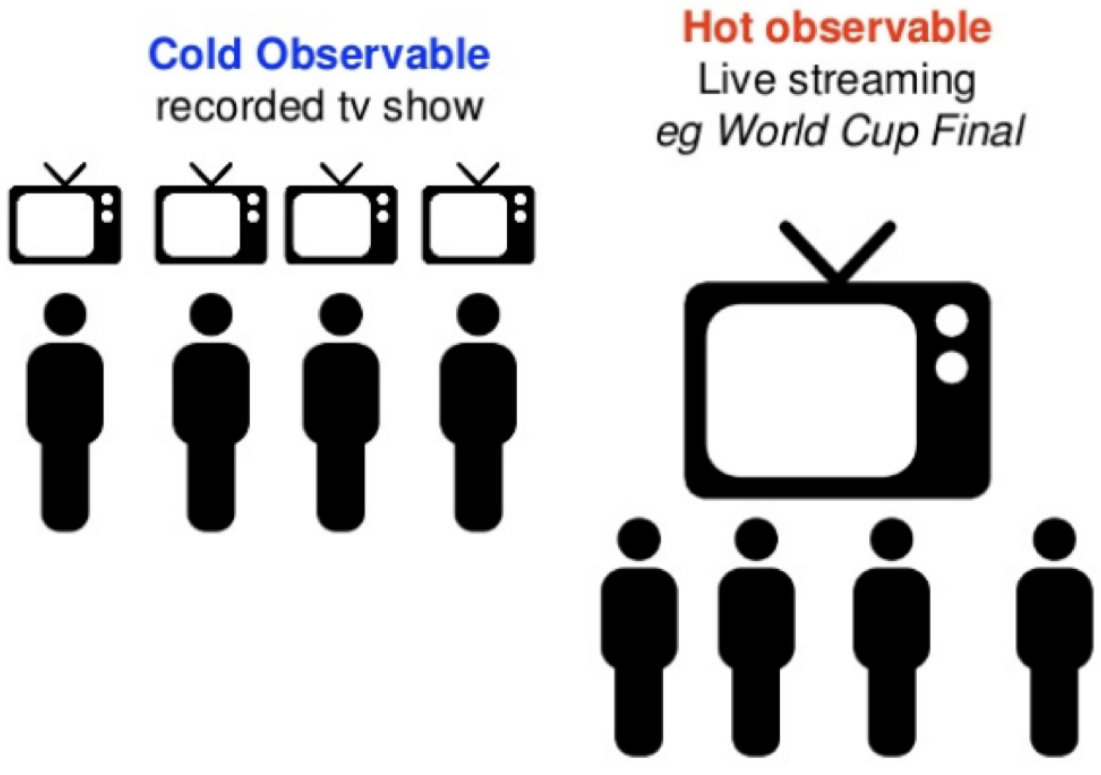
```
import { Observable } from 'rxjs';

const custom$ = Observable.create( observer => {

  observer.next(1);
  observer.next(2);
  observer.next(3);

  observer.complete();

});
```

Contains our custom lo

Speaker notes

What we're doing here is using the .create method to build the observable wrapper around a function we give it (so it can do its observable things)…
The function we give it is going to be what the observable runs when the call to .subscribe is made
And as you can see, that function we create takes in an observer, which is that set of callbacks a consumer provides when they call subscribe.
(consider switching over to IDE and writing this, then naming that function "subscribe", break it down like Andre

By default, Observable are COLD



Things to consider:

4

- **Try/catch** around next calls that may throw errors
- Return a **tear-down function** and use it to dispose internal resources

A "Responsible" Custom Observable

4

```
import { Observable } from 'rxjs';

const custom$ = Observable.create( observer => {

  const intervalId = setInterval(() => {
    try {

      observer.next(server.getStatus());

    } catch (e) {
        observer.error('oops');
    }
  }, 1000);

  // Tear-down function...
  return () => {
    clearInterval( intervalId );
  };

});
```

Speaker notes

here we have an observable that will run a setInterval, so it should handle cleaning that up in the teardown function
And its observable execution makes use of some external code that could fail, so try/catch to be able to emit the error if need be

Unicast vs Multicast

Unicase

Observable only send notifications to a single Observer.

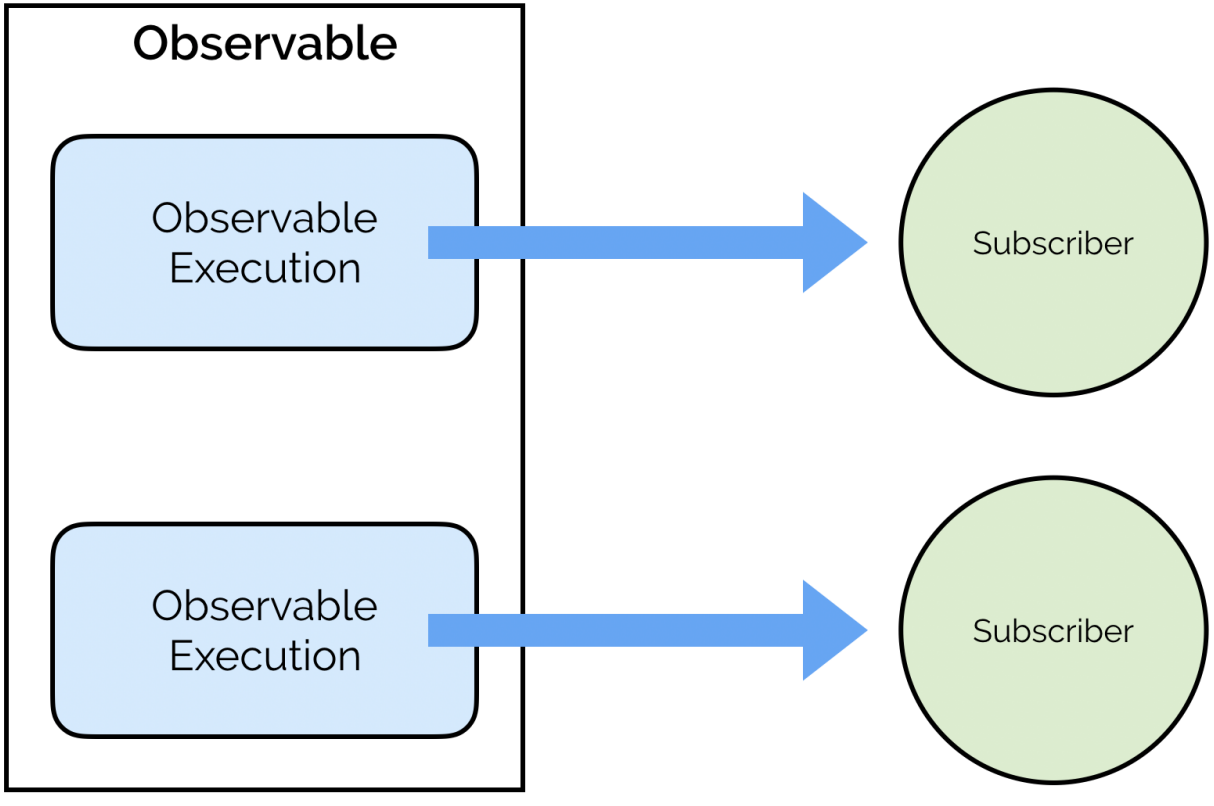
e.g. HttpClient

Multicast

Observable emits values from a Subject: which may have many subscribers

e.g. **EventEmitter** on @Output

Unicast



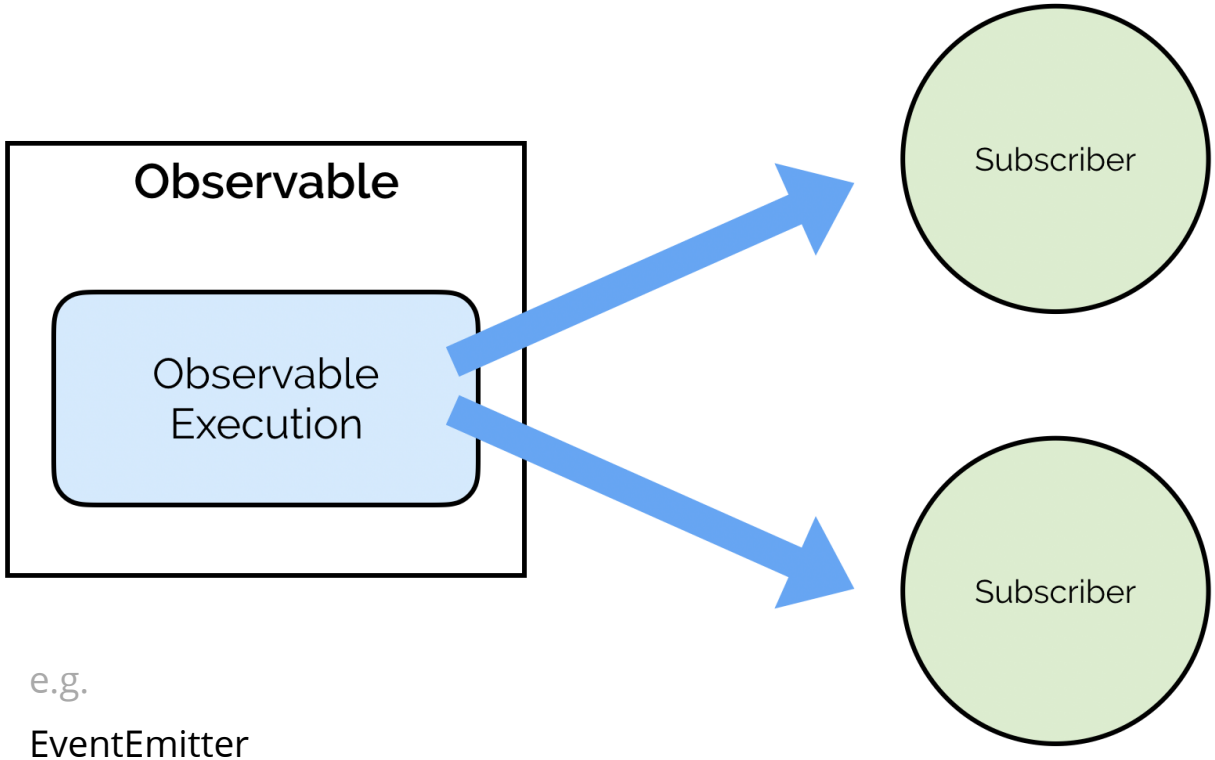
e.g. HttpClient

Speaker notes

Each subscribe runs a new observable execution
HttpClient methods do this.

Now we see why an HttpService could be called multiple times...

Multicast



e.g.

EventEmitter

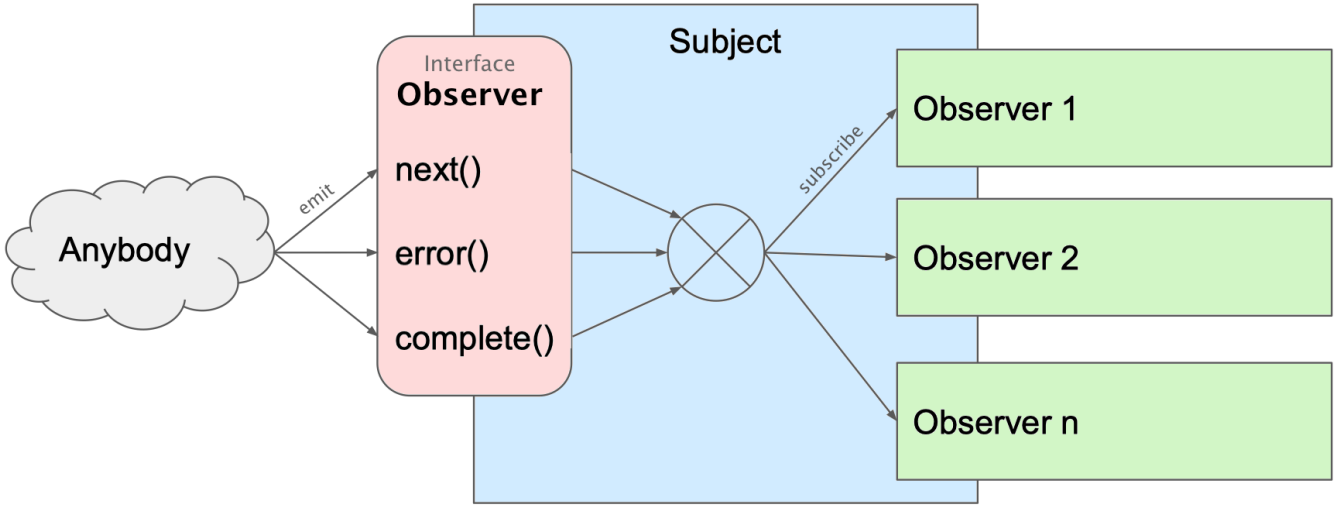
FormControl.valueChanges

Speaker notes

multicast share an Observable Execution amongst subscribers

Like EventEmitters they retain a registry of listeners
FormControl.valueChanges

Multicast: Subject



- Subjects are **hybrids**
- **Subject implements Observable** (consumers can subscribe)
- **Subject implements Observer** (producers can emit)

Multicast: Subject & BehaviorSubject

4

Subject starts with **no** values...

BehaviorSubject starts with **initial value**...

uses **last-emitted value** for new subscribers...

Multicast: Subject


```
@Component({
  ...
})
export class TicketList {

  @Output selectedItem = new EventEmitter<boolean>();

  onSelectItem(checked = true) {
    this.selectedItem.emit(checked);
  }

}
```

```
export class EventEmitter<T> extends Subject<T> {
  ...
  emit(value?: T) { super.next(value); }
  ...
}
```

Using Subject for Subscription cleanup...

4

```
destroyed$ = new Subject<boolean>();

constructor(service: StateService) {

  service.items().pipe(
    takeUntil(this.destroyed$)
  ).subscribe(console.log);

}

ngOnDestroy() {
  this.destroyed$.next(true);
  this.destroyed$.complete();
}
```

Be sure to use the rxjs-no-unsafe-takeuntil tsLint rule.

Speaker notes

Subject to be able to produce next and complete
takeUntil will end the subscription when the destroyed$ emits a value OR completes

Better to use the ***cleanup*** operator

4

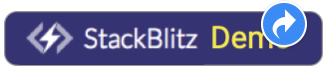
```
constructor(service: StateService) {
  service.items().pipe(

    untilViewDestroyed(this);

  ).subscribe(console.log);
}
```

Be sure to use the [rxjs-no-unsafe-takeuntil] tsLint rule.

Multicast: BehaviorSubject

4

```
import { BehaviorSubject } from 'rxjs';

const source$ = new BehaviorSubject(10);

source$.subscribe(v => console.log(`Observer #1: ${v}`);
source$.next(20);
source$.next(30);

source$.subscribe(v => console.log(`Observer #2: ${v}`);
source$.next(40);
```

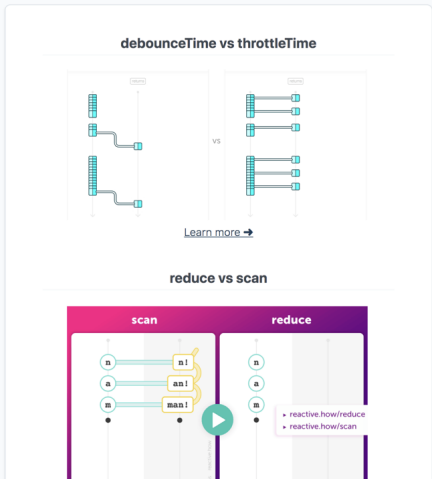


Speaker notes

Observer 1 is going to log 10 as soon as it subscribes, then 20 and 30 and 40
Observer 2 on subscribe will log 30 then 40

Resources

- RxJS Overview
  http://reactivex.io/rxjs/manual/overview.html
- RxJS 5 Book
  https://chrisnoring.gitbooks.io/rxjs-5-ultimate/content/
- RxJS Don't Unsubscribe
  https://medium.com/@benlesh/rxjs-dont-unsubscribe-6753ed4fda87
- The Observer Pattern
  https://en.wikipedia.org/wiki/Observer_pattern



Resources

- Reactive How
  http://reactive.how/
- Interactive Diagrams of Rx Observables
  http://rxmarbles.com
- Learn RxJS
  http://www.learnrxjs.io